Linköping Studies in Science and Technology
Dissertation No. 2205

Pattern-based Programming Abstractions for Heterogeneous Parallel Computing

August Ernstsson

```
int add(int a, int b)
{
    return a + b;
}

auto vsum = Map(add);

vsum(res, v1, v2);
```

LINKÖPING UNIVERSITY

Linköping Studies in Science and Technology
Dissertations, No. 2205

Pattern-based Programming Abstractions for Heterogeneous Parallel Computing

August Ernstsson

Linköping University
Department of Computer and Information Science
Software and Systems
SE-581 83 Linköping, Sweden

Linköping 2022

Edition 1:1

© August Ernstsson, 2022
ISBN 978-91-7929-195-2 (print)
ISBN 978-91-7929-196-9 (electronic)
ISSN 0345-7524

DOI `https://doi.org/10.3384/9789179291969`

Typeset using XƎTEX

Printed by LiU-Tryck, Linköping 2022

*"Simple things should be simple,
complex things should be possible."*

ALAN KAY

POPULÄRVETENSKAPLIG SAMMANFATTNING

Sanden på en sandstrand består huvudsakligen av kiselatomer. Den centrala beräkningskretsen i en dator – kallad *processor* – är också uppbyggd av kisel, som har utmärkta halvledaregenskaper. En processor är minutiöst organiserad i otroligt små transistorer – det får plats lika många transistorer i en processor stor som en tumnagel som det finns sandkorn på en sandstrand. Samhällets utveckling är idag på många sätt beroende av datorkraft, och den ständiga utvecklingen av snabbare och mer energieffektiva processorer är en förutsättning för mycket av det vi idag tar för givet: allt från väderprognoser framtagna dagligen i superdatorer stora som bostadshus till apparna i de mobiltelefoner som vi bär med oss i fickan. Under datorteknikens tidiga utveckling gjordes snabba framsteg inom miniatyrisering av kretstillverkning, vilket resulterade i en mycket snabb förbättring av datorernas beräkningskapacitet. Programmerarna behövde inte göra någonting för att hänga med på tåget. Sedan en tid tillbaka är situationen däremot inte lika enkel.

En dator följer alltid ett *program*, en lång sekvens av korta instruktioner skriven av en programmerare, när den genomför beräkningar. Programmet kan liknas vid de recept vi följer vid matlagning, där komplexa rätter tillagas efter sekvenser av enkla instruktioner. I takt med att vi blir snabbare på att följa instruktionerna blir uppgiften klar på kortare tid. Precis som vid matlagning möter datorteknikens utveckling till slut hinder i form av fysikens lagar: transistorerna blir mycket svåra att krympa när storleken börjar närma sig skalan av enstaka atomer och elektroniska signaler kan inte färdas hur snabbt som helst. Om vi vill skala upp produktiviteten i vår matlagning måste vi till slut gå mot att organisera ett industrikök, bestående av flera avdelningar med olika kompetens och sina egna speciella instruktioner att följa. Utvecklingen av datorteknik har gått i samma riktning. En modern dator består idag av en uppsättning processorenheter där varje enhet följer sitt eget instruktionsflöde. För att få fram vettiga resultat måste enheterna koordineras genom kommunikation och synkronisering, en utmaning som tillkommer programmerarens tidigare uppgift att endast skriva själva beräkningsinstruktionerna.

Egenskapen att en dator har flera processorenheter kallas *parallellism*; om enheterna skiljer sig åt i typ eller kapacitet används termen *heterogenitet*. Uppgiften att programmera sådana datorer är således betecknad parallellprogrammering. Att organisera beräkningarna i en parallell och heterogen dator kan vara utmanande – att effektivt nyttja alla beräkningsresurser manuellt kräver expertkunskaper. Därför bedrivs forskning inom utformning och konstruktion av hjälpmedel som gör att komplexiteten hos moderna datorer kan *abstraheras* utan att prestandaförlusterna blir för stora.

Den forskning som ligger till grund för denna doktorsavhandling angriper problemet genom att undersöka abstraktioner i form av universella *beräkningsmönster*. Forskningen visar hur programmeringsgränssnitt och ramverk kan utformas för att skapa datorprogram som kan anpassa sig till att använda alla beräkningsenheter i olika typer av datorsystem. Flertalet av avhandlingens bidrag är direkta resultat av internationella samarbeten mellan experter på konkreta *tillämpningar* respektive programmeringsmiljöer. Forskningens konkreta resultat inkluderar bland annat möjligheten att automatiskt använda beräkningsresurser i storskaliga klusterdatorer, metoder för erfarna programmerare att integrera egna handoptimerade komponenter i mönsterbaserade program samt en deterministisk parallell slumptalsgenerator. Forskningen lägger stor vikt vid betydelsen av programmerbarhet för att göra parallella och heterogena beräkningar tillgängliga för programmerare utan expertkunskaper. Samtidigt beaktas betydelsen av prestanda och integrationsmöjligheter hos existerande målgrupper för högprestandaprogrammering.

ABSTRACT

Contemporary computer architectures utilize wide multi-core processors, accelerators such as GPUs, and clustering of individual computers into complex large-scale systems. These hardware trends are prevalent across computers of all sizes, from the largest supercomputers down to the smallest mobile phones. While these innovations provide high peak computing performance, software developers find it increasingly difficult to effectively target all the processing resources without expert knowledge in parallelization, heterogeneous computing, communication, synchronization, and so on. To ensure that software can keep up with the development of hardware architectures, advanced high-level programming environments and frameworks are needed to bridge the programmability gap. In addition, as the industry is trending towards increased vertical integration of software development stacks, vendor lock-in presents a risk of coupling software projects to proprietary technologies. Combined with problems of technical debt in large-scale software systems, it is clear that portability and open source are desirable properties of high-level parallel programming environments. One example of a programming framework fulfilling the above criteria is SkePU, a framework for high-level data-parallel pattern programming consisting of a compiler toolchain, programming interface, and run-time system.

The work presented in this thesis proposes a design of the pattern-centric skeleton programming model of the SkePU framework based on modern C++ with variadic template metaprogramming and state-of-the-art compiler technology. The design enables further flexibility, expressivity, and portability and gives rise to several new performance optimization techniques. The focus lies on a strong set of programming abstractions: providing new and extended patterns, improving the data access locality of existing ones, and using both static and dynamic knowledge about program flow. The work combines novel programming interfaces and implementations with practical evaluation on synthetic and real-world applications. Several contributions are results from international collaborations in application-framework co-design: a single-source parallelization approach of skeleton programs on heterogeneous clusters, an extension mechanism for inserting platform-optimized code variants in high-level skeleton programs, and an integrated abstraction for portable parallel deterministic random number generation. The work places a strong emphasis on programmability aspects to make heterogeneous parallel computing accessible to non-experts, while also providing sufficient performance and interface familiarity for the high-performance computing community.

Acknowledgments

First and foremost I want to thank my supervisor at Linköping University (LiU), Professor *Christoph Kessler*, for continued and tireless efforts, experienced supervision, and caring friendship. Also thanks to my secondary supervisor *José Daniel García Sánchez*, professor at University Carlos III of Madrid and member of the ISO C++ Standardization Committee, for being available when we need consultation and for valuable insight into the C++ development process.

Thanks also to my colleagues at or around PELAB, my home at LiU for the past five years and counting. Particular thanks go to *Kristian Sandahl* for being a highly appreciated lab leader, to *Ola Leifler* not only as a colleague at PELAB but also for his and other contributors' work on LaTeX thesis templates, and to *Anne Moe* for administration and help throughout my graduate program. Extra special thanks go to *Johan Ahlqvist* for his enthusiasm in contributing to the work presented in this book and other engaging conversations during his time as part of our small group.

An important acknowledgement goes to everyone who has made direct and lasting contributions to the SkePU framework or conducted research on SkePU since its inception in 2010. A selection of names are listed in chronological order of earliest contribution: *Johan Enmyren*, *Usman Dastgeer*, *Lu Li*, *Oskar Sjöström*, *Tomas Öhberg*, *Henrik Henriksson*, *Johan Ahlqvist*, *Basel Nsralla*, *Joel Almqvist*, and *Erik Tedhamre*.

Thanks also to everyone not mentioned by name who contributed to SkePU indirectly, including but not limited to all project partners in EXA2PRO for involvement that led to the design of SkePU 3, as well as students at Linköping University who provided feedback on SkePU over the years.

I also thank the *Swedish National Supercomputing Centre* (NSC) and SNIC for access to their HPC computing resources through two generations of clusters, Triolith and Tetralith.

Finally, my thanks goes to my family, my friends, and colleagues at LiU and beyond for encouragement and support.

Work presented in this thesis has been partly funded by EU FP7 project *EXCESS* (611183) and EU H2020 project *EXA2PRO* (801015), by the *Swedish National Graduate School in Computer Science* (CUGS), and by SeRC.

August Ernstsson
Linköping, January 2022

Contents

1 Introduction

Contemporary computer architectures are increasingly parallel designs with multiple processor cores. In addition, massively parallel accelerators, such as GPUs, make these systems heterogeneous architectures. This development is a consequence of the power and frequency limitations for single, sequential processors. Parallel architectures help overcome this barrier and maintain Moore's law-like growth of computing power [155]. For programmers and programming languages designed for sequential and homogeneous systems, it is a challenge to utilize the resources available in modern computer systems in an efficient manner. The challenges are many: communication, synchronization, load distribution, and so on. This is especially true if also *performance portability* is desired, as different systems can vary widely in terms of both the number and types of processing cores, as well as in other characteristics such as memory hierarchy.

1.1 Aims and research questions

This thesis aims to introduce the modern approach to high-level parallel programming taken by SkePU version 2 and later. SkePU implements its own interpretation of the skeleton programming concept, which is a widely researched programming model using patterns and parametrizable higher-order functions as programming constructs. Throughout the thesis, the skeleton programming approach is explored, with emphasis on recent research and the current landscape of available skeleton programming frameworks. The thesis aims to give a good overview of SkePU syntax and features, but is not intended to be an exhaustive documentation of the framework. Rather, the approach is to provide insight into the thoughts and design considerations of the contributions that has been made to SkePU over the past few years. During this time, SkePU has seen significant change, both in terms of interface adaption and modernization as well as extensions in feature set and target hardware platforms.

The work on SkePU 2 and SkePU 3 attempts to address the following:

RQ1 How can a contemporary skeleton programming interface utilize *modern C++ capabilities* such as variadic templates and lambda expressions?

RQ2 Can flexibility and type-safety be improved by providing a custom *source-to-source compiler* instead of C-style macros, for backend code generation?

RQ3 How can SkePU be improved for real-world applications, e.g., for scientific computing, by applying *application-framework co-design*?

The thesis goes into detail on six specific contributions, providing answers to the following research questions:

RQ4 How can *lazy evaluation* be utilized in SkePU programs composed of sequences of skeleton operations on the same data set, and specifically, is inter-skeleton *tiling* an optimization technique that can be applied in this scenario?

RQ5 Can CPU-GPU *hybrid execution* of skeletons be implemented as a backend target through the variadic SkePU interface? What is the optimal split ratio of work between CPU and GPU backends, and what is the possible performance gain?

RQ6 Large-scale computations require more resources than what is available in a single computer. What are the possible performance gains of extending the SkePU programming model to target *cluster* execution, and what, if any, are the tradeoffs in regards to capabilities and programmability in such an extension?

RQ7 How can SkePU be utilized or provide benefit for applications which are not a perfect fit for automatic generation of backend-specific code? Should there be a way for *expert programmers* to override backend code generation in cases for which this is desirable?

RQ8 A multi-backend programming framework is expected to behave similarly or, ideally, identical across target systems. What are the limitations of *pseudo-random number generation* (PRNG) in terms of reproducibility across backends, and how can a skeleton programming framework accommodate a *deterministic* PRNG in its interface and implementation?

RQ9 How can the known optimization technique *kernel fusion* be applied in skeleton programming? In particular, to what extent does SkePU allow for manual *skeleton fusion*, and in which ways can automated fusion combine *static* (compile-time) and *dynamic* (run-time) program flow information?

1.2 Published work behind this thesis

This thesis is based on the work presented in eight papers, of which seven are published in peer-reviewed journals or proceedings and one is, as of writing, accepted and awaiting publication.

1. **SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems** [62]
   August Ernstsson, Lu Li, and Christoph Kessler
   This paper was first presented at the **HLPP 2016** symposium in Münster, Germany on July 4, 2016. The journal paper was published in *International Journal of Parallel Programming* in 2017 as **Open Access** under *CC BY 4.0*. Initial prototype and design work of SkePU 2 was carried out as part of August Ernstsson's master's thesis [58]. The same work was also disseminated at the **EXCESS project workshop** in Gothenburg, Sweden on August 26, 2016 and at **MCC 2016** workshop on November 29, 2016. A poster on SkePU 2 based on the contributions in this paper was represented at **HiPEAC 2017** in Stockholm, Sweden.

2. **Extending smart containers for data locality-aware skeleton programming** [60]
   August Ernstsson and Christoph Kessler
   This paper was first presented at **HLPP 2017** in Valladolid, Spain on July 11, 2017. The paper was published in *Concurrency and Computation Practice and Experience* in 2019. The same contribution was also presented at **MCC 2017** by means of a poster and short presentation.

3. **Hybrid CPU–GPU execution support in the skeleton programming framework SkePU** [119]
   Tomas Öhberg, August Ernstsson, and Christoph Kessler
   This paper was first presented at **HLPP 2018**. This journal paper was published in *The Journal of Supercomputing* in 2019 as **Open Access** under *CC BY 4.0*. The contributions in this paper are results of the master's thesis project by Tomas Öhberg [118], supervised and guided by August.

4. **Multi-Variant User Functions for Platform-Aware Skeleton Programming** [61]
   August Ernstsson and Christoph Kessler
   This paper was first presented at **ParCo'19** in Prague, Czech Republic on September 10, 2019. The journal paper was published in *Advances in Parallel Computing* in 2020 as **Open Access** under *CC BY-NC 4.0*. A preview of this contribution was represented at **HLPP 2019** with a poster exhibition and short presentation. The work was also disseminated at the **MCC 2019** workshop in Karlskrona, Sweden on November 27, 2019.

5. **Portable exploitation of parallel and heterogeneous HPC architectures in neural simulation using SkePU** [121]
   Sotirios Panagiotou, August Ernstsson, Johan Ahlqvist, Lazaros Papadopoulos, Christoph Kessler, and Dimitrios Soudris
   This conference paper was presented at **SCOPES'20** in 2020 and published in the proceedings the same year. The paper is the first published result of collaborations within the EU project *EXA2PRO*, and provides results from applying SkePU in a real-world application context.

6. **SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters** [59]
   August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler
   This paper was presented at **HLPP 2020** on July 9, 2020. A revised version was published in *International Journal of Parallel Programming* in May 2021 as **Open Access** under *CC BY 4.0*. Also a direct result of *EXA2PRO* collaborations, the paper introduces SkePU 3, including its new cluster backend.

7. **EXA2PRO: A Framework for High Development Productivity on Heterogeneous Computing Systems** [122]
   Lazaros Papadopoulos, Dimitrios Soudris, Christoph Kessler, August Ernstsson, Johan Ahlqvist, Nikos Vasilas, Athanasios I. Papadopoulos, Panos Seferlis, Charles Prouveur, Matthieu Haefele, Samuel Thibault, Athanasios Salamanis, Theodoros Ioakimidis, Dionysios Kehagias
   This paper summarizes the work and results of *EXA2PRO*. It will be published in *IEEE Transactions on Parallel and Distributed Systems* in April 2022 as **Open Access** under *CC BY 4.0*.

8. **A Deterministic Portable Parallel Pseudo-Random Number Generator for Pattern-Based Programming of Heterogeneous Parallel Systems** [63]
   August Ernstsson, Nicolas Vandenbergen, Jörg Keller, Christoph Kessler
   This joint work within the *EXA2PRO* project resulted in a paper presented at **HLPP 2021** on July 12, 2021. A revised version has been ac-

cepted for publication in a future issue of *International Journal of Parallel Programming*.

Papers 2, 3, 4, and 8 are reproduced in this thesis in large part, presented as main contributions. Paper 1 and 6 have the introduction of SkePU version 2 and SkePU version 3, respectively, as their main contributions, and therefore material from these papers is extended and reworked into the chapters of this thesis that present the history, interface, and implementation of SkePU (Chapters 3, 4, 5, and 7). In addition, experimental results and evaluation from all papers is collected and reproduced in Chapter 15.

Publications in the form of project deliverables from the EXA2PRO project also form part of the basis for this thesis. Most of the deliverables are publicly available in CORDIS[1] except for a few with confidential classification. Of the published deliverables, those with August Ernstsson listed among the authors [26–38] are of relevance to the work presented in this thesis.

1.3 Other work behind this thesis

In addition to the published material, this thesis is shaped by the experience gained from the exposure of the SkePU framework to potential users by the means of numerous tutorials, given, e.g., in conjunction with **HLPP 2019**, **MCC 2019**, **PPoPP 2021** and **eScience 2021** and in teaching through the course *TDDD56: Multicore and GPU programming*.

Design and implementation work for several contributions has been conducted through master's thesis projects, with August Ernstsson acting as supervisor. The thesis project by *Tomas Öhberg* [118] eventually resulted in a publication [119] and is the basis of Chapter 8; the projects by *Joel Almqvist* [3] and *Basel Nsralla* [116] are not adapted into peer-reviewed publications at the time of writing, but are nonetheless represented as important contributions to Chapters 9 and 14, respectively. Evaluation results of these master's thesis projects, including a project by *Erik Tedhamre*[2] are also summarized in Chapter 15.

1.4 Structure

This thesis roughly follows a three-part structure. The initial chapters (1–7) set the context through background and descriptions of programming frameworks, most notably SkePU. Then follows a middle part (Chapters 8–15), the main contributions of the work followed by evaluation

[1]Community Research and Development Information Service, project 801015, `https://cordis.europa.eu/project/id/801015/results`

[2]Erik's project is in the final stages of progress, but there is not yet a report to cite.

results. The final part contains discussion, introspection, and summarizes the key insights.

Chapter 2 presents **background** surrounding the skeleton programming paradigm for high-level parallel programming. Various applications of the skeleton programming model from the scientific community and the industry are also surveyed in this chapter, as **related work**.

Chapter 3 provides an initial concise overview of the SkePU framework, the main topic of the thesis. The deep dive into SkePU then begins with Chapter 4, which explores the **skeleton set** of SkePU. Chapter 5 contains a study of SkePU's data representation abstraction, **smart data-containers**. The interface overview of SkePU is completed with Chapter 6 covering the **standard library**. Once the outwards-facing aspects of SkePU are well introduced, Chapter 7 explains the **implementation** of SkePU with its header library and compiler toolchain.

The subsequent seven chapters presents the *main contributions*. First, we have two backend-related contributions: Chapter 8 presents the **hybrid backend** and Chapter 9 details the two **cluster backends**.

Chapter 10 covers the work on a **data-locality optimization** through lazy evaluation, continuing into Chapter 11 discussing the related topic of **skeleton fusion**.

Chapter 12 details **multi-variant user functions** and Chapter 13 covers a library contribution in depth: the design of a **deterministic parallel random number generator**. Progress towards a modernized **variadic autotuning** system is documented in Chapter 14.

These contribution chapters omits evaluation results, which are collected in Chapter 15 together with other published and unpublished **results** including performance evaluation.

Chapter 16 discusses **limitations** of the approaches explored in the thesis and presents ideas for **future work**. Chapter 17 **concludes** the thesis.

This book is a *doctoral* dissertation, and follows up on a *licentiate* thesis published and defended in late 2020. Additions and changes from this previous book are documented in Appendix A.

2 Background and related work

This chapter starts with an extended problem motivation (Section 2.1), giving a context for the work presented later in the thesis. A series of important concepts are then introduced, starting with high-level parallel programming in Section 2.2 and the specific approach of *skeleton programming* in Section 2.3.

The later sections cover related work of various types. Parts of the related work discussion assumes some familiarity with the SkePU framework; readers who feel a little lost are encouraged to come back after reading Chapters 3–5.

2.1 Motivation

The motivation behind the need for parallel computing, provided in Chapter 1, answers the question of *why* there is a need for (high-level) parallel programming systems to utilize the available computational resources. In this chapter, we focus on the software systems attempting to answer the *how*. On the hardware side, we have access to processing units of ever increasing width, be it traditional CPU-style cores or accelerator devices, and these units are assembled in larger and larger clusters. As of the time of writing, the leading supercomputer cluster in the world (*Fugaku* at RIKEN)

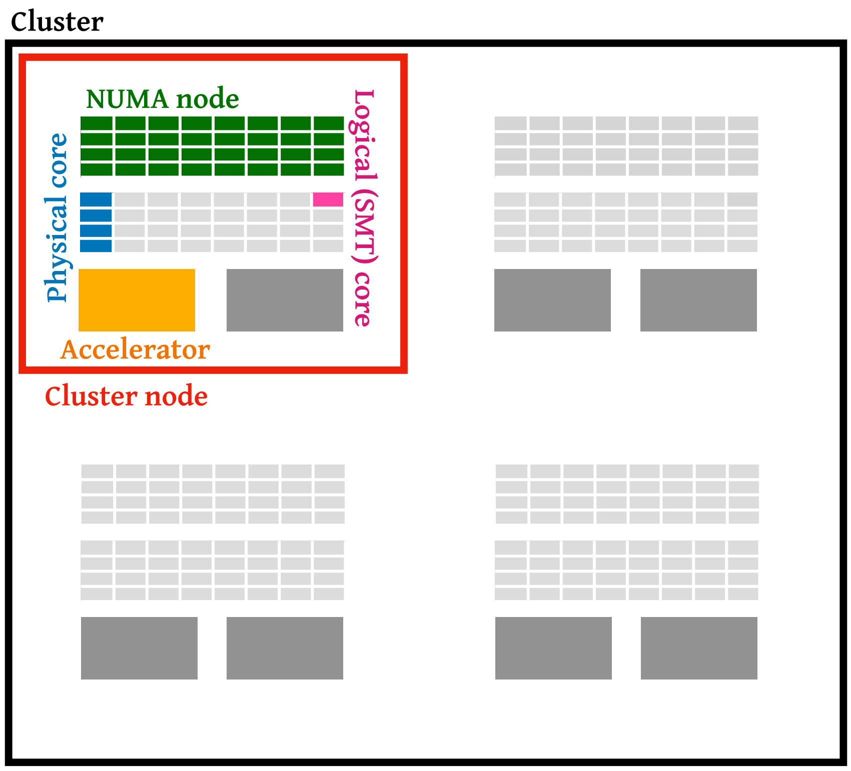


Figure 2.1: Schematic for the compute units of a typical HPC cluster architecture.

has millions of cores[1] and total parallel performance of almost half an exaflop. Figure 2.1 contains a schematic over the various computational resources provided by a modern cluster node. We can see *parallelism* and *heterogeneity* in multiple levels[2].

Our goal is to achieve *portability* of programs, such that they can run without error on many types of systems and architectures. In this work, we primarily investigate high-level *abstractions* to reach this goal, starting with the next section. However, portability can also be reached by *specialization*;, the choice here can have an important impact on the properties of the resulting codebase. Which is better is dependent on the context, but generally, increased abstraction will result in greater *programmer productivity*

[1] A twice-yearly updated list of the most powerful supercomputers in the world is maintained by Top500.org. At the time of writing, the latest version was available at `https://www.top500.org/lists/top500/2021/11/`.

[2] Supercomputers tend to be mostly homogeneous in their node-level structure, but it is not uncommon to find a relatively small amount of specialized nodes, for instance with additional memory or unique accelerators.

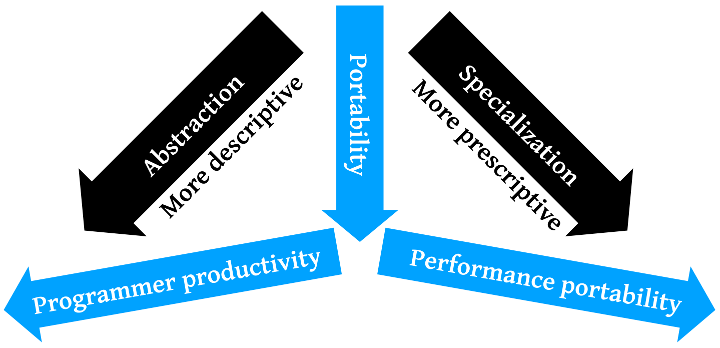


Figure 2.2: Impact from portability through abstraction and specialization.

while specialization is important to maximize *performance portability* [111, 125] (Figure 2.2).

2.2 High-level parallel programming

Programming of parallel hardware is inherently more challenging for the user than traditional sequential programming, especially when the parallel system is heterogeneous, and parallel computing systems need to accommodate this fact in the systems and interfaces presented to the programmer. As expressed by Cole [25], finding the right *abstraction level* is the key to balance the equation, and this is an ever-moving target—as hardware capabilities increase, the penalties imposed by additional levels of abstraction become more forgiving. As it happens, there seems to be some scientific consensus, judging by the breadth of work published on the subject as presented in this chapter, that the time has come for *algorithmic skeletons* (throughout this thesis mostly referred to by the term *skeleton programming*) to be a viable *high-level* abstraction for programming of parallel hardware.

High-level parallel programming frameworks aim to reduce the user-facing complexity of parallel programs while still achieving as good resource utilization as possible. A small number of highly optimized, but still general, programming building blocks are presented through a high-level interface. This category of frameworks include application specific languages, PGAS (Partitioned Global Address Space) interfaces, dataflow models, and more, but most importantly for this thesis: the *skeleton programming* concept.

Two recent literature studies investigate the state of parallel programming languages: Amaral et al. [4] focus on the HPC domain, and Ciccozzi et al. [21] take an even broader look at trends and challenges in parallel programming languages overall.

2.3 Skeleton programming

Skeleton programming [25, 74] is a programming model for parallel systems inspired by *functional programming* (See also Hammarlund et al. [79] for a formal presentation of the relationships between functional and data-parallel programming.) The central abstraction of the concept are the *skeletons* which are inherently parallelizable computational patterns. These patterns are known from functional programming as *higher-order functions*: functions accepting other functions as parameters. Common examples include *map* and *reduce*. The supplied function is applied to a structured set of data according to the semantics of the particular skeleton. Typically, the function is assumed to have no side effects and the computation can thus be reordered and parallelized.

Compositions of skeletons can model entire programs, which are sequential in interface but with parallelizable semantics. Aspects such as communication and synchronization are nowhere to be (explicitly) seen, and even particulars about *how* and *where* computation is performed in the underlying system is decided by the system itself, not the programmer. In other terms: skeleton programming tends to be more on the *declarative* side, at least pertaining to overarching computational structures in a program.

Rabhi and Gorlatch [131] compare patterns in the sense of algorithmic skeletons to *design patterns* from software engineering. They note that while there are similarities and even direct analogues between the two, skeleton patterns are formal constructs used for performance-related reasons, while design patterns are loosely defined and applied, e.g., for reliability. In this thesis, the term *pattern* strictly refers to algorithmic skeletons. Danelutto et al. [44] more recently surveyed and compared the algorithmic skeleton research community to that of parallel design patterns.

Several parallel programming frameworks implement the algorithmic skeleton model [55, 56, 106, 146], some of them for multiple different parallel architectures (*backends*) with a single common interface. Selection of backends can be done with auto-tuning [46]. Examples of skeleton patterns are often divided into two categories: *data parallel* patterns such as the aforementioned map and reduce, and *task parallel* patterns including task farming and parallel divide-and-conquer, among others. Particularities of how the skeleton programming model is adapted in the actual frameworks can differ significantly, visible for instance in the available skeleton set (and even the naming of skeleton patterns), backend set, and naturally also the general program syntax, among others.

2.4 Related work

The skeleton approach to high-level programming of parallel systems was introduced by Cole in 1989 [24, 25]. Since then, many academic skeleton programming frameworks have been presented, and the concept also increasingly found its way into commercial and industrial-strength programming environments, such as Intel *TBB* for multi-core CPU parallelism, Nvidia *Thrust* or Khronos *SYCL* for GPU parallelism, or Google *MapReduce* and Apache *Spark* for cluster-level parallelism over huge data sets in distributed files.

While early skeleton programming environments attempted to define and implement their own programming language, library-based and DSL-based approaches have, by and large, been more successful, due to fewer dependencies and lower implementation effort. Frameworks for skeleton programming became practically most effective in combination with (modern) C++ as base language. Moreover, the approach was fueled by the increasing diversity of processing hardware with upcoming multi-core and heterogeneous parallelism since the early 21st century.

This section first surveys three pattern-based frameworks in more detail: *GrPPI* in Section 2.4.1, *Musket* in Section 2.4.2, and *Kokkos* in Section 2.4.3. All are relatively recent contributions from the academic community and actively maintained and published, and they provide both similarities and differences when compared to SkePU. Some attention is also given to industry-led high-level parallel programming, which are led either by individual corporations or through consortia and standardization committees. *SYCL* is an important standardization initiative and is given extra attention in Section 2.4.4. We then look at two programming frameworks which are not related to data-parallel patterns, but nonetheless relevant to the topics brought up in this thesis: *MLIR* in Section 2.4.5 and *StarPU* in Section 2.4.6. Section 2.4.7 then explores industry efforts. These are important especially as their wide availability makes them targets or dependencies of academic work. The remainder of the related work section is spent on just that: the wide variety of large and small contributions of academic research, most of which come with implementations and programming systems of their own.

2.4.1 GrPPI

GrPPI [135] is a relatively recent interface for generic parallel patterns. Like SkePU, it takes full advantage of modern C++ and is designed as an interface abstracting from and selecting among lower-level frameworks: C++ threads, OpenMP, Intel TBB, and Thrust [13].

The patterns offered by GrPPI (it does not use the term skeletons, but it is a matter of terminology choice) are split into *stream parallel* and *data parallel* groups.

As SkePU does not feature stream parallelism, this is a good opportunity to discuss common stream parallel patterns. In GrPPI, these are:

- **Pipeline**: Pipeline parallelization is in essence the opposite of data parallelism: parallelization is gained not from the *width* of the data set, but from the *depth* of the computation sequence. A pipeline consists of a chain of function calls (which can, but are not required to be, data parallel patterns in themselves). Each function is evaluated in parallel with the others, but due to the dependency chain, each function call are on a different *packet* from the data stream. The pipeline eventually fills up and reaches a steady state where all pipeline stages have independent data to work on.
- **Farm**: Much like a stream *map*, farm computes a transformation of the incoming packets and places the results in the output stream. Each function invocation is ”farmed” out to a set of parallel workers.
- **Filter**: The filter pattern takes as input a stream and returns a stream where packets may be filtered out by a predicate (boolean) function. Parallelization is extracted by computing several stream packets at once, with the requirement that the invocations of the filtering function are independent.
- **Accumulator**: Much like a stream version of *reduce*, the accumulator pattern combines packets from the source stream using an associative and commutative binary combination operator. The output stream is partial ”sums” of the packets in the source stream, with the number of elements dependent on a set window size.

The set of data parallel patterns is as follows:

- **Map**: Map is conceptually the same pattern as the SkePU `Map` presented later in this thesis. As with all pattern libraries, the full capabilities, interface, and implementation can differ significantly.
- **Reduce**: A finite data set is accumulated into a single value by an associative and commutative binary combination operator, like the SkePU `Reduce`.
- **Stencil**: Stencil is the GrPPI name for the same pattern as represented in SkePU by `MapOverlap`.
- **MapReduce**: Unlike the prior data parallel skeletons, GrPPI MapReduce differs in interpretation from the one in SkePU. GrPPI MapReduce is more closely aligned with the big data analytics framework style MapReduce, where the mapping function not only computes a transformation of its argument, but also assigns a *key* and returns a tuple of the processed result and the key. In the reduction phase, a computation following the process in Reduce is performed on each subsequence of tuples with the same key.

- **Divide & Conquer**: Divide and conquer is an established parallel pattern which is missing from SkePU but available in GrPPI. The input data set is recursively broken down into smaller subsequences until a base case is reached. The pattern is parametrizable with splitting, merging, and base case functions.

Listing 2.1 shows a sorting computation using the GrPPI interface.

2.4.2 Musket

Musket [134] approaches the high-level parallel interface not by integrating into an existing language like C++, but rather with a domain-specific language and custom compiler toolchain. Unlike SkePU, the Musket compiler is provided as a plugin to the *Eclipse* integrated development environment, allowing model validation and the resulting errors and warnings to be controlled from a graphical user interface. Musket can target GPUs, but unlike SkePU has investigated OpenACC as the backend interface in addition to CUDA [157].

Musket uses generally the same terminology as SkePU, with *skeletons* parameterized with *user functions*. The skeleton set differs quite a bit, with the fundamental skeleton types in Musket being `map`, `fold`, `mapFold`, `zip`, and two different `shift partition` skeletons. Operand data distribution and collective data communication in cluster execution is exposed to the programmer in Musket, whereas it is abstracted away in SkePU.

Map and fold correspond to the SkePU constructs `Map<1>` and `Reduce`, respectively, and as in SkePU, an explicit fusion of the two is provided in mapFold. The zip skeleton is a way to merge two data structures elementwise, and as such acts like a map with input arity 2, `Map<2>` in SkePU.

The basic skeletons may have variants, such as map having the *mapInPlace* when the input data structure is the same as the output, *mapIndex* and *mapLocalIndex* which can access the index within the processed data set. While similar features exist in SkePU, the approach of expressing them are different. The fact that both fundamental skeleton patterns and auxiliary features are shared between Musket and SkePU under different terminology and syntactical means (and this fact is merely illustrated with the two, and not limited to just the SkePU-Musket comparison) can make it challenging for programmers to go from one to the other, and it also presents a challenge for attempting an approachable categorization and comparison of different skeleton programming implementations.

A sample application using the Musket DSL is provided in Listing 2.2, illustrating the fact that its syntax is strongly evocative of C++ conventions, but a Musket program is not a valid C++ program.

Recent work on Musket has extended the framework to target multi-GPU cluster platforms [157].

Listing 2.1: Excerpt from sample code from the GrPPI repository: Sorting a sequence of integers using Divide & Conquer.

```
/*
 * Copyright 2018 Universidad Carlos III de Madrid
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#include "grppi/grppi.h"

std::vector<range> divide(range r) {
  auto mid = r.first + distance(r.first,r.last)/2;
  return { {r.first,mid} , {mid, r.last} };
}

void sort_sequence(grppi::dynamic_execution & exec, int n) {
  using namespace std;

  std::random_device rdev;
  std::uniform_int_distribution<> gen{1,1000};

  vector<int> v;
  for (int i=0; i<n; ++i) {
    v.push_back(gen(rdev));
  }

  range problem{begin(v), end(v)};

  grppi::divide_conquer(exec,
    problem,
    [](auto r) -> vector<range> { return divide(r); },
    [](auto r) { return 1>=r.size(); },
    [](auto x) { return x; },
    [](auto r1, auto r2) {
      std::inplace_merge(r1.first, r1.last, r2.last);
      return range{r1.first, r2.last};
    }
  );
}
```

Listing 2.2: Complete sample code from the Musket repository: computation of the Frobenius norm.

```
#config PLATFORM GPU CPU_MPMD CUDA
#config PROCESSES 1
#config GPUS 1
#config CORES 4
#config MODE release

const int dim = 8192;

matrix<double,dim,dim,dist,dist> as;

double init(int x, int y, double a){
  a = (double) (x + y + 1);
  return a;
}

double square(double a){
  a = a * a;
  return a;
}

main{
  as.mapIndexInPlace(init());
  mkt::roi_start();
  double fn = as.mapReduce(square(), plus);
  fn = mkt::sqrt(fn);
  mkt::roi_end();
  mkt::print("Frobenius norm is %.5f.\n", fn);
}
```

2.4.3 Kokkos

The *Kokkos* C++ library [18, 150] seems to be gaining traction in the scientific community, often being used for comparative evaluation purposes. Kokkos has its origins as an internal portability layer for linear algebra kernels and brings a strong focus on efficient data representation. Kokkos manages data trough multi-dimensional arrays called `Views`, which offers the programmer the flexibility of declaring array dimensions as a mix of static or dynamic sizes (Listing 2.3). Static size information is taken advantage of for automatic layout decisions and efficient element addressing at run-time. Compared to SkePU, this approach offers more flexibility by not being limited to a maximum of four dimensions, and optimization opportunities arise from the static dimensions. The programming interface for declaring multi-dimensional arrays in Kokkos requires the user to provide the dynamic dimensions first in the list, which is comparable to SkePU user functions enforcing an ordering of parameter types (element-wise first, etc.). There are significant advantages to be gained from embedding programming models in existing high-profile languages such as C++, but these two examples show ways that the host language can limit the expressivity of the framework.

Listing 2.3: Array declaration in Kokkos, with one dynamic and one static dimension size.

```
const size_t N = ...;
Kokkos::View<double*[3]> b ("debugging label", N);
```

Listing 2.4: Parallel ”hello-world” program in Kokkos, from the Kokkos programming guide.

```
#include<Kokkos_Core.hpp>
#include<cstdio>

int main(int argc, char* argv[]) {
   Kokkos::initialize(argc,argv);

   int N = atoi(argv[1]);

   Kokkos::parallel_for("Loop1", N, KOKKOS_LAMBDA (const int& i) {
     printf("Greeting from iteration %i\n",i);
   });

   Kokkos::finalize();
}
```

For parallel computation, Kokkos provides three data-parallel patterns (called *dispatch operations*): `parallel_for`, `parallel_reduce`, and `parallel_scan`. Kokkos' ”reduce” is rather a full mapreduce, and both reductions and scans require an associative operator which is user-defined. Furthermore, Kokkos contains a limited task abstraction and random number generation facilities.

Listing 2.4 shows a complete Kokkos program using the `parallel_for` construct. As of the time of writing, Kokkos 3 [150] is a recent evolution including a new abstraction for multi-dimensional parallel iteration with optional tiling and the ability for hierarchical parallelism, the latter suitable for, e.g., matrix-vector multiplication.

2.4.4 SYCL

Over the past decade, there have been standardization efforts of skeleton-like interfaces. One such instance is *SYCL* [129] from the Khronos Group[3]. The Khronos Group manages open standards, including OpenGL and OpenCL. SYCL is an attempt at bringing heterogeneous C++ programming to as many programmers as possible. While primarily designed as a higher-level abstraction layer to OpenCL or multi-threaded CPU processing, the framework is extensible for other hardware platforms. SYCL is intended

[3]https://www.khronos.org/sycl

Listing 2.5: SYCL 1.2 code sample adapted from Khronos tutorial material.

```
#include <CL/sycl.hpp>
#include <iostream>

int main()
{
  using namespace cl::sycl;

  int data[1024];

  // create a queue to enqueue work to
  queue myQueue;

  // wrap our data variable in a buffer
  buffer<int, 1> resultBuf(data, range<1>(1024));

  // create a command_group to issue commands to the queue
  myQueue.submit([&](handler& cgh)
  {
    // request access to the buffer
    auto writeResult = resultBuf.get_access<access::write>(cgh);

    // enqueue a parallel_for task
    cgh.parallel_for<class simple_test>(range<1>(1024),
      [=](id<1> idx) { writeResult[idx] = idx[0]; }
    );
  });
}
```

both as a programmer-facing interface and a backend target for domain-specific languages and tools, such as BLAS-style libraries or machine learning environments.

SYCL addresses the limitations in OpenCL by providing a single-source interface and by reducing boilerplate and state-machine operations through, for instance, high-level parallel patterns (`parallel_for`). A modern C++ foundation also ensures type safety through the use of templates and lambda expressions.

Listing 2.5 illustrates a minimal SYCL program invoking a `parallel_for` task.

While initially SYCL was primarily available in reference implementations from Codeplay, several projects have since built upon or integrated SYCL in various programming environments. Examples include Intel's *oneAPI* as discussed later and *Celerity* [149], the latter of which adopts (and slightly adapts) SYCL for cluster computations. Celerity is especially interesting when comparing SkePU to SYCL, as both originate as heterogeneous single-node APIs which gets adapted for distributed computing in a later stage. However, the comparison is not completely fair as SYCL exposes more low-level constructs than SkePU and is to a higher degree designed to be a compilation target for other programming environments.

2.4.5 MLIR

High-level languages have a few options for to realizing a programming model and interface in a concrete implementation. Often, the approach taken is a pure library-based one, which for C++ libraries can mean either a separately linked code base with interface header files, or a pure include library with only header files. The other extreme is to implement a custom language and compiler stack. The approach taken by the work in this thesis and the SkePU framework lies somewhere in between, with language extensions processed by an additional precompilation step. As compilers, or metacompilers/precompilers [16], involve a lot of work, most of which could be classified as reinventing the wheel, such tools are typically based on existing compiler infrastructure, examples which include LLVM [94], ROSE [130], and Mercurium [10, 65] for C++; and many more for other languages, including for example Java [81].

MLIR [95] is an effort to address the limitations in LLVM with regard to representation of high-level (e.g., parallelism) constructs. MLIR is primary targeted at the machine learning domain. The framework is designed to be highly modular, and is architectured around a hierarchical chain of internal representations, progressively lowering high-level constructs in a way such that information about program structure is not lost. Transformations are carried out in ways that still allows tracing back source code locations and validating correctness of the transformed structures.

While MLIR is not a high-level parallel programming interface in its own, it could prove to be the basis of future contributions in this domain, as LLVM has been in the past. An early example of a parallel pattern framework built on MLIR is *RISE* [102], described as a spiritual successor to *Lift* (covered in Section 2.4.8).

2.4.6 StarPU

StarPU[4] [8] is a C-based task programming library for hybrid architectures. The goal of StarPU is to provide a unified runtime system for heterogeneous computer systems, including different execution units and programming models. StarPU also offers a high-level C++ interface or, optionally, compiler-extension pragmas.

A task in StarPU is defined in terms of *codelets*. Describing a computational task, codelets are combined with input data to form *tasks*. Tasks are passed to the runtime system asynchronously, and later mapped and scheduled to be executed on any of the available computing resources. The codelets can contain code written in C/C++, CUDA, and OpenCL. StarPU's modular implementation ensures that different scheduling policies and performance models can be used. Examples of scheduling policies include

[4]`http://starpu.gforge.inria.fr`

eager-based, *priority-based*, and *random-based* schedulers. It is also possible to construct custom schedulers using the pre-implemented scheduling components. Similar to SkePU, StarPU performs its own data transfer optimization by caching data on the computational units where it was last accessed.

StarPU has been used in a number of application scenarios, recent examples including finite-volume CFD [39] and seismic wave modeling [108].

The latest versions of StarPU have been extended to target cluster systems [7]. StarPU integrates MPI communication with its existing accelerator-enabled codelets for a system that can utilize all the available computing resources in heterogeneous clusters.

2.4.7 C++ AMP, and other industry efforts

Intel TBB (thread building blocks)[5] is a relatively low-level parallel programming interface with explicit thread parallelism, but providing task scheduling and memory management abstractions as well as data-parallel constructs. While TBB is relatively old, it is continuously maintained and updated, for instance with modern C++ conventions such as lambda expressions. TBB is often one of several implementation targets for higher-level skeleton programming frameworks. *OpenMP* is frequently used as an alternative, being standardized and not controlled by a single actor.

The corresponding role TBB and OpenMP have on CPUs is on GPUs handled by *OpenCL* and *CUDA*. Both are GPGPU programming interfaces at a fairly low level with a lot of manual control flow and data management required from the programmer. OpenCL is defined with a C interface and an industry standard managed by the Khronos consortium, while CUDA uses more expressive C++ and is proprietary to Nvidia GPUs.

Nvidia *Thrust* [13] is a C++ template library with parallel CUDA implementations of common algorithms. It uses common C++ STL idioms, and defines operators (comparable to SkePU user functions) as native functors. The implementation is in effect similar to that of SkePU, as the CUDA compiler takes an equivalent role to the source-to-source compiler presented in this article.

The C++ ISO committee has included a parallel version of STL algorithms in C++17, which as of recently is starting to see wider adoption.

Although Microsoft's solution for C++ parallelism is separate from the standardization efforts, their *C++ AMP* (Accelerated Massive Parallelism) interface is largely similar, but with more explicit data management across devices. C++ AMP provides an `extents` mechanism for emulating higher-dimensionality data structures through arrays.

Recently Intel has collected several existing technologies together with their compiler and profiler toolchains and community language extensions

[5]`https://software.intel.com/content/www/us/en/develop/tools/threading-building-blocks.html`

Listing 2.6: Sample code from the Microsoft documentation: Vector sum with C++ AMP.

```
#include <amp.h>
#include <iostream>
using namespace concurrency;

const int size = 5;

void CppAmpMethod() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[size];

    // Create C++ AMP objects.
    array_view<const int, 1> a(size, aCPP);
    array_view<const int, 1> b(size, bCPP);
    array_view<int, 1> sum(size, sumCPP);
    sum.discard_data();

    parallel_for_each(
        // Define the compute domain, which is the set of threads created.
        sum.extent,
        // Define the code to run on each thread on the accelerator.
        [=](index<1> idx) restrict(amp) {
            sum[idx] = a[idx] + b[idx];
        }
    );
}
```

in what they call *oneAPI*[6]. Their proposed programming language is DPC++, data parallel C++, which is based on standard C++ and SYCL with compiler technology built on the Clang and LLVM stack. Intel is targeting systems using a combination of CPU, GPU, and FPGA compute units with extensibility for other specialized accelerators.

Nvidia is simultaneously providing their own toolchains targeting C++ standard parallelism[7] (`stdpar`). The Nvidia *HPC SDK C++ compiler*, NVC++, targets GPU parallelism using only C++ standard library constructs, as seen in the example in Listing 2.7.

Together, the Intel and Nvidia efforts indicate that the industry is embracing parallel pattern methodologies and moving towards unification and standardization of pattern-based parallel programming. This observation pertains specifically to the domains which favor C++ and similar generalized programming languages, for example traditional HPC applications. Domain-specific toolchains for big data analytics and machine learn-

[6]https://software.intel.com/content/www/us/en/develop/tools/oneapi.html

[7]https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/

Listing 2.7: C++ standard parallelism example from Nvidia.

```
int ave_age =
    std::transform_reduce(std::execution::par_unseq,
                          employees.begin(), employees.end(),
                          0, std::plus<int>(),
                          [](const Employee& emp){
                              return emp.age();
                          })
    / employees.size();
```

ing have shown tendency to accommodate parallel programming faster through dedicated frameworks and tools.

2.4.8 Other related frameworks, libraries, and toolchains

While the number of pattern-based high-level parallel programming systems, libraries, and frameworks is too large to list all of them here, this section tries to cover an assortment of approaches. An attempt has been made to give an idea on the breath of applications of the basic skeleton programming ideas and other pattern-based solutions. Direct competitors to SkePU are also included, as well as work that has been influential in the design of SkePU especially for SkePU 2 and 3. There is a never-ending stream of new frameworks, and it would not be possible to cover them all here. (Examples of more recently proposed frameworks with similarities to the pattern libraries in this chapter include *PHAST* [124] and *SKLP* [97].)

FastFlow

FastFlow [45] is a high-level programming interface targeting *stream parallelism*. FastFlow emphasizes efficiency of basic operations and employs lock-free internal data structures and minimal memory barriers. The FastFlow interface is strongly centered around C++11-style attributes, using source-to-source compilation to generate the FastFlow constructs.

FastFlow was originally designed for multicore CPU execution but added GPU support later.

Lift

Lift [147] is a high-level functional intermediate representation (IR) based on parallel patterns. The goal of Lift is to encode GPU-targeted computations (specifically OpenCL constructs) with an intermediate language which is also adaptable to other computing targets. This language can be targeted by other high-level pattern frameworks. Lift contains the data-parallel patterns `mapSeq`, a map transformation; `reduceSeq`, a reduction; `id`, the identity transform, and `iterate`, which composes a function with itself a spe-

Listing 2.8: Syntax of a simple FastFlow computation.

```
#include <ff/parallel_for.hpp>
using namespace ff;
int main() {
    long A[100];
    ParallelFor pf;
    pf.parallel_for(0,100,[&A](const long i) {
      A[i] = i;
    });
    ...
    return 0;
}
```

cific number of times before applying it to a data set. Lift also has several data-layout patterns such as `split` and `join` in addition to yet more hardware-oriented patterns. The Lift compiler generates efficient backend code by performing optimizations such as barrier elimination and smart data allocation.

Extensions and optimizations to Lift for stencil computations [148] have been carried out without using a specific *stencil* pattern often seen in other pattern frameworks (such as `MapOverlap` in SkePU), demonstrating the strength of composing small building blocks which encode both computation and data layout.

Lift has recently been demonstrated to target high-level synthesis of VHDL code targeting FPGAs. [89]

SkelCL

SkelCL [146] is an OpenCL-based skeleton programming library. It is more limited than SkePU, both in terms of programmer flexibility and available backends. Implemented as a library, it does not require the usage of a precompiler like SkePU, with the downside that user functions are defined as string literals.

SkelCL includes the `AllPairs` skeleton [145], an efficient implementation of certain complex access modes involving multiple matrices. In SkePU 2 matrices are accessed either element-wise or randomly, and `AllPairs` was part of the inspiration for including both the `MapPairs` skeleton and the `MatRow<T>` and `MatCol<T>` container proxies in SkePU 3. This again shows how otherwise similar frameworks based on the same underlying programming model have differences in their approach. Best practices even for fundamental computations such as in this case matrix-matrix multiplications are frequently differing across frameworks.

SkelCL has support for dividing the workload between multiple GPUs, but does not support simultaneous hybrid CPU-GPU execution. As it is based on OpenCL and lacks a precompilation step, the user functions must be defined as string literals, lacking the compile time type checking available in SkePU.

Muesli

Muesli (*Muenster skeleton library*) [22, 56] is a C++ skeleton library built on top of OpenMP, CUDA and MPI, with support for multi-core CPUs, GPUs as well as clusters. Muesli implements both data and task parallel skeletons, but does not have support for as many data parallel skeletons with the same flexibility as in SkePU 3. A `MapStencil` skeleton, similar to `MapOverlap` in SkePU, with backend support for multi-GPU cluster systems was demonstrated at HLPP 2021 [82].

Muesli has support for hybrid execution using a static approach [156], where a single value determines the partition ratio between the CPU and the GPUs, just as in SkePU's hybrid backend. The library also supports hybrid execution using multiple GPUs, with the assumption that they are of the same model. The library currently does not provide an automatic way of finding a good workload distribution which requires the user to manually specify it per skeleton instance.

Marrow

Marrow [106, 143] is a skeleton programming framework for single-GPU OpenCL systems. It provides both data and task parallel skeletons with the ability to compose skeletons for complex computations. The library uses nesting of skeletons to allow for more complex computations. Marrow has support for multi-GPU computations using OpenCL as well as hybrid execution on multi-core CPU, multi-GPU systems. The workload is statically distributed between the CPU threads and the GPUs, just like it is in SkePU. Marrow identifies load imbalances between the CPU and the GPUs and improves the models continuously to adapt to changes in the workload of the system. The partitioning between multiple GPUs is determined by their relative performance, as found by a benchmark suite.

Bones

Bones is a source-to-source compiler based on algorithmic skeletons [117]. It transforms `#pragma`-annotated C code to parallel CUDA or OpenCL using a translator written in Ruby. The skeleton set is based on a well-defined grammar and vocabulary. Bones places strict limitations on the coding style of input programs.

Listing 2.9: Convolution kernel in PSkel, adapted from [126].

```
__stencil__ void stencilKernel(
  Array2D<float> input, Array2D<float> output,
  Mask2D<float> mask, int i, int j)
{
  float accum = 0.0;
  for (int n = 0; n < mask.size; n ++)
    accum += mask.get(n, input, i, j) * mask.getWeight(n);
  output(i,j) = accum;
}
```

PACXX

PACXX is a unified programming model for systems with GPU accelerators [78], utilizing the C++14 language. PACXX was an inspiration in the initial design and prototyping work for SkePU 2 [58], for example using attributes and basing the implementation on Clang. However, PACXX is in itself not an algorithmic skeleton framework.

CU2CL

A different kind of GPU programming research project, *CU2CL* [107] was a pioneer in applying Clang to perform source-to-source transformation; the library support in Clang for such operations has been greatly improved and expanded since then.

PSkel

PSkel [126] is an example of a high-level parallel pattern library focusing only on stencil computations. PSkel provides data abstraction though one, two, and three-dimensional arrays and matching mask objects. The C++ template library is used to specify element-wise stencil kernels which are computed by PSkel offloading to either CUDA, OpenMP, or TBB. Abstractions enable array and mask indexing using either linear or dimensional coordinates.

Qilin

Qilin [103] is a programming model for heterogeneous architectures, based on TBB and CUDA. Qilin provides the user with a number of pre-defined operations, similar to the skeletons in SkePU. The library has support for hybrid execution by automatically splitting the work between a multi-core CPU and a single NVIDIA GPU. Just as in SkePU, one of the CPU threads is dedicated to communication with the GPU. The partitioning is based on linear performance models created from training runs, much like SkePU's auto-tuner implementation.

Lapedo

Recent work in hybrid CPU-GPU execution of skeleton-like programming constructs include *Lapedo* [83], an extension of the *Skel* Erlang library for stream-based skeleton programming, specifically providing hybrid variants of the Farm and Cluster skeletons where the workload partitioning is tuned by models built through performance benchmarking; and Vilches' et al. [115] TBB-based heterogeneous parallel for template, which actively monitors the load balance and adjusts the partitioning during the execution of the for loop. Both approaches exclusively use OpenCL for GPU-based computation.

2.5 Independent surveys

De Sensi et al. [50] have contributed the P3ARSEC benchmark suite, intended to cross-evaluate high-level parallel programming frameworks and libraries, specifically pattern-based ones. Being based on a subset of the original PARSEC [15] benchmark suite, P3ARSEC is intended as a means to compare performance, but just as importantly, programmability aspects. The authors specifically highlight *lines of code*, the total length of a program, and *code churn*, the number of changes lines when converting a previous (often sequential) application to using the high-level interface, as measures of programming effort. The work is also intended to prove the viability of skeleton programming (or pattern-based parallel programming) at large, and the results demonstrate that 12 out of 13 PARSEC benchmark can be expressed by a small set of common patterns, specifically using *FastFlow* [2].

Arvanitou et al. [6] conducted a technical debt (TD; a financial metaphor for measuring software quality and maintainability) [9] investigation on parallel programming using SkePU and StarPU, specifically analyzing the trade-offs between portability, performance, and maintenance. In the study, SkePU was considered representing a highly portable implementation and for StarPU performance was emphasized. The results show that SkePU does not seem to negatively affect technical debt across three studied applications.

2.6 Earlier related work on SkePU

The work presented in this thesis does not stretch back to the inception of SkePU as a skeleton library. Even though the interface has changed in fundamental ways, the current version of the SkePU framework is either directly reliant on, or builds in top of, contributions by the people who have worked on SkePU before.

We refer to earlier SkePU publications [47, 48, 55] for other work relating to specific features, such as smart data-containers.

3 SkePU overview

SkePU is central to all contributions in this thesis. While there will be more detailed coverage of SkePU in the following chapters, the way each individual part of SkePU interact with the others requires that we first start with a high-level overview of the framework, as follows in this chapter.

3.1 Basic constructs

SkePU [55, 59, 62] is a multi-backend skeleton programming framework for heterogeneous parallel systems with a C++11-based interface. A SkePU program defines *user functions* which act as the operators applied in the skeleton algorithms. SkePU contains both a source-to-source transforming precompiler and a runtime library, working in tandem to transform high-level application code and execute it in parallel in the best possible way on available computational units, providing *performance portability*. As the precompiler is aware of the C++ constructs that represent skeletons, it can rewrite the source code and generate backend-specific versions of the user functions.

Listing 3.1 shows an example application implemented on top of SkePU: computation of the Pearson product-moment correlation coefficient.

Listing 3.1: A SkePU program computing the Pearson product-moment correlation coefficient of two vectors.

```
#include <iostream>
#include <cmath>
#include <skepu>

// Unary user function
float square(float a)
{
  return a * a;
}

// Binary user function
float mult(float a, float b)
{
  return a * b;
}

// User function template
template<typename T>
T plus(T a, T b)
{
  return a + b;
}

// Function computing PPMCC
float ppmcc(skepu::Vector<float> &x, skepu::Vector<float> &y)
{
  // Instance of Reduce skeleton
  auto sum = skepu::Reduce(plus<float>);

  // Instance of MapReduce skeleton
  auto sumSquare = skepu::MapReduce(square, plus<float>);

  // Instance with lambda syntax
  auto dotProduct = skepu::MapReduce(
  [] (float a, float b) { return a * b; },
  [] (float a, float b) { return a + b; }
  );

  size_t N = x.size();
  float sumX = sum(x);
  float sumY = sum(y);

  return (N * dotProduct(x, y) - sumX * sumY)
    / sqrt((N * sumSquare(x) - pow(sumX, 2))
      * (N * sumSquare(y) - pow(sumY, 2)));
}

int main()
{
  const size_t size = 100;

  // Vector operands
  skepu::Vector<float> x(size), y(size);
  x.randomize(1, 3);
  y.randomize(2, 4);
```

```
    std::cout << "X: " << x << "\n";
    std::cout << "Y: " << y << "\n";

    float res = ppmcc(x, y);

    std::cout << "res: " << res << "\n";

    return 0;
}
```

For data abstraction, SkePU provides *smart data-containers* which manage coherency states automatically. Smart data-containers are available in different shapes:

- **Vector**, for one-dimensional data sets;
- **Matrix**, suitable for two-dimensional data, e.g., images;
- **Tensor**, for three-dimensional or four-dimensional data sets of fixed size.

SkePU as of version 3 includes the following skeletons:

- **Map**, data-parallel element-wise application of a function with arbitrary arity;
- **MapPairs**, cartesian product-style computation, pairing up two one-dimensional sets to generate a two-dimensional output;
- **MapOverlap**, stencil operation in one or two dimensions with various boundary handling schemes;
- **Reduce**, generic reduction operation with a binary associative operator;
- **Scan**, generalized prefix sum operation with a binary associative operator;
- **MapReduce**, efficient nesting of Map and Reduce;
- **MapPairsReduce**, efficient fusion of MapPairs and Reduce;
- **Call**, a generic multi-variant component for computations that may not fit the other available skeleton patterns.

Figure 3.1 contains a visual analogy for the most central operation in SkePU, which is the composition of skeleton instances from the patterns provided by the framework together with the operators from the user, self-contained code snippets which are the ”user functions”. We illustrate this using a puzzle sketch, where the user function is a small puzzle piece made

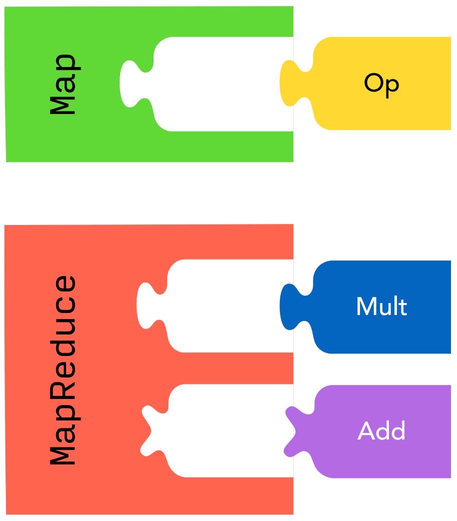


Figure 3.1: Puzzle piece analogy used to illustrate how skeletons are user-parametrizable.

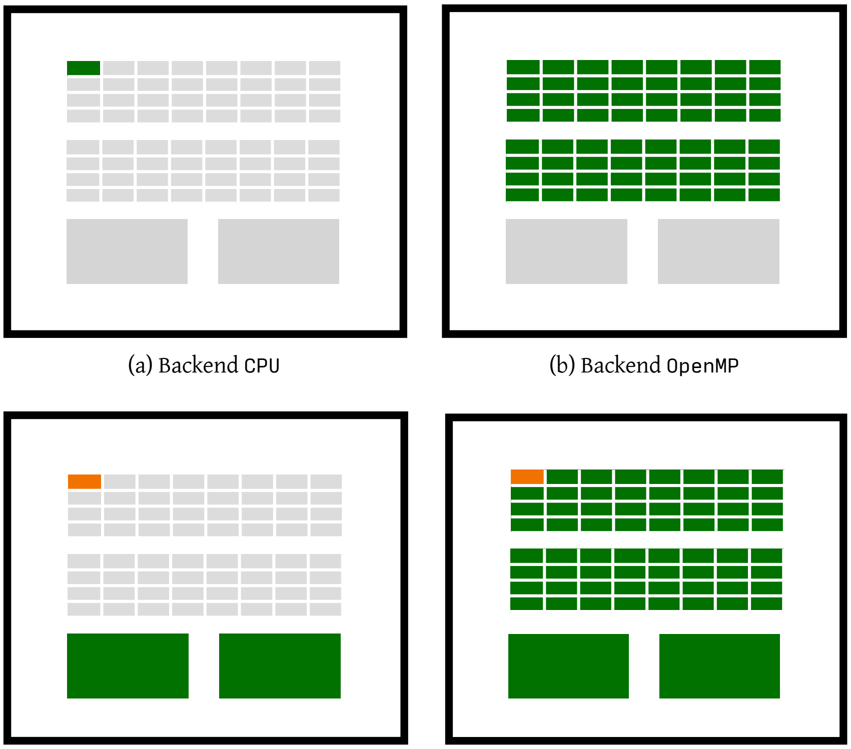
to fit into the empty space of the provided pattern construct. To some extent this analogy works well to convey the fundamental idea, and the shape of the puzzle piece (e.g., the prongs) can be formed to indicate that a certain interface is required from the user code, as seen in Figure 3.1 with the `MapReduce` skeleton (requiring that the reduction function is binary and associative; more on this topic in Section 4.7). However, as will be made clear further on throughout Chapter 4, the skeleton set of SkePU offers flexibility, and in particular, *variadicity* which allows for the skeleton constructs to shape their interfaces to accommodate a wide variety of user function signatures. Therefore, the puzzle-piece analogy can be seen as understating the capabilities of the framework, but nonetheless functional in serving as a first introduction.

Section 4.1 goes into much more depth on the particular modes and features of each individual skeleton.

SkePU provides *smart data-containers* [47], data structures that reside in main memory but can temporarily store subsets of their elements in accelerator memories for access by skeleton backends executing on these devices. Smart data-containers also perform software caching of the operand elements to keep track of valid copies of their element data, resulting in automatic optimization of communication and device memory allocation. Smart data-containers are well suited for iterative computations, where the performance gains can be significant. Smart data-containers are further detailed in Chapter 5.

Compute
Management
Idle

Figure 3.2: Color legend for Figures 3.3 and 3.4.



(a) Backend `CPU`

(b) Backend `OpenMP`

(c) Backend `OpenCL` and `CUDA`

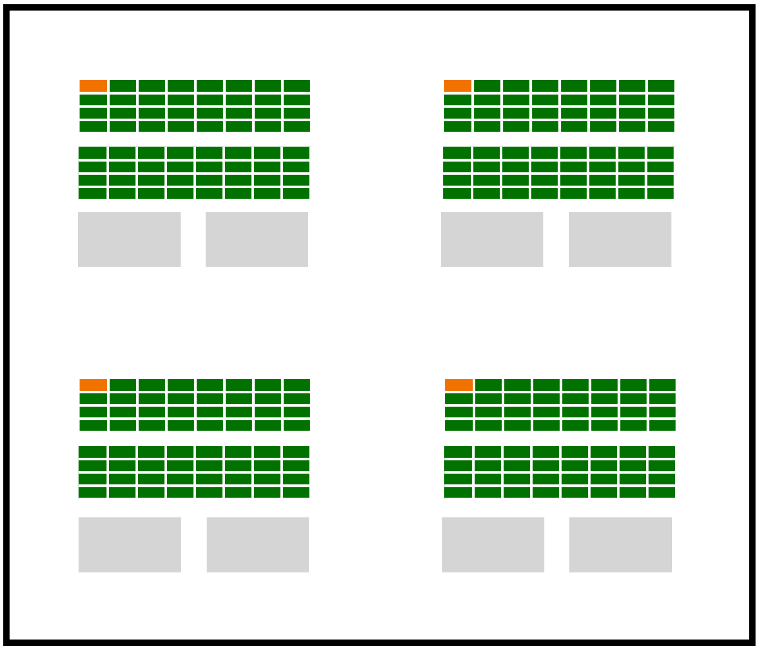
(d) Backend `Hybrid`

Figure 3.3: Node-level backends of SkePU.

