Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

SkePU 2: Language Embedding and Compiler Support for Flexible and Type-Safe Skeleton Programming

by

August Ernstsson

LIU-IDA/LITH-EX-A--16/026--SE

June 16, 2016

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköpings universitet
Institutionen för datavetenskap

Final thesis

SkePU 2: Language Embedding and Compiler Support for Flexible and Type-Safe Skeleton Programming

by

August Ernstsson

LIU-IDA/LITH-EX-A--16/026--SE

June 16, 2016

Supervisor: Lu Li

Examiner: Christoph Kessler

Abstract

This thesis presents SkePU 2, the next generation of the SkePU C++ framework for programming of heterogeneous parallel systems using the skeleton programming concept. SkePU 2 is presented after a thorough study of the state of parallel programming models, frameworks and tools, including other skeleton programming systems. The advancements in SkePU 2 include a modern C++11 foundation, a native syntax for skeleton parameterization with user functions, and an entirely new source-to-source translator based on Clang compiler front-end libraries.

SkePU 2 extends the functionality of SkePU 1 by embracing metaprogramming techniques and C++11 features, such as variadic templates and lambda expressions. The results are improved programmability and performance in many situations, as shown in both a usability survey and performance evaluations on high-performance computing hardware. SkePU's skeleton programming model is also extended with a new construct, *Call*, unique in the sense that it does not impose any predefined skeleton structure and can encapsulate arbitrary user-defined multi-backend computations.

We conclude that SkePU 2 is a promising new direction for the SkePU project, and a solid basis for future work, for example in performance optimization.

Acknowledgements

I am grateful for the guidance and support from my supervisor, Lu Li, as well as co-supervisor and examiner Christoph Kessler. NSC[1], the Swedish national supercomputing centre, has provided valuable resources and support for this project, which I greatly appreciate. I would also like to thank everyone who has worked on the SkePU project before me, thereby providing a foundation for this thesis. Finally, I am thankful to my family and friends for their support and encouragement during my engineering studies.

August Ernstsson
Linköping, June 2016

[1] `https://www.nsc.liu.se`

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

This chapter provides an introduction to the thesis project, starting with the motivation in Section 1.1. In Section 1.2 the aim of the project is described, followed by explicit research questions in Section 1.3. Finally, delimitations are considered in Section 1.4.

Additionally, an overview of the structure of the thesis report is given in Section 1.5.

1.1 Motivation

The well-known *Moore's law*—formulated in the 1960s—states that the number of transistors per area in microprocessors approximately doubles every two years. The law still holds today, and likely for some time to come[1]. However, transistor count is not the only factor affecting performance. During the twentieth century, the power density in microprocessors remained constant (this is known as *Dennard scaling* [22]). Starting in the early 2000s, the power density is now increasing because of static power losses, preventing ever higher clock frequencies. Computer engineering hit the *power wall*.

During the era of Dennard scaling, Moore's law could just as well be interpreted as a doubling of *performance* every two years [2]; today new ideas are required for continued performance increase. Initially, these were processor architecture improvements—such as pipelined and superscalar cores—with little-to-no impact on software. The efficiency of such low-level features rely on the presence of *instruction-level parallelism* (ILP) in sequential machine code. Inevitably, computer engineering struck the *ILP wall* as well.

[1] An argument can be made that Moore's law is a self-fulfilling prediction, as it has been used by the industry for road-mapping future technology development. There are signs that the industry is moving away from this practice [70], substituting *scaling* for *functional diversification* as the road-map target. Whether this means the end of Moore's law is far outside the scope of this thesis.

Multi-core CPUs, GPUs, and other accelerators additionally exploit *thread-level parallelism*. The downside is that parallel and heterogeneous architectures are significantly more difficult to program for, and good automated compilation techniques are few and difficult to construct. Also, even if computations can be easily parallelized, memory latency has not improved at a pace comparable to compute performance. This is a third wall: the *memory wall*.

Due to the power- and ILP walls, the adoption of parallel architectures is a requirement for continuing the increase of computing performance. This applies to systems of all sizes: from HPC super-clusters to embedded systems. The construction of programming environments allowing efficient programming of parallel and heterogeneous computer architectures—without requiring expert programmers—is thus one of the most important research areas in computer science and engineering today.

1.2 Aim

The aim of this thesis was to design and implement a new interface for SkePU [38], a research project based on the concept of skeleton programming implemented as a C++ header library. The SkePU project is about five years old and has developed into a powerful tool showing promising performance gains. Specifically, this project aimed to improve SkePU in terms of *programmability*. A thorough overview of the SkePU project is found in Chapter 3.

A drawback of SkePU compared with similar research tools is that the implementation is heavily based on C macros. This design is not particularly flexible, hindering further development of the tool. It is also unnecessarily difficult for programmers to use because of the lack of type safety.

The improvements to SkePU realized in this thesis project makes programming with SkePU, as well as adding new backends, easier; it also opens up new optimization opportunities for existing target architectures. New language constructs, implemented with a source-to-source precompiler, and template metaprogramming are the basic implementation tools for the new interface.

1.3 Research Questions

1. **Language embedding**
   What are the common approaches to programming-language extension design (e.g., DSEL specification) in contemporary research projects? What are the advantages and disadvantages of the respective approaches?

2. **Type-safe skeleton programming**

How can a modern C++ interface for skeleton programming be designed while retaining type safety?

3. **Source-to-source precompiling**
   Can source-to-source precompiling be applied to a skeleton programming tool, for example to allow for additional target-specific optimization? What tools are best used for an implementation of this? How would such an implementation look like?

1.4 Delimitations

As mentioned in Section 1.1, efficient execution of arbitrary programs on parallel architectures is very difficult to achieve. This project covers only specific cases, while requiring any existing programs to be rewritten targeting the SkePU tool central to the thesis project. Thus, the results presented in this report are not applicable to all forms of parallel programming.

Designing an interface which is both clean, easy to understand and to use—while still allowing enough information to be specified as to be able to perform target-specific optimizations—requires deep understanding in a variety of areas such as computer architectures and compiler construction. It is also necessary to be proficient with many different languages, tools and frameworks already attempting to solve similar problems. Time is a limited resource, so this ideal position will not be achievable. Only a subset of the most common tools will be investigated in some detail and some more are considered only shallowly.

There are also difficulties in assessing the qualities of different ideas for the new interface. Ideally, a variety of proposed solutions should be tested by professionals over a long period of time to evaluate productivity, performance, code quality, etc. Since the calendar time of this project is limited, these evaluations are not possible.

Work on adding other new features to SkePU progressed concurrently with this project. There are also other ideas for the future development of the framework. The improvements made to SkePU in this project cannot be expected to integrate seamlessly with these still hypothetical features, but they should at least be considered when designing both interface and implementation.

1.5 Report Structure

The thesis report begins with an introduction, in Chapter 1, to the problem from a general perspective and to the aim of the project. Chapter 2 provides a background, describing parallel and heterogeneous architectures, programming frameworks, compiler technology and more. An introduction to the SkePU project is given in Chapter 3, where its history and structure are described. Chapter 4 presents related work by giving short overviews

of a large number of tools, either similar to SkePU or otherwise interesting for the thesis. Chapter 5 covers the methodology used in implementation and evaluation during this project. The SkePU 2 framework and source-to-source translator is introduced in Chapter 6, interface, and Chapter 7, implementation. Results and discussion are presented in Chapter 8, and finally conclusions in Chapter 9.

Chapter 2

Background

This chapter presents the theoretical basis for this thesis project. Section 2.1 covers modern computer architectures from a theoretical, model-based perspective, introducing concepts such as the *PRAM model*, while relating the models to practical systems. Established industry standards for parallel programming are given brief introductions in Section 2.2. The problems with developing performance-portable programs targeting parallel computers are also considered, before the concept of *algorithmic skeletons* is presented as a possible mitigation in Section 2.3. Section 2.4 changes focus to C++ and the recent improvements to the language. Generative programming is covered in Section 2.5. The concept of *metaprogramming* is introduced and its approaches in C++ are covered: template and preprocessor metaprogramming, in Section 2.6 and Section 2.7 respectively. Concluding the chapter, *source-to-source transformation* is presented as an alternative to metaprogramming in Section 2.8.

2.1 Programming of Parallel Systems

Parallel computer architectures come in many variants. The range of designs from the relatively simple (but still very advanced) multi-core processors common in today's personal computers, to clusters of such processors forming large supercomputers are but one dimension. Massively multi-core systems such as modern, programmable GPUs (*graphics processing units*) and specialized on-chip solutions such as the heterogeneous Cell processor [9] presents other challenges.

A popular classification system for parallel architectures is Flynn's taxonomy [29], containing four categories:

- **SISD**: Single instruction-stream, single data-stream,
- **SIMD**: Single instruction-stream, multiple data-stream,

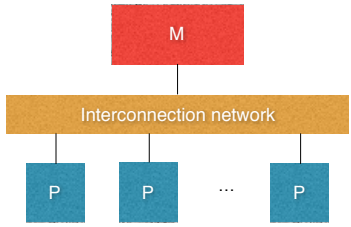


Figure 2.1: A typical shared memory architecture.

- **MISD**: Multiple instruction-stream, single data-stream, and
- **MIMD**: Multiple instruction-stream, multiple data-stream.

Most architectures of interest in this thesis are MIMD architectures; SIMD is also relevant to some extent. SISD architectures can be parallel (e.g., superscalar processors) but this parallelism is hidden from the programmer. MISD architectures are not as common, although pipelined processors and systolic arrays may be regarded as MISD architectures [30].

An attribute of most—if not all—parallel architectures is that computing resources are available in abundance. It is not (anymore) the limiting factor for extracting performance of these systems in typical use cases. Instead, the bottlenecks are communication and synchronization using the available interconnection networks and memory subsystems.

2.1.1 Shared Memory

In a shared memory programming model, all processors share a common memory address space. The physical memory may or may not be shared, depending on implementation; a system with shared physical memory is said to have *uniform memory access* (UMA). A shared memory interface realized on top of a distributed memory architecture is called *non-uniform memory access* (NUMA), as some parts of the address space is local and fast while some parts are remote and slow. Figure 2.1 illustrates a simplified shared memory organization.

A theoretical model for shared memory architectures is the idealized PRAM (*parallel random access memory*) machine, a generalization of the sequential RAM machine. It consists of several processors with access to a common shared memory. The processors operate synchronously, executing one operation each per time step. It is an attractive model to use as the basis for evaluating algorithm performance, since an algorithm which does not perform well on a PRAM machine will not fit into any practical parallel architecture [37]. The model largely ignores issues in communication and synchronization, but for modeling simultaneous memory accesses four forms of PRAM models are used:

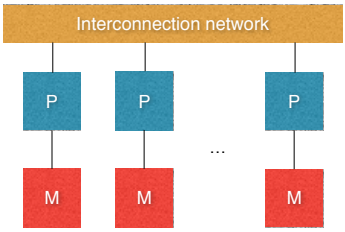


Figure 2.2: A typical message passing architecture.

- **EREW**: Exclusive read, exclusive write,
- **CREW**: Concurrent read, exclusive write,
- **ERCW**: Exclusive read, concurrent write, and
- **CRCW**: Concurrent read, concurrent write.

Shared memory programs use threads for parallel computation. Thread programming often result in low-level, boilerplate code. Also, threading libraries are vendor-specific. A consequence of the shared memory programming is that synchronization must be explicit; absence of synchronization can be difficult to spot and may lead to race conditions.

OpenMP is an extension for writing shared memory programs in C, C++, and Fortran; OpenMP code is both higher level and portable (among shared memory systems).

2.1.2 Message Passing

Models without shared memory, instead using distributed memory, are often referred to as *message passing* models. An important consideration for message passing architectures is what kind of *interconnection network* should be used, as this has major contributions to performance as well as cost.

Most supercomputers are distributed memory architectures with *fat tree* or *switched fabric*-based interconnection networks for inter-node communication. The nodes themselves consist of multiple cores with shared memory.

MPI (Section 2.2.2) is a de-facto standard library specification for programming distributed memory systems.

2.1.3 Heterogeneity and Accelerators

An *heterogeneous* computer architecture uses multiple kinds of processor cores. The cores may differ in size, capabilities, instruction set, memory hierarchy, etc.; the exact meaning of the term is not clearly defined. Heterogeneity is usually visible to the programmer and extra care is necessary to utilize the different cores in an efficient manner.

Accelerator is a term used for co-processors that are optimized for specific tasks and thus not as flexible as CPUs: GPUs, ASICs, FPGAs, etc. An architecture containing one or more accelerators visible to the programmer is thus a heterogeneous architecture. The strength of accelerators lies in large-scale data parallel workloads [45], in contrast to traditional CPUs which are optimized for irregular tasks. Such systems introduce programming challenges: firstly, accelerators need to be programmed in different ways than CPUs, perhaps with different languages and libraries. Secondly, the programmer has to decide which computations to offload to the accelerators and when to do so; for example, if the address spaces are distinct, data transfer time may be the bottleneck for small data sets.

OpenCL (Section 2.2.4) is a framework for developing programs for heterogeneous execution, with support for a wide variety of hardware. More specific accelerator frameworks are also available, such as Nvidia's CUDA (Section 2.2.3) and Microsoft's DirectCompute, both targeting GPUs.

2.1.4 Abstraction Level

When programming parallel systems, the level of abstraction on which to work in must first be decided. Cole [11] suggests that systems can be divided into three rough categories:

1. In the first, the level of abstraction is so high that the user need not be aware of parallelism at all. This leaves the system itself to transform the program to take advantage of the available computational resources.

2. The second category presents a programming model that is close to the physical system implementation, tasking the user to make the choices to make use of the system.

3. A third category covers the middle ground, where parallelism is presented to the user but not at the system level.

An example of the first category are *declarative languages*. Cole discusses the potential advantages for parallelism in these types of systems, stemming from the lack of synchronization and other control structure.

For the third category—where a simplified but still parallel model of the system is presented to the user—there are further considerations, most importantly a communication model. The two established models are *shared memory* and *message passing* [72], covered in Sections 2.1.1 and 2.1.2, respectively.

2.2 Industry Standards

There is no truly universal standard programming model for parallel programming today. Designing such a model would be difficult due to the

variety of parallel and heterogeneous architectures in existence, both today and in the future. Even slight design differences may have important performance implications. We have instead a collection of coexisting *de-facto* standards created and managed by industry consortiums, some directly competing with each other.

In this section five programming models are introduced:

2.2.1 **OpenMP** for shared memory programming,

2.2.2 **MPI** for message passing,

2.2.3 **CUDA** for Nvidia GPUs,

2.2.4 **OpenCL** for GPUs and other accelerators, and

2.2.5 **OpenACC** for high-level accelerator programming.

Of these, none are published by an recognized standards body (such as ISO or IEC), but all except for CUDA are open.

2.2.1 OpenMP

OpenMP (*Open Multi-Processing*) is an open standard for shared memory multiprocessing. It supports programming in C, C++ and Fortran and is built into many high-profile compilers. OpenMP consists of compiler `#pragma` directives (for the C family of languages, as shown in Listing 2.1) and an optional support library. Carefully written OpenMP code can be compiled with any compiler since unknown pragma directives are ignored, generating sequential programs.

