Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Computer Science and Engineering
2018 | LIU-IDA/LITH-EX-A--18/019--SE

Auto-tuning Hybrid CPU-GPU Execution of Algorithmic Skeletons in SkePU

Tomas Öhberg

Supervisor : August Ernstsson
Examiner : Christoph Kessler

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

The trend in computer architectures has for several years been heterogeneous systems consisting of a regular CPU and at least one additional, specialized processing unit, such as a GPU. The different characteristics of the processing units and the requirement of multiple tools and programming languages makes programming of such systems a challenging task. Although there exist tools for programming each processing unit, utilizing the full potential of a heterogeneous computer still requires specialized implementations involving multiple frameworks and hand-tuning of parameters. To fully exploit the performance of heterogeneous systems for a single computation, *hybrid execution* is needed, i.e. execution where the workload is distributed between multiple, heterogeneous processing units, working simultaneously on the computation.

This thesis presents the implementation of a new hybrid execution backend in the algorithmic skeleton framework SkePU. The skeleton framework already gives programmers a user-friendly interface to algorithmic templates, executable on different hardware using OpenMP, CUDA and OpenCL. With this extension it is now also possible to divide the computational work of the skeletons between multiple processing units, such as between a CPU and a GPU. The results show an improvement in execution time with the hybrid execution implementation for all skeletons in SkePU. It is also shown that the new implementation results in a lower and more predictable execution time compared to a dynamic scheduling approach based on an earlier implementation of hybrid execution in SkePU.

Acknowledgments

I would like to thank my supervisor August Ernstsson and my examiner Christoph Kessler for their guidance and valuable feedback during this thesis project. Also, a big thank you to Samuel Thibault, University of Bordeaux, for answering my questions on StarPU and for the assistance with the reintegration.

I would also like to thank NSC[1], the National Supercomputer Centre, for providing valuable accelerator equipped computing resources. Without them this thesis project would not have been possible to accomplish.

Thank you to my fellow master's students—especially Eric, Edward and Sara—for your company and encouragement during the project and for occasionally making me get away from the office desk. I will always be amazed by how simple a problem can appear after a short break!

Finally, I am grateful to my family for their support and patience throughout my studies. I would not be where I am today without it.

Tomas Öhberg
Linköping, June 2018

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

Contents

List of Figures

List of Tables

Listings

1 Introduction

This chapter aims at giving the reader an introduction to this thesis project. It starts with a motivation to the problem in Section 1.1, also providing a brief introduction to the topic. Then the aim of this thesis is presented in Section 1.2, followed by the research questions in Section 1.3, defining the scope of this thesis. The delimitations are given in Section 1.4, presenting some topics outside the scope of this thesis. Finally, an overview of the structure for the rest of the thesis is given in Section 1.5.

1.1 Motivation

For a number of years now, the trend in computer architectures has been to use heterogeneous multi-processor systems. Today most computers contain at least one additional processing unit (PU), such as a graphics processor, apart from the CPU (Central Processing Unit) and research suggests this trend will continue in the future [6, 52]. Looking at the fastest super-computers in the world, 102 out of the 500 systems on the TOP500 list as of November 2017 are heterogeneous [45]. Although the idea of heterogeneous architectures is not new, we still face the challenge of implementing efficient algorithms that utilize the full potential of such systems. The task is not made easier by the variation in number of processing units and their relative performance in different computers.

One concept that attempts to facilitate parallel programming is *skeleton programming*. Skeletons are algorithmic templates—building blocks—specifying the structure of computations. Programmers select an appropriate skeleton and specialize it with so called *user functions*, to solve the problem at hand. One example of a skeleton is the *Map* skeleton. Map applies a single user function element-wise to the elements of an array, producing a new array as output. Instantiating the Map skeleton with a user function returning the squared value of the input argument, would result in a skeleton instance returning a new array containing the values of the input array squared. The separation between the implementation and the underlying computational flow enables skeletons to be implemented and optimized for different parallel architectures by experts, while still being easy and flexible to use for a programmer not familiar with parallel programming. The programmer is only required to understand *what* the skeleton does, not *how* [7].

1.1.1 SkePU

To support development of applications for heterogeneous architectures, the skeleton programming framework *SkePU* implements its skeletons for several different hardware *backends*. Without increasing the burden on the programmer, a skeleton in SkePU can be executed either sequentially or in parallel on a CPU, or on an *accelerator*, such as a *GPU* (Graphics Processing Unit) or a *MIC* (Many Integrated Cores, i.e. Intel's Xeon Phi co-processors). The user functions are defined by the programmer as regular C++ functions and are translated to the supported backends by SkePU's precompiler. At runtime the programmer can choose which backend to use when executing the skeleton, or construct an *execution plan* to let SkePU automatically switch backend depending on the problem size.

An earlier version of SkePU came in two distributions: one standard variant and one integrated with the task based runtime scheduler *StarPU*. Tasks in StarPU can be implemented to execute on different processing units. By splitting the workload of the SkePU skeletons into multiple tasks, StarPU could perform load balancing at runtime. StarPU also has support for running tasks on different PUs simultaneously. This allowed for *hybrid execution* in SkePU, i.e. to divide the workload of a single skeleton and simultaneously let the CPU and a GPU work on the computation. Performance statistics gathered from earlier executions of the skeleton on the machine was used to make predictions of how to distribute the work between the PUs. This solution, however, had one major drawback: SkePU needed to perform several warm-up runs for the runtime system to gather enough execution data on the machine to do accurate predictions. As a result, the initial executions of a program on a specific machine would be slow and the performance of an application unreliable [11].

The current version of SkePU supports a limited work partitioning scheme when multiple accelerators are used to work in parallel on a skeleton. The implementation divides the work equally between the accelerators and will thus work well when all accelerators are of the same model, but might result in load imbalance when not. However, the current version of SkePU lacks support for hybrid execution, meaning that the work of a skeleton cannot be divided between the CPU and accelerators.

1.2 Aim

Given the increasing need for frameworks with support for heterogeneous computing and automatic workload distribution, the aim of this thesis was to implement a hybrid execution backend in the latest version of SkePU. The backend was supposed to partition the work between a multi-core CPU and any number of GPUs or other accelerators. The benefits of such an implementation was three-fold. It would (1) result in an easy-to-use framework for heterogeneous computing, (2) improve performance of already written SkePU-programs and (3) make SkePU able to fully utilize the potential of modern, heterogeneous computer systems. An important aspect of the new hybrid execution implementation was to elude the limitations of the experimental implementation in SkePU 1. This included finding a way to make more reliable predictions at the first execution of a skeleton instance and thus eliminate the need for warm-up runs, as well as to reduce the overhead caused by the dynamic scheduling.

1.3 Research Questions

This thesis will discuss and try to answer the following three research questions:

1. How can the workload of the skeletons in SkePU be partitioned for execution on heterogeneous processing units?

2. How can the optimal workload partitioning in the new hybrid execution backend be predicted for different types of processing units?

3. How can the overhead of the hybrid execution implementation be kept low and without the need for warm-up runs?

1.4 Delimitations

The hybrid execution implementation presented in this thesis only considers execution of individual skeleton calls. A more efficient implementation would examine a number of consecutive skeleton calls, using for instance a data dependency graph. This would give the runtime system ability to exploit the extra information on where the data is currently located, to take the data transfer between CPU memory and other PUs' memory into account when predicting the partitioning. For example the system could prefer keeping the same partitioning for consecutive skeleton calls to prevent data transfers between processing units.

A delimitation was also made to the requirements for the automatic partitioning prediction tuning. As the SkePU framework allows for very complex usage of the skeletons, the tuner was delimited to only work well for skeleton instances where the execution time was bound by the size of the element-wise input.

1.5 Report Structure

The rest of this thesis is structured as follows. First the background needed to understand this thesis is covered by Chapter 2. Chapter 3 introduces the skeleton programming framework SkePU and its available skeletons. In Chapter 4 some related heterogeneous computing frameworks and libraries are introduced. The implementation of the hybrid backend and the auto-tuner is presented in Chapter 5. Chapter 6 presents how the new implementation was evaluated, followed by Chapter 7 where the results of the evaluation are presented. The results and the method applied by this thesis is then discussed in Chapter 8, followed by Chapter 9 with some topics left for future work. Finally the conclusions are presented in Chapter 10.

2 Background

This chapter presents the background required to understand the rest of this thesis. The chapter first gives some important definitions and terms in Section 2.1 and then gives a brief introduction to multi-core and accelerator architectures in Section 2.2. Load balancing is then discussed in Section 2.3. An introduction to skeleton programming is presented in Section 2.4 and finally some of the more common parallel programming frameworks and tools are presented in Section 2.5.

2.1 Definitions

This section defines some important terms and describes how they should be interpreted in the context of this thesis.

Accelerator: A processing unit specialized on a specific type of computations is called an *accelerator*. Accelerators include GPUs, MICs, *ASIC*s (Application-Specific Integrated Circuit) and *FPGA*s (Field-Programmable Gate Array).

Backend: In this thesis a *backend* is an implementation variant of the (SkePU) skeletons for specific hardware. Multiple backends based on different programming languages or frameworks can also target the same hardware. There are for instance two backends in SkePU supporting NVIDIA GPUs.

Discrete architecture: A *discrete architecture* is a computer architecture where the CPU and the accelerator are located on different chips; each with their own memory module. The memory modules are connected to each other via a bus connection. The opposite of a discrete architecture is a fused architecture, see below.

Fused architecture: A *fused architecture* is a computer architecture where the CPU and the accelerator (typically a GPU) share the same chip and are connected to the same memory. The work in this thesis will focus on discrete architectures, but fused architectures are mentioned for completeness.

Heterogeneous computing: The term *heterogeneous computing* is used as a general term for computations where multiple, heterogeneous processing units are used in conjunction.

Hybrid execution: The term *hybrid execution* means that an algorithm is executed in parallel on at least two, heterogeneous processing units at the same time. The term is used both if the processing units are executing the same instructions on different data and if they are executing different parts of the algorithm, passing data between each other in a pipeline fashion.

Many Integrated Cores (MIC): Intel's name for their Xeon Phi series of accelerators (sometimes referred to as co-processors), based on the x86 architecture [16].

Processing unit (PU): In this thesis the term *processing unit* is used as a generic term for a processor chip, including CPUs, GPUs and other accelerators. This is to avoid the confusion the term *device* is causing, as suggested by Mittal and Vetter [35]. Other equivalent terms used in the literature include *computing unit*, *computing element* and *processing element*.

Skeleton instance: A *skeleton instance* is a skeleton specialized with the required *user functions*.

SkePU: A skeleton programming framework. In this thesis SkePU refers to SkePU 2, the second major version of SkePU, unless otherwise stated. See Section 3 for a more thorough description of SkePU.

2.2 Parallel Computer Architectures

In this section some important architectural differences between CPUs and other accelerators (in particular GPUs) are presented. The aim of this section is to convince the reader that different hardware call for vastly different programming approaches and that a broad knowledge base is required from the programmer to get good performance out of a heterogeneous system.

2.2.1 Shared Memory CPU Programming

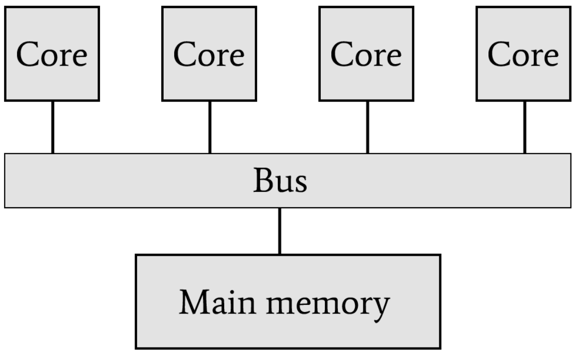


Figure 2.1: An outline of a shared memory system with four processor cores.