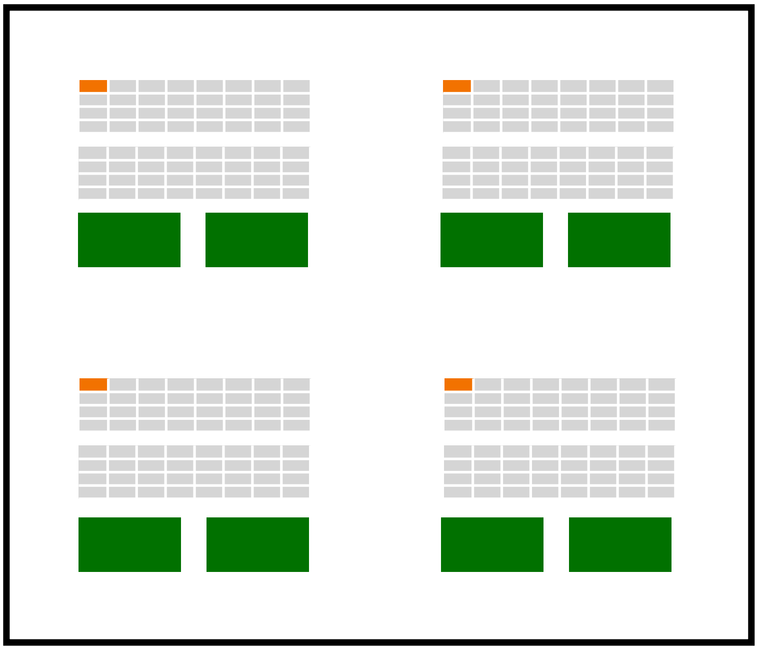
3.2 Backend architecture

SkePU has seven different *backends*, implementing the skeletons for different hardware configurations. These are as follows:

- **Sequential CPU backend:** Mainly used as a reference implementation and baseline, the sequential "CPU" backend (Figure 3.3a) uses only one thread, and thus only one logical CPU core.
- **OpenMP backend:** OpenMP targets multi-core CPUs, and will use anything from one thread to as many as requested by the user (Figure 3.3b). In most cases, there is little reason to use more threads than



(a) Backend `StarPU-MPI`



(b) Backend `StarPU-MPI-CUDA`

Figure 3.4: Cluster-level backends of SkePU.

the target system has logical cores, and for certain classes of workloads, little if any performance increase is gained from going over the number of physical cores. The OpenMP backend does not need to reserve a core for thread management; all threads are used for computation.

- **CUDA backend**: Targets Nvidia GPUs only, either single or multiple. One CPU thread is used to schedule and manage the GPU computations. (Figure 3.3c).
- **OpenCL backend**: Targets single and multiple GPUs of any OpenCL supported model (Figure 3.3c), including other accelerators such as Intel Xeon Phis. OpenCL can also target CPU cores, but SkePU does not surface this capability. Like in the CUDA backend, one CPU thread is used for management.
- **Hybrid backend**: An intermediate control layer that splits up work on two other backends simultaneously. Currently supports the OpenMP backend in combination with either of the CUDA or OpenCL backends (Figure 3.3d). See Chapter 8.
- **StarPU-MPI**: Cluster backend operating in an SPMD mode. This backend uses the StarPU runtime system for scheduling workload across large-scale cluster systems, including supercomputers (Figure 3.4a). This variant uses only CPU cores, and reserves one thread for management duties. See Chapter 9.
- **StarPU-MPI-CUDA**: A variant of the StarPU-MPI cluster backend which instead offloads all computations to GPUs using CUDA (Figure 3.4b).

An additional cluster backend built on top of an alternate programming model is presented in Chapter 9.

Backend selection is either automatic, guided by an auto-tuning system, or manually configured by the application programmer. SkePU abstracts everything related to backend code execution, such as OpenMP directives or OpenCL kernel launching. However, certain configuration parameters are optionally exposed[1] as part of the manual backend selection interface, such as thread count. Smart data-containers provide the abstraction layer for backends with separate or split memory spaces, with data movement handles automatically by SkePU before and after backend delegation of skeleton computations, as discussed above and in greater detail later in the thesis (Section 5.1).

[1]This is in particular useful for debugging and performance measurements.

3.3 History

SkePU (version 1) was introduced in 2010 [55] as a skeleton programming library for heterogeneous single-node but multi-accelerator systems, from the beginning designed for portability to include single- and multi-GPU backends for the C-based OpenCL and for CUDA (which then only partly supported C++), and had thus been technically based on C++03 and on C preprocessor macros as the interface to user functions.

SkePU 2, introduced in 2016 [62], was a major revision of the SkePU [55] library, ushering in ideas from modern C++ to the skeleton programming landscape. Rebuilding the interface from the ground up, the skeleton set was updated to be variadic, leaving the old fixed-arity skeletons from SkePU 1 behind. Variadic skeleton signatures was the first main motivator of SkePU 2: *flexible* skeleton programming.

This rewrite also took the opportunity to integrate patched-on functionality in SkePU 1 into the core design of the programming model. One such example is the absorption of SkePU 1 `MapArray` into the basic SkePU 2 `Map`. `MapArray` was a dedicated skeleton in SkePU 1 created as a clone of `Map` with the ability to accept an auxiliary, random-accessible array operand into the user function, allowing deviations from the strictly functional map-style patterns when demanded by the target application. This was one of the first lessons from practical experience [142] that skeleton patterns are not always perfectly suited to algorithms in real-world application code.

SkePU 2 also introduced the *precompiler*, lifting SkePU from its humble origins as a pure template include-library into a full-fledged *compiler framework*. This, together with the effort to push the C++ type system farther than most, if not all, comparable frameworks enabled the second main motivator of SkePU 2: *type-safe* skeleton programming.

Table 3.1 gives a synopsis of the different features of the three main SkePU versions.

3.4 SkePU 2 design principles

SkePU was conceived and designed in 2010 with the goal of portability across very diverse programming models and toolchains such as CUDA and OpenMP; since then, there has been significant advancements in this field in general and to C++ in particular. C++11 provides fundamental performance improvements, e.g., by the addition of move semantics, `constexpr` values, and standard library improvements. It introduces new high-level constructs: range-for loops, lambda expressions, and type inference among others. C++11 also expands its metaprogramming capabilities by introducing variadic template parameters and the aforementioned `constexpr` feature. Finally, the new language offers a standardized notation for attributes used for language extension. The proposal for this feature explicitly discussed parallelism as a possible use case [110], and it had been successfully used in, for example, the REPARA project [43]. Even though C++11 was standardized in 2011, it was only in the time around the introduction of SkePU 2 that compiler support was getting widespread, see, e.g., Nvidia's CUDA compiler.

For this project, we specifically targeted improvement of the following limitations of SkePU 1:

- **Type safety**: Macros are not type-safe and SkePU 1 does not try to work around this fact. In some cases, errors which semantically belong in the type system will not be detected until run-time. For example, SkePU 1 does not match user function type signatures to skeletons statically, see Listing 7.3. This lack of type safety is unexpected by C++ programmers.

- **Flexibility**: A SkePU 1 user can only define user functions whose signature matches one of the available macros. This resulted in a steady increase of user function macros in the library: new ones have been added ad-hoc as increasingly complex applications has been implemented on top of SkePU. Some additions also required more fundamental modifications of the runtime system. For example, when a larger number of auxiliary containers was needed in the context of `MapArray`, an entirely new `MultiVector` container type [142] had to be defined, with limited smart container features. Real-world applications need more of this kind of flexibility.

  An inherent limitation of all skeleton systems is the restriction of the programmer to express a computation with the given set of predefined skeletons. Where these do not fit naturally, performance will suffer. It should rather be possible for programmers to add their own multi-backend components [49] that could be used together with SkePU skeletons and containers in the same program and reuse SkePU's auto-tuning mechanism for backend selection[2].

- **Optimization opportunity**: Using the C preprocessor for code transformation drastically limited the possible specializations and optimizations which can be performed on user functions, compared to, e.g., template metaprogramming or a separate source-to-source compiler. A more sophisticated tool could, for example, choose between separate *specializations* of user functions, each one optimized for a different target architecture. A simple example is a user function specialization of a vector sum operation for a system with support for SIMD vector instructions.

[2]The initial release of SkePU 2 presented the `Call` skeleton as a first step towards this goal. Later, the addition of multi-variant user functions [61] (Chapter 12) provided further contribution in this direction.

Listing 3.2: Vector sum computation in SkePU 1.

```
BINARY_FUNC(add, float, a, b,
  return a + b;
)

skepu::Vector<float> v1(N), v2(N), res(N);

skepu::Map<add> vec_sum(new add);
vec_sum(v1, v2, res);
```

- **Implementation verbosity**: SkePU 1 skeletons were available in multiple different modes and configurations. To a large extent, the variants were implemented separately from each other with only small code differences. Using the increased template and metaprogramming functionality in C++11, a number of these could be combined into a single implementation without loss of (run-time) performance.

SkePU 2 built on the mature runtime system of SkePU 1: highly optimized skeleton algorithms for each supported backend target, smart data-containers, multi-GPU support, etc. These were preserved and updated for the C++11 standard. This is of particular value for the `Map` and `MapReduce` skeletons, which in SkePU 1 were implemented thrice for unary, binary and ternary variants; in SkePU 2 and later, a single variadic template variant covers all N-ary type combinations. There are similar improvements to the implementation wherever code clarity can be improved and verbosity reduced with no run-time performance cost.

The main changes in SkePU 2 were related to the programming interface and code transformation. SkePU 1 used preprocessor macros to transform user functions for parallel backends; SkePU 2 and 3 instead utilizes a source-to-source translator (precompiler), a separate program based on libraries from the open source Clang project[3]. Source code is passed through this tool before normal compilation. This remains true for SkePU 3 and is discussed in detail in Chapter 7.

Listings 3.2 and 3.3 contains a vector sum computation respectively in SkePU 1 and SkePU 2 syntax, showing the interface changes across versions. Listing 3.4 shows the equivalent code for SkePU 3 for completeness, but the changes from SkePU 2 are trivial.

3.5 SkePU 3 design principles

The, as of the time of writing, all-new SkePU version 3 builds on top of the redesign in SkePU 2 while largely retaining the existing syntax and fea-

[3]http://clang.llvm.org

Listing 3.3: Vector sum computation in SkePU 2.

```
template<typename T>
T add(T a, T b)
{
  return a + b;
}

skepu2::Vector<float> v1(N), v2(N), res(N);

auto vec_sum = skepu2::Map<2>(add<float>);
vec_sum(res, v1, v2);
```

Listing 3.4: Vector sum computation in SkePU 3.

```
template<typename T>
T add(T a, T b)
{
  return a + b;
}

skepu::Vector<float> v1(N), v2(N), res(N);

auto vec_sum = skepu::Map(add<float>);
vec_sum(res, v1, v2);
```

ture set. For SkePU 3 the design focus is on meeting the requirements of real-world skeleton programming and the use of SkePU with HPC clusters, larger-scale applications and build systems. This work was done in close collaboration with partners from both the scientific community and the industry, as part of the EXA2PRO project.

The approach is holistic, with advancements being done in aspects ranging from syntactical simplification of common constructs and idioms to a re-evaluation of the memory coherency model of SkePU data-containers and the introduction of all-new skeletons and other features.

Some particularly important focus areas are as follows:

- **Skeleton set**: `MapPairs` is introduced as a new skeleton, a generalization of the map pattern for cartesian combinations of container elements; as well as `MapPairsReduce`, a sibling to `MapPairs` with efficient partial reduction of results. Other skeletons were revised and updated with new features, including a new syntax and update modes for `MapOverlap`.

- **Smart data-containers**: The container set is amended by the addition of tensors, supporting higher-dimensional access patterns, and new container proxies (`MatRow`, `MatCol`) allowing e.g. for more scalable data movement on clusters.

- **Memory coherency model**: The coherency model of out-of-skeleton container access is clearly defined, to help increase predicability and performance.
- **Syntactical improvements**: Programmability and readability of SkePU-ized code is improved in response to feedback and experiences from users, including developers of large-scale scientific applications.
- **Transparent execution on HPC clusters**: The single-source, wide portability approach of SkePU programs is extended to cover computation over multiple nodes in HPC clusters without any cluster-specific programming constructs in the source code, thus fully abstracting away the underlying distributed platform.
- **Reduction of boilerplate code**: The addition of a standard library is aimed towards eliminating the need to define common user functions, such as simple arithmetic operators. Other parts of the library provide abstractions for common functionality required in SkePU applications, for example complex numbers, linear algebra primitives, and random number generation.

Refinement work of SkePU 3 continues as of writing, and more features and enhancements will be added.

Table 3.1: Overview of SkePU Features

	SkePU 1 (2010) [55]	**SkePU 2** (2016) [62]	**SkePU 3** (2020) [59]
API based on	C, C++ (pre-2011),	C++11,	C++11,
Code generator	C preprocessor	Precompiler (clang)	Precompiler (clang, mcxx)
Skeletons	`Map`, `Reduce`, `Scan`, `MapReduce`, `MapArray`, `MapOverlap`, `Generate`	`Map`, `Reduce`, `Scan`, `MapReduce`, `MapOverlap`, `Call`	`Map`, `Reduce`, `Scan`, `MapReduce`, `MapOverlap`, `Call`, `MapPairs`, `MapPairsReduce`
User functions as	C preprocessor macros	Restricted C++ functions	Restricted C++ functions, plus multi-variant user functions
Compound types	N/A	User structs	User structs plus tuples
Skeleton interface	Not type-safe, fixed arity	Type-safe and variadic	Type-safe and variadic
Containers	`Vector<>`, `Matrix<>`	`Vector<>`, `Matrix<>`	`Vector<>`, `Matrix<>`, `Tensor3<>`, `Tensor4<>`, `MatRow<>`, `MatCol<>`, `Region<>`
Platforms supported	CPU (C, OpenMP), GPU (CUDA, OpenCL)	CPU (C++, OpenMP), GPU (CUDA, OpenCL)	CPU, GPU, hybrid CPU/ GPU, StarPU-MPI, ...
Scheduling (OpenMP)	Static	Static	Static, Dynamic
Memory model	Sequential consistency	Sequential consistency	Weak consistency (default), optionally sequential consistency
Standard library	N/A	N/A	PRNG, complex numbers, BLAS, utilities, etc.

4 Skeleton set

The application programming interface, API, of the SkePU framework is one of its most central aspects. In high-level parallel and heterogeneous programming, the interface is what determines whether the goal of being ”high-level” is met. Being high-level is not an absolute metric; it is a moving target as the field of computer science and engineering progresses. The C programming language was originally seen as quite high-level, and there are still people working as programmers in assembly language who might hold that view. In contemporary high-level parallel programming libraries and frameworks, however, C or Fortran are generally avoided. SkePU makes extensive use of C++ syntactical features such as classes, templates, and lambda expressions. Raw pointers and arrays are absent from the interface as well.

While syntactical aspects are important, the true strength of a high-level programming interface lies in the available constructs. In the skeleton programming paradigm these manifest naturally as the skeleton patterns themselves, and the *skeleton set* is often considered one of the most fundamental defining aspects of a particular skeleton programming implementation. In this chapter, the skeleton set of SkePU is explored in detail in Section 4.1, with a special emphasis on the most fundamental data parallel skeleton `Map`. This skeleton is a natural starting point to introduce characteristic SkePU features such as auxiliary data set access in Section 4.2.1, flexible variadic signatures in Section 4.2.2, tuple returns in Section 4.2.3, and index-dependent computations in Section 4.2.4. Continuing the skeleton set exposé, `MapPairs` is presented in Section 4.3 and `MapOverlap` in Section 4.4; both being specialized variants of the map pattern. Section 4.5 details the `Reduce` skeleton, the similar `Scan` is featured in Section 4.6. This is followed by the fused skeletons `MapReduce` in Section 4.7 and `MapPairsReduce` in Section 4.8. Finally `Call` in is covered in Section 4.9.

While the skeleton patterns are provided by the framework, they need user code to be instantiated and used in applications. Because of backend compatibility requirements, not any C++ code can be used in this way. The chapter therefore continues by covering the different ways a SkePU programmer can adapt the skeletons for their purposes: *user functions* in Section 4.10, *user types* in Section 4.11, and *user constants* in Section 4.12. The recent addition of *strided* element access is discussed in Section 4.13.

This thesis aims to give insight into the design and implementation behind the SkePU framework; the content in this chapter is not intended as a specification on SkePU, nor as an introductory guide to SkePU programming. Such documentation can be found on the SkePU web page.[1]

4.1 Skeleton set

SkePU provides a number of skeletons which represent different *data-parallel* patterns, as mentioned in Chapter 3. The skeletons can be loosely ordered into three groups: the map-based `Map`, `MapPairs`, and `MapOverlap`, being element-wise transformations of data; `Reduce` and `Scan`, two forms of data accumulation patterns with internal dependency structures; and explicit *fusions* of a map-based skeleton in sequence with some form of reduction in `MapReduce` and `MapPairsReduce`. `Call` is a pseudo-skeleton and does not fit into any grouping. Table 4.1 summarizes skeleton attributes and features to show similarities and differences between them.

[1]`https://skepu.github.io`

Table 4.1: Skeleton feature matrix

Feature \ Skeleton	**Map**	**MapPairs**	**MapOverlap**	**Reduce**	**Scan**	**MapReduce**	**MapPairsReduce**	**Call**
Elwise dimension in	1–4	1	1–4	1–4**	1	1–2	1	0
Elwise dimension out	Same as *in*	2	Same as *in*	0–1	1	0	1	0
Indexed	Yes	Yes	Yes	-	-	Yes	Yes	-
Multi-return	Yes	Yes	Yes	-	-	Yes	Yes	-
Elwise parameters	Variadic	Variadic x2	1***	*	*	Variadic	Variadic x2	-
Full proxy parameters	Variadic	Variadic	Variadic	-	-	Variadic	Variadic	Variadic
Uniform parameters	Variadic	Variadic	Variadic	-	-	Variadic	Variadic	Variadic
Region proxy	-	-	Yes	-	-	-	-	-
MatRow/MatCol proxy	Yes	Yes	-	-	-	Yes	Yes	-
Footnotes								
*	Parameters to the user functions can be raw elements from the container or partial results, depending on evaluation							
**	Dimensions higher than 2 are linearized in the current implementation.							
***	A region of elements surrounding the current index is supplied.							

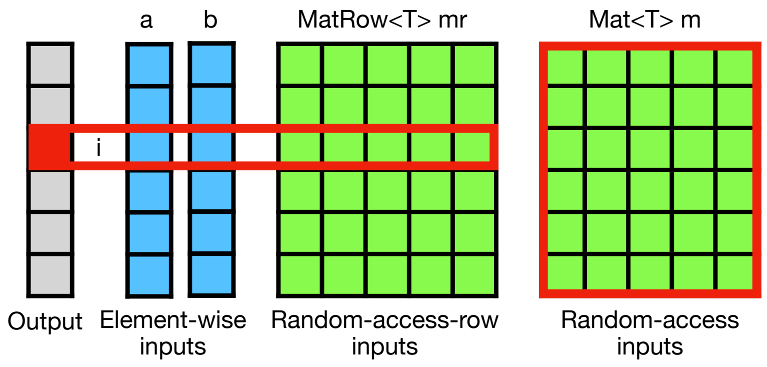


Figure 4.1: Illustrative diagram of the operand access scopes in the `Map` skeleton.

4.2 Map skeleton

Map is a term widely used in programming interfaces, sequential as well as parallel, as a name for a construct that transforms a set of values to another set of values in accordance with some transformation (mapping) function f. This function is typically a pure function, deterministic and without side effects, which aids the compiler or interpreter in automatic program translation and optimization. In a statically typed language like C++, the types of the domain and image are fixed but typically they can be different from each other.

SkePU borrows the *map* label for its `Map` skeleton. While `Map` is and does everything in the preceding paragraph, its versatility and importance in SkePU greatly exceeds that of typical map constructs. `Map` is the *fundamental building block* of the SkePU programming interface: it is the default building block for encoding data parallel computations unless a particularly specific pattern is needed, and in those cases, the vast majority of skeleton patterns

Listing 4.1: Example usage of the Map skeleton.

```
float add(float a, float b)
{
  return a + b;
}

Vector<float> vector_sum(skepu::Vector<float> &v1, skepu::Vector<float> &v2)
{
  auto vsum = skepu::Map(add);
  skepu::Vector<float> result(v1.size());
  return vsum(result, v1, v2);
}
```

in SkePU are directly based upon the foundations of `Map`. Indeed, the names tell the story: `MapReduce`, `MapPairs`, `MapPairsReduce`, and `MapOverlap` are all either specialized variations of `Map` or fusions with another pattern. The important role played by `Map` means that understanding the syntax, capabilities, and limitations of this skeleton is of utmost importance for anyone interested in using or otherwise learning SkePU.

4.2.1 Freely accessible containers inside user functions

Map patterns often only concern themselves with providing a single element from the input data set as argument to the mapping operator. To perform a computation with a non-trivial dependency pattern, the operators can be defined as *lexical closures* which capture the enclosing scope, allowing the use of any free variables inside the operator.

The multi-backend nature of SkePU makes such constructions impractical from an implementation standpoint.[2] The backend environments can have different programming models and the memory spaces are typically separate from the C++ domain perceived by the SkePU user. SkePU therefore require that any auxiliary data structures—limited to smart data-containers and scalar values—are declared as *bound variables* in the user function signature. There are particular rules for how these objects are declared and passed, discussed in Section 4.2.2.

SkePU smart data-containers are C++ objects of intricate class templates, and cannot be made available in a backend execution context. Therefore, smart data-containers as bound variables in user functions are encoded as *proxy containers*, further covered in Section 5.2. Listing 4.2 illustrates the use of auxiliary smart data-containers in the matrix-vector multiplication skeleton instance `mvmult`[3] and Figure 5.5 illustrates how using proxies bring entire container data sets into the user function. This is a `Map` instance with no element-wise inputs, which is a surprisingly powerful construct enabled by the SkePU design principles presented in Section 4.2.2.

4.2.2 Variadic type signatures

The central aspect of `Map` which gives it this flexibility and power is the *variadic interface*. Along with type-safety, flexibility was the main contribution of the original SkePU 2 paper [62] and master thesis [58], and prompted the

[2]SkePU user functions may be defined as lambda expressions, which can act as lexical closures in C++, but SkePU treats them strictly as "syntactic sugar". See Section 4.10.1 for further discussion.

[3]Note that this is not the preferred way to encode matrix-vector multiplication since SkePU 3, with the introduction of the `MatRow` proxy container. A better way is shown in Listing 5.2.

Listing 4.2: Matrix-vector multiply in the SkePU 2 style, without MatRow.

```
template<typename T>
T mvmult_f(skepu::Index1D row, const skepu::Mat<T> m, const skepu::Vec<T> v)
{
  T res = 0;
  for (size_t j = 0; j < v.size; ++j)
    res += m(row.i, j) * v(j);
  return res;
}

skepu::Vector<float> y(height), x(width);
skepu::Matrix<float> A(height, width);
auto mvmult = skepu::Map(mvmult_f);
mvmult(y, A, x);
```

complete API redesign from SkePU 1.[4] The underlying C++11 features which enable this generational leap[5] are designed to be used by framework engineers, and the significant complexity of implementation is elegantly hidden beneath the framework boundaries. For the SkePU user, it means that using the map construct is very easy for trivial computations but enables great adaptivity for more involved situations.

A `Map` skeleton instance and the corresponding user function are *four-way variadic*. Arguments of a call to the instance are effectively grouped into four sets:

- output arguments (see Section 4.2.3),
- element-wise input arguments,
- random-access input arguments, and
- uniform input arguments.

The size (henceforth *arity*) of each group is flexible and up to the user to choose based on the use-case at hand. The only restriction is that there has to be at least one output argument.[6] All `Map`-like skeletons in SkePU use the output container to determine the *degree of parallelism*: each element corresponds to an invocation of the user function and is an independent task that could be mapped and scheduled for execution as a unit. It does not matter how each group is ordered internally, but the relative order of each group must be taken: outputs come first, followed by element-wise containers (if

[4]The original impetus for this change was that the SkePU 1 model of having separate skeletons for unary, binary, and ternary `Map` is not ideal neither from a user nor maintainer perspective in a high-level parallel programming framework.

[5]Mainly variadic templates and advances in template meta-programming: the same techniques behind the implementation of, e.g., `std::tuple` from the C++ standard library.

[6]`Call` is much like a `Map` with no return value or element-wise arguments.

any), followed by random-access containers (if any), and finally uniform scalars (if any).

Element-wise (”elwise”) parameters in a user function are scalar values (or user types, see Section 4.11), with the corresponding arguments in a skeleton invocation being SkePU containers. Each element of the container is uniquely mapped to the parameter of a single user function invocation, in a data-parallel fashion. Random access parameters and arguments are both containers (but expressed slightly differently, as explained later) and all elements are accessible from within a single user function invocation.

In user function definitions, the function signature encodes the outputs as the return type of the function and the rest of the arguments come within the parentheses. Extra care has to be observed when crafting a user function, since SkePU uses the function signature when determining the type information for a skeleton instance. Because random-access container arguments are represented as *container proxy types* (see Sections 4.2.1 and 5.2) in the user function signature, the four groupings have natural separations in the type system. SkePU uses template meta-programming and the pre-compiler to analyze the types in the function header and construct the internal groupings. Figure 5.5 contains an illustration of how the parameter groups bring data from the arguments into the user function in different ways.

Astute readers may notice that the random-access container group is allowed to be empty, in which case the distinction between where the element-wise arguments end and the uniform scalars begin is unavailable. A `Map` instance definition can optionally contain an explicit template argument, as in `auto instance = skepu::Map<N>(...);` where `N` denotes the element-wise arity, and if not present in the construct, SkePU will make a best-guess deduction based on the parameter list (the *formal arguments*) of the user function. Skeleton instances are fully statically typed, so if the deduced arity differs from the *actual arguments* at the skeleton invocation site, a compile-time error occurs.[7]

4.2.3 Multi-valued return

SkePU 3 introduced tuple-like return functionality for cases where a single skeleton instance requires multiple (element-wise) output containers. This way, multiple return values can be computed by the same user function, operating on the inputs in one sweep, potentially improving data locality compared to two separate skeleton invocations after each other. Although the values are returned in a tuple-like manner, the output containers are

[7]The pre-compiler has access to the entire AST and can in principle look at both skeleton instantiation and skeleton invocations for arity deduction; however, SkePU is designed and implemented (Chapter 7) such that programs are semantically sound C++ programs also without the pre-compiler.

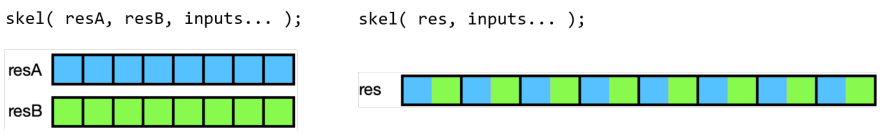


Figure 4.2: Difference in return value storage between using multi-valued return (left) and single-value (by manually managed array-of-struct) return (right).

Listing 4.3: User function with multi-valued return.

```
skepu::multiple<int, float>
multi_f(int a, int b, skepu::Vec<float> c, int d)
{
  return skepu::ret(a * b, (float)a / b);
}
```

Listing 4.4: Using multi-valued return with Map in SkePU 3.

```
skepu::Vector<int> v1(size), v2(size), r1(size), r2(size);
skepu::Vector<float> e(1);

auto multi = skepu::Map<2>(multi_f);

multi(r1, r2, v1, v2, e, 10);
```

completely separate objects (see Figure 4.2). This distinguishes this new feature from the use of custom structs ("user types", see Section 4.11) as return values, as those are stored in array-of-records format.

To use this feature, we specify the return type in the user function signature as `skepu::multiple<T, [U, ...]>`, i.e., analogous to `std::tuple`. Then, at the site of the `return` statement, we construct this compound object by `skepu::ret(expr, [expr, ...])`.

Listing 4.3 shows an example of a user function utilizing multi-valued return.

The skeleton instance declaration and invocation follows the syntax of ordinary `Map`, but instead of supplying one output container as the first argument, specify several of the correct types and order, as in Listing 4.4.

Multi-valued return statements are available in the skeletons which follow the typical map pattern: `Map`, `MapPairs`, and `MapOverlap`.

4.2.4 Index-dependent computations

Another feature of `Map` is the option to access the index for the currently processed container element to the user function. This is handled automatically, deduced from the user function signature. An index parameter's type

Listing 4.5: Index types corresponding to each smart container.

```
struct Index1D { size_t i; };
struct Index2D { size_t row, col; }; // note!
struct Index3D { size_t i, j, k; };
struct Index4D { size_t i, j, k, l; };
```

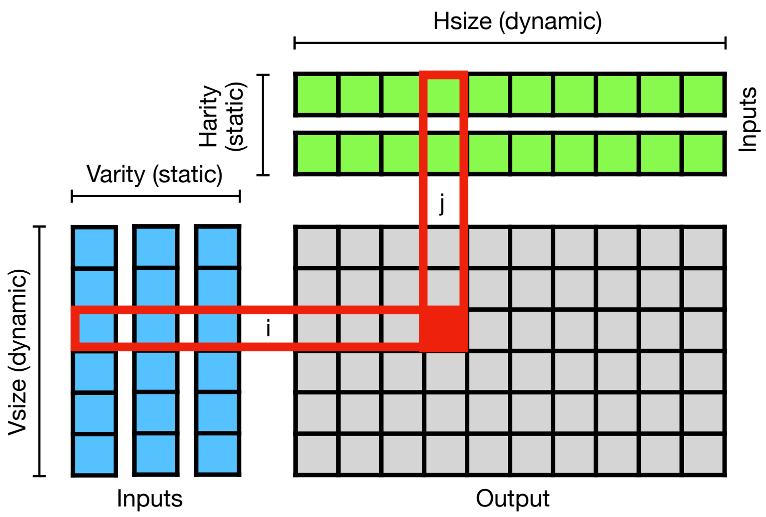


Figure 4.3: Illustrative diagram of the `MapPairs` skeleton.

is one out of four types: `IndexND` where `N` is the dimensionality of the index, as shown in the type declarations in Listing 4.5. This feature replaces the dedicated `Generate` skeleton of SkePU 1, allowing for a commonly seen pattern—calling `Generate` to generate a vector of consecutive indices and then pass this vector to `MapArray`—to be implemented in one single `Map` call.

The Mandelbrot fractal generation in Listing 4.17 is a typical example of a computation where the user function is reliant on the current index into the resulting `Matrix` container.

4.3 MapPairs skeleton

SkePU 3 added an additional top-level skeleton, `MapPairs`. This skeleton represents a Cartesian product-style pattern, operating on two distinct sets of element-wise container inputs. Each vector set may contain an arbitrary number of vector containers, similar to the variadicity of `Map`. All of the vectors in a set are expected to be of the same size. The arities in both di-

Listing 4.6: Example usage of the `MapPairs` skeleton.

```
int mul(int a, int b) { return a * b }

void cartesian(size_t Vsize, size_t Hsize)
{
  auto pairs = skepu::MapPairs(mul);

  skepu::Vector<int> v1(Vsize, 3), h1(Hsize, 7);
  skepu::Matrix<int> res(Vsize, Hsize);
  pairs(res, v1, h1);
}
```

rections are always present in the skeleton construction as explicit template arguments.

Each Cartesian combination of vector set indices generates one user function invocation, the result of which is an element in a `Matrix`. As in `Map`, there is an optional `Index2D` parameter in the user function signature to access this index.

Advanced and more flexible use of `MapPairs` can be carried out similarly to other SkePU skeletons. For instance, it retains flexibility of `Map` with regards to variadicity (5-way variadic, compared to `Map` being four-way):

- Resulting outputs (see Section 4.2.3),
- Element-wise-V (”vertical”, column-aligned) input arguments,
- Element-wise-H (”horizontal”, row-aligned) input arguments,
- Random-access input arguments,
- Uniform input arguments.

A `MapPairs` instance of higher arity looks like

```
auto pairs = skepu::MapPairs<3, 2>(...);.
```

This instance would accept three vertical and two horizontal input vectors.

4.4 MapOverlap skeleton

`MapOverlap` represents a computational pattern with as many names as there are application domains. It is known as a *convolution* in signal processing, *stencil filter* in image processing, *window function* in statistics, and so on. The SkePU name of `MapOverlap` indicates that it is another variant of the archetypal map pattern, which would typically indicate that there is a degree of parallelism equal to the number of elements in the result

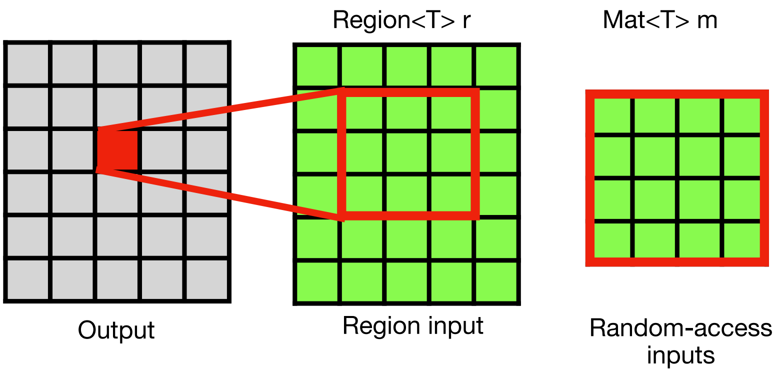


Figure 4.4: Illustrative diagram of the `MapOverlap` skeleton.

container. This is almost true, but not quite: the number of user function invocations—and therefore schedulable tasks—follows this metric, but the ”overlap” part of the name reveals that these tasks are not independent. In a `MapOverlap` user function, not only is a single element-wise mapped element from an input container accessible, so is also a *region* of surrounding elements. The individual regions *overlap* each other, and therefore gives rise to read-after-write dependencies between user function invocations and in general creates a more complex dependency structure between input and output container elements.

In SkePU, the surrounding region is always a *hyper-rectangle*, i.e., a regular multi-dimensional box (in up to four dimensions). The side length of the hyper-rectangle can vary in each dimension, and is defined by a *overlap radius*, which is the number of included elements away from the center element. Therefore, the total amount of elements included in the overlap region is $\prod_i^D(1+2o_i)$, where o_i is the overlap radius for dimension i and D is the number of dimensions of the `MapOverlap` instance as determined from its user function.

A `MapOverlap` example showing a two-dimensional convolution is shown in Listing 4.7.

`MapOverlap` skeleton instances in SkePU can be of several different dimensionality types:

- **1D MapOverlap** applied on
  - `Vector` containers or
  - `Matrix` containers with either *row-wise* or *column-wise* overlap.
- **2D MapOverlap** applied on `Matrix` containers.
- **3D MapOverlap** applied on `Tensor3` containers.

Listing 4.7: Example usage of the `MapOverlap` skeleton.

```
float conv(skepu::Region1D<float> r, const skepu::Vec<float> stencil)
{
  float res = 0;
  for (int i = -r.oi; i <= r.oi; ++i)
      res += r(i) * stencil(i + r.oi);
  return res;
}

skepu::Vector<float> convolution(skepu::Vector<float> &v)
{
  auto convol = skepu::MapOverlap(conv);
  Vector<float> stencil {1, 2, 4, 2, 1};
  Vector<float> result(v.size());
  convol.setOverlap(2);
  return convol(result, v, stencil);
}
```

- **4D MapOverlap** applied on `Tensor4` containers.

Dimensionality of a `MapOverlap` instance is determined by the N in the `RegionND<T>` type used for the element-wise argument in the user function. These are compiler-known types and dictates what variant of the skeleton to use for code generation. Note that the dimensionality of the `MapOverlap` pattern encoded in the skeleton instance does not necessarily match the dimension of the smart data-containers the instance is applied on. In principle, there could be a `MapOverlap` variant for any overlap dimension smaller than or equal to the dimension of the element-wise container input. However, for practical reasons, only the combinations listed above are implemented in SkePU.

Experiences from SkePU users, and in particular the application of SkePU in teaching, has showed that the syntax for `MapOverlap` user functions is one of the more challenging aspects of SkePU. In SkePU 2, the user function acting as a stencil operator was specified with a combination of an explicit pointer parameter and overlap size parameters, and required the user to understand explicit stride addressing of overlap regions.

For SkePU 3 the `MapOverlap` syntax is completely redesigned and simplified. A *container proxy*, `RegionND`, encapsulates the aforementioned parameters, plugging the leaks in the abstraction. Further discussion on this proxy type can be found in Section 5.2.3.

The contemporary syntax for a stencil computation using `MapOverlap` can be seen in Listing 4.7.

4.4.1 Edge handling modes

When `MapOverlap` user functions are evaluated near the edges of the input container, the overlapping region may reach outside the bounds of

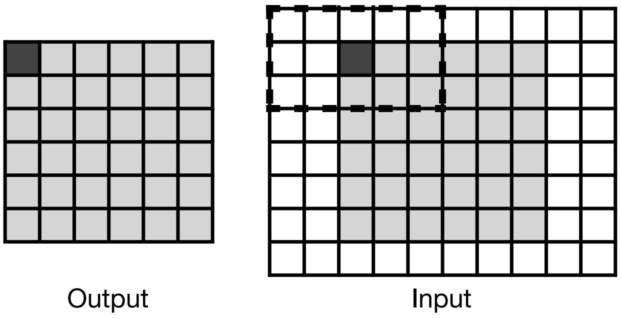


Figure 4.5: Expected input and output container sizes when edge element synthesis disabled, here in 2D `MapOverlap`.

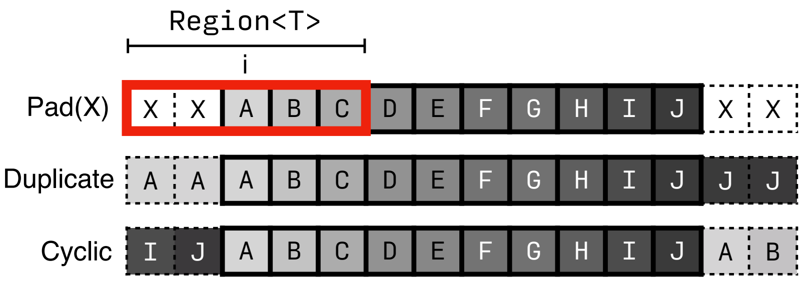

Region<T>
i
Pad(X)
X X A B C D E F G H I J X X
Duplicate
A A A B C D E F G H I J J J
Cyclic
I J A B C D E F G H I J A B

Figure 4.6: Edge handling modes of 1D `MapOverlap`.

the input. The expected behavior of out-of-bounds overlap regions are application-dependent, but to avoid invalid memory accesses, the implementing framework must do something to handle these scenarios. SkePU approaches this problem in several ways. There are a total of four options for edge handling, three of which are proper edge-handling modes:

- no edge handling (default for 2D, 3D, and 4D `MapOverlap`),
- fixed padding with a user-set value,
- duplicate padding of the value closest to the edge (default for 1D `MapOverlap`), or
- cyclic (toroidal) padding.

If the ”no edge handling” option is specified, SkePU requires that the *size* of the input container is larger than the size of the output container, to ensure that all user function evaluations correspond to a well-defined overlap region. Figure 4.5 illustrates this restriction: the overlap radius in this example is 2 in the x-axis and 1 in the y-axis, and the output[8] container

[8]Recall that SkePU always parallelizes skeletons on the output container range.

size is 6×6 elements. The input container is therefore expected to be of size $6 + 2 * 2 = 10$ in the horizontal dimension and $6 + 2 * 1 = 8$ in the vertical dimension.

In all other modes, the output container will be of equal size to the input container, and in cases where the overlap region intersects the container boundaries, SkePU synthesizes virtual elements for out-of-bounds accesses. The properties of each mode is visualized in Figure 4.6.

Synthesis of out-of-bounds elements adds some run-time overhead, but auxiliary memory usage is kept low: proportional to the overlap region size, not to the input data size. Depending on various aspects of the skeleton instance at hand (especially container type), elements in the region may be either pre-allocated or synthesized lazily upon access.

4.4.2 Update modes

For certain use cases of the `MapOverlap` skeleton, specifically iterative workloads with convergency objectives, the way SkePU implements and parallelizes the stencil computation is overly restrictive. SkePU assumes that container operands are either only read from or written to; as a consequence, the implementation can be optimized to minimize the number of copy operations and synchronizations. `MapOverlap` therefore must be operating on two distinct data sets: one input array and one output array (Figure 4.7).

In the application domain of iterative equation solvers, two update schemas are well-known: *Jacobi*, which uses only old values in the update step, and *Gauss-Seidel*, which partially uses new values as well. From the above paragraph, it should be clear that only the Jacobi schema can normally be implemented with `MapOverlap`. Gauss-Seidel operates in wavefronts, which allows the computation to be carried out on just a single copy of the data set: input and output arrays are the same. The drawback of this approach is a greatly reduced degree of parallelism (for an $N \times N$ matrix calculation, Jacobi is $O(N^2)$ parallel operations and Gauss-Seidel enables only $O(N)$.) If successive Gauss-Seidel sweeps are pipelined, however, the steady state forms a bipartition of the container elements. Half the elements, those with *even index parity*, are updated in one iteration; the other half, with odd index parity, are updated in the next. This multi-dimensional checkerboard is referred to as either odd-even or *red-black* update ordering (Figure 4.8) and has recently been implemented in SkePU's `MapOverlap`.

There are different tradeoffs for red-black update mode compared to the default mode. Iterative applications, as shown in Figure 4.9 can reduce the memory footprint by half, and the computation may benefit from the improved convergence rate of the Gauss-Seidel schema. On the other hand, synchronization overhead per average element update is doubled, and the degree of parallelism is reduced to only half of standard `MapOverlap`.

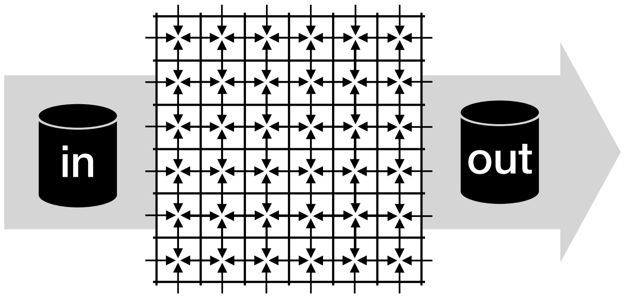


Figure 4.7: MapOverlap normally requires distinct data sets for input and output operands.

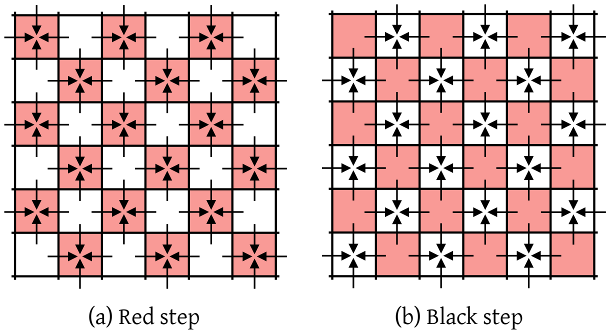
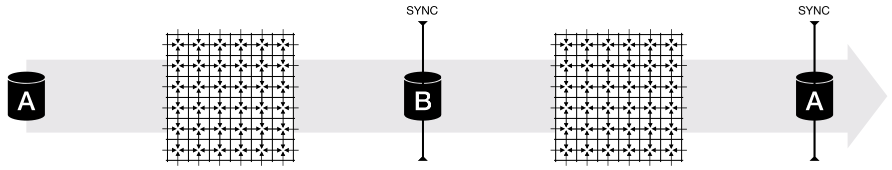


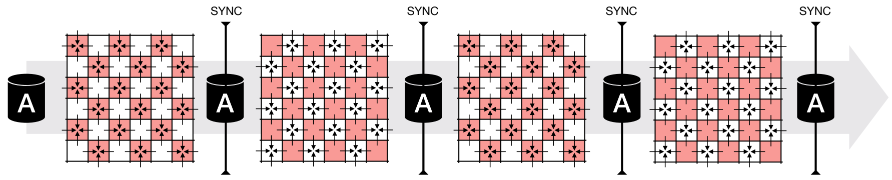
Figure 4.8: Red-black update mode phases.

4.5 Reduce skeleton

Reduce is another well-known pattern in functional programming interfaces. Also known as a *fold*, the main differentiator from the transformation of map is that reduce will turn a collection of values into a single value by the means of a binary reduction operator (here denoted by $\oplus$). Conceptually, this is performed by reducing or folding the value set linearly from the right or left, while carrying a partial result through the chain. This model is typically relaxed by enforcing additional restrictions on the reduction operator, by requiring it to be *associative* and also *commutative*. Associativity of an operator permits an expression to be rearranged with regards to precedence, i.e., order of application of said operator. For instance, the expression $(a \oplus b \oplus c \oplus d)$ can be interpreted as either $(((a \oplus b) \oplus c) \oplus d)$, a left fold; or $((a \oplus b) \oplus (c \oplus d))$, a tree reduction. Which interpretation is the most efficient way to implement the reduction depends on the context: a left fold has excellent spatial locality in its memory access pattern, and is thus highly suitable for efficient sequential processing; while a tree reduc-



(a) Normal update mode



(b) Red-black update mode

Figure 4.9: Two full iterations of `MapOverlap` in different update modes.

tion enables concurrent execution of operators near the leaf nodes of the tree and is therefore suitable for highly parallel scenarios. Commutativity says that the order of the operands themselves can be interchanged: $(a \oplus b)$ is equivalent to $(b \oplus a)$. SkePU can take full advantage of these properties thanks to its multi-backend design, where the same computation will be carried out in different ways depending on the selected backend and other aspects.

Due to the aforementioned restrictions on properties of reduction operators (or rather user functions, in skeleton parlance), the typing limitations on them are quite strict, as the result and both parameters must be of the same single type. This limits what type of reduction computations can be encoded in the `Reduce` skeleton in isolation. It is therefore a common pattern to pre-process a data set before the reduction stage, preferably done by a variant of `Map`. In fact, this sequence of `Map` immediately preceding a `Reduce` is so common that this pattern is available in the separate, fused skeletons `MapReduce` and `MapPairsReduce`. Further discussion on this topic can be found in Section 4.7.

4.5.1 One-dimensional reductions

The basic reduction in SkePU is a linear, one-dimensional reduction over a data set, represented by a smart container. A `Vector` is therefore a natural fit for most reductions, but a `Reduce` skeleton instance will accept smart

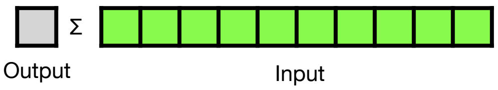


Figure 4.10: Illustrative diagram of the `Reduce` skeleton in 1D mode.

Listing 4.8: Example usage of the `Reduce` skeleton for linear reductions.

```
float min_f(float a, float b)
{
  return (a < b) ? a : b;
}

float min_element(skepu::Vector<float> &v)
{
  auto min_calc = skepu::Reduce(min_f);
  return min_calc(v);
}
```

container arguments of any dimensionality and treat them as linear sets of values.[9]

One common application of reduction can be seen in Listing 4.8, where the computation finds the minimum element in a vector. The syntax is straightforward, but the programmer has to be careful about another aspect of the `Reduce` skeleton: each instance carries with it a *starting value* for reductions, which by default is zero-initialized. It can be set on the instance by supplying the start value in a member function call. In this case the computation is done on `float` elements, so the start value would likely be set to positive infinity, lest the computation would evaluate to `0` if the input data consists of strictly positive values.

4.5.2 Two-dimensional reductions

SkePU also provides two special reduction modes operating on two-dimensional data, i.e., `Matrix`. The first is a reduction in one dimension, along either all rows or all columns. In this scenario the result is passed in a `Vector` output argument, distinguishing this mode from a purely linear reduction of all elements in the matrix.

The second matrix reduction mode instead accepts two user functions at the skeleton instance definition: both satisfying the requirements for reduction operators. The first is being used for reduction in one dimension, just like in the previous paragraph, and the second then reduces the partial results from the first phase, with a scalar result remaining.

[9]The linear interpretation follows the memory layout order presented in Section 5.1, but the commutativity constraint implies that the reduction semantics allows any element order.

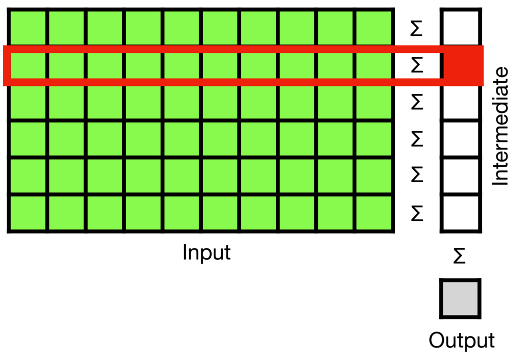


Figure 4.11: Illustrative diagram of the access pattern in two-dimensional Reduce.

Listing 4.9: Example usage of the Reduce skeleton for 2D reductions.

```
float plus_f(float a, float b)
{
  return a + b;
}

float max_f(float a, float b)
{
  return (a > b) ? a : b;
}

float min_element(skepu::Matrix<float> &v)
{
  auto max_sum = skepu::Reduce(plus_f, max_f);
  sum.setReduceMode(skepu::ReduceMode::RowWise);
  return max_sum(v);
}
```

The initial reduction can be either row-wise or column-wise. A setting on the skeleton instance object controls which dimension comes first.

4.6 Scan skeleton

Prefix sums are fundamental building blocks in many parallel algorithms [91]. The generalized pattern is applicable to a wide variety of problems far beyond the pure functional data processing in SkePU, such as pointer manipulation in list ranking algorithms. That said, the generalized prefix sums pattern, like reductions, can be realized in many different ways, each with their strengths and weaknesses, suitable for different execution targets. So their relevancy also applies to skeleton programming interfaces, and SkePU provides a generalized prefix sum pattern with the Scan skeleton. The se-

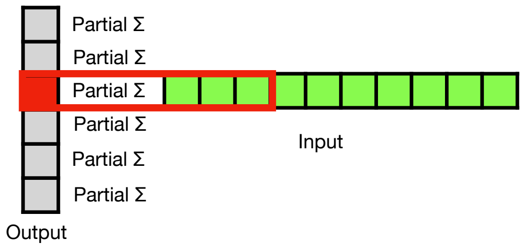


Figure 4.12: Illustrative diagram of the `Scan` skeleton.

Listing 4.10: Example usage of the `Scan` skeleton.

```
float max_f(float a, float b)
{
  return (a > b) ? a : b;
}

skepu::Vector<float> partial_max(skepu::Vector<float> &v)
{
  auto premax = skepu::Scan(max_f);
  skepu::Vector<float> result(v.size());
  return premax(result, v);
}
```

mantics are similar to reduce, with scan producing the equivalent result of computing a reduction on all *subsequences* of the data set, starting with the first element. These partial sums (or the generalized operator) are returned in a linearized container. Whether the element of index *i* is included in the prefix sum result at index *i* in the result container or not is controlled by a parameter on the `Scan` skeleton instance. These variants are known as *inclusive* and *exclusive* prefix sums in the literature.

There is no "MapScan" in SkePU to mirror `MapReduce`, but chaining separate `Map*` and `Scan` skeleton invocations still utilizes the memory management and data movement optimizations built-in to the smart data-containers (see Section 5.1).

4.7 MapReduce skeleton

`MapReduce` is also the solution to the problem presented in Section 4.5 about the requirement of `Reduce` to only accept associative reduction operators. As an example, consider a reduction with the goal to find both the maximum value and the minimum value in a data set. This can be done in a straightforward way with two `Reduce` instances, but for the sake of discus-

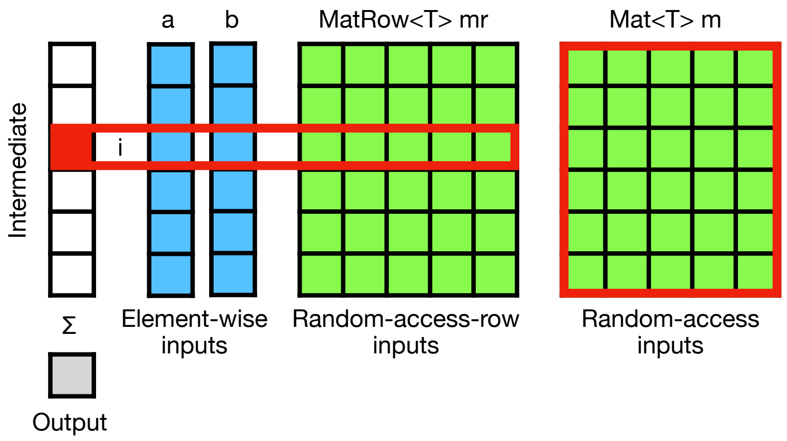


Figure 4.13: Illustrative diagram of the `MapReduce` skeleton.

Listing 4.11: Example usage of the `MapReduce` skeleton.

```
float add(float a, float b)
{
  return a + b;
}

float mult(float a, float b)
{
  return a * b;
}

float dot_product(skepu::Vector<float> &v1, skepu::Vector<float> &v2)
{
  auto dotprod = skepu::MapReduce(mult, add);
  return dotprod(v1, v2);
}
```

sion we want to do with only one skeleton instance.[10] This operator would be non-associative, since it takes scalar values as input and somehow returns a tuple of both a maximum and a minimum partial result. If SkePU allowed non-associative reduction operators, we could encode this as a left fold, with the left hand side operand being the running result and the right hand side being the next value from the data set.

Working within the constraints of SkePU, the solution is given in Listing 4.12. `MapReduce` is used to `preprocess` the initial data set into the `MaxMin` custom data type encoding the reduction results, which allows the reduction part (`max_min_f`) to only work on values of this type. There is no

[10]It might help performance to only do one pass through the data set due to cache effects, so the example is not as arbitrary as it may seem.

Listing 4.12: Using `MapReduce` to compute non-associative reductions.

```cpp
struct MaxMin
{
  float max;
  float min;
};

MaxMin preprocess(float val)
{
  MaxMin res;
  res.max = val;
  res.min = val;
  return res;
}

MaxMin max_min_f(MaxMin a, MaxMin b)
{
  MaxMin res;
  res.max = (a.max > b.max) ? a.max : b.max;
  res.min = (a.min < b.min) ? a.min : b.min;
  return res;
}

void find_max_min(skepu::Vector<float> floats)
{
  auto maxmin = skepu::MapReduce<1>(preprocess, max_min_f);
  maxmin.setStartValue({-INFINITY, INFINITY});
  MaxMin result = maxmin(floats);

  std::cout << "Max: " << result.max << "\n";
  std::cout << "Min: " << result.min << "\n";
}
```

computation here, only a translation of the data format. Because of the efficient fusion of the two phases as SkePU evaluates the skeleton application, the overhead of this transformation is minimal.

4.8 MapPairsReduce skeleton

`MapPairsReduce` is the combination of a `MapPairs` followed by a row-wise or column-wise reduction over the generated matrix elements. Like `MapPairs` it supports arbitrary arities of the vertical and horizontal input `Vector` groups (`<0,0>` and up). It returns a `Vector` containing the row-wise or column-wise reduction, where the reduction dimension is specified as in 2D `Reduce`. Example usage of this skeleton can be seen in Listing 4.13 and Listing D.2; a conceptual illustration is shown in Figure 4.14.

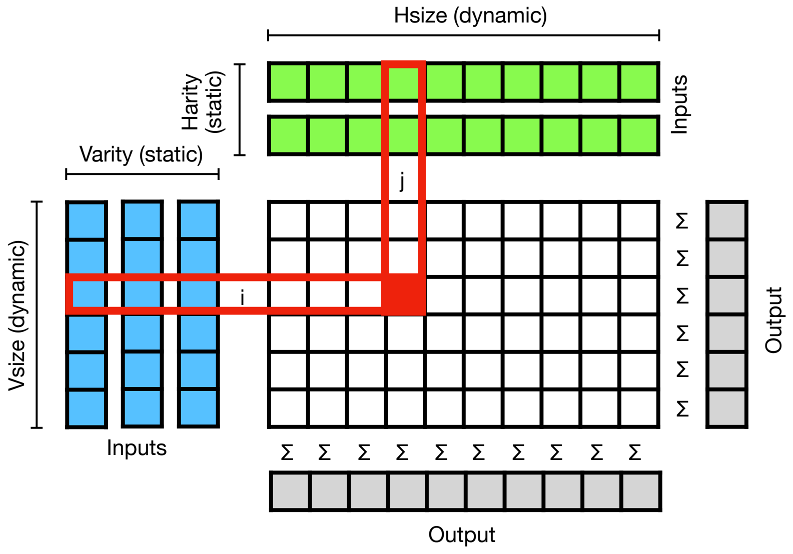


Figure 4.14: Illustrative diagram of the `MapPairsReduce` skeleton.

Listing 4.13: Example usage of the `MapPairsReduce` skeleton.

```cpp
int mul(int a, int b)
{
  return a * b;
}

int sum(int a, int b)
{
  return a + b;
}

void mappairsreduce(size_t Vsize, size_t Hsize)
{
  auto mpr = skepu::MapPairsReduce(mul, sum);

  skepu::Vector<int> v1(Vsize), h1(Hsize);
  skepu::Vector<int> res(Hsize);

  mpr.setReduceMode(skepu::ReduceMode::ColWise);
  mpr(res, v1, h1);
}
```

4.9 Call skeleton

Not all applications have a computational structure that is straightforwardly reformulated as skeleton patterns. This is especially true for the particular skeleton set offered by SkePU, which has a strong focus on data-parallel patterns. At the time of the SkePU 2 interface redesign, the skeletons (especially the core Map building block) was generalized to handle more complex memory access patterns inside of user functions themselves (as discussed in depth in earlier sections, such as 4.2.1). There are advantages of placing chunks of application code inside the user functions like this, as the code is then able to access specialized computing resources (e.g., of external accelerators) while the smart data-containers handle memory transfer and coherency management automatically. The clear downside is that a user function is a sequential block of code: parallelism in SkePU patterns is due to concurrent evaluation of several user function invocations.

To close this gap, the experimental Call skeleton was included in SkePU 2. Call semantics is like that of Map, without the data parallelism. It can therefore be regarded as a *pseudo-pattern*, and is more closely described as a *multi-variant task* (or component). Using escape mechanisms in the form of preprocessor directives, explicit parallelism can be inserted into user function code. The same computation then has to be explicitly provided for all desired backends. Listing 4.14 contains a sorting task using Call in this way.

The Call skeleton has been in part superseded and in part complemented by the work presented in Chapter 12 on multi-variant user functions.

4.10 User functions

User functions are a central component in SkePU programming. In the conceptual definition, and also at their most basic practical application, user functions are the operators which instantiate skeletons.

While most user functions are short, perhaps even single-expression computations—such as the scale function given in Listing 4.15—they are expressed as general C++ functions, and can contain comparatively complex code structures. User function code can have local state variables (allocated on the stack), have conditional branches, nested loop structures iterating over large data sets, and so on. However, as with any high-level parallel or heterogeneous programming interface embedded within C++, there are significant limitations on what type of operations are allowed within the user function scope. The reasons are the same that necessitates *CUDA* to differentiate between `__host__` and `__device__` functions, and *C++ AMP* to introduce a `restrict` keyword, to mention only two such instances. Other interfaces approach the same problem by having clear separation between the sequential (”host”) code and parallel or heterogeneous (”device”) code, such as *OpenCL* with external kernels, or constraining entire applications to a DSL, as done in *Musket* and others. The decision in SkePU to use a single-source model is motivated by cohesive and readable programs, as user functions can be very small and seamlessly interspersed throughout the appli-

Listing 4.14: Example usage of the Call skeleton.

```
void sort_f(skepu::Vec<int> array, size_t nn)
{
#if SKEPU_USING_BACKEND_CL

  size_t idx = get_global_id(0);
  size_t l = nn / 2 + ((nn % 2 != 0) ? 1 : 0);

  for (size_t i = 0; i < l; ++i)
  {
    if (idx % 2 == 0 && idx < nn - 1 && array(idx) > array(idx + 1))
      swap_f(&array(idx), &array(idx + 1));
    barrier(CLK_GLOBAL_MEM_FENCE);

    if (idx % 2 == 1 && idx < nn - 1 && array(idx) > array(idx + 1))
      swap_f(&array(idx), &array(idx + 1));
    barrier(CLK_LOCAL_MEM_FENCE);
  }

#else

  for (size_t c = 1; c <= nn - 1; c++)
    for (size_t d = c; d > 0 && array(d) < array(d-1); --d)
      swap_f(&array(d), &array(d - 1]);

#endif
}

void sort(skepu::Vector<int> &v, skepu::BackendSpec spec)
{
  auto sort = skepu::Call(sort_f);

  spec.setGPUBlocks(1);
  spec.setGPUThreads(v.size());
  sort.setBackend(spec);

  sort(v, v.size());
}
```

Listing 4.15: A basic user function and associated skeleton instance..

```
int scale(int e)
{
  return e * 2;
}

auto vectorscale = skepu::Map(scale);

scale(result, input);
```

cation with minimal syntactical overhead. Tight C++-integration in SkePU allows for an intuitive and recognizable syntax and reduced friction when integrating SkePU skeletons into larger C++ applications.

Due to the inherent limitations discussed in the previous paragraph, SkePU user functions come with restrictions. Conceptually, the goal of parallel execution requires the user functions to be *pure functions*: their computations are deterministic given a set of arguments, and they cannot have side effects. Communication across user function invocations are thus not allowed, nor is dynamic memory allocation as it requires accessing a shared memory heap. Targeting systems with heterogeneity or otherwise distributed memory spaces implies that there by necessity has to be a memory barrier between the unmanaged and managed scopes (see Section 5.3, in particular Figure 5.5). Smart data-containers are made to bridge this gap with as little friction as possible for the programmer, but other data sets and arbitrary pointers cannot be used at all from within user function code. Furthermore, syntactical limitations in certain backend targets preclude usage of many C++ features, such as operator overloading or range-for loops. This restriction, unlike the previous ones listed, are not inherent to the parallel programming domain but limitations in the SkePU pre-compiler, and the set of allowed syntactical constructs grow as SkePU matures. SkePU user functions are best approached as using a C-style subset of C++, unless exceptions are explicitly mentioned.

A skeleton instance always needs a user function to be instantiated (possibly more than one, as with `MapReduce`). The reverse is not true: functions do not need to be mentioned within a skeleton construction for SkePU to treat them as user functions and make them subject to backend code generation. The chance of a function being called within the dynamic scope of another skeleton-instantiating user function is enough. In most aspects, these ”indirect” user functions are subject to the same restrictions. There is an important distinction, however: a skeleton cannot be instantiated with a function with parameters of pointer type, as this represents bridging the gap between unmanaged and managed scope (and thus possibly different memory address spaces), but indirect user functions can accept pointer arguments. As illustrated in Figure 4.15, recursion, either direct or indirect, is not allowed within managed scope.

4.10.1 User functions as lambda expressions

In 4.15, `scale` is a user function defined as a free function. This is one of two ways to define user functions in SkePU; the other is with lambda expression syntax as in Listing 4.16, where the function is written inline with the skeleton instance. Free functions are suitable for cases where a user function is large and an inline definition distracts from the pattern-program flow, or when user functions can be shared across skeleton instances. In most

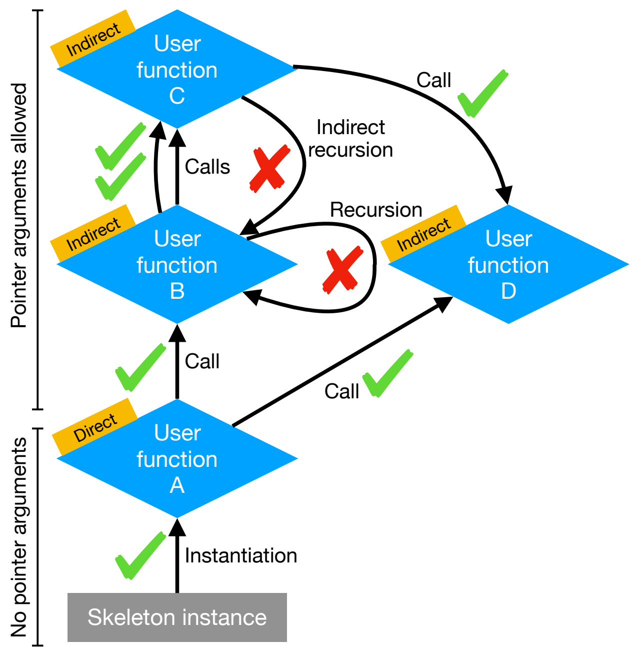


Figure 4.15: User function call graph.