Recent versions of OpenMP also include a unified heterogeneous programming model [45].

2.2.2 MPI

MPI (*Message Passing Interface*) is a message-passing library interface specification managed by the MPI Forum [32]. Bindings for C and Fortran are part of the standard. The standard assumes a distributed memory environment and defines a message passing interface; since message passing can be implemented on shared memory systems, the library can be used on such architectures as well.

2.2.3 CUDA

Nvidia's CUDA[1] (*Compute Unified Device Architecture*) is a pioneering, proprietary, de-facto standard for GPGPU computing. CUDA has its roots

[1] `http://www.nvidia.com/cuda`

Listing 2.1: OpenMP basic example.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
  int th_id, nthreads;

#pragma omp parallel private(th_id)
{
    th_id = omp_get_thread_num();
    printf("Hello World from thread %d\n", th_id);

#pragma omp barrier
    if (th_id == 0)
    {
      nthreads = omp_get_num_threads();
      printf("There are %d threads\n", nthreads);
    }
  }

  return EXIT_SUCCESS;
}
```

in the *Brook* project at Stanford [8, 51] in 2003. The author of Brook later joined Nvidia and CUDA was subsequently released in 2006.

CUDA exclusively targets Nvidia GPUs, limiting the portability of programs targeted at the framework. A relatively high-level programming language (closely based on C++) and APIs, as well as high performance has nonetheless resulted in CUDA being used in a variety of projects.

An example CUDA program[2] can be seen in Listing 2.2.

2.2.4 OpenCL

The OpenCL[3] (*Open Computing Language*) framework is a vendor-neutral open standard for heterogeneous computing. OpenCL was originally developed by Apple and is now managed by the Khronos Group consortium. The framework differs from CUDA in its lower-level programming language, which is based on C, and broader range of target platforms; OpenCL drivers are available for CPUs, GPUs, FPGAs, and other accelerators.

OpenCL C is used for writing computation kernels—the functions executed on accelerator devices. While the language itself is similar to C, the standard library is replaced entirely. The host API is defined in C and C++, but non-standard bindings exist for a variety of programming languages.

Due to the low-level nature of OpenCL and establishment of CUDA, OpenCL has had some difficulty of gaining traction in the field. Tools that automatically transform CUDA code into OpenCL have been proposed for

[2] Original idea by Ingemar Ragnemalm, `http://www.computer-graphics.se/`

[3] `https://www.khronos.org/opencl/`

Listing 2.2: CUDA "Hello World!" program.

```
// Based on example by Ingemar Ragnemalm 2010
// http://www.computer-graphics.se/

#include <stdio.h>

__global__  void hello(char *a, char *b)
{
  a[threadIdx.x] += b[threadIdx.x];
}

#define N 7

int main()
{
  char a[N] = "Hello ";
  char b[N] = {15, 10, 6, 0, -11, 1, 0};
  printf("%s", a); // Prints "Hello "

  char *ad, *bd;
  cudaMalloc(&ad, N);
  cudaMalloc(&bd, N);
  cudaMemcpy(ad, a, N, cudaMemcpyHostToDevice);
  cudaMemcpy(bd, b, N, cudaMemcpyHostToDevice);

  dim3 dimBlock(N, 1);
  dim3 dimGrid(1, 1);
  hello<<<dimGrid, dimBlock>>>(ad, bd);

  cudaMemcpy(a, ad, N, cudaMemcpyDeviceToHost);
  cudaFree(ad);
  cudaFree(bd);

  printf("%s\n", a); // Prints "World!"
  return EXIT_SUCCESS;
}
```

this reason [47].

An example program, similar to the CUDA example in Listing 2.2 but implemented with OpenCL, is presented in Listing 2.3.

2.2.5 OpenACC

OpenACC (*Open Accelerators*) is a standard for accelerator programming on heterogeneous systems. Both the goals and the means of OpenACC is similar to those of OpenMP (Section 2.2.1), but OpenACC is much younger (first demonstrated in 2012) and less widely adopted. OpenACC allows programmers to write high-level constructs targeting heterogeneous accelerators. Listing 2.4 shows how pragma directives are used to annotate otherwise sequential code (compare with Listing 2.1). However, early performance evaluation of OpenACC [71] has shown significant slowdown compared to manual OpenCL in some cases.

2.3 Algorithmic Skeletons

As described in Sections 2.2, 2.1.1, and 2.1.2, parallel programming interfaces are diverse and the underlying systems are fundamentally different. It is not possible to write a low-level program that runs on a wide variety of architectures—even if it was, the performance characteristics would vary significantly between different hardware. Clearly, some higher abstraction level is required for writing *performance-portable* programs for parallel computers. Cole [11] notes that such a system should not be *explicitly* parallel to the programmer, but enforce a structure which is efficiently parallelizable by the system.

In 1989, Cole [11] introduced an approach to parallel programming inspired by functional programming. In functional programming, *higher order functions* are functions accepting other functions as arguments, usually to be applied to a sequence of data. Common examples of such functions are *map*, *scan*, and *reduce* (sometimes known as *fold*). The map function accepts a unary function

$$f\colon a \to b$$

which is applied to each element of the sequence. The other variants take binary functions

$$f\colon (a, b) \to c$$

the properties (e.g., associativity and commutativity) of which restricts the kinds of parallel optimization possible. Higher-order functions with properties suitable for parallelization can be used as *skeletons*.

More generally, algorithmic skeletons are pre-defined, parametrizable generic components with well defined semantics [18], for which efficient parallel or accelerator-specific implementations may exist.

Algorithmic skeletons are categorized into two types [17]:

Listing 2.3: OpenCL "Hello World!" program.

```
// Based on example by Ingemar Ragnemalm 2013 (no error checking)
// http://www.computer-graphics.se/

#include <iostream>
#include <cmath>
#include <CL/cl.h>

const char *src =
"__kernel void hello(__global char* a, __global char* b)"
"{"
" a[get_global_id(0)] += b[get_global_id(0)];"
"}";

constexpr size_t N = 7;

int main(int argc, char** argv)
{
  char a[N] = "Hello ";
  char b[N] = {15, 10, 6, 0, -11, 1, 0};
  std::cout << a; // Prints "Hello "

  // Where to run
  int err;
  cl_device_id id;
  unsigned int no_plat;
  cl_platform_id platform;
  clGetPlatformIDs(1, &platform, &no_plat);
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &id, NULL);
  cl_context ctx = clCreateContext(0, 1, &id, NULL, NULL, &err);
  cl_command_queue cmd = clCreateCommandQueue(ctx, id, 0, &err);

  // What to run
  cl_program prog = clCreateProgramWithSource(ctx, 1, &src, NULL, &err);
  clBuildProgram(prog, 0, NULL, NULL, NULL, NULL);
  cl_kernel kernel = clCreateKernel(prog, "hello", &err);

  // Create space for data and copy a and b to device
  cl_mem buf1 = clCreateBuffer(ctx, CL_MEM_USE_HOST_PTR, N, a, NULL);
  cl_mem buf2 = clCreateBuffer(ctx, CL_MEM_USE_HOST_PTR, N, b, NULL);

  // Run kernel
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &buf1);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &buf2);
  clEnqueueNDRangeKernel(cmd, kernel, 1, NULL, &N, &N, 0, NULL, NULL);
  clFinish(cmd);

  // Read result
  clEnqueueReadBuffer(cmd, buf1, CL_TRUE, 0, N, a, 0, NULL, NULL);
  std::cout << a << "\n"; // Prints "World!"

  // Clean up
  clReleaseMemObject(buf1);
  clReleaseMemObject(buf2);
  clReleaseProgram(prog);
  clReleaseKernel(kernel);
  clReleaseCommandQueue(cmd);
  clReleaseContext(ctx);
  return 0;
}
```

Listing 2.4: OpenACC example program [71].

```
// Initialization: |x[i]| < 1, i = 0,...,size-1

#pragma acc data copy(x[0:size]) // Data movement
{
  while(error > eps)
  {
    error = 0.0;
#pragma acc parallel present (x[0:size])
#pragma acc loop gang vector reduction (+: error )
    for (int i = 0; i < size; ++i)
    {
      x[i] *= x[i];
      error += fabs(x[i]);
    }
  }
}
```

- *Data-parallel* skeletons work on large data sets, where some operation is independently applied to multiple small subsets of the data.
- *Task-parallel* skeletons exploit independence between different tasks.

Tools utilizing the concepts of algorithmic skeletons have been successfully applied in both scientific and commercial environments. Some of them are explicitly modeled after Cole's proposal—SkePU itself is—while some have reached the same conclusions by other means, for example Google's *MapReduce* [21]. A selection of algorithmic skeleton frameworks are covered in Chapter 4.

Note: In this thesis, the term *skeleton programming* is used in the meaning "programming with algorithmic skeletons". The term may have different meanings in other contexts.

2.4 Modern C++

C++ is a multi-paradigm, general purpose programming language originally based on C. C++ syntax is similar to C with the additions of, among other things, classes and templates; these features provide support for object oriented programming and generic programming, respectively. C++ is standardized by ISO/IEC [12].

While C and Fortran are still the most common programming languages used in scientific high-performance computing (HPC), C++ provides a higher abstraction level and more expressivity, while retaining most of the performance characteristics[4]—at least compared to almost any other established higher-level language. Anecdotally, C++ is growing in popularity in HPC applications.

[4]High performance C++ may require avoiding or disabling certain features (for example, run-time type information and exceptions). These features are traditionally not needed in HPC applications.

Listing 2.5: constexpr metaprogramming example.

```
// A. Computing factorial
constexpr int factorial(int n) {
  return (n == 0) ? 1 : n * factorial(n-1);
}

// Test program
int main() {
  constexpr int f = factorial(5);
}
```

The term *modern C++* refers to the new standards C++11 and C++14 (eventually also C++17). These revisions overhaul the language by introducing a multitude of new concepts (e.g., *move semantics*), amending the syntax with new constructs (e.g., *range-based* `for`*-loops*, lambda expressions) and extending the C++ standard library with, for example, threads and regular expressions [12].

Modern C++ allows for higher-level programming than before, while also reducing overhead and improving performance in many cases. C++11 and later versions are starting to be used in programming frameworks targeting parallel heterogeneous architectures, specifically systems consisting of both CPUs and GPUs. See for example PACXX [35] and SYCL [53].

2.4.1 `constexpr` Specifier

The `constexpr` specifier declares that the value of an expression is computable at compile-time. It can be used on variables and on functions. For a variable, it forces the value of the variable to be known at compile-time. For a function, it indicates that the function is computable at compile-time if its actual parameters are known at compile-time.

Compile-time computation is useful for metaprogramming techniques, for example partial evaluation. Compared to *template metaprogramming* (Section 2.6), `constexpr` functions are syntactically more similar to dynamic computations. As an example, consider the factorial function in Listing 2.5 and compare with example A in Listing 2.7.

2.4.2 Unified Attribute Syntax

C++11 brings unified and generalized *attributes* to the language. Attributes gives the compiler information about a programming construct (a type, an object, an expression, etc.) otherwise not possible to encode in the grammar. Although C++14 has three built-in attributes (with arguably the most useful being `[[noreturn]]`), they are typically used for specialized compilers or build environments. For example, a parallelizing compiler may understand a specific attribute on a for-loop to mean that the loop iterations are independent and can be executed in parallel.

Before modern C++, each vendor had to use a custom attribute syntax or would risk to interfere with each other. GNU/IBM attributes uses the `__attribute__((...))` and Microsoft uses `__declspec()`. (These non-standard variants remain in use today, even for C++11 code.) The standard syntax has the form `[[<namespace>::<attribute>]]` where the optional namespace part prevents attribute name collisions when used properly [48].

According to the C++ standard, unrecognized attributes should be ignored.

C++ continues to support `#pragma` directives. Most existing parallelization tools, as covered in Section 2.2, today use pragma directives instead of attributes. However, pragma directives are formally a part of the C++ preprocessor (although this is not always true in practice). While they serve a role similar to that of attributes, pragma directives are defined in an entirely different part of the standard. This alone may indicate that C++ attributes are better suited to annotate source code for parallelizable compilers than pragma directives are. Another reason is that attributes are applied to syntactical constructs, as opposed to pragmas which are bound to a line of source code.

2.5 Generative Programming

Generative programming is defined by Czarnecki and Eisenecker [14] as

> *. . . a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.*

Note that, while SkePU's usage of algorithmic skeletons fits the definition of generative programming fairly well, parallel programming is *not* the target application suggested by Czarnecki and Eisenecker; they are mainly interested in *production efficiency* of software systems, which is outside the scope of this thesis. Nonetheless, their methods and tools are applicable in other contexts.

There are many implementation approaches to generative programming. In this thesis, we are only interested in automatic, compiler-driven techniques (in contrast to, e.g., software development methodologies). For C++, this restricts us to three established options: *metaprogramming* using either preprocessor macros (Section 2.7) or templates (Section 2.6), which can be utilized within the standard C++ compiler phases; or *source-to-source translation* as a separate, initial stage in the compiler chain (Section 2.8).

2.6 Template Metaprogramming

"[Template metaprogramming] is closer to Lisp than C++"

—*Walter E. Brown* [7]

Metaprogramming is the process of writing programs (called *metaprograms*) that represent and manipulate other programs [14]. Metaprograms may also be used for *partial evaluation*: performing computations otherwise done at run-time at compile-time [7].

Templates are the C++ implementation of the paradigm *generic programming*. A construct—class, function, or even variable in C++14—is annotated with one or more template arguments (which can be types or integral values). At compile-time, separate template instantiations will be made for each unique combination of arguments used. Templates in C++ have purely static semantics (every argument of a template is known at compile-time); this allows for a Turing-complete, static programming model called *template metaprogramming*. The limitations of templates—namely immutable data and absence of side effects—effectively makes template metaprogramming a *pure functional* metalanguage of C++ [1]. Template metaprogramming is considered a complicated C++ technique and therefore avoided in some software projects [52].

Template metaprogramming is not a new feature of C++; nonetheless, it is a feature relatively far from the C roots and usually considered a modern feature of the language. The technique is an originally unintended ability of C++ [69] and was discovered in 1994 by Unruh [66], who constructed a program printing consecutive prime numbers as compiler errors. The technique was refined in the following years, most notably by Veldhuizen [69, 68]. Due to advancements in modern C++, as well as compiler improvements in general, most of the initial articles on template metaprogramming are today somewhat outdated. In Listing 2.6 and 2.7, example A illustrates improvements in implementation of template metafunctions and example B shows how the usage of such functions have become cleaner. Note that example A can also be implemented statically in C++11 without templates, using recursive `constexpr` functions (as in Listing 2.5).

Metaprogramming goals which can be accomplished with template metaprogramming include code selection, code generation, and partial evaluation. Common criticisms of template metaprogramming include complicated implementations and difficulty of debugging [52].

Libraries built with template metaprogramming are often combined with preprocessor macros to simplify the interface for the user (see, e.g., Skell BE in Section 4.1).

2.6.1 Expression Templates and DSELs

A common idiom in template metaprogramming is *expression templates*. The syntactical structure of C++ expressions are encoded as nested function

Listing 2.6: C++98 template metaprogramming example.

