Linköping Studies in Science and Technology.
Licentiate Thesis No. 1886

Designing a Modern Skeleton Programming Framework for Parallel and Heterogeneous Systems

August Ernstsson

LINKÖPING UNIVERSITY

Linköping Studies in Science and Technology
Licentiate Thesis No. 1886

Designing a Modern Skeleton Programming Framework for Parallel and Heterogeneous Systems

August Ernstsson

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

Linköping 2020

This is a Swedish Licentiate's Thesis

Swedish postgraduate education leads to a doctor's degree and/or a licentiate's degree.
A doctor's degree comprises 240 ECTS credits (4 years of full-time studies).
A licentiate's degree comprises 120 ECTS credits.

Edition 1:1

ISBN 978-91-7929-772-5
ISSN 0280-7971
URL http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-170194

Typeset using XeTeX

Printed by LiU-Tryck, Linköping 2020

ABSTRACT

Today's society is increasingly software-driven and dependent on powerful computer technology. Therefore it is important that advancements in the low-level processor hardware are made available for exploitation by a growing number of programmers of differing skill level. However, as we are approaching the end of Moore's law, hardware designers are finding new and increasingly complex ways to increase the accessible processor performance. It is getting more and more difficult to effectively target these processing resources without expert knowledge in parallelization, heterogeneous computation, communication, synchronization, and so on. To ensure that the software side can keep up, advanced programming environments and frameworks are needed to bridge the widening gap between hardware and software. One such example is the pattern-centric skeleton programming model and in particular the SkePU project. The work presented in this thesis first redesigns the SkePU framework based on modern C++ variadic template metaprogramming and state-of-the-art compiler technology. It then explores new ways to improve performance: by providing new patterns, improving the data access locality of existing ones, and using both static and dynamic knowledge about program flow. The work combines novel ideas with practical evaluation of the approach on several applications. The advancements also include the first skeleton API that allows variadic skeletons, new data containers, and finally an approach to make skeleton programming more customizable without compromising universal portability.

Acknowledgments

First and foremost I want to thank my supervisor at Linköping University, 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.

Secondly, an important acknowledgement goes to everyone who has made direct and lasting contributions to the SkePU framework since its inception in 2010. Names are listed in chronological order of earliest contribution: *Johan Enmyren*, *Usman Dastgeer*, *Lu Li*, *Oskar Sjöström*, *Henrik Henriksson*, and *Johan Ahlqvist*.

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.

Thirdly, 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 PELAB 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, October 2020

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. 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 the-

sis 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*?

Specifically, the thesis goes into detail on three 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 2 (and 3) interface? What is the optimal split ratio of work between CPU and GPU backends, and what is the possible performance gain?

RQ6 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?

1.2 Published work behind this thesis

This thesis is based on the work presented in six papers, five published and one in the process of publication.

1. **SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems** [30]
   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. Initial prototype and design work of SkePU 2 was carried out as part of August Ernstsson's master's thesis [26]. 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** [28]
   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** [55]
   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. The contributions in this paper are results of the master's thesis project by Tomas Öhberg [54], supervised and guided by August.

4. **Multi-Variant User Functions for Platform-Aware Skeleton Programming** [29]
   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. 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** [57]
   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 *EXA2PRO* project, 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** [27]
   August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler
   This paper was presented at **HLPP 2020** on July 9, 2020. At the time of writing, an updated version is pending submission for a special issue journal publication. Also a direct result of *EXA2PRO* collaborations, the paper introduces SkePU 3, including its new cluster backend.

In addition to the peer-reviewed published material, this thesis is shaped by the experience gained from the exposure of SkePU to potential users by the means of numerous tutorials, given e.g. in conjunction with **HLPP 2019** and **MCC 2019**, and in teaching through the course *TDDD56: Multicore and GPU programming*, and through supervision of master's thesis projects.

Paper 2, 3, and 4 are reproduced in this thesis in large part as individual contributions. Paper 1 and 6 have the introduction of SkePU version 2 and SkePU version 3, respectively, as their main contributions, and therefore the material in these papers is significantly reworked into the chapters of this thesis which present the history, interface, and implementation of SkePU, together with a considerable amount of newly written material. In addition, experimental results and evaluation from all papers is collected and reproduced in the results chapter.

1.3 Structure

This thesis is structured as follows:

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 application programming **interface** of SkePU and the decisions behind it. This chapter also contains a study of SkePU's data representation abstraction, **smart containers**. Once the outwards-facing aspects of SkePU are well understood, Chapter 5 explains the **implementation** of SkePU with its header library and compiler toolchain.

The subsequent three chapters presents three *main contributions* and are based on three of the papers mentioned in Section 1.2: Chapter 6 covers the work on a data-locality optimization, Chapter 7 presents the hybrid backend, and Chapter 8 details multi-variant user functions. These chapters omits the results, which are collected in Chapter 9 together with other published and unpublished **results** including performance evaluation.

Chapter 10 **concludes** the thesis and presents ideas for **future work**.

2 Skeleton programming

This chapter presents skeleton programming—the approach to high-level parallel programming which forms the basis for the work presented in this thesis. It starts with a background and surrounding context of pattern-based parallel programming, and subsequently moves on to related work, surveying the vast field of systems implementing skeleton programming and related ideas.

2.1 Background

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 systems. In this chapter, we assume that the hardware side of the equation has been taken care of. An assumption that largely is true—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 has millions of nodes[1] (and total parallel performance of almost half an exaflop).

A natural follow-up is *how* the underlying issues presented in the motivation should be addressed. 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

[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/top500/2020/06/`.

systems need to accommodate this fact in the systems and interfaces presented to the programmer. As expressed by Cole [12], 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 improve on this situation by reducing the user-facing complexity of programs. 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* [12] concept.

Skeleton programming [12, 33] is a programming model for parallel systems inspired by functional 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 compose entire programs, 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 [61] 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.

Several parallel programming frameworks implement the algorithmic skeleton model [24, 70, 25, 46], some of them for multiple different parallel architectures (*backends*) with a single common interface. Selection of backends can be done with auto-tuning [15]. 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 par-

allel 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.

2.2 Related work

The skeleton approach to high-level programming of parallel systems was introduced by Cole in 1989 [12, 11]. 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 two pattern-based frameworks in more detail: *GrPPI* in Section 2.2.1 and *Musket* in Section 2.2.2. Both 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 2.2.3, while Section 2.2.4 explores further 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.

2.2.1 GrPPI

GrPPI [63] 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.

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.

2.2.2 Musket

Musket [62] 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 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.

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

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);
}
```

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.

2.2.3 SYCL

Over the past decade, there has been standardization efforts of skeleton-like interfaces. One such instance is *SYCL* [60] from the Khronos Group[2]. 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 both as a programmer-facing interface and a backend target for domain-specific lan-

[2] `https://www.khronos.org/sycl`

Listing 2.3: 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]; }
    );
  });
}
```

guages 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 2.3 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* [73], 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.

2.2.4 C++ AMP, and other industry efforts

Intel TBB (thread building blocks)[3] 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* [6] 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 in what they call *oneAPI*[4]. 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[5] (`stdpar`). The Nvidia *HPC SDK C++ compiler*, NVC++, targets GPU parallelism using only C++ standard library constructs, as seen in the example in Listing 2.5.

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

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

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

Listing 2.4: 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];
        }
    );
}
```

Listing 2.5: 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();
```

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

2.2.5 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

Listing 2.6: 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;
}
```

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.

FastFlow

FastFlow [14] 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 REPARA interface FastFlow 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 [71] is a high-level functional IR, intermediate representation, 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 specific 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 [72] 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. [39]

SkelCL

SkelCL [70] is an actively developed OpenCL-based skeleton 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 [69], 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*) [10, 25] 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.

Muesli has support for hybrid execution using a static approach [76], 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 [46, 67] 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 [53]. 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.

PACXX

PACXX is a unified programming model for systems with GPU accelerators [35], utilizing the C++14 language. PACXX was an inspiration in the initial design and prototyping work for SkePU 2 [26], 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* [47] 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 [58] 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 [44] is a programming model for heterogeneous architectures, based on TBB and CUDA. Qilin provides the user with a number of pre-defined

Listing 2.7: Convolution kernel in PSkel, adapted from [58].

```
__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;
}
```

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* [36], 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. [52] 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.

2.3 Independent surveys

De Sensi et al. [20] 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 [7] 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* [1].

Arvanitou et al. [3] conducted a technical debt 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.

2.4 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 [24, 18, 17] for other work relating to specific features, such as smart containers.

3 SkePU overview

SkePU [24, 30, 27] 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-movement coefficient.

For data abstraction, SkePU provides *smart containers* which manage coherency states automatically. Smart 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:

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;
}
```

- **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.

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

SkePU provides *smart containers* [17], 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 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 containers are well suited for iterative computations, where the performance gains can be significant. Smart containers are further detailed in Section 4.13.

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

- **Sequential CPU backend**, mainly used as a reference implementation and baseline.
- **OpenMP backend**, for multi-core CPUs.
- **CUDA backend**, for NVIDIA GPUs, either single and multiple.
- **OpenCL backend**, for single and multiple GPUs of any OpenCL supported model, including other accelerators such as Intel Xeon Phis.
- **Cluster backend**, backed by the StarPU runtime system and MPI for execution on large-scale clusters, including supercomputers. See Section 5.3.4.

- **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. See Chapter 7.

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 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 4.13).

3.1 History

SkePU (version 1) was introduced in 2010 [24] 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 [30], was a major revision of the SkePU [24] 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 [66] that skeleton patterns are not always perfectly suited to algorithms in real-world application code.

SkePU 2 also introduced the *pre-compiler*, 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

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

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.

3.2 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 meta-programming 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 [49], and it had been successfully used in, for example, the REPARA project [13]. 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 9.1. 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 [66] 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 [19] 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 meta-programming 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.

- **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 meta-programming 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 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 5.

[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 [29] (Chapter 8) provided further contribution in this direction.

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

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);
```

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;
}

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

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

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.

3.3 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 feature 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 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 for MapOverlap.

- **Smart 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.

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) [24]	**SkePU 2** (2016) [30]	**SkePU 3** (2020) [27]
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

4 SkePU programming interface design

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 special-

ized 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.

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 order.							
**	Dimensions higher than 2 are linearized in the current implementation.							
***	A region of elements surrounding the current index is supplied.							

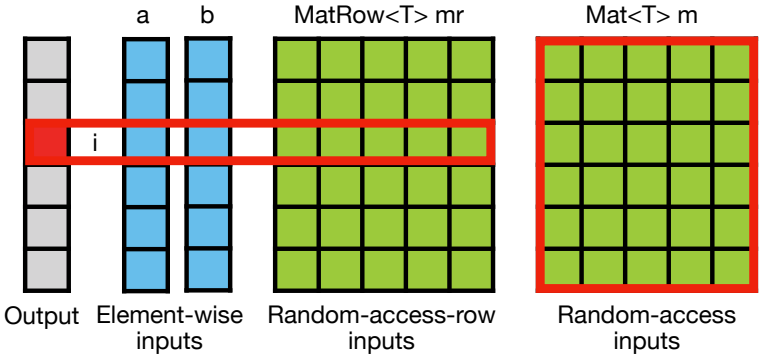


Figure 4.1: Illustrative diagram of the operand access scopes in the Map skeleton.

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 in SkePU are directly based upon the foundations of `Map`. Indeed, the names tell the story: `MapReduce`, `MapPairs`, `MapPairsReduce`, and `MapOverlap` are all

Listing 4.1: Example usage of the Map skeleton.

```
1  float add(float a, float b)
   {
     return a + b;
   }
5
   Vector<float> vector_sum(skepu::Vector<float> &v1, skepu::Vector<float> &v2)
   {
     auto vsum = skepu::Map(add);
     skepu::Vector<float> result(v1.size());
10   return vsum(result, v1, v2);
   }
```

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 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 containers are C++ objects of intricate class templates, and cannot be made available in a backend execution context. Therefore, smart containers as bound variables in user functions are encoded as *proxy containers*, further covered in Section 4.14. Listing 4.2 illustrates the use of auxiliary smart containers in the matrix-vector multiplication skeleton instance `mvmult`[3] and Figure 4.17 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 [30] and master thesis [26], and prompted the complete API redesign from SkePU 1.[4] The underlying C++-11 features which

[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 4.19.

[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.

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);
```

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 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

[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.

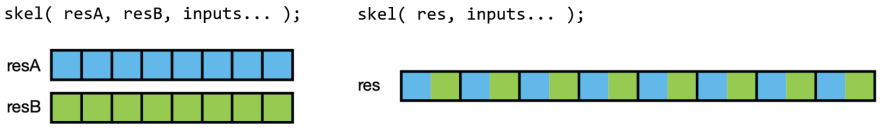


Figure 4.2: Difference in return value storage between using multi-valued return (left) and single-value (by manually managed array-of-struct) return (right).

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 4.14) 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 4.17 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, oper-

[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 5) such that programs are semantically sound C++ programs also without the pre-compiler.

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);
skepu::Vector<float> e(1);

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

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

ating 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 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 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.

Listing 4.5: Index types corresponding to each smart container.

```
1 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; };
```

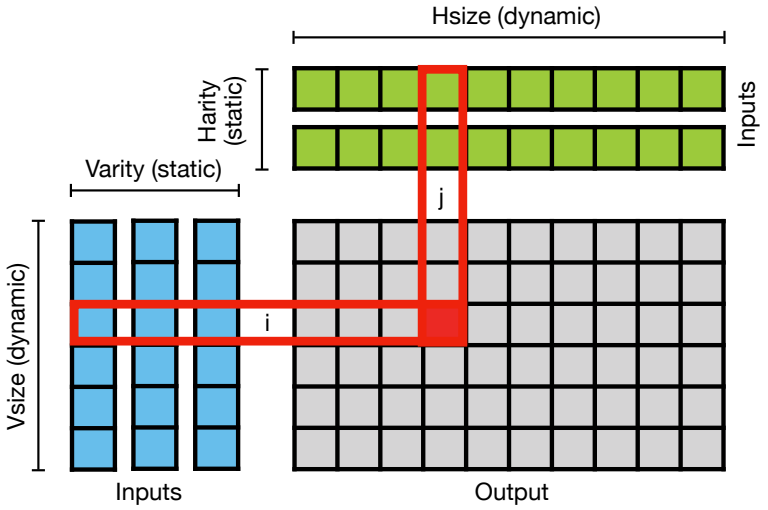


Figure 4.3: Illustrative diagram of the MapPairs skeleton.

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 directions 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.

Listing 4.6: Example usage of the MapPairs skeleton.