Today the CPUs in workstations, clusters, laptops and even cell phones have multiple cores, sharing one single memory module. This type of arrangement where multiple cores share the same physical memory space is called a *shared memory* system. Each core can perform its computations independently of the others, but might communicate through the common memory module with another core. The processor cores are connected to the memory module through a bus connection, as shown in Figure 2.1. Because the computation speed of the processor cores is much higher than the speed and bandwidth of the memory, the memory bus is usually the bottleneck in modern shared memory systems. To reduce this problem, smaller, but faster *cache memories* are used between the main memory and the processor cores. Caches store partial copies of the main memory for faster access of frequently used memory parts. The data is stored in, and transferred to the cache as *cache lines* of a fixed size. Modern processors have a multiple level cache hierarchy, where some of the caches are shared among all cores, while others are local to one single core. Memory consistency protocols are used to keep the local caches synchronized, to ensure a core is not reading locally cached memory that has been modified by another core.

One major disadvantage of the arrangement with caches is *false sharing*. It occurs when multiple cores are frequently accessing adjacent memory cells in main memory; memory cells that are stored in the same cache line. When a part of a cache line is changed, the entire cache line is invalidated by the cache consistency protocol. As a result, noticeable performance overheads will arise when two cores are frequently accessing memory in the same cache line. Even though the changes are made to distinct memory cells, the local caches must be constantly updated to be kept consistent with the other core. To fully utilize a multi-core architecture with caches, the memory access pattern must be carefully considered. Each core should preferably access memory in order, or make many repeated accesses to a few memory cells to exploit the fast, already cached parts of the memory. Two separate cores should prevent making frequent accesses to adjacent memory cells, to prevent false sharing [51]. Apart from a complex cache hierarchy, modern CPUs use large silicon areas to features giving good performance when executing irregular programs with a complex control flow, such as programs containing loops and branches.

2.2.2 Accelerator Programming

Accelerators are PUs specialized at a specific computational task. As the name suggests, an accelerator is supposed to accelerate the execution of the tasks they are specifically designed for. Most accelerators are made to perform well at *data-parallel* tasks, i.e. where each core of the processor can work independently on its piece of data. Computers with an accelerator can either be arranged as a *discrete* or a *fused* architecture. In the first case, the accelerator is located on its own chip, detached from the CPU, with its own memory module. The accelerator memory is connected to the main memory through a bus connection. In the second case, the accelerator chip is fused with the CPU chip, sharing (a part of) the main memory. As discrete architecture accelerators are only able to read their own memory, data must be explicitly copied from main memory to accelerator memory before the computation. When the computation is done on the accelerator the result must be copied back to main memory. Memory transfers usually take up a large proportion of the total execution time for accelerator executions. In fused architectures on the other hand, memory copying is generally not needed as both PUs can access the main memory.

Programming of accelerators calls for specialized tools and languages. The programs executing on an accelerator—called *kernels*—are generally short functions offloaded to the accelerator to take advantage of the parallelism. The kernels are generally launched by a sequential program running on the CPU. In some cases, the kernels themselves can launch other kernels to create *nested* kernel calls. Accelerator kernels are either written by a programmer in special programming languages or generated by tools such as compilers. Examples of both are introduced in Section 2.5.

The most common type of accelerator is the graphical processing unit (GPU). GPUs were originally designed with a fixed pipeline for drawing graphics on a screen, but have since evolved to allow general purpose programming. A modern GPU can consist of thousands of simple processor cores arranged into groups, called *Streaming Multiprocessors* (SM)[1] [9].

GPU kernels are launched as a number of threads. Threads are executed in groups of 32, called *warps*[2]; each warp being executed on a single SM. When a kernel is launched on a GPU,

[1] *Streaming Multiprocessor* is CUDA terminology, OpenCL uses the term *Compute Units*.
[2] *Warp* is CUDA terminology, no equivalent term exists in OpenCL.

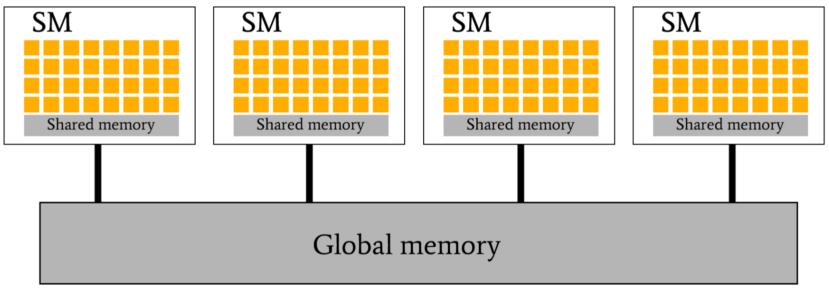


Figure 2.2: An outline of a GPU architecture with four Streaming Multiprocessors (SM).

the host CPU schedules a number of *blocks* of threads[3] to be executed on the GPU. All blocks are distributed between the SMs. When an SM receives a block it partitions the threads of the block into warps to be executed by the cores in the SM. The number of warps that can be executed simultaneously by an SM varies between GPU models. GPUs make use of massive thread parallelism to hide memory latencies. When a warp of threads is stuck waiting for resources such as memory in an SM, another warp of waiting threads will be swapped in and executed in the meantime. Thread switching is performed in hardware with no performance overhead. Hence, to fully utilize a GPU, a kernel should launch many more threads than there are cores. This is the opposite of the case in CPUs where thread switching is usually an expensive operation, implemented in software [9].

All cores executing a warp must execute the same instruction of the kernel simultaneously. In case of a branch in the kernel code taken by at least one thread, all other threads in the warp must still follow the instruction flow of the branch. Threads not actually taking the branch are marked as inactive and the corresponding cores in the SM will thus go idle and waste performance potential until the branch is passed. This feature of the hardware will make GPUs lose performance on kernels with a high number of branches and irregular loops. If a warp of threads hits a branch in the program, the entire warp must follow, even if only as little as one thread might actually do some work. The same thing applies for loops: even if some threads will break out of the loop, the entire warp must keep following the loop instruction flow until all threads in the warp are done with their iterations. Best utilization of the hardware is achieved if kernels have few or regular branches, and loops with the same number of iterations for all threads [9].

GPUs have multiple memory types. The two most important ones are the *global* and *shared* memories. Data moved to and from the CPU is stored in the largest memory module, the *global memory*. This memory is accessible by all SMs. Each SM also has a smaller and faster memory called *shared memory*,[4] accessible only by the SM. Shared memory is not automatically used to speed up memory accesses as in the case of caches. Instead, the programmer is responsible for making use of it. A simplified GPU architecture with the global and shared memory modules are shown in Figure 2.2. The illustration shows four SMs, each containing 32 cores (i.e. one single warp can be executed simultaneously on each SM), colored orange in the illustration. To speed up global memory accesses, GPUs make use of a feature called *coalesced memory access*. When consecutive threads in a warp access consecutive memory cells, the memory management system utilizes the wide bandwidth to let the threads share a single memory transaction, which saves transfer time. This is the opposite of CPUs where this memory access pattern would result in false sharing [9].

[3] *Blocks* and *threads* are CUDA terminology, OpenCL uses the terms *work-group* and *work-item*, respectively.
[4] *Shared memory* is CUDA terminology, OpenCL uses the term *local memory*.

It is apparent that GPU programming, and accelerator programming in general, requires a different approach compared to regular CPU programming. To fully utilize an accelerator equipped system, expert knowledge for the particular accelerator type is needed.

2.3 Load balancing

To take advantage of multiple computational resources, the workload must be distributed between them in some way. For the shortest possible execution time of an algorithm, the optimal distribution is when all units finish their partition of the work at the same time. The workload is then well balanced between the units.

Load balancing strategies are divided into static and dynamic ones. Static strategies only make use of the earlier collected performance statistics to predict the best work distribution and then keep this distribution throughout the execution of the algorithm. Dynamic strategies take the current state of the machine into consideration and can adapt to changes during the execution by actively redistributing the remaining workload. Static strategies have the advantage of simplicity and low overhead, but cannot adapt to changes in the workload caused by e.g. other processes running on the same hardware [54].

Load balancing can be applied on different hardware levels. For multi-core machines, where the workload is distributed between CPU cores, it can be discussed on a core level. In computer clusters load balancing can be implemented on node level, between the nodes in the cluster. In this thesis, however, the load balancing will be considered on PU level, within a single compute node, where the workload is distributed between a CPU and an accelerator.

The goal of load balancing is to find the optimal workload distribution between the units to get a shorter execution time. If the load balancing algorithm makes a misprediction some of the units will go idle while others are finishing their part of the workload, thus wasting potential performance improvements. When performing load balancing on PU level additional things must be considered. One major point is the data transfer times for the workload executed on an accelerator. These times can take up a large proportion of the overall execution time for that PU. However, in the case of the data already residing and being up-to-date in accelerator memory, no data transfers are needed and the overall execution time of the accelerator will be reduced. In addition to this, two algorithms will perform very differently on a PU depending on how well the algorithm fits the computation model of it. Some algorithms will run faster on the CPU compared to the accelerator, while other algorithms will execute faster on the accelerator. In core level load balancing this is not a problem, as the execution time is generally the same on all cores of the CPU.

2.4 Skeleton Programming

Several parallel programming models have been proposed over the years, with the intention to more or less hide the complexity of the underlying parallelism and to facilitate for the programmer. In 1989, Cole [7] suggested to use *higher-order functions* from the functional programming paradigm as a parallel programming model. A higher-order function is a function taking other functions as arguments. The argument functions are then applied by the higher-order function, usually to some sequential data. Cole introduced the idea that parallelizable higher-order functions could be used as *algorithmic skeletons*. Thus, the skeletons provide templates of the computational structure, and let the programmer apply the templates to solve the problem at hand. At the same time the skeletons can internally be implemented and optimized for some parallel architecture. The programmer must not only choose an appropriate skeleton, but also instantiate it with the correct argument functions. In skeleton programming these function arguments are usually called *user functions*.

From the skeleton programmer's perspective, a skeleton can be seen as a sequential computation, but it may in fact hide a complex parallel implementation underneath. This separates the computational structure from the actual implementation. The parallelization can be implemented in an efficient way by a system expert, while the users of the skeleton do not need to know anything about parallelism to use it. The only requirement is to understand the general structure of computation provided by the skeleton.

Algorithmic skeletons can be further divided into *task-parallel* and *data-parallel* skeletons. The *data-parallel* skeletons are higher-order functions consisting of many independent computations, in general applied to the elements of an array. Typical examples of data-parallel skeletons are *Map*, which applies the user function element-wise to a number of input arrays to produce an new array, and *Scan*, which produces an array of the prefix sums of an input array. Both these skeletons are included in SkePU and are described in detail in Section 3.1. As the computations of data-parallel skeletons are independent, synchronization is generally not needed between processing cores when this type of skeletons are implemented. Data-parallel skeletons are therefore well suited to be distributed over multiple PUs, as there is generally no need for expensive inter-PU communication [31].

In contrast, there are dependencies between computations in *task-parallel* skeletons. Two examples of task-parallel skeletons are *Pipeline* and *Farm*. A Pipeline connects a number of computation stages. Each stage will process a work item and pass the result to the next stage. This means that a stage cannot start processing a work item before the computation in the previous stage is done, creating a dependency between the stages. The other task-parallel skeleton, Farm, can be used as a load balancing scheduler. A farmer accepts a queue of tasks that need to be processed. The tasks are then distributed to a number of workers (processor cores). Task-parallel skeletons can encapsulate data-parallelism. As an example, the stages in the Pipeline skeleton could internally be data-parallel [30].

2.5 Parallel Programming Frameworks

There are several programming languages, libraries and other tools to aid programmers in implementation of parallel programs. Some of them are specialized on a specific type of architecture or brand of hardware, while some of them are more general. The most widely adopted ones are presented in this section.

2.5.1 OpenMP

OpenMP (Open Multi-Processing) is an *API* (Application Programming Interface) available for C/C++ and Fortran, which defines a simple, yet flexible, way of writing parallel applications. For C/C++ this is accomplished by the use of `#pragma` preprocessor directives. The programmer uses these `#pragma` directives to tell the compiler which parts of the program to parallelize. Setting up threads and partitioning the workload is taken care of by the compiler. Well written OpenMP annotated code has the advantage that compilers not supporting the OpenMP standard will automatically ignore the `#pragma` directives and compile a working, sequential program. In OpenMP version 4.0 support for offloading computations to accelerators were introduced. However, this feature of OpenMP has not been used in SkePU [40].

An example of calculating the dot product by loop parallelizing and reduction using OpenMP is shown in Listing 2.1.