```
// A. Computing factorial
template<int N>
struct Factorial {
  enum  { RET = Factorial<N-1>::RET * N };
};

template<>
struct Factorial<0> { enum  { RET = 1 }; };

// B. Selecting types
template<int, typename T1, typename>
struct Select { typedef T1 TYPE; };

template<typename T1, typename T2>
struct Select<1, T1, T2> { typedef T2 TYPE; };

// Test program
int main() {
  int f = Factorial<5>::RET;
  Select<1, int, float>::TYPE number = 5;
}
```

templates, specifically overloaded operators. The expression structure is thus encoded in the type. Expression templates were first proposed by Veldhuizen [68].

The main advantage of expression templates is an interface that is barely distinguishable from ordinary C++ code. As a result, *domain-specific languages* (DSLs) can be used within C++ code and parsed by any C++ compiler. Such languages are called domain-specific *embedded* languages (DSELs or EDSLs). A typical example of an DSEL-based framework is *Eigen* for vector and matrix calculations [33] (Section 4.10) and *Skell BE* for automatic parallelization [57] (Section 4.1). *Boost.Proto* is a support library for DSEL applications [49].

2.7 Preprocessor Metaprogramming

C++ inherits the *preprocessor* from its ancestor, C. The preprocessor runs as the first phase when compiling a C++ program, and is responsible for performing the following:

- Inclusion of source and header files. (`#include "..."`)
- Macro expansion. (`#define X ...`)
- Conditional compilation. (`#ifdef`, `#if` etc.)

Although the preprocessor thus has both a conditional and a looping construct, there are practical limitations to the use of the preprocessor as a general purpose programming tool. Preprocessor metaprogramming is nonetheless used for basic tasks in most large C or C++ projects.

Listing 2.7: C++14 template metaprogramming example.

```
// A. Computing factorial
template<int N>
struct Factorial_ {
  static constexpr int RET = Factorial_<N-1>::RET * N;
};

template<>
struct Factorial_<0> { static constexpr int RET = 1; };

template<int N>
constexpr int Factorial = Factorial_<N>::RET;

// B. Selecting types
template<int, typename T1, typename>
struct Select_ { using TYPE = T1; };

template<typename T1, typename T2>
struct Select_<1, T1, T2> { using TYPE = T2; };

template<int I, typename T1, typename T2>
using Select = typename Select_<I, T1, T2>::TYPE;

// Test program
int main() {
  int f = Factorial<5>;
  Select<1, int, float> number = 5;
}
```

There are features of the preprocessor which can be useful and difficult to emulate with other means, such as the "stringification" operator `#` which converts a macro argument to an escaped string representation[5].

2.8 Source-to-Source Transformation

Source-to-source compilers (also called *translators* or *open compilers*) perform *source-to-source transformation*: accepting high-level source code as input and generating source code at a similar level as output. This is in contrast to standard compilers which generate output at a lower level than the input, for example assembly or machine code. Source-to-source compilers can produce output in the same language as the input or in an entirely different language, depending on the application.

A popular use of source-to-source compilers is implementing new programming languages. Instead of constructing an entire compiler stack, source code written in the new language can be translated into an existing language. The implementation of the existing language provides the remaining compilation steps. C is a popular target for this use of source-to-source compilers. For example, the first C++ compiler produced C code as output (at this time, C++ was known as *C with Classes*).

[5]This is a crucial tool for OpenCL program generation in SkePU 1.2.

The terms (source-to-source) *translator* and *preprocessor* are often confused. The difference is that the translator is more sophisticated and must understand the syntax of the target language at a level deeper than that of the preprocessor [55].

2.8.1 ROSE

ROSE[6] is a tool for source transformation (source-to-source compilation). It is also a major research project, originally from *Lawrence Livermore National Laboratory* [54], now supported by a large group of contributors. The purpose of ROSE is to allow straightforward implementation of complex compilation techniques in domain-specific research projects. The kinds of tasks which can be implemented with ROSE include transformation, instrumentation, analysis, verification and optimization of source code. It has stable support for C and C++ with active development of additional language front-ends.

All transformations in ROSE are done on its internal *abstract syntax tree* (AST) representation. ROSE first generates the internal representation by parsing the input program. The AST is then modified during multiple passes, and finally unparsed to generate the output program.

2.8.2 Clang

Clang[7] is a compiler front-end for programming languages in the C family, including C++. It it built on top of LLVM[8], a research project conceived by Lattner et al. in 2002 [42, 43], now used in, and supported by, academic and commercial projects. For example, LLVM is also the basis of Nvidia's CUDA compiler (NVCC[9]). LLVM received the prestigious *ACM Software System Award* in 2012 [31]. The LLVM project is young compared to other popular compiler toolchains (GCC was released in 1987 and ROSE was proposed in 1999 [54])

Although the main goals of Clang are fast and efficient compilations and expressive error messages [41], it is designed with a modular, library-based API. This means that it is relatively simple to build standalone tools based on Clang, including source-to-source translators. This is contrasted to ROSE (Section 2.8.1): ROSE is explicitly designed to be a translator generator. Possible drawbacks of Clang are its relative immaturity and unstable C++ API.

One example research project using Clang as a source-to-source translator tool is CU2CL [47], a tool performing automatic source code transformation from CUDA to OpenCL. The authors cite the modular design

[6] `http://www.rosecompiler.org/`
[7] `http://clang.llvm.org`
[8] `http://llvm.org`
[9] `https://developer.nvidia.com/cuda-llvm-compiler`

of Clang as one reason for the choice, and classifies their translation approach as AST-driven and string-based. Their choice of Clang is noted for its usefulness in situations when the source language is similar to the target language; as only a small part of the source needs translation, the rest of the structure (e.g., comments) of the original source file are retained. After their work was published in 2011, higher-level interfaces for tool development has been added to Clang and more and more projects are being built on its libraries. See for example *gpucc* [73] and *PACXX* [35].

Clang's approach to source-to-source transformation differs from that of ROSE. Where in ROSE the AST is modified and then unparsed to generate output, Clang instead uses the AST as a read-only structure to guide the translation. Modifications are done though string operations, i.e., insertions and removals, potentially retaining more of the input's original structure.

Chapter 3

SkePU

SkePU (*Skeleton Processing Unit*) is an open source skeleton programming (see Section 2.3) framework, started at Linköping University in 2010 by Johan Enmyren et al. [23, 24]. It is a C++ template header library, enabling higher-level skeleton programming for various multi-core and heterogeneous parallel architectures. SkePU was modeled after *BlockLib* [74], a similar project targeting the *IBM Cell processor* [9] from the same research group. SkePU has been part of several international (i.e., EU FP7) research projects and is publicly available as an open source project[1].

The advantages of using SkePU instead of a lower-level interface can be summarized with three concepts:

- *programmability*, as SkePU code is more high-level than code targeting the backends directly;
- *portability* from the existence of backends for a diverse array of target hardware;
- *performance*, as the backends are optimized by domain experts with deeper knowledge of the target architectures than most SkePU users.

SkePU has been extended over time with many different features; for example new backends, auto-tuning and smart container [19] types. There is also support for hybrid execution using integrated StarPU [3] support. As of today, there are backends for sequential C++, OpenMP, OpenCL and CUDA with single or multiple GPUs. There are also experimental backends for, e.g., MPI.

In SkePU, as is customary in this context, the CPU is referred to as the *host* and GPUs and other accelerators are called *devices*.

[1] `http://www.ida.liu.se/labs/pelab/skepu/`

3.1 Smart Containers

SkePU includes three container types: 1D `Vector`, 2D (dense) `Matrix`, and `SparseMatrix`. They are all implemented as class templates, with the contained type as template parameter.

All containers are "smart" in the sense that copying between host and device address spaces are automatically optimized. An MSI-like sequentially consistent implementation ensures that no manual memory management is needed [19].

3.1.1 Vector

The `Vector` class is modeled after `std::vector` and is largely compatible with it. Data is stored in contiguous memory.

3.1.2 Matrix

`skepu::Matrix` is a row-major, contiguous matrix class with an interface similar to that of `Vector`.

3.1.3 Sparse Matrix

A sparse matrix is a matrix in which most elements are zero. `SparseMatrix` is a sparse matrix implementation using the CSR format, storing arrays of elements and their respective indices.

3.1.4 Multi-Vector

`MultiVector` is a wrapper class for allowing any number of vectors to be passed as arguments to user functions. This is only implemented for the `MapArray` skeleton.

3.2 Skeletons

SkePU provides a number of skeletons: `Map`, `Reduce`, `MapReduce`, `MapArray`, `MapOverlap`, `Generate`, `Scan`, and the special `Farm`. A summary of each of the skeletons are provided here; detailed explanation of the different skeletons, what types of user functions can be combined, and which backends are supported can be found in the SkePU User Guide [59].

3.2.1 Map

Accepting k containers of the same type and size, `Map` applies a k-ary function to the k items which share an index, for all indices in the containers. A single container of matching size is returned. There are also variants of each arity which allow for an additional constant argument to be passed along.

3.2.2 Reduce

Accepting a container and returning a scalar, the result of a `Reduce` operation is equivalent to applying a binary commutative and associative operator to each element in the vector and the result. In practice, the operation is efficiently implemented as a binary *reduction tree*.

3.2.3 MapReduce

`MapReduce` combines `Map` and `Reduce` in an efficient manner.

3.2.4 Scan

The `Scan` skeleton is similar to `Reduce`, but returns a container with each partial result. The scan can be either inclusive or exclusive.

3.2.5 MapOverlap and MapOverlap2D

`MapOverlap` accepts one container as input and applies a k-ary operator to k neighboring elements, for each index in the container. The edge handling can be set as either cyclic or constant. For a thorough explanation, see Dastgeer's licentiate thesis [17].

3.2.6 MapArray

`MapArray` behaves similarly to unary `Map`, with the addition of an auxiliary Vector argument which can be accessed in its entirety.

3.2.7 Generate

The `Generate` skeleton accepts an optional constant scalar input and returns a container. Each element is calculated from its index and the constant.

3.2.8 Farm

`Farm` is a task-parallel skeleton using the StarPU run-time, only available in a special version of SkePU.

3.3 User Functions

SkePU uses C preprocessor macros for defining user functions, as exemplified in Listing 3.1.

The list of available user function macros has grown over time and is now quite long:

- `UNARY_FUNC`
  `UNARY_FUNC_CONSTANT`

Listing 3.1: Dot product example from the public SkePU distribution.

```
// following define to enable/disable OpenMP implmentation to be used
/* #define SKEPU_OPENMP */

// following define to enable/disable OpenCL implmentation to be used
/* #define SKEPU_OPENCL */

// With OpenCL, following define to specify number of GPUs to be used.
// Specifying 0 means all available GPUs. Default is 1 GPU.
/* #define SKEPU_NUMGPU 0 */

#include <iostream>

#include "skepu/vector.h"
#include "skepu/mapreduce.h"

// User-function used for mapping
BINARY_FUNC(mult_f, float, a, b,
  return a*b;
)

// User-function used for reduction
BINARY_FUNC(plus_f, float, a, b,
  return a+b;
)

int main()
{
  skepu::MapReduce<mult_f, plus_f> dotProduct(new mult_f, new plus_f);

  skepu::Vector<float> v0(20, (float)2);
  skepu::Vector<float> v1(20, (float)5);

  float r = dotProduct(v0, v1);

  return 0;
}
```

Listing 3.2: Specifying an execution plan in SkePU.

```
skepu::Reduce<plus> globalSum(new plus);
skepu::Vector<double> input(100, 10);
skepu::ExecPlan plan;
plan.add(1, 5000, CPU_BACKEND);
plan.add(5001, 1000000, OMP_BACKEND, 8);
plan.add(1000001, INFINITY, CL_BACKEND, 65535, 512);
globalSum.setExecPlan(plan);
```

- BINARY_FUNC
  BINARY_FUNC_CONSTANT
- TERNARY_FUNC
  TERNARY_FUNC_CONSTANT
- ARRAY_FUNC
  ARRAY_FUNC_CONSTANT
  ARRAY_FUNC_MATR
  ARRAY_FUNC_MATR_CONSTANT
  ARRAY_FUNC_MATR_BLOCK_WISE
  ARRAY_FUNC_SPARSE_MATR_BLOCK_WISE
- VAR_FUNC
- OVERLAP_DEF_FUNC
  OVERLAP_FUNC
  OVERLAP_FUNC_STR
  OVERLAP_FUNC_2D_STR
- GENERATE_FUNC
  GENERATE_FUNC_MATRIX

3.4 Execution Plans and Auto-Tuning

To ensure the most efficient execution possible, the programmer can supply the SkePU runtime system with an execution plan, declaring which backends are to be used for various input sizes [18]. An example of an execution plan specification can be seen in Listing 3.2. If no plan is explicitly defined, SkePU constructs one from default parameters.

A framework for auto-tuning SkePU based on a heuristic optimization algorithm has been proposed [18]. The algorithm first generates an optimal plan for each backend, then generates an overall plan considering all available backends.

3.5 Implementation

The SkePU implementation is largely implemented with preprocessor metaprogramming. User functions are specified with C macros (Listing 3.1) and expanded to C++ structs at compile-time. These structs have member functions, each corresponding to a particular backend, which are called by skeleton instances; conditional compilation controls which of these are generated and called.

The design incurs limitations in the signatures of user functions and weak type safety.

3.6 Criticism

SkePU has been used to parallelize several large industry programs, the results of which includes suggestions for improvements to the framework, specifically on the topic of user-function definitions. In both the thesis project by Sundin [65]—parallelization of a sonar simulation—and the thesis project by Sjöström [58]—translating a flow solver to C++ and SkePU—the authors had difficulty with locating computations which fit into skeleton structures. Only the most general skeleton, MapArray, was used and both authors were required to add new user-function macros to SkePU. Sjöström in particular needed access to multiple auxiliary data structures and was required to construct the MultiVector container [60], losing the benefits of the existing smart containers in the process. Sjöström also commented on the weak type-safety in SkePU.

Chapter 4

Related Work

This chapter presents a selection of research projects with relevance to either the problem domain of this thesis (skeleton programming), or the proposed methods for solving the task at hand (compiler technology and generative programming). There is a lot of research performed in the fields of parallel computing and compiler technology, and many tools and frameworks has been proposed as a result. Some, such as CU2CL, targets a specific niche; while others, for example SkePU itself, aims to be a more general solution.

We first present an overview of the covered topics, starting with algorithmic skeleton frameworks and libraries:

4.1 **Skell BE**: Skeleton framework targeting the Cell BE architecture.

4.2 **SkelCL**: OpenCL skeleton programming library.

4.3 **Thrust**: Template algorithm library for CUDA.

4.4 **Muesli**: Skeleton programming of multi-node cluster computers.

4.5 **Marrow**: Data and task-parallel skeletons for OpenCL systems.

4.6 **Bones**: Algorithmic skeletons in Ruby.

Two task-based parallel programming solutions are also presented:

4.7 **StarPU**: Task programming library for hybrid CPU/GPU architectures.

4.8 **Cilk**: Multi-threaded programming language (a superset of C).

Template metaprogramming is an established C++ technique, and as such there are many frameworks and libraries of various sizes based on it. As template metaprogramming is a suggestion for implementation basis, two typical libraries built using template metaprogramming techniques have been investigated:

4.9 **Boost**: General-purpose C++ libraries.

4.10 **Eigen**: High-performance linear algebra library.

Five tools using Clang as a source-to-source programming library are also introduced in this chapter:

4.11 **CU2CL**: Automatic CUDA to OpenCL conversion.

4.12 **Scout**: Semi-automatic loop vectorization.

4.13 **Clad**: Compile-time automatic differentiation.

4.14 **gpucc**: An optimizing GPGPU compiler.

4.15 **PACXX**: A unified programming model for accelerators using C++14.

We also cover an example usage of C++11 attributes:

4.17 **REPARA**: Transforming applications for parallel and heterogeneous architectures.

Finally, a proposed parallelism extension for C++:

4.18 **C++ Extensions for Parallelism**: Parallel algorithm overloads in the C++ STL.

4.1 Skell BE

Saidani et al. proposed the Skell BE library in a 2009 paper [57]. Skell BE is an algorithmic skeleton library targeting the Cell BE architecture [9]. The library is implemented with a generative programming approach using template metaprogramming to create a DSEL on top of C++. It is possible to define process networks by piping data between the built-in skeletons at compile-time. See Listing 4.1 for a brief example.

Code generated by Skell BE has been shown to be faster than other C++-based libraries [57]. The authors suggest that the meta-programming approach is responsible for this performance gain. More computation is done at compile-time, and more statically defined types can open up new optimization opportunities for the compiler.

4.2 SkelCL

SkelCL (Skeleton Computing Language) [64, 63] is a skeleton programming library similar to SkePU, but focused on OpenCL. Like SkePU, SkelCL is a well-documented, open-source C++ library used as a basis for research on high-level programming of parallel heterogeneous systems.

SkelCL is structured around three desirable requirements of a high-level parallel programming model:

Listing 4.1: A complete Skell BE example [57].