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

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

     skepu::Vector<int> v1(Vsize, 3), h1(Hsize, 7);
     skepu::Matrix<int> res(Vsize, Hsize);
     pairs(res, v1, h1);
10 }
```

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 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.

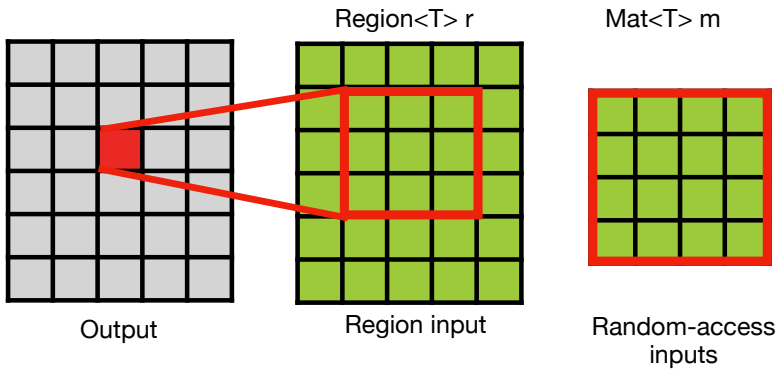


Figure 4.4: Illustrative diagram of the MapOverlap skeleton.

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 types:

- **1D MapOverlap** on
  - vector containers or
  - matrix containers with
    - row-wise overlap,
    - column-wise overlap,
    - row-wise overlap followed by column-wise overlap (two passes), or
    - column-wise overlap followed by row-wise overlap (two passes).
- **2D MapOverlap**
  On matrix containers.
- **3D MapOverlap**
  On three-dimensional tensor containers.
- **4D MapOverlap**
  On four-dimensional tensor containers.

Listing 4.7: Example usage of the MapOverlap skeleton.

```
1  float conv(skepu::Region2D<float> r, const skepu::Mat<float> stencil)
   {
     float res = 0;
     for (int i = -r.oi; i <= r.oi; ++i)
5      for (int j = -r.oj; j <= r.oj; ++j)
         res += r(i, j) * stencil(i + r.oi, j + r.oj);
     return res;
   }

10 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());
15   convol.setOverlap(2);
     return convol(result, v, stencil, 10);
   }
```

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 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 4.14.3.

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 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

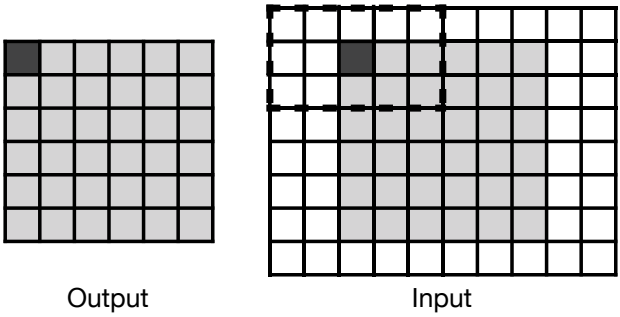


Figure 4.5: Expected input and output container sizes when edge element synthesis disabled, here in 2D `MapOverlap`.

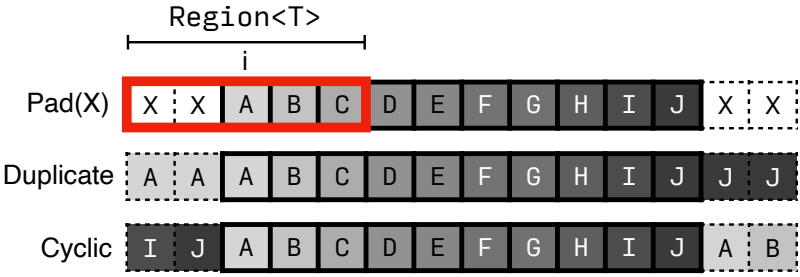


Figure 4.6: Edge handling modes of 1D `MapOverlap`.

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 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.

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

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.

4.5 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 reduction 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.

Figure 4.7: 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);
}
```

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 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.

[9]The linear interpretation follows the memory layout order presented in Section 4.13, but the commutativity constraint implies that the reduction semantics allows any element order.

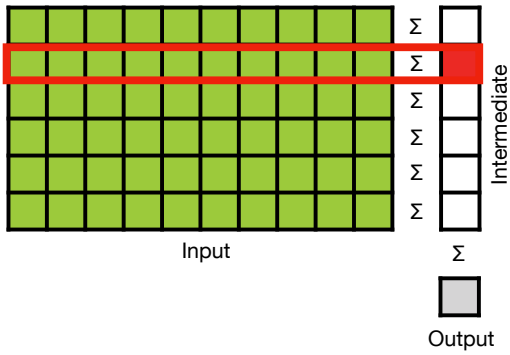


Figure 4.8: Illustrative diagram of the access pattern in two-dimensional `Reduce`.

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

```
1  float plus_f(float a, float b)
   {
     return a + b;
   }
5
   float max_f(float a, float b)
   {
     return (a > b) ? a : b;
   }
10
   float min_element(skepu::Matrix<float> &v)
   {
     auto max_sum = skepu::Reduce(plus_f, max_f);
     sum.setReduceMode(skepu::ReduceMode::RowWise);
15   return max_sum(v);
   }
```

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.

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. [40] 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

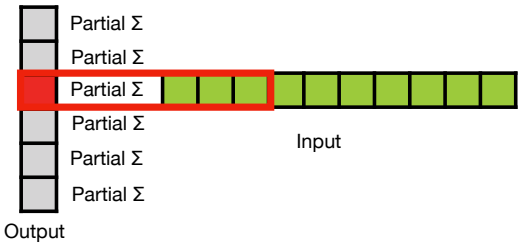


Figure 4.9: Illustrative diagram of the Scan skeleton.

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);
}
```

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 semantics 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 containers (see Section 4.13).

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 straightfor-

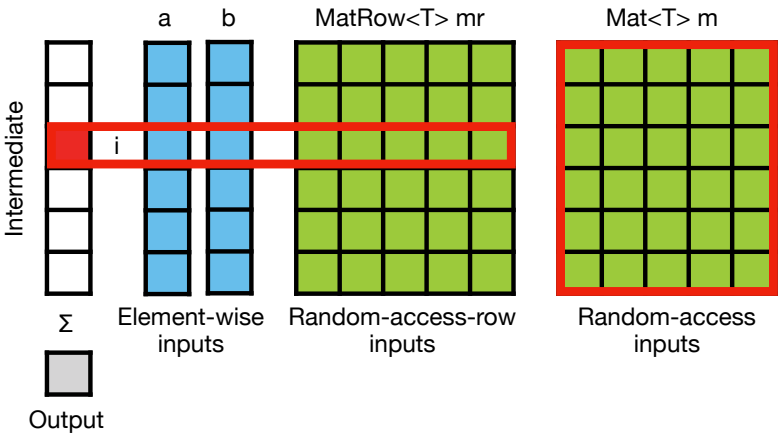


Figure 4.10: Illustrative diagram of the MapReduce skeleton.

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);
}
```

ward way with two `Reduce` instances, but for the sake of discussion 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 computa-

[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.

```
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";
}
```

tion 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.

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

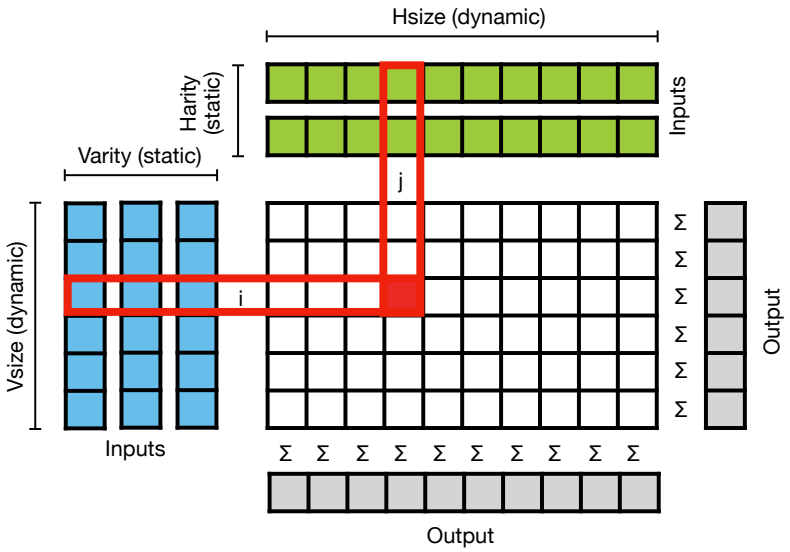


Figure 4.11: Illustrative diagram of the MapPairsReduce skeleton.

Listing 4.13: Example usage of the MapPairsReduce skeleton.

```
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);
}
```

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 the computing resources e.g. of external accelerators while the smart containers handle memory transfer and coherency management automatically. The clear downside is that a user function is a sequential block of

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());
}
```

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 8 on multi-variant user functions.

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);
```

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 application 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 4.15, in particular Figure 4.17).

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

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

Smart 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 4.12, 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 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

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(skepu2::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 = skepu2::Map<0>(mandelbrot_f);
```

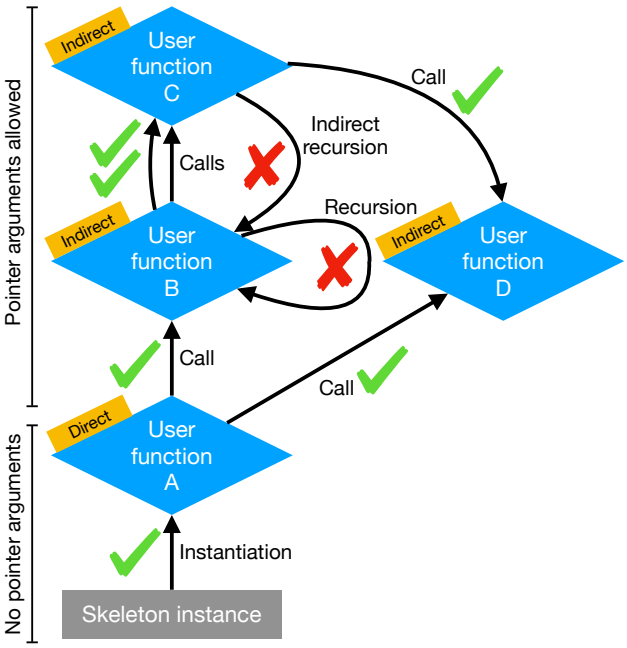


Figure 4.12: User function call graph.

possibility of using a custom `struct` as the element type in smart 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 on objects of type `cplx` are defined as free functions and are treated as user functions by the pre-compiler.

4.12 User constants

The example in Listing 4.17 also uses the related feature *user constants*, e.g., `MAX_ITERS`, which are compile-time constant values that can be read in user functions. These objects are annotated with the `[[skepu2::userconstant]]` attribute.

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 5, but fits with the general aim of SkePU to move away from macros and instead rely on type-safety through the C++ type system.

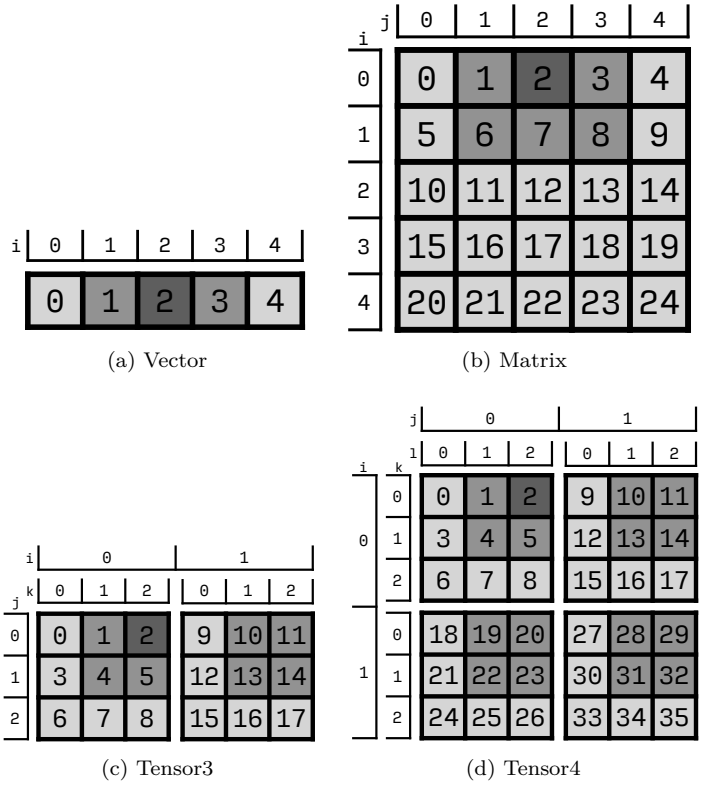
Scalar arguments to user functions (discussed in Sections 4.2.1 and 4.2.2) can be employed as a substitute for macros, but their purpose is typically aimed at scenarios where the value of the argument changes dynamically across skeleton invocations. Therefore, SkePU provides *user constants* to more directly address the need for global, static parameters in user function code. These objects are regular C++ objects of basic type and therefore type-safe. Usage of user constants can also be seen in the N-body simulation in Listing B.2.

4.13 Smart containers

The availability of smart 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 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 containers additionally perform transparent software caching of the operand elements that they wrap, with a MSI-based coherence protocol [17]. Hence, smart 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 allocation. Smart containers can lead to a significant performance gain over "non-smart" 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 [27] extended with *tensors*, which are higher-dimensionality containers, completing the picture in Figure 4.13. 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 meta-programming 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 of SkePU, and as such the container set is restrained to cover up to four dimensions.



(a) Vector (b) Matrix

(c) Tensor3 (d) Tensor4

Figure 4.13: Container indexing and memory layout.

Listing 4.18: Smart container set in SkePU 3.

```
1 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);
```

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 containers in SkePU 3 now covers up to four-dimensional structures; see Listing 4.18 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.

4.13.1 Container indexing

Even though SkePU aims for a high-level programming interface and its 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 4.13 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 4.14.3 for more details.

Figure 4.13 illustrates indexing and memory layout of the four smart container types. Figure 4.13a shows a `Vector` of size 5, Figure 4.13b shows a `Matrix` of size 5×5, Figure 4.13c shows a `Tensor3` of size $2 \times 3 \times 3$, and Figure 4.13d shows a `Tensor4` of size $2 \times 2 \times 3 \times 3$.

4.14 Container proxies

Smart 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

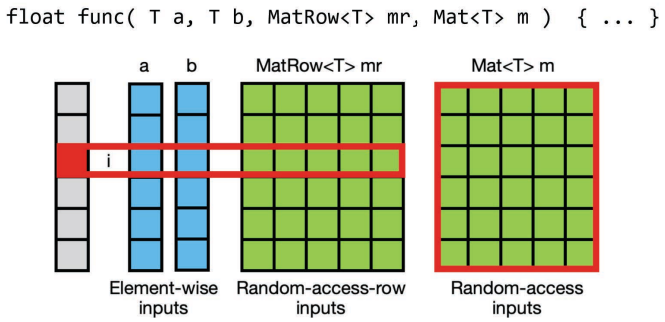


Figure 4.14: Element accessibility for `MatRow` vs. `Mat` parameters in a user function.

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 4.13.1) 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 containers. Each such partial proxy object is covered further in the upcoming sections.

4.14.1 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.

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

Listing 4.19: 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);
```

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 4.14.

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.

4.14.2 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.
3. Otherwise, the skeleton instance is malformed.