Listing 4.16: Skeleton instance with lambda syntax for the user function.

```
auto vsum = Map<2>([](float a, float b) { return a + b; });
```

cases, however, the lambda syntax is superior: it increases code locality while eliminating namespace pollution. There are no run-time differences between the two, as identical code is generated by the pre-compiler.

4.11 User types

For many applications, basic types such as `int` and `float` may not be sufficient in a high-level programming interface. SkePU therefore includes the possibility of using a custom `struct` as the element type in smart data-containers or used as extra argument to a skeleton instance. Even then, there are major restrictions on such types depending on the backends used; the type should not have any features outside those of a C-style `struct` and the memory layout needs to match across backends.

Listing 4.17 demonstrates user types in SkePU with the use of a complex number type `cplx` for Mandelbrot fractal generation. Functions operating

Listing 4.17: Mandelbrot fractal generation in SkePU.

```
[[skepu::userconstant]] constexpr float
  CENTER_X = -.5f,
  CENTER_Y = 0.f,
  SCALE = 2.5f;

[[skepu::userconstant]] constexpr size_t
  MAX_ITERS = 1000;

struct cplx
{
  float a, b;
};

cplx mult_c(cplx lhs, cplx rhs)
{
  cplx r;
  r.a = lhs.a * rhs.a - lhs.b * rhs.b;
  r.b = lhs.b * rhs.a + lhs.a * rhs.b;
  return r;
}

cplx add_c(cplx lhs, cplx rhs)
{
  cplx r;
  r.a = lhs.a + rhs.a;
  r.b = lhs.b + rhs.b;
  return r;
}

size_t mandelbrot_f(skepu::Index2D index, size_t height, size_t width)
{
  cplx a;
  a.a = SCALE / height * (index.col - width/2.f) + CENTER_X;
  a.b = SCALE / height * (index.row - width/2.f) + CENTER_Y;
  cplx c = a;

  for (size_t i = 0; i < MAX_ITERS; ++i)
  {
    a = add_c(mult_c(a, a), c);
    if ((a.a * a.a + a.b * a.b) > 4)
      return i;
  }
  return MAX_ITERS;
}

auto mandelbrot = skepu::Map<0>(mandelbrot_f);
```

on objects of type `cplx` are defined as free functions and are treated as user functions by the pre-compiler.

4.12 User constants

SkePU does not allow C-style macro constants in user function code. This is mainly a side-effect from the way source-to-source compilation is implemented through the Clang tools (see Chapter 7) but fits with the general aim of SkePU to move away from macros and instead rely on type-safety through the C++ type system.

Uniform user function parameters (Section 4.2.2) can be employed as a substitute for macros, but their purpose is typically aimed at scenarios where the value of the argument changes dynamically between skeleton invocations. Therefore, SkePU provides *user constants* to more directly address the need for global, static parameters in user function code. These objects are basic C++ types with literal assignment, thus making them more type-safe than macros, annotated with the `[[skepu::userconstant]]` attribute. Usage of user constants can be seen in the N-body simulation in Listing D.2. The example in Listing 4.17 also uses them, e.g., for `MAX_ITERS`.

4.13 Strided skeletons

Recently, the SkePU skeletons have been extended with a prototype API for *strided* access into the smart data-containers. The stride configuration is set as a persistent property on the skeleton instance object, and results in a sparse addressing of containers upon skeleton invocation. Smart data-containers remain contiguous blocks of data, and stride access effectively causes only a subset of container elements to be read or written. Unlike iterators, which also only address a subset of elements, strided subsets are regular interleavings of indices. Stride configuration and iterators may be combined, but at the time of writing all combinations of skeletons, stride lengths, and backends are not yet implemented in the public SkePU distribution.

4.13.1 Strides `Map`, `MapPairs`, and their reduce variants

Strided access of smart data-containers in skeleton evaluations have use in certain linear algebra applications. For instance, consider a SkePU container initialized with the elements of a matrix in (for the sake of illustration) column-major order. The user may want to compute the dot product of a single column in the matrix and another vector. This can be achieved with a standard `MapReduce` call if non-unit stride length is configured for the parameter corresponding to the packed matrix argument, specifically with the stride length set to the row width of the matrix.

Listing 4.18: SkePU Map call using strided container access.

```
int f(int a, int b) { /* ... */ };

auto mapper = skepu::Map(f);
mapper.setStride(2, 4, 3);

skepu::Vector<int> out(16), in_a(N_A), in_b(N_B);

mapper(out, in_a, in_b); // out stride = 2, in_a stride = 4, in_b stride = 3
```

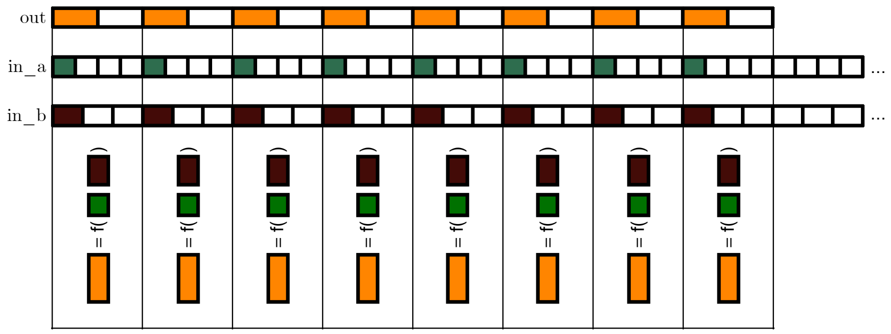


Figure 4.16: Illustration of element access patterns in the program from Listing 4.18.

The stride (step) length semantics in SkePU is borrowed from BLAS. The first element of the container is always accessed, and the rest are decided by stride increments, as illustrated in Figure 4.16. A negative stride length reverses the order of elements.

As SkePU skeletons are variadic, strides for all element-wise arguments must be provided. For `Map`, `MapPairs`, and `MapPairsReduce`, the stride for the output container is declared first in the stride length list, corresponding to container arguments in a skeleton invocation. Refer to Listing 4.18 for a code sample corresponding to Figure 4.16.

4.13.2 Strides in `MapOverlap`

The `MapOverlap` skeleton is not variadic. Due to this, in combination with future plans regarding extensions of the `MapOverlap` syntax and capabilities, the stride semantics is defined differently. The stride length list instead corresponds to step increments in the respective dimension: 1D `MapOverlap` expects one stride length, 2D `MapOverlap` expects two, and so on.

5 Data representation with smart data-containers

This chapter introduces the data representation model used in SkePU, including the concept of smart data-containers and the memory consistency model presented by the SkePU programming model.

5.1 Smart data-containers

The availability of smart data-containers for data abstraction and memory management in SkePU, previously restricted to vector and matrix types, has a significant effect on the usability of a skeleton programming framework. Even though a basic one-dimensional data set can be used to emulate more complex data representations, doing so at a framework level rather than on the user level provides more information to the implementation about access patterns, thus bringing increasing opportunities for optimizing communication- and memory access patterns; while also providing a more intuitive user interface and reduced application code size for users.

SkePU's *smart data-containers* are precompiler-known run-time data structures which reside in main memory, but can temporarily store subsets of their elements in device memory for access by skeleton backends executing on these devices. Smart data-containers additionally perform transparent software caching of the operand elements that they wrap, with a MSI-based coherence protocol [47]. Hence, smart data-containers automatically keep track of valid copies of their element data and their locations, and can, at run-time, automatically optimize communication and device memory al-

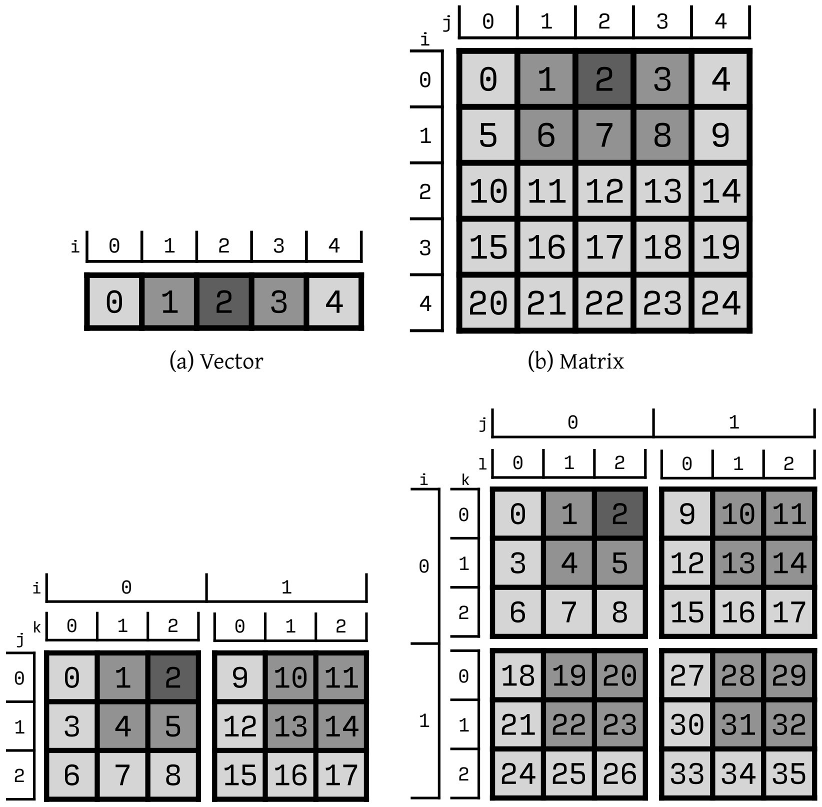

(a) Vector (b) Matrix (c) Tensor3 (d) Tensor4

Figure 5.1: Container indexing and memory layout.

location. Smart data-containers can lead to a significant performance gain over ”non-smart” data-containers, especially for iterative computations on sufficiently large data, where data can stay on the accelerator devices or remain partitioned across cluster nodes.

The SkePU container set is recently [59] extended with *tensors*, which are higher-dimensionality data-containers, completing the picture in Figure 5.1. In SkePU 3 there are tensors of three (`Tensor3<T>`) and four (`Tensor4<T>`) dimensions, complementing the existing one-dimensional `Vector<T>` and two-dimensional `Matrix<T>`. Smart container dimensionality in SkePU is therefore fixed by the framework, though their sizes in each dimension are user-defined. While the template metaprogramming technologies used elsewhere in SkePU *can* be used to implement container types of arbitrary dimension, also providing each skeleton pattern for customizable dimensionality (esp. `MapOverlap`) is currently outside the scope

Listing 5.1: Smart container set in SkePU 3.

```
skepu::Vector<float> v(dim1);
skepu::Matrix<float> m(dim1, dim2);
skepu::Tensor3<float> t3(dim1, dim2, dim3);
skepu::Tensor4<float> t4(dim1, dim2, dim3, dim4);
```

of SkePU, and as such the container set is restrained to cover up to four dimensions.

The interfaces for tensor containers are virtually identical to those of vectors and matrices, differing in the obvious ways of naming and element access as detailed below. Instances of the tensor classes are created with one constructor argument for each dimension. Optionally an additional argument of type `T` specifies the default value of all elements in the container. The full set of smart data-containers in SkePU 3 now covers up to four-dimensional structures; see Listing 5.1 for their definitions.

The set of `Index` object types in SkePU, usable in e.g. user function signatures to identify the index of the element being operated on, is likewise extended with 3D and 4D equivalents (Listing 4.5).

Tensors are available in the skeleton API as element-wise inputs to `Map`, `Reduce`, `MapReduce`, `Scan`, and `MapOverlap`. They are also accessible freely in user functions as proxy objects, where applicable. In some skeleton configurations the dimensionality of element-wise inputs is irrelevant by design, though in Map-based skeletons it can be accessed by using `Index` parameters.

5.1.1 Container indexing

Even though SkePU aims for a high-level programming interface and its data-containers are strongly evoking mathematical terminology, it features zero-based indexing. SkePU is C++-based and it is to be expected that programmers with existing C++ experience are used to this mode of indexing, that arguably exposes implementation details of the containers being represented as memory arrays. Care has to be taken when porting applications from languages popular in the scientific communities that feature one-based indexing, such as *Fortran* and *MATLAB*.

When indexing smart container objects, regardless of dimensionality, the first index is always the most significant, that is, changing this index will cause the biggest jump in the memory offset. This index is typically named `i` with the subsequent indices increasing alphabetically: i, j, k, l. These labels are exposed in the interface of the index types discussed in Section 4.2.4. Figure 5.1 illustrates the memory layout by numbering each element in the containers, and how it relates to each index coordinate.

Formally, the access syntax is

```
container(i,[j, [k, [l]]]) [= value];.
```

Indexing semantics are slightly different in `Region` container proxy types, with the zero index denoting the center element in the region and accepting negative indices. See Section 5.2.3 for more details.

Figure 5.1 illustrates indexing and memory layout of the four smart container types. Figure 5.1a shows a `Vector` of size 5, Figure 5.1b shows a `Matrix` of size 5×5, Figure 5.1c shows a `Tensor3` of size $2 \times 3 \times 3$, and Figure 5.1d shows a `Tensor4` of size $2 \times 2 \times 3 \times 3$.

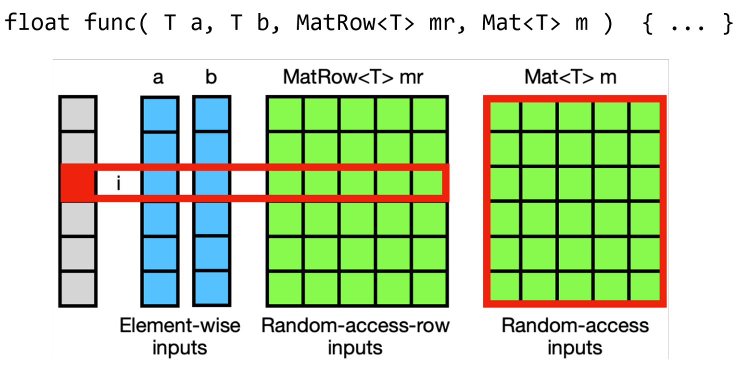
5.2 Container proxies

Smart data-containers typically reside in the *unmanaged scope* of a SkePU program, outside of user functions. They are complex C++ template types and manage coherency states across backends, which makes them, in general, impossible to directly be accessible on the backends themselves. For the basic skeleton formulations with element-wise operand mappings, there is never a need to interface with the container objects themselves in user function code. With random-access parameters as described in Section 4.2.1, this is no longer true. In SkePU 1, this problem was solved with a specific skeleton (`MapArray`) with a pointer parameter in the user function as the way to interface with the full extent of the container data. In SkePU 2 and later, a more general and less leaky abstraction is instead available in most skeletons: the proxy container objects. Expressed in code as `Vec<T>`, `Mat<T>`, `Ten3<T>`, or `Ten4<T>`; these objects provide clear and type-safe access to an entire container's data. Indexing is done just like in unmanaged scope (Section 5.1.1) and the proxy objects provide member fields with container size for each dimension. Otherwise, the proxies have no features, as they are kept lightweight for preserving performance.

In addition to whole-container proxy objects, there are three (or six) instances of proxies for partial container access: `MatRow<T>` and `MatCol<T>` for matrices, and `RegionND<T>` representing the *neighborhood* around a specific element for each of the four dimensions of smart data-containers. Each such partial proxy object is covered further in the upcoming sections.

5.2.1 MatRow proxy

SkePU has since version 2 allowed for flexible parameter lists for user functions, including *random-access* containers (implemented in terms of lightweight *proxy objects*) in addition to the default element-wise inputs. While this allows for powerful expressivity, very little about the access patterns of these random-access containers is known to SkePU, and performance may thus not always be ideal.



```
float func( T a, T b, MatRow<T> mr, Mat<T> m )  { ... }
```

Figure 5.2: Element accessibility for `MatRow` vs. `Mat` parameters in a user function.

Listing 5.2: Matrix-vector multiply using `MatRow` in SkePU 3.

```
template<typename T>
T mvmult_f(const skepu::MatRow<T> mr, const skepu::Vec<T> v)
{
  T res = 0;
  for (size_t i = 0; i < v.size; ++i)
    res += mr(i) * v(i);
  return res;
}

skepu::Vector<float> y(height), x(width);
skepu::Matrix<float> A(height, width);
auto mvmult = skepu::Map<0,0>(mvmult_f);
mvmult(y, A, x);
```

One very common pattern when using `Matrix` as a random-access container parameter is that each user function invocation is only interested in one row of the matrix. This pattern is seen in matrix-vector multiplication and similar multi-reduction-style computations. To improve SkePU performance in these cases, SkePU 3 introduces a new proxy object, `MatRow<T>`. Bridging the gap between element-wise mapped and random-access container arguments, this proxy type when used in a `Map` skeleton instance that maps over vectors (i.e., the result container(s) of the skeleton are `Vector`), makes available one single row of the argument matrix container to the user function, see Figure 5.2.

As an example, matrix-vector multiplication using `MatRow<T>` may be implemented as shown in Listing 4.2. Compared to the closest corresponding SkePU 2 implementation which only provides the more generic `Mat` proxy container, the code is more succinct and there is more information about the access pattern available to SkePU.

There is no change in syntax of skeleton instantiation or skeleton invocation needed for this feature to apply.

The performance benefit of using `MatRow` (where applicable) instead of the more general `Mat` container proxy comes from significantly reduced operand data transfer volume when executing over distributed memory scenarios, both in multi-GPU execution and in cluster execution: the communication pattern with `MatRow` is a scatter operation, while with `Mat` it is a broadcast.

5.2.2 MatCol proxy

Analogous to `MatRow`, SkePU 3 provides a proxy container encoding column accesses to random-access matrix containers in `MatCol`. In most respects `MatCol` behaves just like its sibling, with two major differences. Firstly, SkePU matrices are stored in *row-major order* in memory, and providing a slice or view into a single column of a matrix is therefore not as straightforward. Here SkePU again utilizes its strengths as a high-level multi-backend framework: by using `MatCol`, the programmer declares their *intent* of only accessing elements from a single column, but not the imperative instructions of how this will be done. For a shared memory system and a sufficiently small matrix, SkePU may choose to provide direct, strided access to the underlying container object. On distributed memory, such as multi-GPU or cluster systems, SkePU will create a transposed clone of the matrix container (alternatively viewed as now being stored in column-major order) and divide it among memory subspaces. Even in the shared memory case, SkePU may use information from the application state (such as container size, lineage structures, and tuning data) and decide that transposing is worthwhile.

Secondly, the semantics of *which* column is selected for each invocation of the user function has to be defined. SkePU abides to the following rules:

1. If the result container is a vector, let the current element index be i:
    a) `MatRow` binds to the ith row of the corresponding random-access matrix argument.
    b) `MatCol` binds to the ith column of the corresponding random-access matrix argument.
2. Otherwise, if the result container is a matrix, let the current element index be (i, j):
    a) `MatRow` binds to the ith row of the corresponding random-access matrix argument.
    b) `MatCol` binds to the jth column of the corresponding random-access matrix argument.

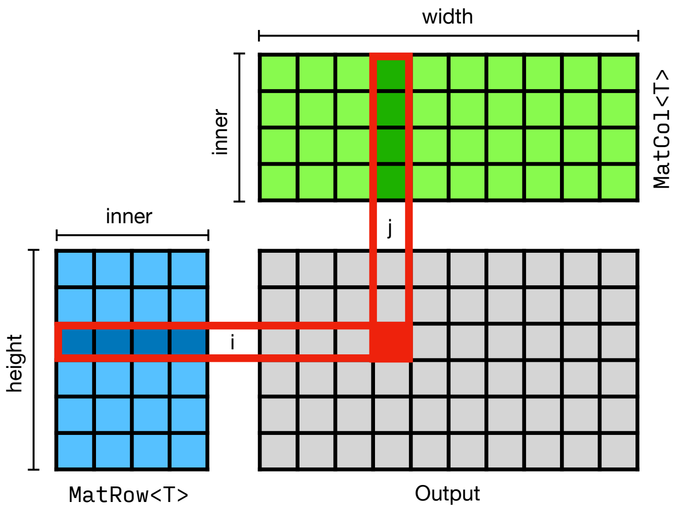


Figure 5.3: `MatRow` and `MatCol` access patterns in the computation in Listing 5.3.

3. Otherwise, the skeleton instance is malformed.

Listing 5.3: Matrix-matrix multiply with `MatRow` and `MatCol`.

```cpp
template<typename T>
T mmmult_f(skepu::MatRow<T> ar, skepu::MatCol<T> bc)
{
  T res = 0;
  for (size_t k = 0; k < ar.cols; ++k)
    res += ar(k) * bc(k);
  return res;
}

skepu::Matrix<float> a(height, inner), b(inner, width), c(height, width);
auto mmmult = skepu::MapPairs<0,0>(mmmult_f<float>);
mmmult(c, a, b);
```

For computations on matrices, these rules are natural and analogous to the element-wise indexing in the `MapPairs` and `MapPairsReduce` skeletons. In fact, `MatRow` and `MatCol` are a perfect fit together with `MapPairs`, extending the skeleton to handle computations with the structure of *matrix-matrix multiplications*. Listing 5.3 shows how such a computation may look.

Matrix-row and matrix-column user function proxy containers are available in user functions for `Map`, `MapReduce`, and `MapOverlap` skeleton instances that satisfy the above requirements.

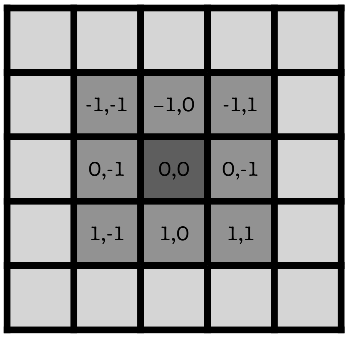


Figure 5.4: Indexing into a `Region2D` matrix proxy with overlap size $(1, 1)$.

5.2.3 Region proxy

Specifically for the `MapOverlap` skeleton, the element-wise iterated input argument container provides access not only to the current element, but rather a *region* of elements surrounding the current index. To achieve a high-level interface for this pattern, SkePU provides the `RegionND<T>` family of types, with N ranging from 1 to 4.

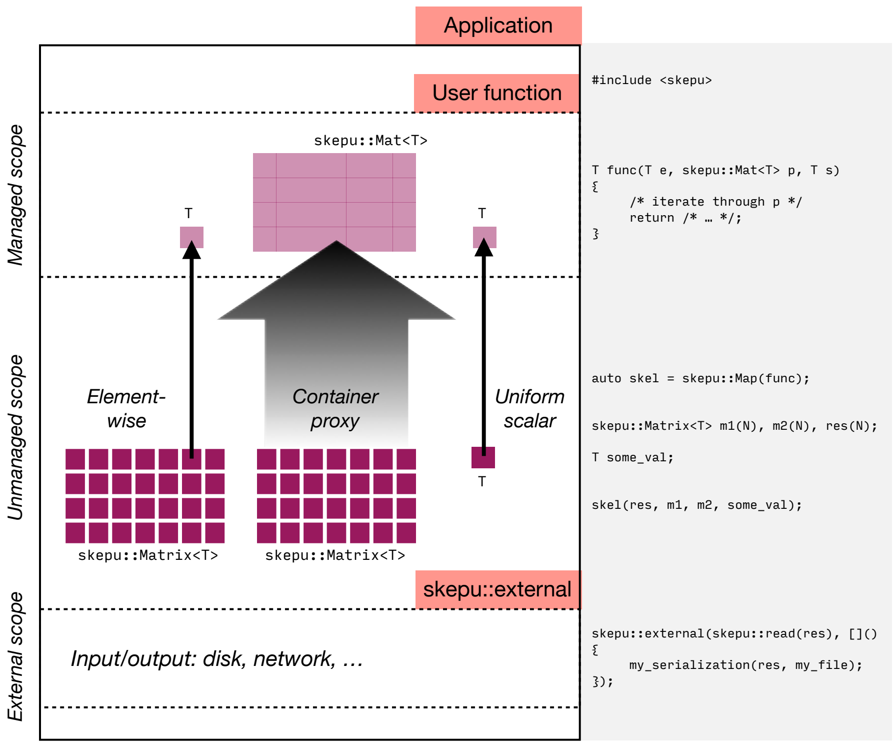
An object of the `RegionND<T>` types can be indexed to access values in the region, and also carries information about the size of the overlap region (which can be set dynamically before a skeleton invocation). See Listing 4.7 for an example of region objects used in a convolution computation.

In Figure 5.1, for each container, the third element (indexed 2 in the least significant index and 0 elsewhere) is darkly shaded, and the surrounding region with a radius of 1 in each dimension is medium shaded. The neighboring elements are important for the `MapOverlap` skeleton. Note that the region is significantly truncated for all dimensions larger than 1; a full four-dimensional radius-1 neighborhood would be $3^4 = 81$ elements in total, including the center element.

Indexing into region objects is zero-based with the *center* element at the **0** position. Positive and negative indices are used in each dimension to access the full region, see Figure 5.4.

5.3 Memory consistency model

Experiences from users of SkePU 2 demonstrated that the dual-mode model of SkePU can be a bit challenging to adapt to. As with, for instance, GPU programming models, SkePU programs execute code in one of two modes, or rather scopes: *unmanaged scope* or *managed scope*. In GPU programming parlance (exposed directly in the CUDA interface) these are known as ”host” and ”kernel” mode. In SkePU, these are represented by being either outside



```
#include <skepu>


T func(T e, skepu::Mat<T> p, T s)
{
    /* iterate through p */
    return /* … */;
}


auto skel = skepu::Map(func);

skepu::Matrix<T> m1(N), m2(N), res(N);

T some_val;

skel(res, m1, m2, some_val);


skepu::external(skepu::read(res), []()
{
    my_serialization(res, my_file);
});
```

Figure 5.5: Scopes with differing capabilties in a SkePU program.

or inside of the *dynamic scope* of a skeleton user function. While syntactically highly similar, the capabilities in each mode are very different. Code residing in unmanaged scope is treated effectively like any C++ environment, as it is the goal of the framework to be possible to embed in existing C++ applications. This means that the programmer can use any C++ constructs and idioms such as classes, dynamically allocated structures, virtual function calls, and so on. Inside a user function, however, the environment is effectively a single-threaded, no-side-effects, C-like land.[1]

These differences also mean that the memory consistency models are different in the two views. SkePU handles memory consistency at the boundary—during entry and exit of a skeleton invocation and the user function evaluation. Inside the user function, side effects are not allowed and therefore random memory reads are disabled, and the coherency model is straightforward.

SkePU 2 separated container accesses in unmanaged scope into two kinds: `[]` array notation and `()` functional notation. Array notation main-

[1]The reason for this is to preserve compatibility with as many accelerator environments as possible, such as OpenCL C or even FPGAs.

tained consistency while functional notation bypassed any checks and enabled direct reads and writes on the internal, host-side memory array. The bracket operator checked for the accessed element's state in the data container's metadata (updated or invalid) and, if necessary, would trigger a (bulk) data movement to update the container's copy in host memory from a currently valid device copy.

During the design of SkePU 3, experiences gained from field observations of SkePU 2 made it clear that this model was confusing, as the checked array notation incurred overhead when used in tight loops, and spurious usage of functional notation could lead to unintended errors. Thus, SkePU 3 removes the array-style angle bracket notation completely, and functional-style element access is not consistency-checked unless explicitly requested by a compile-time command. Functional notation is chosen as it scales naturally to the multi-dimensional container types, matrices and tensors. (Element indexing of smart data-containers is covered in more detail in Section 5.1.1.)

Instead, the programmer should *flush* the whole container instead before doing single-element accesses of user function data, as described below.

Hence, there is no longer a coherency-satisfying single-element access mechanism to SkePU smart data-containers except inside user function proxy objects (`Vec<T>`, `Mat<T>`, etc). However, optional runtime checks outside user functions can be (re-)activated for parenthesis accesses by setting a compiler flag, e.g., for debugging purposes or for backwards compatibility with code written for SkePU 2.

A common pattern in SkePU applications is that smart data-containers are used for a computationally intensive part of the application, and the data is then either handed over to a non-SkePUized section, or serialized e.g. to disk. To accommodate this pattern, it is important that there is a way to ensure consistency of the local container contents. SkePU 3 provides this through the `flush` operation to complete the new consistency model.

Flushing smart container data can be performed on smart container instances or collectively by a variadic free function. Either approach accepts a flush mode enum argument providing options, e.g. if the remote data buffers should be cleaned up or not, as seen in Listing 5.4.

The `flush` (member) functions are known symbols to the pre-compiler, so the presence or absence of flush operations in SkePU source code is subject to static analysis and optimization.

5.3.1 External scope

Recently introduced as part of SkePU 3, the final piece of the puzzle in the SkePU memory consistency model is the *external scope*. Code placed in an external scope is guaranteed to be executed in a sequential and synchronous context, and as long as smart container dependencies are declared correctly, Listing 5.4: Examples of using the flush operation.

```
skepu::Vector<int> v1(n), v2(n);
skepu::Matrix<int> m1(n, n), m2(n, n);

v1.flush(); // FlushMode::Default
m1.flush(); // FlushMode::Default

skepu::flush(v2, m2); // FlushMode::Default

v1.flush(skepu::FlushMode::Dealloc);
m1.flush(skepu::FlushMode::Dealloc);
skepu::flush<skepu::FlushMode::Dealloc>(v2, m2);
```

all data belonging to containers declared as `read` will be made available to read inside the external scope, and all changes to those declared as `write` are kept consistent and available to skeletons as soon as the scope is exited.

The syntax is as follows:

```
skepu::external (
  [ skepu::read(rdcontlist),] [&]() {
    ...
  } [, skepu::write(wrcontlist)]
);
```

where the optional arguments `skepu::read()` and `skepu::write()` list container objects that may be *read from* respectively *written to* main memory in the code block (`...`).

The main purpose of this scope and corresponding construct is to maintain a sequential programming interface even when a SkePU program is launched as a SPMD program, i.e., when the cluster backend is used (see Chapter 9). As the name implies, any operation using external resources, such as a file system or network communication, should be placed within the external scope. This semi-automatic solution with an explicit framing construct allows to not depend on static analysis by the pre-compiler, which may not be feasible in the context of separate compilation and using libraries.

A typical SkePU application may have a program structure of an `external` construct early on to read input data from a file, followed by a sequence of skeleton invocations performing computation, and finally another `external` block for serializing results.

6 Standard library

This chapter presents an overview of a recent addition to SkePU: the *standard library*. This title is somewhat misleading, as SkePU is not a standardized interface; nonetheless, the idea is to provide a collection of functionality offered by the framework that cannot be sorted under the labels of ”skeletons” or ”data abstractions”. As the *library* term emphasizes, these components are generally not precompiler-driven language extensions, and rather implemented as header files operating on top of the core SkePU constructs.

We introduce seven modules from the standard library. Figure 6.1 illustrates the header file names of them, and with the asterisk annotation shows the two exceptions to the aforementioned rule; the marked modules are dependent on the precompiler for their implementation. In addition to what is listed in the chapter, we have ideas for multiple new library modules in the future.

6.1 Deterministic random number generation

Deterministic parallel random number generation is an important component of SkePU’s standard library. As we consider it a key contribution of the work, it is covered in significant detail in Chapter 13.

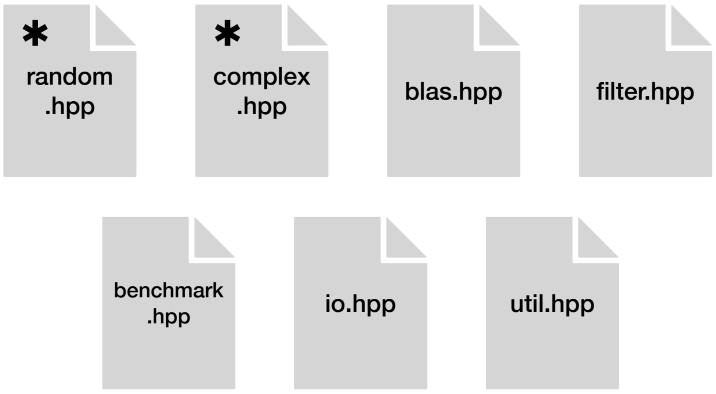


Figure 6.1: SkePU standard library modules as of early 2022.

6.2 Complex numbers

As previously described in Section 4.11, SkePU skeletons can operate on smart data-containers with user-created types. These types have to be defined as simple C-like structs, which means that user types cannot utilize general C++ class features such as member functions or operator overloading. In particular the case of overloaded operators become a problem when writing applications dependent on complex numbers[1]. C-derived languages, including C++, OpenCL, and CUDA, do not have a history of complex numbers as a language-integrated fundamental type, so custom implementations continue to be prevalent. Standard-library complex number implementations are not language-agnostic, and such cannot be used in SkePU user function code.

It has therefore been the case that any SkePU application with complex numbers needed to define their own complex type as a user type. Without access to operator overloading and member functions, accessors and arithmetic operators had to be defined as free functions meeting all user function requirements.

The standard library introduces a complex number class to increase the programmability and readability of SkePU programs. While the standard library files declare the basic types and functions, this part of the library needs to be compiler-known and integrated into the SkePU precompiler's code generation phase. Implementing complex numbers this way also serves as an initial effort to widen the scope of the user type functionality in SkePU, as the complex number library type goes beyond the basic C struct limitations, e.g., by being declared as template types, parameterized on the underlying floating-point type.

[1]In the mathematical sense: numbers consisting of a real part and an imaginary part.

Listing 6.1: Example program using SkePU complex numbers.

```
#include <skepu>
#include <skepu-lib/complex.hpp>

using Complex = skepu::complex::complex<float>;
using InnerType = Complex::value_type; // float

auto complex_dotprod = skepu::MapReduce(
  skepu::complex::mul<Complex>,
  skepu::complex::add<Complex>
);

int main(int argc, char* argv[])
{
  const size_t size{1000};
  skepu::Vector<Complex> v1(size, {1, 1}), v2(size, {1, 0});

  auto res = complex_dotprod(v1, v2);
  std::cout << "Result: " << res << "\n";
}
```

The operator overloads of SkePU complex numbers also extends to combinations of complex numbers with different underlying precision or with real-valued data types. SkePU complex numbers are moreover binary-compatible with `std::complex`, the complex number type offered by the C++ standard library. With the proper type cast, a SkePU smart data-container can be initialized from a corresponding C++ `std::complex` pointer with $O(1)$ time and memory overhead.

Listing 6.1 shows an example program, a complex-valued dot product computation, using the SkePU standard library interface.

6.3 Linear algebra

During the 1970s, an initial subset of what would become BLAS (Basic Linear Algebra Subprograms) level 1 was designed for Fortran as a library specification and implementation of commonly used linear algebra computations. Over the next decade, the library was complemented with level 2 and 3. It has since become a key library for the HPC domain.

The BLAS subroutines have names and signatures following a set pattern, making the specification cohesive and predictable for users. Crucially, the BLAS specification provides each subroutine in variants for each of four different data types[2] which are single and double-precision real and complex numbers.

When designing the interface for SkePU-BLAS, one central goal is interoperability with other BLAS libraries. However, the reference implemen-

[2]Except in cases where such a specialization would be meaningless.

tation of BLAS remains written in Fortran, with the separate CBLAS distribution existing, as well. However, SkePU-BLAS cannot be exactly modeled after CBLAS, since the other, even more important goal is seamless integration of SkePU smart data-containers in each BLAS call. CBLAS parameters are declared as raw pointers, which cannot encode the type signatures of smart data-containers alone[3].

The online repository hosting BLAS references is Netlib[4]. While not a part of the main reference material of the repository, Netlib are endorsing the SLATE project's [1] C++ conversion of the BLAS interface, BLAS++ [71]. The decision was therefore made to design the SkePU-BLAS library in as close accordance with BLAS++ as possible, as this is expected to provide the best chance of cross-library BLAS compatibility in the future for C++-derived languages and frameworks.

A notable difference between BLAS++ and SkePU-BLAS are the fact that SkePU-BLAS uses smart data-container parameters whenever possible. SkePU containers carry information about the inherent size of the vector or matrix, which also renders some parameters redundant[5].

Complex number computations with SkePU-BLAS use smart data-containers of `skepu::complex` values (as described in Section 6.2). The SkePU complex data type is guaranteed to be binary-compatible with `std::complex` which is used in BLAS++.

Listing D.4 contains a full conjugate gradient solver implemented with the SkePU-BLAS interface. The current state of SkePU-BLAS coverage is documented in Appendix C.

6.4 Image filtering and visualization

Image filtering is an application domain which SkePU has a natural affinity to. Thanks to the expressive `Map` skeleton configurations, most pixel-level transformations such as color channel adjustments, color space changes, and layer blending are efficient to describe and result in good performance. In addition, `MapOverlap` has a natural application in stencil (or *convolutional*) image filters, most notably blur operations.

SkePU has the advantage of strong GPU backends, originating as a CPU+GPU library. As the name suggests, GPUs (graphics processing units) excel at processing image-like datasets. Non-trivial image filters are commonly built as "pipelines" of smaller filter stages, which integrates well with

[3]A smart container can in certain contexts be converted to a C pointer, but not the other way around.

[4]`https://www.netlib.org/blas/`

[5]In particular, the `ldX` parameters, where X is some letter, declare the size of the first dimension of pointer argument X, which in SkePU is equivalent to the row length of a `Matrix` container.

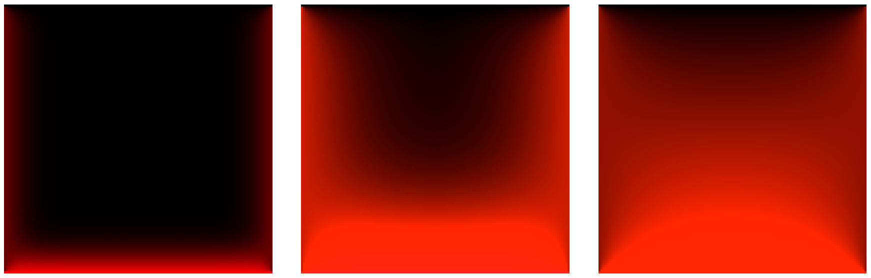

(a) 100 iterations (b) 1000 iterations (c) 10000 iterations

Figure 6.2: Visualizing the progress of a heat diffusion simulation.

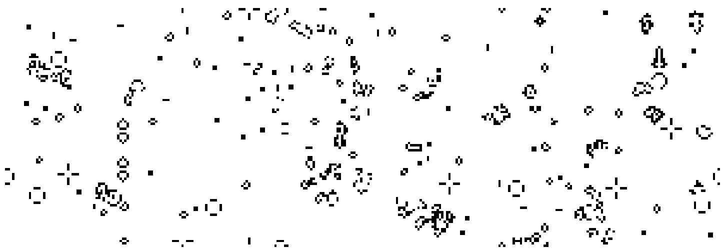


Figure 6.3: Game of life simulation snapshot, computed using SkePU skeletons and rendered with its image processing library component.

the memory management systems in the smart data-containers, ensuring that the pixel data is kept in GPU memory during the entire filter pipeline.

A standard library component for image filtering helps exposing SkePU's natural affinity for such computations by providing a selection of user functions and data types for converting between color spaces and computing common filter stages, often with additional parameters.

However, image filtering is in itself an application domain which does not have the highest relevance for HPC. The filtering functionality of the standard library is first and foremost useful for auxiliary purposes, such as visualization.

A different noise visualization using the image processing constructs can be seen in Figure 15.26.

There are several reasons to why an academically-oriented programming framework like SkePU benefits from an accessible image processing and visualization components:

- **Validation and debugging**: SkePU is suitable for rapid prototyping as a way to test the workload and performance potential of parallelization. A prototype is likely to contain errors, and an iterative programming workflow needs good debugging facilities. Visualizing

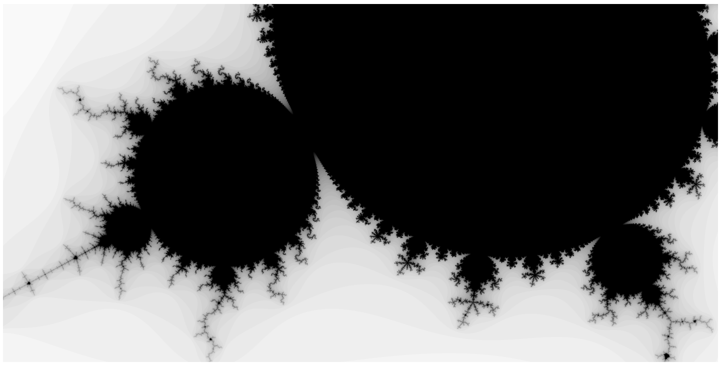


Figure 6.4: Part of the Mandelbrot fractal set (black pixels) generated by SkePU, with the surrounding gradient indicating how many iterations are required to determine the Mandelbrot set membership for each pixel.

the dataset throughout its execution (as seen in Figure 6.2) can be one of several tools to informally verify that the application behaves correctly, and if not, it can be used to track down the stage where the error is introduced and possible causes.

- **Tutorials**: When users are introduced to SkePU for the first time, visualization of the result can be a good way to provide immediate feedback of their work. SkePU consists of data-parallel pattern constructs which are intended to be used on large data sets, and the output of SkePU applications are difficult to convey through textual print-outs alone. ”Game of life” has been used in SkePU tutorials for this purpose (see Figure 6.3).
- **Teaching**: SkePU is used in teaching of parallel programming concepts in masters-level university courses. In each lab assignment, the task often includes visualization. This can be of a performance plot, which can be generated by the library module for time measurement (Section 6.5), or of the working dataset itself. A problem domain such as Mandelbrot fractal generation provides a direct visual intuition to load-balancing problems: in Figure 6.4, the darker pixels take considerably more time to compute than the lighter ones.

A concrete example is described in Section 15.11, where the image filtering components of the standard library are used for the evaluation of high-level skeleton fusion.

Listing 6.2: Example usage of SkePU time measurement utilities.

```
auto time = skepu::benchmark::measureExecTime([&]()
{
  // workload to measure
});

std::cout << "Time: " << time.count() / 1E6 << "\n";

const int repeats = 11;
auto medianTime = skepu::benchmark::basicBenchmark(
  repeats, 0, [&](size_t)
{
  // workload to measure several times
});

std::cout << "Median: " << medianTime.count() / 1E6 << "\n";
```

6.5 Benchmark utilities

Since SkePU is mainly intended, and used, as a scientific research framework, the standard library provides time measurement functionality. The fundamental building block is a collection of lambda-based time measurement functions, which allows for high-level timing code in a similar style to the rest of SkePU. Internally, the library uses `std::chrono` primitives from C++11.

On top of the basic timing utilities, the library provides ”benchmarking” abstractions. These functions run the code snippet multiple times, filtering the results and reporting, e.g., the median duration. Other variants can take lists of backend specifications and run multi-backend comparisons without further user intervention. Example usage is provided in Listing 6.2.

6.6 High-level consistent input and output

Our experience from using SkePU in the wild include observations of prominent boilerplate code. Since many SkePU applications in practice are smaller computational kernels, *input and output* often take up a significant fraction of the entire program. With the change to weak consistency in SkePU 3, this issue became more critical. Writing a SkePU smart data-container to disk or to the command line is a common operation, either as the final step of a program or temporarily during the implementation and debugging process. With weak consistency, however, a container cannot simply be handed over to the `std::ostream` operators, as the values inside can be stale. So rather than forcing the programmer to remember flushes in conjunction with every I/O operation, we extended standard library with a SkePU variant of I/O streams, enforcing consistency of data passing through them. See Listing 6.3 for a short sample usage.

Listing 6.3: Example usage of SkePU consistent I/O streams.

```
skepu::Vector<Atom> atoms(atomcount);
skepu::Matrix<float> energy_out(N, N);

// Compute full or partial result
coulombic(energy_out, atoms, ...);

// Print the data for debugging
skepu::io::cout << "Energy out: " << energy_out << "\n";
```

6.7 General utilities

As mentioned in the introduction, small code segments tend to appear quite often in SkePU programs. The utilities module of the library attempts to remedy this by providing common user functions, such as arithmetic operators, for considerably less verbose skeleton instantiation. The functions are provided as templates whenever possible for maximal utility. This is another aspect of the standard library which gives no real benefit for performance or portability, but rather benefits the general usability of the SkePU framework.

7 Implementation

This chapter provides insight into the implementational aspects behind the SkePU project. Anything presented below is not intended to be required knowledge by users of SkePU as a programming interface and framework, and relying on implementation details (including the generated code) discussed in this chapter could lead to SkePU applications breaking as the implementation evolves.

7.1 Implementation overview

SkePU is implemented in three parts. There is a sequential runtime system, a source-to-source compiler tool, and the parallel runtime system[1] with multiple backends supported. The integration of these parts is illustrated in Figure 7.1.

A SkePU program can be compiled with any standard C++11 compiler, producing a sequential executable. This means that the sequential skeletons can act as a reference implementation, both to users—who can test

[1]The parallel runtime of SkePU 2 and later is based on the original SkePU 1.x backends. By a combination of using new and powerful C++11 language features, offloading boilerplate work to the precompiler, and general improvement of the implementation structure, the verbosity and code size of the implementation was greatly reduced. In some areas, e.g., combining the source code for unary, binary, and ternary `Map` skeletons into a single variadic template, the amount of lines of code was reduced by over 70 percent. [58]

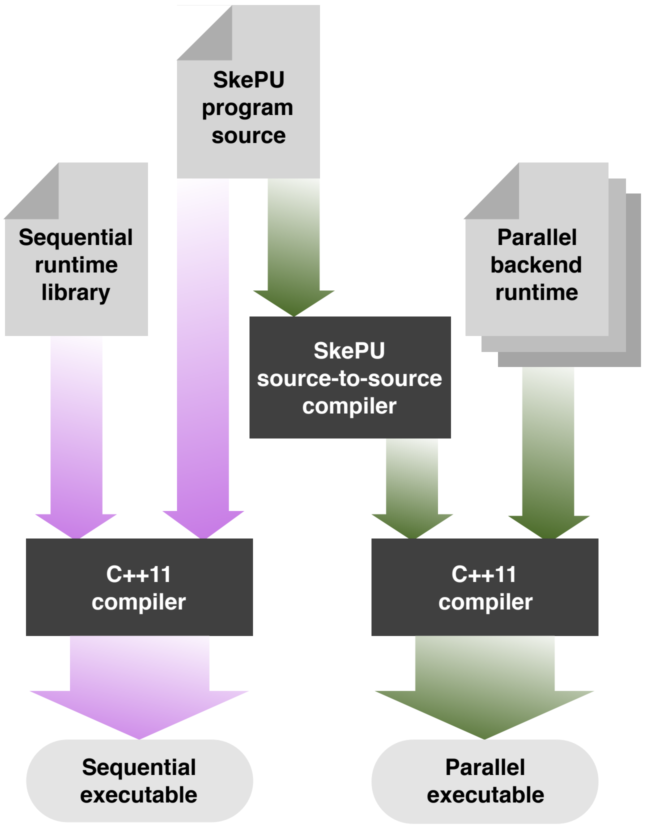


Figure 7.1: SkePU compiler chain.

their applications sequentially at first, with the advantages of simpler debugging and faster builds—and to SkePU backend maintainers.

7.2 Language embedding and type safety

SkePU is an API, or arguably language extension, on top of C++. C++ is not formally a type-safe programming language [17], and SkePU also does not claim type safety in the formal sense. There are, however, design and implementation consequences which affect the perceived type safety of SkePU. In particular, the change from macro expansions to template metaprogramming and source-to-source compilation in SkePU 2 had such consequences (see Section 7.2.1).

One scenario where SkePU is more type-safe than its host language is shown in Listing 7.1. The final skeleton invocation fails at compile-time, Listing 7.1: Example SkePU program illustrating its type-safety behavior.

```
#include <skepu>

double square(double val)
{
  return val * val;
}

int main(int argc, char *argv[])
{
  const size_t N = 10;

  skepu::Vector<float>  vec_float(N, 5.0f);
  skepu::Vector<double> vec_double(N, 5.0);
  skepu::Vector<double> vec_res(N);

  float  scalar_float = 5.0f;
  double scalar_double = 5.0;

  auto skel = skepu::Map(square);


  // Works: same type
  auto res1 = square(scalar_double);

  // Works: implicit conversion
  auto res2 = square(scalar_float);

  // Works: same type
  skel(vec_res, vec_double);

  // Compile time error: type mismatch
  skel(vec_res, vec_float);

  return 0;
}
```

even though C++ normally allows implicit type conversions from `float` to `double`.

Another interesting case is the `Reduce` skeleton. Both parameters and also the return value for the reduce operator needs to be of equal type to allow for optimizations through, e.g., tree reductions. This constraint (related to the associative property) does not apply for general folds, but is enforced in SkePU, as seen in Listing 7.2.

7.2.1 Improved type safety from SkePU 1

One of the goals with the SkePU 2 design was to increase the level of type safety from SkePU 1. In the following example, a programmer has made the mistake of supplying a unary user function to Reduce. Listing 7.3 shows the error in SkePU 1 code, and Listing 7.4 illustrates the same in SkePU 2 syntax.

Listing 7.2: Example SkePU program illustrating the type enforcement of the Reduce skeleton.

```
double add(double lhs, float rhs)
{
  return rhs + lhs;
}

auto fold = skepu::Reduce(add);

//error: no matching function for call to 'Reduce'
//auto fold = skepu::Reduce(add);
//              ^~~~~~~~~~~~~
//note: candidate template ignored: deduced conflicting types
//      for parameter 'T' ('double' vs. 'float')
```

Listing 7.3: Faulty SkePU 1 code.

```
UNARY_FUNC(plus_f, float, a,
  return a;
)

skepu::Vector<float> v(N);
skepu::Reduce<plus_f> globalSum(new plus_f);
globalSum(v);

// In SkePU 1, at run-time:
[SKEPU_ERROR] Wrong operator type!
              Reduce operation require binary user function.
```

Listing 7.4: Faulty SkePU 2 code.

```
float plus_f(float a)
{
  return a;
}

skepu2::Vector<float> v(N);
auto globalSum = skepu2::Reduce(plus_f);
globalSum(v);

// In SkePU 2, at compile-time:
error: no matching function for call to 'Reduce'
   auto globalSum = skepu2::Reduce(plus_f);
                    ^~~~~~~~~~~~~~
note: candidate template ignored: failed template argument deduction
   Reduce(T(*red)(T, T))
```

The SkePU 1 example compiles without problem, and only at run-time terminates with an error message. The message itself is shared between all reduce instances, limiting the information obtained by the user. SkePU 2, on the other hand, halts compilation and prints an error message even before the precompiler has transformed the code. It directs the user to the affected skeleton instance.[2]

7.3 Source-to-source compiler

The role of SkePU's source-to-source precompiler is to transform programs written for the sequential interface for parallel execution. The precompiler has four major tasks:

- **Kernel code generation**: For backends like OpenCL, which are not compatible with C++ syntax or runtime features, the precompiler will generate kernel code compiled and run on the external device.
- **Run-time support**: In addition to the kernel code itself, the precompiler generates "glue code" that launches the device kernel as well as supporting definitions and data structures. This way, the implementation of parallel skeleton patterns in the SkePU library can be simplified and support backend compilers with less feature-rich or stable template metaprogramming implementations.
- **Analysis and optimization**: Having full access to the source code and its AST, the precompiler can suggest or perform optimizations on the program to improve performance while preserving functionality. This aspect of the precompiler is promising but limited so far; it remains one of the promising areas for future work on SkePU.
- **Error checking**: SkePU comes with several limitations on what can and cannot be done in, e.g., the user functions. Many of these restrictions are not enforceable within the C++ type system and can lead to compilation errors in the backends, or even undefined run-time behavior of programs. Since the precompiler is based on the Clang framework, it has access to inline error and warning formatting and will catch common mistakes early on. The checking is inherently limited, since guaranteeing a C++ program's correctness is impossible.

The task of the precompiler is limited by design. Its main purpose is to transform user functions, for example by adding `__device__` keywords for CUDA variants and stringifying the OpenCL variant. A user function is represented as a `struct` with static member functions in the transformed

[2]The message does not directly describe the issue, an aspect which can be further improved with C++11's `static_assert`.

Listing 7.5: Before precompiler transformation.

```
#include <skepu>

template<typename T>
T add(T a, T b)
{
  return a + b;
}

int main()
{
  auto adder = skepu::Map(add<float>);
}
```

program. The precompiler also transforms skeleton instances, redirecting to a completely different implementation accepting the `structs` as template arguments. It also redefines user types for backends where necessary. For some backends such as OpenCL and CUDA, all kernel code is generated by the precompiler.

An example of a transformation of the template user function in Listing 7.5 can be seen in Listing 7.6. In this case, only the sequential CPU (on by default), OpenMP, and OpenCL backends are enabled. Each GPU backend adds significantly more code to the generated output: everything executed as a kernel on the GPU device is generated by the SkePU precompiler, as well as additional boilerplate glue-code used to launch said kernels. Listing 7.7 contains an excerpt of the kernel code generated (in the OpenCL kernel language, as a static text string) for the SkePU program in Listing 7.5 and Listing 7.8 shows part of the code generated on the CPU side to launch the kernel. Note that both of these listings are edited for presentational purposes.

7.4 Backends

After a SkePU program has been processed by the precompiler, the generated source code now has access to the full SkePU runtime library of implementation backends.

The *hybrid backend* is a major contribution and covered in great detail in Chapter 8; similarly, the two *cluster backends* are detailed in Chapter 9.

7.4.1 Sequential CPU backend

The most straightforward of the SkePU backends is the sequential CPU variant, which is different compared to the direct compilation path of SkePU. When using the precompiler and the sequential backend, the application uses the full smart data-container implementation and goes through the

Listing 7.6: After precompiler transformation.

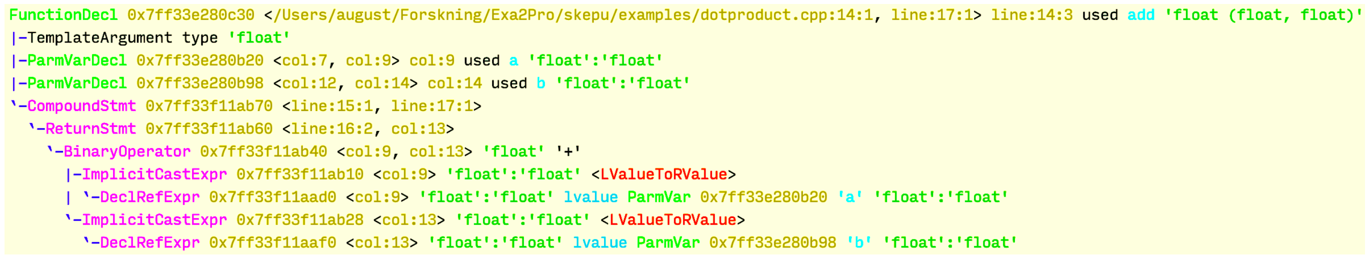
```
#define SKEPU_PRECOMPILED
#define SKEPU_OPENMP
#include <skepu>

template<typename T>
T add(T a, T b)
{
  return a + b;
}

struct skepu_userfunction_adder_add_float
{
  using T = float;
  constexpr static size_t totalArity = 2;
  constexpr static size_t outArity = 1;
  constexpr static bool indexed = 0;
  using IndexType = void;
  using ElwiseArgs = std::tuple<float, float>;
  using ContainerArgs = std::tuple<>;
  using UniformArgs = std::tuple<>;
  typedef std::tuple<> ProxyTags;
  constexpr static skepu::AccessMode anyAccessMode[] = {};
  using Ret = float;
  constexpr static bool prefersMatrix = 0;

#define SKEPU_USING_BACKEND_OMP 1
#undef VARIANT_CPU
#undef VARIANT_OPENMP
#undef VARIANT_CUDA
#define VARIANT_CPU(block)
#define VARIANT_OPENMP(block) block
#define VARIANT_CUDA(block)
  static inline SKEPU_ATTRIBUTE_FORCE_INLINE float OMP(float a, float b)
  {
    return a + b;
  }
#undef SKEPU_USING_BACKEND_OMP

#define SKEPU_USING_BACKEND_CPU 1
#undef VARIANT_CPU
#undef VARIANT_OPENMP
#undef VARIANT_CUDA
#define VARIANT_CPU(block) block
#define VARIANT_OPENMP(block)
#define VARIANT_CUDA(block) block
  static inline SKEPU_ATTRIBUTE_FORCE_INLINE float CPU(float a, float b)
  {
    return a + b;
  }
#undef SKEPU_USING_BACKEND_CPU
};

int main()
{
  skepu::backend::Map<2,
    skepu_userfunction_adder_add_float,
    bool, void> adder(false);
}
```

Listing 7.7: Generated OpenCL kernel.

```
__kernel void add_precompiled_MapKernel_add_float_arity_2(
  __global float* skepu_output,
  __global float *a,
  __global float *b,
  size_t skepu_n,
  size_t skepu_base
)
{
  size_t skepu_i = get_global_id(0);
  size_t skepu_gridSize = get_local_size(0) * get_num_groups(0);

  while (skepu_i < skepu_n)
  {
    skepu_output[skepu_i] = add_float(a[skepu_i], b[skepu_i]);
    skepu_i += skepu_gridSize;
  }
}
```

Listing 7.8: Generated OpenCL kernel launcher code.

```
template<typename Ignore>
static void map
(
  size_t skepu_deviceID,
  size_t skepu_localSize,
  size_t skepu_globalSize,
  skepu::backend::DeviceMemPointer_CL<float> *skepu_output,
  skepu::backend::DeviceMemPointer_CL<float> *a,
  skepu::backend::DeviceMemPointer_CL<float> *b,
  Ignore,
  size_t skepu_n,
  size_t skepu_base
)
{
  skepu::backend::cl_helpers::setKernelArgs(
    skepu_kernels(skepu_deviceID),
    skepu_output->getDeviceDataPointer(),
    a->getDeviceDataPointer(),
    b->getDeviceDataPointer(),
    skepu_n,
    skepu_base
  );
  cl_int skepu_err = clEnqueueNDRangeKernel(
    skepu::backend::Environment<int>::getInstance()
      ->m_devices_CL.at(skepu_deviceID)->getQueue(),
    skepu_kernels(skepu_deviceID),
    1, NULL,
    &skepu_globalSize, &skepu_localSize,
    0, NULL, NULL
  );
  CL_CHECK_ERROR(skepu_err, "Error launching Map kernel");
}
```



```
FunctionDecl 0x7ff33e280c30 </Users/august/Forskning/Exa2Pro/skepu/examples/dotproduct.cpp:14:1, line:17:1> line:14:3 used add 'float (float, float)'
|-TemplateArgument type 'float'
|-ParmVarDecl 0x7ff33e280b20 <col:7, col:9> col:9 used a 'float':'float'
|-ParmVarDecl 0x7ff33e280b98 <col:12, col:14> col:14 used b 'float':'float'
`-CompoundStmt 0x7ff33f11ab70 <line:15:1, line:17:1>
  `-ReturnStmt 0x7ff33f11ab60 <line:16:2, col:13>
    `-BinaryOperator 0x7ff33f11ab40 <col:9, col:13> 'float' '+'
      |-ImplicitCastExpr 0x7ff33f11ab10 <col:9> 'float':'float' <LValueToRValue>
      | `-DeclRefExpr 0x7ff33f11aad0 <col:9> 'float':'float' lvalue ParmVar 0x7ff33e280b20 'a' 'float':'float'
      `-ImplicitCastExpr 0x7ff33f11ab28 <col:13> 'float':'float' <LValueToRValue>
        `-DeclRefExpr 0x7ff33f11aaf0 <col:13> 'float':'float' lvalue ParmVar 0x7ff33e280b98 'b' 'float':'float'
```

Figure 7.2: Clang AST of the add user function from Listing 7.5.

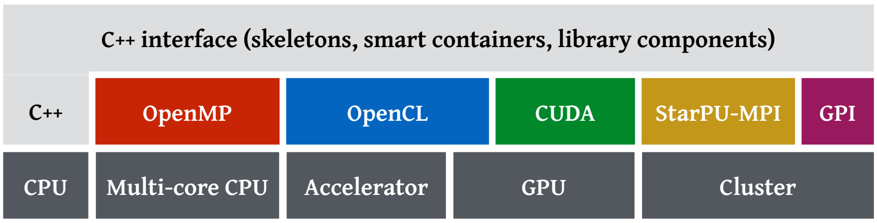


Figure 7.3: Backends available in SkePU.

standard backend selection process. Thus, the motivation for a sequential CPU implementation is a more fair comparison for performance evaluations. The sequential backend is also generally more optimization-oriented compared to the direct compilation, and may have fewer correctness checks and programmer feedback mechanisms. Compared to the multi-core CPU backend, the sequential CPU implementation avoids potential overhead of OpenMP directives and thread management. However, the sequential backend is rarely relevant for real-world computations, where the data sets are sufficiently large.

7.4.2 Multi-core CPU backend: OpenMP

For multi-core systems, SkePU provides a multi-threaded backend based on the industry standard OpenMP interface. Using this backend requires an OpenMP-supported C++ compiler such as GCC or ICPC.

In SkePU 2 and earlier, all skeletons, in particular the `Map` based skeletons, assumed an equal load distribution of the user function executions over the entire range of input container elements. Some applications may however exhibit an irregular workload distribution instead, especially in CPU-affine computations and sometimes even in combination with very short input vectors, which are typically prime targets for the OpenMP backend.

For these cases, SkePU 3 adds support for *dynamic scheduling* in the OpenMP backend. Available scheduling modes in the OpenMP backends are *dynamic*, *guided* self-scheduling, *auto* (for auto-tuned scheduling as implemented in the OpenMP target compiler), and of course *static* which is the default scheduling mode.

In addition, the chunk size (the smallest number of user function invocations to schedule as a group) can be explicitly set in the backend specification.

Performance evaluation results for three load-balancing benchmarks using the OpenMP backend are given in Section 15.9.1.

7.4.3 GPU backends: OpenCL and CUDA

SkePU targets GPUs using either the CUDA or OpenCL frameworks. OpenCL can also target other types of accelerators, such as Intel Xeon Phi, while CUDA is vendor-specific to Nvidia GPUs. Most of the skeletons in SkePU can target several GPUs at once, splitting up the work between them. (A separate hybrid backend can optionally use GPUs and OpenMP simultaneously, with a load-balanced work distribution.)

7.5 C and Fortran language bindings

C++ is a strong and growing language for applications in high-performance computing, but the the lower-level alternatives C and Fortran remain frequently used, in particular for older codebases. (The study by Amaral et al. [4] provides thorough insight into a recent statistical snapshot of the field.) It has therefore been suggested that SkePU ought to provide an interface for C and Fortran, specifically to widen its application target space. As a subproject within EXA2PRO, a prototype was designed to evaluate the feasibility of such an interface. While Fortran was the main goal, one of the conclusions of the attempt is that a C interface should be used as an adaptor layer. Exposing a C API for SkePU integration will also allow for easier extension to other languages in the future.

The SkePU API is inherently based on modern C++ features that do not exist in C nor Fortran, most importantly variadic templates. For this reason, we concluded that there will be no full-fledged Fortran or C API for the complete SkePU interface. The solution for non-C++ applications is to write the actual skeleton code, including instantiation and user functions, in C++. User functions and skeletons form the computational core of an application, which in practice can be just a fraction of an entire application stack. Wrapper translation overhead in the computational core would also come at a significant performance cost.

The SkePU precompiler provides interface layers for smart data-container access through C and Fortran. Since containers are user-instantiated templates, the data access API is not provided as static header and source files; the precompiler generates the wrappers on demand. The full build process involves compilation into multiple binaries and a linking phase as illustrated in Figure 7.4.

Because of the absence of templates, the C and Fortran interface is much wordier than standard C++ SkePU. An example program is shown in Listing 7.9; note that the prototype syntax may change in the future.