```
#include <skell.hpp>

void sqr()
{
  float in[32], out[32];

  pull(arg0_, in);

  for(int i = 0; i < 32; ++i)
    out[i] = in[i] * in[i];

  push(arg1_, out);

  terminate();
}

SKELL_KERNEL(sample, (2,(float const*, float*)))
{
  run(pardo<8>(seq(sqr));
}

int main(int argc, char** argv)
{
  float in[256], out[256];
  skell::environment(argc, argv);
  sample(in, out);
  return 0;
}
```

Listing 4.2: Dot product in SkelCL [61].

```
#include <SkelCL/SkelCL.h>
#include <SkelCL/Zip.h>
#include <SkelCL/Reduce.h>
#include <SkelCL/Vector.h>

using namespace skelcl;

int main()
{
  skelcl::init(); // initialize SkelCL

  // specify calculations using parallel patterns (skeletons):
  Zip<int(int,int)> mult("int func(int x, int y){ return x*y; }");
  Reduce<int(int)>   sum("int func(int x, int y){ return x+y; }", "0");

  // create and fill vectors
  Vector<int> A(1024);
  Vector<int> B(1024);
  init(A.begin(), A.end());
  init(B.begin(), B.end());

  Vector<int> C = sum( mult(A, B) ); // perform calculation in parallel

  std::cout << "Dot product: " << C.front() << std::endl; // access result
}
```

- parallel data types,
- data distribution and redistribution, and
- recurring parallelizable patterns.

SkelCL has, in general, a more limited feature set compared to SkePU 2, but includes features which are not in SkePU such as the AllPairs skeleton [62], an efficient implementation of certain complex access modes involving multiple matrices. In SkePU 2 matrices are accessed either element-wise or randomly.

A possible downside to SkelCL is that the customizable computations performed by a skeleton (comparable with SkePU's "user functions") are specified with string literals and thus not subject to syntactic or semantic checking until run-time. This can be seen in the example in Listing 4.2.

4.3 Thrust

Nvidia *Thrust*[1] [4] is a C++ template library with parallel CUDA implementations of common algorithms. It uses common C++ STL idioms, and defines operators (equivalent to SkePU 2 user functions) as native functors. The fundamentals of the implementation are in effect similar to SkePU 2, as the CUDA compiler takes an equivalent role to the source-to-source compiler presented in this thesis. (In practice, Thrust is limited to Nvidia GPUs and does not include SkePU features such as smart containers and tuning).

4.4 Muesli

The *Muesli*[2] skeleton library [10] is targeted at multi-core cluster computers using MPI and OpenMP execution, and has been ported for GPU execution [25]. It currently contains a limited set of data-parallel skeletons.

4.5 Marrow

Marrow is a flexible skeleton programming framework for single-GPU OpenCL systems [46]. It provides both data and task parallel skeletons with the ability to compose skeletons for complex computations. Marrow aims to avoid the problems of data movement overhead by targeting algorithms and computations based on persistent data schemes, and also by overlapping data movement with computation.

[1] `https://developer.nvidia.com/thrust`
[2] `https://www.wi.uni-muenster.de/research/projects/9220`

Listing 4.3: Convolution in Bones [50].

```
int N = 512 * 512;
#pragma kernel N|neighb(3) -> N|element
for (i=1; i<N-1; i=i+1)
  B[i] = 3 * A[i-1] + 4*A[i] + 3*A[i+1];
#pragma endkernel conv
```

Listing 4.4: StarPU basic example.

```
#include <stdio.h>

static void my_task (int x) __attribute__ ((task));
static void my_task (int x)
{
  printf ("Hello, world! With x = %d\n", x);
}

int main ()
{
#pragma starpu initialize
  my_task (42);
#pragma starpu wait
#pragma starpu shutdown
  return 0;
}
```

4.6 Bones

Bones is a source-to-source compiler based on algorithmic skeletons [50]. It transforms `#pragma`-annotated C code to parallel CUDA or OpenCL using a translator written in Ruby, based on the existing C parser *CAST*[3]. The skeleton set is based on a well-defined grammar and vocabulary. An example of a convolution operation specified in Bones syntax is available in Listing 4.3. Bones places strict limitations on the coding style of input programs.

4.7 StarPU

StarPU[4] is a programming library for heterogeneous multi-core processors. It is not a skeleton library, instead using a task-based model of computation. It is considered in this thesis for its many similarities to SkePU such as aim, age, and implementation.

StarPU [3] uses a mix of pragmas and GNU-style attributes, as seen in Listing 4.4.

[3] `http://cast.rubyforge.org`
[4] `http://starpu.gforge.inria.fr`

Listing 4.5: Cilk Fibonacci computation.

```
cilk int fib(int n)
{
  if (n < 2)
    return n;
  else
  {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return x + y;
  }
}
```

4.8 Cilk

Cilk is a relatively old task-based language and runtime system. Cilk implements a work-stealing scheduler [6] as a means of distributing load between processors. A Cilk computation consists of *processes* and *threads*, forming a directed acyclic graph (DAG) at run time by *spawning* new threads when necessary. Spawns are function calls resulting in the creation of a new thread; the calls are annotated with the `spawn` keyword (see Listing 4.5), one of a few Cilk-specific keywords. Otherwise Cilk code is similar to C.

See also *Wool* [28], a task-based library inspired by Cilk aiming for extremely low-overhead task creation and management.

4.9 Boost C++ libraries

Multiple Boost[5] libraries are based on template metaprogramming, and are regarded as some of the best such libraries available. Abrahams and Gurtovoy [1] has documented their experiences with the Boost implementation.

- *Boost.MPL* [34] is perhaps the most well known of the Boost metaprogramming libraries, aiming to simplify metaprogramming in C++ by providing compile-time algorithms and data structures.
- *Boost.Fusion* focuses on heterogeneous containers and lazily evaluated algorithms.
- *Boost.Proto* is an expression template support library [49] (see Section 2.6.1).

[5] `http://www.boost.org`

Listing 4.6: A simple Scout loop annotation [40].

```
float a[100];
double b[100];
double x;

#pragma scout loop vectorize
for (int i = 0; i < 100; ++i)
{
  x = a[i];
  x = x / b[i];
  a[i] = x;
}
```

4.10 Eigen

Eigen[6] is a numerical linear algebra library built using expression templates. This gives Eigen a clean and type safe interface and support for aggressive partial and lazy evaluation, all decided at compile time. It also performs basic parallelization by explicit vectorization when such instructions are available.

4.11 CU2CL

CU2CL[7] is an automated CUDA to OpenCL source-to-source translator built using the Clang framework [47]. CUDA is more common in practical use than OpenCL thanks to the higher-level interface. Yet OpenCL is supported on a much larger collection of systems than CUDA. CU2CL thus aims to be an automatic translator of CUDA, performing the relevant substitutions and adding boilerplate OpenCL code. For this CU2CL uses Clang, citing the flexible architecture and community support. CU2CL specifically employs the Clang libraries AST, Lex, and Rewrite.

4.12 Scout

Scout[8] performs loop vectorization on C code, targeting a wide array of SIMD instruction formats [40]. The authors note that the approach is only semi-automatic, as the source needs to be annotated with `#pragma` directives as exemplified in Listing 4.6. While Scout uses Clang as a library, a custom build of Clang with a small patch is required for pragma recognition.

[6] `http://www.eigen.tuxfamily.org`
[7] `http://chrec.cs.vt.edu/cu2cl/`
[8] `http://scout.zih.tu-dresden.de`

Listing 4.7: Clad example program.

```
#include "clad/Differentiator/Differentiator.h"

double pow2(double x) { return x * x; }
double pow2_darg0(double); // Body will be filled by Clad.