Listing 4.20: Matrix-matrix multiply with MatRow and MatCol.

```
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 4.20 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.

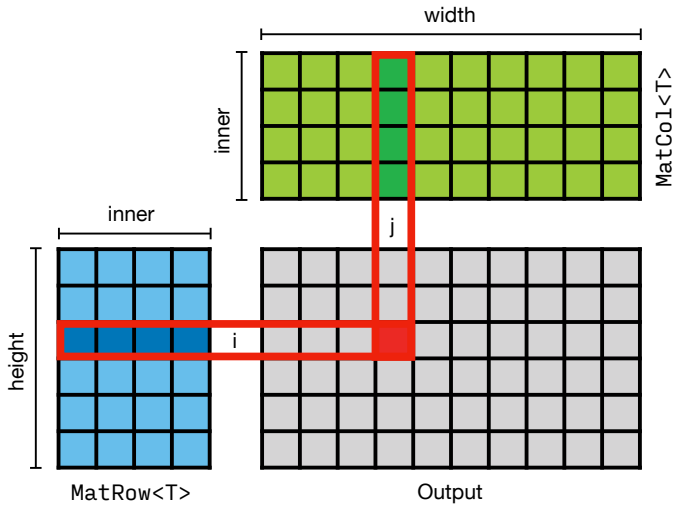


Figure 4.15: `MatRow` and `MatCol` access patterns in the computation in Listing 4.20.

4.14.3 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 4.13, 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 4.16.

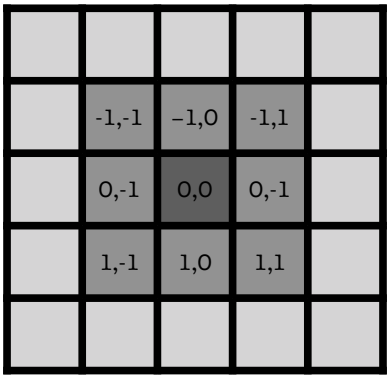


Figure 4.16: Indexing into a `Region2D` matrix proxy with overlap size $(1, 1)$.

4.15 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 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.[11]

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 maintained 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

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

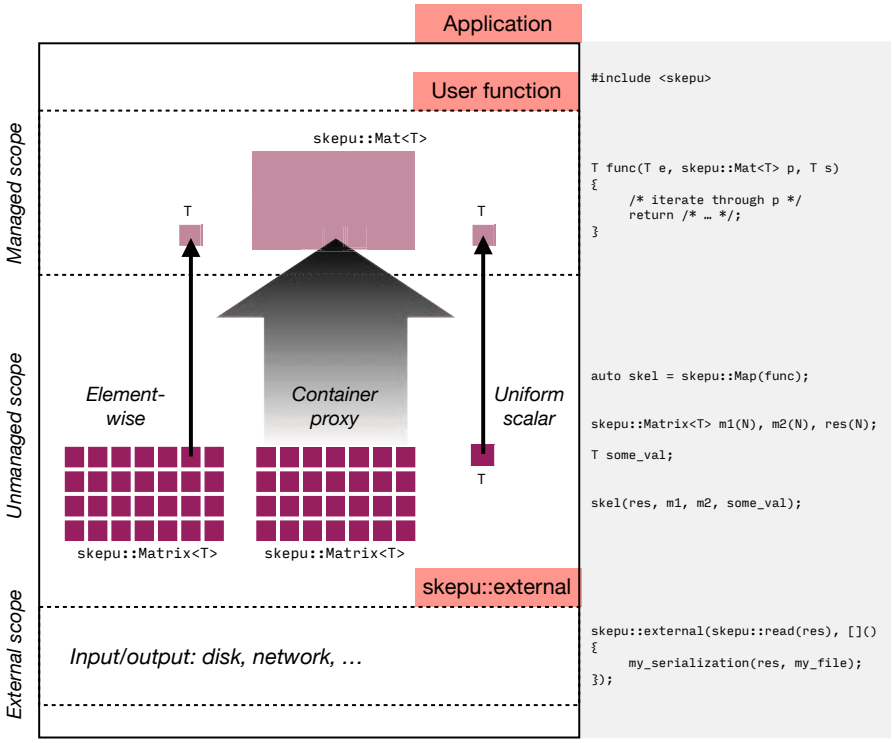


Figure 4.17: Scopes with differing capabilties in a SkePU program.

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 containers is covered in more detail in Section 4.13.1.)

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 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

Listing 4.21: 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);
```

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 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 4.21.

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.

4.15.1 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, 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 Section 5.3.4). 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.

5 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.

5.1 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 5.1.

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

[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. [26]

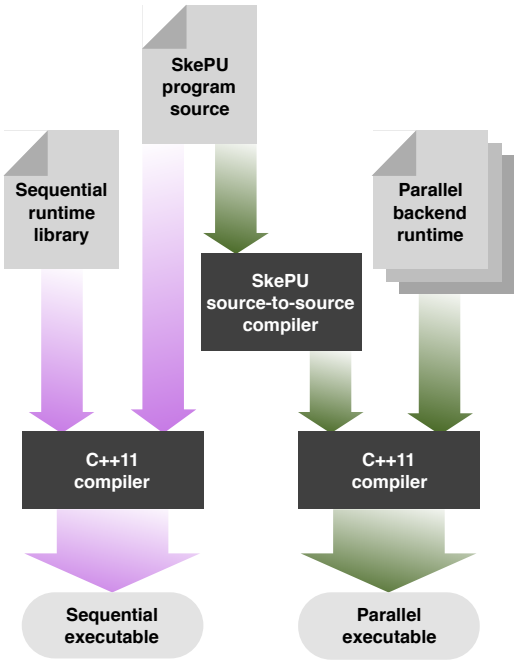


Figure 5.1: SkePU compiler chain.

5.2 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.

Listing 5.1: 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>);
}
```

- **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 pre-compiler 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 program. The precompiler also transforms skeleton instances, redirecting to a completely different implementation accepting the `struct`s 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 5.1 can be seen in Listing 5.2. 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 pre-compiler, as well as additional boilerplate glue-code used to launch said kernels. Listing 5.3 contains an excerpt of the kernel code generated (in the OpenCL kernel language, as a static text string) for the SkePU program in Listing 5.1 and Listing 5.4 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.

In the future, the precompiler role will be expanded to include automated selection of system-specific user function specializations, guided by a platform description language [37]. The precompiler can either select the most appropriate specialization directly, or include multiple variants and generate logic

Listing 5.2: 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 5.3: 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 5.4: 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");
}
```

to select the best one at run-time based on dynamic conditions. The extensibility was an important motivation when deciding to construct SkePU 2 as a precompiler-based framework using the Clang libraries.

5.3 Backends

After a SkePU program has been processed by the pre-compiler, 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 7.

5.3.1 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 pre-compiler and the sequential backend, the application uses the full smart container implementation and goes through the 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.

5.3.2 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.

```
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 5.2: Clang AST of the `add` user function from Listing 5.1.

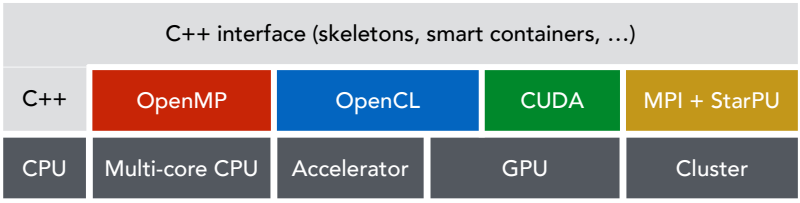
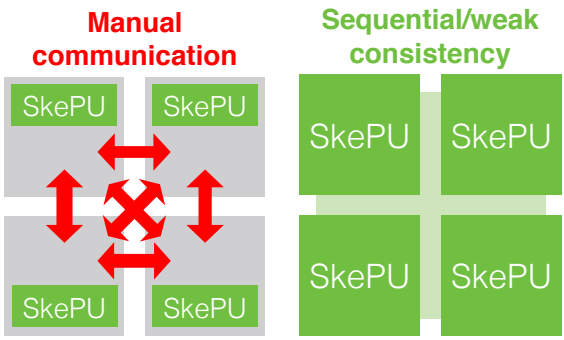


Figure 5.3: Backends available in SkePU.



(a) Outer MPI (b) Inner MPI

Figure 5.4: Cluster execution modes in SkePU.

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 9.7.1.

5.3.3 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.)

5.3.4 Cluster backend: StarPU and MPI

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 [5], 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 [4]. 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 strict 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 4.15) 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 only remaining issue in SPMD execution is that I/O operations need be protected from being executed everywhere. To make sure that such code is executed only by the SPMD master process, such code should be guarded by the `skepu::external` construct covered in Section 4.15.1. The distributed data of the containers to be output is automatically flushed. Before the code block is evaluated `read` containers are gathered, and afterwards the `write` containers are distributed using scattering.

5.4 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 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.

5.5 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 tool chain 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 pre-compilation occurs on a system with the full pre-compiler 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.

5.6 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[2]. *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.

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

6 Extending smart 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

Experimental evaluation is presented later, in Chapter 9.

In this chapter, we present an extension for the SkePU to improve the performance of sequences of transformations on smart 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.

6.1 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 [21] 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* [77]. 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 9.

We first present an overview of big data technologies in Section 6.2. Section 6.3 details our contribution, lazy evaluation and loop tiling for the SkePU skeleton framework, and its implementation. Section 6.4 compares our contribution to compile-time kernel fusion and presents applications better suited to our dynamic solution. Section 6.5 lists related work.

Performance evaluation results are presented later, in Section 9.3.

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

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

6.2.1 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 [56, 75]. 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.

6.2.2 Spark

The goal of Spark is to improve upon the performance of MapReduce, specifically in iterative applications such as machine learning [50], by means of avoiding unnecessary reads and writes to the file system [79]. 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. 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

Listing 6.1: Word count program with Spark in Scala.

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

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.

6.3 Lazily evaluated skeletons with tiling

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

6.3.1 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 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 containers are element-wise transformations, the lineage graph can continue to be built up (otherwise, it is an *evaluation point*, see Section 6.3.4). 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 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.

Listing 6.2: 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]
```

Listing 6.3: After transformation.

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

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

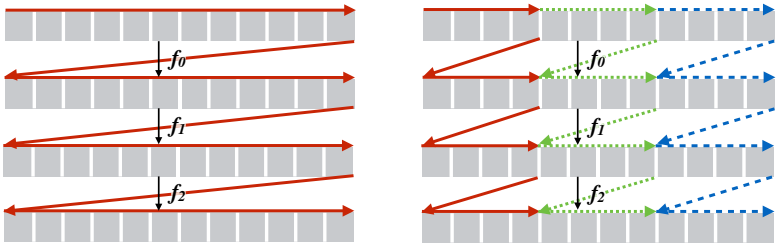
6.3.2 Backend selection

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.

6.3.3 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 [22].

Loop fusion is a known technique for low-level compiler optimization. Two or more loops can be combined into one, as in Listings 6.2 and 6.3. However, in the context of skeleton lineages, the overhead of switching between the contexts of different skeleton invocations for every single element 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.



(a) No tiling (b) Tiling

Figure 6.1: Sequence of skeleton transformations with and without tiling.

Loop tiling (also known as loop blocking or nesting) is a well-known technique for optimizing the locality of reference [41], especially on large, two-dimensional data sets.

Figure 6.1 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 6.1a. With tiling, a chunk of the container space will have all transformations applied before moving on to the next chunk, as in Figure 6.1b. (In practice, there may be multiple containers involved and the destination container may or may not be the same as one of the inputs.)

6.3.4 Evaluation points

In contrast to the read-only RDDs in Spark, SkePU offers much more flexibility in how smart containers can be used. One consequence of this is that operations on smart 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.

6.3.5 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.

6.3.6 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 6.3.7. 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 6.4. The program contains only `Map` skeletons with various transformations and parameter configurations. The resulting lineage graph, combined for all smart containers in the program, shows skeleton invocations and dependencies, see Figure 6.2. 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.

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

Listing 6.4: 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);
```

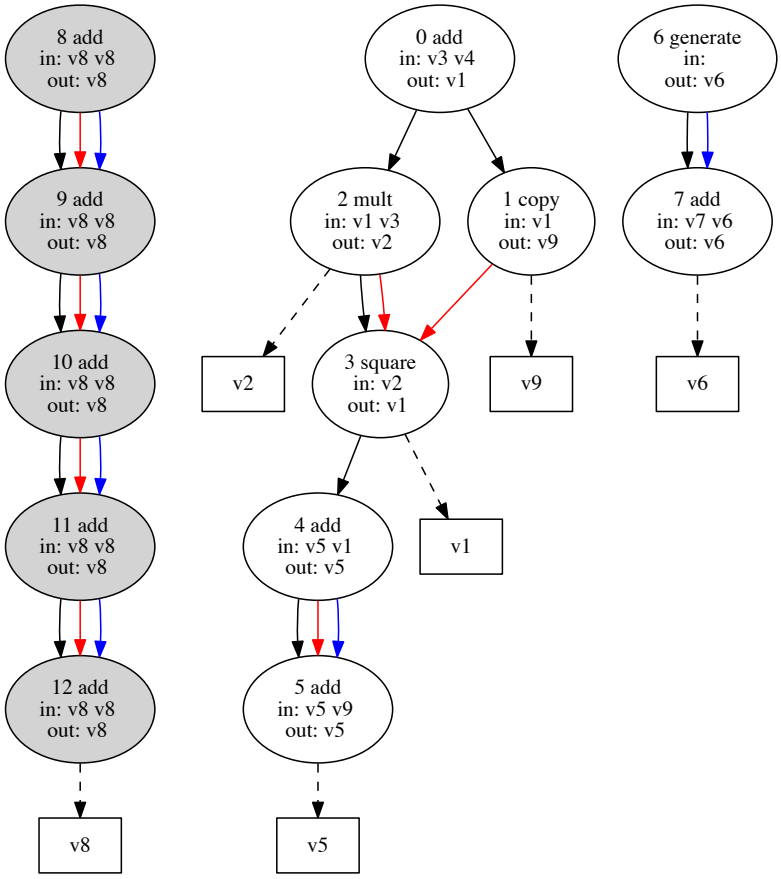


Figure 6.2: Lineage constructed by the program in Listing 6.4.

6.3.7 Lazy tiling for stencil computations

MapOverlap is the SkePU skeleton implementing stencil operations on smart 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 Muller [23] 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 6.3, 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 $i = 0, \ldots, n-1$

1: **procedure** Triangle
2: $B \leftarrow$ tiling block size
3: $overlap_i \leftarrow$ overlap for stencil computation i
4: $indent_i \leftarrow$ prefix sum $overlap_0 + overlap_1 + \ldots + overlap_i$ 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.

The approach is exemplified in Figure 6.3. 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.

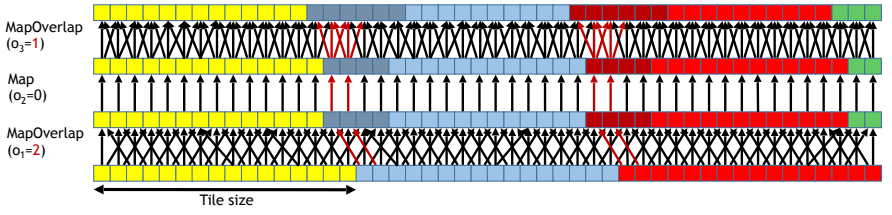


Figure 6.3: Evaluation scheme for tiling sequenced MapOverlap calls.

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 $overlap_0 + overlap_1 + ... + overlap_i$ 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 $BT - 1$

6.4 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* [64]. 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 6.5 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.

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.

6.4.1 Polynomial evaluation using Horner's method

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

Listing 6.5: Manual kernel fusion in SkePU where a, b, and c are smart 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);
```

Listing 6.6: Parallel polynomial evaluation in SkePU. User functions are omitted for brevity.

```
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;
}
```

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

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.