2.5.2 TBB

TBB (Threading Building Blocks) is a multi-core programming library developed by Intel. The library provides a number of parallelized C++ building blocks and synchronizations primitives. Some of the building blocks resembles algorithmic skeletons. TBB also includes support for building dependency and data flow graphs and execute them using a scheduler with multiple scheduling strategies [28].

```cpp
#include <vector>
#include <omp.h>

const int NUM_THREADS = 4;

float dot_product(std::vector<float> in1, std::vector<float> in2) {
    int size = std::min(in1.size(), in2.size());
    std::vector<float> tmp(size);

    #pragma omp parallel for num_threads(NUM_THREADS)
    for(int i = 0; i < size; ++i) {
        tmp[i] = in1[i] * in2[i];
    }

    float res = 0;
    #pragma omp parallel for num_threads(NUM_THREADS) reduction(+:res)
    for(int i = 0; i < size; ++i) {
        res += tmp[i];
    }

    return res;
}
```

Listing 2.1: Example of multi-threaded dot product using OpenMP in C++.

```cpp
#include "tbb/task_group.h"

using namespace tbb;

int Fib(int n) {
    if(n < 2) {
        return n;
    } else {
        int x, y;
        task_group g;
        g.run([&]{ x = Fib(n-1); }); // spawn a task
        g.run([&]{ y = Fib(n-2); }); // spawn another task
        g.wait();                    // wait for both tasks to complete
        return x + y;
    }
}
```

Listing 2.2: Example of recursively computing the n:th Fibonacci number using TBB [28].

An example of computing the n:th Fibonacci number by using the task scheduler in TBB is shown in Listing 2.2.

2.5.3 MPI

MPI (Message Passing Interface) is a message-passing library interface specification with several implementations. Language bindings to C and Fortran are part of the standard. The specification's main purpose is to provide an API for message-passing between processes with different address spaces. This is typically used for communication between nodes in distributed memory systems, such as computer clusters, but can also be used for interprocess communication in a single CPU [36].

An example of passing a message from one node to another using C and MPI is shown in Listing 2.3. The same program is launched on several nodes by MPI and each process is given an identifier called *rank*.

```c
#include "mpi.h"

int main( int argc, char *argv[])
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0)    /* code for process zero */
    {
        strcpy(message,"Hello, there");
        MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else if (myrank == 1)  /* code for process one */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
    return 0;
}
```

Listing 2.3: Example of message passing between two nodes using MPI [36].

2.5.4 CUDA

CUDA is NVIDIA's general purpose GPU computing platform and toolkit. It allows programmers to implement computation kernels in a C++ based language and execute them on the GPU. CUDA is only supported on NVIDIA GPUs, reducing its portability, although there exist tools to transform CUDA programs to the more portable language OpenCL [24]. The CUDA toolkit also includes a number of GPU-accelerated libraries with algorithms from areas such as deep learning, linear algebra and signal processing [37]. An example CUDA program is shown in Listing 2.4, printing *Hello World* by making a nested CUDA kernel call. The `__global__` keywords indicate kernel functions, to be executed on a CUDA enabled GPU.

2.5.5 OpenCL

OpenCL (Open Compute Language) is an open standard for parallel processing that is maintained by the Khronos Group. Programs using OpenCL define a computation kernel in the OpenCL programming language, a subset of C++. In contrast to CUDA where kernels are in-lined in regular C/C++ code, kernels in OpenCL are written as strings and compiled by the hardware drivers on the target machine at run-time. OpenCL is supported on a wide verity of PUs, including Intel and AMD CPUs; Intel, AMD and NVIDIA GPUs and Intel Xeon Phi MICs [23].

An example C++ program using OpenCL is shown in Listing 2.5. The program prints *Hello World* from a single kernel running on a GPU. Note that the kernel function is written as a string literal.

2.5.6 OpenACC

OpenACC (Open Accelerators) is a high-level programming model aimed at parallelizing code for accelerator equipped systems. OpenACC resembles OpenMP in style, as it uses `#pragma` directives and compiler support to generate code for different hardwares. The standard supports multiple types of hardware, including CPUs, GPUs and MICs [48]. However, some research suggests the raised abstraction level comes with a price of lower performance compared to manually written and optimized GPU code [26, 34]. An example of Jacobi iteration in OpenACC is shown in Listing 2.6.

```
#include <stdio.h>

__global__ void childKernel() {
    printf("Hello ");
}

__global__ void parentKernel() {
    // launch child
    childKernel<<<1,1>>>();
    if (cudaSuccess != cudaGetLastError()) {
        return;
    }
    // wait for child to complete
    if (cudaSuccess != cudaDeviceSynchronize()) {
        return;
    }
    printf("World!\n");
}

int main(int argc, char *argv[]) {
    // launch parent
    parentKernel<<<1,1>>>();
    if (cudaSuccess != cudaGetLastError()) {
        return 1;
    }
    // wait for parent to complete
    if (cudaSuccess != cudaDeviceSynchronize()) {
        return 2;
    }
    return 0;
}
```

Listing 2.4: Example of Hello World in CUDA [9].

2.5.7 Other Frameworks

There exist many other libraries and tools for parallel programming. Some of them worth to mention are:

- *OpenHMPP* (Open Hybrid Multi-core Parallel Programming) [15], a programming model for heterogeneous systems using `#pragma` directives, similar to OpenMP and OpenACC.
- *Cilk*[5], a family of C/C++ based languages for multi-thread computations on the CPU, developed by Intel.
- *C++ AMP* (Accelerated Massive Parallelism)[6], a programming model by Microsoft with skeleton-like constructs to offload C++ code to data-parallel accelerators.
- *SYCL* (pronounced ”sickle”) is a C++ abstraction layer that tightly integrates OpenCL code in regular C++ programs, to provide better type support and ease of use for heterogeneous programming [47].
- The recent C++ standard *C++17*, which adds experimental support for parallel implementations of standard library algorithms, including Map and Reduce skeletons [29].
- *Compute shaders*, high-level, graphics oriented GPU-programming languages available through APIs such as OpenGL, Vulkan and DirectX.

[5]Cilk: `https://www.cilkplus.org/cilk-history`
[6]C++ AMP: `https://msdn.microsoft.com/en-us/library/hh265136.aspx`

```
#include <CL/cl.hpp>
#include <vector>
#include <iostream>

const std::string kernelSrc = "\n" \
    "__kernel void helloKernel() {        \n" \
    "   printf(\"Hello World\");          \n" \
    "}                                    \n" \
    "\n";


int main(void) {
    std::vector<cl::Platform> platforms;
    std::vector<cl::Device> gpus;

    // Find a GPU to execute on
    cl::Platform::get(&platforms);
    platforms[0].getDevices(CL_DEVICE_TYPE_GPU, &gpus);
    cl::Context context(gpus[0]);

    // Create a OpenCL program from the source code
    cl::Program::Sources sources(1, std::make_pair(kernelSrc.c_str(), kernelSrc.
        length()+1));
    cl::Program program(context, sources);

    // Compile the OpenCL kernel
    cl_int err = program.build();
    cl::Kernel kernel(program, "helloKernel", &err);
    if(err != CL_SUCCESS) {
        std::cout << "A error occured." << std::endl;
        return 1;
    }

    // Enqueue the kernel launch to the command queue of the GPU
    cl::CommandQueue queue(context, gpus[0]);
    queue.enqueueTask(kernel);

    return 0;
}
```

Listing 2.5: Example of Hello World in OpenCL.

```c
while(error > tol && iter < iter_max)
{
    error = 0.0;
    #pragma acc parallel loop reduction(max:error)
    for(int j = 1; j < n-1; j++)
    {
        #pragma acc loop reduction(max:error)
        for(int i = 1; i < m-1; i++)
        {
            A[j][i] = 0.25 * ( Anew[j][i+1] + Anew[j][i-1]
                    + Anew[j-1][i] + Anew[j+1][i]);
            error = fmax( error, fabs(A[j][i] - Anew[j][i]));
        }
    }
    #pragma acc parallel loop
    for(int j = 1; j < n-1; j++)
    {
        #pragma acc loop
        for (int i = 1; i < m-1; i++)
        {
            A[j][i] = Anew[j][i];
        }
    }
    if(iter % 100 == 0)
        printf("%5d, %0.6f\n", iter, error);
    iter++;
}
```

Listing 2.6: Example of Jacobi iteration using OpenACC [39].

3 SkePU 2

The skeleton programing framework *SkePU* (Skeleton Processing Unit)[1] is an open source research project, developed at Linköping University. SkePU was first presented in 2010 by Enmyren and Kessler [17, 18] as a macro based skeleton library written in C++. By implementing the skeletons for different *backends* support for a variety of hardware are enabled, including execution on multi-core CPUs and GPUs. SkePU has been used to parallelize multiple industry-class applications, including computational fluid dynamics flow solver EDGE [42] and an underwater acoustics simulation tool [46].

The initial implementation of SkePU had several limitations to its design. Some of the skeletons had constraints to the number of input arguments, as every combination of input arguments required a separate macro. This led to code duplication and a high maintenance overhead. The use of preprocessor macros also made the implementation lack type-safety, resulting in many type errors only being detected during run-time.

SkePU was redesigned in 2016 by Ernstsson [20] to take advantage of modern C++11 features and become type-safe. The new version is called *SkePU 2* and the rest of this thesis refers to this version unless otherwise stated. The new version of SkePU makes use of template meta-programming to reduce code duplication and to make the library more general. SkePU 2 allows any number of input arguments to the skeletons for example. The original skeletons were redesigned and generalized. The current version of SkePU includes five data-parallel skeletons and one task-parallel. Some of the skeletons have variants for one and two-dimensional data. All skeletons are described in detail in Section 3.1.

SkePU has since its first version supported multiple heterogeneous architectures by implementing all skeleton types for four different backends. These include two backends for CPU (a sequential one and a parallelized implementation using OpenMP), a CUDA backend available for NVIDIA GPUs and an OpenCL backend for any supported accelerator[2], including GPUs from AMD and NVIDIA, and MICs (i.e. Intel's Xeon Phi).

As a replacement for the preprocessor macros, SkePU 2 makes use of a precompiler to perform source-to-source translation of the user functions for the different backends. The precompiler identifies all user functions and expands them to a C++ struct, containing the implementations for the different backends. The user can during this precompilation step

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

[2]OpenCL can also execute on CPUs, but SkePU's OpenCL backend is optimized for accelerators.

choose which backends to generate implementations for. The result of the precompilation step is a new source file which can be fed to a regular compiler. SkePU 2 is also designed to compile a valid sequential program in case the precompiler is not invoked on the input files first, similar to how OpenMP produces a sequential program if compiled using a compiler without OpenMP support.

3.1 Skeletons in SkePU

This section describes the six skeletons available in SkePU 2 in detail. These are Map, Reduce, MapReduce, MapOverlap, Scan and Call.

3.1.1 Map

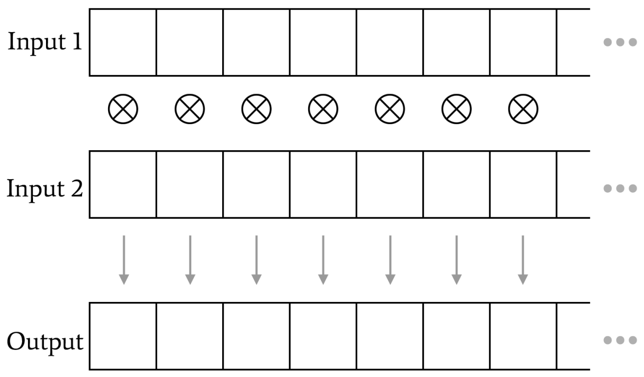


Figure 3.1: An example of the Map skeleton with two input arrays.

The skeleton *Map* applies the user function element-wise to all elements of an array, to produce a new array. The most simple example would be scaling all elements of an array by a constant factor. In that case, the user function provided to the Map skeleton would be a function with a single input argument, returning that argument multiplied by the constant factor. In SkePU the Map skeleton can also be used with any number of input arrays, including no input arrays. As an output array must always be provided, this array determines the number of element-wise operations in case the skeleton instance takes no input arrays. The argument count of the user function must always match the number of input arrays and all input arrays must be of equal size. More formally, element i of the output array is the result of a call to the user function with element i of each input array as arguments. In some literature Map is defined as a skeleton with a single input array while Map with two input arrays is referred to as *Zip*. In SkePU the Map skeleton also supports matrices. The Map skeleton is a good candidate for parallel implementation, as all computations (i.e. all computed elements in the result array) are independent of one another [7].