Listing 7.9: Sample program utilizing the prototype SkePU-Fortran bindings.

```
program skepu_in_fortran
  use skepu
  use f_skeleton
  implicit none

  integer :: N
  integer :: vec_handle, mat_handle
  integer :: i
  integer :: val, newval

  write(*,*) 'Enter size: '
  read(*,*) N

  ! Allocate
  vec_handle = skepu_create_vector_int(N)
  mat_handle = skepu_create_matrix_int(N, N)

  call skepu_write_matrix_int(vec_handle, 0, 0, 1)

  ! Initialize
  do i = 0, N-1
    call skepu_write_vector_int(vec_handle, i, i)
  end do

  ! Some in-line processing
  do i = 0, N-1
    val = skepu_read_vector_int(vec_handle, i)
    newval = val * 2
    call skepu_write_vector_int(vec_handle, i, newval)
  end do

  ! Call FORTRAN wrapper to C wrapper with skeleton call
  vec_handle = square(vec_handle, vec_handle)

  ! Logging
  call skepu_flush_vector_int(vec_handle)
  write(*,*) 'Contents of vector: '
  do i = 0, N-1
    val = skepu_read_vector_int(vec_handle, i)
    write(*,fmt="(i0,1x)",advance="no") val
  end do
  write(*,*) ''

  ! Deallocate
  call skepu_delete_vector_int(vec_handle)

end program skepu_in_fortran
```

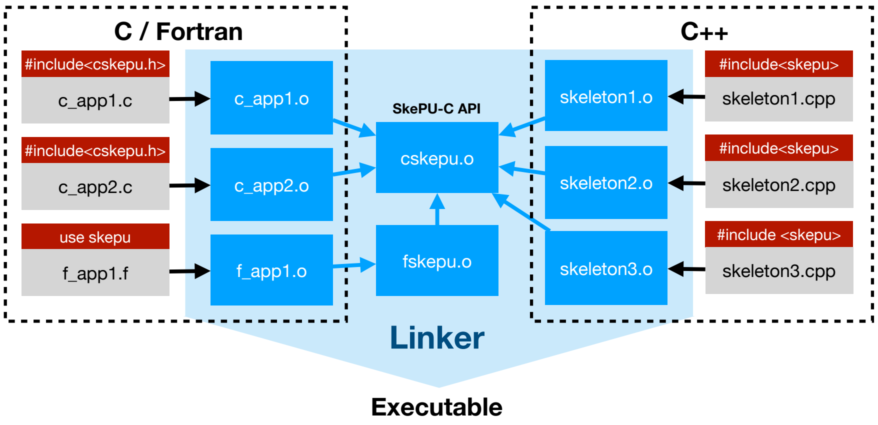


Figure 7.4: Build system for SkePU applications using the Fortran interface.

7.6 Continuous integration and testing

SkePU is primarily a research-oriented project, distributed as open-source and not marketed as a commercial product. However, part of the goal of SkePU is to be a viable tool for integration into existing or new C++ applications as a way to parallelize computation in a portable and performant manner; or at the very least be a good choice for prototyping skeleton programming or high-level parallelism in general. This goal requires some level of stability and reliability of SkePU as well as an accessible installation process. As SkePU has matured as a framework over the past few years, it has been evolving in these aspects too, and as of now has an established continuous integration system in place, including automated testing facilities ranging from unit-level tests (for instance, on smart data-container operations) as well as system-level tests of the entire build-and-run process of SkePU applications. The testing infrastructure is relatively new and the number and types of tests are steadily increasing.

7.7 Dependencies

SkePU requires the target platform to provide a C++11-conforming compiler. C++11 support in compilers is quite mature today, and support is available in all recent versions of GCC, Clang, and the Intel, Microsoft, and Nvidia toolchains. Access to the precompiler tool is also necessary for parallel builds, so by extension a development system needs to be able to build LLVM and Clang. These code repositories and the generated build files are quite large, several hundreds of megabytes in total. However, the SkePU toolchain is designed to allow for cross-precompilation. In other words, all decisions based on the architecture and available accelerators, etc., are made after the precompilation step, and it is possible to split the build process of SkePU programs such that the precompilation occurs on a system with the full precompiler LLVM stack installed. The final compilation step only needs the backend compiler, and can be done, e.g., on an embedded system with a small storage footprint.

Cmake is used throughout LLVM and also for the SkePU examples. For testing, SkePU relies on the *ctest* build environment (included as part of Cmake) and on *Catch 2* for the test program implementation.

7.8 Availability

SkePU is made available to the general public through an open-source distribution of all its source code, including the source-to-source compiler. It is published with a permissive modified four-clause BSD license and hosted on GitHub[3]. *Cmake* support and extensions helps making the SkePU compiler toolchain possible to integrate in existing application build systems. Documentation and code samples are hosted at the SkePU website, and several recent publications on SkePU are available as open access.

[3] `https://skepu.github.io`

8 Hybrid CPU-GPU skeleton execution

This chapter is closely based on the following publication:

Tomas Öhberg, August Ernstsson, and Christoph Kessler. "Hybrid CPU–GPU execution support in the skeleton programming framework SkePU." in: *The Journal of Supercomputing* (Mar. 2019). ISSN: 1573-0484. DOI: `10.1007/s11227-019-02824-7`

Experimental evaluation is presented later, in Chapter 15.

In this contribution, initiated during Tomas Öhberg's master's thesis [118] supervised by August Ernstsson, we present a hybrid execution backend for SkePU. The backend is capable of automatically dividing the workload and simultaneously executing the computation on a multi-core CPU and any number of accelerators, such as GPUs. We show how to efficiently partition the workload of skeletons such as `Map`, `MapReduce`, and `Scan`[1] to allow hybrid execution on heterogeneous computer systems. We also show a unified way of predicting how the workload should be partitioned based on performance modeling.

[1]The implementation has since been extended to cover `MapPairs` and `MapPairsReduce`, which were introduced later.

8.1 Introduction

An effective parallel programming framework should not only let the programmer implement the applications to run on any processing unit the hardware provides, but also to run on *all* processing units, dividing the workload between multiple processing units, possibly of different kind. This way of simultaneously executing an algorithm on multiple, heterogeneous processing units is referred to as *hybrid execution.* To relieve the burden of partitioning and scheduling from the programmers, the frameworks should preferably figure out the best way to divide the workload automatically. Such a system must take the relative performance of the hardware components of the system executing the application into consideration, as well as the characteristics of the computation.

The rest of this chapter is structured as follows: Section 8.2 presents the new hybrid backend implementation and how the workload is partitioned in all skeletons. This is followed by Section 8.3, where the auto-tuner is described. Related libraries and frameworks with support for heterogeneous architectures are discussed in Chapter 2.

The results of performance evaluations made on the hybrid execution implementation are presented in Chapter 15.

8.2 Workload partitioning and implementation

The old implementation of hybrid execution in SkePU 1 used the StarPU library as a backend. This implementation was ported to SkePU 2 and later as a baseline to compare the new hybrid backend to. See Section 2.4.6 for further background on the StarPU programming environment.

The new hybrid execution implementation in SkePU is made as a new backend, allowing the programmer to explicitly choose whether or not to use it. During precompilation the hybrid backend is automatically included if the OpenMP and either CUDA or OpenCL is selected. The hybrid backend works with both CUDA and OpenCL. Which accelerator implementation will be used is determined by availability and the programmer's preference.

In the first stage of a skeleton invocation, the workload is partitioned into two parts by the hybrid backend: one for the CPU and one for the accelerators. The CPU and accelerator parts are then further divided between the CPU threads and any number of accelerators respectively. The hybrid skeleton implementations use OpenMP, where the first thread will manage the accelerators and the rest of the threads will work on the CPU partition. The implementation is very similar to the already existing OpenMP backend, in order to match its performance. To reduce duplication of code within SkePU, the accelerator partition is computed by the already existing CUDA or OpenCL backend implementations. To make this work, some of the internal APIs of the accelerator backends (CUDA and OpenCL) had to

Listing 8.1: Using the hybrid backend with a manually set partition ratio.

```
const int NUM_THREADS = 16;
const int NUM_GPUS = 1;
const float PARTITION_RATIO = 0.2;

skepu::Vector<int> in, out;

skepu::BackendSpec spec(skepu::Backend::Type::Hybrid);
spec.setCPUThreads(NUM_THREADS);
spec.setDevices(NUM_GPUS);
spec.setCPUPartitionRatio(PARTITION_RATIO);
skeleton_instance.setBackend(spec);

skeleton_instance(out, in);
```

be generalized to work on subparts of containers. As both accelerator backends already have support for multi-accelerator computations, also the hybrid backend has support for hybrid execution with multiple accelerators. The workload partitioning in the accelerator backends is however, still limited, as the work is evenly divided between all accelerators. This works well when all accelerators are of the same type, but will not be optimal in case different accelerator models are used.

The workload is partitioned according to a single parameter: the *partition ratio*. The ratio defines the proportion of the workload that should be computed by the CPU; the rest is computed by the accelerators. The partition ratio can either be manually set by the programmer, or automatically tuned per skeleton instance to make SkePU predict the optimal partition ratio for a given input size. The auto-tuning will be described later in this paper. How to use hybrid execution with a manually configured partition ratio is shown in Listing 8.1. This example shows how to set up hybrid execution for 16 CPU threads and one accelerator, where 20% of the workload will be computed by the CPU threads, the rest by the accelerator. Which accelerator implementation (CUDA or OpenCL) to use is specified by compiler flags.

`Map` is highly data parallel by nature and is therefore straightforward to partition. The ratio defines how many output elements to compute on the CPU, the rest is computed by the accelerator backend. The CPU partition is further divided into equal sized blocks, one for each CPU thread. The partitioning scheme of the `Map` skeleton is shown for three CPU threads in Figure 8.1.

`Reduce` is performed in two steps. The partition ratio defines how many input elements to be reduced on the CPU, the rest is reduced by the accelerators. The CPU partition is further divided into equally sized blocks, one per CPU thread. First, each CPU thread and the accelerator backend reduce their block of the input data to produce a temporary array of partial reduc-

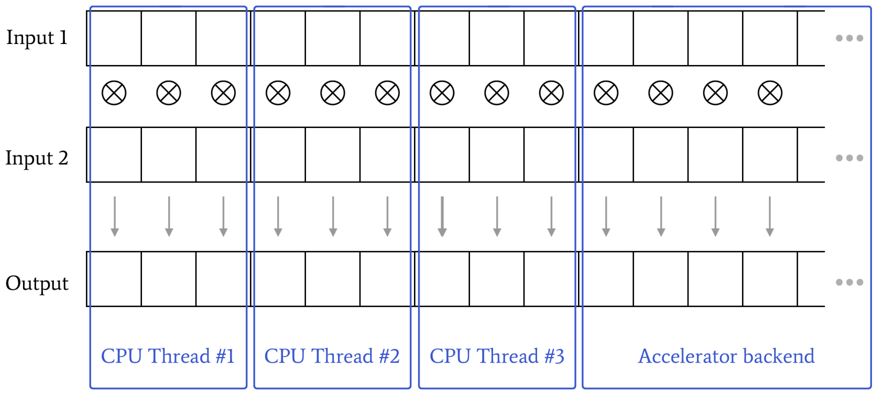


Figure 8.1: Partitioning of the `Map` skeleton.

tions. This small array is then reduced by a single CPU thread to a global result. Partitioning of the Reduce skeleton for one-dimensional input containers with two CPU threads is shown in Figure 8.2.

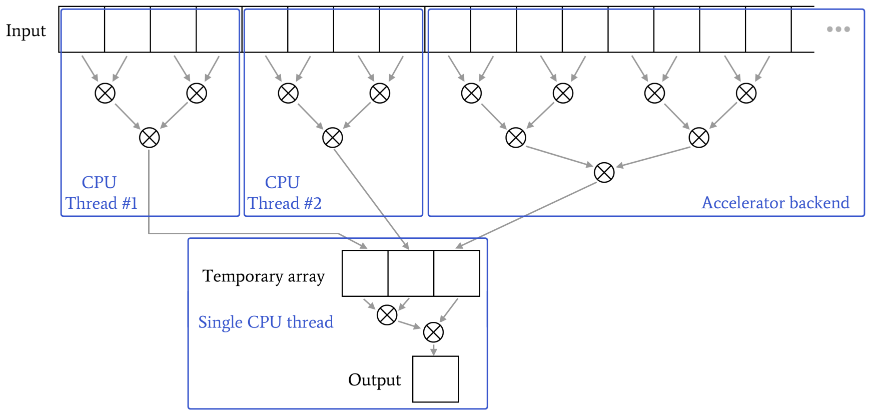


Figure 8.2: Partitioning of Reduce skeleton.

`MapReduce` is implemented in a similar way to the Reduce skeleton. The input arrays are first partitioned as in the Reduce skeleton and the CPU partition is evenly divided between the threads. Each CPU thread and the accelerator backend reduce their part of the data, by first performing the `Map` step. The intermediate results are then reduced down by a single CPU thread. Partitioning of the `MapReduce` skeleton is shown in Figure 8.3.

`MapOverlap` is similar to the `Map` skeleton. The partition ratio defines how many output elements to compute on the CPU, the rest being computed by the accelerators. The CPU partition is then divided into one block per

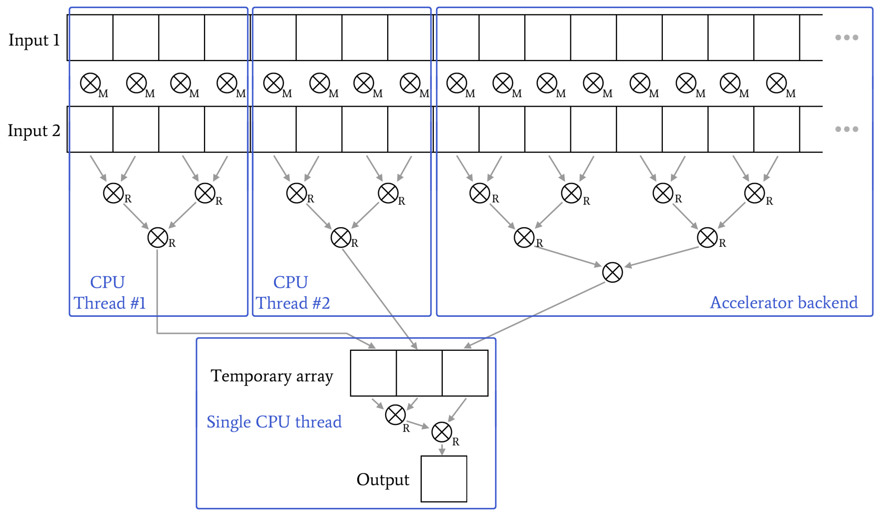


Figure 8.3: Partitioning of `MapReduce` skeleton.

CPU thread. Extra consideration had to be taken to all variations of edge handling and different corner cases caused by the size of the overlap region. Partitioning of the one-dimensional `MapOverlap` skeleton with an overlap of 1 element on each side is shown in Figure 8.4. The work of a single user function call is highlighted in yellow.

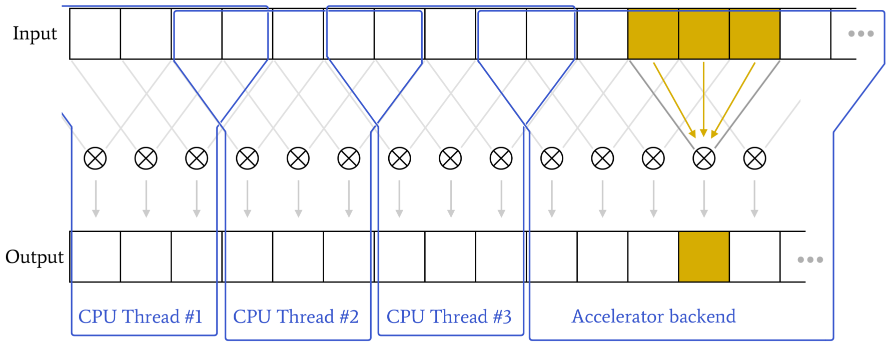


Figure 8.4: Partitioning of `MapOverlap` skeleton.

`Scan` has more data dependencies than the other skeletons and requires a more complex partitioning implementation. The input array is partitioned into a CPU and an accelerator part as before, and the CPU partition is further divided into equally sized blocks, one per CPU thread. Each CPU thread and the accelerator backend start by performing a local scan of their block of the input data. After this step, each block misses the scan offset of the preceding blocks. The last resulting element of the local scan of each CPU block are collected into an temporary array and a single CPU thread performs a scan on that array. This produces an array of the missing offset values of each block. In the second step, each CPU thread (except for the first, as its block is already complete) combine their local scan result with the missing value from the array. For the CPU, the local scan step and the combining step require the same number of operations, one per element in the block. This makes the two steps take approximately the same amount of time. This is not the case for the accelerators on the other hand, especially not a GPU. The first step is much less data parallel and takes longer than the second step where a number of independent operations are made on different data elements. This means that a GPU will go idle if the first and second steps are to be made synchronized with the CPU, as it will finish its second step much faster than the CPU. This was solved by letting the accelerator backend take care of the last part of the input array. As nothing is dependent of the result of the last block, the result of the local scan of the accelerators' partition is not needed in the missing values array. We can thus let the accelerators spend more time on the first step than the CPU threads are spending, and only make the accelerators check that the CPUs have produced the array of missing values before starting the second step. This makes load balancing between CPU and accelerators much easier and utilizes the available processing capacity better. Partitioning of the `Scan` skeleton is shown in Figure 8.5.

Apart from the partitions shown here, there are also variants for `Map` on matrices, one-dimensional and two-dimensional Reduce on matrices as well as row-wise `MapOverlap` on matrices. The implementation partitions the elements between the PUs based on the partition ratio, just as if it was an array. In the case of `Reduce` and `MapOverlap`, the matrix is partitioned row-wise, so all PUs get whole rows to operate on. This can make it hard to balance the workload for matrices with few rows. However, in connection with automatic backend selection tuning [48], such cases would probably not select the hybrid execution at all. A more sophisticated partitioning for matrices would be hard to realize, especially for the `MapOverlap` skeleton, due to the many corner cases and complex data access patterns.

8.2.1 StarPU backend implementation

To show the advantages of the static workload partitioning in the new hybrid execution backend, the experimental StarPU integration from SkePU 1 was ported to SkePU 2. Similar to SkePU, StarPU uses its own custom data management system. In order to keep SkePU's smart container API [47], the smart containers automatically transfer the control to StarPU once they are used with the StarPU backend. The control is taken back by SkePU once the

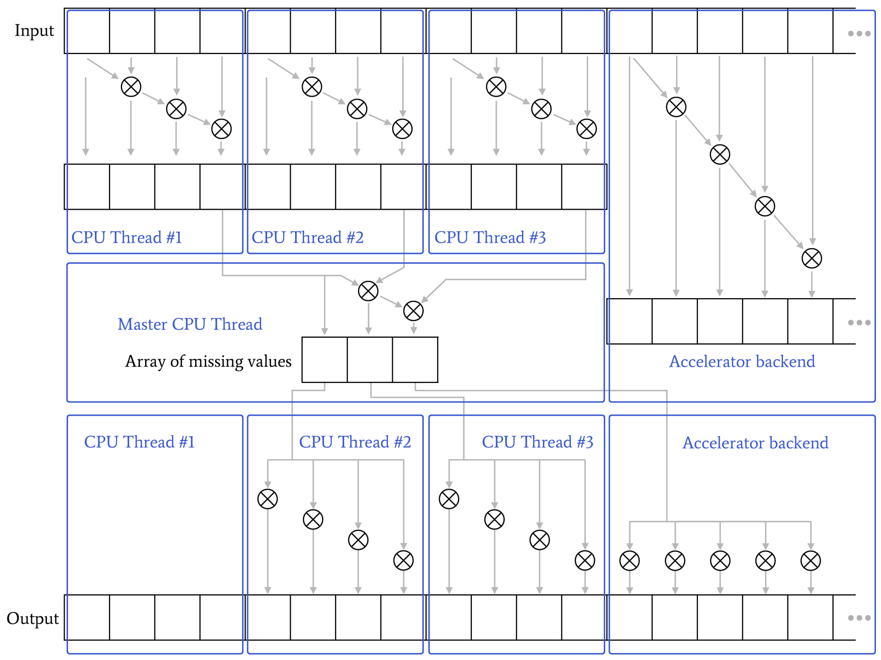


Figure 8.5: Partitioning of `Scan` skeleton.

container is used with one of the other backends. The memory management code in SkePU 2 was not changed since SkePU 1, allowing this part of the code to be reused from the old SkePU 1 integration of StarPU. StarPU is integrated into SkePU 2 as a separate backend, just as our hybrid execution implementation. This, together with the memory management implementation allows the already existing SkePU backends to be used alongside the StarPU backend. No changes had to be made to the API of SkePU, apart from adding StarPU as an extra backend type. Currently only the main features of the `Map`, `Reduce` and `MapReduce` skeletons are ported, but more skeletons and features will be ported in the future.

As StarPU is a task-based programming framework, a SkePU skeleton invocation must be mapped to a number of tasks. This is done by splitting the workload into a number of equal-sized chunks. The programmer can manually tweak the number of chunks, as this affects the performance. More chunks are desirable for larger input sizes as it makes load balancing easier, but for small input sizes too many chunks will lead to significant scheduling overheads. The StarPU backend has two implementation variants, one using OpenMP and one using CUDA. The already existing OpenMP and CUDA backend implementations could not be reused due to the abstraction gap between SkePU and StarPU. This gap was noticed already during the integration of StarPU into SkePU 1 and it has since grown even more in SkePU 2 with the increased use of metaprogramming and other high-level C++ features. StarPU uses a lower level C-style API and passes arguments using void pointers and runtime type casting. SkePU 2, on the other hand, builds argument lists at compile time using variadic templates and parameter packs. It was still possible to integrate them by implementing the StarPU functions as static member functions, creating the argument handling code at compile time.

8.3 Auto-tuning

Apart from manually setting the partition ratio, SkePU can automatically predict a suitable partition ratio by performance benchmarking. Due to how general and flexible the SkePU framework is, implementing an auto-tuner that works well for every imaginable skeleton instance may not be possible. The execution time could, for example, be bound by the size of the random access containers, by uniform arguments, or even be data-dependent on container elements. Predicting optimal partition ratio for such skeleton instances would require very sophisticated and time consuming algorithms and/or user interaction. Instead focus was put on implementing a tuner that would give good predictions for common cases, where the execution time grows linearly with the size of the element-wise accessed containers. For specific skeleton instances the partition ratio can always be set by hand.

The auto-tuning presented in this paper resembles the tuning presented by Luk et al. [103] in their Qilin framework, although our tuner extends this work by supporting multiple accelerators. This is possible as our implementation sees multiple accelerators as one single device, thanks to the already existing multi-device implementations in the CUDA and OpenCL backend. Our tuner builds two execution time models, one for the CPU and one for the accelerator backend. At tuning time the programmer must choose the number of CPU threads and accelerators to tune for. The tuning is performed once per each skeleton instance in the application for a specific machine. In case the configuration of the machine changes (for example if accelerators are added/removed or if more or less CPU cores are used) the skeleton instance has to be re-tuned. The tuning is performed on the OpenMP and CUDA/OpenCL backends. The OpenMP backend will be executed with one thread less than specified by the programmer, as one thread in the hybrid backend will be fully dedicated to running the accelerator backend. The programmer can choose upper and lower limits to the input size, as well as the number of input sizes to benchmark. The input sizes to benchmark is then evenly spread over the interval defined by the limits. The OpenMP and CUDA/OpenCL backends are executed five times on each input size, and the median value is inserted into the execution time model of that backend. This is made to minimize the impact of temporary fluctuations.

Once the execution time benchmarks are stored in the model, the model is fitted to a linear curve using least-squares fitting. As the execution time grows linearly with all skeletons (assuming the user function takes $\mathcal{O}(1)$ time), a linear approximation of the execution time is sufficient for our needs. The fitting will create two linear equations on the form:

$$t = ax + b \tag{8.1}$$

where t is the execution time, x is the input size and a and b are parameters found by the least-squares fitting.

Let us consider a problem size N and a (CPU) partition ratio R. The partition size of the CPU will then be NR and the partition size of the accelerator $N(1 - R)$. This gives us execution times t_{cpu} and t_{acc} for the CPU and accelerator respectively:

$$t_{cpu} = a_{cpu}NR + b_{cpu} \tag{8.2}$$

$$t_{acc} = a_{acc}N(1 - R) + b_{acc} \tag{8.3}$$

The workload is perfectly balanced between the CPU and the accelerator if $t_{cpu} = t_{acc}$. Combining the equations 8.2 and 8.3 and solving for the partition ratio R gives:

$$R = \frac{a_{acc}N + b_{acc} - b_{cpu}}{N(a_{cpu} + a_{acc})} \tag{8.4}$$

At runtime Equation (8.4) is used to predict the optimal partition ratio for a given input size N. In practice the value of R can be in three intervals: the interval $0 < R < 1$, where hybrid execution is predicted the optimal strategy, and the value of R is used as the partition ratio; $R \leq 0$, where accelerator-only execution is considered optimal; or $R \geq 1$, where CPU-only execution is considered optimal. In the last two cases, the hybrid backend will automatically fall back to executing the skeleton using the OpenMP or CUDA/OpenCL backends, as the overhead of hybrid execution is predicted to be too high.

9 Skeleton programming on large-scale cluster systems

This chapter is partly based on the following publication:

August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. "SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters." In: *International Journal of Parallel Programming* 49 (2021), pp. 846–866. DOI: `10.1007/s10766-021-00704-3`

The SkePU-GPI section of the chapter is adapted from the following master's thesis:

Joel Almqvist. "Integrating SkePU's algorithmic skeletons with GPI on a cluster." LIU-IDA/LITH-EX-A–22/002–SE. MA thesis. Department of Computer and Information Science, 2022

Implementation of the StarPU-MPI cluster backend detailed in this chapter was initiated as a student project; Joel Almqvist likewise implemented the StarPU-GPI backend as part of a master's thesis project. Both were conducted internally at Linköping University, supervised by August Ernstsson. The StarPU-MPI backend has since been further integrated as a proper SkePU backend and was extended with new ideas as part of the EXA2PRO project. The GPI backend detailed later remains an internal prototype.

Experimental evaluation is presented later, in Chapter 15.

9.1 Background

For any kind of parallel programming application, the key to large-scale performance is execution on cluster systems. A computational cluster consists of a large number of computers, called *nodes*, linked together by a high-speed interconnection network along with peripheral systems such as high-throughput persistent storage. Each individual node is typically very similar to a stand-alone computing workstation, albeit especially powerful and with server-grade components such as large CPUs and ECC memory. This modular cluster architecture built on off-the-shelf components is used in supercomputers and data centers and makes such systems relatively cost-efficient and scalable. As there are often many thousands of nodes, the cluster is typically work-time shared between several users and running programs simultaneously. Specialized job and resource management software (for example, SLURM[1]) schedules and maps user jobs to subsets of the available nodes. Note that *cloud computing* may use cluster resources, but the term implies an additional level of virtualization and resource distribution. In this text, we are only interested in running programs directly on the cluster through its job scheduler, as is the common practice for scientific high-performance computing workloads.

Cluster systems have some fundamental differences from what we will from now call ”single-node” computing. Firstly, the memory in a cluster is split up among its nodes; they are so-called *distributed memory* systems. This is similar to how accelerators (e.g., GPUs) on single-node platforms come with their own memory pool and address space, but different in that the accelerator memory is heterogeneous with the main memory, often smaller and with different performance characteristics. The main memory in a cluster is homogeneously split up in equal parts among all nodes. Secondly, control flow works differently. Whereas a single-node system can run a single process with several individual threads providing parallelism, the cluster architecture necessitates at least one individual process on each node, typically launched from a common program executable (see Figure 9.1). This is known as the *SPMD* model (single program, multiple data). Such processes can only communicate by using the interconnection network, called *message passing*.

Note that the cluster environment of SPMD, distributed memory, and message passing can be naturally simulated on a single-node system. Individual processes may be launched from the same executable and implicitly are assigned their own memory address space (oblivious to the fact that the virtual memory feature of the operating system maps all addresses to the same physical memory). In practice, the above means that any programming environment targeting clusters can run also on single-node systems.

[1]`https://slurm.schedmd.com/`

As they will run without the assumptions of shared memory, there is typically significant overhead, and in practice high-performance software designed for clusters combine two programming models into one: distributed memory programming for inter-node communication and synchronization, and shared-memory programming in the inter-node context.

Unfortunately, the inverse of the situation detailed in the previous paragraph is not true: a programming framework, like SkePU, designed for shared-memory, single-node platforms is not trivial to extend to clusters[2]. In this chapter, we will investigate two approaches of achieving cluster execution from SkePU single-source programs: one relatively mature implementation using *StarPU-MPI* in Section 9.2, and one recent project using *GPI* in Section 9.3.

9.2 StarPU-MPI backend

StarPU is a high-level runtime system designed to provide a unified, task-based programming model for heterogeneous systems, including both CPUs and GPUs. The goal of StarPU is to create an API which handles mapping and scheduling, requiring only the programmer to create tasks for the runtime system to schedule (see Section 2.4.6 for more discussion on StarPU itself). StarPU has recently been expanded to handle clusters of nodes using MPI [7].

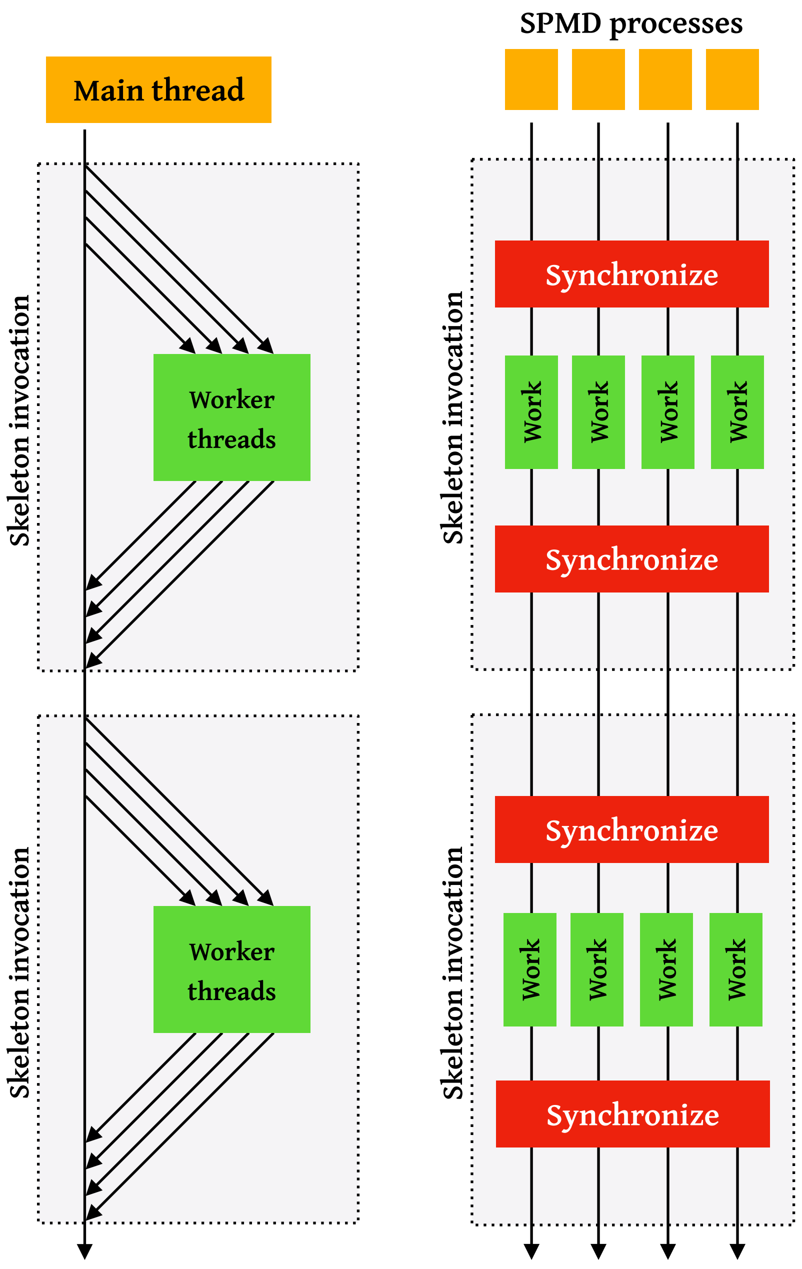
SkePU 3 provides two different modes of using cluster resources:

- **Outer MPI mode**: the application code already contains explicit MPI code for cluster-level parallel execution, using SkePU only locally on each node for execution of skeletons on multicore CPU and/or accelerators.
- **Inner MPI mode**: The application does not contain any MPI (nor other parallelization) code. If an environment for MPI parallel execution is available (usually, multiple nodes on a cluster), then skeletons can transparently execute in parallel across these nodes if selecting the MPI backend.

Outer and Inner MPI mode are mutually exclusive, i.e., for applications that are pre-parallelized using explicit MPI code the MPI backends of all skeletons are disabled.

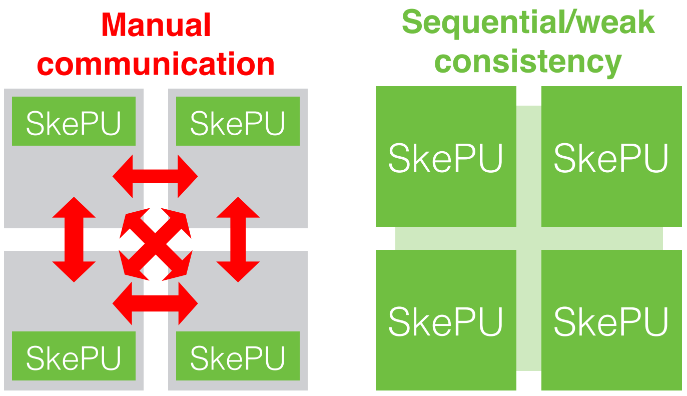
The implementation of inner MPI parallelism is technically based on generating StarPU task code using the MPI interface of the StarPU runtime system [8], which detaches each node's generated send and receive operations into special CPU *codelets* that are exposed to StarPU as separate tasks for dynamic scheduling [7].

[2]The alternative would be designing a new skeleton programming framework targeting clusters from the start, such as Musket [157].



(a) Single-node SkePU (b) Cluster-SkePU

Figure 9.1: Conceptual control behavior of SkePU programs on single nodes and in SPMD on clusters.



(a) Outer MPI (b) Inner MPI

Figure 9.2: Cluster execution modes in SkePU.

Distributed variants of the smart data-containers (`Vector`, `Matrix` etc.) with the same interface as the node-local counterparts come with default distributions, and each cluster node runs one copy of the SkePU executable atop a local instance of StarPU in SPMD style. Execution over distributed container operands follows the ”owner-computes rule”, stating that each node only executes those operations that calculate (write) elements it owns (i.e., are part of its local partition of the result container).

For using inner MPI parallelism, no syntactic changes in SkePU code are required, thus following SkePU’s portability principle. The illusion of a single SkePU process performing all the work on a single node even with the MPI backend is maintained by implementing the `Reduce` skeleton by an MPI Allreduce operation so that the reduction result is available on each of the SPMD processes. The weak memory consistency model of SkePU (see Section 5.3) applies also to distributed containers: the programmer must explicitly `flush` (i.e., gather) them back to the master (i.e., the rank `0` process) before the most recent values of elements of remote partitions can be accessed by a read access on the master, or after a write access by the master.

The remaining issue with SPMD execution is that certain operations (e.g., I/O) need be protected from being executed everywhere. To make sure that such code is executed only by some ”master” process, such code should be guarded by the `skepu::external` construct covered in Section 5.3.1. The distributed data of the operands to `skepu::external` is automatically flushed. Before the code block is evaluated, `read` containers are gathered, and afterwards the `write` containers are distributed using scattering.

Listing 9.1: GASPI notifications.

```
if (node_id == 0)
{
  gaspi_read  (... , node_1, ...);
  gaspi_notify(... , node_1, ...);
}
// Node 0 is now able to perform work without
// needing to wait for a respone from the read
```

Listing 9.2: MPI epochs in one-sided communication.

```
MPI_Win_fence(..., window_1, ...)
if (node_id == 0)
{
  MPI_Get(..., node_1, ...);
}
MPI_Win_fence(..., window_1, ...)
// Any work done by node 0 here has to
// wait for a remote respone from Get
```

9.3 GPI backend

The aim of this contribution is integrate SkePU and GPI in order to help determine whether this integration is likely to work, and whether the performance behavior of an *partitioned global address space* (PGAS)-based backend would be different from that of the StarPU-MPI task-oriented approach. A prototype implementation has therefore been designed and implemented, and then compared to the existing cluster backend in order to analyze its performance and thus evaluate strengths and weaknesses with its design. One aspect which is of particular interest is whether the new prototype overlaps computations and communications, as previous evaluation of GPI required this for good performance [76].

9.3.1 GASPI and GPI

Global Address Space Programming Interface (GASPI)[3] is a PGAS specification created at Fraunhofer Institute for Industrial Mathematics (ITWM) for message passing in clusters. It is similar to MPI in how it is used, as it is neither a language nor an extension of one, but rather library-based, usable within an existing programing language. GPI[4] is an open-source implementation of the GASPI API, also created at ITWM. GPI focuses on *overlapping* computations with communications with its one-sided communica-

[3]http://www.gaspi.de/

[4]https://www.itwm.fraunhofer.de/en/departments/hpc/products-and-services/gpi-programming-model-future-supercomputers.html

tion primitives, in order to reduce the time spent waiting. These primitives differ from MPI in that they may execute at any point in the program lifetime once the communication buffers, called *segments*, have been initialized. GPI does not use a concept similar to MPI's fences and epochs [164], instead the communications are synchronized through notifications. These may be sent by any node, to any other node, and before modifying its segment a node must wait for such notifications. The difference is subtle, but allows for less tightly linked communications; for example, in MPI, if a node performs a read, it must wait for its request to finish before leaving the epoch (Listing 9.2). In GPI, such a wait would not occur unless explicitly stated, as illustrated in Listing 9.1. The effect of this is that GPI allows for many concurrent requests, some of which may span over what would be multiple epochs in MPI. The actual transfers within GPI are based around segments, which are chunks of memory that are globally visible for their group.

The motivation behind the creation of GASPI was MPI's insufficient performance at a large node counts [77]. To achieve this, GPI attempts to overlaps computations and communications while avoiding bulk-synchronous communication patterns. A comparison between the two concluded that, for GPI to outperform MPI, it needs to overlap the communication and computations appropriately [140]. Certain concepts within GPI are clearly aimed at this goal, such as notifications which attempt to reduce the number of global (or group) synchronization points. Furthermore, the concepts of priority queues is used to separate different kinds of local requests. This allows for a more fine-grained control of the communication process, which is important when trying to overlap computations and communications.

9.3.2 Implementation

The GPI prototype only implements a subset of SkePU's features. The selected skeletons are `Map`, `Reduce` and `MapReduce`, combined with the containers `Matrix` and `Vector`. These were selected as they are among the most commonly used constructs within SkePU and together they suffice to solve a relatively large number of problems. Furthermore, by using SkePU's flexible `Map`, it is possible to emulate some other patterns, with suboptimal performance. Furthermore, the prototype does not utilize the SkePU precompiler infrastructure. Recall that any SkePU program is a valid C++ program—this means that any backend which is not strictly dependent on code generation, e.g., for OpenCL kernel code, can be prototyped independently of the precompiler stack.

In this section, we assume that each cluster *node* runs exactly one *process*[5]. While the terms are used interchangeably here, the implementation is generalizable to multiple processes on each node.

[5]With a separate level of thread parallelism utilizing multi-core node architectures.

Listing 9.3: An example of syntax which the GPI prototype is unable to handle.

```
skepu::Matrix<int> m(N , M);

int add_f(int a, int b) { return a + b; }

auto add = skepu::Map(add_f);

// If the random value is greater than 0
if (std::rand() > 0) { add(m, m, m); }

// If the current time is even
if (std::chrono::system_clock::now() % 2 == 0) { add(m, m, m); }
```

9.3.3 Design

The execution model of the prototype matches that of GPI and is SPMD, just like the StarPU-MPI backend. Internally, depending on the construct, the backend may create an OpenMP parallel region to handle the task using multiple threads, reducing parallelism overhead internally on each cluster node. The implementation assumes homogeneity among the nodes: as nothing like the StarPU task management and scheduling layer is present here, the GPI backend is more sensitive to load imbalances. Furthermore, the program structure leads to a sort of determinism which is leveraged to help deduce individual node workloads asynchronously. As long as the user does not access the node IDs, use if-statements with arguments derived from node-local sources, or similar, as shown in Listing 9.3, the control flow of the program is the same for all nodes. Hence, using only its ID and the total node count, a node is able to deduce global information about a SkePU call since every other node will perform the same SkePU call with the same arguments in identical sequence. Allowing for deduction about the other nodes states is important, as it reduces communication and synchronization work.

9.3.4 Synchonization and state tracking

Thanks to the program structure explained above, it is possible to create a global order of every SkePU call, for all the nodes. These calls are then divided into phases, and each phase is given an incrementally increasing and unique operation number. For example, `Map` is divided into a *waiting* phase, an *execution* phase and a *finished* phase, which would correspond to operation numbers N, $N + 1$ and $N + 2$. By using these operation numbers, a node is thus able to deduce the state another node is in currently and which operations it has not yet executed. An important caveat is that a node can only deduce which operations another node has not executed if

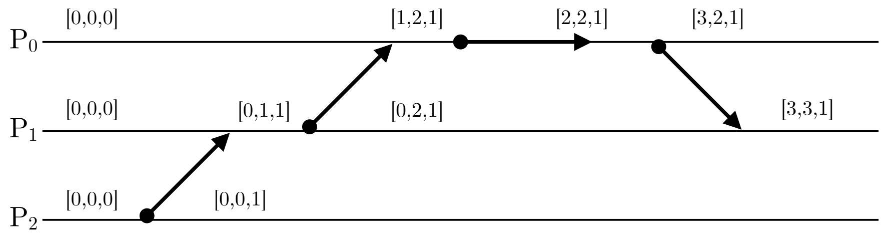


Figure 9.3: Vector clock schema for three processes; each arrow is an event. (Figure adapted from [3].)

it itself has executed more operations than the other one. In other words, it knows which operations a node that is lagging behind will do, but it knows nothing about what a node that has drifted ahead has done. Furthermore, the operation numbers are bound to the nodes themselves, whereas most other parts of the synchronization process are bound to a data-container object. Hence, the operation numbers are used to order the other aspects of the synchronization process, which are primarily oriented around data-container state.

The method for propagating the nodes' different operation numbers is based on a synchronization schema known as *vector clocks* [163] (Figure 9.3). In this schema, every node has its own counter which is incremented whenever an *event* occurs. An event corresponds to either an independent calculation or communication. Furthermore, every node stores a vector of the highest occurred counter for all the existing nodes, including itself. Whenever it then receives a communication event, it updates any values within its vector if the incoming counter at the corresponding index is greater. This results in a causal system where it is possible to see which events occurred before any given state.

The GPI backend's synchronization model is heavily based on the vector clock schema, with the crucial difference that every node has the same local events. This does not mean that every node necessarily performs the same computations, but rather that every node performs the same SkePU operations. What a node does in these phases may vary; for example when indexing a smart data-container, some nodes must fetch the value remotely whereas others need not. As such the values within the prototype's vector clock correspond to certain phases of a SkePU operation.

The synchronization process is based on operation numbers and constraints. A constraint is a tuple containing a node ID and an operation number. At the end of an operation, a node adds constraints to indicate which nodes might read from it during this operation. Then, when the node wants to modify its data, it has to fulfill all of its constraints before it is safe to proceed. A constraint can be said to be fulfilled when the remote node has reached an operation number larger than the constraint. Due to the SPMD nature, a node is able to deduce how the access pattern of an operation is going to look for all the other nodes and set the constraints properly. The primary issue here is the random access proxy-containers available in skeletons such as `Map`. Here the user function determines the access pattern and a node is unable to deduce it without essentially executing the function with the remote data. As such, to ensure correct behavior, every operation with a random access pattern is assumed to read from every other node.

9.3.5 Consistency model and double buffering

The ultimate purpose of the constraints and operation numbers discussed above is to allow for a weak memory consistency. Every node has a double buffer of its local smart data-container partition, which other nodes are unable to read from. Whenever an operation modifies the data, it always writes the changes to the double buffer. To allow remote nodes to access this new data, a node must periodically flush its double buffer and put it into the actual GASPI segment. This is done as sparingly and late as possible by letting nodes deduce whether a remote node might need to read from them during an operation, and only then perform the flush. For example a simple `get(i)` operation will make the node holding i flush its changes, but not any other node. In order to track the changes every node saves the remote state of every other node through the following two fields within the `Matrix` class: the last operation which modified it and the last operation when it flushed its changes. With these fields, every part of the data container is able to deduce at which operation number a remote is safe to read from and when it is not. It also adds fine grained synchronization where parts of a data container may be flushed while others are not. Thus the operation number where the flushed data is accessible may differ depending on which node is being read.

Flushes refer to the process of moving the data from the double buffer into the GASPI segment where it may be read by other nodes. They are done in two cases, firstly if the local node needs to modify its container which already is dirty. Then the existing changes are flushed into the GASPI segment and the new ones applied to the double buffer. The second case is if the local node deduces that another node needs to read data which exists in the double buffer, which is only accessible locally. This architecture thus allows for every node to save two states of its container, and makes it the nodes' responsibility to guarantee that the correct state is available at certain operation numbers. The local nodes do, however, only have knowledge of the operations it has processed so far, and as flushes only are done when deemed necessary that an idle waiting period may occur.

9.3.6 Communication pattern

The benefit of the weak consistency model is that it allows for an infinite drift as long as there are no data dependencies. A node only needs to wait for a remote node if either it has not produced and flushed the needed data, or if it needs to read a local value and is yet to do so. With the states of remote nodes being propagated indirectly through the vector clock and local deductions the expected communication is kept to a minimum. The communications between nodes fall into two categories, polling for information about the remote nodes state and reading remote elements from one of its smart data-containers. In particular, communication across different data containers are never done as a data container is responsible for all communications with its remote partitions. This makes the design easier, as the GASPI segments within every data container only needs to be able to store remote data from other partitions. The dimensions of this segment is thus adapted at these specific communication patterns and nothing else.

9.3.7 Data representation

This section describes how the `Matrix` class is implemented in the GPI backend. It is the largest class in the GPI backend with a wide set of responsibilities, most importantly tracking the state of its local and remote elements as well as providing an accessor interface for them. Furthermore it also works as an alias for the vector class by implementing a few extra functions such as a one-dimensional constructor. The motivation for the aliasing is that the differences between the two is minuscule implementation-wise, and this solution was the most straightforward one while also not replicating any code unnecessarily.

9.3.8 Data transfers and caching

The `Matrix` class is important to the data transfers of the prototype as it manages the GASPI segments, which are the destination and origin of all read requests. The elements of the `Matrix` is stored in a GASPI segment, but this segment also has extra memory allocated for the transfer of remote elements. The size of this segment is the size of the local partition plus the global size of the container. Thus every remote element has its own unique position in the segment at the cost of high memory usage. While the elements in `Matrix` also are stored in the same segment, logically the two parts are kept separate and referred to as the data container segment and communication buffer, respectively. While the data container segment is only ever used for storing local values, the usage of the communication buffer are many and determined by the algorithmic skeleton.

One of `Matrix` most important functions is `proxy_get` which is called from the proxy container dummy class and allows for the access of any element, both local and remote. This class is used by certain skeletons, namely `Map` and `MapReduce`, and its purpose is to provide the user with a random accessing pattern of the elements. `proxy_get` returns a value from the data container, given an index, in a thread-safe manner. Conceptually this function may either return the local or cached value instantly or it may need to transfer it. If this is the case then it also transfers all elements of the remote node and puts them in the communication buffer. Any further readings to this node will now read the cached data.

By leveraging the state tracking fields mentioned in Section 9.3.4, as well as tracking the state of the cache, the values within it may be reused for multiple operations. Thus the transfers are very large but the transferred elements may be reused for multiple operations, limiting the amount of transfers and amortizing the overhead.

9.4 Conclusions

This chapter has presented two approaches for cluster execution of SkePU programs. Both the StarPU-MPI backend and the GPI backend are positioned as abstractions one level above other programming frameworks, which to some extent couples SkePU's performance to that of the underlying library. This is in particular the case with StarPU, which provides dynamic scheduling through its task interface. This gives SkePU load-balancing options not present to this level in other backends, but comes with task management overhead. Several of the evaluations in Chapter 15 are applied to the StarPU backend; the GPI backend has seen limited initial evaluation (Section 15.8). Future work will perform further evaluation and performance analysis of the two approaches.

10 Extending smart data-containers for data locality awareness

This chapter is closely based on the following publication:

August Ernstsson and Christoph Kessler. "Extending smart containers for data locality-aware skeleton programming." In: *Concurrency and Computation: Practice and Experience* 31.5 (2019), e5003. DOI: `10.1002/cpe.5003`

Material from the above paper is ©2018 *John Wiley & Sons, Ltd.* and included in this thesis with permission from the copyright holder.

Experimental evaluation is presented later, in Chapter 15.

In this chapter, we present an extension for the SkePU to improve the performance of sequences of transformations on smart data-containers. By using lazy evaluation, SkePU records skeleton invocations and dependencies as directed by smart container operands. When a partial result is required by a different part of the program, the run-time system will process the entire lineage of skeleton invocations; tiling is applied to keep chunks of container data in the working set for the whole sequence of transformations. The approach is inspired by big data frameworks operating on large clusters where good data locality is crucial. We also consider benefits other than data locality with the increased run-time information given by the lineage structures, such as backend selection for heterogeneous systems.

10.1 Introduction

In data centers and supercomputers, parallelization is taken to a different level. A large number of computer *nodes* are integrated with an interconnection network, processing large amounts of data. In these systems, computational resources typically exist in abundance, while data access is the performance bottleneck. In big data analytics, frameworks with interfaces similar to algorithmic skeletons have been constructed to solve mostly the same problems of programmability, performance, and portability. The MapReduce programming model [51] from Google was the first successful such framework. It gained popularity outside of Google though the open-source implementation *Hadoop*[1]. An evolution of MapReduce is *Spark*[2], like Hadoop open-source and maintained by the Apache Software Foundation.

As data access latency is important in big data scenarios, Spark and related frameworks have developed techniques to optimize data locality and reduce the number of unnecessary loads and stores. However, memory accesses are also an important consideration on the smaller scale of single-chip parallelism, as the memory technology and interfaces have not improved at the same pace as multi-core processors. This is known as the *memory wall* [158]. In this contribution, we apply ideas from big data frameworks to skeleton programming and evaluate the results.

This chapter contains a description of the implementation of lazy evaluation and loop tiling of sequences of skeleton invocations as well as a discussion of application scenarios. Performance evaluation of loop tiling on example applications using the `Map` and `MapOverlap` skeletons is presented in Chapter 15.

We first present an overview of big data technologies in Section 10.2. Section 10.3 details our contribution, lazy evaluation and loop tiling for the SkePU skeleton framework, and its implementation. Section 10.4 compares our contribution to compile-time kernel fusion and presents applications better suited to our dynamic solution. Section 10.5 lists related work.

Performance evaluation results are presented later, in Section 15.3.

10.2 Large-scale data processing with MapReduce and Spark

This section introduces the *MapReduce* and *Spark* programming environments for big data applications. These models share a basic functional interface with algorithmic skeletons, but the trade-offs and design choices are different. In big data contexts, the cost of computation is small compared

[1]`http://hadoop.apache.org`
[2]`http://spark.apache.org`

to the cost of data movement, and as such higher-level programming languages such as Java or Python are typically used.

10.2.1 MapReduce

The MapReduce programming model is designed for large-scale data processing on clusters. It is a high-level model while still being flexible enough to allow any computation to be expressed with a sequence of the fundamental programming construct: the MapReduce block. As the name suggests, this building block has similar semantics to the MapReduce skeleton in SkePU, but there are additional parts to accommodate the cluster scenario. MapReduce can roughly be deconstructed into three steps:

1. **Map phase**
   This step performs a transformation of each key-value pair in the input sequence to zero, one, or several key-value pairs in the output sequence, perhaps in different domains.
2. **Shuffle phase**
   The shuffle phase will shuffle and sort the key-value pairs so that elements with identical keys are located on the same node.
3. **Reduce phase**
   The reduce phase will, for each key, perform some accumulation of the set of values associated with this key.

Due to this separation of computation into super-steps with communication only occurring at well-defined points, the MapReduce model can be considered an implementation of the *bulk-synchronous programming* (BSP) model [120, 152]. Each MapReduce super-step begins with each node reading its assigned subset of the input data from secondary storage, and ends with a write of the output to disk.

10.2.2 Spark

The goal of Spark is to improve upon the performance of MapReduce, specifically in iterative applications such as machine learning [113], by means of avoiding unnecessary reads and writes to the file system [161]. Spark does this by introducing an abstraction called *resilient distributed datasets* (RDDs), read-only collections of arbitrary data partitioned over a set of nodes. Spark classifies the computations that can be done on RDDs into two classes: *transformations* and *actions*. In essence, computations that can preserve the current partitioning (i.e., can be done locally on the residing node) are transformations. These include, but are not limited to, `map`, `filter`, `union`, and `intersection`. A transformation on an RDD always returns a new RDD.

Listing 10.1: Word count program with Spark in Scala.

```scala
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Actions, in contrast, all require some form of collection or reduction of the RDD objects. Typical actions include `reduce`, `count` and `collect`.

The reason for classifying computations into transformations and actions are that operations on RDDs are lazy. The computation is not carried out immediately upon evaluation of a function call, instead recorded into a *lineage graph* associated with the returned placeholder RDD. As long as only transformations are performed, Spark can build up the lineage graph without performing any actual computations.

10.3 Lazily evaluated skeletons with tiling

In this section, we present an approach to apply the idea of lineages and lazy evaluation to skeleton programming.

10.3.1 Basic approach and benefits

Lazy evaluation of skeleton invocations works by, instead of computing the skeleton algorithm immediately at the call site, recording the skeleton, smart container arguments, and the surrounding context. The system will detect dependencies across skeleton invocations and build a graph, a *lineage*, where the nodes are skeleton invocations with recorded contexts and the edges represent dependencies.

A lineage graph is therefore fundamentally tied to the smart data-containers. A single smart container can be used as input or output in multiple nodes of a lineage. As long as the operations on smart data-containers are element-wise transformations, the lineage graph can continue to be built up (otherwise, it is an *evaluation point*, see Section 10.3.4). The lineage graph essentially encodes a partial order of skeleton invocations, where the ordering relation models dependencies and thus is dependent on the containers used as arguments. As the lineage graph contains all dependency information, the physical call order of skeleton invocations is irrelevant and the runtime system is free to execute skeleton invocations in any sequence that is compatible with the dependency-carried partial ordering. The runtime can as such aim to find the sequence that offers the best locality of reference. As will be clear later, evaluations of skeleton invocations will in

Listing 10.2: Before transformation.

```
for i in 1 to N do
  a[i] = a[i] * b[i]

for i in 1 to N do
  c[i] = a[i] + a[i]
```

practice not even follow a strict sequence, as the runtime will interleave different phases on a per-element basis.

With the introduction of lazy skeletons in SkePU, the following areas will all have opportunities for performance improvements, among others:

- **Backend selection** on lineage level instead of single invocation level.
- **Cache-aware** skeleton algorithms with tiling applied to sequences of skeleton transformations.
- **Big data** scenarios, by further applying tiling on secondary storage-aware smart data-containers.
- **Secure smart data-containers**, where data is stored in encrypted form in off-chip memory and is decrypted only when part of the current working set.

10.3.2 Backend selection

The lineage makes more information available to the run-time backend selection mechanism. Instead of greedily applying the tuning parameters to a single skeleton invocation at a time, the backend can be selected for the whole sequence at once. A typical consideration for backend selection is whether the advantage of performing a computation on an accelerator is worth the effort of moving data back and forth. For a sequence of computations, data movement will only happen once in either direction but the computational load will encompass all skeleton invocations in the lineage.

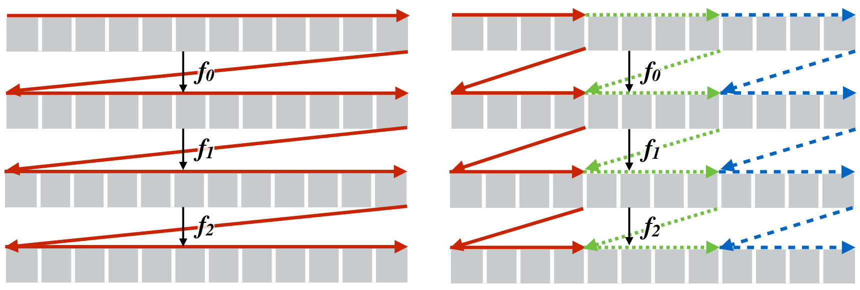
10.3.3 Loop optimization

By collecting information about a series of data transformations (`Map` skeleton invocations) we can apply several loop optimization techniques to improve the temporal locality of reference [52].

Loop fusion is a known technique for low-level compiler optimization. Two or more loops can be combined into one, as in Listings 10.2 and 10.3. However, in the context of skeleton lineages, the overhead of switching between the contexts of different skeleton invocations for every single el-

Listing 10.3: After transformation.

```
for i in 1 to N do
  a[i] = a[i] * b[i]
  c[i] = a[i] + a[i]
```



(a) No tiling (b) Tiling

Figure 10.1: Sequence of skeleton transformations with and without tiling.

ement is too large for it to be practical. Locality can still be preserved, though, if the switching is done in between processing chunks of elements.

Loop tiling (also known as loop blocking or nesting) is a well-known technique for optimizing the locality of reference [92], especially on large, two-dimensional data sets.

Figure 10.1 conceptually illustrates tiling of skeleton lineages. f_0, f_1, and f_2 are skeleton transformations lazily recorded in a lineage. At evaluation time, without tiling, each transformation will be processed completely before moving on (as is the case without lazy evaluation), see Figure 10.1a. With tiling, a chunk of the container space will have all transformations applied before moving on to the next chunk, as in Figure 10.1b. (In practice, there may be multiple containers involved and the destination container may or may not be the same as one of the inputs.)

10.3.4 Evaluation points

In contrast to the read-only RDDs in Spark, SkePU offers much more flexibility in how smart data-containers can be used. One consequence of this is that operations on smart data-containers cannot easily be separated into transformations and action classes, and from there decide when to apply the lazily accumulated skeleton invocations. A smart container's lineage will be evaluated if it is used as an input to `Reduce` or `Scan`, if it is used as a random-access input argument to `Map`, `MapReduce`, `MapOverlap`, or `Scan`, or if individual elements are accessed.

Lineages can also be discarded without evaluation, as skeleton algorithms are semantically free of side effects. This will occur if a smart container is destroyed or reused as output before an evaluation point has been reached.

10.3.5 Further application areas

The proposed technique offers an automatic solution to make skeleton programming more cache-aware. However, the approach is general enough to accommodate any scenario in which there is a substantial cost associated with loading and storing data to and from the current *working area*. In big data processing, the working area is instead the primary memory of a compute node, and the load and store operations are slow network communication or disk I/O.

As security and integrity become increasingly important aspects of computing, we can also envision use-cases where data is permanently encrypted in main memory and only decrypted and moved to a smaller, isolated (and secured by other means) memory area for processing. The cost of decrypting and encrypting data would be significant compared to the typically relatively simple Map transformations.

10.3.6 Implementation

The technique described above has been implemented in SkePU for its `Map` skeleton, and subsequently extended to also include the `MapOverlap` skeleton as described later in Section 10.3.7. For lazy skeleton evaluation, we use C++11 lambda expressions to capture the context of a skeleton invocation (container iterators, uniform arguments, and skeleton settings such as a user-set backend specification). Lineages are graphs of linked nodes, each node containing a function object resulting from the lambda expression as well as dependency information as addresses of the container arguments. The containers themselves need not be captured, as the lifetime of a lineage is tied to the scope of a smart container. Unevaluated lineage nodes will simply be discarded when the associated containers go out of scope.

The syntax for SkePU is unchanged. Lazy evaluation and lineage construction occur automatically, with optional API added for explicit control, such as requesting the evaluation of container. Adding a node to a lineage will create dependencies to *all* nodes with corresponding container arguments, as such, an operation is introduced to remove transitive dependencies from the lineage graph.[3]

An example program can be seen in Listing 10.4. The program contains only `Map` skeletons with various transformations and parameter configura-

[3]This could also be done at node insertion time, but may induce some overhead when the lineage grows large.

Listing 10.4: Program generating a lineage graph when evaluated lazily.