Listing 6.7: 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;
}
```

Noticing that the formula is a repeated sequence of multiplications and additions, a data-parallel implementation can be expressed as in Listing 6.6. Manual kernel fusion can be applied to the loop body as in Listing 6.5, 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).

6.4.2 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 $x^2x^8 = x^2((x^2)^2)^2$ 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 6.7. In this case, in addition to the properties of the program in Listing 6.6, invocations of the `mult` skeleton instance is skipped for loop iterations where the corresponding bit in x is 0. The resulting lineage will look very different when varying the exponent.

6.4.3 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

Listing 6.8: 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);
}
```

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 6.8.

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 6.4 illustrates the lineage building and evaluation in this application.

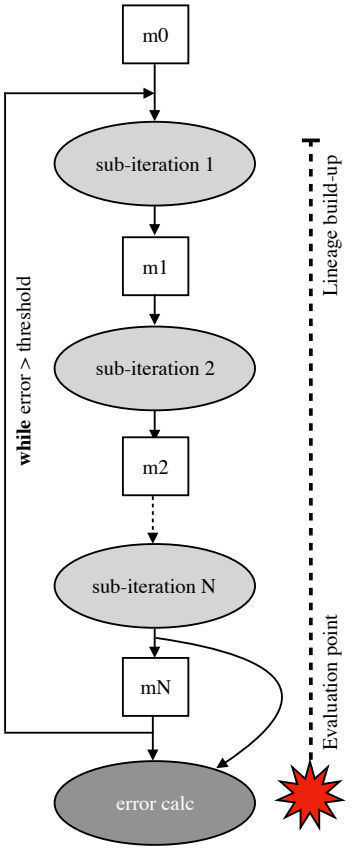


Figure 6.4: Lineage building for iterative heat propagation using MapOverlap.

6.5 Related work

Apache Spark is the primary inspirational source for this work, introducing lineages and related concepts for big-data processing on large distributed memory clusters. Spark is a development of MapReduce [21] and the Hadoop implementation. For a related discussion on big data frameworks and the connections to skeleton-like programming paradigms, see [51].

Another very recent project is *FlashR* by Zheng et al.[80]. 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 [64] for GPU-based systems.

7 Hybrid CPU-GPU skeleton evaluation

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 9.

In this contribution, 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 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.

7.1 Introduction

The ever-growing demand for higher performance in computing, puts requirements on modern programming tools. Today parallelism stands for the majority of the performance potential and even if heterogeneous, multi-core and accelerator equipped systems have been the norm for more than a decade, we still face the challenge of automatically exploiting the performance po-

tential of such systems. 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.

This chapter presents a new hybrid execution backend for SkePU. The work is based on Öhberg's master's thesis [54]. The main contribution is the introduction of workload partitioning implementations for all data parallel skeletons in SkePU, capable of dividing the work between an arbitrary number of CPU cores and accelerators

The rest of this chapter is structured as follows: Section 7.2 starts by introducing the task-based programming library StarPU. Section 7.3 presents the new hybrid backend implementation and how the workload is partitioned in all skeletons. This is followed by Section 7.4, 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 9.

7.2 StarPU

The old implementation of hybrid execution in SkePU 1 used the StarPU library as a backend. This implementation was ported to SkePU 2 as a baseline to compare the new hybrid backend to. StarPU[1] 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 **eager-based**,

[1] http://starpu.gforge.inria.fr

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 [74].

StarPU has been used in a number of application scenarios, recent examples including finite-volume CFD [8] and seismic wave modeling [48].

7.3 Workload partitioning and implementation

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 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 7.1. This example shows how to set up hybrid execution for 16 CPU threads and one accelerator, where 20% of the workload will be com-

Listing 7.1: 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);
```

puted 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 7.1.

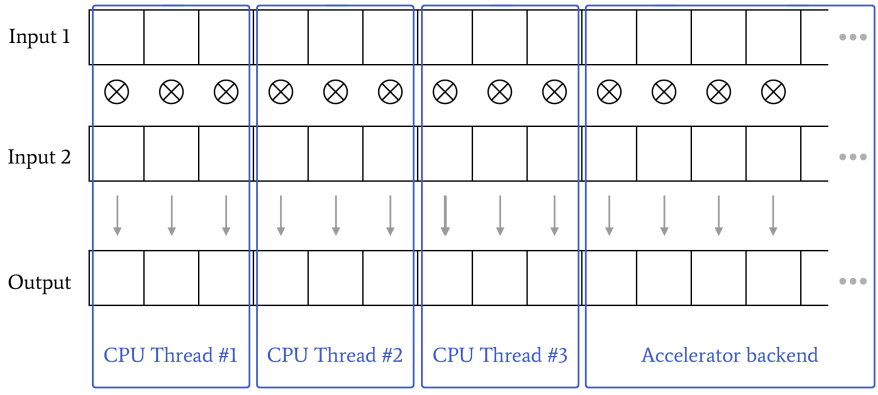


Figure 7.1: Partitioning of the Map skeleton.

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 reductions. 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 7.2.

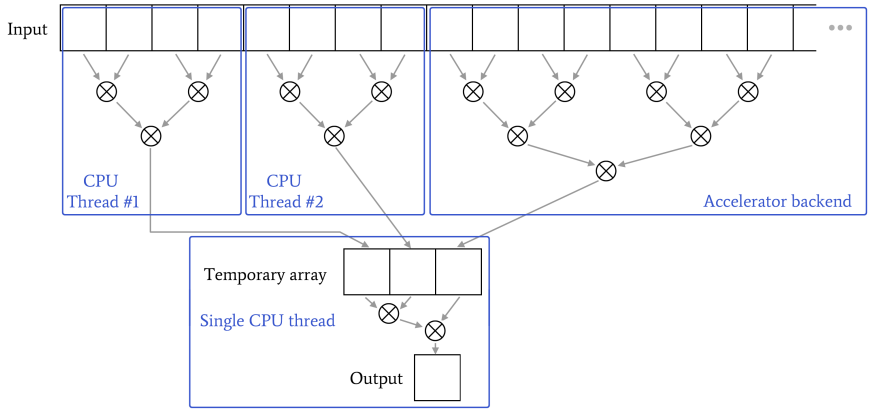


Figure 7.2: Partitioning of Reduce skeleton.

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 7.3.

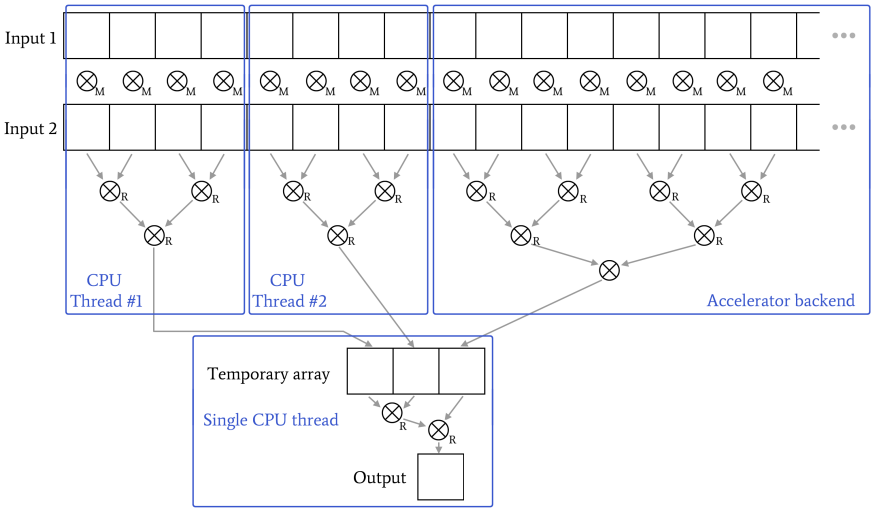


Figure 7.3: Partitioning of MapReduce skeleton.

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 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 7.4. The work of a single user function call is highlighted in yellow.

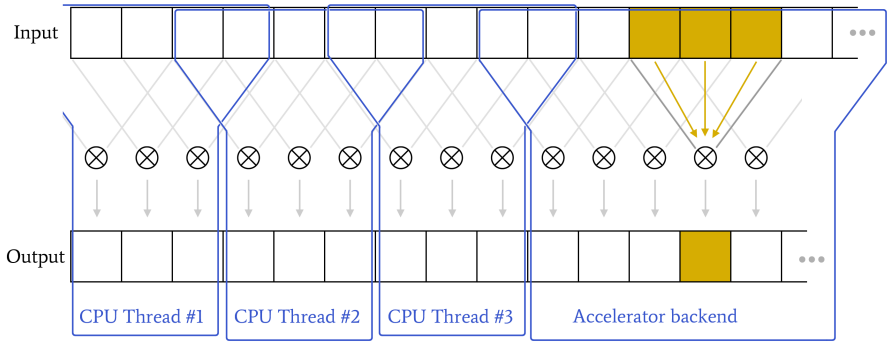


Figure 7.4: Partitioning of MapOverlap skeleton.

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 7.5.

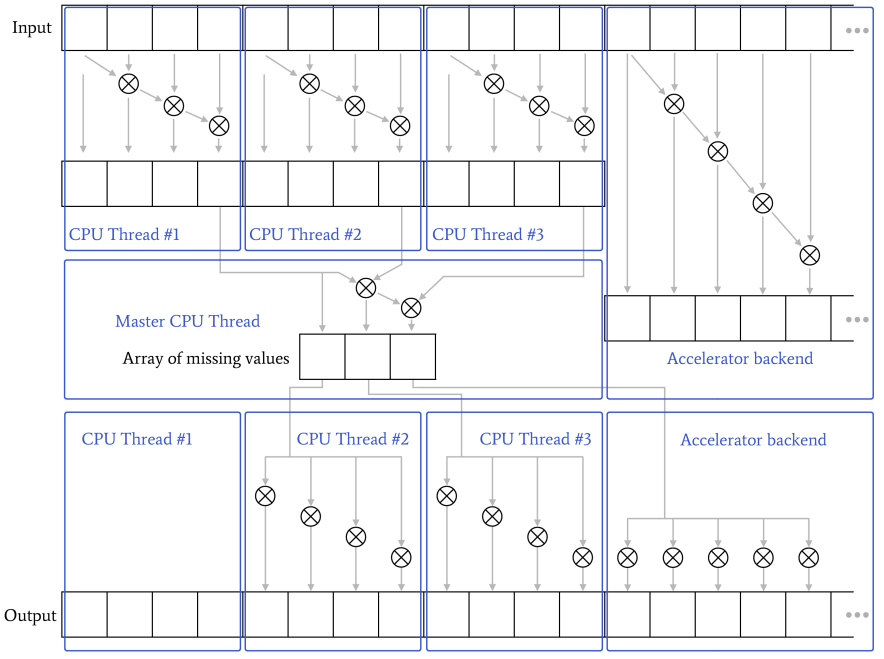


Figure 7.5: Partitioning of Scan skeleton.

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 Map 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 [18], 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.

7.3.1 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 [16], 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 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.

7.4 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 con-

tainer 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. [44] 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:

$$t = ax + b \tag{7.1}$$

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:

$$t_{cpu} = a_{cpu}NR + b_{cpu} \tag{7.2}$$

$$t_{acc} = a_{acc}N(1 - R) + b_{acc} \tag{7.3}$$

The workload is perfectly balanced between the CPU and the accelerator if $t_{cpu} = t_{acc}$. Combining the equations 7.2 and 7.3 and solving for the partition ratio R gives:

$$R = \frac{a_{acc}N + b_{acc} - b_{cpu}}{N(a_{cpu} + a_{acc})} \tag{7.4}$$

At runtime Equation (7.4) 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.

8 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 9.

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.

8.1 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 [31]. Our work is also tightly integrated with a platform modeling system [37] 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 8.2, followed by several use cases in Section 8.3.

8.2 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 [37].

A platform description (such as the one given in Listing 8.1) 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 8.3 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 8.4). Here XPDL metaprogramming queries or other statically evaluated expressions can be used. As the model in Listing 8.1 declares the platform to support this extension (line 7 in Listing 8.1), 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 a standard naming schema, with one directory for each user function. A *component implementation descriptor* file defines the hardware platform and

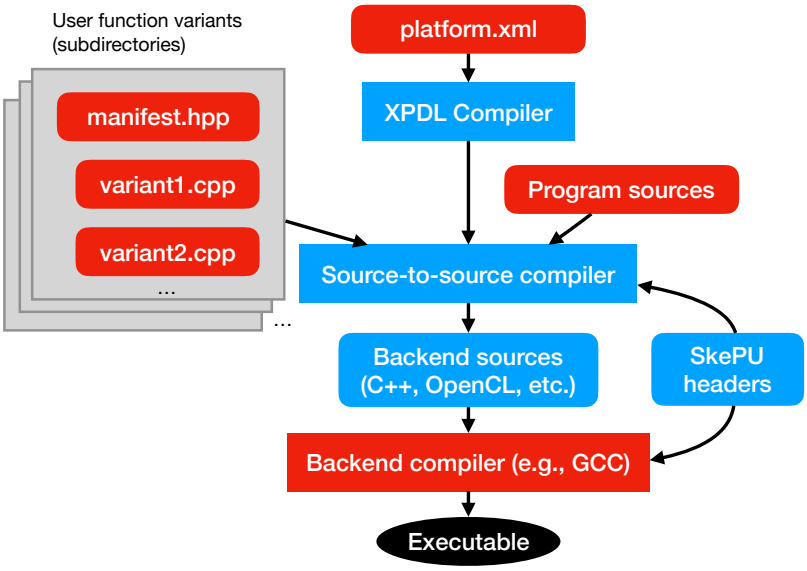


Figure 8.1: Overview of the components involved in SkePU variant selection and subsequent build process.

Listing 8.1: XPDL model for an Intel Xeon Gold 6130 CPU. Please refer to XPDL publications [37] 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>
```

run-time requirements for each variant. See Figure 8.1 for an illustration of the workflow: the outlined rectangles denote directories in the file system and the filled rectangles represent files.

Listing 8.2: A SkePU program performing element-wise vector addition.

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

   int main(int argc, char *argv[])
   {
5    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);
   }
```

8.3 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.

8.3.1 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* [43, 42, 45], 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 8.2. 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 8.3 contains a variant of `add` that is defined in a separate file as outlined in Section 8.2. This file is referenced from the manifest, as seen in Listing 8.4. 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

Listing 8.3: 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 8.4: 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
  )
};
```

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.

8.3.2 Generalized multi-variant components with the Call skeleton

The version 2 revision of SkePU [30] 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. 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.

[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.

A common class of computations that fit the above criteria are sorting algorithms. Another example is the fast Fourier transform (FFT) [78], 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 8.1 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.