int main()
{
  clad::differentiate(pow2, 0);
  printf("Result is %f\n", pow2_darg0(4.2));
  return 0;
}
```

4.13 Clad

Clad[9] is a compiler tool for automatic differentiation, a middle-ground between numeric and symbolic differentiation [67]. Clad performs source-to-source translation using Clang libraries to generate derivatives of C++ functions at (pre-)compile time. It can either produce valid C++ source code or compiled object files as output. As seen in Listing 4.7, the user defines a function, then *declares* the derivative of it and "calls" `clad::differentiate`, resulting in Clad filling in the definition. The authors note that the overhead of the extra compilation step is negligible for practical cases [67].

4.14 gpucc

A very recent Clang- and LLVM-based project is *gpucc* [73], an open-source alternative to Nvidia's CUDA compiler. `gpucc` is compatible with CUDA source code and includes a complete compiler chain, for example including a front-end and code generator. Aside from being the first open CUDA compiler, it focuses on improving both compile-time and run-time performance, outperforming `nvcc` in some tests.

4.15 PACXX

PACXX is a unified programming model for systems with GPU accelerators [35], utilizing the new C++14 language. PACXX shares many fundamental choices with SkePU 2 as proposed in this thesis, for example using modern C++ including attributes and basing the implementation on Clang. However, PACXX is not an algorithmic skeleton framework, and the compiler tool generates executables directly.

[9] `https://github.com/vgvassilev/clad`

4.16 HOMP

HOMP (*Heterogeneous OpenMP*) is a prototype implementation of the OpenMP 4 accelerator model [45]. It uses ROSE to generate CUDA code targeting GPUs.

4.17 REPARA

C++11 unified attributes (described in Section 2.4.2) are relatively new. One project which already has embraced the new notation is REPARA[10]. REPARA is an EU FP7 project, aiming to transform applications to reach high performance and energy efficiency on parallel and heterogeneous architectures, while preserving source code maintainability. On the surface, the REPARA methodology is similar to that of OpenMP; C++ code is annotated with information on which regions are parallelizable and what pattern to use. REPARA fully utilizes the advantage of attributes over pragma directives: attributes can be placed on any syntactic construct, not individual source lines [15].

In more detail, the REPARA methodology consists of multiple, separate steps. After the programmer annotates C++ code, a source-to-source translator (built into *Cevelop* IDE[11]) generates an *abstract immediate representation* (AIR). One of several parallel programming models is selected as target, and the AIR is transformed into *parallel programming model specific code*. The remaining steps are the standard C++ compiler phases, generating an executable program [15].

Programming models supported by REPARA include OpenMP, TBB, Cilk and FastFlow.

4.18 C++ Extensions for Parallelism

The C++ standardization committee has proposed extensions to the C++ STL for parallel algorithms [13]. The goal is to support possible realizations on a broad class of computer architectures. These extensions may be integrated into the main C++ specification in the future, perhaps as early as C++17.

The proposal extends the existing C++ algorithm collection with parallel overloads. A leading *execution policy* argument is added to the function call of an algorithm, and static "tag dispatching" directs to the relevant template definition. Parallel overloads exist for a large number of algorithms, including `std::for_each`, `std::find`, and `std::sort`.

[10] `http://repara-project.eu`
[11] `https://cevelop.com`

4.19 Others

As already covered, there are many different parallel and heterogeneous systems in use. There may be an even greater number of software tools and frameworks to program for those architectures. All of them cannot be covered in this thesis, but for the interested reader we list a few more relevant tools.

- **Intel TBB**[12]
  Intel Threading Building Blocks, a task parallel C++ template library [56].
- **FastFlow**[13]
  C++ framework for parallel stream-based applications [16].
- **CUDPP**
  library of parallel primitives [36].
- **BlockLib**
  Skeleton library for Cell BE; ancestor of SkePU [74].
- **Skandium**
  Java library for shared memory programming [44].
- **DatTel**
  C++ template skeletons [5].
- **QUAFF**
  C++, MPI-based template metaprogramming skeleton library [27].

[12] `https://software.intel.com/intel-tbb`
[13] `http://calvados.di.unipi.it/fastflow`

Chapter 5

Method

A common three-step separation of the method was applied for this thesis project: *pre-study* (Section 5.1), *implementation*, and *evaluation*. The steps were mostly executed in the listed order. The implementation phase consisted of interface specification (Section 5.2) architecture design (Section 5.3), and the actual implementation work (Section 5.4).

Two aspects of SkePU 2 were evaluated as part of the project: the new programming interface, with an empirical survey among computer science students (Section 5.5); and run-time performance of SkePU 2 programs, comparing backends and to SkePU 1 (Section 5.6).

5.1 Pre-Study

The first part of the method employed in this project is a pre-study of relevant source material. This includes:

- Existing tools used for providing programming environments for parallel and heterogeneous architectures.
- Programming concepts, frameworks, and tools used in compiler technology.
- Articles, conference papers and other research publications covering relevant topics.

The pre-study provided fundamental knowledge and understanding necessary for completing the implementation and evaluation. Prior and related work was found in large part by starting on publications from earlier SkePU development and following the references to previous papers, and citations by subsequent publications.

5.2 Interface Specification

A feature list and an application programming interface (API) for the new framework was designed with the previous version as a starting point. An internal list of conceptual ideas was used for guidance, but the author's own knowledge of and experience—with, for example, modern programming with C++—was the largest source of ideas in the end. The interface specification was first used as the basis of a prototype implementation and later continually revised as the project progressed.

5.3 Architecture Design and Prototyping

The first implementation step was to construct a prototype. The prototype implemented a subset of the interface specification—the MapReduce skeleton—selected specifically to cover many of the common design choices. This prototype was used as a basis for the mid-term evaluation and discussion, where the decision was made to base the final architecture and implementation on the prototype by incremental improvement.

To a large extent, the architecture and class design of SkePU 1 was preserved in the SkePU 2 runtime, drastically reducing the work which had otherwise been necessary in this area.

5.4 Implementation

Three approaches was considered for the implementation of SkePU 2: one based on the ROSE framework (see Section 2.8.1), one using Clang (Section 2.8.2), and one employing template metaprogramming (see Section 2.6) and the Boost libraries. Any combination of these were also possible.

Almost the entire project was written in C++, with some usage of shell and Python scripting for assisting in the evaluation.

A source code management and version control system repository, Git, was used to manage the source code for the implementation as well as documentation.

5.5 Usability Evaluation

To gain an understanding of how the new SkePU programming interface is received among users, we asked 16 master-level students to participate in an empirical survey. The respondents were presented with a questionnaire and were tasked to fill in answers without outside help. The responses were non-mandatory and anonymous, but some background information (such as age and experience with C++ and parallel programming) were collected.

The participants first read a single paragraph of background context, presenting SkePU and the algorithmic skeleton concept. They were told

that the survey was part of a thesis project proposing a new interface for SkePU programming. The respondents' main task was then to read two small SkePU programs, presented in both SkePU 1 and SkePU 2 syntax, grading the code clarity of both versions and then comparing the two. There was also an opportunity to leave a comment describing the reason for the grading. To avoid biasing one or the other SkePU 2 version, the order in which the versions were listed was randomized.

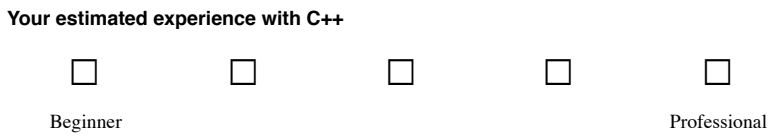


Figure 5.1: Example question from the survey.

The participants had no prior experience with the SkePU project and got only a short time (1–2 minutes) to understand each code example. In all grading questions, the scale went in five steps, presented as check-boxes with explanations on the extreme ends, see the example in Figure 5.1.

5.6 Performance Evaluation

Evaluating the new SkePU tool was done by porting the example programs available in the official SkePU distribution to the new syntax. The statistics of each implementation was considered, such as lines of code. In some cases, the improved expressivity allowed for new optimizations. To evaluate this, and also make sure that the new syntax does not result in a general performance regression, each application was evaluated for execution time as well.

The system used for testing had two eight-core Intel Xeon E5-2660 "Sandy Bridge" processors at 2.2GHz, with 64 GB DDR3 1600 MHz memory and was equipped with multiple Nvidia Tesla k20x GPU accelerators. 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.

5.6.1 Example Programs

The following is a list of test programs mentioned in this thesis.

- **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.
- **Mandelbrot Fractal**
  Uses 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 afterward. In fact, there are no non-constant inputs to the user function, as the index into the output container is all that is needed to calculate the return value.

- **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 takes a unique two-dimensional element index, 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.

- **Cumulative Moving Average**
  Calculates the average value of elements in a vector, up to and including itself. Performed in two passes: a prefix sum using Scan followed by unary Map with element index to compute the average value.

- **Median Filtering**
  Performs median filtering of an image, with a single MapOverlap skeleton instance and a quite complex user function. Performance evaluations run on random noise images with one-byte gray-scale pixels. The problem size parameter of this program is the side of the stencil area (i.e., overlap), so the number of processed pixels grows quadratically.

Subsets of these programs are used for usability evaluation, compilation time testing, and run-time performance evaluation.

5.7 Testing

The proposed interface for SkePU 2 is incredibly flexible. As the number and variety of programs which can be constructed are so numerous, it is difficult to test the implementation for correct behavior. Even 100 % code coverage is far from enough, since template instantiation in C++ can give very different behavior depending on the template parameters. Aside from the aforementioned example programs, SkePU 2 has therefore been tested with a fuzzy method. A meta-program (written in Python) generates random SkePU 2 programs conforming to a predefined pattern; the randomness affects which skeletons are used, their types and arities, container sizes etc. At run-time, the container arguments are initialized with random data as well. In this way, both the compiler and runtime system can be tested. The

program generator does not provide reference output, so we instead rely only on successful compilation and inter-backend comparison of the output.

5.8 Presentation of Results

The results from this thesis consist of source code for the new SkePU implementation, along with the results and discussion from evaluations as presented in this thesis report. The source code repository itself will be handed over to Linköping University, as it contains both up-to-date code and the entire commit history.

Starting some time after this report is published, stable parts of SkePU 2 will be available as open source on the SkePU website[1]. A separate technical documentation and user guide will also be written and provided with the open source distribution.

A paper on SkePU 2 and the results from this thesis has been accepted for HLPP 2016[2] and will be presented on July 4 2016 at HLPP in Münster, Germany [26].

[1] `http://www.ida.liu.se/labs/pelab/skepu/`

[2] 9th International Symposium on High-Level Parallel Programming and Applications

Chapter 6

Interface

This chapter introduces the SkePU 2 programming interface in Section 6.1. The most fundamental constructs in SkePU 2 are the skeletons, as described in Section 6.2. Skeletons are parametrizable with user functions, presented in Section 6.3.

6.1 Introduction

The SkePU 2 interface is defined in terms of C++ header files and can be used independently from the source-to-source translator. However, to actually utilize the parallel functionality of SkePU, the translator has to be used; in this mode, there are additional restrictions on the program code. A user function can, for example, not contain side effects (e.g., by allocating memory) or use most standard library functionality.

A full interface specification will be published separately, along with the open-source release of SkePU 2.

6.2 Skeletons

Skeletons are declared with an inferred type (using the `auto` specifier) and defined by assignment from a factory function, as exemplified in Listing 6.1. The actual type of a skeleton should be regarded as unknown to the programmer. The assigned variable is annotated with the `[[skepu::instance]]` C++11-style attribute to guide the source-to-source translator.

A skeleton is invoked with the call operator, with the arguments ordered according to the user function. The output container, where applicable, is passed by reference as the first argument. Smart containers may be passed by reference or by iterator, the latter allowing operations on partial vectors or matrices. A particular grouping of arguments is required by SkePU 2: all

Listing 6.1: SkePU 2 example application: PPMCC calculation.

```
#include <iostream>
#include <cmath>
#include <skepu2.hpp>

// User functions
template<typename T>
[[skepu::userfunction]]
T square(T a)
{
  return a * a;
}

template<typename T>
[[skepu::userfunction]]
T mult(T a, T b)
{
  return a * b;
}

template<typename T>
[[skepu::userfunction]]
T plus(T a, T b)
{
  return a + b;
}

using T = float; // Set precision
using namespace skepu2;

int main(int argc, char *argv[])
{
  // Size and smart containers
  size_t N = 100;
  Vector<T> x(N), y(N);

  // Skeleton instances
  auto vsum   [[skepu::instance]] = Reduce(plus<T>);
  auto dotp   [[skepu::instance]] = MapReduce<2>(mult<T>, plus<T>);
  auto vsumsq [[skepu::instance]] = MapReduce<1>(square<T>, plus<T>);

  // Perform computations
  T sumX = sum(x);
  T sumY = sum(y);

  T res = (N * dotp(x, y) - sumX * sumY)
    / sqrt((N * vsumsq(x) - pow(sumX, 2))
         * (N * vsumsq(y) - pow(sumY, 2)));;

  return 0;
}
```

	Map	Reduce	Scan	MapReduce	MapOverlap	Call
Elwise vector	•	•	•	•	With overlap	
Elwise matrix	•	•	•	•	With overlap	
Elwise arity	$N \geq 0$	1	1	$N > 0$	1	0
Extra containers	•			•	•	•
Scalars	•			•	•	•
Indexing	•			•		

Table 6.1: SkePU 2 skeletons and their features and attributes

Listing 6.2: Example usage of the Map skeleton.

```
[[skepu::userfunction]]
float sum(float a, float b)
{
  return a + b;
}

Vector<float> vector_sum(Vector<float> &v1, Vector<float> &v2)
{
  auto vsum [[skepu::instance]] = Map<2>(sum);
  Vector<float> result(v1.size());
  return vsum(result, v1, v2);
}
```

element-wise containers must be grouped first, followed by all random-access containers, and scalar arguments last.

There are six skeletons available in SkePU 2, fewer than in SkePU 1, as the generalized Map now covers the use-cases of MapArray and Generate. See Table 6.1 for a list of the skeletons.

6.2.1 Map

Map is greatly expanded compared to SkePU 1. A Map skeleton accepts N containers for any integer N including 0. These containers must be of equal size, as does the output container. As one element from each of these containers will be passed as arguments to a call to a user function, we refer to these containers as *element-wise arguments*. Map additionally takes any number of SkePU containers which are accessible in their entirety inside a user function—called *random access arguments*—thus rendering MapArray from SkePU 1 redundant. These parameters are declared to be either in, out, or inout arguments and only copied (e.g., between the CPU and an accelerator) when necessary. Finally, scalar arguments can also be included, passed unaltered to the user function. The Map skeleton is thus *three-way variadic*, as each group of arguments is handled differently and is of arbitrary size.

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 param-

Listing 6.3: Example usage of the Reduce skeleton.

```
[[skepu::userfunction]]
float min_f(float a, float b)
{
  return (a < b) ? a : b;
}

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

eter's type is one out of two `structs`, `skepu2::Index1D` for vectors and `skepu2::Index2D` for matrices. 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.

See Listing 6.2 for an example usage of the Map skeleton.

6.2.2 Reduce

The Reduce skeleton is a generic reduction operation with an associative operator available in multiple variants. A vector is reduced in only one way while five options exist for matrices. A reduction on a matrix may be performed in either one or two dimensions (for two-dimensional reduction the user supplies two user functions), both either row-wise or column-wise. The fifth mode treats the matrix as a vector (in row-major order) and is the only mode available if an iterator into a matrix is supplied.

See Listing 6.3 for an example usage of the Reduce skeleton.

6.2.3 MapReduce

MapReduce is a combination of Map and Reduce and offers the features of both, with the limitation that the element-wise arity must be at least 1. See Listing 6.4 for an example usage of the MapReduce skeleton.

6.2.4 Scan

The Scan skeleton implements two variants of the prefix sum operation generalized to any associative binary operator. The variants are inclusive or exclusive scan, where the latter supports a user-defined starting value.

See Listing 6.5 for an example usage of the Scan skeleton.

Listing 6.4: Example usage of the MapReduce skeleton.

```
[[skepu::userfunction]]
float add(float a, float b)
{
  return a + b;
}

[[skepu::userfunction]]
float mult(float a, float b)
{
  return a * b;
}

float dot_product(Vector<float> &v1, Vector<float> &v2)
{
  auto dotprod [[skepu::instance]] = MapReduce<2>(mult, add);
  return dotprod(v1, v2);
}
```

Listing 6.5: Example usage of the Scan skeleton.

```
[[skepu::userfunction]]
float max_f(float a, float b)
{
  return (a > b) ? a : b;
}

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

Listing 6.6: Example usage of the MapOverlap skeleton.

```
[[skepu::userfunction]]
float conv(
  int overlap, size_t stride,
  const float *v, const float *stencil, float scale
)
{
  float res = 0;
  for (int i = -overlap; i <= overlap; ++i)
    res += stencil[i + overlap] * v[i*stride];
  return res / scale;
}

Vector<float> convolution(Vector<float> &v)
{
  auto convol [[skepu::instance]] = MapOverlap(conv);
  Vector<float> stencil {1, 2, 4, 2, 1};
  Vector<float> result(v.size());
  convol.setOverlap(2);
  return convol(result, v, stencil, 10);
}
```

6.2.5 MapOverlap

MapOverlap is a one or two-dimensional stencil operation. Parameters for specializing the boundary handling are available, and there is specific support for separable 2D stencils.

See Listing 6.6 for an example usage of the MapOverlap skeleton.

6.2.6 Call

Call is a completely new skeleton for SkePU 2. It is not a skeleton in a strict sense, as it does not enforce a specific structure for computations. Call simply invokes its user function in parallel. The programmer can provide arbitrary computations as explicit user function backend specializations, which must include at least a sequential general-purpose CPU backend as a default variant. Naming conventions are used to locate the source code files containing the implementation variants for various supported backend types. The direction (in, out, inout) of parameter data flow follows the same principles as for the Map skeleton described above. Call provides seamless integration with SkePU features such as smart containers and auto-tuning of back-end selection.

Basically, Call extends the traditional skeleton programming model in SkePU with the functionality of user-defined *multi-variant components* (i.e., "PEPPHER" components [20]) with auto-tunable automated variant selection. The current interface is limited in that variants can only be coarsely targeted at SkePU backends, with a cumbersome syntax. In the future this shall be extended and integrated with a platform description language such as XPDL [39].

Listing 6.7 contains an example application of the Call skeleton, integer sorting, which does not translate well to data-parallel skeleton programming. Two distinctly different algorithms are selected depending on whether the Call instance is executed on CPU or GPU. (Note that the example is just an illustration; the CPU insertion sort algorithm is inefficient, and the even-odd sorting in the GPU variant works only inside a single work group.)

6.3 User Functions

A skeleton is parameterized by user-defined components to create skeleton instances. These components are defined as ordinary (free) functions in SkePU 2, contrary to the macro syntax of SkePU 1, and are called *user functions*. A user function is annotated with the `[[skepu::userfunction]]` attribute, declaring the user's intent and helping the precompiler to recognize their usage. Many examples of user functions can be seen in the code listings referenced throughout Section 6.2, for example Listing 6.2.

Alternatively, user functions can be specified using C++11 lambda expressions (closures). This may be preferred to free functions, as lambdas do

Listing 6.7: Example usage of the Call skeleton.

```
[[skepu::userfunction]]
void swap_f(int *a, int *b)
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

[[skepu::userfunction]]
void sort_f(int *array, size_t nn)
{
#if SKEPU_USING_BACKEND_CL

  // Even-odd sort
  // Multiple invocations in parallel
  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 // SKEPU_USING_BACKEND_CPU

  // Insertion sort
  // A single, sequential invocation
  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(skepu2::Vector<int> &v, skepu2::BackendSpec spec)
{
  auto sort [[skepu::instance]] = skepu2::Call(sort_f);

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

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

Listing 6.8: User function specified with lambda syntax.

```
Vector<float> vector_sum(Vector<float> &v1, Vector<float> &v2)
{
  auto vsum [[skepu::instance]] = Map<2>([](float a, float b)
  {
    return a + b;
  });

  Vector<float> result(v1.size());
  return vsum(result, v1, v2);
}
```

not pollute the enclosing namespace and the userfunction attribute is not needed. However, lambdas are only used as syntactic sugar in SkePU 2, and not every lambda can be used. For example, the lambda must be stateless and can thus not capture any variables. Listing 6.8 uses lambda syntax for a program with identical semantics to that in Listing 6.2.

It is possible to insert backend-specific code into user function bodies by using preprocessor directives. This bypasses the source-to-source translator (the sections are copied without being parsed) and should as such be used as a last resort, or for temporary testing or debugging purposes. It is currently also the recommended way to do per-backend specialization of user functions, as seen in Listing 6.7.