An example of the Map skeleton is shown in Figure 3.1, where a binary user function, denoted $\otimes$, is applied element-wise to two arrays to produce a result array.

3.1.2 Reduce

The skeleton *Reduce* uses a binary user function to aggregate the elements of one array. Usually this is seen as a sequential computation where the binary user function is first applied to the two foremost elements of the array, then to the result of the first computation and the third element, and so on. To take advantage of parallel processing, the operations must be reordered, usually as a symmetric reduction tree. The reordering of computations assumes that the binary user function is commutative and associative. A typical example usage of reduce would be

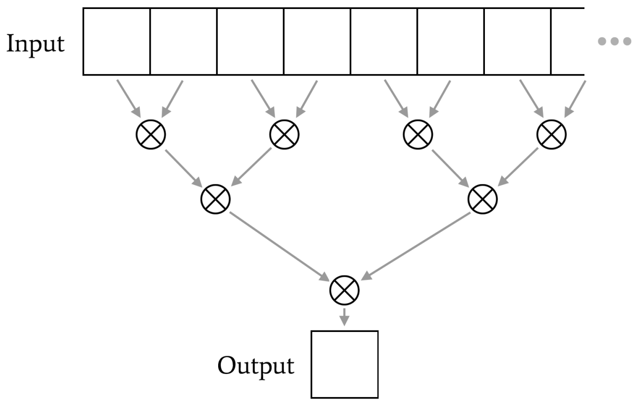


Figure 3.2: An example of the Reduce skeleton.

to sum up the numbers of an array, in which case the user function would be an addition function.

Reduce is available in two variations: one-dimensional and two-dimensional. The one-dimensional variant takes either an array or a matrix as input and reduces it in one dimension. The result is a scalar or an array respectively. When a matrix is used, the reduction direction can be specified as either row-wise or column-wise. Two-dimensional reduction can be seen as two one-dimensional reductions applied after one another. Only matrices can be used as input. The reduction is first performed in one direction and results in an array. The array is then reduced to produce the final scalar. In two-dimensional Reduce, the direction of the first reduction can be specified and different user functions might be used for the two directions. It is also possible to set an initial value of the reduction.

An example of a one-dimensional Reduce on an array is shown in Figure 3.2, where a binary user function, denoted $\otimes$, reduces the array to a single value. The figure shows Reduce implemented as a symmetric reduction tree.

3.1.3 MapReduce

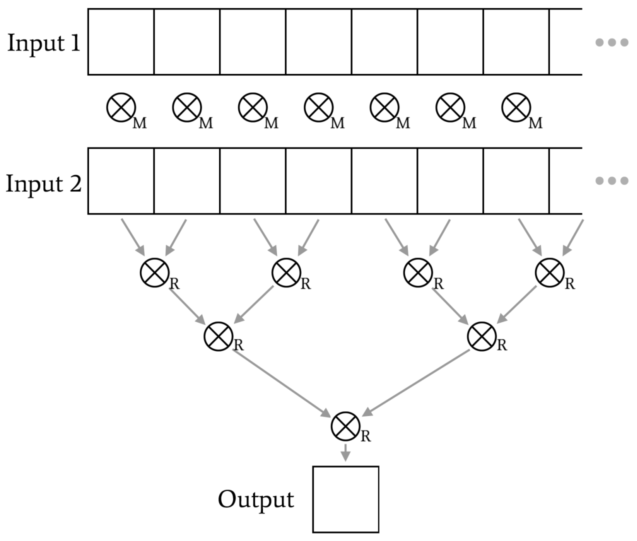


Figure 3.3: An example of the MapReduce skeleton with two input arrays.

The Map and Reduce skeletons are often combined into one skeleton called *MapReduce*. The skeleton resembles the more general MapReduce programming model [14] used by e.g. Google and Apache Hadoop. The programming model MapReduce is a highly scalable framework for processing large data sets on computer clusters, usable in many different areas of computer science, such as data mining and machine learning. The skeleton MapReduce lacks some of the features of the programming model, such as the sorting step performed after the Map step to redistribute the data between the nodes of the cluster. The skeleton is therefore less scalable, but nevertheless highly parallelizable.

The MapReduce skeleton accepts two user functions, one for the Map part and one for the Reduce part. In the first step, Map is performed on a number of input arrays using the Map user function, producing a temporary array. In the second step, Reduce is performed on the temporary array using the Reduce user function, resulting in a single value. In practice the Map and Reduce steps are performed interleaved to avoid the need to allocate the extra intermediate array.

An example of the MapReduce skeleton is shown in Figure 3.3. The two user functions are denoted $\otimes_M$ and $\otimes_R$ in the figure. In the example two arrays are combined in the Map step by applying the user function $\otimes_M$. The result of this operation is then reduced in the Reduce step using the user function $\otimes_R$.

In contrast to the Reduce skeleton, MapReduce in SkePU only accepts one-dimensional arrays as arguments.

3.1.4 MapOverlap

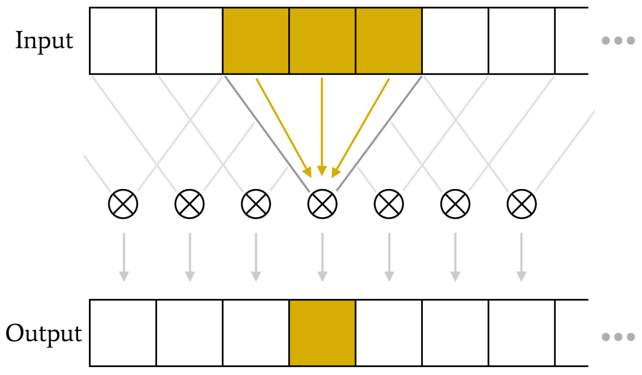


Figure 3.4: An example of the MapOverlap skeleton with a total width of three elements.

The *MapOverlap* skeleton is a stencil operation and can be seen as a generalized variant of the Map skeleton. The skeleton takes either an array or a matrix as input and produces a output container of the same type and dimensions. The difference to the Map skeleton is that the user function in MapOverlap has access to a region of elements. In SkePU the region is called overlap, but it is often referred to as the *window* in signal processing. The overlap defines how may elements are accessible on each side of the center element in the input data structure. The skeleton has one- and two-dimensional variants, the former working on either arrays or matrices, the latter working only on matrices. Two-dimensional MapOverlap currently only supports *separable filters* (i.e. a two-dimensional filter composed of two one-dimensional filters applied one after the other), but the programmer can choose if the first pass should be row-wise or column-wise. In the two-dimensional case, MapOverlap can use different overlap radii for the row and the column direction of the matrix.

For the first and last elements of the computed array, the overlap will need access to elements that are out-of-bounds. This overlap area around the edges of the containers can be handled in three different ways. The edge handling scheme can either be set to *pad*, whereby a value supplied by the programmer will be used; or to *cyclic*, to cycle back to the values in the other end of the data container; or to *duplicate*, to repeat the closest edge value.

An example of the MapOverlap skeleton for one-dimensional data is shown in Figure 3.4. The skeleton works like the Map skeleton, but the user function, denoted $\otimes$, has access to neighboring elements of the computed element. In this example the user function has access to one element on each side of the center element, as illustrated by the yellow color.

3.1.5 Scan

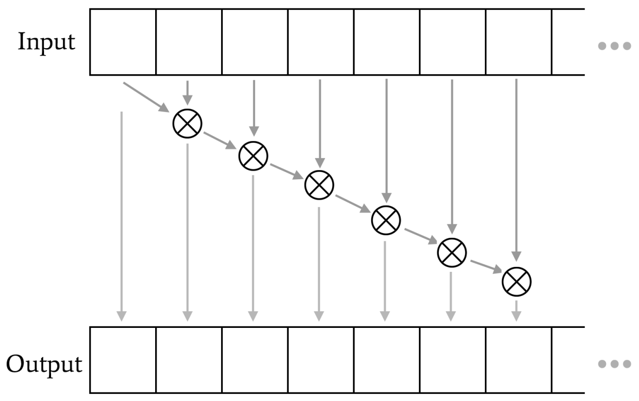


Figure 3.5: An example of the Scan skeleton.

The *Scan* skeleton is a generalized variant of prefix sum. Scan takes a single array as input. The output is an array of the same length as the input array, where the reduction of the first i elements of the input array is written to element i of the result array. In the case of regular prefix sum, the user function would be the addition operator, and element i of the result array would contain the sum of the first i elements of the input array. Computations might be reordered in the Scan skeleton, just as in the case of Reduce, requiring the user function to be a commutative and associative binary function. Scan can be either *inclusive* or *exclusive*. Inclusive Scan will include element i in value i of the result array, exclusive Scan will not. Exclusive Scan therefore needs an initial value, which will be the first element of the result array.

An example of the inclusive Scan skeleton is shown in Figure 3.5. Here the user function $\otimes$ is applied sequentially to the elements of an array. The partial result for each index of the input array is written to the same index of the output array. Note that the example shows a sequential implementation of the Scan skeleton, where the computations create a chain of dependencies. Parallel implementations of Scan require the operations to be reordered.

3.1.6 Call

The Call skeleton is a bit special, as it does not provide a specific structure of computation. Instead it allows for implementation of so-called *multi-variant components*[13], by letting the programmer provide CPU, OpenMP and accelerator implementations of the computational structure. Call is used to implement custom algorithms, not implementable using the existing skeletons. The advantage of using Call compared to making the implementation separate from SkePU, is that Call can use SkePU's smart containers and backend selection features. A typical usage example of Call is sorting. Sorting does not fit into the computational structure of any other skeleton in SkePU, but can be implemented with Call. This allows for different sorting algorithms to be used for different backends. Because the computational structure is not defined for Call, the programmer is also responsible for the implementation of a hybrid

```cpp
#include <skepu2.hpp>


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

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

float dot_product(skepu2::Vector<float> &v1, skepu2::Vector<float> &v2) {
    // Create an instance of the MapReduce skeleton
    auto dotprod = skepu2::MapReduce<2>(mult, add);
    // Call the instance with two vectors
    return dotprod(v1, v2);
}
```

Listing 3.1: Example of dot product using MapReduce in SkePU 2.

execution variant. For this reason the Call skeleton is not discussed more in this thesis. How the Call interface should be extended to let the programmers provide hybrid execution implementations is left for future work.

3.2 Smart Containers

To hide the complexity of memory management on multi-PU systems, SkePU implements so called *smart container* types. The types are as of today, *Vector* and *Matrix*. Experimental support for *Sparse Matrix* also exists. The smart containers provide interfaces resembling the interfaces of the C++ standard library containers. In addition to that, they take care of memory management to move and synchronize the containers between multiple heterogeneous PUs. Any subsection of the smart containers' memory can be copied to accelerator memory for use in skeleton computations. At the same time the container implementations keep track of which accelerators have partial copies of the data and which parts of the data in CPU and accelerator memory are valid. Unnecessary copying of data is avoided by the use of lazy memory copying, where the data reside in accelerator memory until needed elsewhere. Consecutive skeleton calls to accelerators are sped up this way, as the container does not need to be copied back to the CPU memory between the calls [10].

3.3 Code Example

A small example of using the MapReduce skeleton in SkePU 2 for a dot product computation can be seen in Listing 3.1. Two functions (*add* and *mult*) are defined and included as user functions when creating an instance of the MapReduce skeleton. The compiler will translate the *add* and *mult* functions together with the MapReduce skeleton instance into the supported backend languages. Note how the parallelism is completely hidden by the use of a skeleton in comparison to the OpenMP implementation of dot product presented in Listing 2.1.