```
// Smart containers

skepu::Vector<float>
    v1(size, 1), v2(size, 2), v3(size, 3),
    v4(size, 4), v5(size, 5), v6(size, 6),
    v7(size, 7), v8(size, 8), v9(size, 9);

// User functions

float add_f(float a, float b)
{
  return a + b;
}

float sub_f(float a, float b)
{
  return a - b;
}

float mult_f(float a, float b)
{
  return a * b;
}

float square_f(float a)
{
  return a * a;
}

// Skeletons

auto add      = skepu::Map(add_f);
auto sub      = skepu::Map(sub_f);
auto mult     = skepu::Map(mult_f);
auto square   = skepu::Map(square_f);
auto copy     = skepu::Map([](float a) { return a; });
auto generate = skepu::Map<0>([](skepu::Index1D index, float start)
{
  return index.i + start;
});

// Transformations

add(v1, v3, v4);
copy(v9, v1);
mult(v2, v1, v3);
square(v1, v2);
add(v5, v5, v1);
add(v5, v5, v9);
add(v6, v7, generate(v6, 5.f));

for (int i = 0; i < 5; i++)
  add(v8, v8, v8);
```

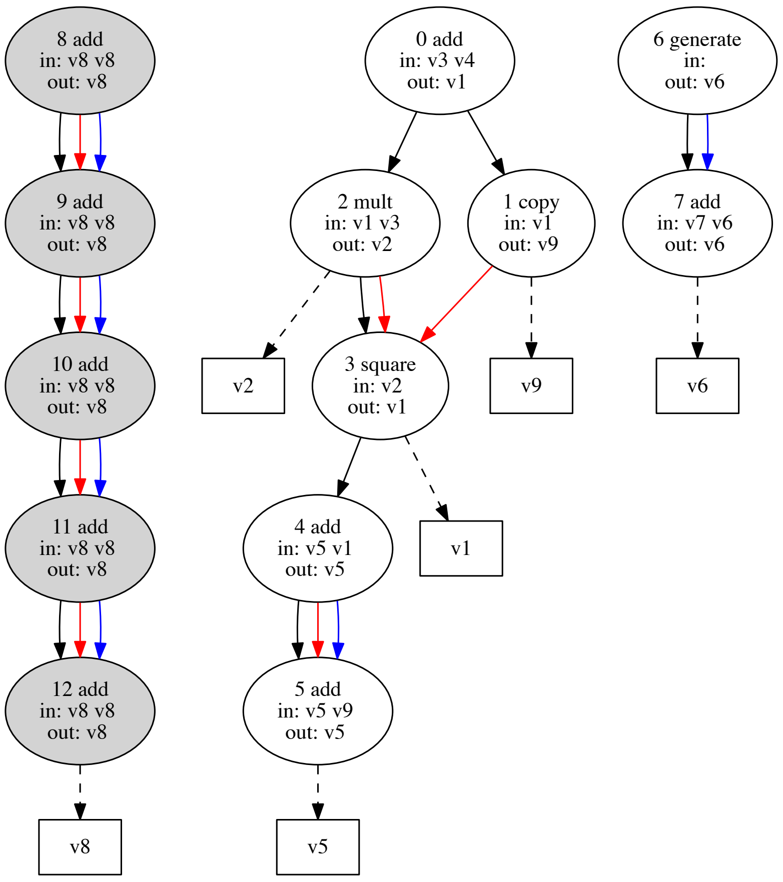


Figure 10.2: Lineage constructed by the program in Listing 10.4.

tions. The resulting lineage graph, combined for all smart data-containers in the program, shows skeleton invocations and dependencies, see Figure 10.2. The graph also shows the starting nodes for evaluating each container. Each node is a recorded skeleton invocation and shows a global incrementing timestamp, skeleton name, input and output container arguments. The directed edges represent dependencies; only the black edges are true data dependencies while red edges indicate write-after-read dependencies and blue edges correspond to write-after-write dependencies. Note that Spark only has true dependencies due to the read-only nature of RDDs. The existence of these false dependencies in SkePU could open up for a smart container "renaming" optimization where possible, similar to register renaming in a microprocessor.

10.3.7 Lazy tiling for stencil computations

MapOverlap is the SkePU skeleton implementing stencil operations on smart data-containers. To extend the lazy evaluation and tiling implementation for MapOverlap instances, care must be taken to ensure that dependencies from the input elements to the output elements are not violated during the lazy evaluation. Naively tiling the computation like Map forms dependencies across tile borders. Eissfeller and Müller [53] proposed the *triangle model* for reducing data transfer times for iterative computations. A variant of that method is implemented in SkePU for tiling MapOverlap skeletons.

We start by observing that there are "safe" regions in the data sets where all of the elements can be computed for each skeleton invocation in the lineage, each tile independently of the others. These regions shrink for each subsequent invocation in the lineage, as the cumulative overlap will increase with each stencil computation. When illustrating the safe regions for each tile and iteration as in Figure 10.3, a triangle pattern emerges. An initial approach for the tiling, which handles the unsafe elements in a separate phase from the non-overlapping regions, is given in Algorithm 1.

Procedure 1 Stencil tiling, first approach

```
Input: Lineage of $n$ dependent stencil computations $i = 0, \dots, n-1$
 1: procedure TRIANGLE
 2:     $B \leftarrow$ tiling block size
 3:     $overlap_i \leftarrow$ overlap for stencil computation $i$
 4:     $indent_i \leftarrow$ prefix sum $overlap_0 + overlap_1 + ... + overlap_i$ for all $i$
 5:     for each tile T, from left to right do                          ▷ Phase 1
 6:         for all instances i, in dependence order do  ▷ Elements in tile $T$ without dependencies from other tiles
 7:             Compute instance $i$ from $BT+indent_i$ to $B(T+1)-indent_i-1$
                                                                        ▷ Phase 2
 8:         for all instances i, in dependence order do         ▷ Remaining elements in tile $T$
 9:             Compute instance $i$ from $BT$ to $BT + indent_i - 1$
10:             Compute instance $i$ from $B(T+1) + indent_i$ to $B(T+1) - 1$
```

Though this first approach can be reworked to improve access locality and reduce the number of skeleton context switches, by merging phase 1 and 2 when possible without violating dependencies. This is used for the implementation in SkePU and is formulated in Algorithm 2.

Procedure 2 Stencil tiling, improved

```
Input: Lineage of $n$ dependent stencil computations $i = 0, \ldots, n-1$
 1: procedure TRIANGLE2
 2:     $B \leftarrow$ tiling block size
 3:     $overlap_i \leftarrow$ overlap for stencil computation $i$
 4:     $indent_i \leftarrow$ prefix sum $overlap_0 + overlap_1 + \ldots + overlap_i$ for all $i$
                                                        ▷ First tile, phase 1
 5:     for all instances i, in dependence order do
 6:         Compute instance $i$ from $0$ to $B - indent_i - 1$
                                                        ▷ Inner tiles
 7:     for each tile T, from left to right do
 8:         for all instances i, in dependence order do
 9:             Compute instance $i$ from $B(T-1) - indent_i$ to $BT + indent_i - 1$
                                                        ▷ Last tile, phase 2
10:     $T \leftarrow$ last tile
11:     for all instances i, in dependence order do
12:         Compute from $BT - indent_i$ to $BT - 1$
```

The approach is exemplified in Figure 10.3. A sequence of three MapOverlap calls, with overlaps of 1, 0, and 2 in that order, is evaluated lazily with a block size of 16. The sequence is computed in three phases, but the size of the region for each phase varies slightly from the block size and also across the different skeleton calls. The darker shaded elements indicates the overlapping, unsafe regions of the data at each point in the lineage sequence.

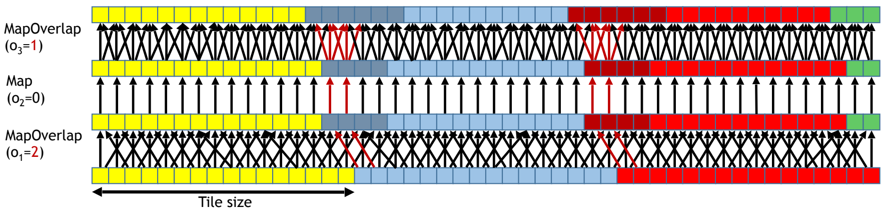


Figure 10.3: Evaluation scheme for tiling sequenced MapOverlap calls.

10.4 Applications and comparison to kernel fusion

Building the skeleton lineage at runtime is a dynamic approach to optimizing sequences of skeleton invocations. The contrasting static approach is to analyze data flow at build time and combine user functions and the skeleton instances they are used in. This approach is known as *kernel fusion* [138].

Listing 10.5: Manual kernel fusion in SkePU where a, b, and c are smart data-containers. User functions are omitted for brevity.

```
float mult_f(float a, float b)
{
  return a * b;
}

float add_f(float a, float b)
{
  return a + b;
}

auto mult = skepu::Map(mult);
auto add = skepu::Map(add);

mult(res, a, b);
add(res, res,  c);


// Fused:

float fused_mult_add_f(float a, float b, float c)
{
  return a * b + c;
}

auto muladd = skepu::Map(mult_add_f);

muladd(res, a, b, c);
```

Kernel fusion is supported in SkePU by manually combining user functions, achievable thanks to the flexibility provided by SkePU skeletons.

The example given in Listing 10.5 illustrates how element-wise multiply-add operations using two binary `Map` skeleton instances can be fused into a single ternary `Map`. Data locality is improved.

Kernel fusion has been implemented in skeleton programming frameworks as an automatic optimization technique where the data flow across skeleton invocations can be determined at compile time. Our proposed tiling approach is more general, applicable to applications with data- or parameter-dependent skeleton sequences. It is also conceivable to combine the two approaches, with just-in-time fusion of kernels based on the lineage.[4]

The following sections contain two applications where the sequence of skeleton invocations are dynamic and data-dependent, exemplifying situations where lineage-based tiling is applicable.

[4]The topic of skeleton fusion as applied to SkePU has been investigated further in later work, presented in Chapter 11.

Listing 10.6: Parallel polynomial evaluation in SkePU. User functions are omitted for brevity.

```cpp
skepu::Vector<float> horner_eval_nonfused(
  skepu::Vector<float> &coeffs, skepu::Vector<float> &x_vals)
{
  size_t degree = coeffs.size() - 1;
  auto mult = skepu::Map(mult_f);
  auto add = skepu::Map<1>(add_f);

  skepu::Vector<float> res(x_vals.size(), coeffs(degree));

  for (int i = degree-1; i >= 0; --i)
  {
    mult(res, res, x_vals);
    add(res, res,  coeffs(i));
  }

  return res;
}
```

10.4.1 Polynomial evaluation using Horner's method

Horner's method for polynomial evaluation is based on rewriting a polynomial

$$p(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n$$

as $p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x)))$ to reduce the number of operations, i.e., exponentiations of x, required for evaluation.

Noticing that the formula is a repeated sequence of multiplications and additions, a data-parallel implementation can be expressed as in Listing 10.6. Manual kernel fusion can be applied to the loop body as in Listing 10.5, but there is a bigger optimization opportunity of improving data locality across loop iterations. Simply fusing two kernels is not applicable here, and the number of loop iterations is dependent on the input data (polynomial degree).

10.4.2 Exponentiation by repeated squaring

Exponentiation by repeated squaring computes the value x^n in $\mathcal{O}(\log n)$ multiplications. For example, x^{10}, naively computed by nine multiplications, can be written as $x^2 x^8 = x^2((x^2)^2)^2$ and by repeatedly squaring x this is reduced to four multiplications. The rewritten form is analogous to the binary representation of x, e.g. $10_{10} = 1010_2$.

This is implemented as a data-parallel SkePU program in Listing 10.7. In this case, in addition to the properties of the program in Listing 10.6, invocations of the `mult` skeleton instance is skipped for loop iterations where

Listing 10.7: Parallel exponentiation by squaring in SkePU.

```
skepu::Vector<float> exp_by_squaring(
  size_t exp, skepu::Vector<float> &x_vals)
{
  auto bitmap = generate_bitmap(exp);

  auto square = skepu::Map(square_f);
  auto mult = skepu::Map(mult_f);

  skepu::Vector<float> res(x_vals.size(), 1);

  for (int i = bitmap.size()-1; i >= 0; --i)
  {
    square(res, res);
    if (bitmap[i] == 1)
      mult(res, res, x_vals);
  }

  return res;
}
```

the corresponding bit in x is 0. The resulting lineage will look very different when varying the exponent.

10.4.3 Heat propagation

A heat propagation algorithm uses iterative stencil computations to find the convergent temperature in some shape. The iterative pattern should fit the lazy evaluation scheme, but for determining the stop condition a reduction on the entire data is typically used to find the *error* (the maximum temperature difference on some point the volume) after each iteration. As the reduction will break the lineage formation and prevent tiling across iterations, the iteration loop can be unrolled. The error calculation is only done after every few iterations. Inside the unrolled iteration loop, there is perfect opportunity for lineage building. The implementation can be seen in Listing 10.8.

With an unrolling factor R, $R - 1$ intermediate data structures are required in addition to the default two. R MapOverlap calls, each with the prior ones output as input, are followed by a MapReduce call to find the maximum error. The reduction causes the lineage to be evaluated at the end of each loop iteration. Figure 10.4 illustrates the lineage building and evaluation in this application.

10.5 Related work

Apache Spark is the primary inspirational source for this work, introducing lineages and related concepts for big-data processing on large dis-

Listing 10.8: Heat propagation, unrolled to enable lineage construction.

```
// Main MapOverlap skeleton
auto kernel = skepu::MapOverlap([](skepu::Region1D<double> a)
{
  double sum = 0;
  for(int i = -a.oi; i <= a.oi; ++i)
    sum += a(i);
  return sum / (a.oi*2 + 1);
});
kernel.setOverlap(1);

// Error calcualtion skeleton
auto max_diff = skepu::MapReduce(
  [](double a, double b) { return abs(a - b); },
  [](double a, double b) { return (a > b) ? a : b; }
);

// Initialize volume with non-uniform values
auto init = skepu::Map<0>([](skepu::Index1D index, size_t size) {
  size_t left = index.i;
  size_t right = size - index.i - 1;
  return (double)(left < right ? left : right);
});

skepu::Vector<double> m0(size), m1(size), m2(size), m3(size), m4(size);

init(m0, size);

int iters = 0;
double error = INFINITY;
while (error > ERR_TOLERANCE)
{
  kernel(m1, m0);
  kernel(m2, m1);
  kernel(m3, m2);
  kernel(m4, m3);
  iters += 4;

  // Lineage is evaluated before the reduction here
  error = max_diff(m3, m4);
  std::swap(m0, m4);
}
```

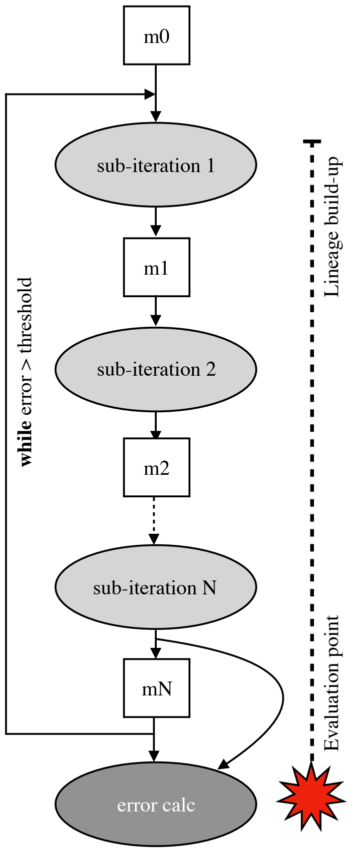


Figure 10.4: Lineage building for iterative heat propagation using MapOverlap.

tributed memory clusters. Spark is a development of MapReduce [51] and the Hadoop implementation. For a related discussion on big data frameworks and the connections to skeleton-like programming paradigms, see [114].

Another very recent project is *FlashR* by Zheng et al.[162]. Matrix operations are evaluated lazily in a memory-hierarchy-aware manner targeting the *R* programming language and SSDs for machine learning applications.

The run-time approach to optimizing skeleton sequences presented in this paper can be contrasted with compile-time techniques, such as the fusion optimizing framework by Sato and Iwasaki [138] for GPU-based systems.

11 High-level skeleton fusion

SkePU exposes a set of core patterns, *skeletons*, which form the building blocks of SkePU programs. Each skeleton can be *instantiated* by parameterizing it with a user function operator, forming a callable object that may be *invoked* at several times in a program. Each invocation of a skeleton instance conceptually forms an atomic computation in the sense that backend selection (mapping) and scheduling are done per invocation. Normally, this selection is done just-in-time, at the point in the dynamic program execution when the corresponding invocation expression is encountered. Without a global view of what skeleton instances are invoked and when, this restricts backend selection and scheduling to locally optimal but globally suboptimal choices at best.

11.1 Comparison to lineages

The first steps towards a more flexible system were taken with the addition of lazy evaluation of skeleton invocations, as described in Chapter 10. Lazy skeleton evaluation works by separating the evaluation time from the invocation point, postponing it for some later time. By repeatedly building up the graphs (DAGs) of invocations, the *lineages*, more information about the dynamic execution environment of the program is revealed to the SkePU run-time system, which can be exploited in backend selection and scheduling. However, because of the global invocation graph may be dependent of the results of those very same computations, the lineage will in general be a sub-graph of the global invocation graph.

The lineage system as implemented by lazy evaluation is a run-time only approach. To extract more information and to more accurately model the global view, utilization of the compiler is required. Program analysis at compile-time (or pre-compile-time) gives information about all possible invocation graphs at the global level, but also due to data dependencies cannot deduce the actual dynamic execution path of the program. However, by combining static analysis and dynamic lineage information, optimizations can be realized that are unavailable from only one of the approaches on its own.

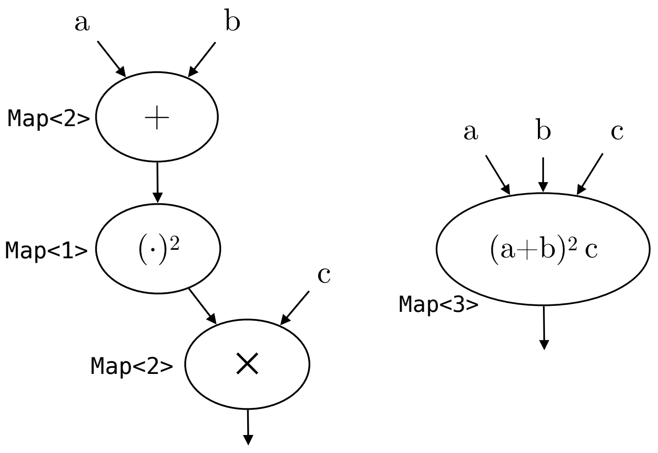
Note that the lineage approach, because of state saving, management, and restoration overhead, is a good fit for multi-core CPU targets but is not scalable to GPU backends.

11.2 Kernel fusion

One such optimization is *kernel fusion*. Conceptually similar to a fundamental compiler optimization, *loop fusion* [84], the idea is to fuse multiple ”kernels” (as the terminology suggests, this optmimization is highly suitable for GPU targets), in the case of skeleton programming, these would be skeleton invocations. In addition to data locality benefits, fusion reduces overhead of GPU kernel launches.

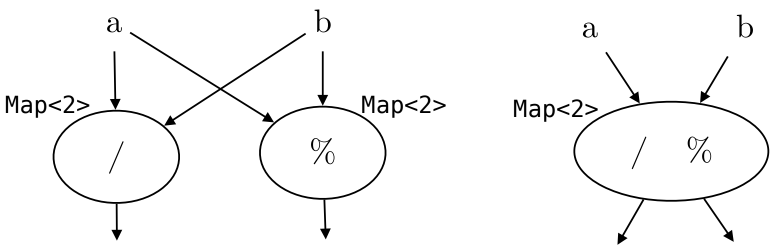
SkePU already provides the fused skeleton patterns `MapReduce` and `MapPairsReduce` in addition to the separate `Map`, `MapPairs`, and `Reduce`; but it is left to the programmer to know about these fused patterns and when to use them. The reason for this are several, and includes reduction of overhead from repeated kernel invocations as well as data access locality improvements. Through static analysis in the compiler, most cases of chained Map+Reduce patterns can be identified, but not all. Situations may occur when e.g. the Reduction portion is guarded by a conditional branch. It may be unlikely to be skipped over in practice, but the compiler must be careful and cannot do anything.

However, automatic kernel fusion is not a new idea [54, 66, 117, 129, 138]; to some pattern libraries the fusion can even be a natural part of the compilation environment. For extended and automatic kernel fusion in SkePU to be a relevant scientific contribution, it would have to do something new or better than the prior work. Since SkePU already has a highly flexible skeleton interface, we think that the idea of *high-level skeleton fusion* can be the foundation for such a contribution in the future.



(a) Before fusion (b) After fusion

Figure 11.1: Skeleton fusion of a sequence of three `Map` skeleton instances.



(a) Before fusion (b) After fusion

Figure 11.2: Skeleton fusion of two `Map` skeleton instances operating independently on the same data set.

11.3 Types of fusions

This section investigates the types of fusions allowed by SkePU's variadic template interface. Three types of fusions are to be considered:

- **Linear chains**, as seen in Figure 11.1, denoted in Table 11.1 with a >> symbol. Sequences of skeleton instances where the output of one instance is the input of the next can be eligible for fusion. Intermediate data sets are eliminated, so this fusion type can reduce memory pressure and utilize temporal access locality.

Listing 11.1: SkePU code demonstrating manual skeleton fusion, corresponding to Figure 11.1.

```
skepu::Vector<int> a(N), b(N), res(N);

// Without fusion
auto s1 = skepu::Map<2>([](int a, int b) -> int { return a + b; });
auto s2 = skepu::Map<1>([](int a)        -> int { return a * a; });
auto s3 = skepu::Map<2>([](int a, int b) -> int { return a * b; });

skepu::Vector<int> tmp1(N), tmp2(N);
s1(tmp1, a, b);
s2(tmp2, tmp1);
s3(res, tmp2);

// With fusion
auto f = skepu::Map<3>([](int a, int b, int c)
{
  int tmp = a + b;
  return tmp * tmp * c;
});

f(res, a, b);
```

- **Independent** computations on partially the same data sets, as seen in Figure 11.2, denoted in Table 11.1 with a `||` symbol. As long as skeleton invocations are independent and applying the same skeleton pattern on equal-sized data sets, control logic and synchronization can be shared between them. When input operands are shared between them, temporal access locality is improved after fusion.

- **Loop-carried chains.** If the same skeleton instance is called repeatedly on the same data set, it can conceptually be unrolled into separate skeleton instances forming a linear chain, and fusion may thus be applicable by the same means as the first fusion type.

Table 11.1 lists all identified types of fusions that can be encoded within the variadic skeleton interface itself. The same relations are visualized in the context of the full skeleton set of SkePU in Figure 11.3. A necessary requirement for linear fusion is that there exists a direct dependency between a pair of skeletons listed in Table 11.1. Further conditions must be met for fusion to be guaranteed applicable, e.g., there can be no dependencies linking intermediate data sets to skeleton invocations outside the fusion. `MapOverlap` fusions must also operate on data of the same dimensionality.

As an example, consider the restriction to only `Map` and `Reduce` skeletons. `Map` fusions utilize the already present variadic `Map` approach in SkePU, and `Map+Reduce` fusions utilize the `MapReduce` skeleton which is among the fused patterns already present in the framework. Note that not

Listing 11.2: SkePU code demonstrating manual skeleton fusion, corresponding to Figure 11.2.

```
skepu::Vector<int> a(N), b(N), res1(N), res2(N);

// Without fusion
auto s1 = skepu::Map<2>([](int a, int b) -> int { return a / b; });
auto s2 = skepu::Map<1>([](int a, int b) -> int { return a % b; });

s1(res1, a, b);
s2(res2, a, b);

// With fusion
audo f = skepu::Map<2>([](int a, int b) -> skepu::multiple<int, int>
{
  return skepu::ret(a / b, a % b);
});

f(res1, res2, a, b);
```

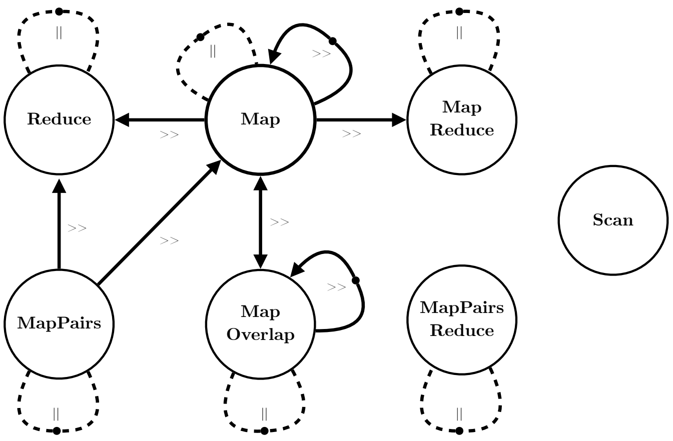


Figure 11.3: A graph illustration of the relationships in Figure 11.1. A directed arrow between two nodes indicates that sequences of such skeletons can be fused. A dashed arrow indicates that independent fusion is possible.

all possible fusions are implemented by the existing skeletons. For example, `Reduce` can operate in either 1D or 2D mode, but `MapReduce` only reduces in 1D, which adds further constraints on when fusion can be applied.

Each fusion type can be iterated and composed, so, e.g., `Map+Map+Reduce` is first matched by fusing `Map+Map` to a compound virtual single `Map` instance. `Map+Reduce` is then matched in the next iteration and replaced by a virtual `MapReduce` node. The reverse fusion order would have the same end result.

Skeleton A	Op	Skeleton B		Result skeleton
Map(*f*)	>>	Map(*g*)	=	Map(g∘*f*)
Map(*f*)	>>	Reduce(*g*)	=	MapReduce(*f*, *g*)*
Map(*f*)	>>	MapReduce(*g*, *h*)*	=	MapReduce(*g*∘*f*, *h*)*
Map	>>	MapOverlap[r]	=	MapOverlap[r]
MapOverlap[r]	>>	Map	=	MapOverlap[r]
MapOverlap[r]	>>	MapOverlap[s]	=	MapOverlap[r+s]
MapPairs	>>	Map	=	MapPairs
MapPairs	>>	Reduce	=	MapPairsReduce*
Map(*f*)	||	Map(*g*)	=	Map(<*f*, *g*>)
MapOverlap[r]	||	MapOverlap[r]	=	MapOverlap[r]
MapPairs	||	MapPairs	=	MapPairs
Reduce(*f*)	||	Reduce(*g*)	=	Reduce(<*f*, *g*>)
Scan(*f*)	||	Scan(*g*)	=	Scan(<*f*, *g*>)
MapReduce(*f*, *h*)*	||	MapReduce(*g*, *h*)*	=	MapReduce(<*f*, *g*>, *h*)*
MapPairsReduce*	||	MapPairsReduce*	=	MapPairsReduce*

*) Need not be part of the interface; internal implementation node only.

Table 11.1: All possible skeleton fusions in SkePU and their resulting skeleton types.

11.4 Example: N-body simulation

Consider the computational core of an N-body simulation program in Listing 11.3. Lines 6-7 are guaranteed to be evaluated in sequence, and each one uses only the outputs of the other as input (except for the very first invocation in the first iteration). They happen to be invocations to the same skeleton instance, but that does not matter for fusion selection (though it can potentially reduce code size of generated fused skeleton instance). This should be a good target for fusing into a single skeleton, reducing the number of kernel calls and selection overhead by a factor of two in the iterative loop. The `particles` container would in practice be the only input and

Listing 11.3: N-body iteration loop in SkePU.

```
nbody_init(particles, cbrt_np);

// Iterative computation loop
for (size_t i = 0; i < iterations; ++i)
{
  nbody_simulate_step(doublebuffer, particles, particles);
  nbody_simulate_step(particles, doublebuffer, doublebuffer);
}
```

only output to the fused skeleton invocation. There is, however, one big problem with this fusion: the skeleton invocations act as global synchronization points. As SkePU user functions cannot contain communication or synchronization, this loop cannot be manually fused. An automatic fusion implementation built into the framework could detect this synchronization constraint and would have to adapt the pattern accordingly after fusion, which brings us to the next section.

Another, fully fusible, example using image filtering operations with accompanying performance evaluation is described in Section 15.11.

11.5 Future work

While manual fusion is a strong feature arising as a consequence of SkePU's flexible skeleton interface, automating the fusion process is of interest for future work. SkePU should not expect that the user knows when manual fusion is critical for performance; an automated fusion system could also remove the need for explicit `MapReduce` and `MapPairsReduce` skeleton constructs, simplifying the API surface of the framework. SkePU's precompiler is well positioned to, through static analysis, identify cases where fusion conditions are met and generate fused skeleton instances.

However, our experience is that such cases are fairly rare in SkePU programs, as the interface makes it natural to find trivial cases of fusion already in the programming process. We therefore propose a direction for future work in an *optimistic* static fusion system, i.e., code generation of fusible invocation sequences that *may* occur during runtime, but cannot be statically guaranteed. Combined with the lazy evaluation system from Chapter 10, target fusions can be disambiguated at runtime, and the lineage evaluator can select from pre-fused skeleton instances for matching invocation nodes.

12 Multi-variant user functions

This chapter is closely based on the following publication:

August Ernstsson and Christoph Kessler. “Multi-variant User Functions for Platform-aware Skeleton Programming.” In: *Proc. of ParCo-2019 conference, Prague, Sep. 2019, in: I. Foster et al. (Eds.), Parallel Computing: Technology Trends, series: Advances in Parallel Computing, vol. 36, IOS press.* Mar. 2020, pp. 475–484. DOI: `10.3233/APC200074`

Experimental evaluation is presented later, in Chapter 15.

This contribution extends the multi-backend approach of SkePU by providing the possibility for the programmer to provide additional variants of user functions tailored for different scenarios, such as platform constraints. This chapter introduces the overall approach of multi-variant user functions, provides several use cases including explicit SIMD vectorization for supported hardware, and evaluates the result of these optimizations that can be achieved using this extension.

12.1 Introduction

The core contribution of this work is a generalization of the variant selection mechanism for the skeleton programming framework SkePU, where the problem-specific, sequential user code used to customize a skeleton at skeleton instantiation can be provided in several variants, some of which might even be platform-specific. This is done in a general-purpose programming environment, which differentiates the approach from existing domain-specific variant selection [64]. Our work is also tightly integrated with a platform modeling system [85] allowing build-time lookup of eligible variants going beyond only algorithmic choice or minor variations in performance tuning parameters. The approach is powerful and flexible enough to allow selection based on hardware architecture, levels of heterogeneity, software installations, and more.

The idea and implementation of the core contribution is presented in Section 12.2, followed by several use cases in Section 12.3.

12.2 Idea and implementation

There are multiple scenarios where a user function with a singular definition can be too restrictive for the purposes of performance: use cases include algorithms with different tradeoffs in time complexity versus memory complexity (some platforms may have very limited memory space available per execution thread), instruction set architecture differences such as native double or half precision floating point arithmetics, the existence of SIMD vector instructions, or other hardware-accelerated implementations of common computations. Since these attributes are *constrained* on the underlying platform, the software-defined code variants must somehow be declared compatible only with the appropriate hardware configurations. For this we employ a combination of *language attributes*, annotations at source-code level that are recognized by the SkePU source-to-source compiler, in addition to the *platform description language* XPDL [85].

A platform description (such as the one given in Listing 12.1) is supplied to the SkePU source-to-source compiler and depending on the attributes in the model, user function variants are either included or removed from the resulting program. In this example, the user function variant in Listing 12.3 requires the *Intel AVX* extension to the instruction set. The list of variants for each user function and their prerequisites for inclusion are declared in a *manifest file* (example given in Listing 12.4). Here XPDL metaprogramming queries or other statically evaluated expressions can be used. As the model in Listing 12.1 declares the platform to support this extension (line 7 in Listing 12.1), this vectorized variant will be included for variant selection at run-time. In cases where library or binary compatibility is not required for the extension, this filtering of eligible variants can also happen at run-time, as long as the XPDL model is available for querying. This approach is preferred when a single program executable might run on different hardware configurations.

User function variants are defined externally from the main source file. The variants are placed in individual source files in subdirectories, following

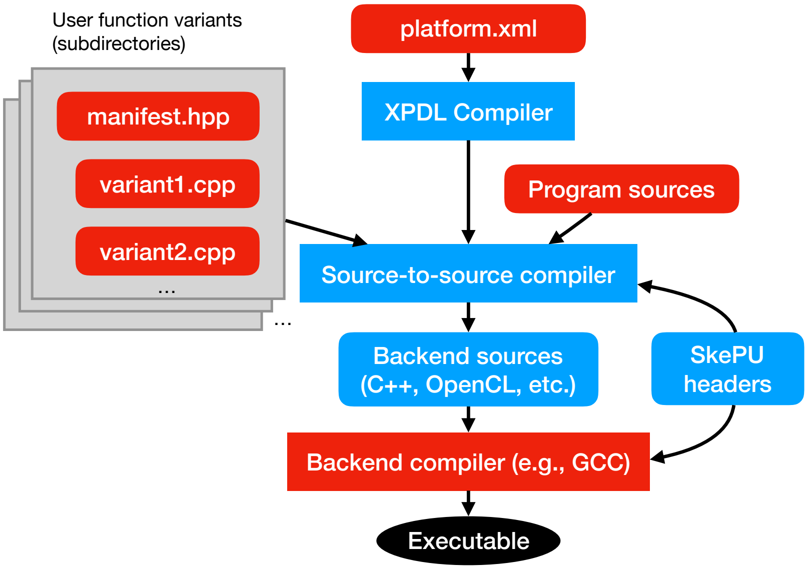


Figure 12.1: Overview of the components involved in SkePU variant selection and subsequent build process.

Listing 12.1: XPDL model for an Intel Xeon Gold 6130 CPU. Please refer to XPDL publications [85] and documentation for details about the syntax.

```
<?xml version="1.0" encoding="UTF-8"?>
<xpdl:model xmlns:xpdl="http://www.xpdl.com/xpdl_cpu"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.xpdl.com/xpdl_cpu xpdl_cpu.xsd ">
  <xpdl:component type="cpu" />
  <xpdl:cpu name="Intel_Xeon_Gold_6130" num_of_cores="16"
    num_of_threads="32" isa_extensions="avx avx2">
  <xpdl:group prefix="core_group" quantity="16">
    <xpdl:core frequency="2.1" unit="GHz" />
    <xpdl:cache name="L1" size="32" unit="KiB" set="16" />
    <xpdl:cache name="L2" size="1" unit="MiB" set="16" />
  </xpdl:group>
  <xpdl:cache name="L3" size="22" unit="MiB" set="1" />
  <xpdl:power_model type="power_model_Gold_6130"></xpdl:power_model>
  </xpdl:cpu>
</xpdl:model>
```

a standard naming schema, with one directory for each user function. A *component implementation descriptor* file defines the hardware platform and run-time requirements for each variant. See Figure 12.1 for an illustration of the workflow: the outlined rectangles denote directories in the file system and the filled rectangles represent files.

Listing 12.2: A SkePU program performing element-wise vector addition.

```
float add(float a, float b) { return a + b; }

int main(int argc, char *argv[])
{
  const size_t size = N; // multiple of 8
  auto vector_sum = skepu::Map(add);
  skepu::Vector<float> v1(size), v2(size), res(size);
  vector_sum(res, v1, v2);
}
```

12.3 Use cases

In this section we present two use cases in detail: user function vectorization and multi-variant components with the `Call` skeleton. We also provide further examples for application of multi-variant components at the end of the section.

12.3.1 Vectorization example

As an example of where user function variants are applicable, consider instruction set extensions for SIMD vectorization. These extensions allow the processor to compute the same instruction in parallel over multiple data items, even from a single thread. Many compilers today are *auto-vectorizing* [93, 98, 104], but this optimization requires a number of preconditions to be satisfied, such as the correct data alignment and no pointer aliasing; and even then, additional compiler flags are often required. For a high-level parallel program such as a SkePU application, aggressive inlining and loop unrolling must also be applied by the backend (external to SkePU) compiler before there is even an opportunity for auto-vectorization.

For the aforementioned reasons, vectorization is a good motivational use case for multi-variant user functions. Consider the SkePU program in Listing 12.2. The program performs element-wise addition of two vectors using the SkePU `Map` skeleton with arity 2. The user function `add` is trivial, with two inputs (one from each vector) and the function body returning the sum of the two elements. This user function is straight-forward for the SkePU source-to-source compiler to handle when generating output for all backends: sequential CPU, OpenMP, CUDA, and OpenCL; it is just a matter of copying the function body. However, by this approach, the CPU backends will not be guaranteed optimal performance in the case of the hardware platform supporting SIMD ISA extensions. As such, it makes sense to provide a variant of `add` and make it available for run-time selection.

Listing 12.3 contains a variant of `add` that is defined in a separate file as outlined in Section 12.2. This file is referenced from the manifest, as seen in

Listing 12.3: Variant of the `add` user function with explicit vectorization.

```
#pragma skepu vectorize 8
void add(float* c, const float *a, const float *b)
{
  __m256 av = _mm256_load_ps(a);
  __m256 bv = _mm256_load_ps(b);
  __m256 cv = _mm256_add_ps(av, bv);
  _mm256_store_ps(c, cv); // return by pointer
}
```

Listing 12.4: Manifest file for user function `add`.

```
skepu::VariantList {
  skepu::Variant("add_avx",
    skepu::Requires(
      xpdl::includes<xpdl::cpu_1::isa_extensions, xpdl_avx>::value
    ), skepu::Backend::Type::CPU
  )
};
```

Listing 12.4. In this case, there needs to be a *block* of eight elements available for the function to enable the use of SIMD instructions, which is different in signature to the default variant.[1] This variant uses compiler intrinsic functions which map directly to Intel AVX instructions. The elements in this variant are passed and returned by pointer, and the component implementation descriptor contains the specification of how many elements it accepts in one block (here illustrated by an inline `pragma`). The elements in the array have to be copied to intermediate vector registers before computation.

12.3.2 Generalized multi-variant components with the Call skeleton

The version 2 revision of SkePU [62] introduced an atypical skeleton construct known as `Call`. The Call skeleton, unlike all other skeleton constructs in SkePU and other typical skeleton programming libraries, does not encode a computational pattern, but rather is an entry point for a self-contained *component* for arbitrary computations. This construct is highly useful in SkePU for two main reasons: firstly, not all computations can be efficiently expressed as data-parallel algorithms, which is the type of patterns present in SkePU, and it is desirable to let generic computations integrate with the smart container and backend selection and tuning systems within SkePU.

[1]The need for framework support in this example is not a universal trait; user function variants can be defined with the same signature and even without any required platform constraints.

Secondly, the optimal way to structure computations is in general different for different parallel backends; there needs to be a way to provide *variants* also for these non-skeleton computations.

A common class of computations that fit the above criteria are sorting algorithms. Another example is the fast Fourier transform (FFT) [159], which has several highly optimized implementations available at library level. In cases such as FFT, an instance of `Call` can be instantiated with a naive sequential FFT algorithm as the default user function, and additional user function variants are specified as shown in Figure 12.1 and implemented as thin wrappers over libraries such as FFTW for CPU and CuFFT for Nvidia GPUs. Both the backend type and the presence of libraries in the target system is specified and taken into account for variant selection.

12.3.3 Other use cases

There are a number of other use cases for when multi-variant user functions can be useful for improving performance portability. Below are some suggestions: The user can specify a hand-optimized user function variant to be used only with a certain backend, such as CUDA (declared via the platform attribute in the user function's component implementation descriptor), while the generic auto-generated user function is used for all other backends. Even within the same backend and the same platform constraints, complex user functions may offer multiple variants implementing the same computation by different algorithmic approaches. Selection between the variants can be controlled by input size and shape, as well as other run-time properties such as idle resources and memory pressure. See e.g. the CellSort sorting algorithm [72] where the algorithm used is closely coupled to the characteristic architecture and instruction set of the Cell processor. When SkePU skeletons are invoked from a language other than C++, components that have a variant defined for that language would have lower overhead due to bridging and data representation and would open up for improved compiler optimization.

12.4 Related work

High-level parallel programming using skeletons or patterns [25] allows to model semantics as well as parallelization-relevant properties (such as type of parallelism, data access pattern, data locality constraints) of a computation using special predefined generic constructs (called skeletons or patterns) at a level of abstraction that is clearly above that of source code (such as OpenMP, OpenCL or CUDA). Existing skeleton programming frameworks include SkePU [55, 62], FastFlow [2], Marrow [106], GrPPI [135], Thrust [13] and others.

None of these skeleton programming frameworks considered automated, platform-specific operator specialization for multi-element groups in skeleton instantiations or calls. Lift, [147] on the other hand is a framework consisting of a functional pattern-based programming language, a compiler and an intermediate representation with pre-defined skeleton-like constructs for the hierarchical, functional modeling of data-parallel computations. It allows for (cost-model directed) rewriting of Lift IR trees by a design space exploration process to automatically take into account platform-specific structures such as SIMD operations, data transfers and data layout transformations, which can be expressed by OpenCL-specific constructs. While Lift is more general than our method, it requires the programmer to specify skeleton instances as a hierarchically nested functional decomposition of multiple primitive operators. In contrast, our approach is based on the simpler SkePU programming API, which is more high-level and does not require special tooling nor automated design space exploration nor an explicit intermediate representation.

PetaBricks is another framework which also exposes algorithmic variant (”choice”) selection [5, 128]. In contrast to SkePU, PetaBricks is task-oriented with a more involved run-time scheduling system, and does not integrate a platform modeling subsystem into the toolflow.

It is also possible to take a more domain-specific approach. *SLinGen* [144] is a generative programming environment for linear algebra which outputs optimized C code, including optional vectorization driven by intrinsics. The *Cl1ck* system for matrix computations [64] focuses on generating multiple alternative application variants for a single operation.

The limitations of compiler auto-vectorization are explored by Larsen et al. [93] who also suggest improvements to the programming language and environment to facilitate the optimization in more scenarios.

13 A deterministic portable parallel pseudo-random number generator

This chapter is based on the following publication:

August Ernstsson, Nicolas Vandenbergen, Jörg Keller, and Christoph Kessler. “A Deterministic Portable Parallel Pseudo-Random Number Generator for Pattern-Based Programming of Heterogeneous Parallel Systems.” In: *International Journal of Parallel Programming* (). To appear.

Experimental evaluation is presented later, in Chapter 15.

13.1 Introduction

Many scientific applications, such as Monte Carlo simulations, use pseudo-random number generators (PRNGs) as part of the computation. In the interest of correctness and debugging, deterministic parallel or heterogeneous execution of such a program that remains consistent with sequential execution also in terms of generated random numbers is a desirable property, which however requires a *deterministic* parallel pseudo-random number generator. This becomes a challenge with SkePU’s design of late decision about sequential, parallel or accelerator execution.

In this chapter, we present the principle, API and implementation of a deterministically parallelized portable PRNG extension to SkePU that exhibits the same behavior regardless where and with how many resources a SkePU program is executed. Our deterministic PRNG parallelization also relaxes the implicit dependence structure of applications using the PRNG. We show that the implementation is scalable on both multi-core CPU and GPU resources, and hence supports the universal portability of SkePU code even in the presence of PRNG calls. It also leads to more compact source code. Core contributions are the determinism and the high-level language integration of our approach. While our solution is prototyped and evaluated in SkePU, where it is important due to the execution unit of a skeleton call being statically unknown, the approach could be adapted and integrated into other frameworks for high-level portable pattern-based parallel programming.

The remainder of this chapter is organized as follows: Sections 13.2 and 13.3 introduce background about determinism and parallel random number generators, shows two motivating examples of previous workarounds used with SkePU to achieve deterministic parallel PRNG behavior, and discusses their drawbacks. In Section 13.4, previous efforts to implement and utilize pseudo-random numbers in SkePU are examined. Section 13.5 explains three fundamental parallelization methods for PRNGs and presents the new API and implementation of the new built-in deterministic parallel PRNG in SkePU. Section 13.6 discusses related work.

13.2 Determinism in heterogeneous parallel computing

In general, a program is called *deterministic* if it always yields the same result for identical inputs (see e.g. Lu and Scott [101] for other possible definitions of deterministic parallel programs). For most programs, determinism is a necessary condition for correctness: we would expect the program to behave consistently over several runs or something is wrong. Sometimes, however, nondeterminism may be part of the program design, such as in Monte Carlo simulations. Nondeterminism may be introduced into a program if it is dependent on external or environmental conditions (and this is not considered part of the program input). For example, the program may access the current time or assume a certain structure of the host file system. The use of pseudo-random-number libraries does not typically introduce nondeterminism alone, as these are dependent on a *seed* which may, e.g., come as part of the program input or by polling the system clock. System libraries may also expose sources of randomness through high-entropy hardware noise [11].

In a parallel programming context, *race conditions* can act as a source of nondeterministic behavior. Race conditions occur when several program threads (or processes, etc.) are not properly synchronized, and the order of thread operations determines how the program behaves. Avoiding race conditions and related phenomena is a significant reason why parallel programming can be challenging and motivates the use of high-level frame-

Listing 13.1: SkePU program demonstrating the possibility of nondeterministic behavior across backends.

```
#include <skepu>

float add(float lhs, float rhs) { return rhs + lhs; }
auto fold = skepu::Reduce(add);

int main(int argc, char *argv[])
{
  size_t N = atoi(argv[1]);
  auto spec = skepu::BackendSpec{argv[2]};
  skepu::setGlobalBackendSpec(spec);

  float el = 1 / ((float) N);
  skepu::Vector<float> vec(N, el);

  float res = fold(vec);
  std::cout << "Result: " << res << "\n";
}
```

Listing 13.2: Test runs of the program in Listing 13.1.

```
% ./parallel/nondeterminism 100000000 CPU
Result: 0.25
% ./parallel/nondeterminism 100000000 OpenMP
Result: 0.88587
% ./parallel/nondeterminism 100000000 OpenCL
Result: 1
```

works. The parallel patterns of a framework such as SkePU abstract away the synchronization challenges for the user, who can assume that the pattern implementations are deterministic and race-free.

However, there is another source of nondeterminism when parallel programs are expected to adapt to a changing execution environment. This may be as simple as altering the number of program threads, but is especially notable in multi-backend execution on heterogeneous environments such as heterogeneous CPU-GPU systems. Floating-point number representations used widely, especially in high-performance contexts, are imperfect and operations on them lack several key properties of real numbers [73]. For example, the associative property of addition does not hold. Associativity is assumed in parallel computations of reductions in, e.g., SkePU. Listing 13.1 contains an example SkePU program where the result is not deterministic across different backends.

Listing 13.2 demonstrates the result of the same computation on different backends.[1] From the result, we can guess that the sequential CPU

[1]Note that the expected result (in real number arithmetic) is 1, and if the `float` declarations are changed to `double`, the CPU and OpenMP backends both produce the expected value.

backend adds together the small values linearly, and eventually the accumulator value is so large that the dynamic range difference renders further additions meaningless. The parallel backends seem to be using one accumulator variable per thread, which are all of the same order of magnitude when the accumulators are added up.

It can be argued whether the nondeterministic floating point behavior discussed here is the fault of the parallel framework or of the underlying backends. It should, in either case, be clear that it is desirable for a high-level parallel program to be deterministic across backends as far as possible.[2]

13.3 Parallel pseudo-random number generation

A *pseudo-random number generator* is a finite state automaton. Each time it is invoked, its *output function* computes and outputs a pseudo-random number in a pre-defined range from the current inner state, and transitions the inner state via the state transition function (also called *update function*) into the follow-up state. The generator only receives input upon the time of seeding, when the seed is processed by the initialization function to produce an initial inner state. Thus, the generator only has a very limited amount of randomness, which is stretched over many outputs, i.e. pseudo-random. Still, current generators pass statistical tests such as Diehard. The complexity to achieve this may lie in the output function and/or the update function. For a complex output function, the update function can be as simple as a counter [86].

If an output of m bits is produced, the inner state comprises more than m bits. The state transition function mostly is non-bijective[3]. Thus, the state graph of the PRNG comprises one node for each state x, and a directed edge $(x, f(x))$ for the transition from x to its follow-up state $f(x)$, assuming f as the state transition function. Thus each node has an outdegree of exactly 1, but the indegree can vary. An example state graph is shown in Figure 13.1.

Flajolet and Odlyzko [67] investigated the expected structure of state graphs if all possible transition functions are equally likely. The graph falls into a small number of weakly connected components, of which one comprises the majority of the nodes (about 75%). Each component comprises a cycle with a number of trees directed towards the cycle, where the largest tree is expected to comprise 50% of all nodes. The expected length of the longest cycle is less than $2\sqrt{N}$, where N is the number of nodes, i.e. quite short. Trees are ragged with depth about $\sqrt{N}$.

[2]The rest of this chapter works to address the issue of determinism in parallel pseudo-random number generators, and solving the problems of floating point computations is left for future work.

[3]A notable exception is the linear congruential generator with transition function $f(x) = ax + b \bmod N$ for a, b chosen such that the period is maximum [87], e.g. a prime and $b = 0$.

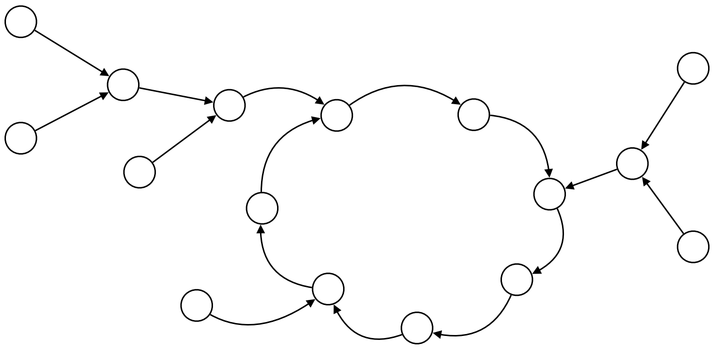


Figure 13.1: State space of a pseudo-random number generator.

The sequence of generated pseudo-random numbers is only dependent on the seed. In a sequential program with a deterministic program flow, the calls to the pseudo-random number generator will produce exactly the same numbers at the same program place if the seed is fixed. If the program is parallelized, then the PRNG state becomes a shared resource. Moreover, the order of calls to the PRNG changes: consider e.g. a nested loop with one call per iteration of the inner loop, where the outer loop is parallelized, so that now the first iterations of all instances of the inner loop call the PRNG first. Still, a deterministic parallel execution with results similar to the sequential version (and independent of the number of threads used to parallelize the outer loop) demands that the sequence of PRNG outputs for each inner loop execution remains unchanged, e.g. to do debugging in the sequential version when the parallel version has an error. This calls for a deterministic PRNG implementation as part of the parallelization.

13.4 Previous manual parallelization of PRNG in SkePU programs

With previous versions of SkePU, a deterministic parallel random number generator behavior had been achieved by the two workarounds described in the following. However, we will show that both have drawbacks.

13.4.1 Monte Carlo pi calculation—index-based scrambling

As a first example, we consider a simple Monte Carlo simulation, namely probabilistic Pi approximation. This computation can be easily expressed as a `MapReduce` instance, see Listing 13.3, where the user function needs to generate two pseudo-random numbers, one per dimension. Here, a deterministic parallel PRNG was simulated by an *index-scrambling technique*, i.e., generation of pseudo-random numbers does not follow the automaton-

Listing 13.3: Ad-hoc deterministic pseudo-random number generation by index scrambling in a Monte Carlo method for Pi calculation in SkePU.

```
#include <iostream>
#include <skepu>

// Define c, s, s2, s3, MY_RAND_MAX as preprocessor constants

float scramble(int in)
{
  return ((((int)(10*s*s2*in + 4*c*s3 + 5*in + 10*s*in)) % MY_RAND_MAX)
          / ((float)MY_RAND_MAX));
}

float monte_carlo_sample(skepu::Index2D index)
{
  float x = scramble(index.row);
  float y = scramble(index.col);
  // check if (x,y) is inside region:
  return ((x*x + y*y) < 1) ? 1.f : 0.f;
}

float add( float lhs, float rhs ) { return lhs + rhs; }

int main(int argc, char *argv[])
{
  auto montecarlo = skepu::MapReduce<0>(monte_carlo_sample, add);

  const size_t samples = atoi(argv[1]);
  montecarlo.setDefaultSize(samples, samples);

  float pi = montecarlo() / (samples * samples) * 4;
  std::cout << pi << "\n";
}
```

based best-practice technique described above; instead, they are calculated independently of each other based on a transformation of the index in the parallelized main loop. In the code example in Listing 13.3, the `scramble` function itself has been extracted from a SPH (Smoothed Particle Hydrodynamics) simulation code. The drawback of the index scrambling method is that it may not really produce random numbers of high quality but can expose more regular patterns.

13.4.2 Markov Chain Monte Carlo methods in LQCD—PRNG with explicit state

The code excerpt in Listing 13.4 is extracted from a Lattice QCD mini-application which computes the Yang-Mills theory of the $SU(3)$ group. This computation is typically done by applying the Metropolis algorithm, a common Markov Chain Monte Carlo (MCMC) based method. The Metropolis calculations are performed on a 4D tensor whose elements are structures of complex number arrays, with a 81-point ($3\times3\times3\times3$) stencil computation required to evaluate the Metropolis acceptance function. For an in-depth introduction to MCMC methods in LQCD, see [80].

Unlike the conventional Monte Carlo method showcased in Listing 13.3, MCMC methods are inherently sequential. Thus, a PRNG for MCMC methods has to be stateful, i.e. a finite state automaton as outlined in Section 13.3. This conflicts with the requirement that SkePU user functions must be side-effect free. The chosen solution for the user functions of Listing 13.4 is a sequential PRNG which is algorithmically equivalent to POSIX `drand48` but has an explicit state argument instead of `drand48`'s internal state variable.

For such an approach, the PRNG state has to be explicitly managed. As a dedicated PRNG state container is not a viable solution due to syntactical constraints of the `MapOverlap` skeleton, the state is embedded directly in the data set. This has the drawback of having an unusually large memory footprint for a PRNG. Specifically, the memory requirement for storing the PRNG states grows by $O(L^4)$ where L is the side length of the 4D tensor, i.e. linearly with the problem size. Usage of the proposed new library PRNG inside SkePU is expected to lower the memory footprint of PRNG state storage to $O(p)$ where p is the number of computational units used in the selected backend.

While it would be possible to adapt the index-based scrambling technique of Listing 13.3 to perform the initial seeding of the resulting parallel PRNG, Listing 13.4 contents itself with using the `Scan` skeleton to force a non-repeating state set into existence. While this is viable as a quick and dirty solution to deterministic parallel PRNG seeding, it is likely to produce random numbers of suboptimal quality; in that respect, a mathematically robust library solution is preferable.

13.5 Designing a deterministic PRNG for SkePU

We will now introduce a more systematic approach that provides deterministic parallel random number generation for use in SkePU, together with an API extension of SkePU 3 that makes PRNG streams a fundamental part of the API. We will start by discussing inherent challenges to pseudo-random number generation in parallel programming and proceed step by step towards a deterministic PRNG implementation at the framework level.

13.5.1 Global synchronization

A straightforward approach to random number generation in parallel applications is to consider the PRNG as a shared resource. As such, the PRNG needs to be protected by the appropriate synchronization operations during access, to avoid race conditions such as multiple threads reading the

Listing 13.4: Simplified SkePU code for an explicit-state parallel PRNG for Markov-chain-based LQCD applications (Section 13.4.2).

```
typedef uint64_t PRNGState;

// Seeding:
skepu::Tensor4<PRNGState> ones(L, L, L, L, 1), prngs(L, L, L, L);
auto seedPRNGs = skepu::Scan([] (PRNGState x, PRNGState y) { return x+y;});
seedPRNGs(prngs, ones);

// Extracting:
inline PRNGState statelessDrand48(PRNGState prng)
{
  return (0x5deece66d * prng + 11) % (1LL<<48);
}
inline double normalize(PRNGState prng)
{
  return (double)prng / (double)(1LL<<48);
}

// Parallel state management:
struct localGauge; // 36 double-precision complex numbers
struct localGaugeAndPRNG
{
  localGauge data;
  PRNGState prng;
};
skepu::Tensor4<localGaugeAndPRNG> gaugeField(L, L, L, L);

// Gauge randomization:
localGaugeAndPRNG randomizeGauge(PRNGState prng)
{
  localGaugeAndPRNG gaugeNew;
  for (int idx = 0; idx < 36; idx++) {
    prng = statelessDrand48(prng);
    gaugeNew.data.at(idx).re = normalize(prng);
    prng = statelessDrand48(prng);
    gaugeNew.data.at(idx).im = normalize(prng);
  }
  gaugeNew.prng = prng;
  return gaugeNew;
}

// Metropolis step:
localGaugeAndPRNG localUpdate(skepu::Region4D<localGaugeAndPRNG> stencil)
{
  localGaugeAndPRNG proposal = randomizeGauge(stencil(0,0,0,0).prng);
  double limen = someDeterministicStencilArithmetic(stencil, proposal);
  stencil(0,0,0,0).prng = statelessDrand48(proposal.prng);
  if (normalize(stencil(0,0,0,0).prng) >= limen) {
    stencil(0,0,0,0).data = proposal.data;
  }
  return stencil(0,0,0,0);
}

auto metropolisUpdate = skepu::MapOverlap(localUpdate);
for (int iter = 0; iter < Niter; iter++) {
  metropolisUpdate(gaugeField, gaugeField);
}
```

same random value, which would decrease the quality of the random number stream, or even the PRNG state itself being corrupted due to simultaneous writes.

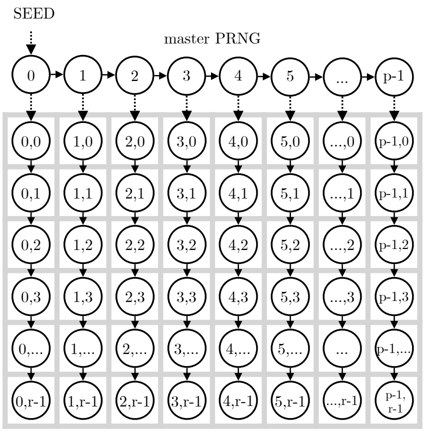
This approach ensures a high-quality random number stream as each value is generated in the same manner as in a sequential program. Any random number generator can be used in this approach, including external entropy sources, since synchronization guarantees protected sequentialized access. This synchronization does however add significant overhead and is unfeasible in massively parallel accelerators such as GPUs. Only if the synchronization method guarantees a deterministic order of accesses to the critical section containing the PRNG state (which is usually not the case for ordinary lock-based synchronization), the random number stream generated from this method will be itself deterministic. We cannot predict in which order the threads will generate a value from the PRNG and update the state space.

13.5.2 Stream splitting

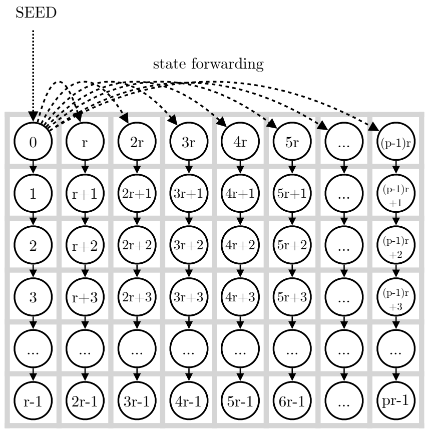
With the goal of avoiding or minimizing global synchronization of the PRNG state, we consider a different approach [69]. As a PRNG state has to be considered a shared resource for proper operation, we can get around the synchronization requirement by assigning each individual thread its own PRNG state. A thread-private PRNG stream does not need protected access and will yield a perfect sequential series of random values by itself. However—aside from a large increase in memory space consumed by the replicated states—with several or many parallel threads in the system, the aggregate random number stream over all task invocations will differ greatly from a sequential program.

Whether data-parallel tasks are assigned in blocks or interleaved, we effectively have *split* the single PRNG stream into many shorter sequences distributed over the working set in the same pattern as the data-parallel tasks. The resulting pattern can be seen in Figure 13.2a. This degrades the quality of the random values in aggregate, which is undesirable for sensitive applications.

There is another unfortunate consequence of this approach: ensuring determinism in the random value stream is possible, but with significant restrictions. Due to the aforementioned parallelization of the computation using the PRNG, the observed PRNG stream across the data set is a mangled mixture of (a potentially large number of) individual streams. This mangling has to be replicated in the sequential execution of the program to preserve determinism; and worse, all parallel backends have to observe the same such mapping. This can prove tricky when the parallel backends vary significantly in properties such as the available parallelism degree. A consequence of this behavior is also that in any execution of the program


SEED
master PRNG
0
1
2
3
4
5
...
p-1
0,0 1,0 2,0 3,0 4,0 5,0 ...,0 p-1,0
0,1 1,1 2,1 3,1 4,1 5,1 ...,1 p-1,1
0,2 1,2 2,2 3,2 4,2 5,2 ...,2 p-1,2
0,3 1,3 2,3 3,3 4,3 5,3 ...,3 p-1,3
0,... 1,... 2,... 3,... 4,... 5,... ... p-1,...
0,r-1 1,r-1 2,r-1 3,r-1 4,r-1 5,r-1 ...,r-1 p-1, r-1

(a) Stream-splitting approach to parallel pseudo-random number generation.


SEED
state forwarding
0 r 2r 3r 4r 5r ... (p-1)r
1 r+1 2r+1 3r+1 4r+1 5r+1 ... (p-1)r +1
2 r+2 2r+2 3r+2 4r+2 5r+2 ... (p-1)r +2
3 r+3 2r+3 3r+3 4r+3 5r+3 ... (p-1)r +3
... ...
r-1 2r-1 3r-1 4r-1 5r-1 6r-1 ... pr-1

(b) State-forwarding approach to parallel pseudo-random number generation.

Figure 13.2: Approaches for parallelizing a PRNG sequence.

for which deterministic random values are desired, the maximal number of threads has to be known *a priori*, before even executing a sequential back-end variant. If the degree of parallelism ever is increased, e.g. by moving to a larger processor, GPU, or cluster, the previous runs are invalidated with respect to the determinism criterion.

13.5.3 State forwarding

The approach taken in this work is *state forwarding*. We attempt to side-step the issues of both the global synchronization as well as the stream splitting approaches. This is done by utilizing properties of the PRNG state spaces. A true sequential single-stream variant of the program is taken as the gold standard output, and the goal is to replicate the same output on any parallel backend, without the need of global synchronization or advance knowledge of parallelism degree. As in the stream splitting approach, data-parallel work items are deterministically mapped across available computational units (threads). This means that the number of tasks assigned to each thread is known ahead of time, and for simplicity without the loss of generalization we assume the work can be split evenly among threads.

Furthermore, we assume that the number of times a PRNG state is updated (i.e., the number of times a random value is generated) is known ahead of time for each work unit. Combining the knowledge of *work unit count* and *random calls per work unit*, we know exactly how many state-forwards each thread will generate in the respective data-parallel construct (i.e., skeleton invocation).

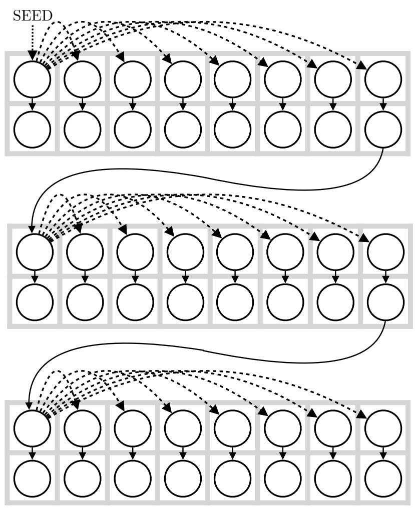
We can therefore, for each thread, *pre-forward* the state of the PRNG and store a copy of the forwarded state. These per-thread forwarded clones of the original PRNG can now act as the thread-private PRNG streams in the stream-splitting approach, with the additional property that when interleaved during the data-parallel execution, the aggregate observed stream now is equivalent to the sequential stream, which was the primary hurdle in the stream-splitting approach. Figure 13.2b illustrates the resulting pattern.

Still, the extra memory footprint of the thread-private PRNG states persists and will lead to additional overhead. The state-forwarding adds an additional computation step before the execution of the tasks, which can in the worst case be equally costly as the PRNG value extraction process itself (though it can also be parallelized). Properties of the PRNG state space have to be exploited to speed up the forwarding process and reduce the induced overhead.

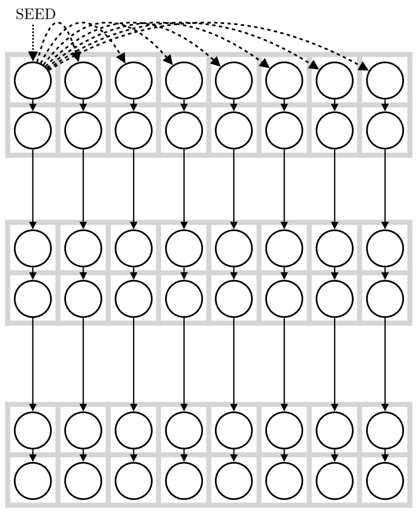
The leapfrog resp. sequence splitting method for state forwarding, introduced by Celmaster and Moriarty [19] for use with vector computers, considers a special case that allows to parallelize the forwarding phase of the PRNG. A linearly congruential PRNG with factor a is partitioned into p linearly congruential PRNGs each to be used r times, which are defined based on the same linear factor a, by $seed(i) = (a^r \cdot seed(i-1)) \mod m$ for $i = 1, ..., p$, $rand(i, 0) = seed(i)$ and $rand(i, j) = a \cdot rand(i, j-1) \mod m$. Hence, the p PRNGs equally partition the period of the seed PRNG in contiguous subsequences of length r. First, the a^{ir} for $i = 0, ..., p-1$ and the seed sequence can be calculated in parallel by a `Scan` in $O(\log p)$ steps, using the property $a^{2k} \mod m = ((a^k \mod m)^2) \mod m$. Then the $rand$ calls are independent for each i. (For reasonably low numbers of p such as for a current multicore CPU, sequential computation of the seeds should be faster; this is done in the current implementation.) The leapfrog / sequence splitting method scales well but is known to have problems for lcg with power-of-2 values for modulus and p. Skipping can also be applied for counter-based PRNGs [137] with output functions based on block ciphers for better statistics at a higher cost.

13.5.4 Optimizing long or iterated skeleton chains by pre-forwarding

While some applications may consist of a single parallelized step (such as a parallel for loop or skeleton call; we will use the latter here), others, in particular larger applications, will have multiple phases which are individually parallelized. A common example is iterative applications where each iteration in turn consists of one or more skeleton calls. To achieve good efficiency, we need to ask the question: *when* is the PRNG state split and



(a) No pre-forwarding between iterations.



(b) Pre-forwarding the PRNGs once before the iterative loop.

Figure 13.3: Container indexing and memory layout.

forwarded for the purposes of parallelization in a skeleton invocation scenario?

In a naive implementation of the state-splitting approach, the state splitting and forwarding step (see Figure 13.3a) is done right before each skeleton call. On the other hand, if we have a known number of skeleton calls (determinable by static analysis, lineage building [60], or program instrumentation), we only need to perform the splitting and forwarding of the PRNG states once per application. This is referred to as *pre-forwarding* and is illustrated in Figure 13.3b. In practice, restrictions such as data-dependent control flow (e.g., branches or iteration bounds) may limit the degree to which pre-forwarding can be applied, and application programmers may benefit from awareness of the cost-reduction opportunities from pre-forwarding already during program design.

13.5.5 API extension design

We have implemented the state-forwarding approach in the skeleton programming framework SkePU 3. SkePU did not previously have a random number generation component, and as shown in Section 13.4, previous manual implementations of PRNG-like functionality in SkePU applications have been ad-hoc and substantially different from each other. A baseline contribution of a framework-level PRNG library in SkePU is the programmability gains from reducing the effort of designing probabilistic applications

Listing 13.5: User function with calls to the SkePU random number generator.