6.4 Explicit Backend Selection

While automatic backend selection is a major feature of SkePU, it is also possible to explicitly request a backend when invoking a skeleton instance. In SkePU 1, this was done completely statically by calling the appropriate member function (e.g., `.CPU` or `.CU`) on the instance. SkePU 2 instead takes a more dynamic approach with a `BackendSpec` object member on the skeleton instance. This object encodes the requested backend type and parameters applicable to the backend, such as the number of threads to use in OpenMP. The user may simply set this member before invocation, as in Listing 6.7; the runtime system always checks this structure before dispatching the call to any backend. Although there may be a small performance cost with this dynamic approach, we believe the increased flexibility will be useful for users of SkePU 2 in the future. For example, entirely new tuning mechanisms can be created on top of SkePU 2 without any modification to its implementation.

The runtime system is allowed to override the user's backend request, for example in the case where a certain backend is not available.

Chapter 7

Implementation

In this chapter the implementation of SkePU 2 is presented. This includes the overarching architecture in Section 7.1, the implementation of the skeleton runtime (Section 7.2.1 covers the sequential variant and Section 7.2.2 the parallel backend variants) .

Section 7.3 introduces the Clang-based source-to-source compiler, including its purpose, implementation, invocation and distribution.

7.1 Architecture

SkePU 2 consists of three fundamental parts: a sequential skeleton interface and runtime, a source-to-source translator, and the parallel runtime system with multiple backends supported. The integration of these parts is illustrated in Figure 7.1.

7.2 Skeletons

As in SkePU 1, skeletons in SkePU 2 are implemented as distinct class templates. Some details are tightened up in SkePU 2, primarily to avoid code duplication; for example, a common base class contains members shared between all skeletons (such as execution plans). A major defining characteristic of SkePU 2 however, is the fact that each skeleton is implemented in two different ways. A sequential implementation is used when a SkePU 2 program is compiled without initial source-to-source transformation, with the far more interesting parallel backend implementations are used normally.

To facilitate the multiple definitions, skeleton instances must be declared with an `auto` inferred type. For this reason, the instances are constructed by calling a factory function, which has the added benefit of the possibility to infer template parameters for the skeleton types. That constructors in C++ cannot infer these parameters is a known limitation and the raison d'être for

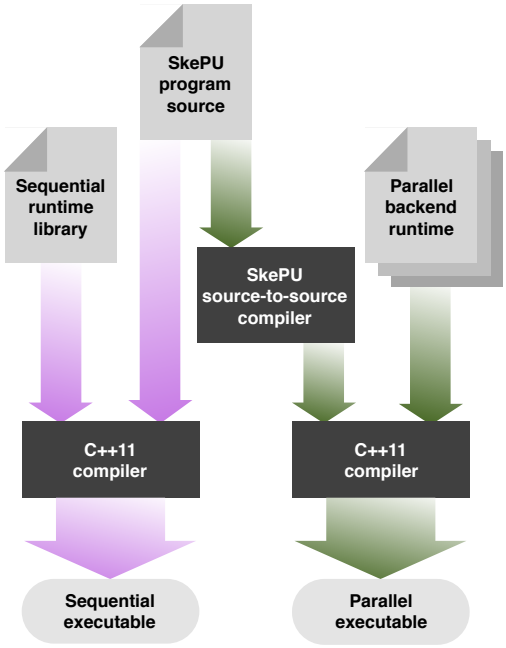


Figure 7.1: SkePU 2 compiler chain.

STL `std::make_` functions, e.g., `std::make_tuple` and `std::make_unique`. We considered following this style in SkePU 2 but in the end chose the approach of mimicking constructors.

7.2.1 Sequential Skeleton Variants

SkePU 2 programs are valid C++11 programs, with identical semantics regardless of whether the SkePU attributes are handled or ignored. They can thus be compiled with any C++11-conforming compiler. Programs generated this way will utilize a straightforward, sequential implementation of skeletons. The intention is that this mode should be used for application development—as the sequential skeletons guarantee identical output to the parallel implementation (barring hardware-related limitations of parallel algorithms[1]), applications can be developed and initially tested without the precompiler.

With the use-case of development and debugging in mind, the sequential interface has been constructed to perform many run-time checks which would be unacceptable in a high-performance context, e.g., container size and bounds checks.

[1]Such as limited memory sizes.

This straightforward and clean implementation of the skeletons can also act as a reference when constructing complex parallel algorithms. In this project, a variant of fuzzy testing was been performed, where input data has been randomly generated and the output of a program using the preprocessed, parallel run-time was compared to that of the sequential variant.

7.2.2 Parallel Backends

The backends are separate OpenMP, OpenCL, and CUDA implementations of all skeletons accompanied by another single-threaded CPU variant. The existence of a CPU backend separate from the sequential implementation is partly a consequence of the slight differences in the interface between preprocessed and non-preprocessed SkePU 2 code (e.g., a different set of template parameters) and partly a way to provide an efficient single-threaded variant, without the bounds checking etc. that the sequential runtime system includes.

Many implementation decisions such as parallel algorithms and the entire container implementation have been preserved from SkePU 1. However, the architecture of the skeletons has changed a lot.

7.3 Source-to-Source Compiler

The SkePU 2 *source-to-source compiler* (also *source-to-source translator* or just *precompiler*) knows about the skeleton library interface and recognizes attributes in the source code. A program targeting the skeleton interface which is properly annotated with attributes can be transformed for heterogeneous parallel execution by the precompiler.

The role of SkePU 2's source-to-source precompiler is to transform a subset of programs written for the sequential interface for parallel execution. It is guided by attributes, `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. While most SkePU 2 attributes are not strictly needed to recognize skeleton usage in a C++ program, they provide for a straightforward implementation of the precompiler—but more importantly, they help declare the programmer's intent and thus generate better diagnostics.

The job of the precompiler is limited by design. Its main purpose is to transform user functions, for example adding `__global__` 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.

Listing 7.1: Before transformation.

```
template<typename T>
[[skepu::userfunction]]
T arr(skepu2::Index1D row, T *m, const T *v, T *v2 [[skepu::out]])
{
  // body
}
```

Listing 7.2: After transformation.

```
struct skepu2_userfunction_arr_float {
  using T = float;
  constexpr static size_t totalArity = 4;
  constexpr static size_t anyContainerArity = 3;
  constexpr static bool indexed = 1;
  static skepu2::AccessMode anyAccessMode[anyContainerArity];
  using Ret = float;

#define SKEPU_USING_BACKEND_CUDA 1
  static inline SKEPU_ATTRIBUTE_FORCE_INLINE __device__ float
  CU(skepu2::Index1D row, T *m, const T *v, T *v2) {
    // body
  }
#undef SKEPU_USING_BACKEND_CUDA

#define SKEPU_USING_BACKEND_OMP 1
  static inline SKEPU_ATTRIBUTE_FORCE_INLINE float
  OMP(skepu2::Index1D row, T *m, const T *v, T *v2) {
    // body
  }
#undef SKEPU_USING_BACKEND_OMP
};

skepu2::AccessMode
skepu2_userfunction_arr_float::anyAccessMode[anyContainerArity] {
  skepu2::AccessMode::ReadWrite,
  skepu2::AccessMode::Read,
  skepu2::AccessMode::Write,
};

"#define SKEPU_USING_BACKEND_CL 1
static float arr_float(index1_t row, __global float * m,
  __global const float * v, __global float * v2) {
  typedef float T;
  // body
}"
```

Listing 7.3: Internal OpenCL kernel launch in SkePU 1.

```
// Sets the kernel arguments
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&in_p);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&out_p);
clSetKernelArg(kernel, 2, sizeof(size_t), (void*)&numElem);
clSetKernelArg(kernel, 3, sizeof(typename MapFunc::CONST_TYPE),
  (void*)&const1);

globalWorkSize[0] = numBlocks * numThreads;
localWorkSize[0] = numThreads;

// Launches the kernel (asynchronous)
err = clEnqueueNDRangeKernel(
  m_kernels_CL.at(i).second->getQueue(),
  kernel, 1, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL);
printCLError(err, "Error launching kernel\n");
```

Listing 7.4: Internal OpenCL kernel launch in SkePU 2.

```
CLKernel::map(
  i,                                // Device ID
  numThreads,                       // Local work size
  numBlocks * numThreads,           // Global work size
  std::get<EI>(elwiseMemP)...,      // Elwise container args
  std::get<AI-arity>(anyMemP)...,   // Random container args
  get<CI, CallArgs...>(args...)..., // Scalar args
  outMemP[i],                       // Output container
  res.getParent().total_cols(),     // Container width
  numElem,                          // Container size
  i * numElemPerSlice               // Start index
);
```

For some backends such as OpenCL and CUDA, all kernel code is generated by the precompiler. OpenCL programming usually includes a type-unsafe border between the host program and kernel invocations on the device (that is, `void*` pointers to kernel arguments), but the SkePU 2 precompiler generates typed wrapper functions resulting in a programming interface very similar to that of CUDA. This is, however, invisible to end-users, as device kernels are not called directly in the SkePU interface, but greatly simplifies internal backend development and debugging. In Listings 7.3 and 7.4 OpenCL kernel calls directly from the implementations of the Map skeleton are shown. Note that the SkePU 2 variant accepts more arguments as it supports optional element indexing and is variadic in the user function arguments (the struct `CLKernel` is generated by the precompiler, passed to the Map implementation as a template type parameter).

An example of a transformation[2] of the template user function in Listing 7.1 can be seen in Listing 7.2.

[2]Slightly altered and reformatted for presentation purposes.

Listing 7.5: An attribute definition in Clang.

```
def SkepuUserFunction : InheritableAttr
{
  let Spellings = [CXX11<"skepu", "userfunction">];
  let Subjects = SubjectList<[Function]>;
  let Documentation = [Undocumented];
}
```

Listing 7.6: A diagnostic definition in Clang.

```
def err_skepu_no_userfunction_body: Error
<
  "[SkePU] Declaration of function %0 with
   skepu::userfunction attribute is not a definition"
>;
```

7.3.1 Patching Clang

Clang, while primarily a compiler front-end, is built with a library-oriented architecture [47]. The SkePU 2 source-to-source translator is built on top of Clang's parsing, tooling and rewriting libraries. However, some features of the translator cannot be realized within the constraints of the library interface and needs to be integrated at the source level. Attribute recognition and handling is the primary requirement for this in SkePU 2, but we also improve the usability of the precompiler by integrating custom diagnostics, i.e., errors and warnings, for common mistakes (such as forgetting the `[[skepu::userfunction]]` attribute on a user function passed to a skeleton instance).

An example of an attribute definition added to Clang as part of SkePU 2 is presented in Listing 7.5. In this example, the `[[skepu::userfunction]]` attribute is defined in Clangs attribute definition DSL. It is added to the file `include/clang/Basic/Attr.td` relative to the root of the Clang repository.

SkePU 2 diagnostics are added to the file `include/clang/Basic/DiagnosticASTKinds.td` as in Listing 7.6. This example defines an error signaling that the `[[skepu::userfunction]]` attribute has been placed on a function declaration without a definition, which is not allowed.

7.3.2 Invocation

The current version of the SkePU 2 source-to-source translator is invoked on a single source file. The source file may depend on headers, which have to be reachable for parsing by the tool, but only the main file will be subject to source-to-source translation. The output from the translator is a transformed copy of the original file. A standard C++ compiler (or specialized compiler depending on which backends are enabled, e.g, `nvcc`) should then be used for normal compilation.

In the future, we plan to construct a compiler driver to simplify this process. It would function similarly to `nvcc`, performing both the initial transformation step and target compilation with a user-definable compiler.

7.3.3 Distribution

A few options exist for the distribution of a Clang tool. One possibility is to create a pull request and hope to be accepted as a part of Clang itself; this was not attempted, however, as SkePU 2 is still immature and probably not relevant enough. A second option, commonly seen with software of all kinds, is to build executable binaries for multiple platforms and make them available for example on the web or in package management systems. This is a more realistic approach and may be an interesting future possibility. However, the current plan is to distribute the patch file in a Git-compatible format along with the SkePU 2 runtime library source code. A SkePU 2 user must then clone the Git repositories by herself, check out a specified version and apply the patch. This approach is applied by Scout [40] where it is, to an extent, automated.

Chapter 8

Results and Discussion

In this section, we present empirical results from SkePU 2. First the usability improvements are covered in Section 8.1, followed by an example of improved type safety in Section 8.2. Improvements to the implementation of the existing SkePU libraries are presented in Section 8.3 and, finally, performance results in Section 8.4. A discussion section accompanies each of these sections, and discussion of the methodology is presented in Section 8.5.

8.1 Usability Survey

The interface of SkePU 2 improves on that of SkePU 1 with increased clarity and a syntax that looks and feels more native to C++, making SkePU 2 more usable than its predecessor. A survey was issued to 16 participants, all master-level students in computer science (see Section 5.5). The results of this survey are presented in this section.

8.1.1 Example Programs

The respondents were presented with two short example programs, each in two different versions (SkePU 1 and SkePU 2). The programs were presented exactly as in Listing 8.1, 8.2, 8.3, and 8.4; with no explanation other than the program title ("vector sum" and "Taylor series", respectively) and any descriptive symbol names in the code. The participants were instructed to spend one to two minutes to read each example. To avoid biasing either of the SkePU versions, the order of introductions was reversed in half of the questionnaires.

- **Version A**: Presented the SkePU 2 version first.
- **Version B**: Presented the SkePU 1 version first.

The order of the examples was consistent: the trivial vector sum followed by the more complex Taylor series approximation.