3.4 User Functions

SkePU 2 lets the programmer define user functions both as lambda expressions and as free functions[3]. Because skeletons and their user functions must be able to execute on different backends, there are some limitations as to what is allowed in the user functions. OpenMP and

[3]A free function is function that is not a member function of a class or struct.

```
skepu2::ExecPlan plan;

// Up to 2000 elements, use sequential implementation:
plan.add(1, 2000, skepu2::Backend::Type::CPU);
// Then, up to 200000 elements, use OpenMP with 8 CPU threads:
plan.add(2001, 200000,  skepu2::Backend::Type::OpenMP, 8);
// Then use OpenCL with max 65535 threads and max 512 blocks:
plan.add(200001, INFINITY, skepu2::Backend::Type::OpenCL, 65535, 512);

skeleton_instance.setExecPlan(plan);
```

Listing 3.2: Example of defining an execution plan.

CUDA are both able to use C++ features, but because OpenCL is based on C, the syntax of the user functions must follow the C standards. C-style structs are allowed, but the user function must not allocate any new memory. Only a small subset of the standard library functions are available, such as *pow*, *abs* and *sqrt* [20].

Three types of arguments are passed to the user function in the skeleton call: *element-wise* arguments, *random access* arguments and *uniform* arguments. *Element-wise* arguments are containers where only one element from the container is passed to each user function call. These arguments typically determine the input size of the problem. *Random access* arguments are containers where the user function can access any element at any time. *Uniform* arguments are constant arguments passed directly to the user function, having the same value in every user function call. All skeletons do not have support for all types of arguments.

3.5 Backend Specification and Execution Plans

The programmer can explicitly specify on which processing unit to execute a skeleton call by setting the *backend specification*. This specification also includes parameters to the backend, such as number of threads or number of accelerators to use. In the case of a program pre-compiled without support for the selected backend, the runtime system will automatically fall back to an available backend.

Another way to specify which PUs to use is to define an *execution plan*. The plan specifies which backend to use for a specified input data size interval. An example of how to set up such an execution plan is shown in Listing 3.2. In this example a plan is created where the sequential backend is used for small input sizes, the multi-core OpenMP backend is used for medium sized input and the OpenCL backend is used for large input sizes.

3.6 Automatic Backend Selection and Tuning

Execution plans can be built by SkePU through an automatic tuning process. Once the tuning is invoked by the user, it can produce an execution plan with the fastest backend for each input size range. The backend selection tuner samples a number of input sizes for the skeleton instance and then benchmarks each available backend. The fastest backend for each range is then added to the execution plan.

The tuning mechanism was more evolved in the first version of SkePU, where also parameters to the backends were tuned, not only which backend to use. Adaptive tuning algorithms were used to only make performance samples where necessary to speed up training time. Pruning techniques were also employed to reduce the search space. Examples of backend specific parameters that could be tuned is the max thread-per-block and the max blocks-per-grid parameters to the CUDA and OpenCL backends as well as the number of threads in the OpenMP backend [12, 32].

3.7 Hybrid Execution with StarPU in SkePU 1

SkePU 1 was available in two distributions: one standard edition and one with experimental support for hybrid execution of skeletons. To provide support for asynchronous and hybrid execution and dynamic load balancing, the latter version of SkePU was integrated with the task-parallel runtime scheduling system StarPU[4]. The implementation divided the workload of a skeleton into smaller tasks for StarPU to schedule on the available *workers*. By providing StarPU with implementations both for CPU and CUDA, the workload could be divided between CPU cores and GPUs, thus enabling hybrid execution [11].

The StarPU framework includes several dynamic scheduling policies, e.g. the *greedy* policy, where the tasks are assigned to the first available worker; or the *work stealing* policy, where workers can steal tasks from other workers when their own task queue is empty. To provide a more sophisticated scheduling policy for heterogeneous architectures, StarPU also includes a *HEFT* (Heterogeneous Earliest Finish Time) [50] scheduler. In this scheduler performance models for the different PUs are used to predict how new tasks should be scheduled to minimize the finish time, taking predicted execution time of already scheduled tasks on the PU into consideration [2].

In order to use the StarPU runtime system, the skeleton calls in SkePU had to be decomposed into a number of tasks. The number of tasks to divide each skeleton invocation into was decided by the programmer, adding another parameter to tune or manually configure. More tasks are desirable for large input sizes, as it facilitates load balancing. For small input sizes however, too many tasks will reduce performance because of the increased scheduling overhead. Several of the HEFT based schedulers in StarPU were tested with SkePU, but they all had one major drawback: the need for warm-up runs. The HEFT schedulers will eventually lead to good performance, but the skeletons in SkePU 1 had to be executed several for the performance models to gather enough execution data to produce accurate predictions for the scheduler [11].

3.8 Multi-accelerator Support

The CUDA and OpenCL backends are internally implemented in two ways: a single device implementation capable of executing the skeleton on a single accelerator, and a multi-device implementation where the workload of the skeleton is distributed between an arbitrary number of accelerators. How many accelerators should be used to execute a skeleton can be set at run-time per skeleton invocation. The multi-accelerator implementation uses a trivial workload partitioning scheme, that splits the input data into equally sized partitions. This works well when all accelerators are of the same type, or at least have similar performance. When that is not the case the slowest accelerator will become a bottleneck.

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

4 Related Work

In this chapter earlier work on what has been done in the field of heterogeneous computing will be presented, as well as some frameworks similar to SkePU.

Heterogeneous computing has been an active research area over the last couple of years. Mittal and Vetter [35] present an extensive survey of heterogeneous computing techniques where CPU(s) and GPU(s) are used in parallel. In their survey they conclude that a majority of the techniques studied need some manual tweaking from the programmer to work. Typically by manually setting up the partition sizes of the workload or by measuring the relative performance of the PUs. They call for new frameworks that can perform these tasks automatically.

4.1 Earlier Implementations of Hybrid Execution

Grewe and O'Boyle [22] present a static task partitioning scheme for OpenCL programs on heterogeneous architectures. They show that different applications have different characteristics and group application executions into three categories based on how the best performance was achieved: CPU-only execution, GPU-only execution or using hybrid execution. This is done by executing the applications with 11 different partition ratios, from 100% on the GPU, via 90% on GPU and 10% on CPU and so on, to 100% on the CPU. They report that it is vital to classify instances which belong to the GPU-only category correctly. These applications tend to have a significantly higher performance when using GPU-only execution, compared to if the work is partitioned. The CPU-only category shows a similar behavior, but the performance drop in case of a misprediction is not as significant as in the GPU-only case. In the hybrid execution category on the other hand, the performance of the second best partitioning is closer to the best one, meaning that a misprediction here would not be as severe as for the other two categories.

In their paper, they present a partitioning scheme that uses static analysis on the OpenCL computation kernels and a hierarchical classification mechanism to predict the best partitioning. Their solution extracts 12 code features from the OpenCL computation kernel at compilation time, including the number of integer and floating point operations, the number of memory accesses and the data size to transfer. These features are then supplemented with an additional feature, the input data size, at run-time. The feature set is then used in a two-layer predictor to predict the optimal partitioning. The first layer consists of two binary predictors that classify if the application is either a CPU- or GPU-only problem. If the first layer cannot decide if the problem is in the CPU- or GPU-only category, the problem is considered a hybrid execution problem and sent to the second layer. In the second layer another predictor is used to predict which of the 11 partition ratio classes (including CPU-only or GPU-only) the problem belongs to.

They evaluate their system on 47 different benchmarks executed with a number of different input data sizes. A total of 220 unique program-input pairs are tested. Their predictor implementation is compared to four different scheduling strategies: two simple baseline strategies, always choosing CPU-only or GPU-only execution; an oracle, always choosing the optimal partitioning from the 11 classes as an upper bound; and a dynamic mapping scheme performing load balancing during run-time. The dynamic mapping scheme divides the work into a number of blocks, and the blocks are then sent to the CPU or the GPU, whenever they finish their previous block.

They report that their static approach gives an average speedup of 1.57 over the dynamic approach, an average speedup of 3.02 over the CPU-only approach and a speedup of 1.55 over the GPU-only approach. Their two-level predictor manages to classify a GPU-only problem correctly in 91% of the cases. The CPU-only problem category is predicted correctly in 95% of the cases. The last group of instances where hybrid execution gives the best performance has a lower result, due to it being a harder classification problem. However, the authors state that the resulting performance is within 80% of the optimal performance in 65% of the cases.

Their evaluation is performed on a machine using two quad-core Intel Xeon E5530 CPUs and an AMD Radeon HD 5970 GPU.

In a paper from 2016, Contassot-Vivier and Vialle [8] present a way to implement Jacobi relaxation on systems equipped with three different processing units, namely a CPU, a GPU and a MIC. Jacobi relaxation is an iterative process that can be used for modeling heat transfer or electrical potential diffusion in a regular grid. The objective of the relaxation is to eventually reach a stable state for the grid given some condition. In the paper they use a fixed number of iterations as the termination condition. For each iteration, the new value of a grid point is calculated as the average of the old values of that point and its four closest neighboring grid points.

Their solution is implemented in three versions: one parallelized using OpenMP for the multi-core CPU, one using *offloaded* OpenMP for the MIC and one using CUDA for the GPU. All implementations use device dependent optimizations and take the different memory models of the PUs into consideration. The grid is divided into three slices, one slice for each PU with the CPU slice in the middle. The entire grid is kept in CPU memory, while the MIC and the GPU only keep a copy of their slice of the grid, plus one extra line from the neighboring PU (in both cases, the CPU). In each iteration step of the Jacobi relaxation, the PUs update their part of the grid. When an iteration is done, the CPU fetches the boundary rows in the MIC's and the GPU's memories and overwrites the corresponding rows in main memory. Then its own boundary rows are uploaded to each PU before the next iteration begins.

They evaluate their implementation on two machines. The first one with two hexa-core Intel Xeon E5-2620 CPUs, an Intel Xeon Phi 3120 MIC and a NVIDIA GTX Titan Black GPU, the other one with two octa-core Intel E5-2640 CPUs, an Intel Xeon Phi 5100 MIC and a NVIDIA Tesla K40m GPU.

To find the best partitioning they first evaluate the performance of the individual PUs to find a relative computation performance. Partitioning of the work is done manually, according to this relative performance. They report that this scheme results in the optimal performance for one of the evaluation systems, but that the optimal partitioning of the other system is slightly different than their theoretically optimal partitioning.

Shen et al. [41] present a way to partition workloads on heterogeneous platforms. To categorize heterogeneous platforms they suggest two metrics: first the *relative hardware capability*, i.e. the relative throughput of the GPU and the CPU; and secondly the *GPU computation to data transfer gap*, the GPU throughput to data-transfer bandwidth ratio. They measure overall execution time per PU, for the GPU split into data transfer and kernel execution time. Their decision algorithm is split into three steps. In the first step a model of the optimal partition ratio is built. The optimal partitioning is a partitioning such that $T_C = T_G + T_D$, i.e. the execution time of the CPU part (denoted T_C) is equal to that of the GPU part (denoted T_G) including the its data transfer time, T_D. They then find models for T_C, T_G and T_D, all dependent on the partitioning point, β based on two metrics: relative hardware capability and computation to data transfer gap. Their model also handles three types of applications: those where no data transfer to GPU is needed, those with data transfer size proportional to the GPU partition size and those with a fixed data transfer size.

In the second step, the model is used to predict the optimal partitioning, by calculating β in the model from the first step. This is done by profiling the application, using different input sizes to measure execution and data transfer time. These measurements are used in the model to calculate β. Two types of training are implemented. The first type is online training, where the application is executed with the given problem size once for each PU to measure the execution times. The other type is offline training, where linear regression models are built for the execution and data transfer times, with the input size as a parameter. In online training the training cost grows with the number of problem sizes to train for, and is thus best suited when a few, specific input sizes executed many times in a row. Offline training has a fixed overhead and is better suited when many different, unknown input sizes are expected. In the offline training model the memory hierarchy is also taken into consideration by splitting the range of input sizes into six smaller intervals. The intervals are divided according to memory hierarchy details, such as size of the different cache levels and the maximum size of the GPU memory. They assume the execution/data transfer time can be estimated as a linear function inside each of these six intervals and together the intervals form a model over the entire data size range.