```
float uf( skepu::Random<5> &prng, skepu::Region1D<float> region )
{
    float res = 0;
    for (int i = -2; i <= 2; ++i)  // 1D stencil with random weights
        res += region(i) * prng.getNormalized();
    return res;
}
```

on top of SkePU, as well as readability benefits from having a unified system for random number generation across all SkePU programs.

Random number extraction in user functions

As explained in Chapter 4, a SkePU skeleton is defined entirely by its type (e.g., `Map`), the signature of its instantiating user function, and state properties set on the resulting skeleton instance (such as `.setOverlap(...)` for `MapOverlap` instances). PRNG extraction is made available in all skeletons with a fully data-parallel mapping stage, which is the entire skeleton set except for `Reduce` and `Scan`.[4]

As such, the user function signature ("header") itself should encode the use of random number extraction. This is analogous to the preexisting option for mapping user functions to request the index of the currently processed element (see Listing 13.3). Therefore, we encode PRNG reliance in the same way. At the start of the parameter list (after the index parameter, if any), a parameter of type `skepu::Random<N>&` is added. `N` is a compile-time constant used in SkePU's *template metaprogramming*-based implementation to deduce the number of random values extracted by the user function in the *dynamic extent* of its evaluation. N is required to be known ahead-of-time for the state forwarding to work and determinism to be preserved.[5] A compilation option allows for run-time verification that the extraction count is obeyed.

Value extraction is carried out by a call to one out of two member functions of the `skepu::Random<N>` object. `random.get()` produces integers in $[0, \text{SKEPU\_RAND\_MAX})$ while `random.getNormalized()` returns real numbers in $[0, 1)$. Each call corresponds to one extraction and state update of the PRNG stream. A basic example of a user function with 5 random number extractions is shown in Listing 13.5.

[4]`Reduce` and `Scan` are parallelized through tree reductions reliant on the associativity property of their user functions.

[5]If determinism is not required by the application, `N` can be treated as an upper bound, which instead guarantees that no sub-sequences of random numbers are overlapping.

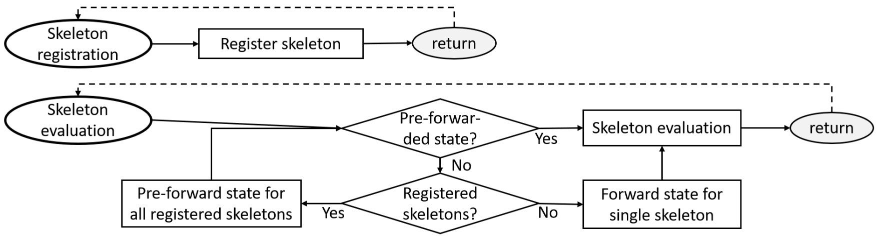


Figure 13.4: Flow-chart of the deterministic PRNG implementation. Here ellipses are events and boxes correspond to processes.

SkePU user functions are allowed to call other functions, subject to some but not all restrictions of skeleton-instantiating user functions. As the extraction count *N* is only required for instantiation, passing a PRNG stream object to indirect user functions is instead done with a `skepu::Random<>*` parameter with no positional requirement.

PRNG streams and skeleton invocations

Once a `skepu::Random<N>&`-enabled user function is present, a skeleton can be instantiated as usual. In addition to the skeleton instance, a PRNG stream object needs to be defined in the program: an object of type `skepu::PRNG`. Initialization of the PRNG stream takes an optional *seed* integer argument. The seed changes the deterministic sequence generated in the stream and can be assigned from an external entropy source (e.g., a timestamp) if non-determinism across program runs is preferred.

The stream object is a state machine which registers skeletons ahead of invocation time. Also in this way PRNG streams work like SkePU's index parameters: the stream is not part of a skeleton call's argument list. Instead they are registered as `skeleton.setPRNG(prng)`, if the programmer wants explicit control over stream objects and the way they map to skeleton instances; if no stream is registered, the skeleton picks a global PRNG stream by default at invocation time. The full flow chart of the registration and evaluation process is shown in Figure 13.4. In short, several skeleton instances may be *registered* before reaching an *evaluation* event. Only at this point is the PRNG sequence split across computational units and forwarded to the appropriate state. The input size (i.e., the maximum degree of parallelism) has direct impact on the forwarding leaps and is only known at the evaluation point from the input arguments to the skeleton call.[6] In subsequent skeleton invocations, the PRNG object checks for existing forwarded state and skips directly to evaluation (refer to Figure 13.3b).

[6]The input size is assumed to be uniform over a sequence of skeleton calls.

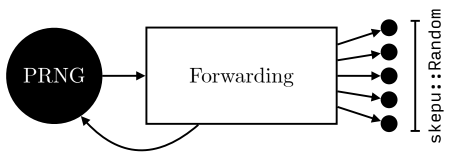
Listing 13.6: Pi approximation using the new SkePU PRNG API.

```
#include <iostream>
#include <skepu>

int monte_carlo_sample(skepu::Random<2> &random)
{
  float x = random.getNormalized();
  float y = random.getNormalized();
  // check if (x,y) is inside region:
  return ((x*x + y*y) < 1) ? 1 : 0;
}

int add(int lhs, int rhs) { return lhs + rhs; }

int main(int argc, char *argv[])
{
  auto montecarlo = skepu::MapReduce<0>(monte_carlo_sample, add);

  skepu::PRNG prng;          // optional
  montecarlo.setPRNG(prng); // optional

  const size_t samples = atoi(argv[1]);
  montecarlo.setDefaultSize(samples);

  double pi = (double)montecarlo() / samples * 4;
  std::cout << pi << "\n";
}
```

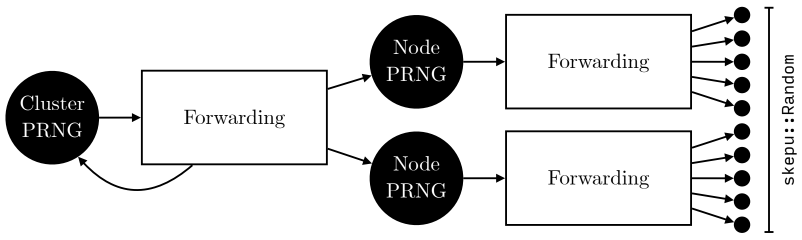
Listing 13.6 shows a variant of the Monte Carlo Pi calculation algorithm using the new SkePU API. Implementation with a `MapReduce<0>` skeleton enables a data-parallel computation without explicit container allocation, as the algorithm needs no element-wise input data to the user function; all input is derived from the PRNG stream. Internally, SkePU will use two containers: one input data set for the split PRNG sub-sequence states, and one output data set for the results of the user function invocations. Note, however, that SkePU will optimize the size of these intermediate data sets; they grow by $O(p)$, the number of computational units, and not $O(n)$, problem size (here the sample count).

Our prototype implementation handles multiple PRNG stream objects across different skeleton calls, but a single skeleton call (and thus its user function) can only receive values from one PRNG stream per invocation. `skepu::Random` usage can be combined with most other SkePU features, with a notable exception being dynamic scheduling for multi-core execution introduced [59] in SkePU 3.

The SkePU implementation of deterministic PRNG streams cover a wide set of backend targets. For *OpenMP*, *OpenCL*, and *CUDA*, the forwarding is straightforward as the worker threads are homogeneous, running on equal hardware resources. In SkePU's *hybrid backend* which simultaneously targets multi-core CPU and GPU resources, the normal forwarding process is



(a) Forwarding on node-level backends.



(b) Forwarding on cluster-level backends.

Figure 13.5: Differences in state-forwarding approach in cluster backend.

done twice in sequence: first once for the GPU, then the same stream is forwarded again for the CPU threads. The end result is a single set of forwarded thread-specific state objects (Figure 13.5a), but with non-equal step count. Finally, we have an early prototype implementation targeting the *cluster* backend. In this case, parallelization is done in two steps: once among nodes through StarPU and once among CPU cores by means of OpenMP. The forwarding process for PRNG streams is therefore hierarchical, as illustrated in Figure 13.5b.

13.6 Related work

Kneusel [86] has a chapter on parallel PRNGs, but with respect to deterministic execution only reports a manual construction of duplicating the state variable for each thread, plus skipping a number of states in order to achieve the same state as in a sequential execution. He also explains counter-based PRNGs and their suitability for parallelization because they allow skipping any given number of states with constant effort. Fog [68] discusses requirements on PRNGs in parallel computations, but focuses on avoiding overlapping sequences in different threads by combining generators, while L'Ecuyer et al. [90] focus on providing independent streams and substreams. Salmon et al. [137] focus on output functions for counter-based PRNGs to provide fast skipping of states but still provide good statistical quality. All do not focus on deterministic execution independent of parallelization, and have static mapping of tasks to threads in mind.

Leiserson et al. [96] argue that SPRNG [109], which provides a deterministic parallel PRNG, shows poor performance on Cilk programs and thus is not suitable for massive parallelism. They propose pedigrees, a mechanism to achieve a kind of linearization (i.e. a kind of equivalence to a sequential execution) in a Cilk program independent of the Cilk scheduler. However, they do not address pattern-based parallelization.

Parallel PRNG specifically for GPU include the cuRAND library for CUDA, SYCL-PRNG for SYCL, and work by Ciglarič et al. [23] for OpenCL. The Thrust skeleton library for CUDA also includes a PRNG library. Passerat et al. [123] discuss general aspects of PRNG on GPGPUs. GASPRNG [70] is an early attempt at realizing the full SPRNG generator set on CUDA GPUs, including clusters of CUDA GPU nodes.

14 Towards a modernized auto-tuner

Parts of this chapter are based on the following bachelor's thesis project, supervised and developed jointly by the author:

Basel Nsralla. "Modernizing and Evaluating the Auto-Tuning Framework of SkePU 3." Bachelor's Thesis. Department of Computer and Information Science, 2022

14.1 Background

A high-level parallel programming framework with multiple backend targets presents a problem for the user: *how to choose the appropriate backend for a particular program run*? Predicting which backend is the best one depends on several factors. Firstly, different backends are *better suited* for different applications. A multicore CPU processor is designed to handle complex control flow logic (MIMD), while a GPU accelerator excels at managing large numbers of threads following the same instruction sequence (SIMD), and so on. Secondly, applications may not even agree on the *metric* which defines the "best" backend. For some applications execution time is important, while others are more concerned about power consumption [100]. Finally, the *problem size* is a crucial property when tuning against different types of parallel backends, as the overhead of scheduling a task on a back-

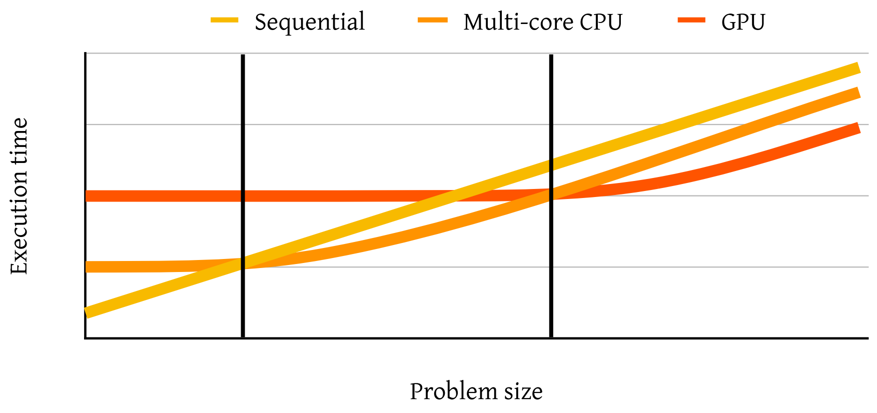


Figure 14.1: Idealized performance behavior of a multi-backend application, with crossover points.

end is typically independent of problem size and therefore amortized over tasks with large problem sizes.

Figure 14.1 illustrates an hypothetical idealized run of some data-parallel application. With large problem sizes, backends with greater parallelism complete the task the fastest, but these backends are also expected to have significant time penalty at startup. For a GPU, this time includes driver overhead, data movement, and possibly even kernel compilations. Sequential computation has very little startup costs and the execution time curve is close to linear. This model is a good rough approximation of the performance behavior of data-parallel applications such as those in SkePU and is the basis of its auto-tuner implementation[1]. If we assume that the performance curves as seen in Figure 14.1 are reasonably convex, the answer to the question stated at the top of the chapter is then transformed into the task of finding the *crossover points* where the fastest backend changes. These points are marked in the figure with vertical lines. A related approach is to generate an *execution plan*, which separates the problem size parameter space into intervals and records the predicted best backend for each interval. In either case, the tuner estimates the performance at various problem sizes, either by offline estimations or by *sampling* the problem itself with generated input data.

14.2 SkePU variadic tuner design

Work on the original auto-tuner in SkePU predates this thesis. Dastgeer et al. [46, 48] implemented the auto-tuning strategies of SkePU 1, and while

[1]In practice many other factors have to be accounted for, including data locality effects from cache hierarchies and smart data-container coherency states.

the underlying tuner is still present, the interface is not a perfect fit with the variadic template metaprogramming-enabled skeletons of SkePU 2 and later. Element-wise operands of the same skeleton instance share the same size, but other operands such as random-access data-containers are decoupled from the element-wise size and adds another dimension to the problem size parameter space. When the parameter space grows, tuning strategies need to be re-evaluated. At the same time, the auto-tuning field is evolving, with new optimizations being introduced [99]. SkePU needs a modernized tuner implementation, and the rest of this chapter presents an early prototype of a variadic auto-tuner interface for it.

Work on the modernized auto-tuner thus far does not include revised prediction models, search pruning, or other similar tuning strategies, but is rather aimed at generating variadic execution plans with existing tuning infrastructure.

14.3 Implementation

This section elaborates on three key parts of the variadic auto-tuner implementation: generation of multi-dimensional problem size sequences from argument sequences in SkePU user functions, sampling such sequences into concrete input data sets, and from the resulting performance samples generate an execution plan with persistent serialization to file storage.

14.3.1 Multi-dimensional argument sequences

Each user function has a set of parameters, classified into element-wise, random-access, and uniform types. Each of those parameters are based on data types with a fixed dimension, e.g., `Matrix` is always 2D and `Vector` is always 1D. Some exceptions are found in skeletons where it is possible to call the skeleton with either 1D or 2D data types. This makes the dimension of the parameters incompatible between the auto-tuner and the expected input argument. To solve that problem, dimension collapsing is applied on the specific parameter. Collapsing means that, after determining the dimensions for an ambiguous skeleton (for instance 1D `MapOverlap`), the auto-tuner can change the dimensions of the input to match the one the auto-tuner deduced.

Collapsing the argument dimensions will not affect the search. On the other hand, if the dimension expectation at compile time is not the largest possible, a dimension extension will be required. This extension would affect the search in the execution plan as the search space will then not contain the extended dimensions and a new auto-tuning process would be needed. This problem has not been observed in practice. Dimension collapsing has been a very rare occasion and will only happen if the skeleton

Listing 14.1: Declarative sequence generation.

```
Permutation<
  Group<Standard<Dimensions<ret...>>, Standard<Dimensions<elwise...>>>,
  Single<Ultra<Dimensions<cont...>>>,
  Single<Ultra<Dimensions<uni...>>>
>()
```

does not specify the data container type in the declaration, but rather specifies pointers or index types as element-wise arguments. Future development of the tuning infrastructure can take these edge cases into account.

To perform the auto-tuning, multiple variations of the input need to be generated; each variation forms a sample sequence. The sequence generator takes the dimensions of the parameter types and the sampling limit as input, and generates a sample of all possible sizes for each parameter. Each sequence will be used by the auto-tuner and run on each backend to identify the apparent ”best” backend for it. The sizes are incremented by a power of two; the choice of exponential size increment was done to form an estimation for the dynamic range of sizes and is subject to change in the future.

The argument sequences are generated systematically, as many skeletons have a defined relationship between their arguments. To solve this problem, a declarative interface was implemented to manage the relationships between the arguments. An example of how the interface can be used can be seen in Listing 14.1. The `Group` type groups two or more parameter types to always have the same size, `Single` makes the parameter independent, `Standard` and `Ultra` determines if the permutations of the arguments should be applied on an inner level, `Ultra` would create combinations considering the inner dimensions of the parameter, `Standard` ignores the inner level combinations, which means that `Standard` and `Ultra` would be equivalent on a one-dimensional parameter. This interface is used only by SkePU developers; it is not part of the user-facing SkePU API.

14.3.2 Sampler

The `Sampler` class template takes a skeleton class as its template argument. `Sampler` represents an abstract base class that constructs the types of the different argument categories as if they were user types. This is needed because the sampler is invoked in the same way a SkePU program is. The argument categories are using declarations that exist in every skeleton.

The types constructed are often containerized versions of the same type, with some exceptions. Those exceptions occur in `Mat<T>`, `MatRow<T>`, `MatCol<T>`; all specialized to yield `Matrix<T>`. Similarly, `Vec<T>` yields `Vector<T>` instead. Containerizing is important as the internal parts of

SkePU will call the user function based on the skeleton type, and the parameters will be extracted from the provided containers, the same way a normal SkePU program would invoke the skeleton instance.

Sampler objects are specialized for skeletons which expose operand interfaces not compatible with the basic `Map` assumptions. While `Map` can be applied to smart data-containers of any dimensionality (1D–4D), the skeleton will interpret the problem space as a linearization of the input, and tuning is thus always one-dimensional in the element-wise space.

While `MapOverlap` can also be applied for any dimensionality, here the size in each dimension is directly impacting the pattern evaluation, and thus also the performance characteristics. Configured *overlap radius* presents yet more parameters for the input space, one for each overlap dimension. If no overlap size has been configured, the sampler class assumes an overlap radius heuristically.

14.3.3 Execution plan and persistence

The expected output of any auto-tuning implementation is an execution plan (encoding a model), which represents the result of the auto-tuning in a form of mapping from an input size to the expected optimal backend for that input. The execution plan uses linear search to find the target input size, and returns the backend as a `BackendSpec` which encapsulates information about the backend in more detail. `BackendSpec` is SkePU's preexisting representation of each backend configuration.

As execution plans are expensive to generate, serialization to file-system storage is utilized to persist the execution plan for each compilation. The `ExecutionPlan` class was made serializable to JSON format using the *nlohmann* JSON C++ library[2]. To distinguish between serialized execution plans throughout changes to a program, a version number constant is maintained. In the tuner prototype, this number is generated by a compiler macro; in the future, the SkePU precompiler can be extended to provide a unique ID for each code generation. A comparison at runtime between the version number of the program and the saved version in the serialized execution plan is then enough to determine whether the file storage is invalidated.

The auto-tuner starts by looking, for each skeleton instance, for a previously generated execution plan, and returns it if found. Otherwise, it proceeds by decomposing the skeleton call into its parameter categories passed to the dimension builder. The dimensions are passed to the argument sequence generation which returns the sequences consisting of sizes to be applied. The sequences are then passed to the `Sampler`, which builds the sample data. The sample data, namely the constructed arguments, are then

[2]`https://github.com/nlohmann/json`

Listing 14.2: Matrix-vector multiplication in SkePU using `Mat`.

```cpp
#include <skepu>

template<typename T>
T mvmult_f(skepu::Index1D row, const skepu::Mat<T> m, const skepu::Vec<T> v)
{
  T res = 0;
  for (size_t i = 0; i < v.size; ++i)
    res += m(row.i, i) * v(i);
  return res;
}

int main(int argc, char *argv[])
{
  size_t rows = atoi(argv[1]);
  size_t cols = atoi(argv[2]);
  auto spec = skepu::BackendSpec{argv[3]};

  skepu::Vector<float> v(cols), r(rows);
  skepu::Matrix<float> m(rows, cols);

  auto mvprod = skepu::Map(mvmult_f<float>);
  mvprod(r, m, v);
}
```

passed to the benchmarking system which executes the skeleton using all the different backends and finds the fastest one. The fastest backend is then added to the execution plan as the best backend for the particular sample size. The execution plan is fully constructed when all the samples are tested and gets flushed to the disk. The plan is then used as a source for the backend choice, by providing the problem sizes at invocation time.

14.4 Future work

With the internal implementation of a variadic auto-tuner in place, immediate future work will be about exposing an interface for multi-parameter tuning to SkePU programmers. Practical SkePU applications expose more problem size parameters than is actually relevant for tuning. Consider, for example, a matrix-vector multiplication $y = Ax$. Shown in Listing 14.2, the vector y is produced element-wise from a `Map` skeleton instance, with the user function signature exposing one `Mat` and one `Vec` proxy-container parameter. The sampler system in the variadic tuner identifies four distinct tuning parameters: the output vector size, the input matrix height *and* width, and the input vector size.

As is also evident from the full program in Listing 14.2, the problem has actually only two independent size parameters. The output vector size is coupled to the height of the matrix, and the input vector size is coupled to the matrix width. Switching to the `MatRow` construct introduced in SkePU 3 gives the compiler and runtime more information about the access patterns, but even this can only eliminate one of the two superfluous dimensions (`MatRow` encodes the coupling between the output vector size and the matrix height), and the solution is not general. SkePU therefore needs a tuning interface wherein the user can declare such couplings explicitly, and furthermore express whether any scalar user function parameters are considered performance-impacting so the sampler can register tuning intervals for them.

15 Evaluation results

This chapter collects quantitative results from evaluations done on SkePU in general as well as the specific contributions presented throughout the thesis.

Being a high-level programming interface, we are interested in the usability of SkePU for programmers of different skill levels. Section 15.1 therefore presents a survey on the usability of SkePU code, including aspects of readability and programmer feedback through error messages.

However, as a parallel programming framework, the *speedup* relative to sequential processing is one of the most important measurable metrics for SkePU. Absolute speedup numbers require optimized sequential implementations of application in addition to ”SkePU-ized” code performing computations generating exactly the same result. Such performance evaluation can be found in Section 15.6.

Evaluation results from the mini-apps in the EXA2PRO project is summarized in Sections 15.6.4, 15.6.5, 15.6.6, and 15.7.2.

Because of the effort of constructing such benchmarks, performance evaluation of specific innovations or additions to SkePU is often done by comparing relative to SkePU itself. Section 15.2 evaluates SkePU backends against each other and in addition compares skeleton structures introduced in SkePU 2 to the closest corresponding construction in SkePU 1.2.

Section 15.3 evaluates the performance of the lineages optimization (Chapter 10), Section 15.4 presents performance results of the hybrid backend implementation from Chapter 8, and Section 15.5 evaluates the multi-variant user functions of Chapter 12.

Section 15.7 presents results related to the deterministic parallel random number generator as implemented in SkePU.

Section 15.8 and Section 15.10 include selected results from two master's thesis projects, covering the SkePU-GPI cluster backend and the modernized variadic auto-tuner, respectively.

Section 15.9 contains micro-benchmarks (synthetic code or small-scale applications) on features introduced in SkePU 3, and Section 15.11 finishes with an evaluation of high-level skeleton fusion.

15.1 SkePU usability evaluation

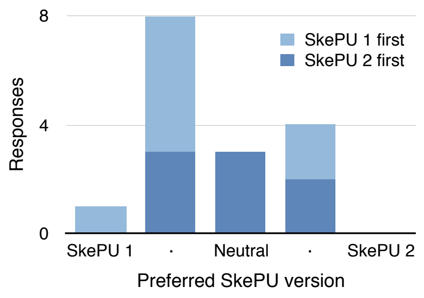
The results in this section were first published in the paper *SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems* [62]. The survey was based on the work-in-progress programming interface for SkePU 2.

15.1.1 SkePU 2 prototype survey

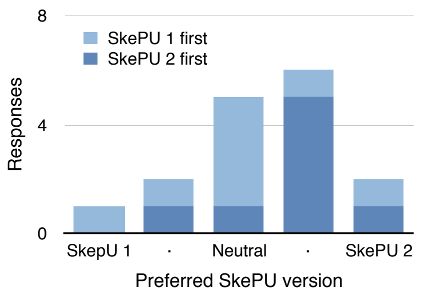
The interface introduced in SkePU 2 (and still being the basis of SkePU 3) aims to improve on that of SkePU 1 with increased clarity and a syntax that looks and feels more native to C++. To evaluate this, a survey was issued to 16 participating respondents, all master-level students in computer science. The participants were presented with two short example programs: one very simple and one somewhat complex, each both in SkePU 1 and SkePU 2 syntax. To avoid biasing either of the SkePU versions, the order of introductions was reversed in half of the questionnaires. See the thesis [58] for more discussion on the survey, including the code examples presented to the respondents.

Note, however, that the survey was issued when the syntax of SkePU 2 was not yet finalized. At the time C++11 attributes were required to guide the precompiler: `skepu::userfunction` on user functions, `skepu::instance` on skeleton instances, `skepu::usertype` on user-defined `struct` types appearing in user functions, and `skepu::userconstant` for global constants on `constexpr` global variables. The attributes allowed for a straightforward implementation of the precompiler, and the reasoning was that clearer expression of intent from the programmer could improve any error messages emitted.

Figure 15.1 presents the responses comparing the two SkePU versions in terms of code clarity (to the question *How would you rate the clarity of this code in relation to the previous example?*). The usability evaluation indicates that the SkePU 1 interface is sometimes preferred to the SkePU 2 variant, at



(a) Simple example



(b) Complex example

Figure 15.1: Comparison of code clarity, SkePU 1 vs. SkePU 2.

least when the user is not used to C++11 attributes as indicated by the free-form comments in the survey. We realized that the decision to use attributes as a fundamental part of the syntax needed to be revisited.

In the more complex example, respondents generally considered the SkePU 2 variant to be clearer. We believe that the reason for this is the fact that it has fewer user functions and skeleton instances than the SkePU 1 version (thanks to the increased flexibility offered in SkePU 2). The user functions are also fairly complex, so the macros in SkePU 1 may be more difficult to understand.

The results can still be considered valid for SkePU 3, since the interface of the specific skeletons in the survey has not changed much except for the attributes. However, an updated and expanded usability survey of state-of-the-art SkePU interface would be of general interest.

15.1.2 SkePU 3 survey

A similar survey was conducted by *Erik Tedhamre* as part of his in-progress master's thesis project. We present an early summary of the results here. Unlike the SkePU 2 prototype survey, which was comparing generations of SkePU syntax, for this one the goal was to compare SkePU to common parallel programming interfaces taught to university students: OpenMP and CUDA. Respondents were picked at random among university students in computer science-related programs.

Implementations of three applications[1] were selected for comparison purposes. The implementations in OpenMP and CUDA were taken from the benchmark repositories, while the SkePU variants were written by Erik Tedhamre under supervision from August Ernstsson. Each respondent was randomly assigned one of the three applications.

[1]Breadth-first search (`bfs`) from Rodinia [20], covariance (`covariance`) from PolyBench [160], and finite-difference time-domain method for 2D data (`fdtd-2d`) from PolyBench.

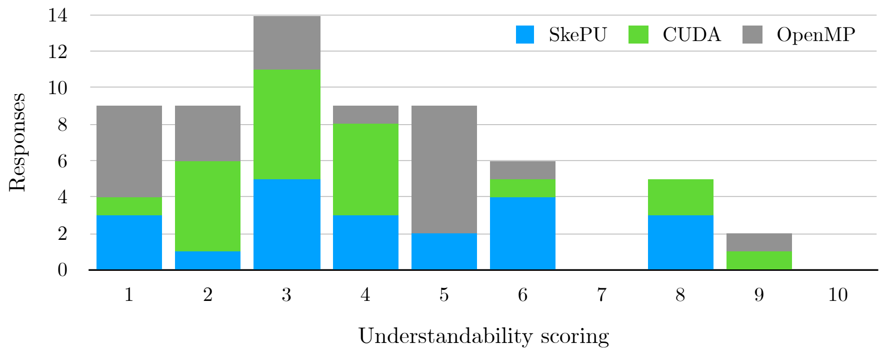


Figure 15.2: Understandability scoring frequency results from the SkePU 3 survey. The figure shows the aggregate of all three applications for each programming model.

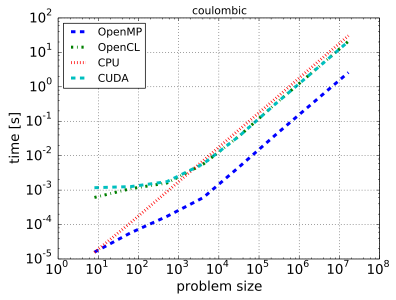
A summary of the scoring for all three programming models is shown in Figure 15.2.

SkePU achieves the best average score. The difference is, however, quite small and the spread is significant. The survey results are planned to be discussed in detail in an upcoming master's thesis report by Erik Tedhamre, including correlations between framework ranking and a control question designed to evaluate the programming proficiency of the respondents. Preliminary results indicate that stronger programmers tend to rank SkePU understandability higher, relative to CUDA and OpenMP.

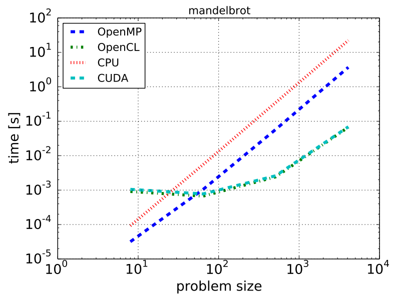
15.2 Initial SkePU 2 performance evaluation

The results in this section were first published in the paper *SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems* [62]. Experiments were carried out on an early version of SkePU 2 and on hardware available at the time.

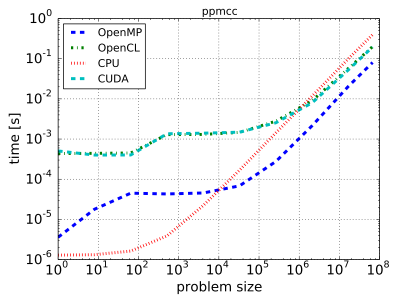
The system used for testing consists of two eight-core Intel Xeon E5-2660 ”Sandy Bridge” processors at 2.2 GHz with 64 GB DDR3 1600 MHz memory, and a Nvidia Tesla k20x GPU. The test programs were compiled with GCC g++ 4.9.2 or, when CUDA was used, Nvidia CUDA compiler 7.5 using said g++ as host compiler. Separate tests were conducted on consumer-grade development systems, showing similar results after accounting for the performance gap. The framework has also been tested on multi-GPU systems using CUDA and OpenCL, and a Xeon Phi accelerator using the Intel's OpenCL interface, the latter shown in Figure 15.4. All tests include data movement to and from accelerators, where applicable.



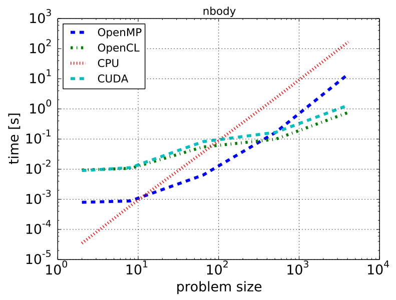
(a) Coulombic potential



(b) Mandelbrot fractal



(c) Pearson product-movement correlation coefficient



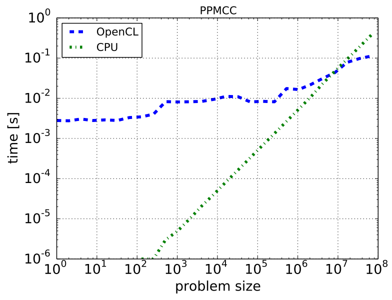
(d) N-body simulation

Figure 15.3: Test program evaluation results. Log-log scale.

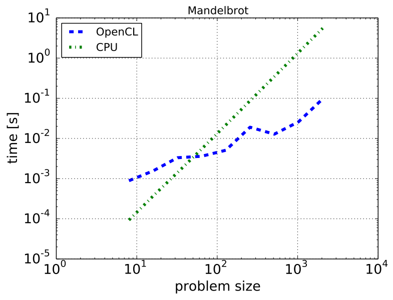
Results are shown in Figure 15.3. The following test programs were evaluated:

- **Pearson product-movement correlation coefficient**
  A sequence of three independent skeletons: one Reduce, one unary MapReduce and one binary MapReduce. The user functions are all trivial, containing a single floating point operation. The problem size is the vector length.

- **Mandelbrot fractal**
  A Map skeleton with a non-trivial user function. There is no need for copy-up of data to a GPU device in this example, but the fractal image is copied down from device afterwards. In fact, there are no non-uniform inputs to the user function, as the index into the output container is all that is needed to calculate the return value. The problem size is one side of the output image.

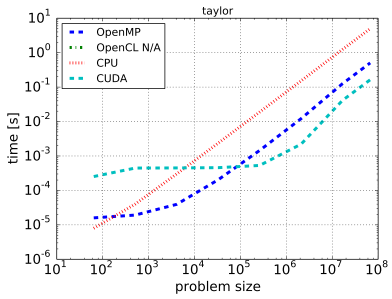


(a) Pearson product-movement correlation coefficient

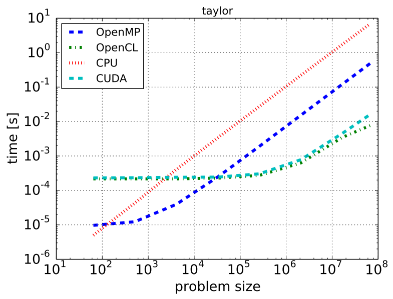


(b) Mandelbrot fractal

Figure 15.4: Evaluation results on Xeon Phi using OpenCL.



(a) SkePU 1.2



(b) SkePU 2

Figure 15.5: Comparison of Taylor series approximation

- **Coulombic potential**
  This program calculates electrical potential in a grid from a set of charged particles, as an iterative computation invoking one Map skeleton per iteration. The user function takes one argument, a random-access vector containing the particles. It also receives a unique two-dimensional index from the runtime, from which it calculates the coordinates of its assigned point in the grid.

- **N-body simulation**
  Performs an N-body simulation on randomized input data. The program is similar to Coulombic potential, both in its iterative nature and the types of skeletons used.

The preview release of SkePU 2 had not been optimized for performance. Even so, it has already shown to match or surpass the performance of SkePU 1.2 in some tests. However, the results vary with the programs tested and seem particularly dependent on the choice of compiler. A mature optimizing C++11 compiler is required for SkePU 2 to be competitive performance-wise.

In cases where the increased flexibility of SkePU 2 and later allows a program to be implemented more efficiently—for example by reducing the amount of auxiliary data or number of skeleton invocations—SkePU 2 may outperform SkePU 1 significantly. Figure 15.5 shows such a case: approximation of the natural logarithm using Taylor series. For SkePU 1, this is implemented by a call to Generate followed by a call to MapReduce; in SkePU 2 and later a single MapReduce is enough, reducing the number of GPU kernel launches and eliminating the need for $\mathcal{O}(n)$ auxiliary memory.

15.3 Performance evaluation of lineages

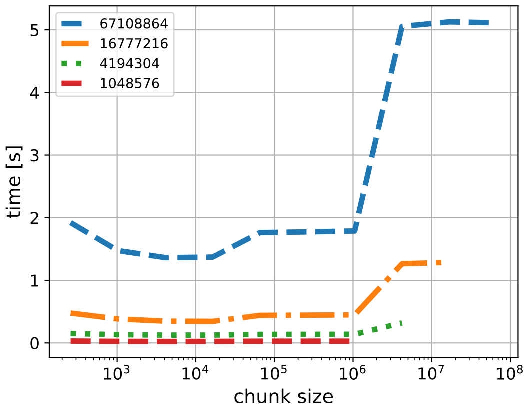
The results in this section were first published in the paper *Extending smart containers for data locality-aware skeleton programming* [60]. Experiments were carried out on an experimental branch of SkePU 2 and on hardware available at the time.

We have conducted performance evaluation of the lazy tiling approach by benchmarking the applications presented earlier: *Horner's method* in Section 10.4.1, *exponentiation by repeated squaring* in Section 10.4.2, and heat propagation from Section 10.4.3. The first two are sequences of `Maps` while the latter uses `MapOverlap`.

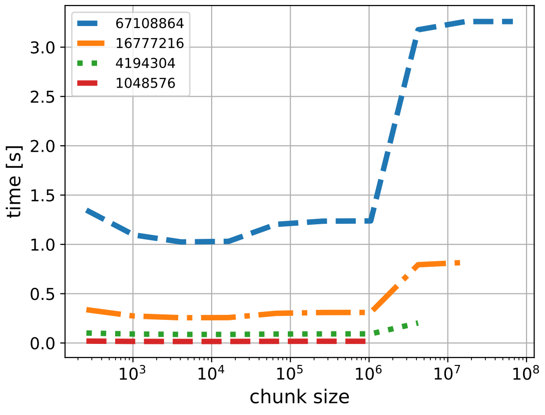
All performance evaluation was performed on Intel Xeon CPU model E5-2630L at 2.40 GHz and hyperthreading enabled. The cache memory hierarchy was as follows: 32 kB L1 data cache, 256 kB L2 cache, and 15 MB L3 cache. All programs were precompiled with the SkePU source-to-source compiler into C++ source files, which were then processed by the GCC C++ compiler version 5.4.1 in C++11 mode. The optimization level was set at -O3 with no other flags explicitly set. SkePU's built-in benchmarking API uses standard C++11 functions (`std::chrono` library) for wall-clock-based time measurements.

15.3.1 Sequences of Maps

Both Horner's method and exponentiation by repeated squaring are iterative sequences of `Map` skeletons with small user functions and virtually ideal situations for lineage building. With a trivial user function, the benchmark is memory bound and tiling should give a significant performance improvement. The source code is very similar to Listings 10.6 and 10.7, but instrumented with measurement directives and additional constructs for explicitly requesting lineage evaluation.



(a) Horner's method



(b) Exponentiation by repeated squaring

Figure 15.6: Benchmark results for Map sequences.

The lineage length is data dependent for both examples. We used a coefficient vector of size 8 for Horner's method and the exponent 87 for repeated squaring. Note that the exponent not only defines the length of the lineage but also the shape, i.e., how many calls to `mult` occur in between iterations.

Results are visible in Figure 15.6, where each line represents one of four problem sizes (`size_r` in the source code). Different chunk sizes (`chunksize`) are tested for each problem size. We can see that a significant speedup can be achieved from loop tiling, as long as the problem size is large enough. The optimal chunk size seems relatively consistent across

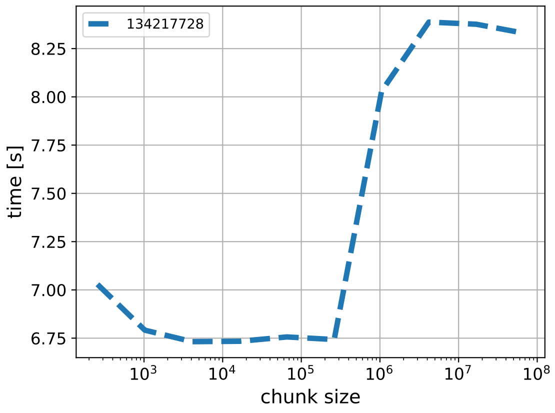


Figure 15.7: Heat propagation with tiled MapOverlap.

the tests, but this will be dependent on the memory hierarchy of the target system. There is still some overhead with the lambda expression-based implementation, but the advantages of tiling are still apparent. A chunk size equal to the problem size will result in the same behavior as if the lazy tiling is not enabled, and is included as the last data point for each series as a baseline. Comparing that with the optimal chunk size shows that the tiling implementation can provide over 3x speed-up. Considering the attributes of the benchmarks as described above, this is likely to be a best-case scenario.

15.3.2 Heat propagation

We have also instrumented the code from the heat propagation example in Listing 10.8 for performance evaluation. Using an unrolling factor of 4 gives the results in Figure 15.7. Compared to the Map benchmarks, this application is less well-suited for the lineage-based tiling, as there is more work per element and the algorithm is relatively more computation bound. A single container is also not used more than twice in the scope where the lineage is built up. Even so, Figure 15.7 shows a similar behavior to the Map benchmarks with a big performance gain once the chunk size fits in the caches. Both the maximum performance gain and the relative overhead of lazy evaluation is less significant here, but the optimal chunk size still has 23 % speed-up when compared with no tiling.

15.4 Hybrid backend

The results in this section were first published in the paper *Hybrid CPU–GPU execution support in the skeleton programming framework SkePU* [119].

The implementation was evaluated on a system consisting of two octa-core Intel Xeon E5-2660 (16 cores in total) clocked at 2.2 GHz with 64 GB of memory and a Nvidia Tesla K20x GPU with 2688 processor cores and 6 GB of device memory. The programs were compiled with nvcc (v7.5.17) using g++ (v4.9.2) as host compiler.

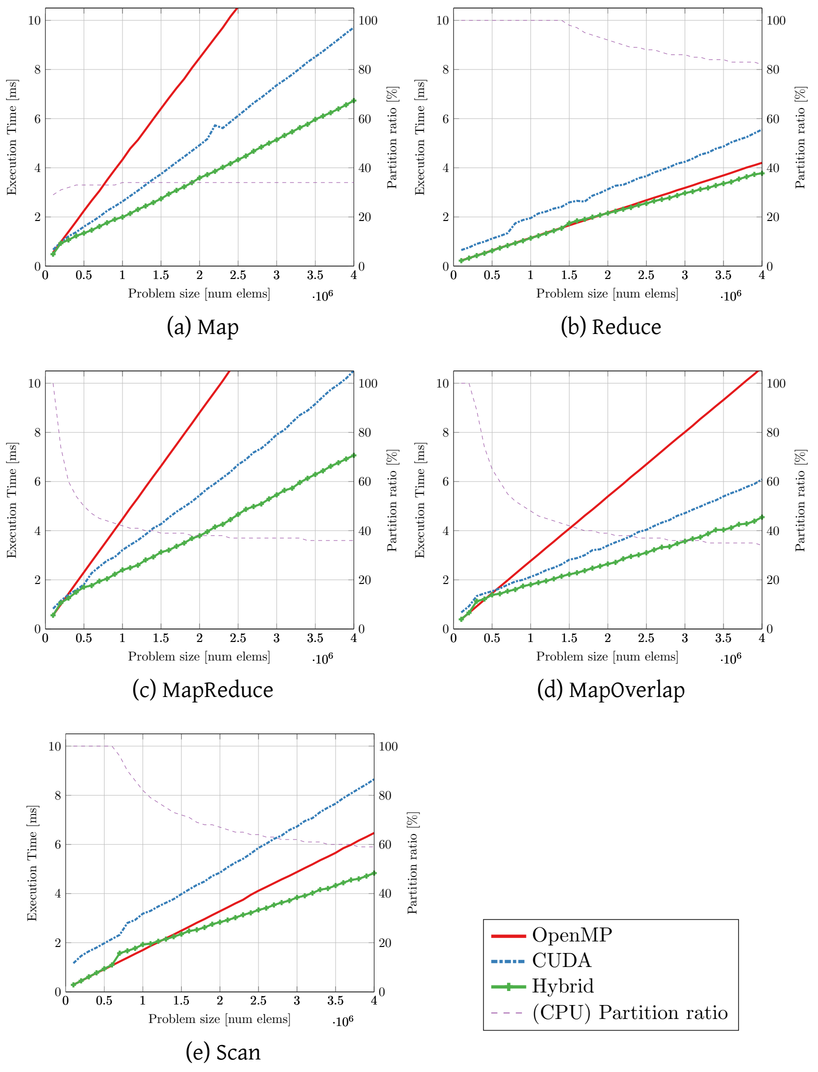
15.4.1 Single skeleton evaluation

First each skeleton type was evaluated with typical user functions. Input sizes ranging from $100,000$ to $4,000,000$ in increments of $100,000$ were used. Each input size and backend combination was executed seven times and the median execution time was noted to eliminate outliers caused by other operating system processes occasionally running on the CPU. The predicted partition ratio used in the hybrid backend was also noted for each input size. The hybrid backend was tuned with the auto-tuner a single time and the same execution time model was then used for all input sizes. The results are shown in Figure 15.8. As can be seen in the graphs, the hybrid backend improves upon the performance of the OpenMP and CUDA backends for all skeletons, at least as the input size grows. For most skeletons the hybrid backend even manages to match the performance of the OpenMP and CUDA backends for small input sizes, by switching to CPU-only or GPU-only execution. For the Scan skeleton however, a leap in the hybrid backend curve can be seen, where the partition ratio prediction switches from CPU-only to hybrid too early, as the predictor overestimates the performance of hybrid execution. This is likely due to the extra complexity of the hybrid execution implementation of the Scan skeleton, where the performance of the CPU and the accelerator partitions do not completely match the performance of the OpenMP and CUDA/OpenCL backends used in the auto-tuning.

15.4.2 Generic application evaluation

To evaluate the implementation in a more realistic context, we also compared the performance of the new hybrid scheme on some of the example applications provided with the SkePU source code. A presentation of the applications and which skeletons they use is shown in Table 15.1. In the Skeletons column, the number within <> tells the arity of the skeleton instance, i.e. how many element-wise accessed input containers it uses.

The applications were executed with five different configurations. First with the sequential CPU backend as a baseline. Then the OpenMP, CUDA and



(a) Map

(b) Reduce

(c) MapReduce

(d) MapOverlap

(e) Scan

Figure 15.8: Execution time of skeletons

hybrid backends. For the hybrid backend, all skeletons were tuned with the auto-tuner. Tuning was done with 10 steps and the tuning time was not included in the measured execution time. Finally, we used an *oracle* to find the upper limit to the speedup possible to achieve with the hybrid backend implementation, given an optimal partition ratio choice. Oracles has been used in earlier research to show the upper bound of hybrid execution implementations [75, 103, 141]. We let the oracle execute the application using the hybrid backend with a manually set partition ratio for each skeleton in-

Application	Algorithm	Skeletons
CMA	Cumulative moving average	Map<1>, Scan
Coulombic	Coulombic potential	Map<1>
Dotproduct	Dot product	MapReduce<2>
Mandelbrot	Mandelbrot fractal	Map<0>
PPMCC	Pearson product-moment correlation coeff.	Reduce, MapReduce<1>, MapReduce<2>
PSNR	Peak signal to noise ratio	Map<2>, MapReduce<2>
Taylor	Taylor series expansion of $\log(1+x)$	MapReduce<0>

Table 15.1: List of applications used in the evaluation.

stance, ranging from 0% to 100% in increments of five percentage points. For multi-skeleton applications all combinations of ratios were tested. The fastest of these execution times was then saved as the oracle's time. All backends, including all partition ratio combinations tested by the oracle, were executed seven times, and the median execution time was used. The results are shown in Figure 15.9. The figure shows that the hybrid backend improves upon the OpenMP and CUDA backends in most applications. By comparing the hybrid bar to the oracle bar we can see that the auto-tuning finds good partition ratios, but there is some room for improvement. According to the oracle two of the applications (PSNR and Taylor) do not gain from hybridization, at least not the tested problem sizes. This is also found by the auto-tuner in the Taylor case, as it falls back to CPU-only execution. PSNR is the only application where the hybrid backend fails to improve upon the performance of the OpenMP and CUDA backends. The reason for this is that the auto-tuning finds the optimal partition ratio to be 40% for the Map skeleton and CPU-only for the MapReduce skeleton. Although this is the optimal partition ratio for each individual skeleton instance, it is not the optimal choice when both skeletons are considered because of the need to move data between CPU and GPU memory. According to the oracle, offloading all data to the GPU gives the best execution time in this case.

15.4.3 Comparison to dynamic hybrid scheduling using StarPU

Finally, we show the improvement over the experimental hybrid execution implementation based on the StarPU runtime system that was implemented in SkePU 1. To make a fair comparison, parts of the old StarPU implementation was ported to SkePU 2. We also compared the execution time to the OpenMP and CUDA backends. As StarPU is supposed to get better over time by learning how to schedule the work, we tried executing the same skeleton multiple times. Each backend was executed 30 times in a row. New input containers were allocated each time to rule out the impact of data locality. The results are shown in Figure 15.10. In the graphs we can see the stability of the OpenMP, CUDA and hybrid backends. It is also apparent that the hybrid backend with the auto-tuning manages to find a good load balance and improves upon the execution time of the individual processing units. The

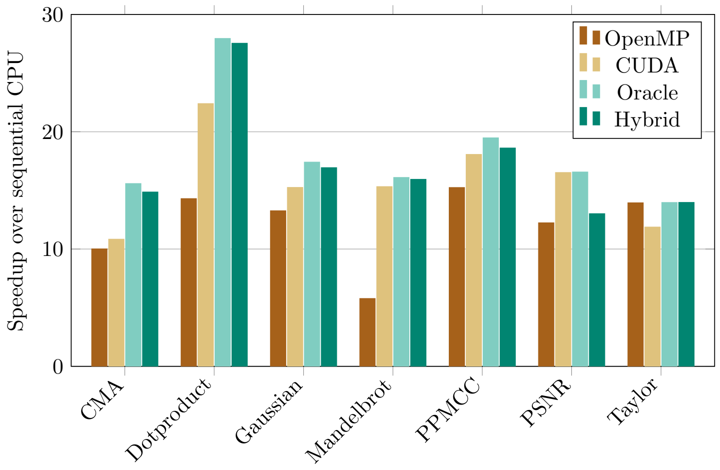


Figure 15.9: Speedups comparisons of example applications

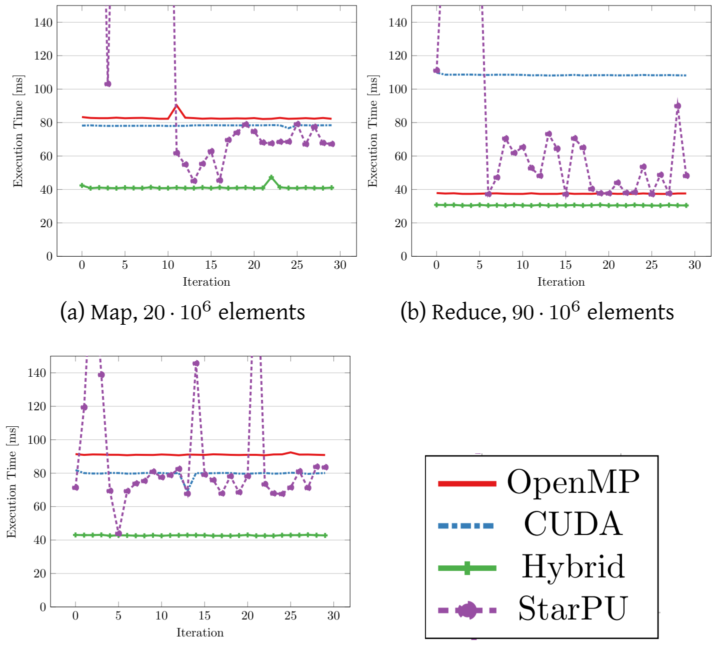
performance of the StarPU backend is unstable, even though it manages to match the performance of the hybrid backend in some iterations. For the Reduce skeleton both the hybrid and the StarPU backend have a hard time to improve the performance, as the skeleton works much better on the CPU compared to the GPU.

The execution time of the StarPU backend stabilizes somewhat with time, but it is still uneven after 20-30 repeated executions. This is likely due to the low number of tasks (manually found to be between 3 and 14) each skeleton instance had to be divided into for the best performance. This in turn is a result of the relatively small input size that was used in the evaluation. StarPU comes with a substantial overhead and might therefore be better suited for applications with even larger input sizes. The StarPU backend can also be of interest for special kinds of user functions with a very skewed workload, where adaptation is needed at runtime. But as these corner cases were not the target of the new auto-tuning implementation, no experiments were performed with such applications for this paper.

15.5 Evaluation of multi-variant user functions

The results in this section were first published in the paper *Multi-variant User Functions for Platform-aware Skeleton Programming* [61].

We present performance evaluations for two distinct use cases for multi-variant user functions: vectorization of `Map`-type skeleton applications on real and complex numbers, and specialization of the algorithms used in the user function of a stencil-type image filtering operation using `MapOverlap`.



(a) Map, $20 \cdot 10^6$ elements

(b) Reduce, $90 \cdot 10^6$ elements

(c) MapReduce, $20 \cdot 10^6$ elements

Figure 15.10: Execution time of repeated invocations of the same skeleton

15.5.1 Vectorization

To demonstrate the performance gained from vectorization of user functions in a scenario in which automatic compiler optimization might be prohibited, we test the example from Section 12.3.1 using the Intel C++ Compiler v.18.0.1. -O3 level optimization is enabled for all benchmarks, and the results are presented as the average of 100 runs. All computations are performed on single-precision floating point data. The target system uses Intel Xeon Gold 6130 processors. Two vectorization scenarios are evaluated:

Element-wise vector addition: Three variants are compared: no vectorization, and vectorization by a factor of four and eight, respectively.

Element-wise vector multiplication of complex numbers: Complex numbers stored in struct-of-arrays format, with four input data containers in total. Three versions are tested: no vectorization, factor eight direct vectorization, and a refactored vectorized version using fused multiply add (FMA) vector instructions.

For scalar element addition, the results show that there is always a benefit of vectorization if available. However, as seen in Figure 15.11 the overhead of loading and storing vector registers is significant when there is only one vector instruction to compute. The choice between four element vec-

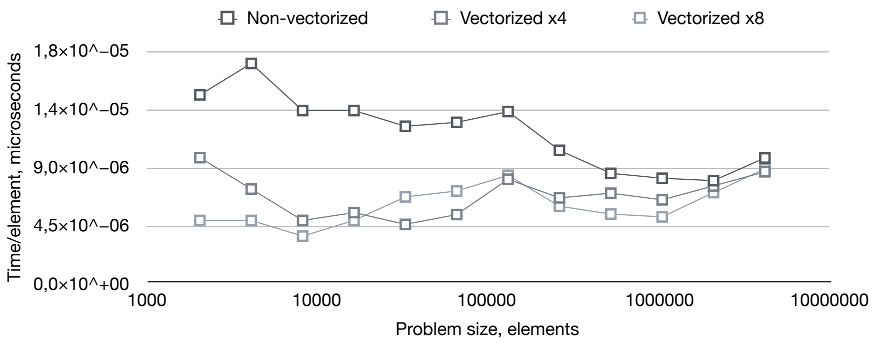


Figure 15.11: Element-wise vector addition, three variants. Execution time normalized (per element).

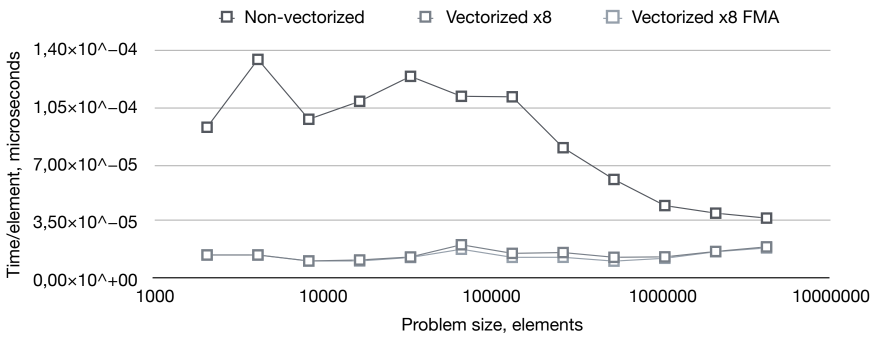


Figure 15.12: Element-wise complex vector multiplication, three variants. Execution time normalized (per element).

tor instructions and eight element variants does not matter as much, as the best performer is inconsistent. It is clear that more computation is required to get the most out of manual vectorization.

We also evaluate complex number multiplication (Figure 15.12). The complex numbers are stored in cartesian form and multiplied element-wise according to $(a+bi)\times(c+di) = (ac-bd)+(ad+bc)i$. There are more vector instructions to amortize the register transfer overhead over in this case, even though the number of inputs is doubled. An alternate version with FMA instructions provides more efficient computation but at the cost of reducing this amortization factor.

15.5.2 Median filtering

To demonstrate and evaluate the application of multi-variant user functions to provide different algorithmic approaches to the same computation, we look at the median filtering operation on images. For each pixel in the out-

Table 15.2: User function variants for median filtering.

Variant	**Time**	**Memory**	**Dependencies**
Double loop	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	None
Histogram	$\mathcal{O}(n + \|D\|)$	$\mathcal{O}(\|D\|)$	None
`qsort`	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	C standard library

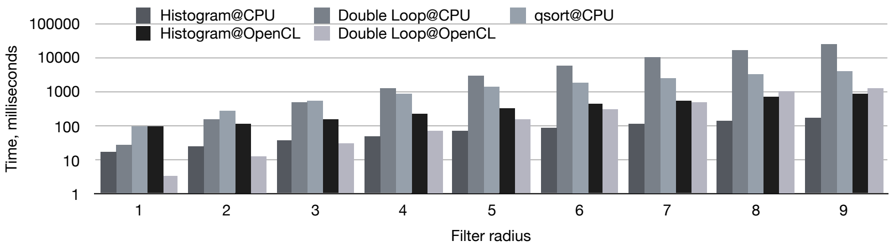


Figure 15.13: Median filtering using different median computation algorithms.

put image, the filter selects the *median* value of all pixels in a region surrounding the corresponding pixel in the input image. The region is defined by a *radius*, the same in both x and y dimensions. Using the `MapOverlap` skeleton, the image filter is then implemented directly by providing the median-finding algorithm as the user function. This can be done in several ways: by sorting the elements in the region, brute-force counting search, or by a histogram collection, among others. The characteristics of the aforementioned three approaches are compared in Table 15.2 (in the table, n denotes input size and $|D|$ denotes the size of the value domain).

A comparison of execution times for the different variants is presented in Figure 15.13. The OpenCL variants target a single Nvidia Tesla K20c GPU. The radius is varied in the range 1-9 pixels, but note that this has an effect in two dimensions and will scale the input region in the user function quadratically. The input image is fixed at 512×512 pixels, in 24-bit RGB format. The results show that there is no algorithm that is optimal across both backends; we even see that, on the GPU, the best variant varies with the filter radius.

15.6 Application benchmarks of SkePU 3

The results in this section were first published in the paper *SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters* [59].

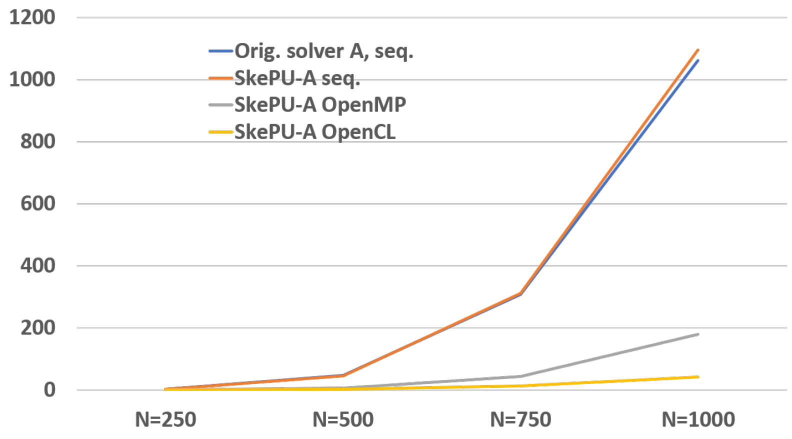


Figure 15.14: Execution times (seconds) of the SkePU 3 port of Variant A of the Embedded Runge-Kutta ODE solver implementation in the *Libsolve* library [88], solving the Brusselator 2D-MIX problem for 4 different system sizes.

15.6.1 Libsolve ODE solver

Figure 15.14 shows SkePU 3 performance results for an embedded ODE solver from the *Libsolve* library[2] [88], solving the Brusselator 2D-MIX problem with 7 stage vectors for four different system sizes ($N = 250$, 500, 750, 1000 rows) on a server with 12 cores Xeon(R) CPU E5-2630L and a K20c GPU, with pre-selected single-node CPU and GPU backends respectively. The solver core uses 9 different skeleton instances (of `Map`, `Reduce` and `MapReduce`) with an average of 63 calls to skeleton instances per time step; it iterates over 1976 time steps in total for the largest scenario in Figure 15.14, for which it performs 124,532 calls to skeleton instances in total.

15.6.2 N-body

Figure 15.15 shows performance results for the N-body scenario of Section 4.8 using the OpenMP backend, taken on the same server. There is a slight increase in execution time, although too small to account for an inlining issue (discussed in Section 4.8). A likely explanation for the slowdown is due to the change in memory access pattern. Depending on the environment, the more significant improvement in memory footprint might be enough to prefer the MapPairsReduce variant.

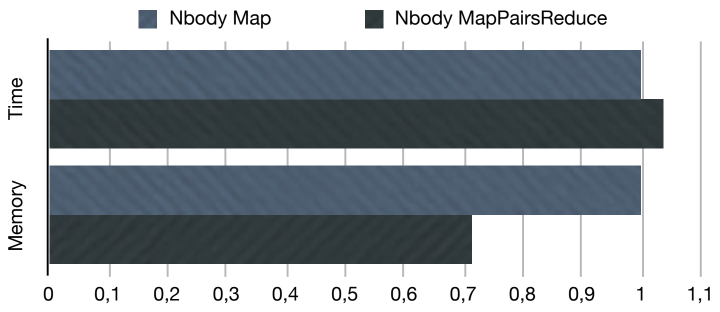


Figure 15.15: Normalized execution time and memory footprint for two variants of N-body: the `Map` variant (Listing D.1) and `MapPairsReduce` variant (Listing D.2).

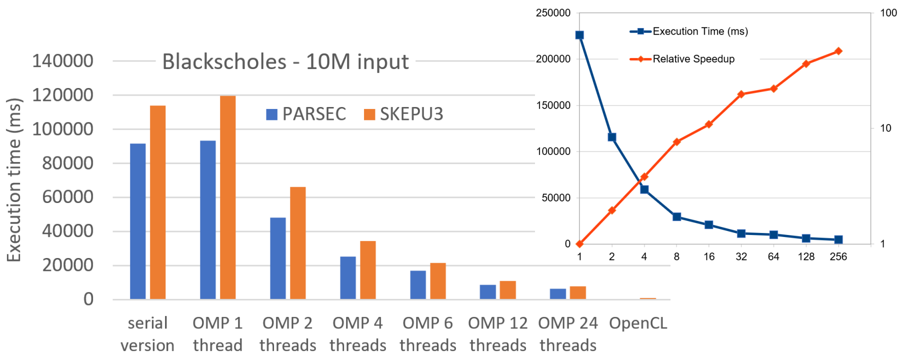


Figure 15.16: Execution time (ms) of the SkePU 3 port of the PARSEC benchmark Blackscholes on its largest input set. Left: Time with serial, OpenMP, OpenCL backends in SkePU and for manually multithreaded code in PARSEC. Right: Time and speedup with MPI backend on the cluster of Figure 15.18.

15.6.3 Blackscholes and Streamcluster

Execution time results for SkePU 3 ports of PARSEC benchmarks Blackscholes and Streamcluster on the same server can be found in Figures 15.16 and 15.17. The results show that the SkePU abstraction overhead compared to the hand-multithreaded PARSEC code is small (Blackscholes) or very small (Streamcluster), and that SkePU provides further targets for free (here, OpenCL for Blackscholes). The Streamcluster benchmark also exhibits a common problem encountered in SkePU-izing legacy C/C++ code: arrays containing a pointer-based data structure (e.g., a directed graph), if packaged e.g. in a `Vector` container, work very well with the OpenMP backend but are not portable to execution on e.g. a GPU with a different address

[2]Libsolve repository: https://github.com/UBT-AI2/rk

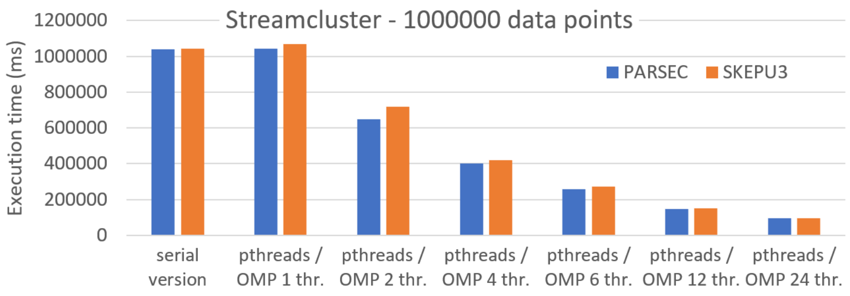


Figure 15.17: Execution times of the SkePU 3 port of the PARSEC benchmark Streamcluster on 10^6 data points.

space, as host addresses are not portable to device memory. For such cases, more advanced container types (e.g., directed graphs) would be required, which is left for future work.

15.6.4 Brain simulation

The results in this section were first published in the paper *Portable Exploitation of Parallel and Heterogeneous HPC Architectures in Neural Simulation Using SkePU* [121].

Figure 15.18 shows the scaling behavior of the SkePU 3 port of a brain simulation mini-application [121] performing 200 time steps with 90000 neurons and dense synapse connectivity using up to 32 nodes (each node having two Xeon Gold 6130 with 16 cores each) of the *Tetralith* cluster at NSC Linköping. The version that uses the `MatRow` container proxy benefits from more scalable communication compared to using the default `Mat` container. For comparison, the diagram also shows a manual MPI parallelization of the SkePU code (i.e., outer-MPI SkePU) where the communication structure corresponds to that of the MatRow version; while the scaling behavior is similar, it also shows that the execution time overhead of using SkePU with the StarPU-MPI based backend is here up to a factor of 2.

15.6.5 CO_2 capture

The results in this section were first published in the paper *EXA2PRO: A Framework for High Development Productivity on Heterogeneous Computing Systems* [122]. The experiments were conducted by our partners in the EXA2PRO project.

This application pertains to CO_2 capture systems described in [42], implemented with advanced programming models and evaluated on hetero-

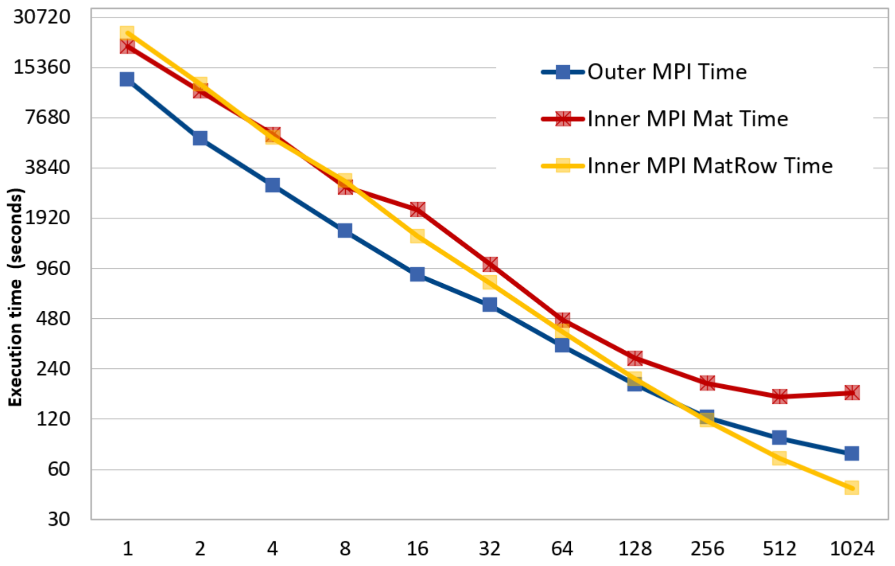


Figure 15.18: Execution time (in seconds, logarithmic scale) of the SkePU 3 port of a brain simulation mini-application [121] performing 200 time steps with 90000 neurons using up to 32 nodes (each with 32 cores) of the Tetralith cluster. "Outer MPI" refers to a manual MPI parallelization, the two "Inner-MPI" versions use SkePU's StarPU-MPI backend instead.

geneous cluster systems to assess their scalability and speedup. Problem size ranged from 4×10^4 to 4×10^6 algebraic equations, whereas tests are performed in up to 1000 CPU threads and to GPUs in a local cluster[3], in addition to the ARIS[4] and Piz Daint[5] supercomputers. The Map skeleton of SkePU is used as illustrated in Listing D.5.

In the CO_2 capture case, the core operation ported to SkePU is represented by the non-linear system of algebraic equations that are used to model the chemical process system [42]. The ported functions are used in a sequence of two calculation stages. In the first stage, the functions are used within a constrained optimization problem formulation, to determine the optimum solution under steady-state operation. In the second stage, the *Karush-Kuhn-Tucker* optimality conditions [14] are used to transform the problem into an equivalent algebraic formulation [139]. In both stages, the constraints of the optimization problem and their gradients are implemented in SkePU. The algorithms used in the two stages are *IPOPT* [154] and *PITCON* [133]. The former calls the functions in SkePU to calculate the

[3] 4-core Intel Xeon E-2174G@3.8 GHz and Nvidia Quadro P620
[4] 20-core Intel Xeon E5-2680v2@2.8 GHz
[5] 12-core Intel Xeon E5-2690v3@2.6 GHz and Nvidia Tesla P100

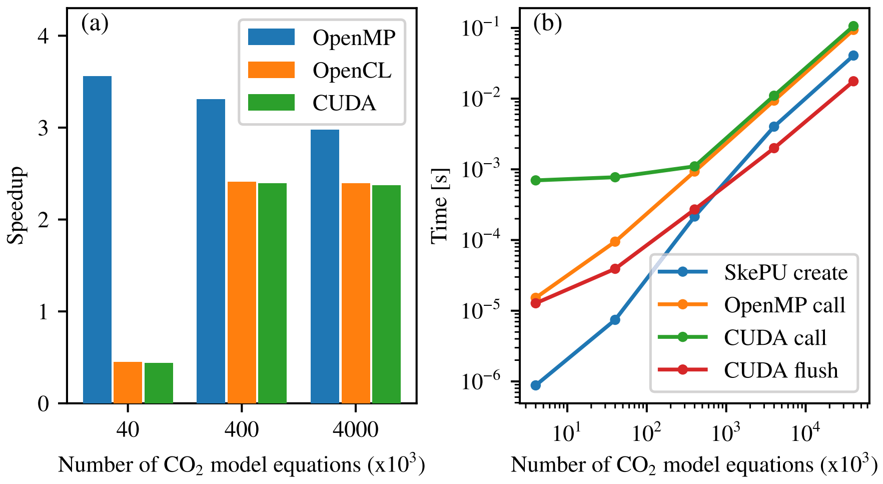


Figure 15.19: CO$_2$ capture [122]: (a) Speedup using SkePU for various backends on a single node, considering all tasks of Listing D.5. (b) Breakdown of time for tasks create of SkePU vector, call with OpenMP, call with CUDA and flush with CUDA. (Results on a single node of the local cluster for both figures).

Hessian and to perform the Newton step for the solution of the non-linear equations.

Figure 15.19a depicts the speedup based on the total execution time, using SkePU parallel backends over single-core execution. The total execution time comprises the durations of data allocation, skeleton invocation, and flushing data to CPU memory if GPUs are used.

Figure 15.20a shows that when only the call task is considered, speedup is observed due to the use of SkePU in both the OpenMP and CUDA cases. The same pattern also appears in Figure 15.20b, but Piz Daint enables higher speedup, especially for the CUDA case and large number of equations, thanks to the faster GPU.

Figure 15.20d shows the execution time, in a log-log plot, of 1000 simulations of a CO$_2$ capture process model consisting of 4×10^8 equations for an increasing number of CPU cores and up to 64 nodes. Overall, the method scales nicely up to 200 cores. However, as the number of cores increases, and a fixed workload is distributed to the parallel cores, the task size per core decreases. Since the task size becomes lower than 1ms, the runtime overhead per task [151] has significant impact on the execution time.

The effort of applying SkePU to the CO$_2$ capture was performed by a senior engineer, already familiar with the application, that required about 2 weeks of training on SkePU and 4 weeks of development.

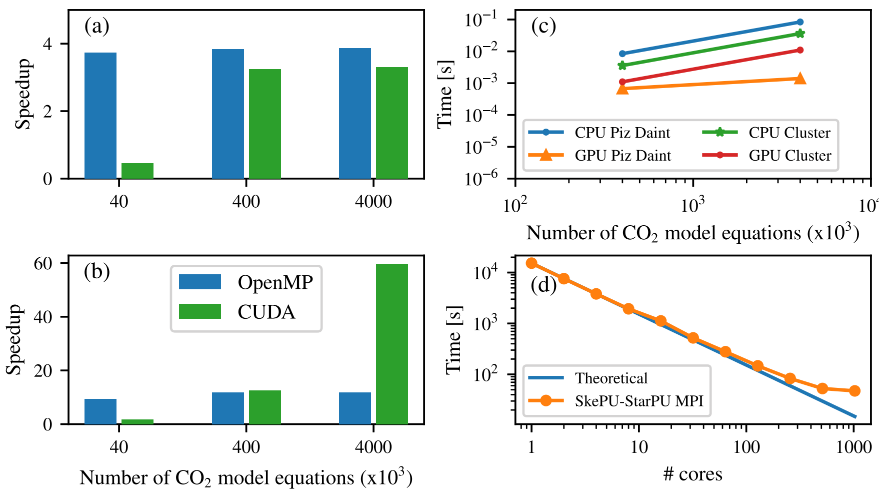


Figure 15.20: CO_2 capture [122]: (a) Speedup when only the call task is considered in all cases (local cluster). (b) Speedup for increasing number of equations when only the call task is considered (Piz Daint). (c) Execution time on CPUs and GPUs of Piz Daint and local cluster. (d) Execution time on ARIS using SkePU-StarPU MPI backend for multiple MPI nodes.

15.6.6 Supercapacitor simulation

The results in this section were first published in the paper *EXA2PRO: A Framework for High Development Productivity on Heterogeneous Computing Systems* [122]. The experiments were conducted by our partners in the EXA2PRO project.

Metalwalls [105] is a classical molecular dynamic code dedicated to the accurate simulation of electrochemical systems like supercapacitors, devices able to store energy under electrostatic form. The typical simulated system is made of two carbon electrodes immersed in an ionic liquid. At each time step, the evaluated mini-app computes (with a matrix-free conjugate gradient) the charge density on electrodes such that they conserve a constant potential.

The original code is written in Fortran 90 (Listing 15.1) and is parallelized with MPI. With the introduction of OpenACC directives, a single GPU version is also available. Listing 15.2 shows how its serial version has been rewritten in the SkePU framework with a single `MapPairsReduce`. In addition, a DFE implementation using MaxJ[6] was also produced.

[6]MaxJ is Maxeler's programming language for their DFE accelerator hardware. See e.g. Voss et al. [153].

Listing 15.1: Original code for the serial version of a single kernel of Metalwalls.