Listing 8.1: Vector sum example in SkePU 1.

```
BINARY_FUNC(sum, int, a, b,
  return a + b;
)

Vector<float> vector_sum(Vector<float> &v1, Vector<float> &v2)
{
  Map<sum> vsum(new sum);
  Vector<float> result(v1.size());

  vsum(v1, v2, result);
  return result;
}
```

Listing 8.2: Vector sum example in SkePU 2.

```
[[skepu::userfunction]]
float sum(float a, float b)
{
  return a + b;
}

Vector<float> vector_sum(Vector<float> &v1, Vector<float> &v2)
{
  auto vsum [[skepu::instance]] = Map<2>(sum);
  Vector<float> result(v1.size());

  vsum(result, v1, v2);
  return result;
}
```

Listing 8.3: Approximation by Taylor series in SkePU 1.

```
UNARY_FUNC_CONSTANT(kth_term, float, float, k, x,
  float temp_x = pow(x, k);
  int sign = ((int)k % 2 == 0) ? -1 : 1;
  return sign * temp_x / k;
)

BINARY_FUNC(plus, float, a, b,
  return a + b;
)

GENERATE_FUNC(index_init, float, float, index, seed,
  return index + 1;
)

float taylor_approx(float x, size_t N)
{
  skepu::MapReduce<kth_term, plus> taylor(new kth_term, new plus);
  skepu::Generate<index_init> vec_init(new index_init);

  taylor.setConstant(x);

  skepu::Vector<float> terms(N);
  vec_init(N, terms);

  return taylor(terms);
}
```

Listing 8.4: Approximation by Taylor series in SkePU 2.

```
[[skepu::userfunction]]
float kth_term(skepu2::Index1D index, float x)
{
  int k = index.i + 1;
  float temp_x = pow(x, k);
  int sign = (k % 2 == 0) ? -1 : 1;
  return sign * temp_x / k;
}

[[skepu::userfunction]]
float plus(float a, float b)
{
  return a + b;
}

float taylor_approx(float x, size_t N)
{
  auto taylor [[skepu::instance]] = skepu2::MapReduce<0>(kth_term, plus);

  taylor.setDefaultSize(N);

  return taylor(x);
}
```

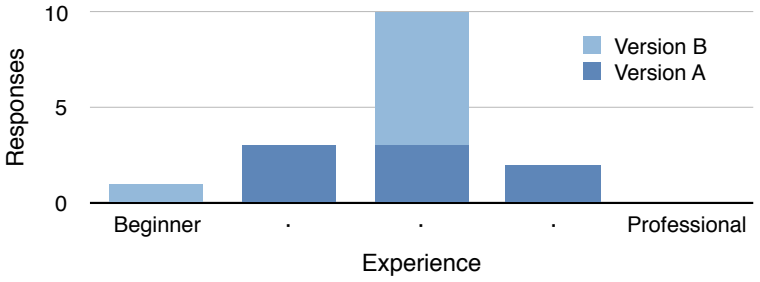
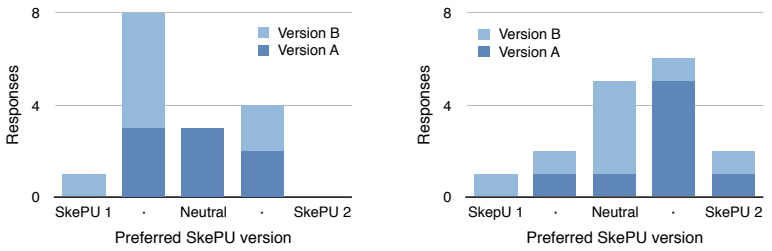


Figure 8.1: Survey participants' estimated C++ experience.

8.1.2 Survey Responses

All participants were students in advanced-level computer science or engineering programs. They were instructed to estimate their C++ experience on a five-step scale from *beginner* to *professional*. The results (Listing 8.1) show that most respondents considered themselves as having average experience (the data seemingly conforms to a normal distribution). When the participants later graded their understanding of the example programs, the vector sum example showed a small negative correlation between estimated experience and level of understanding while the Taylor series example showed no correlation at all.

Figure 8.2 presents the results of the final question for each code example, where the participants were constructed to compare the two SkePU versions in terms of code clarity (*How would you rate the clarity of this code*



(a) Vector sum (b) Taylor series

Figure 8.2: Comparison of code clarity, SkePU 1 vs. SkePU 2.

in relation to the previous example?). The scale went from *less clear* to *more clear* in five steps. The questionnaires which had SkePU 1 first have had their responses inverted afterwards, so a different scale is presented in Figure 8.2.

We can see that the first example program was very divisive, with a slight skew to SkePU 1 as the clearer version. For the second program, Taylor series, there are clearly more responses preferring SkePU 2 than SkePU 1.

8.1.3 Discussion

The usability evaluation indicates that the SkePU 1 interface is sometimes preferred to the SkePU 2 variant, at least when the user is not used to C++11 attributes. From the comments left in the survey, we realized that is important to clearly specify that SkePU attributes does not alter the semantics of the program. There might even be a reason to reduce the attribute usage in the interface; for example, the `[[skepu::instance]]` attribute can be removed only at the cost of a more complex detection mechanism in the source-to-source translator.

In the more complex Taylor series 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 Taylor series program is also presented after the participants have already seen their first use of C++11 attributes, in the vector sum example. This may skew the result towards preferring SkePU 2, which would distort the data but indicate that the attributes are easy to get accustomed to. Some respondents mention the attributes as having a negative effect also for the second example, however.

There is a clear trend in the data suggesting that a respondent favors the first version of the presented example program. It was therefore good that

Listing 8.5: Invalid SkePU 1 code.

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

float vsum(skepu::Vector<float> &v) {
  skepu::Reduce<plus_f>
    globalSum(new plus_f);
  return globalSum(v);
}
```

Listing 8.6: Invalid SkePU 2 code.

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

float vsum(skepu2::Vector<float> &v) {
  [[skepu::instance]] auto globalSum
    = skepu2::Reduce(plus_f);
  return globalSum(v);
}
```

we made two different versions of the survey. We assume that the biases now cancel out in the aggregate, which might not be true in practice.

It is not clear why the respondents' grading of their C++ experience shows a small negative correlation to their understanding of the example programs. A possible explanation may be that the relatively new C++11 attributes catches experienced users off guard; but a more likely reason is that the sample size is too small, especially for the extreme ends of the experience spectrum.

8.2 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 8.5 shows the error in SkePU 1 code, and Listing 8.6 illustrates the same in SkePU 2 syntax.

The SkePU 1 example compiles without errors or warnings, and only at run-time terminates with the error message in Listing 8.7. The message itself is shared between all reduce instances, limiting the information obtained by the user.

SkePU 2, on the other hand, fails to compile and prints an error message even before the precompiler has transformed the code. (The message does not directly describe the issue, an aspect which can be further improved by clever usage of C++11's `static_assert`.) It directs the user to the affected