8.3.3 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 [32] 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.

8.4 Related work

High-level parallel programming using skeletons or patterns [12] 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 [24, 30], FastFlow [1], Marrow [46], GrPPI [63], Thrust [6] and others.

None of these skeleton programming frameworks considered automated, platform-specific operator specialization for multi-element groups in skeleton instantiations or calls. Lift, [71] 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 [2, 59]. 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* [68] 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 [31] focuses on generating multiple alternative application variants for a single operation.

The limitations of compiler auto-vectorization are explored by Larsen et al. [42] who also suggest improvements to the programming language and environment to facilitate the optimization in more scenarios.

9 Results

This chapter collects quantitative results from evaluations done on SkePU in general as well as the specific contributions presented throughout the thesis.

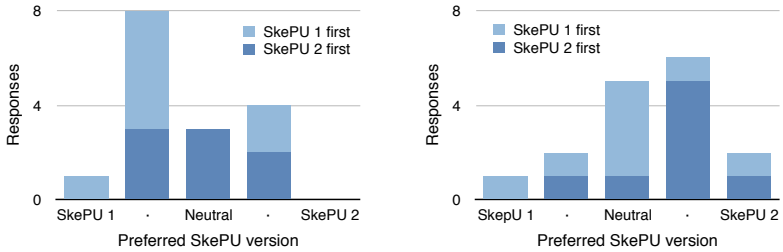
Being a high-level programming interface, we are interested in the usability of SkePU for programmers of different skill levels. Section 9.1 therefore presents a survey on the usability of SkePU code, including aspects of readability and programmer feedback through error messages.

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 9.6.

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 9.2 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 9.3 evaluates the performance of the lineages optimization (Chapter 6), Section 9.4 presents performance results of the hybrid backend implementation from Chapter 7, and Section 9.5 evaluates the multi-variant user functions of Chapter 8.

Finally, Section 9.7 contains micro-benchmarks (synthetic code or small-scale applications) on features introduced in SkePU 3.



(a) Simple example (b) Complex example

Figure 9.1: Comparison of code clarity, SkePU 1 vs. SkePU 2.

9.1 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* [30]. The survey was based on the work-in-progress programming interface for SkePU 2.

9.1.1 Readability

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 [26] 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 9.1 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

Listing 9.1: 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);
```

Listing 9.2: Faulty SkePU 2 code.