In the last step, the optimal β is predicted and used to actually partition the workload. Lower bounds are used for the CPU and the GPU parts, to ensure the workload is not distributed in a way such that hardware computing power would be wasted. For example that the size of one PUs partition is smaller than the number of cores on that PU. In such cases, all workload is moved to the other PU. In case both partition parts are above the lower bounds, both the CPU and the GPU will be used. The GPU size is rounded up, to ensure that a multiple of the *warp size* (the number of threads that are executed simultaneously as one group) is used, as this results in better performance. The rest of the input data is scheduled on the CPU.

They also extend their work to multi-GPU systems with identical as well as non-identical GPUs. In the case of identical GPUs, they treat the GPUs as a single device and divide the workload evenly between them. For non-identical GPUs, they extend their model built in the first step to produce partition sizes for all PUs. They argue that their solution can be used to handle any number and types of accelerators, as they only base their models on the execution and data transfer time, not architectural details.

Evaluation is performed using 13 OpenCL applications, each executed with six different input sizes, where each input size falls into one of the intervals used by the offline trainer. They compare their solution to a CPU-only and a GPU-only scheduling policy, as well as an oracle, which finds the optimal β by the means of binary search. They report that their solution is within a relative performance difference of 10% compared to the oracle in most cases. Their work is evaluated on six different platforms, four of them with a single GPU and two of them with multiple GPUs. For their main evaluation a hexa-core (12 threads using hyper threading) Intel Xeon E5-2620 CPU and a NVIDIA Tesla K20 GPU is used. For the multi-GPU experiments they use one machine with two identical GPUs, two NVIDIA GTX 560 and one with two non-identical GPUs, a NVIDIA Tesla C2050 and a Quadro 600.

Luk, Hong and Kim [33] present an adaptive mapping technique to perform dynamic load balancing in their heterogeneous programming framework called *Qilin*. The framework supports two different APIs. The first of them is the *Stream-API* which uses common data-parallel operations on arrays, similar to skeletons in skeleton programming, to solve a problem. Qilin uses a data dependency graph and dynamic compilation to translate the Stream-API operations into computation kernels for the CPU and the GPU. Larger kernels are made by optimizing and merging operations in the data dependency graph to minimize the kernel launch overhead.

The second alternative is to use the *Threading-API*, where the programmer can implement custom computational kernels with support for the CPU using Intel's Threading Building Blocks library and for the GPU using NVIDIA's CUDA language.

Qilin supports hybrid execution by splitting the work between the CPU and a GPU. For operations where the result of the CPU and the GPU part of the execution needs to be merged, Qilin automatically identifies this and creates the necessary kernels to merge the results into one.

The runtime system uses an adaptive mapping technique to partition the work between the CPU and the GPU. Qilin first makes test executions of the kernels on the CPU and the GPU with different input sizes to gather execution time data. The execution time of each kernel is approximated by two linear curves, one for the CPU and one for the GPU. The theoretically optimal partitioning can be found by calculating the intersection point of the two curves. By storing the curves in a database, they can be reused between multiple executions of the applications, without the need to retrain the kernels.

Similar to Grewe and O'Boyle they evaluate their system by comparing their adaptive mapping solution to GPU-only and CPU-only execution. They also compare to an oracle, always picking the best partition ratio out of 11 tested, in steps of 10 percentage points. In the evaluation of the adaptive model, they first train the model once and then execute the program as many times as possible within one hour. The average execution time per iteration is then calculated by dividing the number of complete iterations by the total time. This way they amortize the training time and the dynamic compilation overhead, as the authors assumes the real world use of such a system would be many consecutive executions of the same kernels.

On eight benchmarks from different domains, they find that their adaptive mapping technique performs better than both the CPU-only and the GPU-only scheme. They also report that the result of their adaptive mapping is within 94% of the optimal execution time as given by the oracle. For some benchmarks the performance of their adaptive mapping is even better than the oracle, due to the finer granularity in the partitioning ratio for the adaptive mapping technique. They evaluate Qilin on a system consisting of two quad-core Intel Core 2 CPUs and a NVIDIA 8800 GTX GPU. They also demonstrate their technique's ability to adapt to software and hardware changes by replacing the CPU and the GPU, as well as change the compiler.

4.2 MapReduce Frameworks

Hong et al. [25] present a framework for MapReduce that will allow the programmer to write the code once and run it on either a CPU or CUDA enabled GPUs. In their research they also try hybrid execution with both a CPU and a GPU at the same time, but report that the performance improvements is never above 10% and sometimes even slower than executing on a single processing unit. According to the paper the reason for this behavior is the fact that GPUs are many times faster than CPUs. Offloading some of the work to the CPU will therefore not make much of a difference. The other thing they identify as a bottleneck is a major scheduling overhead for hybrid execution, especially when transferring non-array data between processing units. Their results are evaluated on eight applications with typical MapReduce problems from different research fields. Their experiments are performed on a machine with a NVIDIA GTX280 GPU equipped with 240 cores and 1 GB of video memory and a Intel quad-core CPU clocked at 2.4 GHz.

Chen, Hou and Agrawal [5] show a way to implement MapReduce with hybrid execution scheduling on *fused* CPU-GPU architectures, where the GPU is integrated on the same chip as the CPU. On these architectures the CPU and the GPU share a part of the same physical memory, resulting in no latency for copying data from one processing unit to the other, which is exploited to get a low scheduling overhead. They propose two scheduling solutions. In the first one, called the *map-dividing scheme*, one CPU thread works as a scheduler and the rest of the CPU and GPU threads are considered workers. The input data is divided into a large number of blocks. The scheduler assigns blocks to the workers, which performs the map and reduce functions. When all individual blocks are reduced, the last reduction steps are performed by the GPU. The scheduler thread communicates with the workers using an array of structs called *worker info*, located in the *zero copy buffer* (the part of the memory that is shared between the CPU and the GPU). Each *worker info* contains a `has_task` flag, together with the current block offset and size. When a worker is done with its task, it sets the `has_task` flag to 0. The scheduler thread on the CPU loops over the worker info structs and assigns a new block to the workers that are done with their last block. This solution takes advantage of the fused architecture with the shared memory between the CPU and the GPU to perform a low overhead scheduling and load balancing. The second solution is called the *pipelining scheme* and implements a producer-consumer model. In this scheme all map operations are performed by either the CPU or the GPU, while the reduce operations are performed by the other processing unit. This solution is implemented in two ways. First using a dynamic scheduling approach, similar to that in the *map-dividing scheme*, but also using static scheduling. They argue that, since all map operations are performed on the same processing unit where all worker threads have the same processing speed, there is not really a need for load balancing. This also means that the scheduler thread on the CPU can be used as a worker thread, potentially leading to higher performance. Their results show that the *map-dividing scheme* performs best on average. The *pipelining scheme* also works well when the GPU is used for the map part and the CPU for the reduce part, especially with static scheduling. In their experiments they evaluate their result using five typical MapReduce benchmark problems. They use a AMD Fusion A8-3850 chip, consisting of a quad-core CPU and a Radeon HD 6550D GPU programmed using OpenCL.

4.3 Linear Algebra Libraries

Some attempts at implementing linear algebra libraries for heterogeneous CPU/GPU systems have been made, for example by Humphrey et al. [27] and Tomov, Dongarra and Baboulin [49]. They present different ways to implement factorization algorithms using hybrid execution on heterogeneous systems. Both papers suggest that in heterogeneous systems, the processing units should be used for what it does more naturally: the CPU should take care of irregular and serial sequences of the programs, while massively parallel sequences should be offloaded to the GPU.

4.4 Related Frameworks

This section will introduce a number of frameworks similar to SkePU, targeting heterogeneous or multi-accelerator architectures.

4.4.1 Marrow

Marrow is a C++, algorithmic skeleton framework with both data and task-parallel skeletons. Marrow includes skeletons *Pipeline*, *Loop*, *For*, *Map* and *MapReduce*. The skeletons are executed on GPUs using OpenCL. The framework has support for partitioning of the workload between multiple, heterogeneous GPUs, based on performance benchmarks. Marrow uses skeleton nesting, where multiple skeletons are combined into one compound computation to reduce runtime overhead and speed up the computations [1].

Since lately, Marrow also supports multi-core CPU execution as well as hybrid execution with a CPU and multiple GPUs [43].

4.4.2 Qilin

Qilin is an experimental heterogeneous programing API written in C++. It has support for a number of skeleton-like *operations*, similar to Reduce and Map with predefined user functions. More complex operations can be built manually, by specifying the CPU and GPU implementations using Intel Threading Building Blocks (TBB) for the CPU and CUDA for the GPU. Qilin uses lazy, dynamic compilation of the API calls to be able to adapt to changes in the runtime environment. By merging multiple small, element-wise operations into one large operation, the scheduling overhead can be reduced, similar to how Marrow can merge skeletons. The system supports hybrid execution on a CPU and a GPU and uses training runs to build linear execution time models for the CPU and the GPU. The models are only calculated once per kernel, as they are stored in a database for later reuse [33].

4.4.3 Muesli

Muesli (*Muenster Skeleton Library*) is a C++ skeleton library for heterogeneous systems. The library contains data-parallel skeletons *Map Zip*, *Fold* (i.e. Reduce) and *MapStencil* and task-parallel skeletons *Farm*, *Pipeline*, *Divide and Conquer*, and *Branch and Bound*. In their implementation Map is unary Map taking a single element-wise array, and Zip is a binary Map implementation taking two input arrays and a binary user function. The library currently supports multi-accelerator and hybrid execution of the data-parallel skeletons using OpenMP and CUDA. The partitioning has to be set up manually by the programmer, as no tuning mechanism is implemented yet [53].

The library also supports Intel Xeon Phi MICs and distribution of work between compute nodes in a cluster [19].

4.4.4 SkelCL

SkelCL is a OpenCL based skeleton programing library for multi-GPU systems. It includes seven data-parallel skeletons: *Map*, *Zip*, *Reduce*, *Scan*, *MapOverlap*, *Allpairs* and *Stencil*. The skeletons in SkelCL are less flexible than in SkePU, e.g. there is only support for unary and binary Map, the latter being Zip. Just like in SkePU, the library implements its own vector and matrix containers to hide the multi-PU memory management from the user. The containers automatically move data between CPU and GPU memory when needed. The user functions are defined as plain strings in SkelCL, just as in OpenCL. This means there is no compile-time syntax or type checking, as there is in SkePU 2. Skeletons i SkelCL can be executed only on GPUs. The framework supports both single and multi-GPU execution, but not hybrid execution on both CPU and GPU simultaneously [3, 44].

4.4.5 ImageCL

ImageCL is a domain-specific language with a source-to-source compiler that translates ImageCL kernels to OpenCL code. The language is specialized at image filter operations, but also supports general stencil operations, resembling the MapOverlap skeleton in SkePU. ImageCL has support for hybrid execution on multi-core and multi-accelerator systems by using OpenCL. Hardware dependent optimizations are also performed by letting the source-to-source translator generate a number of candidate implementations and use machine learning to pick the best implementation for each device. In contrast to SkePU, ImageCL also has support for running the stencil operations on multiple computer nodes in a cluster, where each node internally might use hybrid execution by dividing the work between the CPU and any number of accelerators [21].

4.4.6 StarPU

StarPU is a task-based runtime scheduling system written in C, with optional `#pragma` compiler-directives. The system provides a platform for development of task-based applications for heterogeneous architectures and has support for hybrid execution. The most central structure in StarPU is called *codelet*. A codelet defines a computational task and can provide several implementation variants, including variants for accelerators or CPUs exploiting various vector instruction sets. Tasks are formed by attaching data containers to a codelet and are submitted to StarPU's runtime scheduler. The scheduler can then decide where to execute the task depending on available hardware resources and which implementation variants are provided by the codelet. StarPU features a generic framework for scheduling and includes several scheduling policies. Examples include greedy scheduling and work stealing. Custom policies can also be defined by the programmer. Just as SkePU, StarPU implements a custom data management system to hide and optimize data movements between PUs [2].