```
double V[num_atoms], z[num_atoms], q[num_atoms];
for (int i = 0; i < num_atoms; i++)
{
  vi = 0.0;
  for (int j = 0; j < num_atoms; j++)
  {
    zij = z[j] - z[i];
    pot_ij = exp(-zij*zij;) + zij*erf(zij));
    V[i] = V[i] - q[j] * pot_ij;
  }
}
```

Listing 15.2: SkePU code for the serial version of a single kernel of Metalwalls.

```
real_t map_function(
  skepu::Index2D i, const real_t zi,
  const real_t zj, skepu::Vec<real_t> q)
{
  real_t zij = zi - zj;
  real_t qj = q[i.col];
  return - qj * ( exp(-zij * zij) + zij * erf(zij) );
}

real_t plus (real_t a, real_t b) { return a + b; }

auto pairs2reduce = skepu::MapPairsReduce(map_function, plus);
pairs2reduce.setReduceMode(skepu::ReduceMode::RowWise);
pairs2reduce(V, z, z, q);
```

These implementations have been evaluated on the test-case used in production [112] with 42490 electrode atoms. In the context of material science for supercapacitors, this is a large test case. It is however small considering computing platform capabilities and is thus a limiting factor for scalability.

Results are given in Figure 15.21. The SkePU implementation degrades performance by nearly a factor three. Data locality of the generated implementation cannot yet compete with the original optimized version. However, in terms of programming effort, the compute intensive part of the application (2000 LOC in total) was successfully ported to SkePU in about two weeks. The SkePU version (450 kernel LOC) is much simpler and slightly shorter as it is very close to a serial implementation.

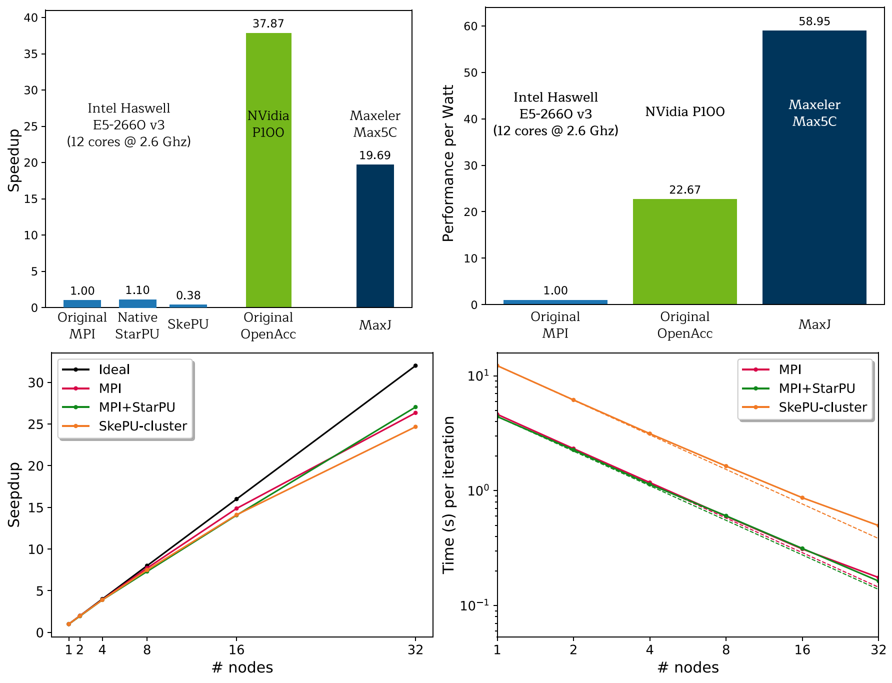


Figure 15.21: Comparison of software technologies on Metalwalls in terms of execution time (top left), performance per Watt (top right), multi-node speedup (bottom left) and multi-node time to solution on Piz Daint. [122]

15.6.7 Conjugate gradient

To gain insight into the performance potential of the SkePU-BLAS interface part of the standard library (see Section 6.3), we evaluated the conjugate gradient solver (slightly modified from the code in Listing D.4) implemented using the BLAS API. The program was run on a 12-core Xeon server (two six-core Xeon E5-2630L CPUs with two-way hardware multi-threading, thus 24 hyper-threads) with a Nvidia K20c GPU and 64 GiB main memory. The parameters were set to a problem size of N=25000 and a maximum number of 100 iterations, executed on sequential CPU, OpenMP backend with various thread counts, and OpenCL and CUDA backends for GPU. The system ran Ubuntu 18.04.5 LTS; GCC 10.3.0 was used as CPU backend compiler with -O3 optimization level. The diagram in Figure 15.22 shows the median times of 5 runs. We observe a speedup of about 8x if using all CPU hardware threads; execution on GPU is only a little faster.

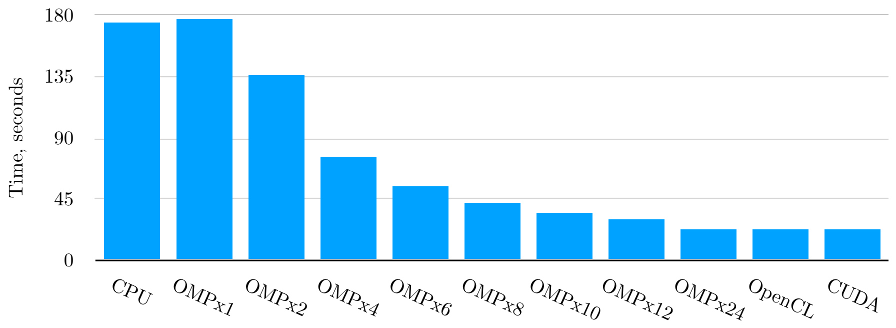


Figure 15.22: Performance evaluation of conjugate gradient solver implemented in SkePU using the BLAS API.

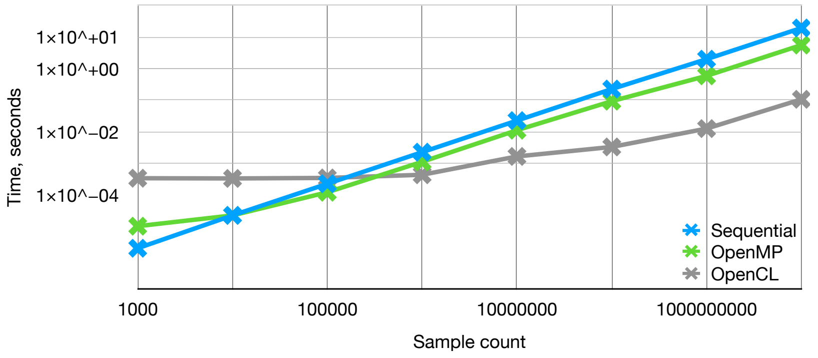


Figure 15.23: Monte-Carlo Pi calculation with varying sample count on different backends.

15.7 Experimental evaluation of deterministic PRNG

The results in this section were first presented in the paper *A Deterministic Portable Parallel Pseudo-Random Number Generator for Pattern-Based Programming of Heterogeneous Parallel Systems* [63].

For the performance evaluations in this section, we use the same server as in Section 15.6.7. Results are presented as the median of several measurement runs (varying between the programs).

15.7.1 Monte-Carlo Pi approximation

We begin with the probabilistic Pi calculation from Section 13.4.1. SkePU code using the new `skepu::Random` API is shown in Listing 13.6.

Figure 15.23 contains the performance results from executing the SkePUized program on various backends. The Monte-Carlo Pi calculation

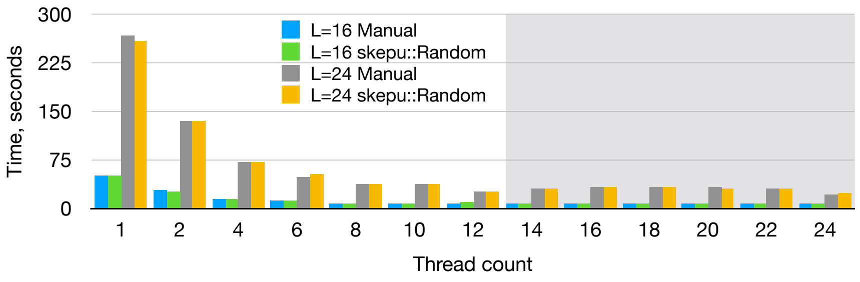


Figure 15.24: Time (seconds) for 10 iterations of LQCD with lattice sizes $L = 16$ and $L = 24$ for varying number of hardware threads in the OpenMP backend.

algorithm is an interesting stress test due to the random number generation dominating the total work. The application scales well on the GPU using the OpenCL backend (up to 180x speedup compared to sequential in the presented results), even though the work done in the user function is very lightweight.

15.7.2 LQCD Mini-Application

For the LQCD mini-application introduced in Section 13.4.2, SkePU code using the new PRNG API is shown in Listing 15.3.

Figure 15.24 shows the times of 10 iterations of LQCD with the OpenMP backend, comparing the manual workaround of Listing 13.4 to the new version using `skepu::Random` of Listing 15.3. We can see that no new overheads are introduced while code complexity decreases (see Sect. 15.7.5).

15.7.3 Miller-Rabin primality testing

The Miller-Rabin primality test [132] is a probabilistic algorithm to determine for a given number if it is likely prime or not. The actual test gets two inputs: n, the number to be tested for primality, and a value a in the range $\{2...n-2\}$. The test performs a computation on a and n, and depending on the result it outputs "n is prime" or "n is composite". While the latter answer is always true, there is a certain probability that the former answer is wrong, and this probability can be reduced by doing the computation repeatedly with randomly chosen a, see Listing 15.4. This can be easily parallelized, as the t iterations are independent (except for calls to the PRNG), but for comparability it is helpful that the random choices are similar to the sequential version.

Listing 15.3: Markov-chain-based LQCD application with new SkePU PRNG API

```
// Data management:
struct localGauge; // 36 double-precision complex numbers
skepu::Tensor4<localGauge> gaugeField( L, L, L, L);

// Gauge randomization:
localGauge randomizeLocalGauge( skepu::Random<> *prng)
{
    localGauge gaugeNew;
    for (int idx = 0; idx < 36; idx++) {
        gaugeNew.at(idx).re = prng->getNormalized();
        gaugeNew.at(idx).im = prng->getNormalized();
    }
    return gaugeNew;
}

// Metropolis step:
localGauge localUpdate( skepu::Random<73>& prng,
                        skepu::Region4D<localGauge> stencil )
{
    localGaugeAndPRNG update = randomizeLocalGauge( prng);
    double limen = someMatrixArithmetic( stencil, update);
    if (prng.getNormalized() >= limen) {
        stencil( 0,0,0,0) = update;
    }
    return stencil( 0,0,0,0);
}

...
skepu::PRNG prng;
auto metropolisUpdate = skepu::MapOverlap( localUpdate);
metropolisUpdate.setPRNG( prng);
for (int iter = 0; iter < Niter; iter++) {
    metropolisUpdate( gaugeField, gaugeField);
}
```

Listing 15.4: Pseudocode of Miller-Rabin probabilistic primality testing

```
int test( int n )
{
   result = true;
   for (int i = 1; i <= t; i++) {
      int a = rand();
      result = result & millerrabin( n, a );
      // if (result == false) break;
   }
   return result;
}
```

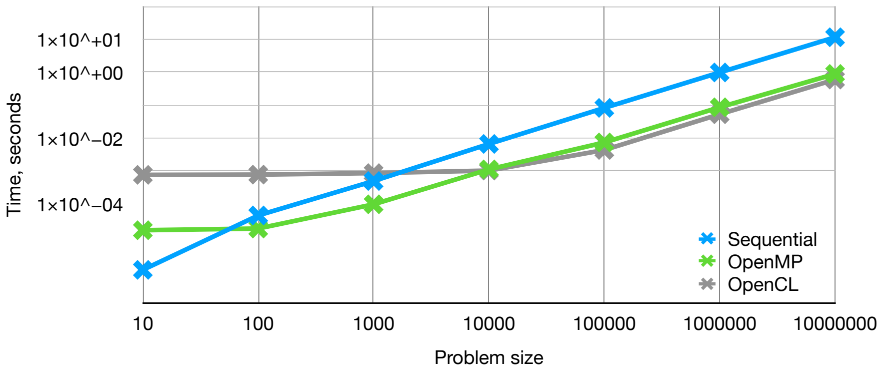


Figure 15.25: Miller-Rabin primality test with varying sample count on different backends.

Our SkePU implementation of the Miller-Rabin algorithm is largely based on an open-source implementation in C++ by Larsen[7] where the main Monte-Carlo parallelism is expressed by a `MapReduce<0>` skeleton instance.

Parallel performance of the SkePUized Miller-Rabin application can be seen in Figure 15.25. Instruction flow is highly divergent throughout the algorithm due to data-dependent control flow, which is challenging for the GPU backend: it is just barely faster than multi-core CPU computation. This property distinguishes the program from the Monte-Carlo sampling algorithm wherein the PRNG values have no effect on control flow. For the multi-core OpenMP backend we observe speedup up to 13x.

15.7.4 Natural noise generation

The PRNG construct now built-in to SkePU generates uniformly distributed real or integer values ("white" noise; Figure 15.26a). When other distributions are desired, post-processing of the generated data in application space can be a solution. One such scenario is for the generation of natural-looking noise patterns where the value distribution is dependent on factors such as signal frequency[8]. One way of generating such "colored" noise is with a gradient noise algorithm, also known as *Perlin noise* [127]. The algorithm (Listing 15.5) first generates n-dimensional grids of uniformly distributed values at different grid densities. The values are interpreted as gradients for the resulting n-dimensional noise pattern, and for each sample point, the neighbor gradients are interpolated to produce a noise value. The sampling process is repeated for each grid density (frequency) level, taking a

[7]C.S. Larsen: The Miller-Rabin primality test in C++. `https://github.com/cslarsen/miller-rabin`

[8]Used for example in computer-generated image production or as initial values in simulations.

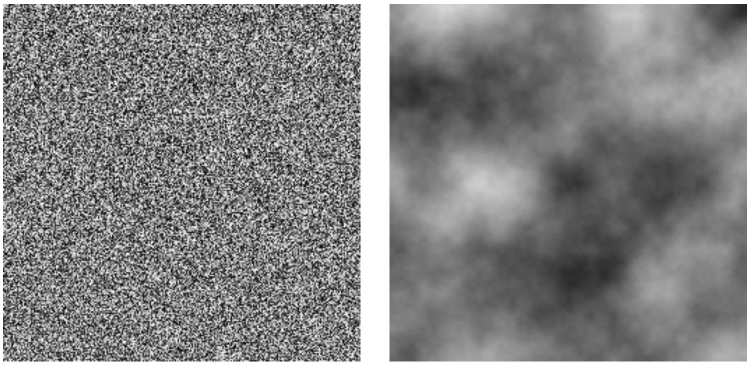

(a) Uniformly distributed noise. (b) Colored (approximately Perlin) noise.

Figure 15.26: Two-dimensional noise variants generated by a SkePU application.

Listing 15.5: Skeleton call site in the 3D natural noise generator program.

```
skepu::Tensor3<float> noise(outDepth, outHeight, outWidth, 0);
auto grid_gen  = skepu::Map(generate_grid_point_value); // uses PRNG
auto noise_gen = skepu::Map(render_value_noise_3d);
for (int i = 0; i < octaves; ++i)
{
  // ... (computation of grid sizes and amplitude omitted)
  skepu::Tensor3<float> grid(gridWidth, gridHeight, gridDepth);
  grid_gen(grid); // consumes PRNG values
  noise_gen(noise, noise, grid, amplitude, outDepth, outHeight, outWidth);
}
```

weighted sum of the individual samples as the resulting output value. A typical output is found in Figure 15.26b.

Performance evaluation of the noise generator is done on a 3D domain. It produces 256 time-linked textures, with the spatial domain as a square with side length sampled at powers of two. The program generates the entire 3D domain in one sweep with the SkePU tensor container, with 10 superimposed octaves of noise. The results in Figure 15.27 indicate very good scalability with just over 15x speedup on OpenMP and 43x on OpenCL.

15.7.5 Programmability evaluation

In all[9] of Pi calculation, Lattice QCD and Miller-Rabin primality test, the implementations see a reduction in lines-of-code count after applying the SkePU PRNG API. This effect primarily comes from abstracting the imple-

[9]The natural noise generator was written without a prior reference implementation.

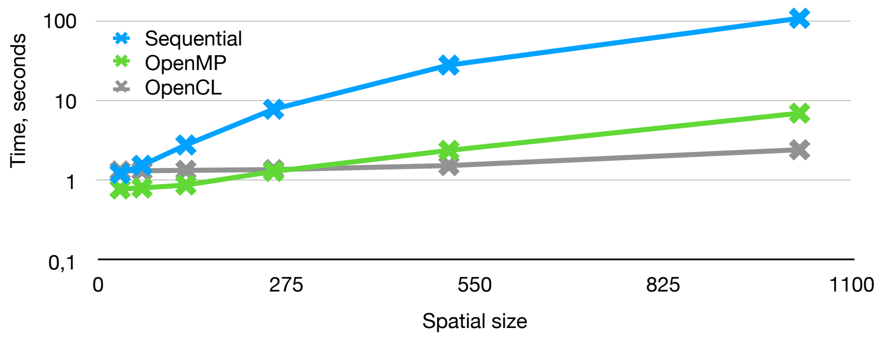


Figure 15.27: 3D natural noise generation results, varying the two spatial dimension side lengths. Vertical time axis is in logarithmic scale.

mentation details of the PRNG engine itself. The entire code base of the SkePUized LQCD application is reported by `sloccount` to be 1,212 lines of code before applying the new `skepu::Random` API, and 1,137 afterwards. This amounts to a reduction by 6.2%. In addition, the change simplifies the data structure hierarchy, and fewer skeleton calls and user function declarations are necessary.

15.8 SkePU-GPI cluster backend

The results in this section are selected from the master's thesis project of *Joel Almqvist* [3].

Experimental evaluation of the SkePU-GPI prototype (Section 9.3) was conducted on the Tetralith cluster. Here, matrix-vector multiplication (Figure 15.28a), matrix-matrix multiplication (Figure 15.28b), and N-body simulation (Figure 15.28c) are presented. The GPI backend was compared against the StarPI-MPI backend (Section 9.2), as they both target the same type of platforms: large-scale cluster systems.

At this point, the results are inconclusive. Both cluster backends demonstrate performance overhead for small node counts, but not always for the same problems. The general trend of the GPI backend is that it shows very good scaling behavior when the node count is increased, but as the sequential execution times are high, it needs a relatively large number of nodes to reach competitive performance. For one of the problems, N-body simulation, the GPI prototype notably outperforms the StarPU-MPI backend significantly.

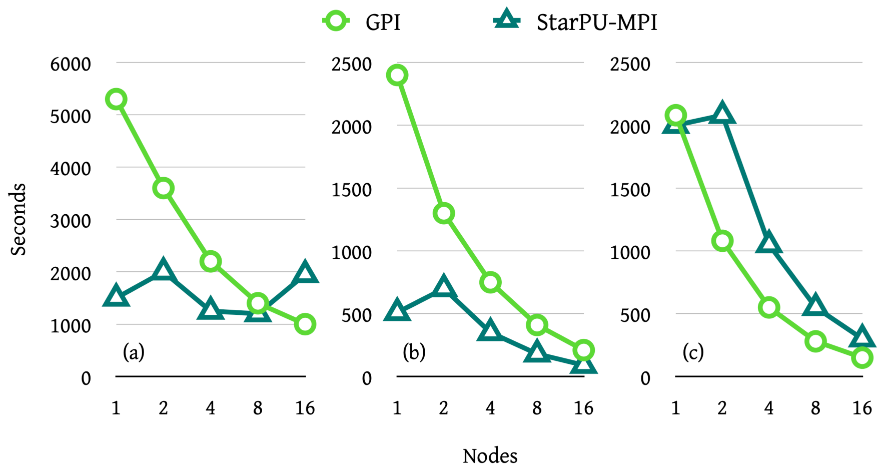


Figure 15.28: Performance of the GPI backend prototype compared to the StarPU-MPI backend.
a) Matrix-vector multiplication program of size 50000 for 10^4 iterations.
b) Matrix-matrix multiplication program of size 18000×18000.
c) N-body program with 2×10^5 particles running for 20 iterations.

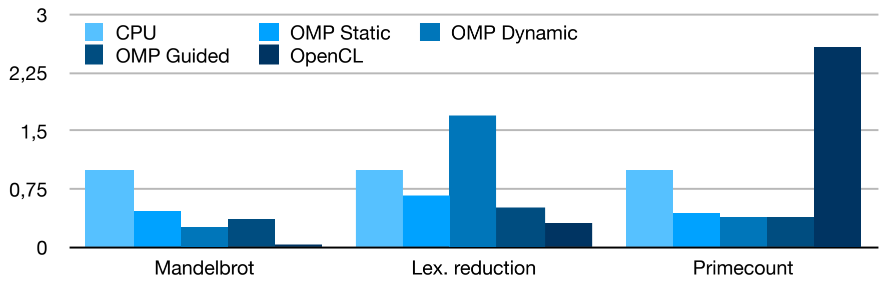


Figure 15.29: Execution time (normalized to the sequential CPU backend time) for three irregular-load benchmarks.

15.9 Microbenchmarks of SkePU 3

The results in this section are first published in *SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters* [59].

15.9.1 OpenMP scheduling modes

For the same machine, Figure 15.29 shows the positive performance effect of using dynamic scheduling in three data-parallel benchmarks with irregular workload, in spite of the runtime overhead of dynamic scheduling: (1)

Table 15.3: Microbenchmark results of vector initialization, seconds.

	With GPU backends	No GPU backends
Seq. consistency `v[i]`	0.899	0.308
Weak consistency `v(i)`	0.313	0.310

Generating a 1024×1024 Mandelbrot image using the SkePU 3 `Map` OpenMP backend with different scheduling modes. Dynamic scheduling (chunksize 16) outperforms the static default mode. (2) Lexicographic reduction finding the maximum among 10^8 date/time tuples. `Guided` dynamic scheduling (chunksize 8) outperforms the static default mode. (3) Counting prime numbers using `MapReduce` where dynamic scheduling performs best. Results for the sequential CPU and OpenCL backends are provided as reference.

15.9.2 SkePU memory consistency model

To illustrate the motivation behind the change of consistency model for SkePU smart containers (Section 5.3), we have measured the execution time through a microbenchmark. Allocating and initializing the elements of a SkePU vector using a simple for-loop results in the numbers in Table 15.3. If the SkePU application is compiled without either GPU or CUDA backends there is no appreciable overhead, but as soon as those device copies are present it is approximately 3x faster to use non-managed access operators.

15.10 Variadic tuner prototype

The results in this section are selected from the master's thesis project of *Basel Nsralla* [116].

Figure 15.30 shows the prototype variadic auto-tuner (Chapter 14) applied to a Mandelbrot fractal generation program for an initial performance evaluation. There are outlier data points for small problem sizes, suggesting that a more thorough experimental setup is required, but overall for large problems the tuner finds the fastest backend.

15.11 High-level skeleton fusion

The impact of skeleton fusion in SkePU (covered in Chapter 11) is evaluated by utilizing the image filtering component of SkePU's standard library, introduced in Section 6.4. The experimental setup uses a filter pipeline with eight stages, each modeled by a `Map` skeleton instance. This particular filter performs one lighten operation followed by a saturation adjustment; fur-

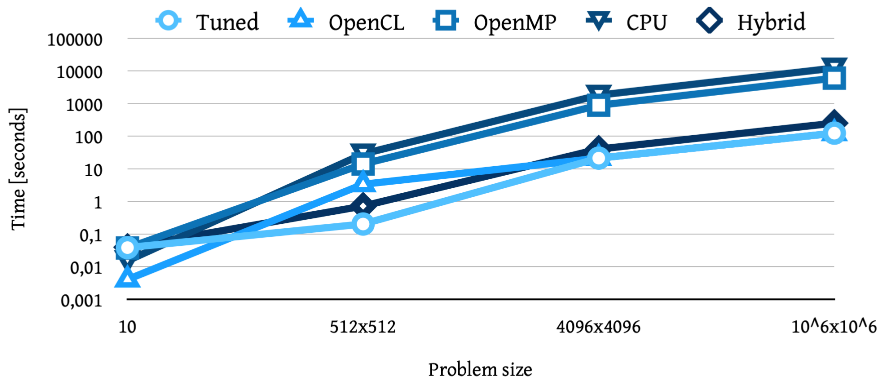


Figure 15.30: Log-scale plot of Mandelbrot execution times on different sizes and backends, and when using the variadic tuner prototype.

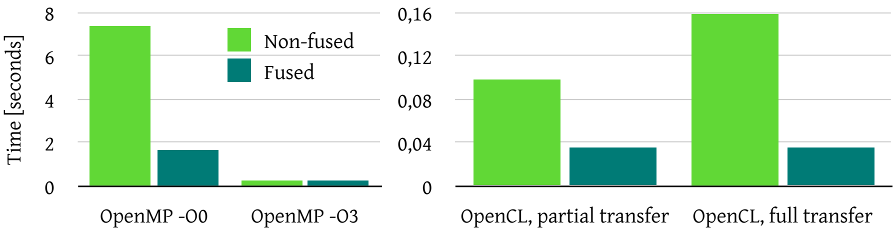


Figure 15.31: Execution time of an image filter pipeline, as separate skeleton instances and as a single fused instance (partial transfer includes only input and output data containers; full transfer includes also intermediate data containers).

ther stages are for operations such as color space conversions. The entire filter pipeline is visualized in Figure 15.32.

In the measurements we use a 16 megapixel input image and only the filter pipeline execution time is measured for OpenMP. For OpenCL we also include data transfer times to and from the GPU. The results are presented in Figure 15.31. We observe that high-level skeleton fusion has significant impact on GPU execution times; notably, elimination of the smart data-containers necessary for storing intermediate values in the non-fused variant is a big part of the gain, but not the only benefit. For OpenMP execution, we can determine that skeleton fusion acts as an optimization pass, with the impact being significantly more noticeable if the backend compiler optimizations are disabled. With optimizations on, the fused variant is only marginally faster. It is possible that fusion can make user functions overly large, preventing inlining optimizations in the backend compiler, partly nullifying the gains from data locality improvements.

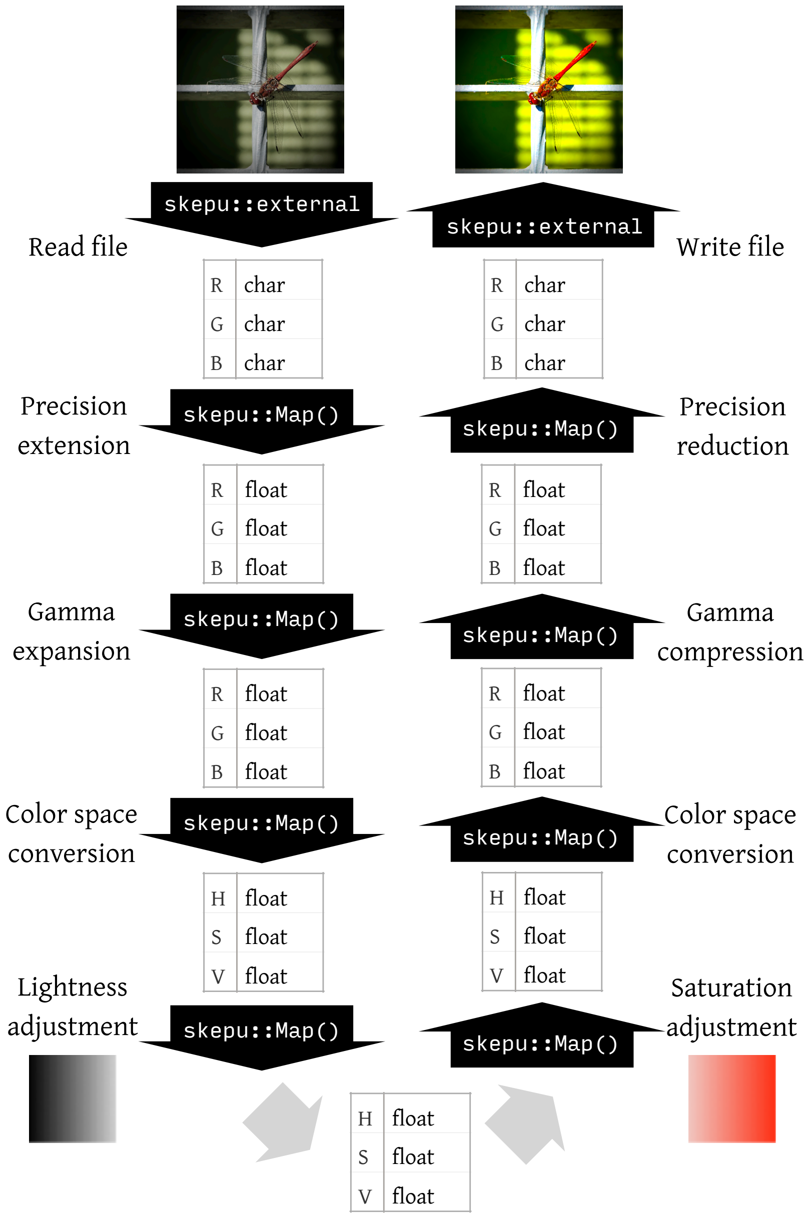


Figure 15.32: Image filter pipeline using SkePU components, including from the standard library.

16 Limitations and future work

Naturally, the contributions detailed in this thesis generally tend to be results of successful research and development efforts. However, the problem formulation given in the introduction is very broad, and one would be naïve to think that the high-level data-parallel pattern approach explored in this body of work is a completely general solution to all scenarios of accessible high-performance computing. This chapter aims to give the reader an idea of some important limitations of the work (in Section 16.1), as well as exploring directions of future work (in Section 16.2).

Most of the limitations and directions of future work discussed in this chapter come from experiences from working with this research and receiving feedback from various sources, not the least peer reviews and discussions in connection with conferences, workshops, collaborations or other such venues. While the topics brought up in this chapter focus on the SkePU framework first and foremost, the underlying problems and opportunities can apply to other projects.

16.1 Limitations

This section discusses areas where we know either SkePU, its programming model, or implementation choices have limitations when it comes to practical usage.

16.1.1 Applicability of data-parallel patterns

Focusing solely on data-parallel patterns, even when—as in the case of SkePU—the interface provides a significant amount of flexibility, limits what types of applications can be modeled in the language. The research documented in this thesis does for example not include *task-based* or *stream-based* parallelism, which both can be described by using parallel pattern constructs. Tasks and streams are useful for applications where the problem is determined to a smaller degree statically, and provides means to dynamically adapt to variations in computation or in data flow. In a data-parallel SkePU program, the call graph of parallel component invocations is largely determined at program design time (with some variations allowed by purely *sequential* control logic, such as determining the number of iterations in a heat diffusion simulation). This is not true for task parallelism, where the task graph can and often will depend on e.g. input values. Similarly, the size of input and output data sets are always known when a pattern construct is invoked, so the parallel *work* is finite and fixed between the sequential control points. A stream pattern, on the other hand, is designed to continue processing data in parallel as long as there is data to consume on ingoing links, as required by application archetypes such as audio or video processing or network packet handling.

16.1.2 Dynamic data structures

The data structures offered by SkePU and covered in this thesis have quite strict restrictions. The vectors, matrices, and tensors are all contiguous array objects, e.g. they must be represented by a compact region of memory which cannot change size. This design goes hand-in-hand with the data-parallel patterns, but it is conceptually equally possible to apply a pattern such as *map* on a linked data structure such as a list. Linked structures have advantages in that they are dynamic, being able to grow and shrink at will and at arbitrary positions. However, linked data often has poor performance properties and traversing them in parallel is either wasteful or requires a sophisticated implementation to avoid sequentialization. Thus, in practice the bigger restriction on SkePU containers is the fact that each *element* of a smart data-container is also of fixed size. Pointers can never be present in the user types of a smart data-container.

16.1.3 Limitations of language embedding

Any framework which, like SkePU, is architectured around a custom compiler stack faces several challenges. Keeping up with new language versions, integrating into complex build systems of target applications, and perhaps most of all, providing a robust compilation phase able to handle the most obscure edge-cases of complex languages (especially when the host language is C++). SkePU is for example not well suited for code bases reliant on macro expansions. The embedding choice does bring many benefits, especially for deployment, but sometimes otherwise promising research ideas are hindered by the baggage of a host language.

16.2 Future work

The need for parallelism shows no sign to reverse. In fact, the trends are suggesting a continued increase in parallelism on all levels and parallel architectures are spreading to more and more application domains.

We can identify some core areas of future work that apply to most pattern-based parallel programming environments. These are about diversification of the feature set of a framework, and we will cover three such areas with the general motivation as well as a concrete example applying to SkePU. These are *adding new backend targets* (Section 16.2.1) and *additional parallel patterns* (Section 16.2.2).

Furthermore, the programmability aspects of skeleton programming frameworks, while strong compared to manual hand-optimized programming on multiple low-level backend targets, still leave some things to be desired. *Testing, debugging, and visualization facilities* (Section 16.2.3) and *higher-level dynamic langauge interfaces* (Section 16.2.4) are therefore interesting areas of further research and implementation work.

16.2.1 Further backend targets: reconfigurable accelerators

During the EXA2PRO project, one of the options considered was to provide a skeleton programming backend for reconfigurable devices. Maxeler's[1] *dataflow engine* (DFE) hardware was considered, with applications in areas such as computational finance [12], but in the end the proposed solution (ad-hoc integration through the `Call` skeleton) is far from satisfactory from a programmability perspective. Maxeler's programming environment *MaxJ* is Java-based, making automated integration with SkePU and its collection of C-family backends difficult. Extending SkePU's flexible skeleton interface for FPGA backends remain a topic of great interest for future work, however. SkePU would be better suited to integrate with one of the OpenCL-based FPGA interfaces now available [40, 41].

[1] `https://www.maxeler.com`

16.2.2 Extending the parallel pattern set: stream parallelization

The skeleton set in SkePU is constantly being reevaluated. Occasionally, progress allows skeletons to be removed, as they are absorbed into others. This happened with `Generate` and `MapArray` from SkePU 2, being absorbed into a generalized `Map`. Conceptually, `MapOverlap` is a prime target for merging with `Map`, where the region objects could be grouped with the other container proxy arguments, with possible advantages being implementation of patterned applications which need region access into multiple input containers simultaneously.

However, most interest lies in adding all-new patterns to SkePU through additional skeletons. SkePU focuses on data parallelism, which makes it a less than ideal fit for several common applications, e.g. from popular benchmark suites. Extending SkePU for *stream parallelism* at this point appears the most interesting direction, as recent SkePU contributions including lineage-backed lazy evaluation [60] are based on related ideas. Developing SkePU for stream parallelism would extend its target use-cases to streaming workloads on embedded systems, which are following the general trend of increasing heterogeneity [136].

New smart data-containers used in conjunction with existing skeletons can also enable new application areas. An example of potential new smart data-container formats are *graph structures*.

16.2.3 Testing, debugging, and visualization

High-level parallel programming frameworks face many challenges, and most choose to focus on either performance or programmability aspects. Perhaps not enough efforts are spent on investigating correctness, through work on testing infrastructure and debugging facilities. Programmer productivity is not only decided by how few lines of code is required for a certain application, but also the path to get there. Debugging facilities for correctness and performance will be important areas for future work on SkePU and for high-level parallel frameworks in general.

16.2.4 Higher-level language interface

While C++ is undoubtedly a relatively high-level programming language compared to C or Fortran, it is also extremely large and complex. This can be a detriment to users of the language, where the sheer complexity of C++ turns away especially newer users. Industry trends also see a strong interest in programming interfaces tied to even more high-level languages and runtimes, often interpreted or just-in-time-compiled. The compiled nature of C++ adds another barrier for accessibility and ease-of-use. There is of course a downside to the use of high-level dynamic languages: any computation expressed completely in, e.g., Python will be up to several orders of magnitude slower than equivalent C++ code. Therefore, a common practice is to offer high-level interfaces as thin wrappers over optimized components often implemented in C or C++. An interesting direction of future work for SkePU and other C++-based skeleton programming environments is therefore to offer integrated high-level interfaces to increase accessibility while as far as possible preserving the performance characteristics.

17 Conclusions

We are currently in the middle of a gradual process of ever-increasing levels of parallelism and heterogeneity in computer hardware, without any signals of when or if it will stop. The changes are placing pressure on programming models to adapt at a similar pace, or else we will have no efficient means to utilize the resources at hand.

The solution proposed in this work is high-level parallel programming models. Parallel patterns are especially suitable for programmers without expert knowledge about parallel systems design, and specifically the data-parallel algorithmic skeletons and their implementation in SkePU have been presented in this thesis. Skeleton programming frameworks such as SkePU provide a way for the user to focus on the application and algorithms at hand without having to consider subtle details of communication, synchronization, load balancing, and other hardware-specific issues. Therefore, the resulting applications can attain performance-portability across the ever-widening landscape of parallel hardware configurations—from low-power embedded systems to large-scale clusters, with the entire spectrum possibly utilizing complex heterogeneous architectures—with minimal or no guidance from the programmer.

The work presented in this thesis has specifically demonstrated how skeleton programming frameworks can be extended and improve as foundational languages such as C++ evolve to become more expressive and powerful (RQ1). Our approach is based on source-to-source compilation of C++ programs (RQ2) where variadic template metaprogramming constructs for skeletons and smart data-containers are compiler-known types. Together with the multi-backend runtime library, the source-to-source compiler forms the SkePU framework.

Building on these foundations, we have introduced contributions published in peer-reviewed journals and conferences. Among these, we have means to improve hardware utilization through run-time optimizations of computational task graphs (RQ4), improved portability to hybrid CPU-GPU systems (RQ5) and large-scale cluster systems (RQ6), as well as allowing expert programmers to get the most out of the framework with multi-variant user functions in skeletons (RQ7). We have presented a deterministic parallel pseudo-random generator (RQ8) and other library functionality integrating with the skeleton constructs. The contributions are evaluated in a series of experiments ranging from microbenchmarks to full-scale scientific applications; including carbon dioxide capture, supercapacitor, and lattice quantum chromodynamics simulations as part of application-framework co-design efforts in the EU project EXA2PRO (RQ3).

The foundational improvements to SkePU presented in the thesis opened up a large field of potential new development directions and possible features. We have also described ongoing work, such as an alternate approach to cluster execution, a modernized variadic tuner, and a high-level skeleton fusion system (RQ9), contributions which are not yet published elsewhere. The SkePU programming framework is actively used as a research tool and for teaching, with several new projects just about to start.

Bibliography

[1] Ahmad Abdelfattah, Hartwig Anzt, Aurelien Bouteiller, Anthony Danalis, Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, Stephen Wood, Panruo Wu, Ichitaro Yamazaki, and Asim YarKhan. *Roadmap for the Development of a Linear Algebra Library for Exascale Computing: SLATE: Software for Linear Algebra Targeting Exascale*. SLATE Working Notes 01, ICL-UT-17-02. 2017-06 2017.

[2] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. "Fastflow: High-Level and Efficient Streaming on Multicore." In: *Programming multi-core and many-core computing systems*. John Wiley & Sons, Ltd, 2017. Chap. 13, pp. 261–280. ISBN: 9781119332015. DOI: 10.1002/9781119332015.ch13.

[3] Joel Almqvist. "Integrating SkePU's algorithmic skeletons with GPI on a cluster." LIU-IDA/LITH-EX-A–22/002–SE. MA thesis. Department of Computer and Information Science, 2022.

[4] Vasco Amaral, Beatriz Norberto, Miguel Goulão, Marco Aldinucci, Siegfried Benkner, Andrea Bracciali, Paulo Carreira, Edgars Celms, Luís Correia, Clemens Grelck, Helen Karatza, Christoph Kessler, Peter Kilpatrick, Hugo Martiniano, Ilias Mavridis, Sabri Pllana, Ana Respício, José Simão, Luís Veiga, and Ari Visa. "Programming languages for data-Intensive HPC applications: A systematic mapping study." In: *Parallel Computing* 91 (2020), p. 102584. ISSN: 0167-8191. DOI: 10.1016/j.parco.2019.102584.

[5] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. "PetaBricks: A Language and Compiler for Algorithmic Choice." In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: ACM, 2009, pp. 38–49. ISBN: 978-1-60558-392-1. DOI: `10.1145/1542476.1542481`.

[6] Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Nikolaos Nikolaidis, Aggeliki-Agathi Tzintzira, Areti Ampatzoglou, and Alexander Chatzigeorgiou. "Investigating Trade-offs between Portability, Performance and Maintainability in Exascale Systems." In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2020, pp. 59–63. DOI: `10.1109/SEAA51224.2020.00020`.

[7] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. "StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators." In: *Recent Advances in the Message Passing Interface*. Ed. by Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra. Vol. 7490. LNCS. Springer, 2012, pp. 298–299. ISBN: 978-3-642-33518-1. DOI: `10.1007/978-3-642-33518-1_40`.

[8] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures." In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198. DOI: `10.1002/cpe.1631`.

[9] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)." In: *Dagstuhl Reports* 6.4 (2016). Ed. by Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman, pp. 110–138. ISSN: 2192-5283. DOI: `10.4230/DagRep.6.4.110`.

[10] Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. "Nanos Mercurium: A Research Compiler for OpenMP." In: *Proceedings of the European Workshop on OpenMP*. Vol. 8. 2004, p. 56.

[11] Boaz Barak and Shai Halevi. "A Model and Architecture for Pseudo-Random Generation with Applications to /dev/random." In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. Alexandria, VA, USA: Association for Computing Machinery, 2005, pp. 203–212. ISBN: 1595932267. DOI: `10.1145/1102120.1102148`.

[12] Tobias Becker, Oskar Mencer, Stephen Weston, and Georgi Gaydadjiev. “Maxeler Data-Flow in Computational Finance.” In: *FPGA Based Accelerators for Financial Applications*. Ed. by Christian De Schryver. Cham: Springer International Publishing, 2015, pp. 243–266. ISBN: 978-3-319-15407-7. DOI: 10.1007/978-3-319-15407-7_11.

[13] Nathan Bell and Jared Hoberock. “Thrust: A Productivity-Oriented Library for CUDA.” In: *GPU Computing Gems, Jade Edition* (2011).

[14] L.T. Biegler. *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*. MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2010. ISBN: 9780898717020.

[15] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. “The PARSEC Benchmark Suite: Characterization and Architectural Implications.” In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’08. Toronto, Ontario, Canada: Association for Computing Machinery, 2008, pp. 72–81. ISBN: 9781605582825. DOI: 10.1145/1454115.1454128.

[16] David Broman, Peter Fritzson, Görel Hedin, and Johan Åkesson. “A Comparison of Two Metacompilation Approaches to Implementing a Complex Domain-Specific Language.” In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC ’12. Trento, Italy: Association for Computing Machinery, 2012, pp. 1919–1921. ISBN: 9781450308571. DOI: 10.1145/2245276.2232092.

[17] Denis Caromel, Ludovic Henrio, and Mario Leyton. “Type Safe Algorithmic Skeletons.” In: *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. 2008, pp. 45–53. DOI: 10.1109/PDP.2008.29.

[18] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns.” In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.07.003.

[19] William Celmaster and K.J.M Moriarty. “A method for vectorized random number generators.” In: *Journal of Computational Physics* 64.1 (1986), pp. 271–275. ISSN: 0021-9991. DOI: 10.1016/0021-9991(86)90032-X.

[20] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. “Rodinia: A benchmark suite for heterogeneous computing.” In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[21] Federico Ciccozzi, Lorenzo Addazi, Sara Abbaspour Asadollah, Björn Lisper, Abu Naser Masud, and Saad Mubeen. "A Comprehensive Exploration of Languages for Parallel Computing." In: *ACM Comput. Surv.* 55.2 (Jan. 2022). ISSN: 0360-0300. DOI: `10.1145/3485008`.

[22] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. *The Münster Skeleton Library Muesli - A Comprehensive Overview.* ERCIS Working Paper No. 7. 2009.

[23] Tadej Ciglarič, Erik Štrumbelj, et al. "An OpenCL library for parallel random number generators." In: *The Journal of Supercomputing* 75.7 (2019), pp. 3866–3881. DOI: `10.1007/s11227-019-02756-2`.

[24] Murray Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming." In: *Parallel Computing* 30.3 (2004), pp. 389–406. ISSN: 0167-8191. DOI: `10.1016/j.parco.2003.12.002`.

[25] Murray I. Cole. *Algorithmic skeletons: Structured management of parallel computation.* Pitman and MIT Press, Cambridge, Mass., 1989.

[26] EXA2PRO Consortium: Dionysios Kehagias, Dimitrios Tsoukalas, Maria Mathioudaki, Olivier Aumage, Christoph Kessler, August Ernstsson, Johan Ahlqvist, Alexander Chatzigeorgiou, Apostolos Ampatzoglou, and Nikolaos Nikolaidis. *D5.3 – Initial report on the development of front-end tools.* Tech. rep. Ares(2020)2324772. 2020.

[27] EXA2PRO Consortium: Christoph Kessler, August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, Tomas Öhberg, Lazaros Papadopoulos, Sotirios Panagitiou, Nicolas Vandenbergen, Sakis Papadopoulos, Mathieu Haefele, and Samuel Thibault. *D2.1 – Final prototype implementation of EXA2PRO high-level programming interface.* Tech. rep. Ares(2020)6195810. 2020.

[28] EXA2PRO Consortium: Christoph Kessler, August Ernstsson, Suejb Mehmeti, Lazaros Papadopoulos, Samuel Thibault, and Alexander Chatzigeorgiou. *D3.1 – Early specification of a modular composition framework architecture.* Tech. rep. Ares(2019)569103. 2020.

[29] EXA2PRO Consortium: Christoph Kessler, August Ernstsson, and Samuel Thibault. *D2.1 – Initial specification of EXA2PRO high-level programming interface.* Tech. rep. Ares(2018)5582990. 2018.

[30] EXA2PRO Consortium: Christoph Kessler, Stavroula Zouzoula, Johan Ahlqvist, August Ernstsson, Suejb Mehmeti, Oleg Sysoev, Tobias Becker, Alexander Cramb, and Nils Voss. *D3.6 – Final version of composition and performance modelling framework.* Tech. rep. Ares(2020)8012131. 2020.

[31] EXA2PRO Consortium: Suejb Mehmeti, Christoph Kessler, August Ernstsson, Johan Ahlqvist, Mahder Gebremedhin, Lazaros Papadopoulos, Samuel Thibault, and Alexander Chatzigeorgiou. *D3.2 – First version of the composition framework.* Tech. rep. Ares(2019)6780466. 2019.

[32] EXA2PRO Consortium: Suejb Memeti, Christoph Kessler, August Ernstsson, Samuel Thibault, and Henrik Henriksson. *D2.2 – Final specification of EXA2PRO high-level programming interface.* Tech. rep. Ares(2019)6780458. 2018.

[33] EXA2PRO Consortium: Lazaros Papadopoulos, et al. *D8.6 – Final report.* Tech. rep. 2021.

[34] EXA2PRO Consortium: Athanasios Salamanis, et al. *D5.4 – Initial report on verification and testing of the EXA2PRO framework.* Tech. rep. Ares(2020)4059106. 2020.

[35] EXA2PRO Consortium: Athanasios Salamanis, et al. *D5.6 – Final report on the integration of the EXA2PRO framework.* Tech. rep. Ares(2021)6027979. 2020.

[36] EXA2PRO Consortium: Athanasios Salamanis, et al. *D5.8 – Final report on verification and testing of the EXA2PRO framework.* Tech. rep. 2021.

[37] EXA2PRO Consortium: Athanasios Salamanis, Christoph Kessler, August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Samuel Thibault. *D5.2 – Initial report on the integration of the EXA2PRO framework.* Tech. rep. Ares(2020)2324766. 2020.

[38] EXA2PRO Consortium: Dimitris Tsoukalas, Angeliki Tsintzira, Alexandros Chatziantoniou, Adamantios Stavridis, Christoph Kessler, August Ernstsson, and Suejb Mehmeti. *D5.1 – Verification and testing strategy.* Tech. rep. Ares(2019)2912482. 2019.

[39] Jean Marie Couteyen Carpaye, Jean Roman, and Pierre Brenner. "Design and analysis of a task-based parallelization over a runtime system of an explicit finite-volume CFD code with adaptive time stepping." In: *Journal of Computational Science* 28 (2018), pp. 439–454. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2017.03.008.

[40] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. "From OpenCL to high-performance hardware on FPGAS." In: *22nd International Conference on Field Programmable Logic and Applications (FPL).* 2012, pp. 531–534. DOI: 10.1109/FPL.2012.6339272.

[41] Erik H D'Hollander. "Empowering Parallel Computing with Field Programmable Gate Arrays." In: *Parallel Computing: Technology Trends* 36 (2020). Ed. by I et al. Foster, p. 16. DOI: 10.3233/APC200020.

[42] Theodoros Damartzis, Athanasios I. Papadopoulos, and Panos Seferlis. "Optimum synthesis of solvent-based post-combustion CO2 capture flowsheets through a generalized modeling framework." In: *Clean Technologies and Environmental Policy* 16.7 (2014), pp. 1363–1380. DOI: 10.1007/s10098-014-0747-2.

[43] Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. "Parallelizing High-Frequency Trading Applications by Using C++11 Attributes." In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 3. 2015, pp. 140–147. DOI: 10.1109/Trustcom.2015.623.

[44] Marco Danelutto, Gabriele Mencagli, Massimo Torquati, Horacio González–Vélez, and Peter Kilpatrick. "Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming." In: *International Journal of Parallel Programming* 49.2 (2021), pp. 177–198. DOI: 10.1007/s10766-020-00684-w.

[45] Marco Danelutto and Massimo Torquati. "Structured Parallel Programming with "core" FastFlow." In: *Central European Functional Programming School*. Vol. 8606. LNCS. Springer, 2015, pp. 29–75. DOI: 10.1007/978-3-319-15940-9_2.

[46] Usman Dastgeer, Johan Enmyren, and Christoph W. Kessler. "Auto-Tuning SkePU: A Multi-Backend Skeleton Programming Framework for Multi-GPU Systems." In: IWMSE '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 25–32. ISBN: 9781450305778. DOI: 10.1145/1984693.1984697.

[47] Usman Dastgeer and Christoph Kessler. "Smart Containers and Skeleton Programming for GPU-Based Systems." In: *International Journal of Parallel Programming* 44.3 (2016), pp. 506–530. ISSN: 1573-7640. DOI: 10.1007/s10766-015-0357-6.

[48] Usman Dastgeer, Lu Li, and Christoph Kessler. "Adaptive Implementation Selection in the SkePU Skeleton Programming Library." In: *Revised Selected Papers of the 10th International Symposium on Advanced Parallel Processing Technologies - Volume 8299*. APPT 2013. Stockholm, Sweden: Springer-Verlag, 2013, pp. 170–183. ISBN: 9783642452925. DOI: 10.1007/978-3-642-45293-2_13.

[49] Usman Dastgeer, Lu Li, and Christoph Kessler. "The PEPPHER Composition Tool: Performance-Aware Composition for GPU-Based Systems." In: *Computing* 96.12 (Dec. 2014), pp. 1195–1211. ISSN: 0010-485X. DOI: 10.1007/s00607-013-0371-8.

[50] Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. "Bringing Parallel Patterns Out of the Corner: The P3ARSEC Benchmark Suite." In: *ACM Trans. Archit. Code Optim.* 14.4 (Oct. 2017). ISSN: 1544-3566. DOI: 10.1145/3132710.

[51] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

[52] Peter J. Denning. "The Locality Principle." In: *Commun. ACM* 48.7 (July 2005), pp. 19–24. ISSN: 0001-0782. DOI: 10.1145/1070838.1070856.

[53] Horst Eissfeller and Silvia Melitta Müller. "The Triangle Method for Saving Startup Time in Parallel Computers." In: *Proceedings of the Fifth Distributed Memory Computing Conference*. The Fifth Distributed Memory Computing Conference. Apr. 1990, pp. 568–572. DOI: 10.1109/DMCC.1990.555436.

[54] Kento Emoto and Kiminori Matsuzaki. "An automatic fusion mechanism for variable-length list skeletons in sketo." In: *International Journal of Parallel Programming* 42.4 (2014), pp. 546–563. DOI: 10.1007/s10766-013-0263-8.

[55] Johan Enmyren and Christoph W. Kessler. "SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems." In: *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*. HLPP '10. Baltimore, Maryland, USA: Association for Computing Machinery, 2010, pp. 5–14. ISBN: 9781450302548. DOI: 10.1145/1863482.1863487.

[56] Steffen Ernsting and Herbert Kuchen. "Algorithmic Skeletons for Multi-Core, Multi-GPU Systems and Clusters." In: *International Journal of High Performance Computing and Networking* 7.2 (Apr. 2012), pp. 129–138. ISSN: 1740-0562. DOI: 10.1504/IJHPCN.2012.046370.

[57] August Ernstsson. "Designing a Modern Skeleton Programming Framework for Parallel and Heterogeneous Systems." Licentiate Thesis. 2020. ISBN: 978-91-7929-772-5.

[58] August Ernstsson. "SkePU 2: Language Embedding and Compiler Support for Flexible and Type-Safe Skeleton Programming." LIU-IDA/LITH-EX-A–16/026–SE. MA thesis. Linköping, Sweden: Department of Computer and Information Science, 2016.

[59] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. "SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters." In: *International Journal of Parallel Programming* 49 (2021), pp. 846–866. DOI: 10.1007/s10766-021-00704-3.

[60] August Ernstsson and Christoph Kessler. "Extending smart containers for data locality-aware skeleton programming." In: *Concurrency and Computation: Practice and Experience* 31.5 (2019), e5003. DOI: 10.1002/cpe.5003.

[61] August Ernstsson and Christoph Kessler. “Multi-variant User Functions for Platform-aware Skeleton Programming.” In: *Proc. of ParCo-2019 conference, Prague, Sep. 2019, in: I. Foster et al. (Eds.), Parallel Computing: Technology Trends, series: Advances in Parallel Computing, vol. 36, IOS press.* Mar. 2020, pp. 475–484. DOI: `10.3233/APC200074`.

[62] August Ernstsson, Lu Li, and Christoph Kessler. “SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems.” In: *International Journal of Parallel Programming* 46 (2017), pp. 62–80. ISSN: 1573-7640. DOI: `10.1007/s10766-017-0490-5`.

[63] August Ernstsson, Nicolas Vandenbergen, Jörg Keller, and Christoph Kessler. “A Deterministic Portable Parallel Pseudo-Random Number Generator for Pattern-Based Programming of Heterogeneous Parallel Systems.” In: *International Journal of Parallel Programming* (). To appear.

[64] Diego Fabregat-Traver and Paolo Bientinesi. “Automatic Generation of Loop-Invariants for Matrix Operations.” In: *2011 International Conference on Computational Science and Its Applications.* June 2011, pp. 82–92. DOI: `10.1109/ICCSA.2011.41`.

[65] Roger Ferrer, Sara Royuela, Diego Caballero, Alejandro Duran, Xavier Martorell, and Eduard Ayguadé. “Mercurium: Design decisions for a S2S compiler.” In: *Cetus Users and Compiler Infastructure Workshop in conjunction with PACT*. Vol. 2011. 2011.

[66] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. “Optimizing CUDA code by kernel fusion: application on BLAS.” In: *The Journal of Supercomputing* 71.10 (2015), pp. 3934–3957. DOI: `10.1007/s11227-015-1483-z`.

[67] Philippe Flajolet and Andrew M. Odlyzko. “Random Mapping Statistics.” In: *Advances in Cryptology - Proc. EUROCRYPT ’89 Workshop on the Theory and Application of Cryptographic Techniques, Houthalen, Belgium*. Ed. by Jean-Jacques Quisquater and Joos Vandewalle. Vol. 434. LNCS. Springer, 1989, pp. 329–354. DOI: `10.1007/3-540-46885-4\_34`.

[68] Agner Fog. “Pseudo-Random Number Generators for Vector Processors and Multicore Processors.” In: *Journal of Modern Applied Statistical Methods* 14 (Dec. 2015), pp. 308–334. DOI: `10.22237/jmasm/1430454120`.

[69] Paul Frederickson, Robert Hiromoto, Thomas L. Jordan, Burton Smith, and Tony Warnock. “Pseudo-Random Trees in Monte Carlo.” In: *Parallel Comput.* 1.2 (Dec. 1984), pp. 175–180. ISSN: 0167-8191. DOI: `10.1016/S0167-8191(84)90072-3`.

[70] Shuang Gao and Gregory D. Peterson. “GASPRNG: GPU accelerated scalable parallel random number generator library.” In: *Computer Physics Comm.* 184.4 (2013), pp. 1241–1249. ISSN: 0010-4655. DOI: `10.1016/j.cpc.2012.12.001`.

[71] Mark Gates, Piotr Luszczek, Ahmad Abdelfattah, Jakub Kurzak, Jack Dongarra, Konstantin Arturov, Cris Cecka, and Chip Freitag. *C++ API for BLAS and LAPACK.* Tech. rep. 02, ICL-UT-17-03. Revision 02-21-2018. 2017-06 2017.

[72] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. “CellSort: High Performance Sorting on the Cell Processor.” In: *Proceedings of the 33rd International Conference on Very Large Data Bases.* VLDB ’07. Vienna, Austria: VLDB Endowment, 2007, pp. 1286–1297. ISBN: 978-1-59593-649-3.

[73] David Goldberg. “What Every Computer Scientist Should Know about Floating-Point Arithmetic.” In: *ACM Comput. Surv.* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300. DOI: `10.1145/103162.103163`.

[74] Horacio González-Vélez and Mario Leyton. “A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers.” In: *Software: Practice and Experience* 40.12 (2010), pp. 1135–1160. DOI: `10.1002/spe.1026`.

[75] Dominik Grewe and Michael O’Boyle. “A static task partitioning approach for heterogeneous systems using OpenCL.” In: *Compiler Construction.* Berlin, Heidelberg: Springer, 2011, pp. 286–305. ISBN: 978-3-642-19861-8. DOI: `10.1007/978-3-642-19861-8_16`.

[76] Daniel Grünewald. “BQCD with GPI: A case study.” In: *2012 International Conference on High Performance Computing & Simulation (HPCS).* 2012, pp. 388–394. DOI: `10.1109/HPCSim.2012.6266942`.

[77] Daniel Grünewald and Christian Simmendinger. “The GASPI API specification and its implementation GPI 2.0.” English. In: *7th International Conference on PGAS Programming Models.* Ed. by Michele Weiland, Adrian Jackson, and Nicholas Johnson. United Kingdom: University of Edinburgh, Oct. 2013, pp. 243–248. ISBN: 978-0-9926615-0-2.

[78] Michael Haidl and Sergei Gorlatch. “PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14.” In: *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC.* LLVM-HPC ’14. New Orleans, Louisiana: IEEE Press, 2014, pp. 1–11. ISBN: 978-1-4799-7023-0.

[79] Per Hammarlund and Björn Lisper. "On the Relation between Functional and Data Parallel Programming Languages." In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. FPCA '93. Copenhagen, Denmark: Association for Computing Machinery, 1993, pp. 210–219. ISBN: 089791595X. DOI: 10.1145/165180.165211.

[80] Masanori Hanada. *Markov Chain Monte Carlo for Dummies*. arXiv 1808.08490. 2018. arXiv: 1808.08490 [hep-th].

[81] Görel Hedin and Eva Magnusson. "JastAdd—an aspect-oriented compiler construction system." In: *Science of Computer Programming* 47.1 (2003). Special Issue on Language Descriptions, Tools and Applications (L DTA'01), pp. 37–58. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(02)00109-0.

[82] Nina Herrmann, Herbert Kuchen, and Breno A. de Melo Menezes. "Stencil Calculations with Algorithmic Skeletons for Heterogeneous Computing Environments." In: Presented at HLPP 2021. Virtual, Romania.

[83] Vladimir Janjic, Christopher Brown, and Kevin Hammond. "Lapedo: hybrid skeletons for programming heterogeneous multicore machines in Erlang." In: *Parallel Computing: On the Road to Exascale* 27 (2016), p. 185. DOI: 10.3233/978-1-61499-621-7-185.

[84] Ken Kennedy and Kathryn S. McKinley. "Maximizing loop parallelism and improving data locality via loop fusion and distribution." In: *Languages and Compilers for Parallel Computing*. Ed. by Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 301–320. ISBN: 978-3-540-48308-3. DOI: 10.1007/3-540-57659-2_18.

[85] Christoph Kessler, Lu Li, Aras Atalar, and Alin Dobre. "XPDL: Extensible Platform Description Language to Support Energy Modeling and Optimization." In: *Proc. 44th International Conference on Parallel Processing Workshops, ICPP-EMS Embedded Multicore Systems, in conjunction with ICPP-2015*. Beijing, 2015. DOI: 10.1109/ICPPW.2015.17.

[86] Ronald T. Kneusel. *Random Numbers and Computers*. Cham (CH): Springer, 2018.

[87] Donald E. Knuth. *The Art of Computer Programming Volume 2: Seminumerical Algorithms*. 3rd. Boston, MA: Addison-Wesley Longman, 1997.

[88] Matthias Korch and Thomas Rauber. "Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining." In: *J. Parallel Distributed Comput.* 66.3 (2006), pp. 444–468. DOI: 10.1016/j.jpdc.2005.09.003.

[89] Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. “High-Level Synthesis of Functional Patterns with Lift.” In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming.* ARRAY 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 35–45. ISBN: 9781450367172. DOI: 10.1145/3315454.3329957.

[90] Pierre L’Ecuyer, David Munger, Boris N. Oreshkin, and Richard J. Simard. “Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs.” In: *Math. Comput. Simul.* 135 (2017), pp. 3–17. DOI: 10.1016/j.matcom.2016.05.005.

[91] Richard E. Ladner and Michael J. Fischer. “Parallel Prefix Computation.” In: *J. ACM* 27.4 (Oct. 1980), pp. 831–838. ISSN: 0004-5411. DOI: 10.1145/322217.322232.

[92] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. “The Cache Performance and Optimizations of Blocked Algorithms.” In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS IV. Santa Clara, California, USA: ACM, 1991, pp. 63–74. ISBN: 0-89791-380-9. DOI: 10.1145/106972.106981.

[93] Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A. McKee, Sven Karlsson, and Ayal Zaks. “Parallelizing more Loops with Compiler Guided Refactoring.” In: *2012 41st International Conference on Parallel Processing.* Sept. 2012, pp. 410–419. DOI: 10.1109/ICPP.2012.48.

[94] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.” In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04).* Palo Alto, California, Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[95] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation.” In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[96] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. “Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms.” In: *Proc. 17th Symposium on Principles and Practice of Parallel Programming.* New Orleans, Louisiana, USA: ACM, 2012, pp. 193–204. ISBN: 9781450311601. DOI: 10.1145/2145816.2145841.

[97] Vasco Leitão and João L. Sobral. “SKLP: Flexible Skeletons with Pluggable Adapters.” In: Presented at HLPP 2021. Virtual, Romania.

[98] David Levine, David Callahan, and Jack Dongarra. "A comparative study of automatic vectorizing compilers." In: *Parallel Computing* 17.10 (1991), pp. 1223–1244. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(05)80035-3.

[99] Lu Li, Usman Dastgeer, and Christoph Kessler. "Pruning Strategies in Adaptive Off-Line Tuning for Optimized Composition of Components on Heterogeneous Systems." In: *2014 43rd International Conference on Parallel Processing Workshops*. 2014, pp. 255–264. DOI: 10.1109/ICPPW.2014.42.

[100] Lu Li and Christoph Kessler. "MeterPU: A generic measurement abstraction API." In: *The Journal of Supercomputing* 74.11 (2018), pp. 5643–5658. DOI: 10.1007/s11227-016-1792-x.

[101] Li Lu and Michael L. Scott. "Toward a Formal Semantic Framework for Deterministic Parallel Programming." In: *Distributed Computing*. Ed. by David Peleg. Vol. 6950. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 460–474. ISBN: 978-3-642-24100-0. DOI: 10.1007/978-3-642-24100-0_43.

[102] Martin Lücke, Michel Steuwer, and Aaron Smith. "Integrating a Functional Pattern-Based IR into MLIR." In: *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. CC 2021. Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 12–22. ISBN: 9781450383257. DOI: 10.1145/3446804.3446844.

[103] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping." In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2009, pp. 45–55.

[104] Saeed Maleki, Yaoqing Gao, Maria J. Garzar´n, Tommy Wong, and David A. Padua. "An Evaluation of Vectorizing Compilers." In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. Oct. 2011, pp. 372–382. DOI: 10.1109/PACT.2011.68.

[105] Abel Marin-Laflèche, Matthieu Haefele, Laura Scalfi, Alessandro Coretti, Thomas Dufils, Guillaume Jeanmairet, Stewart K. Reed, Serva Alessandra, Roxanne Berthin, Camille Bacon, Sara Bonella, Benjamin Rotenberg, Paul A Madden, and Mathieu Salanne. "MetalWalls: A classical molecular dynamics software dedicated to the simulation of electrochemical systems." In: *Journal of Open Source Software* 5.53 (Sept. 2020), p. 2373. DOI: 10.21105/joss.02373.

[106] Ricardo Marques, Hervé Paulino, Fernando Alexandre, and Pedro D. Medeiros. "Algorithmic Skeleton Framework for the Orchestration of GPU Computations." In: *Euro-Par 2013 Parallel Processing*. Ed. by Felix Wolf, Bernd Mohr, and Dieter an Mey. Vol. 8097. LNCS. Berlin,

Heidelberg: Springer Berlin Heidelberg, 2013, pp. 874–885. DOI: 10.1007/978-3-642-40047-6_86.

[107] Gabriel Martinez, Mark Gardner, and Wu-chun Feng. "CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures." In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. 2011, pp. 300–307. DOI: 10.1109/ICPADS.2011.48.

[108] Víctor Martínez, David Michéa, Fabrice Dupros, Olivier Aumage, Samuel Thibault, Hideo Aochi, and Philippe O.A. Navaux. "Towards Seismic Wave Modeling on Heterogeneous Many-Core Architectures Using Task-Based Runtime System." In: *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2015, pp. 1–8. DOI: 10.1109/SBAC-PAD.2015.33.

[109] Michael Mascagni and Ashok Srinivasan. "Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation." In: *ACM Trans. Math. Softw.* 26.3 (Sept. 2000), pp. 436–461. ISSN: 0098-3500. DOI: 10.1145/358407.358427.

[110] Jens Maurer and Michael Wong. *Towards support for attributes in C++ (Revision 6)*. Tech. rep. N2761. ISO/IEC JTC1/SC22/WG21, 2008.

[111] Simon McIntosh-Smith, Michael Boulton, Dan Curran, and James Price. "On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures." In: *Supercomputing*. Ed. by Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer. Cham: Springer International Publishing, 2014, pp. 53–75. ISBN: 978-3-319-07518-1. DOI: 10.1007/978-3-319-07518-1_4.

[112] Trinidad Méndez-Morales, Nidhal Ganfoud, Zhujie Li, Matthieu Haefele, Benjamin Rotenberg, and Mathieu Salanne. "Performance of microporous carbon electrodes for supercapacitors: Comparing graphene with disordered materials." In: *Energy Storage Materials* 17 (2019), pp. 88–92. ISSN: 2405-8297. DOI: 10.1016/j.ensm.2018.11.022.

[113] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. "MLlib: Machine Learning in Apache Spark." In: *J. Mach. Learn. Res.* 17.1 (Jan. 2016), pp. 1235–1241. ISSN: 1532-4435.

[114] Claudia Misale, Maurizio Drocco, Marco Aldinucci, and Guy Tremblay. "A Comparison of Big Data Frameworks on a Layered Dataflow Model." In: *Parallel Processing Letters* 27.01 (2017), p. 1740003. DOI: 10.1142/S0129626417400035.

[115] Angeles Navarro, Francisco Corbera, Andres Rodriguez, Antonio Vilches, and Rafael Asenjo. "Heterogeneous parallel_for template for CPU–GPU chips." In: *International Journal of Parallel Programming* 47.2 (2019), pp. 213–233. DOI: 10.1007/s10766-018-0555-0.

[116] Basel Nsralla. "Modernizing and Evaluating the Auto-Tuning Framework of SkePU 3." Bachelor's Thesis. Department of Computer and Information Science, 2022.

[117] Cedric Nugteren and Henk Corporaal. "Introducing 'Bones': A Parallelizing Source-to-source Compiler Based on Algorithmic Skeletons." In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. GPGPU-5. London, United Kingdom: ACM, 2012, pp. 1–10. ISBN: 978-1-4503-1233-2. DOI: 10.1145/2159430.2159431.

[118] Tomas Öhberg. "Auto-tuning Hybrid CPU-GPU Execution of Algorithmic Skeletons in SkePU." MA thesis. Department of Computer and Information Science, 2018.

[119] Tomas Öhberg, August Ernstsson, and Christoph Kessler. "Hybrid CPU–GPU execution support in the skeleton programming framework SkePU." In: *The Journal of Supercomputing* (Mar. 2019). ISSN: 1573-0484. DOI: 10.1007/s11227-019-02824-7.

[120] Matthew Felice Pace. "BSP vs MapReduce." In: *Procedia Computer Science* 9 (2012). Proceedings of the International Conference on Computational Science, ICCS 2012, pp. 246–255. ISSN: 1877-0509. DOI: 10.1016/j.procs.2012.04.026.

[121] Sotirios Panagiotou, August Ernstsson, Johan Ahlqvist, Lazaros Papadopoulos, Christoph Kessler, and Dimitrios Soudris. "Portable Exploitation of Parallel and Heterogeneous HPC Architectures in Neural Simulation Using SkePU." In: *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '20. St. Goar, Germany: Association for Computing Machinery, 2020, pp. 74–77. ISBN: 9781450371315. DOI: 10.1145/3378678.3391889.

[122] Lazaros Papadopoulos, Dimitrios Soudris, Christoph Kessler, August Ernstsson, Johan Ahlqvist, Nikos Vasilas, Athanasios I. Papadopoulos, Panos Seferlis, Charles Prouveur, Matthieu Haefele, Samuel Thibault, Athanasios Salamanis, Theodoros Ioakimidis, and Dionysios Kehagias. "EXA2PRO: A Framework for High Development Productivity on Heterogeneous Computing Systems." In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 792–804. DOI: 10.1109/TPDS.2021.3104257.

[123] Jonathan Passerat-Palmbach, Claude Mazel, and David R.C. Hill. "Pseudo-Random Number Generation on GP-GPU." In: *Proc. IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulation.* June 2011, pp. 146–153. DOI: `10.1109/PADS.2011.5936751`.

[124] Biagio Peccerillo and Sandro Bartolini. "PHAST - A Portable High-Level Modern C++ Programming Library for GPUs and Multi-Cores." In: *IEEE Transactions on Parallel and Distributed Systems* 30.1 (2019), pp. 174–189. DOI: `10.1109/TPDS.2018.2855182`.

[125] S.J. Pennycook, J.D. Sewall, and V.W. Lee. "Implications of a metric for performance portability." In: *Future Generation Computer Systems* 92 (2019), pp. 947–958. ISSN: 0167-739X. DOI: `10.1016/j.future.2017.08.007`.

[126] Alyson Pereira, Luiz Ramos, and Luís Góes. "PSkel: A stencil programming framework for CPU-GPU systems." In: *Concurrency and Computation Practice and Experience* 27 (Apr. 2015), pp. 4938–4953. DOI: `10.1002/cpe.3479`.

[127] Ken Perlin. "Improving Noise." In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques.* SIGGRAPH '02. San Antonio, Texas: Association for Computing Machinery, 2002, pp. 681–682. ISBN: 1581135211. DOI: `10.1145/566570.566636`.

[128] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. "Portable Performance on Heterogeneous Architectures." In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS '13. Houston, Texas, USA: ACM, 2013, pp. 431–444. ISBN: 978-1-4503-1870-9. DOI: `10.1145/2451116.2451162`.

[129] Ralph Potter, Paul Keir, Russell J. Bradford, and Alastair Murray. "Kernel Composition in SYCL." In: *Proceedings of the 3rd International Workshop on OpenCL.* IWOCL '15. Palo Alto, California: ACM, 2015, 11:1–11:7. ISBN: 978-1-4503-3484-6. DOI: `10.1145/2791321.2791332`.

[130] Dan Quinlan. "ROSE: Compiler support for object-oriented frameworks." In: *Parallel Processing Letters* 10.02n03 (2000), pp. 215–226. DOI: `10.1142/S0129626400000214`.

[131] Fethi A. Rabhi and Sergei Gorlatch. *Patterns and skeletons for parallel and distributed computing.* Springer-Verlag, London, UK, 2003. ISBN: 1-85233-506-8.

[132] Michael O Rabin. "Probabilistic algorithm for testing primality." In: *Journal of Number Theory* 12.1 (1980), pp. 128–138. ISSN: 0022-314X. DOI: `10.1016/0022-314X(80)90084-0`.

[133] Werner C. Rheinboldt and John V. Burkardt. “A Locally Parameterized Continuation Process.” In: *ACM Trans. Math. Software* 9.2 (1983). DOI: 10.1145/357456.357460.

[134] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. “Musket: A Domain-Specific Language for High-Level Parallel Programming with Algorithmic Skeletons.” In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC ’19. Limassol, Cyprus: Association for Computing Machinery, 2019, pp. 1534–1543. ISBN: 9781450359337. DOI: 10.1145/3297280.3297434.

[135] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. “A generic parallel pattern interface for stream and data processing.” In: *Concurrency and Computation: Practice and Experience* 29.24 (2017), e4175. DOI: 10.1002/cpe.4175.

[136] Kathrin Rosvall and Ingo Sander. “Flexible and Tradeoff-Aware Constraint-Based Design Space Exploration for Streaming Applications on Heterogeneous Platforms.” In: *ACM Trans. Des. Autom. Electron. Syst.* 23.2 (Nov. 2017). ISSN: 1084-4309. DOI: 10.1145/3133210.

[137] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. “Parallel Random Numbers: As Easy As 1, 2, 3.” In: *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. SC ’11. Seattle, Washington: ACM, 2011, 16:1–16:12. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063405.

[138] Shigeyuki Sato and Hideya Iwasaki. “A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming.” In: *Programming Languages and Systems: 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*. Ed. by Zhenjiang Hu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 79–94. ISBN: 978-3-642-10672-9. DOI: 10.1007/978-3-642-10672-9_8.

[139] Panos Seferlis and Johan Grievink. “Process design and control structure screening based on economic and state controllability criteria.” In: *Computers & Chemical Engineering* 25 (Jan. 2001), pp. 177–188. DOI: 10.1016/S0098-1354(00)00641-4.

[140] Faisal Shahzad, Markus Wittmann, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard Wellein. “PGAS implementation of SpMVM and LBM using GPI.” English. In: *7th International Conference on PGAS Programming Models*. Ed. by Michele Weiland, Adrian Jackson, and Nicholas Johnson. United Kingdom: University of Edinburgh, Oct. 2013, pp. 172–184. ISBN: 978-0-9926615-0-2.

[141] Jie Shen, Ana Lucia Varbanescu, Yutong Lu, Peng Zou, and Henk Sips. “Workload Partitioning for Accelerating Applications on Heterogeneous Platforms.” In: *IEEE Transactions on Parallel and Distributed Systems* 27.9 (2016), pp. 2766–2780. DOI: `10.1109/TPDS.2015.2509972`.

[142] Oskar Sjöström, Soon-Heum Ko, Usman Dastgeer, Lu Li, and Christoph Kessler. “Portable Parallelization of the EDGE CFD Application for GPU-based Systems using the SkePU Skeleton Programming Library.” In: *Advances in Parallel Computing, Volume 27: Parallel Computing: On the Road to Exascale. Proc. of ParCo-2015 conference, Edinburgh, UK, Sep. 2015.* Ed. by Gerhard R. Joubert, Hugh Leather, Mark Parsons, Frans Peters, and Mark Sawyer. IOS Press, Apr. 2016, pp. 135–144. DOI: `10.3233/978-1-61499-621-7-135`.

[143] Fábio Soldado, Fernando Alexandre, and Hervé Paulino. “Execution of compound multi-kernel OpenCL computations in multi-CPU/multi-GPU environments.” In: *Concurrency and Computation: Practice and Experience* 28.3 (2016), pp. 768–787. DOI: `10.1002/cpe.3612`.

[144] Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. “Program Generation for Small-scale Linear Algebra Applications.” In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: ACM, 2018, pp. 327–339. ISBN: 978-1-4503-5617-6. DOI: `10.1145/3168812`.

[145] Michel Steuwer, Malte Friese, Sebastian Albers, and Sergei Gorlatch. “Introducing and Implementing the AllPairs Skeleton for Programming Multi-GPU Systems.” In: *International Journal of Parallel Programming* 42.4 (2013), pp. 601–618. DOI: `10.1007/s10766-013-0265-6`.

[146] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. “SkelCL - A Portable Skeleton Library for High-Level GPU Programming.” In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 2011, pp. 1176–1182. DOI: `10.1109/IPDPS.2011.269`.

[147] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. “Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation.” In: *Proc. CGO 2017, Austin, USA*. IEEE, 2017. DOI: `10.1109/CGO.2017.7863730`.

[148] Larisa Stoltzfus, Bastian Hagedorn, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. “Tiling Optimizations for Stencil Computations Using Rewrite Rules in Lift.” In: *ACM Trans. Archit. Code Optim.* 16.4 (Dec. 2019). ISSN: 1544-3566. DOI: `10.1145/3368858`.

[149] Peter Thoman, Philip Salzmann, Biagio Cosenza, and Thomas Fahringer. “Celerity: High-Level C++ for Accelerator Clusters.” In: *Euro-Par 2019: Parallel Processing*. Ed. by Ramin Yahyapour. Vol. 11725. LNCS. Cham: Springer International Publishing, 2019, pp. 291–303. ISBN: 978-3-030-29400-7. DOI: 10.1007/978-3-030-29400-7_21.

[150] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. “Kokkos 3: Programming Model Extensions for the Exascale Era.” In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[151] Georgios Tzanos, Vineet Soni, Charles Prouveur, Matthieu Haefele, Stavroula Zouzoula, Lazaros Papadopoulos, Samuel Thibault, Nicolas Vandenbergen, Dirk Pleiter, and Dimitrios Soudris. “Applying StarPU runtime system to scientific applications: Experiences and lessons learned.” In: *POMCO 2020-2nd International Workshop on Parallel Optimization using/for Multi-and Many-core High Performance Computing*. HAL Id: hal-02985721. 2020.

[152] Leslie G. Valiant. “A Bridging Model for Parallel Computation.” In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 0001-0782. DOI: 10.1145/79173.79181.

[153] Nils Voss, Tobias Becker, Oskar Mencer, and Georgi Gaydadjiev. “Rapid Development of Gzip with MaxJ.” In: *Applied Reconfigurable Computing*. Ed. by Stephan Wong, Antonio Carlos Beck, Koen Bertels, and Luigi Carro. Cham: Springer International Publishing, 2017, pp. 60–71. ISBN: 978-3-319-56258-2. DOI: 10.1007/978-3-319-56258-2_6.

[154] Andreas Wächter and Lorenz Biegler. “On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming.” In: *Mathematical programming* 106 (Mar. 2006), pp. 25–57. DOI: 10.1007/s10107-004-0559-y.

[155] M Mitchell Waldrop. “The chips are down for Moore’s law.” In: *Nature News* 530.7589 (2016), pp. 144–147. DOI: 10.1038/530144a.

[156] Fabian Wrede and Steffen Ernsting. “Simultaneous CPU–GPU execution of data parallel algorithmic skeletons.” In: *International Journal of Parallel Programming* 46.1 (2018), pp. 42–61. DOI: 10.1007/s10766-016-0483-9.

[157] Fabian Wrede and Herbert Kuchen. “Towards High-Performance Code Generation for Multi-GPU Clusters Based on a Domain-Specific Language for Algorithmic Skeletons.” In: *International Journal of Parallel Programming* 48 (2020), pp. 713–728. DOI: 10.1007/s10766-020-00659-x.

[158] Wm. A. Wulf and Sally A. McKee. “Hitting the Memory Wall: Implications of the Obvious.” In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588.

[159] Yasuhito Ogata, Toshio Endo, Naoya Maruyama, and Satoshi Matsuoka. “An efficient, model-based CPU-GPU heterogeneous FFT library.” In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. Apr. 2008, pp. 1–10. DOI: 10.1109/IPDPS.2008.4536163.

[160] Tomofumi Yuki and Louis-Noël Pouchet. *Polybench 4.0*. 2015. URL: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[161] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Spark: Cluster Computing with Working Sets.” In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. Boston, MA: USENIX Association, 2010, p. 10.

[162] Da Zheng, Disa Mhembere, Joshua T. Vogelstein, Carey E. Priebe, and Randal Burns. “FlashR: Parallelize and Scale R for Machine Learning Using SSDs.” In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’18. Vienna, Austria: ACM, 2018, pp. 183–194. ISBN: 978-1-4503-4982-6. DOI: 10.1145/3178487.3178501.

[163] Xiong Zheng and Vijay Garg. “An Optimal Vector Clock Algorithm for Multithreaded Systems.” In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 2019, pp. 2188–2194. DOI: 10.1109/ICDCS.2019.00215.

[164] Judicael A. Zounmevo, Xin Zhao, Pavan Balaji, William Gropp, and Ahmad Afsahi. “Nonblocking Epochs in MPI One-Sided Communication.” In: *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 475–486. DOI: 10.1109/SC.2014.44.

A Additions and changes from the licentiate thesis

The author's licentiate thesis [57] was published about one year before this dissertation was finalized. In the Swedish postgraduate education system, a licentiate degree comprises 120 ECTS credits or two years of full-time studies, half of the 240 ECTS credits of a doctor's degree. This doctor's dissertation is written as a direct iteration upon the earlier licentiate thesis. Thus, a reader will notice significant overlap in the contents of these two books. This appendix summarizes the most important additions and changes.

A.1 New contributions

In the concluding remarks of the licentiate thesis, five possible directions of future work were discussed, in no particular order:

- SkePU standard library
- Evaluating SkePU in further application domains
- Modernize the SkePU tuner
- Skeleton fusion
- Extended programmability survey

All of these five items have been investigated to at least some degree, but most are still ongoing work.

Design and implementation of a *standard library* for SkePU has been a major focus area in the intervening year and resulted in one publication [63], about *deterministic parallel pseudo-random number generators* (Chapter 13) and further concrete contributions to the EXA2PRO project, including SkePU-BLAS (Section 6.3 and Appendix C). The standard library and its components are covered in detail as the topic of Chapter 6.

SkePU has seen further evaluation, including real-world application domains as part of the EXA2PRO project. Chapter 15 has been extended with several sections, including the EXA2PRO results and other experiments. However, there is always more work to do here, and this topic remains highly relevant future work even after the publication of this dissertation.

The *tuning infrastructure* of modern SkePU has been the topic of a Bachelor's thesis project by Basel Nsralla [116], supervised by August. The implementation is presented in Chapter 14.

Skeleton fusion opportunities within the design constraints of SkePU have been investigated and partially implemented and evaluated. The progress is documented in Chapter 11.

Finally, a master's thesis project by *Erik Tedhamre* is currently ongoing, investigating SkePU 3 in terms of performance and programmability aspects. While the full results were not ready to cover in this dissertation, a preview of a comparison study of understandability of SkePU, CUDA, and OpenMP programs is included in Chapter 15.

Work has continued on the cluster backend of SkePU. The StarPU-MPI backend has been extended and a new prototype backend using GPI separately developed in collaboration with master's thesis student Joel Almqvist. This topic is greatly expanded and dedicated its own chapter (9).

A.2 Other changes

Compared to the licentiate thesis, the structure of chapters is changed. Chapters covering the interface of SkePU have been updated with new additions and changes to the API.

B Definitions

B.1 Abbreviations

API	Application programming interface
ASIC	Application-specific integrated circuit
AST	Abstract syntax tree
DFE	Dataflow engine (Maxeler FPGA)
DSEL	Domain-specfic embedded language (also EDSL)
EU FP7	European Union Seventh Framework Programme
EXA2PRO	European Union Horizon 2020 project 801015
EXCESS	European Union FP7 project 611183
FPGA	Field-programmable gate array
GCC	GNU Compiler Collection
GPGPU	General-purpose graphics processing unit
HLPP	International Symposium on High-Level Parallel Programming and Applications
HPC	High-performance computing
ICPC	Intel C++ compiler
IDE	Integrated development environment
IEC	International Electrotechnical Commission
IR	Intermediate representation
ISO	International Organization for Standardization
LLVM	The LLVM Compiler Infrastructure

MCC	Nordic Workshop on Multi-Core Computing
MIMD	Multiple instruction streams, multiple data streams (Flynn's taxonomy)
MPI	Message passing interface (standard API)
MSI	Modified–shared–invalid (cache coherence protocol)
NVCC	Nvidia's CUDA compiler
PGAS	Partitioned global address space
PRNG	Pseudo-random number generator
SIMD	Single instruction stream, multiple data streams (Flynn's taxonomy)
SPMD	Single program, multiple data
STL	C++ Standard Template Library
TBB	Intel Threading Building Blocks

B.2 Domain-specific terminology

Accelerator
Broad term, referring to a processing unit more specialized than a general CPU. Examples: GPU, FPGA, ASIC, DSP.

Heterogeneous (system or architecture)
Containing processing units of different types, such as CPUs with efficiency cores and performance cores, or systems with one or more CPUs and one or more accelerators.

Performance-portable (parallel program)
Program which can be executed on different parallel and heterogeneous architectures with reasonable performance and without significant code reengineering effort.

Precompiler
See "source-to-source compiler".

(Algorithmic) skeleton
Parameterizable generic component with well defined semantics, for which (sometimes multiple) parallel or accelerator-specific implementations exist.

Superscalar (computer architecture)
Processor core utilizing instruction-level parallelism by duplicating execution units, thereby executing multiple instructions per clock cycle.

Source-to-source compiler
Compiler tool which does transform input source code to output source code on a similar abstraction level, such as C++ code to C or

C++ code. Compare with a typical C++ compiler producing assembly code or an executable binary.

B.3 SkePU-specific terminology

Backend
: See Section 3.2. A type of programmable computation unit targeted for parallelization by SkePU.

Container proxy (also *proxy container*)
: Lightweight backend wrapper for smart data-containers. See Section 5.2.

Elwise parameter
: See Section 4.2.2.

Lineage
: See Chapter 10.

Multi-variant (user function)
: See Chapter 12.

Random-access parameter
: See Section 4.2.1.

Smart (data-)container
: See Section 5.1. A C++ object of a type such as `skepu::Vector`, holding a collection of values of some templated type. SkePU intelligently manages the memory of the container, including distribution over clusters and copies on external devices, transparently to the programmer.

Skeleton
: See Section 4.1. A computational pattern encoded in the SkePU framework as compiler-known C++ classes. Example: `skepu::Map`.

Skeleton instance
: See Section 4.1. Callable objects created in a SkePU program by instantiating a skeleton class.

Skeleton invocation
: See Section 4.1. When a skeleton instance is applied one or more smart data-containers. The invocation may be synchronous or asynchronous.

User function

See Section 4.10. C++ function acting as an operator used when instantiating a skeleton. Applied to container elements as part of a skeleton invocation.

C SkePU-BLAS API

This appendix documents the SkePU-BLAS coverage and API interface as of the publication of this thesis. Details are subject to change in the future; please refer to the SkePU user guide and related documentation for up-to-date information.

Currently, the SkePU-BLAS coverage is limited to level-1 BLAS and dense operations from level 2 and 3. Unsupported level-2 and level-3 BLAS functions are omitted below for brevity.

CBLAS function	SkePU-BLAS signature

SROTG
DROTG
CROTG
ZROTG
Setup Givens rotation
	`template<typename T>`
	`void rotg (T *a, T *b, T *c, T *s)`

SROT DROT CSROT ZDRO Apply Givens rotation	`template<typename TX, typename TY, typename TS = scalar_type<TX, TY>> void rot ( size_type n, Vector<TX> & x, stride_type incx, Vector<TY> & y, stride_type incy, TS c, TS s )`
SROTMG DROTMG Setup modified Givens rotation	Not available
SROTM DROTM Apply modified Givens rotation	Not available
SSWAP DSWAP CSWAP ZSWAP Swap x and y	`template<typename TX, typename TY> void swap ( size_type n, Vector<TX> & x, stride_type incx, Vector<TY> & y, stride_type incy )`
SSCAL DSCAL CSSCAL CSCAL x := a * x	`template<typename TX, typename TS> void scal ( size_type n, TS alpha, Vector<TX> & x, stride_type incx )`

SCOPY
DCOPY
CCOPY
ZCOPY
y := x
	`template<typename TX, typename TY>`
	`void copy (`
	` size_type                         n,`
	` Vector<TX> BLAS_CONST&            x,`
	` stride_type                       incx,`
	` Vector<TY> &                      y,`
	` stride_type                       incy`
	`)`

SAXPY
CAXPY
DAXPY
ZAXPY
y := a * x + y
	`template<typename TX, typename TY,`
	`         typename TS = scalar_type<TX, TY>>`
	`void axpy (`
	`  size_type                         n,`
	`  TS                                alpha,`
	`  Vector<TX> BLAS_CONST&            x,`
	`  stride_type                       incx,`
	`  Vector<TY> &                      y,`
	`  stride_type                       incy`
	`)`

SDOT
DDOT
CDOTC
ZDOTC
Dot product
	`template<typename TX, typename TY>`
	`scalar_type<TX, TY> dot (`
	` size_type                         n,`
	` Vector<TX> BLAS_CONST&            x,`
	` stride_type                       incx,`
	` Vector<TY> BLAS_CONST&            y,`
	` stride_type                       incy`
	`)`

DSDOT SDSDOT Dot product with extended precision accumulation	Not available

CDOTU
ZDOTU
Dot product
	`template<typename TX, typename TY>`
	`scalar_type<TX, TY> dotu (`
	` size_type                         n,`
	` Vector<TX> BLAS_CONST&            x,`
	` stride_type                       incx,`
	` Vector<TY> BLAS_CONST&            y,`
	` stride_type                       incy`
	`)`

SNRM2
DNRM2
SCNRM2
DZNRM2
Euclidian norm
	`template<typename T>`
	`real_type<T> nrm2 (`
	`  size_type                         n,`
	`  Vector<T> BLAS_CONST&             x,`
	`  stride_type                       incx`
	`)`

SASUM
DASUM
SCASUM
DZASUM
Sum of absolute values
	`template<typename T>`
	`T asum (`
	`  size_type                         n,`
	`  Vector<T> BLAS_CONST&             x,`
	`  stride_type                       incx`
	`)`

ISAMAX
IDAMAX
ICAMAX
IZAMAX
Index of max absolute value
	`template<typename T>`
	`size_type iamax (`
	`  size_type                         n,`
	`  Vector<T> BLAS_CONST&             x,`
	`  stride_type                       incx`
	`)`

SGEMV
DGEMV
CGEMV
ZGEMV
Matrix-vector multiply
	`template<typename TA, typename TX, typename TY,`
	`         typename TS = scalar_type<TA, TX, TY>>`
	`void gemv(`
	`  blas::Op`
	`  size_type                         m,`
	`  size_type                         n,`
	`  TS                                alpha,`
	`  Matrix<TA> BLAS_CONST&            A,`
	`  size_type                         lda,`
	`  Vector<TX> BLAS_CONST&            x,`
	`  stride_type                       incx,`
	`  TS                                beta,`
	`  Vector<TY> &                      y,`
	`  stride_type                       incy`
	`)`

SGER
DGER
CGERC
ZGERC
Rank-1 update
	`template<typename TX, typename TY, typename TA,`
	`          typename TS = scalar_type<TA, TX, TY>>`
	`void ger (`
	`  size_type                          m,`
	`  size_type                          n,`
	`  TS                                 alpha,`
	`  Vector<TX> BLAS_CONST&             x,`
	`  stride_type                        incx,`
	`  Vector<TY> BLAS_CONST&             y,`
	`  stride_type                        incy,`
	`  Matrix<TA> &                       A,`
	`  stride_type                        lda`
	`)`

GCERU
ZGERU
Rank-1 update
	`template<typename TX, typename TY, typename TA,`
	`          typename TS = scalar_type<TA, TX, TY>>`
	`void geru (`
	`  size_type                          m,`
	`  size_type                          n,`
	`  TS                                 alpha,`
	`  Vector<TX> BLAS_CONST&             x,`
	`  stride_type                        incx,`
	`  Vector<TY> BLAS_CONST&             y,`
	`  stride_type                        incy,`
	`  Matrix<TA> &                       A,`
	`  stride_type                        lda`
	`)`

SGEMM
DGEMM
CGEMM
ZGEMM
Matrix-matrix multiply
	`template<typename TA, typename TB, typename TC>`
	`void gemm (`
	`  blas::Op                           transA,`
	`  blas::Op                           transB,`
	`  size_type                          m,`
	`  size_type                          n,`
	`  size_type                          k,`
	`  scalar_type<TA, TB, TC>            alpha,`
	`  Matrix<TA> BLAS_CONST&             A,`
	`  size_type                          lda,`
	`  Matrix<TB> BLAS_CONST&             B,`
	`  size_type                          ldb,`
	`  scalar_type<TA, TB, TC>            beta,`
	`  Matrix<TC>&                        C,`
	`  size_type                          ldc`
	`)`

D Application source code samples

D.1 N-body simulation

Listing D.1: N-body simulation code using `Map`.