```
[[skepu::userfunction]]
float plus_f(float a) {
  return a;
}

skepu2::Vector<float> v(N);
auto [[skepu::instance]] globalSum = skepu2::Reduce(plus_f);
globalSum(v);
```

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.

9.1.2 Improved type safety

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 9.1 shows the error in SkePU 1 code, and Listing 9.2 illustrates the same in SkePU 2 syntax.

The SkePU 1 example compiles without problem, and only at run-time terminates with the error message in Listing 9.3. 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

Listing 9.3: Error messages from SkePU 1 and 2.

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

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

the affected skeleton instance. (The message does not directly describe the issue, an aspect which can be further improved with C++11's `static_assert`.)

9.2 Initial SkePU 2 performance evaluation

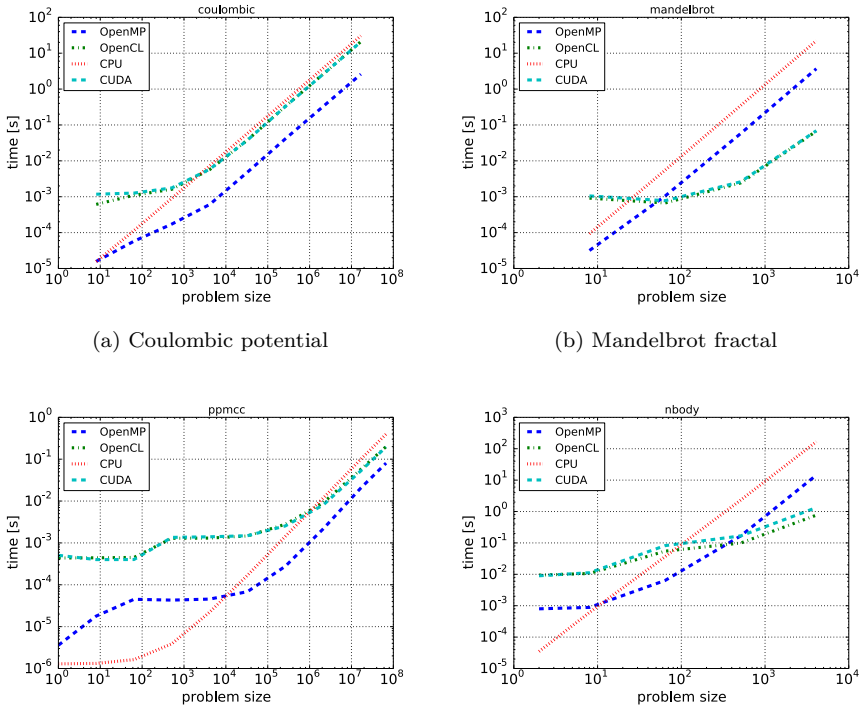
The results in this section were first published in the paper *SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems* [30]. Experiments were carried out on an early version of SkePU 2 and on hardware available at the time.

The system used for testing consists of two eight-core Intel Xeon E5-2660 "Sandy Bridge" processors at 2.2GHz 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 9.3. All tests include data movement to and from accelerators, where applicable.

Results are shown in Figure 9.2. 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



(a) Coulombic potential

(b) Mandelbrot fractal

(c) Pearson product-movement correlation coefficient

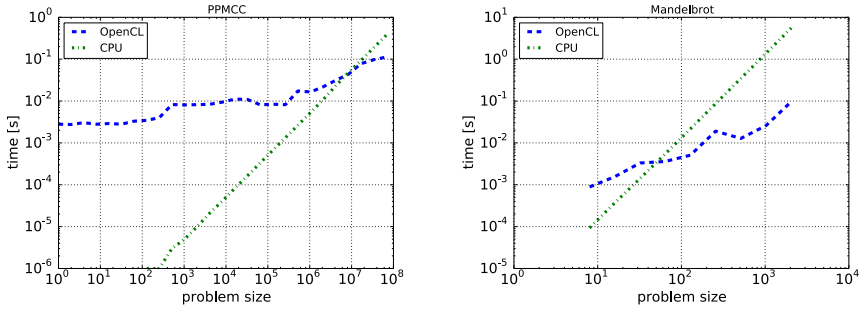
(d) N-body simulation

Figure 9.2: Test program evaluation results. Log-log scale.

that is needed to calculate the return value. The problem size is one side of the output image.

- **Coulombic potential**
  Calculates electrical potential in a grid, from a set of charged particles. 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.

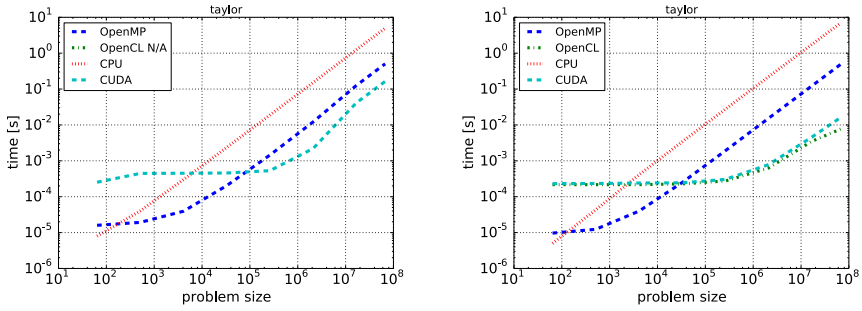
- **N-body simulation**
  Performs an N-body simulation on randomized input data. The program is similar to Coulombic potential, both in its iterative nature and the types of skeletons used.



(a) Pearson product-movement correlation coefficient

(b) Mandelbrot fractal

Figure 9.3: Evaluation results on Xeon Phi using OpenCL.



(a) SkePU 1.2

(b) SkePU 2

Figure 9.4: Comparison of Taylor series approximation

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 9.4 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.

9.3 Performance evaluation of lineages

The results in this section were first published in the paper *Extending smart containers for data locality-aware skeleton programming* [28]. 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 6.4.1, *exponentiation by repeated squaring* in Section 6.4.2, and heat propagation from Section 6.4.3. The first two are sequences of `Map`s while the latter uses `MapOverlap`.

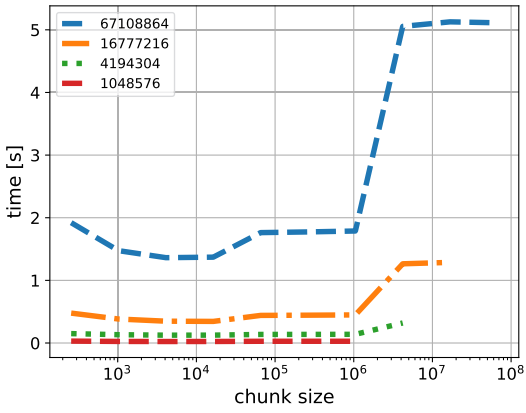
All performance evaluation was performed on Intel Xeon CPU model E5-2630L at 2.40 GHz and hyperthreading enabled. The cache memory hierarchy was as follows: 32 kB L1 data cache, 256 kB L2 cache, and 15 MB L3 cache. All programs were precompiled with the SkePU source-to-source compiler into C++ source 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.

9.3.1 Sequences of Maps

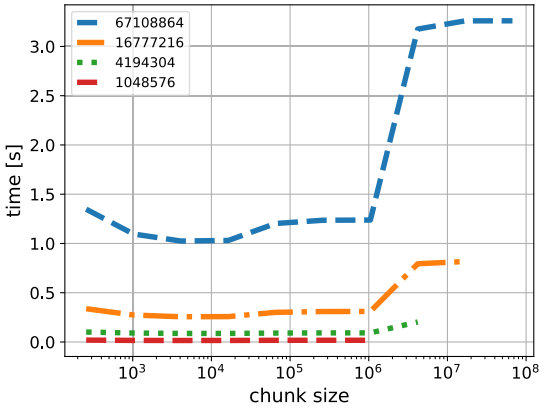
Both Horner's method and exponentiation by repeated squaring are iterative sequences of `Map` skeletons with small user functions and virtually ideal situations for lineage building. With a trivial user function, the benchmark is memory bound and tiling should give a significant performance improvement. The source code is very similar to Listings 6.6 and 6.7, but instrumented with measurement directives and additional constructs for explicitly requesting lineage evaluation.

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 9.5, 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 the tests, but this will be dependent on the memory hierarchy of the target system. There is still



(a) Horner's method



(b) Exponentiation by repeated squaring

Figure 9.5: Benchmark results for Map sequences.

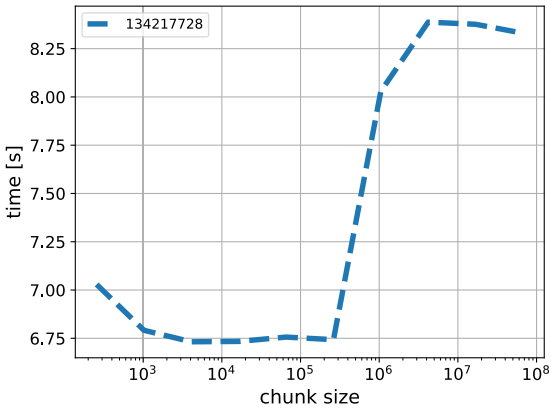


Figure 9.6: Heat propagation with tiled MapOverlap.

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.

9.3.2 Heat propagation

We have also instrumented the code from the heat propagation example in Listing 6.8 for performance evaluation. Using an unrolling factor of 4 gives the results in Figure 9.6. 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 9.6 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.

9.4 Hybrid backend

The results in this section were first published in the paper *Hybrid CPU–GPU execution support in the skeleton programming framework SkePU* [55].

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.

9.4.1 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 9.7. 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.

9.4.2 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 9.1. 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 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

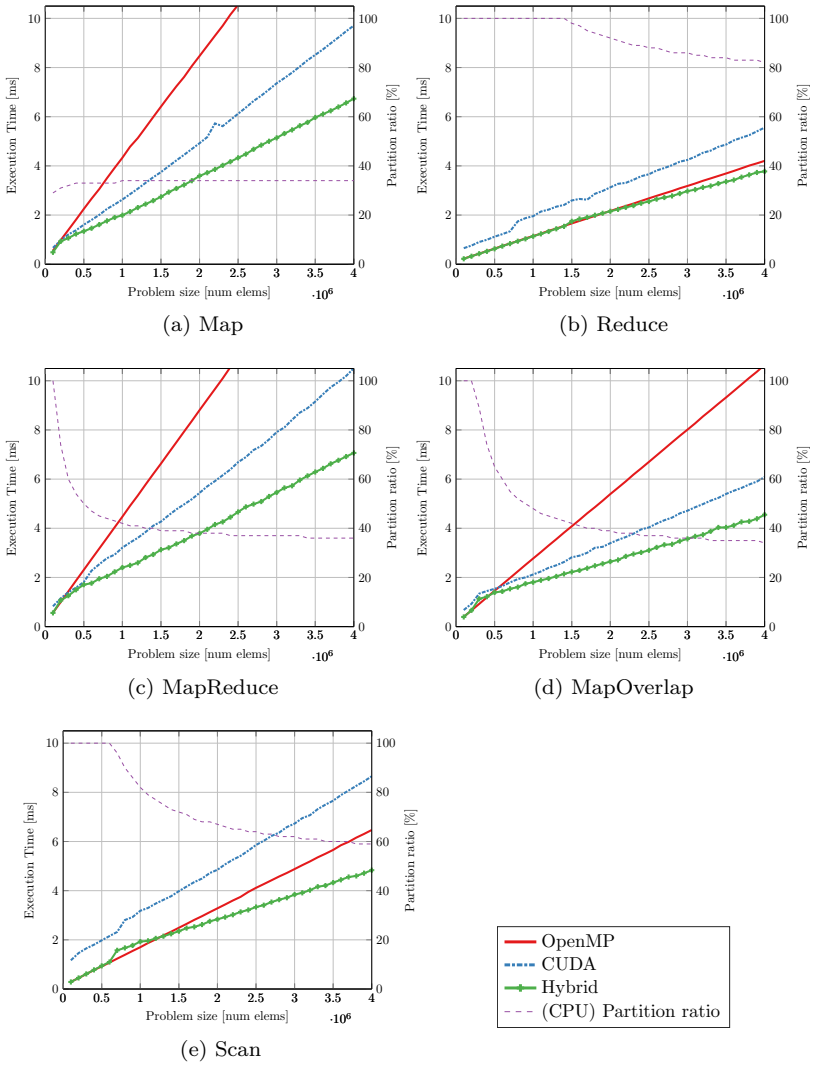


Figure 9.7: Execution time of skeletons

used in earlier research to show the upper bound of hybrid execution implementations [44, 34, 65]. We let the oracle execute the application using the hybrid backend with a manually set partition ratio for each skeleton instance, 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

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 9.1: List of applications used in the evaluation.

in Figure 9.8. 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.

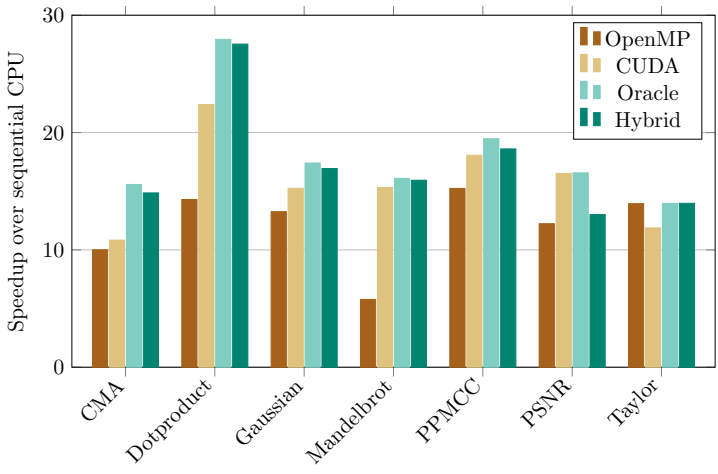


Figure 9.8: Speedups comparisons of example applications

9.4.3 Comparison to dynamic hybrid scheduling using StarPU

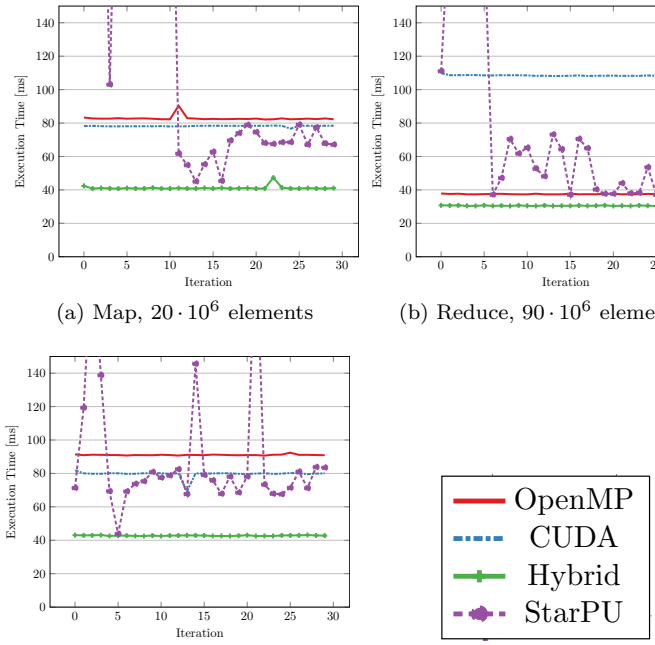
Finally, we show the improvement over the experimental hybrid execution implementation based on the StarPU runtime system that was implemented in SkePU 1. To make a fair comparison, parts of the old StarPU implementation was ported to SkePU 2. We also compared the execution time to the OpenMP and CUDA 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 9.9. 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 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.

9.5 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* [29].

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 9.9: Execution time of repeated invocations of the same skeleton

9.5.1 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 8.3.1 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 9.10 the overhead of loading and storing vector registers is significant when there is only one vector instruction to compute. The choice between four element vector instructions

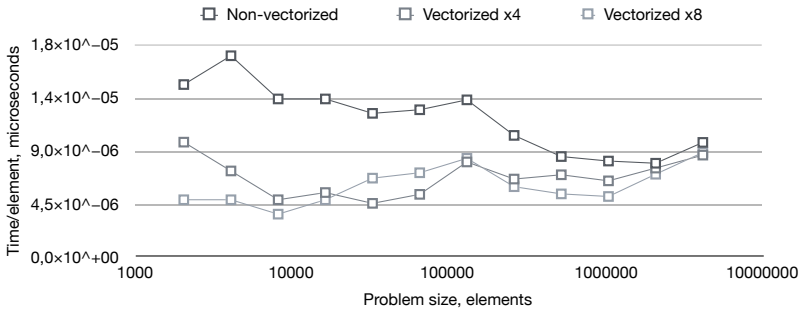


Figure 9.10: Element-wise vector addition, three variants. Execution time normalized (per element).

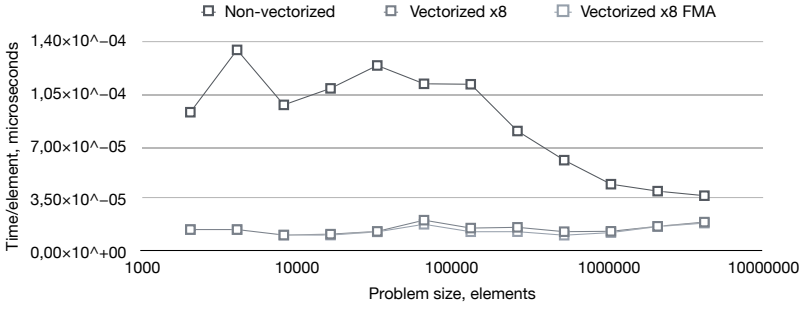


Figure 9.11: Element-wise complex vector multiplication, three variants. Execution time normalized (per element).

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 9.11). 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.

9.5.2 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 output 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*,

Table 9.2: User function variants for median filtering.

Variant	Time	Memory	Dependencies
Double loop	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	None
Histogram	$\mathcal{O}(n + \lvert D\rvert)$	$\mathcal{O}(\lvert D\rvert)$	None
`qsort`	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	C standard library

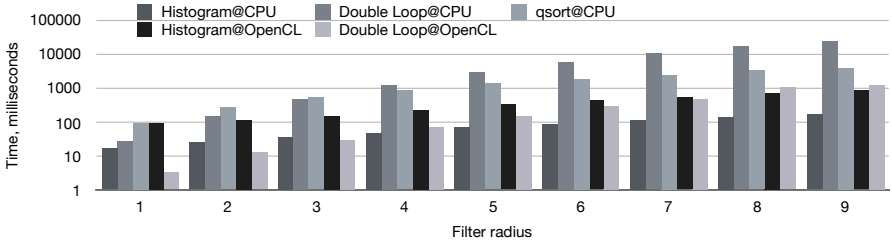


Figure 9.12: Median filtering using different median computation algorithms.

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 9.2 (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 9.12. 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.

9.6 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* [27].

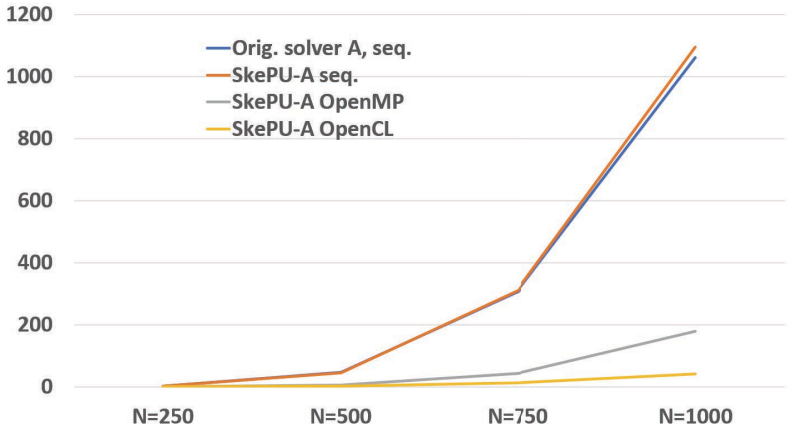


Figure 9.13: Execution times (seconds) of the SkePU 3 port of Variant A of the Embedded Runge-Kutta ODE solver implementation in the *Libsolve* library [38], solving the Brusselator 2D-MIX problem for 4 different system sizes.

9.6.1 Libsolve ODE solver

Figure 9.13 shows SkePU 3 performance results for an embedded ODE solver from the *Libsolve* library[1] [38], 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 9.13, for which it performs 124,532 calls to skeleton instances in total.

9.6.2 N-body

Figure 9.14 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.

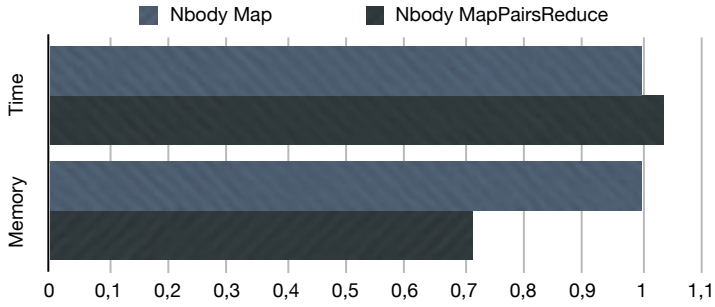


Figure 9.14: Normalized execution time and memory footprint for two variants of N-body: the `Map` variant (Listing B.1) and `MapPairsReduce` variant (Listing B.2).

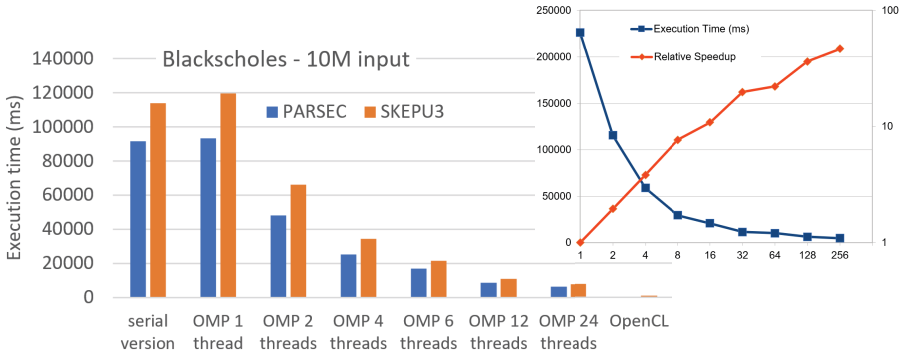


Figure 9.15: Execution time (ms) of the SkePU 3 port of the PARSEC benchmark Blackscholes on its largest input set. Left: Time with serial, OpenMP, OpenCL backends in SkePU and for manually multithreaded code in PARSEC. Right: Time and speedup with MPI backend on the cluster of Figure 9.17.

9.6.3 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 9.15 and 9.16. 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 space,

[1]Libsolve repository: https://github.com/UBT-AI2/rk

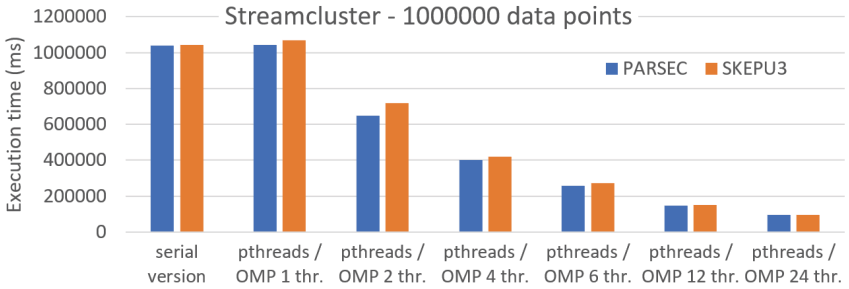


Figure 9.16: Execution times of the SkePU 3 port of the PARSEC benchmark Streamcluster on 10^6 data points.

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.

9.6.4 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* [57].

Figure 9.17 shows the scaling behavior of the SkePU 3 port of a brain simulation mini-application [57] 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.

9.7 Microbenchmarks 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* [27].

9.7.1 OpenMP scheduling modes

For the same machine, Figure 9.18 shows the positive performance effect of using dynamic scheduling in three data-parallel benchmarks with irreg-

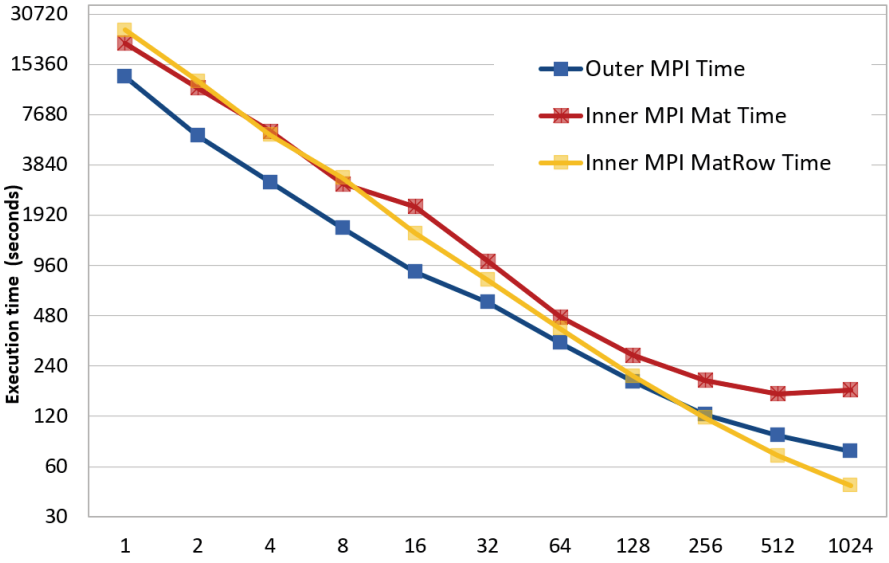


Figure 9.17: Execution time (in seconds, logarithmic scale) of the SkePU 3 port of a brain simulation mini-application [57] performing 200 time steps with 90000 neurons using up to 32 nodes (each with 32 cores) of the Tetralith cluster. "Outer MPI" refers to a manual MPI parallelization, the two "Inner-MPI" versions use SkePU's StarPU-MPI backend instead.

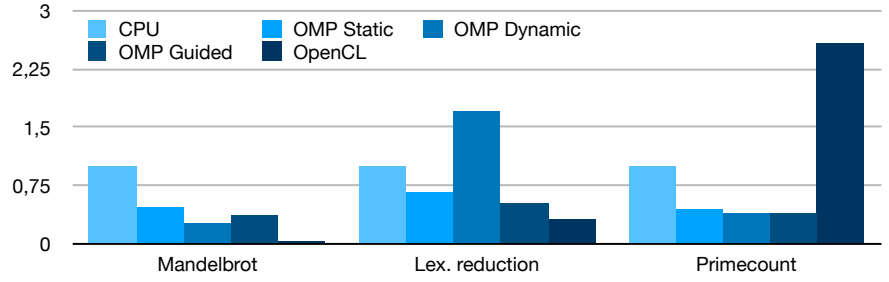


Figure 9.18: Execution time (normalized to the sequential CPU backend time) for three irregular-load benchmarks.

ular workload, in spite of the runtime overhead of dynamic scheduling: (1) 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.

Table 9.3: 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

9.7.2 SkePU memory consistency model

To illustrate the motivation behind the change of consistency model for SkePU smart containers (Section 4.15), 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 9.3. 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.

10 Conclusion and future work

This chapter concludes the thesis by summarizing the contributions and reflecting back on the work in a wider context. It also provides suggestions for directions of future work.

10.1 Conclusion

Algorithmic skeletons is a high-level interface to parallel computing, especially suitable for programmers without expert knowledge about parallel systems design. 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 adapt and improve as foundational languages such as C++ evolve to become more expressive and powerful. Building on these results, manifesting as what currently is SkePU version three, specific contributions have been introduced to improve hardware utilization though run-time optimizations of computational task graphs and hybrid CPU-GPU execution, as well as allowing also expert programmers to get the most out of

the framework with multi-variant user functions in skeletons. The thesis has also shown how SkePU has been adapted to target supercomputer clusters, which is important for large-scale HPC application domains.

The foundational improvements to SkePU presented in the thesis opened up a large field of potential new development directions and possible features, and therefore, the work has still only begun.

10.2 Future work

This section explores areas of interest for future work related to SkePU. Naturally, the focus on the SkePU framework makes these ideas skewing practical over theoretical.

10.2.1 Modernize the SkePU tuner

The auto-tuning mechanism in SkePU should be improved with the same techniques used in the rest of SkePU, such as variadic interface and taking advantage of the precompiler. This will enable more intelligent exploration of parameter space and targeting parameters outside of only the size of container arguments in skeleton invocations.

10.2.2 Skeleton fusion

SkePU 3 provides `MapReduce` and `MapPairsReduce` as explicitly separate skeleton constructs from the constituent parts, `Map`, `MapPairs`, and `Reduce`. The ”fused” skeletons provide measurable gain compared to separating the invocations, which motivates their existence. The programmer is however required to be aware of these skeletons and make explicit choices at design-time regarding matters of performance optimization, which would ideally be the responsibility of the framework. Several other skeleton programming interfaces fuse skeletons automatically, with some even being designed around transforming sequences and nestings of skeleton calls, with auto-tuning determining the best realization. SkePU has different challenges which make a fully general skeleton transformation system likely unfeasible, but it also means that a fusion in SkePU could be a worthwhile contribution to the scientific community and not just a catch-up of features in competing projects.

10.2.3 SkePU standard library

One of the goals of SkePU is to make efficient parallel and heterogeneous programming as simple and concise as possible. For common and straightforward computations, such as computing the sum of a vector, SkePU can still be needlessly verbose:

1. User function is defined: often four lines of code even for a single-expression function if formatting standards have to be followed.
2. Skeleton instance is declared and assigned: separate statement on one line, with a named object being introduced in the scope.
3. Invoking the skeleton: with the invocation being an expression (not a statement), this is already very lightweight.

Using lambda expressions combines step 1 and 2 and avoids introducing a named function in the surrounding scope. Still, C++ lambda expressions are syntactically heavy for single-expression functions. Therefore, we are exploring the addition of a *standard library* for SkePU. Such a library would define common user function operators and also ready-made skeleton instances for direct use. It would also provide fundamental types often needed by SkePU applications which do not exist at language-level in C++ or the target backends, e.g. complex numbers. The standard library would also be adopted internally inside SkePU for operations on smart containers.

10.2.4 Evaluating SkePU in further application domains

Skeleton programming is one out of many approaches to high-level parallel programming. Not every computation can be naturally described with skeleton patterns, or might not be *efficiently* expressed this way. However, each implementation of the skeleton paradigm has its own strengths and weaknesses; e.g., SkePU can re-use a lot of C or C++ code when integrating into existing code bases, but is generally limited to data-parallel computations. In an attempt to understand the applicability of SkePU in real-world use, we aim to evaluate how SkePU works in more and larger-scale applications than before. The project partners in *EXA2PRO* and existing benchmark suites, such as *PARSEC* [7] and *Rhodinia* [9], are examples of sources of such code bases. Results from these studies is also expected to improve SkePU itself in a closed feedback loop, as is already the case with several features new in SkePU 3.

10.2.5 Extending the skeleton set of SkePU, such as with 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 [28] are based on related ideas.

New smart containers used in conjunction with existing skeletons can also enable new application areas. An example of potential new smart container formats are *graph structures*.

10.2.6 Extended programmability survey

The usability survey conducted early in the work on SkePU 2 showed promising results but was limited in scope and is in some aspects outdated. A new survey would compare SkePU-ized application code against lower-level parallel programming interfaces and possibly also alternative high-level parallel programming frameworks.

A Definitions

A.1 Abbreviations

API	Application programming interface
ASIC	Application-specific integrated circuit
AST	Abstract syntax tree
DSEL	Domain-specfic embedded language (also EDSL)
EU FP7	European Union Seventh Framework Programme
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
IDE	Integrated development environment
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
LLVM	The LLVM Compiler Infrastructure
MCC	Nordic Workshop on Multi-Core Computing
MSI	Modified–shared–invalid (cache coherence protocol)
NVCC	Nvidia's CUDA compiler
STL	C++ Standard Template Library
TBB	Intel Threading Building Blocks

A.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 both 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.

Pre-compiler
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.

A.3 SkePU-specific terminology

Backend
See Section 5.3. A type of programmable computation unit targeted for parallelization by SkePU.

Container proxy
See Section 4.14.

Elwise parameter
See Section 4.2.2.

Lineage
See Chapter 6.

Multi-variant (user function)
See Chapter 8.

Random-access parameter
See Section 4.2.1.

Smart container
See Section 4.13. 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 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.

B Application source code samples

Listing B.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)
```