Listing 8.7: 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 compiler time:
error: no matching function for call to 'Reduce'
   [[skepu::instance]] auto globalSum = skepu2::Reduce(plus_f);
                                        ^~~~~~~~~~~~~~
note: candidate template ignored: failed template argument deduction
   Reduce(T(*red)(T, T))
```

skeleton instance. The precompiler tool itself does not need to implement any form of type checking.

8.3 Parallel Runtime Improvements

The implementation of SkePU backends, i.e., the parallel runtime, has been significantly improved in SkePU 2. There are two major reasons for this: the move to C++11 and the introduction of a source-to-source precompiler. The improvements include the following:

- Reduced code size, in some cases approaching 70 %, for example by elimination of separate implementation of unary, binary, and ternary Map skeleton.
- Reduced code duplication, for example by more sharing of wrapper member functions by different backends.
- Improved type safety, especially along the C++-OpenCL boundary as detailed in Section 7.3.

The SkePU project has grown dynamically over multiple years. An inevitable part of this thesis project was to read though the code-base and understand the structure and intention, followed by a partial re-implementation of the skeletons. This procedure has, to some extent, functioned as a code audit and resulted in the discovery of a number of potential bugs and inconsistencies.

8.4 Performance Evaluation

The performance of SkePU 2 was evaluated in three ways: compile times, presented in Section 8.4.1; performance of SkePU 2 backends in Section 8.4.2; and comparison between SkePU 1 and SkePU 2 in Section 8.4.3.

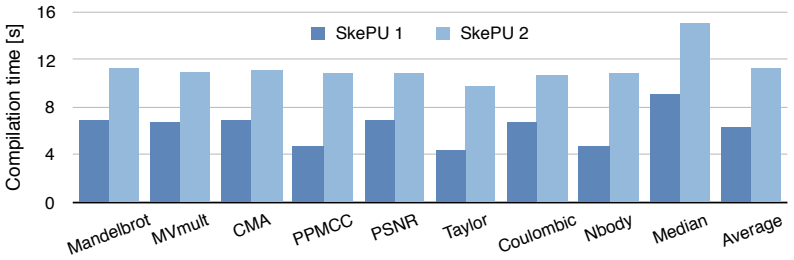


Figure 8.3: Comparison of compilation durations, SkePU 1 vs. SkePU 2.

8.4.1 Compile-Time Performance

SkePU 2 contains more compilation steps than SkePU 1, so it is reasonable to expect that compilation times are longer. To quantify the difference, a set of example programs written in two versions each (targeting SkePU 1 and SkePU 2, respectively) were recompiled three times, and the average of the durations recorded and presented in Figure 8.3. Here it can be seen that it is not uncommon for SkePU 2 compilation times to be twice that of SkePU 1.

Note that as a consequence of the difference in features across SkePU versions, some programs differ in the number of skeleton instances used or in other ways. The core algorithm is shared in all examples, however. Additionally, as SkePU 1 cannot enable both OpenCL and CUDA in a single compilation, all tests here are only generating CPU, CUDA and OpenMP backends. As the majority of OpenCL compilation is done just-in-time, adding the OpenCL backend does not adversely prolong the compilation time.

8.4.2 Performance Comparison of Backends

Results of the performance comparison of backends with various test programs can be seen in Figure 8.4 and 8.5b. In three out of six cases, the GPU backends (which are very close to each other) beat the CPU backends with a margin on large problem sizes. In the other three cases, OpenMP is the fastest. The sequential CPU backend is almost always faster for extremely small problem sizes.

8.4.3 Performance Comparison of SkePU versions

In cases where the increased flexibility of SkePU 2 allows a program to be implemented more efficiently—for example by reducing the number of skeleton invocations—SkePU 2 may outperform SkePU 1 significantly. Figure 8.5 shows such a case: approximation of the natural logarithm using Taylor series. For SkePU 1, this is implemented by a call to Generate followed by a

call to MapReduce; in SkePU 2 a single MapReduce is enough, reducing the number of GPU kernel launches and eliminating the need for $O(N)$ auxiliary memory.

8.4.4 Discussion

Google Benchmark[1] was used for run-time performance evaluation. This library has not been commonly used for scientific work, which may be a reason for readers to question the validity of the results (and, by extension, the conclusions as well). However, the results it reported were at least consistent across runs.

There are many difficulties in performing a fair performance evaluation for a complex library such as SkePU 2. It includes multiple skeletons with infinitely[2] many type combinations, each skeleton implemented on a set of distinct backend architectures. On top of that, the physical systems (particularly GPUs), while supporting a common programming model, can have very different attributes which affect the decisions made by the run-time. The evaluation presented in this thesis is thus not a completely fair illustration of SkePU 2 performance, but hopefully a good indication of what can be expected in typical use.

Results from the performance evaluation indicate that either the Scan and MapOverlap skeletons—or the test programs using them—may have performance bottlenecks, as the GPU backends are the slowest in these evaluations. Especially the Median filtering example behaves very unexpectedly.

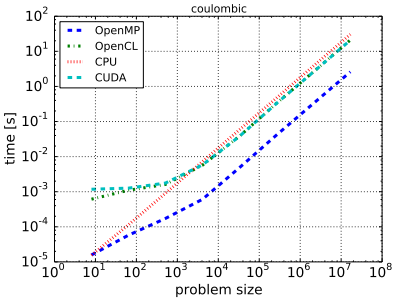
8.5 Method Discussion

For the source-to-source compiler, we selected an architecture based on Clang fairly early in the process. ROSE was considered as an alternative but dropped before any prototyping work began. If the project had had a larger scope in terms of time, multiple different prototypes could have been constructed and evaluated. In the end, Clang has proven to fit the task well.

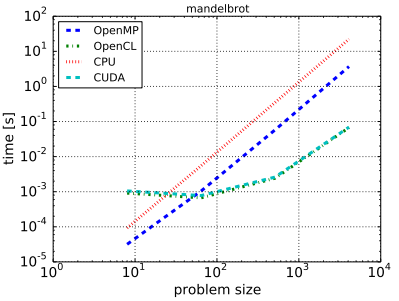
The usability survey gathered a total of 16 responses, which may be in the lower range in which reasonable conclusions can be made. However, even though the survey was presented to a specific audience of students, the reported backgrounds were surprisingly varied, with different fields of study and programming experience. The survey itself was well structured, with two versions of the questionnaire handed out to avoid biasing either of the SkePU versions. Especially the free-form comment field, a late addition, turned out to be very useful when interpreting the results.

[1] `https://github.com/google/benchmark`

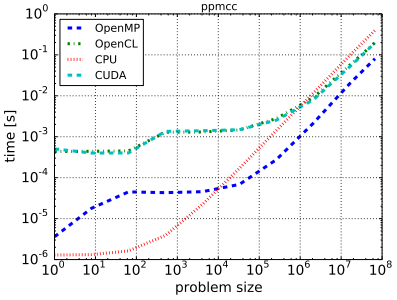
[2] The SkePU 2 interface does allow for infinitely many type and arity combinations, but compiler implementations will enforce a (possible configurable) limit here. Template instantiation depth has been the limiting factor in the author's experience.



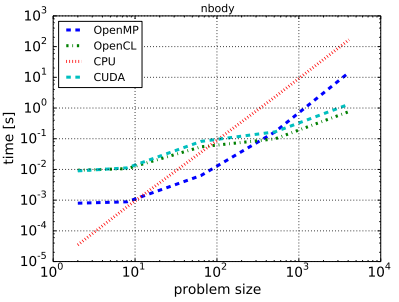
(a) Coulombic potential



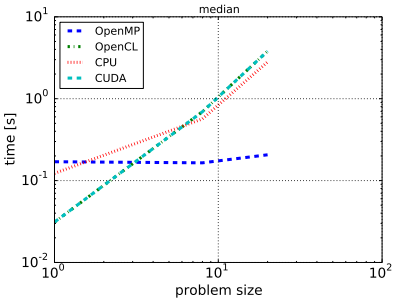
(b) Mandelbrot fractal



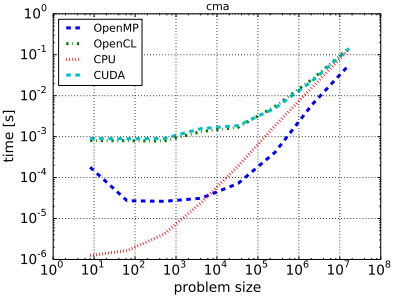
(c) Pearson product-movement correlation coefficient



(d) N-body simulation

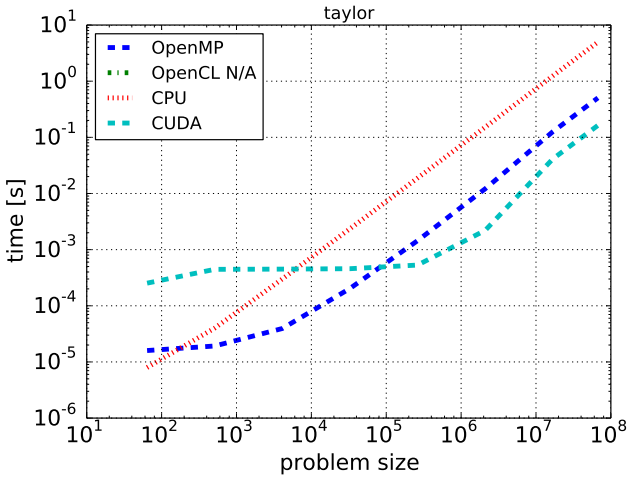


(e) Median filtering

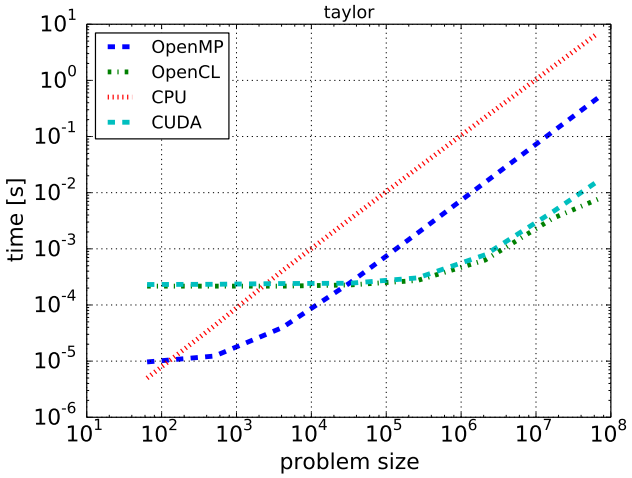


(f) Cumulative moving average

Figure 8.4: Test program evaluation results. Log-log scale.



(a) SkePU 1.2



(b) SkePU 2

Figure 8.5: Comparison of Taylor series approximation.

Chapter 9

Conclusions

This section concludes the thesis by presenting the conclusions reached during the project, based on the research questions (Section 9.1). We also consider the relevance of the project in a wider context in Section 9.2 and possibilities for future work in Section 9.3.

9.1 Revisiting the Research Questions

In this section, conclusions are made in relation to the research questions from Section 1.3.

9.1.1 Language Embedding

Chapter 4 lists several research projects in the form of C++ frameworks, all of which more or less deliberately define domain-specific embedded languages. For some of them this is just a C++ header interface, among them SkePU 1; while some define their own extensions to the core language. The most popular way to extend C++ is by preprocessor `pragma` directives, which the compiler usually can ignore while still producing a valid, but limited, executable. The REPARA project uses C++11 attributes in a similar way, as does SkePU 2 as proposed.

9.1.2 Type-Safe Skeleton Programming

By constructing a proposed next-generation version of the SkePU framework, we have demonstrated a way to design and implement a type-safe C++ interface for skeleton programming. The proposed framework is based on research of similar projects and the recent evolution of the C++ language. The chosen approach builds on C++11, specifically variadic templates and attributes, and is supported by a source-to-source translation tool. While

not very common, each of those separate techniques have precursors in various contexts, even in parallel programming. However, the combination applied in a skeleton programming framework is unprecedented.

9.1.3 Source-to-Source Precompiling

We have shown that a source-to-source compiler tool, in our case built on the Clang libraries, can, with promising results, be used as a precompiler in a skeleton programming framework. There are plans to extend the translator with target-specific optimization capabilities in the future, but for now only a rudimentary and limited implementation exists.

9.2 Relevance

As described in the introduction in Chapter 1, parallel and heterogeneous computer architectures are increasingly important to consider for more and more types of programmers. The programming frameworks in popular usage today, e.g., OpenCL, are often fairly low-level and requires hardware-specific optimization of, for example, memory handling. SkePU provides a high-level abstraction model where domain experts have optimized the implementation of common computational patterns. With SkePU 2 we have a reconsidered, next-generation SkePU interface and an updated implementation. With the move to C++11, and improvements such as a native syntax for user functions, extended argument combinations for skeletons, and the introduction of a new Call skeleton for user-defined multi-variant components integrating with powerful SkePU features such as auto-tuning and smart containers, we have extended the number of situations where SkePU can be successfully applied.

9.3 Future Work

In the future, the precompiler role will be expanded to include selection of system-specific user function specializations, guided by a platform description language [39]. The precompiler can either select the most appropriate specialization directly, or include multiple variants and generate logic to select the best one at run-time based on dynamic conditions.

Some features of SkePU 1 has yet to be re-implemented in SkePU 2. This includes the auto-tuner, which will be straightforward to port for existing functionality. We are also considering to extend the auto-tuner to support tuning on a broader set of inputs.

As mentioned in Section 7.3.2, we also have plans for the construction of a higher-level compiler driver.

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. *C++ template metaprogramming: Concepts, tools, and techniques from Boost and beyond*. Pearson Education, 2004.
(Cited on pages 17 and 34.)

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.
(Cited on page 1.)

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.
(Cited on pages 23 and 33.)

[4] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems, Jade Edition*, 2011.
(Cited on page 32.)

[5] Holger Bischof, Sergei Gorlatch, and Roman Leshchinskiy. Generic parallel programming using C++ templates and skeletons. In *Domain-Specific Program Generation*, pages 107–126. Springer, 2004.
(Cited on page 38.)

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55 – 69, 1996.
(Cited on page 34.)

[7] Walter E. Brown. Modern template metaprogramming: A compendium, 2014. CppCon presentation.
(Cited on page 17.)

[8] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.
(Cited on page 10.)

[9] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation–a performance view. *IBM Journal of Research and Development*, 51(5):559–572, Sept 2007.
(Cited on pages 5, 23, and 30.)

[10] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. The Münster skeleton library Muesli - a comprehensive overview, 2009. ERCIS Working Paper No. 7.
(Cited on page 32.)

[11] Murray I. Cole. *Algorithmic skeletons: Structured management of parallel computation*. Pitman and MIT Press, 1989.
(Cited on pages 8 and 12.)

[12] ISO C++ Standards Committee. Working draft, standard for programming language C++. Technical Report N4296, 2014.
(Cited on pages 14 and 15.)

[13] ISO C++ Standards Committee. Working draft, technical specification for C++ extensions for parallelism, revision 1. Technical Report N3960, 2014.
(Cited on page 37.)

[14] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA, USA, 2000.
(Cited on pages 16 and 17.)

[15] 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*, volume 3, pages 140–147. IEEE, 2015.
(Cited on page 37.)

[16] Marco Danelutto and Massimo Torquati. Structured parallel programming with "core" FastFlow. In *Central European Functional Programming School*, volume 8606 of *LNCS*, pages 29–75. Springer, 2015.
(Cited on page 38.)

[17] Usman Dastgeer. *Skeleton Programming for Heterogeneous GPU-based Systems*. Linköping University Electronic Press, 2011.
(Cited on pages 12 and 25.)

[18] 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*, pages 25–32. ACM, 2011.
(Cited on pages 12 and 27.)

[19] Usman Dastgeer and Christoph Kessler. Smart containers and skeleton programming for GPU-based systems. *International Journal of Parallel Programming*, 44(3):506–530, 2016.
(Cited on pages 23 and 24.)

[20] Usman Dastgeer, Lu Li, and Christoph Kessler. The PEPPHER composition tool: performance-aware composition for GPU-based systems. *Computing*, 96(12):1195–1211, 2013.
(Cited on page 50.)

[21] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
(Cited on page 14.)

[22] Robert H Dennard, VL Rideout, E Bassous, and AR Leblanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.
(Cited on page 1.)

[23] Johan Enmyren. A skeleton programming library for multicore CPU and multi-GPU systems. Master's thesis, Linköping University, Linköping, Sweden, 2010. LIU-IDA/LITH-EX-A--10/037--SE.
(Cited on page 23.)

[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*, pages 5–14. ACM, 2010.
(Cited on page 23.)

[25] Steffen Ernsting and Herbert Kuchen. Algorithmic skeletons for multi-core, multi-GPU systems and clusters. *Int. Journal of High Performance Computing and Networking*, 7:129–138, 2012.
(Cited on page 32.)

[26] August Ernstsson, Lu Li, and Christoph Kessler. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. Accepted for HLPP-2016, Münster, Germany, 4–5 July 2016.
(Cited on page 43.)

[27] Joel Falcou, Jocelyn Sérot, Thierry Chateau, and Jean-Thierry Lapresté. QUAFF: efficient C++ design for parallel skeletons. *Parallel Computing*, 32(7):604–615, 2006.
(Cited on page 38.)

[28] Karl-Filip Faxén. Wool-a work stealing library. *SIGARCH Comput. Archit. News*, 36(5):93–100, June 2009.
(Cited on page 34.)

[29] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 100(9):948–960, 1972.
(Cited on page 5.)

[30] Michael J. Flynn and Kevin W. Rudd. Parallel architectures. *ACM Computing Surveys*, 28(1):67–70, 1996.
(Cited on page 6.)

[31] The Association for Computing Machinery. ACM honours computing innovators. `http://awards.acm.org/press_releases/tech_awards_2012.pdf`, 2013. [Press release; accessed 2016-01-27].
(Cited on page 20.)

[32] Message Passing Interface Forum. MPI: A message-passing interface standard, version 3.1. Technical report, 2015.
(Cited on page 9.)

[33] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3: A C++ linear algebra template library. `http://eigen.tuxfamily.org/`. [online; accessed 2016-01-30].
(Cited on page 18.)

[34] Aleksey Gurtovoy and David Abrahams. The Boost C++ metaprogramming library. Technical report, 2002.
(Cited on page 34.)

[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, pages 1–11, Piscataway, NJ, USA, 2014. IEEE Press.
(Cited on pages 15, 21, and 36.)

[36] Mark Harris. CUDPP: CUDA data parallel primitives library. `http://gpgpu.org/developer/cudpp`. [online; accessed 2015-11-15].
(Cited on page 38.)

[37] Jörg Keller, Christoph Kessler, and Jesper Träff. *Practical PRAM programming*. New York: Wiley, 2001.
(Cited on page 6.)

[38] Christoph Kessler. SkePU: Autotunable multi-backend skeleton programming framework for multicore CPU and multi-GPU systems. `http://www.ida.liu.se/labs/pelab/skepu/`. [online; accessed 2016-06-12].
(Cited on page 2.)

[39] 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*, 2015.
(Cited on pages 50 and 72.)

[40] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E Nagel. Scout: a source-to-source transformator for SIMD-optimizations. In *Euro-Par 2011: Parallel Processing Workshops*, pages 137–145. Springer, 2012.
(Cited on pages xii, 35, and 59.)

[41] Chris Lattner. Clang: a C language family frontend for LLVM. `http://clang.llvm.org`. [online; accessed 2016-01-26].
(Cited on page 20.)

[42] Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.
(Cited on page 20.)

[43] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
(Cited on page 20.)

[44] Mario Leyton and José M Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 289–296. IEEE, 2010.
(Cited on page 38.)

[45] Chunhua Liao, Yonghong Yan, Bronis R de Supinski, Daniel J Quinlan, and Barbara Chapman. Early experiences with the OpenMP accelerator model. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 84–98. Springer, 2013.
(Cited on pages 8, 9, and 37.)

[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*, volume LNCS 8097, pages 874–885. Springer, 2013.
(Cited on page 32.)

[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)*, pages 300–307. IEEE, 2011.
(Cited on pages 12, 20, 35, and 58.)

[48] Jens Maurer and Michael Wong. Towards support for attributes in C++ (revision 6). Technical Report N2761, 2008.
(Cited on page 16.)

[49] Eric Niebler. Proto: A compiler construction toolkit for DSELs. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD '07, pages 42–51, New York, NY, USA, 2007. ACM.
(Cited on pages 18 and 34.)

[50] 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, pages 1–10, New York, NY, USA, 2012. ACM.
(Cited on pages xii and 33.)

[51] Nvidia. CUDA parallel computing platform. `http://nvidia.com/cuda`. [online; accessed 2016-01-27].
(Cited on page 10.)

[52] Zoltán Porkoláb, József Mihalicza, and Ádám Sipos. Debugging C++ template metaprograms. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 255–264, New York, NY, USA, 2006. ACM.
(Cited on page 17.)

[53] 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, pages 11:1–11:7, New York, NY, USA, 2015. ACM.
(Cited on page 15.)

[54] Dan Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.
(Cited on page 20.)

[55] Daniel Quinlan, Chunhua Liao, Thomas Panas, Robb Matzke, Markus Schordan, Rich Vuduc, and Qing Yi. ROSE user manual: A tool for building source-to-source translators, draft 0.9.6a. `http://www.rosecompiler.org/ROSE_UserManual/ROSE-0.9.6a-UserManual.pdf`. [online; accessed 2016-01-29].
(Cited on page 20.)

[56] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.
(Cited on page 38.)

[57] Tarik Saidani, Joel Falcou, Claude Tadonki, Lionel Lacassagne, and Daniel Etiemble. Algorithmic skeletons within an embedded domain specific language for the Cell processor. In *PACT'09: 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 67–76. IEEE, 2009.
(Cited on pages xii, 18, 30, and 31.)

[58] Oskar Sjöström. Parallelizing the Edge application for GPU-based systems using the SkePU skeleton programming library. Master's thesis, Linköping University, Linköping, Sweden, 2015. LIU-IDA/LITH-EX-A--15/001--SE.
(Cited on page 28.)

[59] Oskar Sjöström and Christoph Kessler. SkePU user guide. Technical report, 2015.
(Cited on page 24.)

[60] 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 *Proc. ParCo-2015*, September 2015.
(Cited on page 28.)

[61] Michel Steuwer. SkelCL – a skeleton library for heterogeneous systems. `http://skelcl.uni-muenster.de`. [online; accessed 2016-06-14].
(Cited on pages xii and 31.)

[62] Michel Steuwer, Malte Friese, Sebastian Albers, and Sergei Gorlatch. Introducing and implementing the AllPairs skeleton for programming multi-GPU systems. *International Journal of Parallel Programming*, 42(4):601–618, 2013.
(Cited on page 32.)

[63] Michel Steuwer and Sergei Gorlatch. SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems. In Victor Malyshkin, editor, *Parallel Computing Technologies: 12th International Conference, PaCT 2013, St. Petersburg, Russia, September 30 - October 4, 2013*, pages 258–272, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
(Cited on page 30.)

[64] 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.
(Cited on page 30.)

[65] Patricia Sundin. Adaptation of algorithms for underwater sonar data processing to GPU-based systems. Master's thesis, Linköping University, Linköping, Sweden, 2013. LIU-IDA/LITH-EX-A–13/029–SE.
(Cited on page 28.)

[66] Erwin Unruh. Prime number computation. Technical report, ANSI X3J16-94-0075/ISO WG21-462, 1994.
(Cited on page 17.)

[67] V Vassilev, M Vassilev, A Penev, L Moneta, and V Ilieva. Clad — automatic differentiation using Clang and LLVM. *Journal of Physics: Conference Series*, 608(1):012055, 2015.
(Cited on page 36.)

[68] Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.
(Cited on pages 17 and 18.)

[69] Todd Veldhuizen. Template metaprograms. *C++ Report*, 7(4):36–43, 1995.
(Cited on page 17.)

[70] M Mitchell Waldrop. The chips are down for Moore's law. *Nature News*, 530(7589):144–147, 2016.
(Cited on page 1.)

[71] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC—first experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.
(Cited on pages xii, 12, and 14.)

[72] Barry Wilkinson and Michael Allen. *Parallel programming*. Prentice Hall, New Jersey, 1999.
(Cited on page 8.)

[73] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. gpucc: An open-source GPGPU compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 105–116, New York, NY, USA, 2016. ACM.
(Cited on pages 21 and 36.)

[74] Markus Ålind, Mattias V. Eriksson, and Christoph W. Kessler. BlockLib: A skeleton library for Cell Broadband Engine. In *Proceedings of the 1st International Workshop on Multicore Software Engineering*, IWMSE '08, pages 7–14, New York, NY, USA, 2008. ACM.
(Cited on pages 23 and 38.)

Appendix A

Glossary

A.1 Abbreviations

API	Application programming interface
ASIC	Application-specific integrated circuit
AST	Abstract syntax tree
CSR	Compressed sparse row (a sparse matrix storage format)
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
HPC	High-performance computing
IDE	Integrated development environment
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
LLVM	The LLVM Compiler Infrastructure (not an acronym)
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 Terms

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.

(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.

På svenska

In English