```
// Particle data structure that is used as an element type.
struct Particle
{
  float x, y, z;
  float vx, vy, vz;
  float m;
};

constexpr float G [[skepu::userconstant]] = 1;
constexpr float delta_t [[skepu::userconstant]] = 0.1;

/*
 * Array user-function that is used for applying nbody computation,
 * All elements from parr and a single element (named 'pi') are accessible
 * to produce one output element of the same type.
 */
Particle move(skepu::Index1D index, Particle pi,
  const skepu::Vec<Particle> parr)
{
  size_t i = index.i;

  float ax = 0.0, ay = 0.0, az = 0.0;
  size_t np = parr.size;

  for (size_t j = 0; j < np; ++j)
  {
    if (i != j)
    {
      Particle pj = parr[j];

      float rij = sqrt((pi.x - pj.x) * (pi.x - pj.x)
                     + (pi.y - pj.y) * (pi.y - pj.y)
                     + (pi.z - pj.z) * (pi.z - pj.z));

      float dum = G * pi.m * pj.m / pow(rij, 3);

      ax += dum * (pi.x - pj.x);
      ay += dum * (pi.y - pj.y);
      az += dum * (pi.z - pj.z);
    }
  }

  Particle newp;
  newp.m = pi.m;

  newp.x = pi.x + delta_t * pi.vx + delta_t * delta_t / 2 * ax;
  newp.y = pi.y + delta_t * pi.vy + delta_t * delta_t / 2 * ay;
  newp.z = pi.z + delta_t * pi.vz + delta_t * delta_t / 2 * az;

  newp.vx = pi.vx + delta_t * ax;
  newp.vy = pi.vy + delta_t * ay;
  newp.vz = pi.vz + delta_t * az;

  return newp;
}

Particle init(skepu::Index1D index, size_t np)
{
  // Initialize positions and accelerations
}

auto nbody_init = skepu::Map<0>(init);
auto nbody_simulate_step = skepu::Map<1>(move);

void nbody(skepu::Vector<Particle> &particles, size_t iterations)
{
  size_t np = particles.size();
  skepu::Vector<Particle> doublebuffer(particles.size());

  nbody_init(particles, np);

  for (size_t i = 0; i < iterations; i += 2)
  {
    nbody_simulate_step(doublebuffer, particles, particles);
    nbody_simulate_step(particles, doublebuffer, doublebuffer);
  }
}
```

Listing D.2: N-body simulation code using `MapPairsReduce` in SkePU 3.

```
// Particle data structure that is used as an element type.
struct Particle
{
  float x, y, z;
  float vx, vy, vz;
  float m;
};

constexpr float G [[skepu::userconstant]] = 1;
constexpr float delta_t [[skepu::userconstant]] = 0.1;

struct Acceleration
{
  float x, y, z;
};

Acceleration influence(skepu::Index2D index, Particle pi, Particle pj)
{
  Acceleration acc;

  if (index.row != index.col)
  {
    float rij = sqrt((pi.x - pj.x) * (pi.x - pj.x)
                   + (pi.y - pj.y) * (pi.y - pj.y)
                   + (pi.z - pj.z) * (pi.z - pj.z));
    float dum = G * pi.m * pj.m / pow(rij, 3);

    acc.x = dum * (pi.x - pj.x);
    acc.y = dum * (pi.y - pj.y);
    acc.z = dum * (pi.z - pj.z);
  }
  else
  {
    acc.x = 0;
    acc.y = 0;
    acc.z = 0;
  }
  return acc;
}

Acceleration sum(Acceleration lhs, Acceleration rhs)
{
  Acceleration res = lhs;
  res.x += rhs.x;
  res.y += rhs.y;
  res.z += rhs.z;
  return res;
}

Particle update(Particle p, Acceleration a)
{
  Particle res = p;

  res.x += delta_t * p.vx + delta_t * delta_t / 2 * a.x;
  res.y += delta_t * p.vy + delta_t * delta_t / 2 * a.y;
  res.z += delta_t * p.vz + delta_t * delta_t / 2 * a.z;
  res.vx += delta_t * a.x;
  res.vy += delta_t * a.y;
  res.vz += delta_t * a.z;

  return res;
}

Particle init(skepu::Index1D index, size_t np)
{
  // Initialize positions and accelerations
}


auto nbody_init = skepu::Map<0>(init);
auto nbody_influence = skepu::MapPairsReduce<1, 1>(influence, sum);
auto nbody_update = skepu::Map<2>(update);

void nbody(skepu::Vector<Particle> &particles, size_t iterations)
{
  size_t np = particles.size();
  skepu::Vector<Acceleration> accel(np);

  nbody_init(particles, np);

  for (size_t i = 0; i < iterations; ++i)
  {
    nbody_influence(accel, particles, particles);
    nbody_update(particles, particles, accel);
  }
}
```

D.2 Game of life

Listing D.3: Conway's game of life implemented with `MapOverlap` in SkePU 3.

```cpp
#include <skepu>
#include <skepu-lib/io.hpp>

// Default population generator user-function
char initializer(skepu::Random<1> &prng)
{
  return (prng.get() % 100) < 10;
}

// Population update user-function
char updater(skepu::Region2D<char> r)
{
  char neighbors = 0;
  neighbors += r(-1, -1) + r(-1, 0) + r(-1, +1);
  neighbors += r( 0, -1)            + r( 0, +1);
  neighbors += r(+1, -1) + r(+1, 0) + r(+1, +1);

  bool activate = !r(0, 0) && (neighbors == 3);
  bool stay = r(0, 0) && ((neighbors == 2) || (neighbors == 3));
  return (activate || stay) ? 1 : 0;
}


int main(int argc, char *argv[])
{
  // Command line inputs
  if (argc < 5)
  {
    skepu::io::cout << "Usage: "
      << argv[0] << " height width iterations backend\n";
    exit(1);
  }
  const float height = atof(argv[1]);
  const float width = atof(argv[2]);
  const float iters = atof(argv[3]);
  auto spec = skepu::BackendSpec{argv[4]};
  skepu::setGlobalBackendSpec(spec);

  // Skeletons
  auto init = skepu::Map(initializer);
  auto update = skepu::MapOverlap(updater);
  update.setOverlap(1, 1);
  update.setEdgeMode(skepu::Edge::Cyclic);

  // Data containers
  skepu::Matrix<char> current(height, width), next(height, width);

  // Default population
  init(current);

  // Simulate evolution
  for (size_t i = 0; i < iters; ++i)
  {
    WritePngFileBinaryMatrix(current, i); // Note: not given here
    update(next, current);
    current.swap(next);
  }
}
```

D.3 Conjugate gradient

Listing D.4: Conjugate gradient computation implemented with SkePU-BLAS.

```
#include <skepu>
#include <skepu-lib/io.hpp>
#include <skepu-lib/blas.hpp>

template<typename T>
void conjugate_gradient(
  skepu::Matrix<T> BLAS_CONST& A,
  skepu::Vector<T> BLAS_CONST& b,
  skepu::Vector<T> &x)
{
  size_t N = b.size();
  assert(A.size_i() == N && A.size_j() == N && x.size() == N);
  skepu::Vector<T> p(N), r(N), Ap(N);

  // Set up initial r and p
  skepu::blas::copy(N, b, 1, r, 1);
  skepu::blas::gemv(skepu::blas::Op::NoTrans,
    N, N, -1.f, A, N, x, 1, 1.f, r, 1);
  skepu::blas::copy(N, r, 1, p, 1); // p := r

  float rTr = skepu::blas::dot(N, r, 1, r, 1); // rTr = r * r

  for (size_t k = 0; k < N; ++k)
  {
    // Compute alpha
    skepu::blas::gemv(skepu::blas::Op::NoTrans,
      N, N, 1.f, A, N, p, 1, 0.f, Ap, 1); // Ap := A * p
    float tmp = skepu::blas::dot(N, p, 1, Ap, 1); // tmp := p * Ap
    float alpha = rTr / tmp;

    // Update x
    skepu::blas::axpy(N,  alpha,  p, 1, x, 1); // x := x + alpha * p

    // Update r
    skepu::blas::axpy(N, -alpha, Ap, 1, r, 1); // r := r - alpha * Ap

    // Compute beta
    float rTr_new = skepu::blas::dot(N, r, 1, r, 1); // rTr_new := r*r
    float beta = rTr_new / rTr;

    // Early exit condition
    if (sqrt(rTr_new) < 1e-10f)
      return;

    // Update p
    skepu::blas::scal(N, beta, p, 1); // p := beta * p
    skepu::blas::axpy(N, 1.f, r, 1, p, 1); // p := r + p

    rTr = rTr_new;
  }
}
```

```
int main(int argc, char *argv[])
{
  if (argc < 3)
  {
    skepu::external([&]{ std::cout << "Usage: "
      << argv[0] << " size backend\n"; });
    exit(1);
  }

  const size_t n = atoi(argv[1]);
  auto spec = skepu::BackendSpec{argv[2]};
  skepu::setGlobalBackendSpec(spec);

  using T = float;
  const size_t N = n;

  skepu::Vector<T> b(N);
  skepu::Matrix<T> A(N, N);
  skepu::Vector<T> x(N, 0);

  conjugate_gradient(A, b, x);

  return 0;
}
```

D.4 CO_2 capture

Listing D.5: CO_2 capture application kernel in SkePU. [122]

```
#include <skepu>
...
Modules co2_model_skepu( // definition of user function
  skepu::Index1D index,
  const skepu::Vec<int> jvar,
  const skepu::Vec<Feed> jfeed,
  const skepu::Vec<Vars> vars)
{
  Modules m;
  Feed feed_ = jfeed(index.i);
  const Vars vars_ = vars(index.i);

// Declaration of variables v, e, pr, coef, hv
...

  int jvar_ = jvar(ABS_TOP);
  int *jfeed_ = &feed_.abs_top[0];
  // Core operations of CO2 capture model
  get_variables(jvar_, jfeed_, vars_.x, &v);
  get_enthalpy(&v, &e);
  lagrange(v.leng, v.presbot, &coef, &pr);
  get_helper_variables(&v, pr.prescp, &hv);
  ev_f(&v, &e, &pr, &coef, &hv, &m.f_abs_top[0]);

  ... // For other modules in the process
  return m;
}
...
extern "C" void x2p_skepu(
   const int n, const double x[],
   const int m, double f[])
{
  skepu::Vector<Modules> vector(MODEL_SIZE);
  // instantiate Map skeleton to build a "function"
  auto calculate = skepu::Map<0>(co2_model_skepu);

  // call the skeleton instance
  calculate(vector, jvar_skepu_vec(),
    jfeed_skepu_vec(), vars_skepu_vec(n, x));

  vector.flush();
  copy_skepu_vector_to_fortran(vector, f);
}
```

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology
Linköping Studies in Arts and Sciences

Linköping Studies in Statistics
Linköping Studies in Information Science

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall**: Semantisk analys av processbeskrivningar i naturligt språk, 1977, ISBN 91- 7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System, 1978, ISBN 91- 7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91- 7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372- 805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani:** Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN 91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91-7373-207-9.

No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91-7373-208-7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91-7373-212-5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91-7373-258-3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-supported Interorganisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X.

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationssystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing – An Empirical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5.

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004, ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modelling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 1503 **John Wilander**: Contributions to Specification, Implementation, and Execution of Secure Software, 2013, ISBN 978-91-7519-681-7.

No 1506 **Magnus Ingmarsson**: Creating and Enabling the Useful Service Discovery Experience, 2013, ISBN 978-91-7519-662-6.

No 1547 **Wladimir Schamai**: Model-Based Verification of Dynamic System Behavior against Requirements: Method, Language, and Tool, 2013, ISBN 978-91-7519-505-6.

No 1551 **Henrik Svensson**: Simulations, 2013, ISBN 978-91-7519-491-2.

No 1559 **Sergiu Rafiliu**: Stability of Adaptive Distributed Real-Time Systems with Dynamic Resource Management, 2013, ISBN 978-91-7519-471-4.

No 1581 **Usman Dastgeer**: Performance-aware Component Composition for GPU-based Systems, 2014, ISBN 978-91-7519-383-0.

No 1602 **Cai Li**: Reinforcement Learning of Locomotion based on Central Pattern Generators, 2014, ISBN 978-91-7519-313-7.

No 1652 **Roland Samlaus**: An Integrated Development Environment with Enhanced Domain-Specific Interactive Model Validation, 2015, ISBN 978-91-7519-090-7.

No 1663 **Hannes Uppman**: On Some Combinatorial Optimization Problems: Algorithms and Complexity, 2015, ISBN 978-91-7519-072-3.

No 1664 **Martin Sjölund**: Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models, 2015, ISBN 978-91-7519-071-6.

No 1666 **Kristian Stavåker**: Contributions to Simulation of Modelica Models on Data-Parallel Multi-Core Architectures, 2015, ISBN 978-91-7519-068-6.

No 1680 **Adrian Lifa**: Hardware/Software Codesign of Embedded Systems with Reconfigurable and Heterogeneous Platforms, 2015, ISBN 978-91-7519-040-2.

No 1685 **Bogdan Tanasa**: Timing Analysis of Distributed Embedded Systems with Stochastic Workload and Reliability Constraints, 2015, ISBN 978-91-7519-022-8.

No 1691 **Håkan Warnquist**: Troubleshooting Trucks – Automated Planning and Diagnosis, 2015, ISBN 978-91-7685-993-3.

No 1702 **Nima Aghaee**: Thermal Issues in Testing of Advanced Systems on Chip, 2015, ISBN 978-91-7685-949-0.

No 1715 **Maria Vasilevskaya**: Security in Embedded Systems: A Model-Based Approach with Risk Metrics, 2015, ISBN 978-91-7685-917-9.

No 1729 **Ke Jiang**: Security-Driven Design of Real-Time Embedded System, 2016, ISBN 978-91-7685-884-4.

No 1733 **Victor Lagerkvist**: Strong Partial Clones and the Complexity of Constraint Satisfaction Problems: Limitations and Applications, 2016, ISBN 978-91-7685-856-1.

No 1734 **Chandan Roy**: An Informed System Development Approach to Tropical Cyclone Track and Intensity Forecasting, 2016, ISBN 978-91-7685-854-7.

No 1746 **Amir Aminifar**: Analysis, Design, and Optimization of Embedded Control Systems, 2016, ISBN 978-91-7685-826-4.

No 1747 **Ekhiotz Vergara**: Energy Modelling and Fairness for Efficient Mobile Communication, 2016, ISBN 978-91-7685-822-6.

No 1748 **Dag Sonntag**: Chain Graphs – Interpretations, Expressiveness and Learning Algorithms, 2016, ISBN 978-91-7685-818-9.

No 1768 **Anna Vapen**: Web Authentication using Third-Parties in Untrusted Environments, 2016, ISBN 978-91-7685-753-3.

No 1778 **Magnus Jandinger**: On a Need to Know Basis: A Conceptual and Methodological Framework for Modelling and Analysis of Information Demand in an Enterprise Context, 2016, ISBN 978-91-7685-713-7.

No 1798 **Rahul Hiran**: Collaborative Network Security: Targeting Wide-area Routing and Edge-network Attacks, 2016, ISBN 978-91-7685-662-8.

No 1813 **Nicolas Melot**: Algorithms and Framework for Energy Efficient Parallel Stream Computing on Many-Core Architectures, 2016, ISBN 978-91-7685-623-9.

No 1823 **Amy Rankin**: Making Sense of Adaptations: Resilience in High-Risk Work, 2017, ISBN 978-91-7685-596-6.

No 1831 **Lisa Malmberg**: Building Design Capability in the Public Sector: Expanding the Horizons of Development, 2017, ISBN 978-91-7685-585-0.

No 1851 **Marcus Bendtsen**: Gated Bayesian Networks, 2017, ISBN 978-91-7685-525-6.

No 1852 **Zlatan Dragisic**: Completion of Ontologies and Ontology Networks, 2017, ISBN 978-91-7685-522-5.

No 1854 **Meysam Aghighi**: Computational Complexity of some Optimization Problems in Planning, 2017, ISBN 978-91-7685-519-5.

No 1863 **Simon Ståhlberg**: Methods for Detecting Unsolvable Planning Instances using Variable Projection, 2017, ISBN 978-91-7685-498-3.

No 1879 **Karl Hammar**: Content Ontology Design Patterns: Qualities, Methods, and Tools, 2017, ISBN 978-91-7685-454-9.

No 1887 **Ivan Ukhov**: System-Level Analysis and Design under Uncertainty, 2017, ISBN 978-91-7685-426-6.

No 1891 **Valentina Ivanova**: Fostering User Involvement in Ontology Alignment and Alignment Evaluation, 2017, ISBN 978-91-7685-403-7.

No 1902 **Vengatanathan Krishnamoorthi**: Efficient HTTP-based Adaptive Streaming of Linear and Interactive Videos, 2018, ISBN 978-91-7685-371-9.

No 1903 **Lu Li**: Programming Abstractions and Optimization Techniques for GPU-based Heterogeneous Systems, 2018, ISBN 978-91-7685-370-2.

No 1913 **Jonas Rybing**: Studying Simulations with Distributed Cognition, 2018, ISBN 978-91-7685-348-1.

No 1936 **Leif Jonsson**: Machine Learning-Based Bug Handling in Large-Scale Software Development, 2018, ISBN 978-91-7685-306-1.

No 1964 **Arian Maghazeh**: System-Level Design of GPU-Based Embedded Systems, 2018, ISBN 978-91-7685-175-3.

No 1967 **Mahder Gebremedhin**: Automatic and Explicit Parallelization Approaches for Equation Based Mathematical Modeling and Simulation, 2019, ISBN 978-91-7685-163-0.

No 1984 **Anders Andersson**: Distributed Moving Base Driving Simulators – Technology, Performance, and Requirements, 2019, ISBN 978-91-7685-090-9.

No 1993 **Ulf Kargén**: Scalable Dynamic Analysis of Binary Code, 2019, ISBN 978-91-7685-049-7.

No 2001 **Tim Overkamp**: How Service Ideas Are Implemented: Ways of Framing and Addressing Service Transformation, 2019, ISBN 978-91-7685-025-1.

No 2006 **Daniel de Leng**: Robust Stream Reasoning Under Uncertainty, 2019, ISBN 978-91-7685-013-8.

No 2048 **Biman Roy**: Applications of Partial Polymorphisms in (Fine-Grained) Complexity of Constraint Satisfaction Problems, 2020, ISBN 978-91-7929-898-2.

No 2051 **Olov Andersson**: Learning to Make Safe Real-Time Decisions Under Uncertainty for Autonomous Robots, 2020, ISBN 978-91-7929-889-0.

No 2065 **Vanessa Rodrigues**: Designing for Resilience: Navigating Change in Service Systems, 2020, ISBN 978-91-7929-867-8.

No 2082 **Robin Kurtz**: Contributions to Semantic Dependency Parsing: Search, Learning, and Application, 2020, ISBN 978-91-7929-822-7.

No 2108 **Shanai Ardi**: Vulnerability and Risk Analysis Methods and Application in Large Scale Development of Secure Systems, 2021, ISBN 978-91-7929-744-2.

No 2125 **Zeinab Ganjei**: Parameterized Verification of Synchronized Concurrent Programs, 2021, ISBN 978-91-7929-697-1.

No 2153 **Robin Keskisärkkä**: Complex Event Processing under Uncertainty in RDF Stream Processing, 2021, ISBN 978-91-7929-621-6.

No 2168 **Rouhollah Mahfouzi**: Security-Aware Design of Cyber-Physical Systems for Control Applications, 2021, ISBN 978-91-7929-021-4.

No 2205 **August Ernstsson**: Pattern-based Programming Abstractions for Heterogeneous Parallel Computing, 2022, ISBN 978-91-7929-195-2.

Linköping Studies in Arts and Sciences

No 504 **Ing-Marie Jonsson:** Social and Emotional Characteristics of Speech-based In-Vehicle Information Systems: Impact on Attitude and Driving Behaviour, 2009, ISBN 978-91-7393-478-7.

No 586 **Fabian Segelström:** Stakeholder Engagement for Service Design: How service designers identify and communicate insights, 2013, ISBN 978-91-7519-554-4.

No 618 **Johan Blomkvist:** Representing Future Situations of Service: Prototyping in Service Design, 2014, ISBN 978-91-7519-343-4.

No 620 **Marcus Mast:** Human-Robot Interaction for Semi-Autonomous Assistive Robots, 2014, ISBN 978-91-7519-319-9.

No 677 **Peter Berggren:** Assessing Shared Strategic Understanding, 2016, ISBN 978-91-7685-786-1.

No 695 **Mattias Forsblad:** Distributed cognition in home environments: The prospective memory and cognitive practices of older adults, 2016, ISBN 978-91-7685-686-4.

No 787 **Sara Nygårdhs:** Adaptive behaviour in traffic: An individual road user perspective, 2020, ISBN 978-91-7929-857-9.

No 811 **Sam Thellman:** Social Robots as Intentional Agents, 2021, ISBN 978-91-7929-008-5.

Linköping Studies in Statistics

No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.

No 10 **Karl Wahlin:** Roadmap for Trend Detection and Assessment of Data Quality, 2008, ISBN 978-91-7393-792-4.

No 11 **Oleg Sysoev:** Monotonic regression for large multivariate datasets, 2010, ISBN 978-91-7393-412-1.

No 13 **Agné Burauskaite-Harju:** Characterizing Temporal Change and Inter-Site Correlations in Daily and Sub-daily Precipitation Extremes, 2011, ISBN 978-91-7393-110-6.

No 14 **Måns Magnusson:** Scalable and Efficient Probabilistic Topic Model Inference for Textual Data, 2018, ISBN 978-91-7685-288-0.

No 15 **Per Sidén:** Scalable Bayesian spatial analysis with Gaussian Markov random fields, 2020, 978-91-7929-818-0.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN 9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998, ISBN 9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN 91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000, ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X.

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN 91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden – Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.

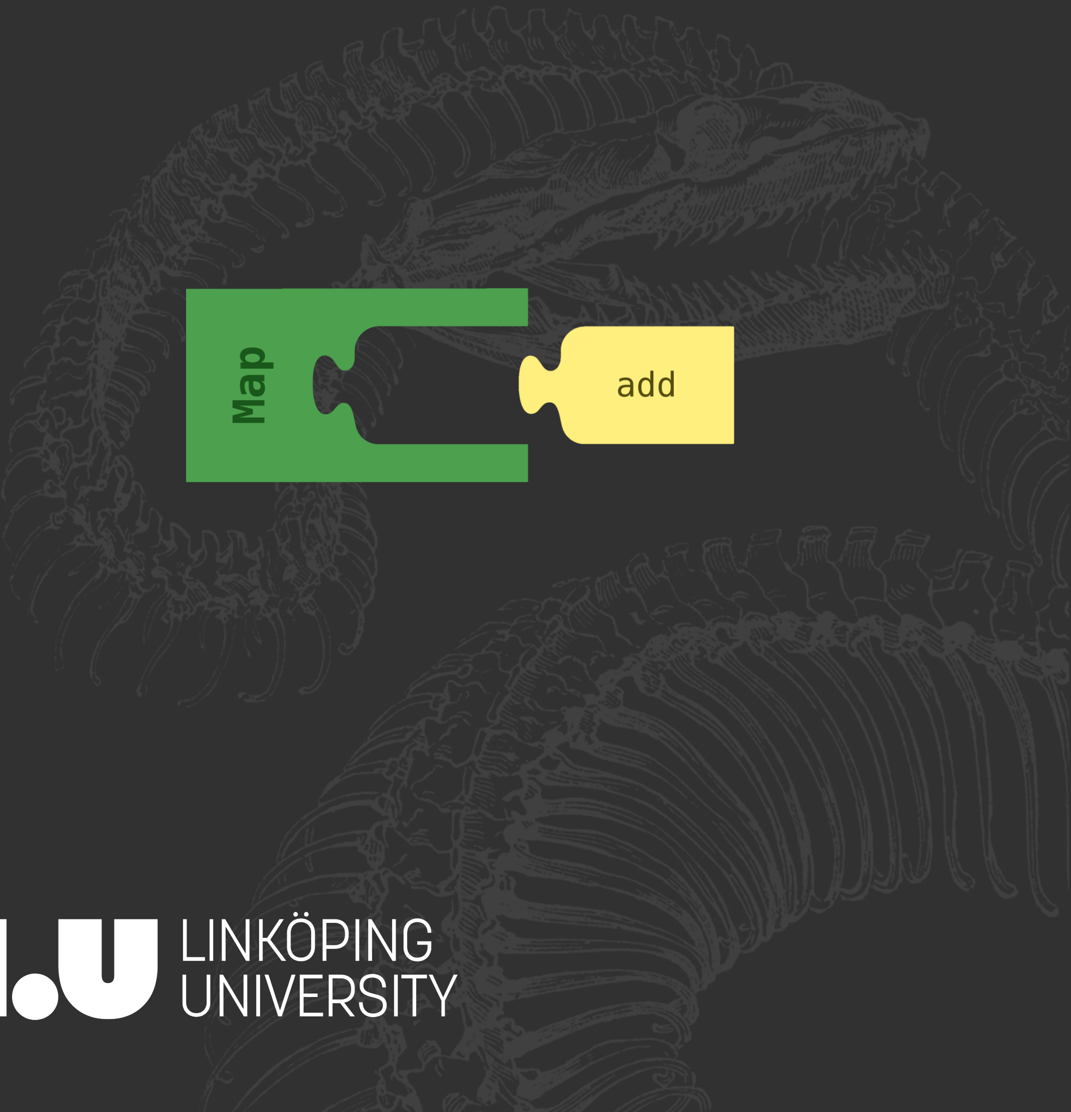
No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponentbaserade informationssystem, 2006, ISBN 91-85643-22-X.

FACULTY OF SCIENCE AND ENGINEERING

Linköping Studies in Science and Technology, Dissertation No. 2205, 2022
Department of Computer and Information Science

Linköping University
SE-581 83 Linköping, Sweden

www.liu.se



LINKÖPING
UNIVERSITY