4.4.7 STAPL

The parallel programming framework *STAPL* (Standard Template Adaptive Parallel Library) is a parallelized extension to C++ standard library, written using modern meta-programming features. The framework contains distributed data structures and provides parallel implementations of the algorithms in the standard library as well as some additional algorithms. Custom data structures and algorithms can be provided by the programmer to further extend the framework. The data structures are accessed through interfaces called *views*. Views allow the algorithms to see the same data container in different ways. For example, the same matrix can be seen as a row-major or column-major order matrix, or even as the underlying sequential array. Algorithms can be executed on multi-core, multi-node computer clusters by the use of OpenMP and MPI. STAPL automatically adapts to the execution environment by selecting the most appropriate algorithm variation and by tuning the communication scheme between the cores. The framework does not have support for hybrid execution of algorithms [4].

5 Design and Implementation

The implementation work in the thesis was divided into three steps. The first step was to design and implement a new hybrid execution backend with workload partitioning for all the skeletons in SkePU. This step is described closer in Sections 5.1 and 5.2. The result of this step would allow the user to run any skeleton in SkePU with hybrid execution by manually deciding how the workload should be partitioned between the CPU and accelerators. The second step was to design and implement a tuner to make SkePU predict the optimal workload partitioning automatically, based on execution time benchmarking. This step is described in Sections 5.3 and 5.4. Finally, to have something to compare the new implementation to, the runtime scheduler from SkePU 1 based on the StarPU library was ported to SkePU 2. The reimplementation of it is described in Section 5.5.

5.1 Implementation of the Hybrid Backend

The new hybrid execution backend is implemented with support for five of the skeletons in SkePU, namely: Map, Reduce, MapReduce, MapOverlap and Scan. No hybrid execution implementation was made for the Call skeleton, as the definition of the skeleton requires the programmer to provide hybrid execution variants for each skeleton instance. How the API of the Call skeleton should be extended to allow such hybrid execution implementation variants is left for future work.

The implementation of a new hybrid execution backend was made in two steps: first the implementation of workload partitioning for each skeleton and then implementation of an auto-tuner, capable of predicting how to optimally partition the work between the PUs. The hybrid backend is designed to partition the workload of a skeleton invocation into two pieces: one for the CPU and one for the accelerators. The hybrid backend is designed to support any number of CPU cores and any number of accelerators, as long as the accelerators are using the same implementation (CUDA or OpenCL).

As hybrid execution only ought to be of interest when at least one accelerator and multiple CPU cores work together, compilation of the new hybrid execution backend is automatically activated if the program is precompiled with OpenMP and at least one of the accelerator backends (CUDA and OpenCL). SkePU will automatically choose an available accelerator implementation and if both are available, the choice might be overruled by the programmer's preference. To actually use the hybrid backend, it must be selected for each skeleton invocation.

```
skepu2::BackendSpec spec(skepu2::Backend::Type::Hybrid);
spec.setDevices(2);
spec.setCPUThreads(16);
spec.setCPUPartitionRatio(0.2);
skeleton_instance.setBackend(spec);

// Invoke skeleton with some data
skepu2::Vector<int> in, out;
skeleton_instance(out, in);
```

Listing 5.1: Example of using the hybrid backend with manually set partition ratio.

This is done in the same way as for the already existing SkePU backends, leaving full control to the user to choose whether or not to use hybrid execution. Just as for the other backends, a fall-back mechanism ensures that another backend will be selected if the hybrid execution backend is selected by the user's program, but not included in the precompilation. An example of how to set up an backend specification to use the hybrid backend is presented in Listing 5.1. In the example a backend specification is created with the hybrid execution backend configured for two accelerators and 16 CPU threads, where 20% of the work will be executed by the CPU threads and the rest by the two accelerators.

The user has, just as for the other backends, three ways to specify how to use the hybrid backend. Either to (1) always use the hybrid backend by explicitly selecting it, as shown in Listing 5.1; or to (2) manually configure an execution plan, including the hybrid backend for one of the input data size intervals; or to (3) let the automatic backend selection tuning create an execution plan, where the hybrid backend might be included.

A design choice was made to keep the workload partitioning and the hybrid backend implementation and the hybrid tuner separated. This allows the partitioning to be optimized in the future without any need to change the tuner. Likewise could the tuner be improved, exchanged or even made selectable in the future without making changes to the hybrid backend and the partitioning.

The implementation was based on the latest release of SkePU at the time this thesis was written, which was version 2.

5.2 Workload Partitioning

In the first step of the implementation a new hybrid execution backend with workload partitioning for all skeletons was implemented. To keep a coherent interface between the skeletons, the workload partitioning was designed to only use one parameter to decide the partitioning: the *partition ratio*. This ratio defines the proportion of the work that should be computed on the CPU; the rest of the work is computed on an accelerator backend. The CPU partition is then further divided into blocks, one for each CPU thread. The accelerator backend also has the opportunity to further divide the work between multiple accelerators, possibly accelerators of different types. As just a single partition ratio parameter is used, the same auto-tuning implementation can be used to tune all skeletons, without requiring any deeper knowledge of how the work is actually partitioned. The implementation assumes that no workload balancing is needed between the CPU cores. The same assumption is already made in the OpenMP backend and holds for most data-parallel problems.

One important aspect of the implementation of each skeleton was to make sure that the execution times of the CPU and the accelerator partitions of the hybrid backend would match the execution time of the already existing OpenMP and CUDA/OpenCL backends respectively. In case the hybrid backend is executed with N input elements and a partition ratio of 50%, the execution time of the CPU and accelerator partitions should match the execution time of the OpenMP and CUDA/OpenCL backends respectively, with $N/2$ input elements. This is of importance as the already existing backends are used to benchmark and build performance models for the auto-tuner. If the performance differs much, the tuner will make bad predictions.

This was partly realized by calling the already existing accelerator backend implementations. Several positive consequences follows from this: code duplication is avoided and future optimizations to the accelerator backends will automatically be included in the hybrid backend. Some changes to the internal interfaces of the accelerator backends were needed in order to make them general enough to accept skeleton computations on partial containers. For the OpenMP part of the hybrid backend it was not possible to call the already existing OpenMP backend, as some memory management code and synchronization between the CPU and the accelerator partitions was needed. Nevertheless, most of the actual skeleton execution part of the OpenMP backend was copied to the hybrid backend implementation to ensure the performance would match as closely as possible.

The skeleton implementations in the hybrid backend spawns a number of OpenMP threads, chosen by the programmer in the same way as in the OpenMP backend. One of the threads is responsible for the accelerator backend, while the rest of the threads will work on the CPU partition of the problem. The already existing accelerator backends are implemented as blocking calls, i.e. calls where the executing thread will not return until the computation on the accelerators is finished. Unfortunately, this means that the thread making the call will go idle once the data is uploaded to the accelerator and it is busy computing, thus wasting CPU performance. Some tests were conducted to see if the operating system scheduler could manage to fill this idle time by using one more CPU thread than number of CPU cores (where one thread managed the accelerator backend). However, these tests showed a significant performance drop for most skeletons compared to using the same number of threads as CPU cores, probably due to scheduling overheads and less efficient cache usage.

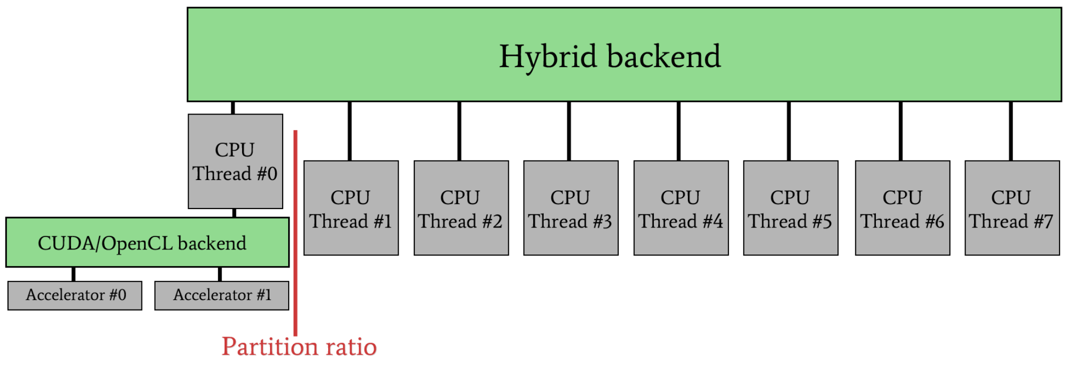


Figure 5.1: Schematic figure of the partitioning scheme with eight CPU threads and two accelerators.

During the implementation of the hybrid backend a constant execution time increase was noticed for the CUDA partition, compared to running the CUDA backend directly with the same problem size as the partition. It was found that this was caused by the fact that the CUDA backend was called from the last thread in the OpenMP thread group. Calling CUDA from a non-main thread forces the CUDA runtime to copy its context in order to safely support multi-threaded use of the runtime system. By instead letting the first OpenMP thread handle the accelerator backends the problem was solved, as the first thread in the group of OpenMP threads is usually the main thread, which already has a CUDA context.

A schematic image of the workload partitioning with eight CPU threads and two accelerators is shown in Figure 5.1.

Each skeleton ensures that the workload of the CPU and accelerator partitions are large enough to actually use hybrid execution. When a skeleton is invoked with a too small or too large partition ratio, causing the number of work items for one of the partitions to be too small, all workload is put on the other partition. When one of the partitions is empty, the hybrid backend falls back to the already existing OpenMP and CUDA/OpenCL backend implementations in order to minimize the overhead caused by the hybrid backend. Generally the hybrid backend falls back to CPU-only or accelerator-only execution, if there is not enough work items to occupy all CPU threads, or if the GPU does not have enough work items to fill a warp.

Details on how the workload was distributed for each skeleton type is presented below.

5.2.1 Partitioning of Map

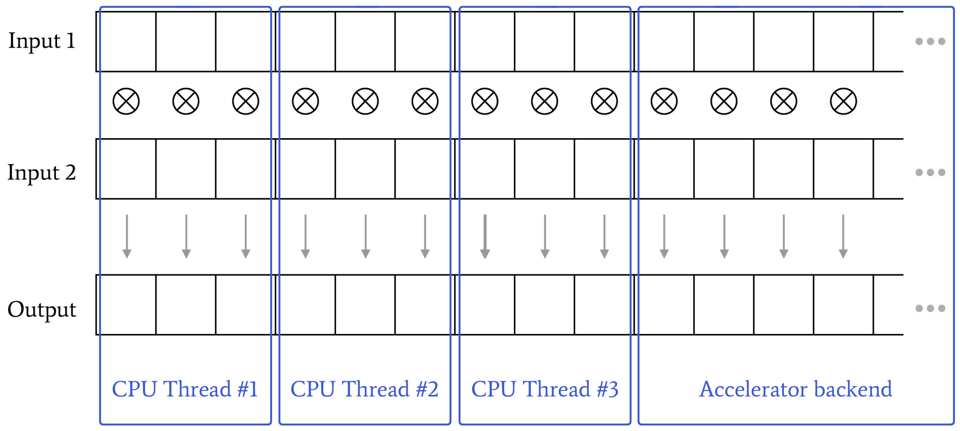


Figure 5.2: Partitioning of the Map skeleton with three CPU threads.

Due to the parallel properties of the Map skeleton, the partitioning scheme follows naturally. In this skeleton the partition ratio decides the number of output elements to compute on the CPU and how many to compute on an accelerator backend. The element-wise arrays are divided into two parts, where the second part is calculated on the accelerator backend and the first part is further divided into blocks, one block for each CPU thread to process.

An outline of the partitioning scheme for the Map skeleton, executed with three CPU threads is shown in Figure 5.2.

5.2.2 Partitioning of Reduce

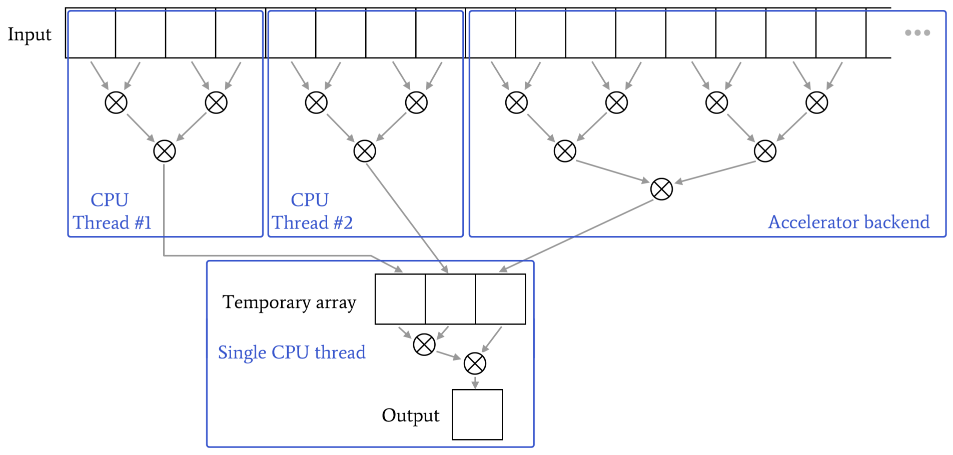


Figure 5.3: Partitioning of the Reduce skeleton with two CPU threads.