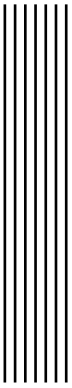
```
                    + (pi.z - pj.z) * (pi.z - pj.z));

      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 B.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);
  }
}
```



Bibliography

[1] 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.

[2] 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.

[3] Elvira Maria Arvanitou, Apostolos Ampatzoglou, Nikolaos Nikolaidis, Angeliki Tsintzira, Areti Ampatzoglou, and Alexander Chatzigeorgiou. "Investigating Trade-offs between Portability, Performance and Maintainability in Exascale Systems." In: *Presented at 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 20)*. Aug. 2020.

[4] 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. Springer, 2012, pp. 298–299. ISBN: 978-3-642-33518-1.

[5] 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.

[6] Nathan Bell and Jared Hoberock. "Thrust: A Productivity-Oriented Library for CUDA." In: *GPU Computing Gems, Jade Edition* (2011).

[7] 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.

[8] 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* (2017). ISSN: 1877-7503.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. "Rodinia: A benchmark suite for heterogeneous computing." In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009, pp. 44–54.

[10] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. *The Münster Skeleton Library Muesli - A Comprehensive Overview*. ERCIS Working Paper No. 7. 2009.

[11] Murray Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming." In: *Parallel computing* 30.3 (2004), pp. 389–406.

[12] Murray I. Cole. *Algorithmic skeletons: Structured management of parallel computation*. Pitman and MIT Press, Cambridge, Mass., 1989.

[13] Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. "Parallelizing High-Frequency Trading Applications by using C++ 11 Attributes." In: *IEEE Trustcom/BigDataSE/ISPA 2015*. Vol. 3. IEEE. 2015, pp. 140–147.

[14] 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.

[15] Usman Dastgeer, Johan Enmyren, and Christoph W Kessler. "Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems." In: *Proceedings of the 4th International Workshop on Multicore Software Engineering*. ACM. 2011, pp. 25–32.

[16] 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.

[17] Usman Dastgeer and Christoph Kessler. "Smart containers and skeleton programming for GPU-based systems." In: *International Journal of Parallel Programming* (June 2016). DOI: 10.1007/s10766-015-0357-6.

[18] Usman Dastgeer, Lu Li, and Christoph Kessler. "Adaptive implementation selection in the SkePU skeleton programming library." In: *Advanced Parallel Processing Technologies*. Springer, 2013, pp. 170–183.

[19] Usman Dastgeer, Lu Li, and Christoph Kessler. "The PEPPHER composition tool: performance-aware composition for GPU-based systems." In: *Computing* 96.12 (2013), pp. 1195–1211.

[20] 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.

[21] 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. URL: http://doi.acm.org/10.1145/1327452.1327492.

[22] Peter J. Denning. "The Locality Principle." In: *Commun. ACM* 48.7 (July 2005), pp. 19–24. ISSN: 0001-0782. DOI: 10.1145/1070838.1070856.

[23] H. Eissfeller and S. M. Muller. "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.

[24] 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*. ACM. 2010, pp. 5–14.

[25] Steffen Ernsting and Herbert Kuchen. "Algorithmic skeletons for multi-core, multi-GPU systems and clusters." In: *Int. J. of High Perf. Computing and Networking* 7 (2 2012), pp. 129–138.

[26] 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: Linköping University, 2016.

[27] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. "SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters." In: *Presented at HLPP 2020, pending journal submission* (2020).

[28] 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.

[29] 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.

[30] 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* (2017), pp. 1–19. ISSN: 1573-7640. DOI: 10.1007/s10766-017-0490-5.

[31] D. Fabregat-Traver and P. 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.

[32] 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.

[33] 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.

[34] Dominik Grewe and Michael O'Boyle. "A static task partitioning approach for heterogeneous systems using OpenCL." In: *Compiler Construction*. Springer. 2011, pp. 286–305.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] P. Larsen, R. Ladelsky, J. Lidman, S. A. McKee, S. Karlsson, and A. 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.

[43] 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: https://doi.org/10.1016/S0167-8191(05)80035-3.

[44] 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.

[45] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. 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.

[46] 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*. Vol. LNCS 8097. Springer, 2013, pp. 874–885.

[47] Gabriel Martinez, Mark Gardner, and Wu-Chun Feng. "CU2CL: A CUDA-to-OpenCL translator for multi-and many-core architectures." In: *IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2011, pp. 300–307.

[48] V. Martínez, D. Michéa, F. Dupros, O. Aumage, S. Thibault, H. Aochi, and P. 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)*. Oct. 2015, pp. 1–8.

[49] Jens Maurer and Michael Wong. *Towards support for attributes in C++ (Revision 6)*. Tech. rep. N2761. ISO/IEC JTC1/SC22/WG21, 2008.

[50] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. "Mllib: Machine learning in Apache Spark." In: *Journal of Machine Learning Research* 17.34 (2016), pp. 1–7.

[51] 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.

[52] 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.

[53] 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.

[54] Tomas Öhberg. "Auto-tuning Hybrid CPU-GPU Execution of Algorithmic Skeletons in SkePU." MA thesis. Linköping University, Software and Systems, 2018, p. 68.

[55] 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.

[56] Matthew Felice Pace. "BSP vs MapReduce." In: *Procedia Computer Science* 9 (2012), pp. 246–255. ISSN: 1877-0509. DOI: http://dx.doi.org/10.1016/j.procs.2012.04.026.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] Fethi A. Rabhi and Sergei Gorlatch. *Patterns and skeletons for parallel and distributed computing*. Springer-Verlag, London, UK, 2003. ISBN: 1-85233-506-8.

[62] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. "Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons." In: *Proc. Symposium on Applied Computing (SAC'19)*. ACM, 2019, pp. 1534–1543.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. "SkelCL–A portable skeleton library for high-level GPU programming." In: *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)*. 2011.

[71] 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.

[72] 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.

[73] 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. Cham: Springer International Publishing, 2019, pp. 291–303. ISBN: 978-3-030-29400-7.

[74] Universite de Bordeaux, CNRS, and Inria. *StarPU Handbook ver. 1.2.4*. Tech. rep. 2018.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Cluster computing with working sets." In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. USENIX Association Berkeley, CA, USA. 2010, pp. 10–10.

[80] 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.

Department of Computer and Information Science
Linköpings universitet

Licentiate Theses

Linköpings Studies in Science and Technology
Faculty of Arts and Sciences

No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)

No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.

No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.

No 104 **Shamsul I. Chowdhury: S**tatistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.

No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.

No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.

No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.

No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.

No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.

No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.

No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.

No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.

No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.

No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.

No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.

No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.

No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.

No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.

No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.

No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.

No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.

No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.

No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.

No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.

No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.

No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.

No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.

No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge- Bases, 1991.

No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.

No 318 **Lena Strömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.

No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.

No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.

No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.

No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.

No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Systems, 1992.

No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.

No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.

No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.

No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.

No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.
No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.
No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.
No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.
No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agent-teoretiskt perspektiv, 1994.
No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94 **Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylprogram, 1995.
No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.

No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591 **Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.
No 598 **Rego Granlund:** C^3Fire - A Microworld Supporting Emergency Management Training, 1997.
No 599 **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
No 607 **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
No 609 **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
FiF-a 4 **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
FiF-a 6 **Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
No 615 **Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
No 623 **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
No 626 **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
No 627 **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
No 629 **Gunilla Ivefors:** Krigsspel och Informationsteknik inför en oförutsägbar framtid, 1997**.**
No 631 **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
No 639 **Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997.
No 640 **Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
No 643 **Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997.
No 653 **Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 13 **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
No 674 **Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
No 676 **Jan Håkegård**: Hierarchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
No 668 **Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
No 675 **Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
FiF-a 14 **Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
No 695 **Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
No 700 **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
FiF-a 16 **Marie-Therese Christiansson:** Inter-organisatorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
No 712 **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
No 719 **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
No 723 **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
No 725 **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
No 730 **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
No 731 **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
No 733 **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
No 734 **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
FiF-a 21 **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
FiF-a 22 **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
No 737 **Jonas Mellin:** Predictable Event Monitoring, 1998.
No 738 **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
FiF-a 25 **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
No 742 **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
No 748 **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
No 751 **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader, 1999.
No 752 **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
No 753 **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.

No 754 **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
No 766 **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
No 769 **Jesper Andersson:** Towards Reactive Software Architectures, 1999.
No 775 **Anders Henriksson:** Unique kernel diagnosis, 1999.
FiF-a 30 **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
No 787 **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 788 **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
No 790 **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
No 791 **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
No 800 **Anders Subotic:** Software Quality Inspection, 1999.
No 807 **Svein Bergum**: Managerial communication in telework, 2000.
No 809 **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.
FiF-a 32 **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.
No 808 **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.
No 820 **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.
No 823 **Lars Hult:** Publika Gränsytor - ett designexempel, 2000.
No 832 **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.
FiF-a 34 **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.
No 842 **Magnus Kald:** The role of management control systems in strategic business units, 2000.
No 844 **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.
FiF-a 37 **Ewa Braf**: Organisationers kunskapsverksamheter - en kritisk studie av ”knowledge management”, 2000.
FiF-a 40 **Henrik Lindberg:** Webbaserade affärsprocesser - Möjligheter och begränsningar, 2000.
FiF-a 41 **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.
No. 854 **Ola Pettersson:** Deliberation in a Mobile Robot, 2000.
No 863 **Dan Lawesson**: Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.
No 881 **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.
No 882 **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.
No 890 **Annika Flycht-Eriksson:** Domain Knowledge Management in Information-providing Dialogue systems, 2001.
FiF-a 47 **Per-Arne Segerkvis**t: Webbaserade imaginära organisationers samverkansformer: Informationssystemarkitektur och aktörssamverkan som förutsättningar för affärsprocesser, 2001.
No 894 **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.
No 906 **Lin Han**: Secure and Scalable E-Service Software Delivery, 2001.
No 917 **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.
No 916 **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002.
FiF-a-49 **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
FiF-a-51 **Per Oscarsson:** Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
No 919 **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.
No 915 **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001.
No 931 **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.
No 933 **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.
No 938 **Bourhane Kadmiry**: Fuzzy Control of Unmanned Helicopter, 2002.
No 942 **Patrik Haslum**: Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002.
No 956 **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002.
FiF-a 58 **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002.
No 964 **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.
No 973 **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002.
No 958 **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002.
FiF-a 61 **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002.
No 985 **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002.
No 982 **Diana Szentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002.
No 989 **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002.
No 990 **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002.
No 991 **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002.

No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002.
No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.
No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.
No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.
FiF-a 62 **Lennart Ljung:** Utveckling av en projektivitetsmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.
No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.
No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.
No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.
No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.
No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.
No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.
No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems, 2003.
FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.
No 1024 **Aleksandra Tešanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.
No 1034 **Arja Vainio-Larsson**: Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.
No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befarade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.
FiF-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.
No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.
No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.
No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.
FiF-a 71 **Emma Eliason:** Effektanalys av IT-systems handlingsutrymme, 2003.
No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.
No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.
FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.
No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.
No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.
FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.
No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.
No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.
No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.
No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.
No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.
FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.
No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.
No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.
No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.
No 1130 **Ioan Chisalita:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.
No 1138 **Thomas Gustafsson:** Maintaining Data Consistency in Embedded Databases for Vehicular Systems, 2004.
No 1149 **Vaida Jakonienė:** A Study in Integrating Multiple Biological Data Sources, 2005.
No 1156 **Abdil Rashid Mohamed**: High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.
No 1162 **Adrian Pop:** Contributions to Meta-Modeling Tools and Methods, 2005.
No 1165 **Fidel Vascós Palacios:** On the information exchange between physicians and social insurance officers in the sick leave process: an Activity Theoretical perspective, 2005.
FiF-a 84 **Jenny Lagsten:** Verksamhetsutvecklande utvärdering i informationssystemprojekt, 2005.
No 1166 **Emma Larsdotter Nilsson:** Modeling, Simulation, and Visualization of Metabolic Pathways Using Modelica, 2005.
No 1167 **Christina Keller:** Virtual Learning Environments in higher education. A study of students' acceptance of educational technology, 2005.
No 1168 **Cécile Åberg:** Integration of organizational workflows and the Semantic Web, 2005.
FiF-a 85 **Anders Forsman:** Standardisering som grund för informationssamverkan och IT-tjänster - En fallstudie baserad på trafikinformationstjänsten RDS-TMC, 2005.
No 1171 **Yu-Hsing Huang:** A systemic traffic accident model, 2005.
FiF-a 86 **Jan Olausson:** Att modellera uppdrag - grunder för förståelse av processinriktade informationssystem i transaktionsintensiva verksamheter, 2005.
No 1172 **Petter Ahlström**: Affärsstrategier för seniorbostadsmarknaden, 2005.
No 1183 **Mathias Cöster**: Beyond IT and Productivity - How Digitization Transformed the Graphic Industry, 2005.
No 1184 **Åsa Horzella**: Beyond IT and Productivity - Effects of Digitized Information Flows in Grocery Distribution, 2005.
No 1185 **Maria Kollberg**: Beyond IT and Productivity - Effects of Digitized Information Flows in the Logging Industry, 2005.
No 1190 **David Dinka**: Role and Identity - Experience of technology in professional settings, 2005.

No 1191 **Andreas Hansson**: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.
No 1192 **Nicklas Bergfeldt:** Towards Detached Communication for Robot Cooperation, 2005.
No 1194 **Dennis Maciuszek:** Towards Dependable Virtual Companions for Later Life, 2005.
No 1204 **Beatrice Alenljung:** Decision-making in the Requirements Engineering Process: A Human-centered Approach, 2005.
No 1206 **Anders Larsson:** System-on-Chip Test Scheduling and Test Infrastructure Design, 2005.
No 1207 **John Wilander:** Policy and Implementation Assurance for Software Security, 2005.
No 1209 **Andreas Käll:** Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.
No 1225 **He Tan:** Aligning and Merging Biomedical Ontologies, 2006.
No 1228 **Artur Wilk:** Descriptive Types for XML Query Language Xcerpt, 2006.
No 1229 **Per Olof Pettersson:** Sampling-based Path Planning for an Autonomous Helicopter, 2006.
No 1231 **Kalle Burbeck:** Adaptive Real-time Anomaly Detection for Safeguarding Critical Networks, 2006.
No 1233 **Daniela Mihailescu:** Implementation Methodology in Action: A Study of an Enterprise Systems Implementation Methodology, 2006.
No 1244 **Jörgen Skågeby:** Public and Non-public gifting on the Internet, 2006.
No 1248 **Karolina Eliasson:** The Use of Case-Based Reasoning in a Human-Robot Dialog System, 2006.
No 1263 **Misook Park-Westman:** Managing Competence Development Programs in a Cross-Cultural Organisation - What are the Barriers and Enablers, 2006.
FiF-a 90 **Amra Halilovic:** Ett praktikperspektiv på hantering av mjukvarukomponenter, 2006.
No 1272 **Raquel Flodström**: A Framework for the Strategic Management of Information Technology, 2006.
No 1277 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Embedded Systems, 2006.
No 1283 **Håkan Hasewinkel:** A Blueprint for Using Commercial Games off the Shelf in Defence Training, Education and Research Simulations, 2006.
FiF-a 91 **Hanna Broberg:** Verksamhetsanpassade IT-stöd - Designteori och metod, 2006.
No 1286 **Robert Kaminski:** Towards an XML Document Restructuring Framework, 2006.
No 1293 **Jiri Trnka:** Prerequisites for data sharing in emergency management, 2007.
No 1302 **Björn Hägglund:** A Framework for Designing Constraint Stores, 2007.
No 1303 **Daniel Andreasson**: Slack-Time Aware Dynamic Routing Schemes for On-Chip Networks, 2007.
No 1305 **Magnus Ingmarsson:** Modelling User Tasks and Intentions for Service Discovery in Ubiquitous Computing, 2007.
No 1306 **Gustaf Svedjemo**: Ontology as Conceptual Schema when Modelling Historical Maps for Database Storage, 2007.
No 1307 **Gianpaolo Conte**: Navigation Functionalities for an Autonomous UAV Helicopter, 2007.
No 1309 **Ola Leifler**: User-Centric Critiquing in Command and Control: The DKExpert and ComPlan Approaches, 2007.
No 1312 **Henrik Svensson**: Embodied simulation as off-line representation, 2007.
No 1313 **Zhiyuan He:** System-on-Chip Test Scheduling with Defect-Probability and Temperature Considerations, 2007.
No 1317 **Jonas Elmqvist:** Components, Safety Interfaces and Compositional Analysis, 2007.
No 1320 **Håkan Sundblad:** Question Classification in Question Answering Systems, 2007.
No 1323 **Magnus Lundqvist**: Information Demand and Use: Improving Information Flow within Small-scale Business Contexts, 2007.
No 1329 **Martin Magnusson:** Deductive Planning and Composite Actions in Temporal Action Logic, 2007.
No 1331 **Mikael Asplund:** Restoring Consistency after Network Partitions, 2007.
No 1332 **Martin Fransson:** Towards Individualized Drug Dosage - General Methods and Case Studies, 2007.
No 1333 **Karin Camara:** A Visual Query Language Served by a Multi-sensor Environment, 2007.
No 1337 **David Broman:** Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments, 2007.
No 1339 **Mikhail Chalabine:** Invasive Interactive Parallelization, 2007.
No 1351 **Susanna Nilsson:** A Holistic Approach to Usability Evaluations of Mixed Reality Systems, 2008.
No 1353 **Shanai Ardi:** A Model and Implementation of a Security Plug-in for the Software Life Cycle, 2008.
No 1356 **Erik Kuiper:** Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008.
No 1359 **Jana Rambusch**: Situated Play, 2008.
No 1361 **Martin Karresand:** Completing the Picture - Fragments and Back Again, 2008.
No 1363 **Per Nyblom:** Dynamic Abstraction for Interleaved Task Planning and Execution, 2008.
No 1371 **Fredrik Lantz:** Terrain Object Recognition and Context Fusion for Decision Support, 2008.
No 1373 **Martin Östlund:** Assistance Plus: 3D-mediated Advice-giving on Pharmaceutical Products, 2008.
No 1381 **Håkan Lundvall:** Automatic Parallelization using Pipelining for Equation-Based Simulation Languages, 2008.
No 1386 **Mirko Thorstensson:** Using Observers for Model Based Data Collection in Distributed Tactical Operations, 2008.
No 1387 **Bahlol Rahimi:** Implementation of Health Information Systems, 2008.
No 1392 **Maria Holmqvist:** Word Alignment by Re-using Parallel Phrases, 2008.
No 1393 **Mattias Eriksson**: Integrated Software Pipelining, 2009.
No 1401 **Annika Öhgren:** Towards an Ontology Development Methodology for Small and Medium-sized Enterprises, 2009.
No 1410 **Rickard Holsmark:** Deadlock Free Routing in Mesh Networks on Chip with Regions, 2009.
No 1421 **Sara Stymne:** Compound Processing for Phrase-Based Statistical Machine Translation, 2009.
No 1427 **Tommy Ellqvist**: Supporting Scientific Collaboration through Workflows and Provenance, 2009.
No 1450 **Fabian Segelström:** Visualisations in Service Design, 2010.
No 1459 **Min Bao**: System Level Techniques for Temperature-Aware Energy Optimization, 2010.
No 1466 **Mohammad Saifullah**: Exploring Biologically Inspired Interactive Networks for Object Recognition, 2011

No 1468 **Qiang Liu**: Dealing with Missing Mappings and Structure in a Network of Ontologies, 2011.
No 1469 **Ruxandra Pop**: Mapping Concurrent Applications to Multiprocessor Systems with Multithreaded Processors and Network on Chip-Based Interconnections, 2011.
No 1476 **Per-Magnus Olsson**: Positioning Algorithms for Surveillance Using Unmanned Aerial Vehicles, 2011.
No 1481 **Anna Vapen**: Contributions to Web Authentication for Untrusted Computers, 2011.
No 1485 **Loove Broms:** Sustainable Interactions: Studies in the Design of Energy Awareness Artefacts, 2011.
FiF-a 101 **Johan Blomkvist:** Conceptualising Prototypes in Service Design, 2011.
No 1490 **Håkan Warnquist:** Computer-Assisted Troubleshooting for Efficient Off-board Diagnosis, 2011.
No 1503 **Jakob Rosén:** Predictable Real-Time Applications on Multiprocessor Systems-on-Chip, 2011.
No 1504 **Usman Dastgeer:** Skeleton Programming for Heterogeneous GPU-based Systems, 2011.
No 1506 **David Landén:** Complex Task Allocation for Delegation: From Theory to Practice, 2011.
No 1507 **Kristian Stavåker**: Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units, 2011.
No 1509 **Mariusz Wzorek:** Selected Aspects of Navigation and Path Planning in Unmanned Aircraft Systems, 2011.
No 1510 **Piotr Rudol:** Increasing Autonomy of Unmanned Aircraft Systems Through the Use of Imaging Sensors, 2011.
No 1513 **Anders Carstensen:** The Evolution of the Connector View Concept: Enterprise Models for Interoperability Solutions in the Extended Enterprise, 2011.
No 1523 **Jody Foo:** Computational Terminology: Exploring Bilingual and Monolingual Term Extraction, 2012.
No 1550 **Anders Fröberg:** Models and Tools for Distributed User Interface Development, 2012.
No 1558 **Dimitar Nikolov:** Optimizing Fault Tolerance for Real-Time Systems, 2012.
No 1582 **Dennis Andersson:** Mission Experience: How to Model and Capture it to Enable Vicarious Learning, 2013.
No 1586 **Massimiliano Raciti:** Anomaly Detection and its Adaptation: Studies on Cyber-physical Systems, 2013.
No 1588 **Banafsheh Khademhosseinieh:** Towards an Approach for Efficiency Evaluation of Enterprise Modeling Methods, 2013.
No 1589 **Amy Rankin:** Resilience in High Risk Work: Analysing Adaptive Performance, 2013.
No 1592 **Martin Sjölund:** Tools for Understanding, Debugging, and Simulation Performance Improvement of Equation-Based Models, 2013.
No 1606 **Karl Hammar:** Towards an Ontology Design Pattern Quality Model, 2013.
No 1624 **Maria Vasilevskaya:** Designing Security-enhanced Embedded Systems: Bridging Two Islands of Expertise, 2013.
No 1627 **Ekhiotz Vergara:** Exploiting Energy Awareness in Mobile Communication, 2013.
No 1644 **Valentina Ivanova:** Integration of Ontology Alignment and Ontology Debugging for Taxonomy Networks, 2014.
No 1647 **Dag Sonntag:** A Study of Chain Graph Interpretations, 2014.
No 1657 **Kiril Kiryazov:** Grounding Emotion Appraisal in Autonomous Humanoids, 2014.
No 1683 **Zlatan Dragisic:** Completing the Is-a Structure in Description Logics Ontologies, 2014.
No 1688 **Erik Hansson:** Code Generation and Global Optimization Techniques for a Reconfigurable PRAM-NUMA Multicore Architecture, 2014.
No 1715 **Nicolas Melot:** Energy-Efficient Computing over Streams with Massively Parallel Architectures, 2015.
No 1716 **Mahder Gebremedhin:** Automatic and Explicit Parallelization Approaches for Mathematical Simulation Models, 2015.
No 1722 **Mikael Nilsson:** Efficient Temporal Reasoning with Uncertainty, 2015.
No 1732 **Vladislavs Jahundovics:** Automatic Verification of Parameterized Sytems by Over-Approximation, 2015.
FiF 118 **Camilla Kirkegaard:** Adding Challenge to a Teachable Agent in a Virtual Learning Environment, 2016.
No 1758 **Vengatanathan Krishnamoorthi:** Efficient and Scalable Content Delivery of Linear and Interactive Branched Videos, 2016.
No 1771 **Andreas Löfwenmark:** Timing Predictability in Future Multi-Core Avionics Systems, 2017.
No 1777 **Anders Andersson:** Extensions for Distributed Moving Base Driving Simulators, 2017.
No 1780 **Olov Andersson:** Methods for Scalable and Safe Robot Learning, 2017.
No 1782 **Robin Keskisärkkä**: Towards Semantically Enabled Complex Event Processing, 2017.
No 1783 **Daniel de Leng**: Spatio-Temporal Stream Reasoning with Adaptive State Stream Generation, 2017.
No 1827 **Johan Falkenjack**: Towards a Model of General Text Complexity for Swedish, 2018.
No 1836 **Magdalena Granåsen**: Exploring C2 Capability and Effectiveness in Challenging Environments: Interorganizational Crisis Management, Military Operations and Cyber Defence, 2019.
No 1848 **Alachew Mengist**: Methods and Tools for Efficient Model-Based Development of Cyber-Physical Systems with Emphasis on Model and Tool Integration, 2019.
No 1871 **Klervie Toczé**: Latency-aware Resource Management at the Edge, 2020.
No 1881 **Chih-Yuan Lin**: A Timing Approach to Network-based Anomaly Detection for SCADA Systems, 2020.
No 1886 **August Ernstsson**: Designing a Modern Skeleton Programming Framework for Parallel and Heterogeneous Systems, 2020.

FACULTY OF SCIENCE AND ENGINEERING

Linköping Studies in Science and Technology. Licentiate Thesis No. 1886, 2020
Department of Computer and Information Science

Linköping University
SE-581 83 Linköping, Sweden

www.liu.se