The SkePU Reduce skeleton has three different implementations, one for arrays with a scalar as output and two for matrices, where the output is either an array (i.e. one-dimensional reduction) or a scalar (i.e. two-dimensional reduction).

In the case of an array as input, the input data is partitioned between the CPU and the accelerator backend according to the partition ratio. The CPU partition is then divided equally between the CPU threads. Each CPU thread and the accelerator backend perform reduction on its part of the input and all partial reductions are collected in a temporary array. The reduction of the entire array is then computed by a single CPU thread reducing the temporary array down to a single scalar.

For the matrix reductions, the input matrix is divided horizontally according to the partition ratio. The rows of the CPU partition are evenly divided between the threads and the rest of the rows are computed by the accelerator backend. In the one-dimensional reduction case, the results can be directly written to the result vector by the PUs. In the case of two-dimensional reduction, each CPU thread and the accelerator backend perform first a row-wise reduction and then a column-wise reduction on its part of the matrix. The resulting scalar values from each CPU thread and from the accelerator backend are collected in a temporary array. These values are then reduced down to a scalar value by a single CPU thread.

The horizontal partitioning of the matrices is coarse-grained and works best for matrices with a high number of rows, as too few rows can be hard to partition evenly according to the partition ratio. More complex partitioning schemes could result in better results, but this one was chosen to minimize the need for inter-PU communication and synchronization.

An outline of the partitioning scheme for the Reduce skeleton on vectors, executed with two CPU threads is shown in Figure 5.3.

5.2.3 Partitioning of MapReduce

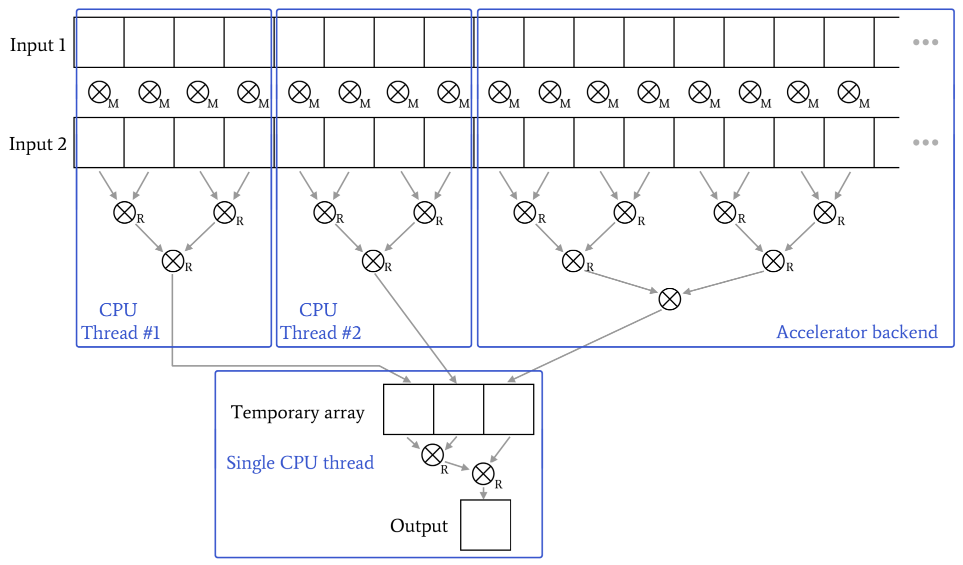


Figure 5.4: Partitioning of the MapReduce skeleton with two CPU threads.

Due to the similarities between MapReduce and the Reduce skeleton, the implementation of the former is done in a similar way. The partition ratio divides the input arrays into one partition for the CPU and one partition for the accelerator backend. The CPU partition is then evenly divided into blocks, one for each CPU thread. Each CPU thread and the accelerator backend then execute the Map step and reduce their block of the input to a single scalar value.

All these values are collected in a temporary array and a single CPU thread reduces them down to the global result.

An outline of the partitioning scheme for the MapReduce skeleton, executed with two CPU threads is shown in Figure 5.4.

5.2.4 Partitioning of MapOverlap

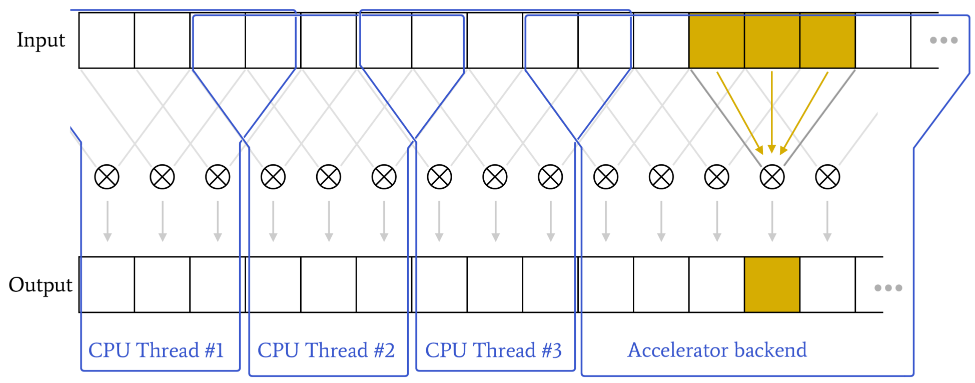


Figure 5.5: Partitioning of the MapOverlap skeleton with three CPU threads.

The MapOverlap skeleton comes in three variations: one for vectors, and two for matrices. For matrices there exist different implementations for row-wise and column-wise MapOverlap. Partitioning of MapOverlap for vectors is done in the same way as the regular Map skeleton, by dividing the output array into a CPU partition and an accelerator partition. The CPU partition is further divided between the CPU threads. To be able to work on parts of a vector, the accelerator backends needed some generalizations to their internal interfaces.

Row-wise MapOverlap needed similar work on the accelerator backends to work on partial matrices. Here, as in the Reduce implementation, the matrix is divided horizontally into a CPU partition and an accelerator partition, where each CPU thread and the accelerators are assigned a number of rows to work on.

The accelerator backend implementation of column-wise matrix Reduce proved to be hard to generalize for partial matrices without a significant amount of work. Due to time constraints this variant of MapOverlap was not considered for hybridization.

An outline of the MapOverlap partitioning scheme with three CPU threads is shown in Figure 5.5.

5.2.5 Partitioning of Scan

Scan contains more complicated dependencies between the output values that must be taken into consideration to get an effective implementation. Like before, the input array is divided into one CPU partition and one accelerator partition and the CPU part is then further divided into one block for each thread. The execution is performed in two steps. In the first step each CPU thread and the accelerator backend perform a local Scan on their block of the input array, then in the second step the local Scans are combined with the missing offset values from the proceeding blocks; forming the global Scan result.

After the first step with the local Scan, the element of each block lacks the sum[1] of all the elements in the proceeding blocks. The sum of each block is already calculated: it is the last element of the local Scan of the block. An array of the missing values that must be added to

[1]Note that the words add/sum is used in this section, as if the Scan was a prefix sum, but other binary operations can also be used.

each block is produced by letting each CPU block copy its last calculated value to a temporary array and by letting a single CPU thread perform a Scan on that array. The temporary array then contains the missing values for each block. All CPU threads are synchronized before the single thread performs a Scan on the temporary array to make sure the local Scan of each CPU thread is finished. In the second step, the accelerator thread and each CPU thread (except for the first thread, which has no preceding blocks) add the missing value from the temporary array to all elements in their block. When this is done, global Scan has been produced. An

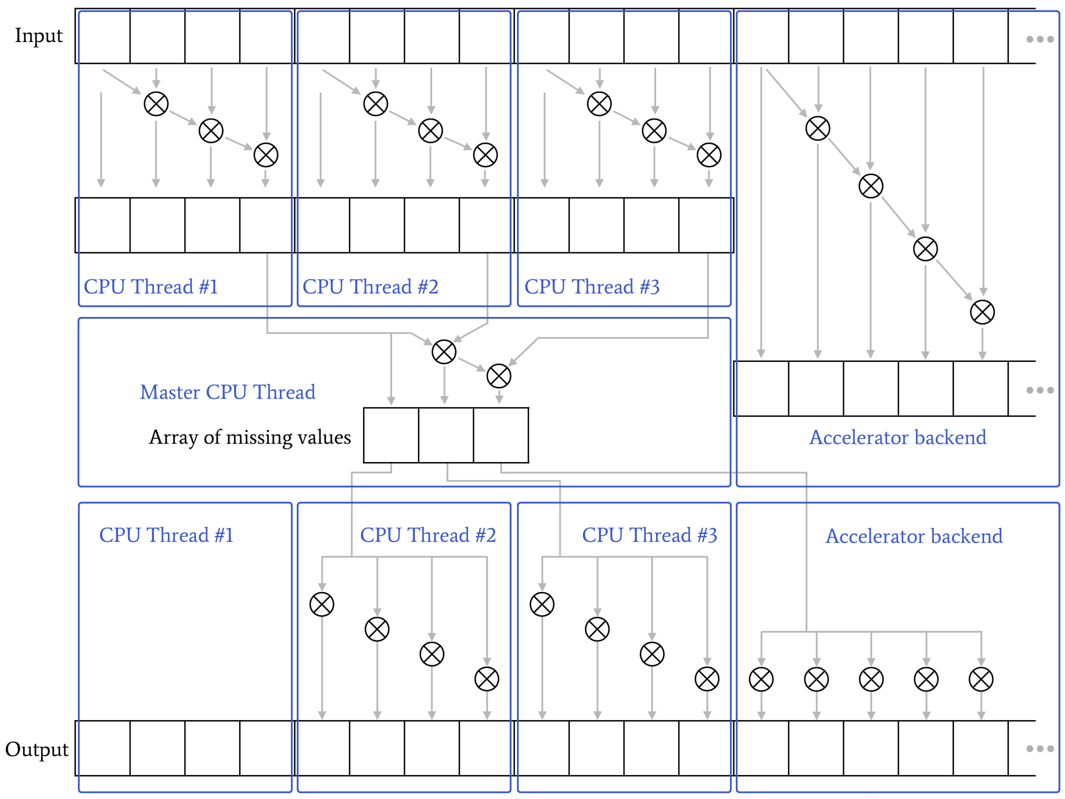


Figure 5.6: Partitioning of the Scan skeleton with three CPU threads.

outline of the partitioning of the Scan skeleton executed with three CPU threads can be seen in Figure 5.6.

Two things are important to notice here. The first thing is that nothing depends on the result of the last Scan block, i.e. that of the accelerator. The second thing is that the computation time of the two steps are quite different on a CPU and an accelerator (in particular a GPU). For a CPU the first and the second step take approximately the same amount of time to execute, as they perform the same number of computations (as many operations as the number of elements in the block). But for the GPU the first step takes much more time than the second step. The first step, with the local Scan, contains data dependencies between the computations, but in the second step all operations are independent. Due to the embarrassingly parallel nature and memory access pattern of the second step, a GPU can execute this very fast by using coalesced memory access.

In case the accelerator thread is included in the synchronization before the second step starts achieving good speedups can be hard. Because, if the partition ratio chosen to make the local Scan complete at the same time for the CPU and the accelerators, the accelerators will finish their second step quickly and go idle for a long time before the CPU threads are done with their second step. This will waste a lot of performance potential. As the accelerators are taking care of the last block, they do not need to be included in the synchronization.

Instead, the CPU threads synchronize themselves and start the second step when they have produced the temporary array without waiting for the accelerators. When the accelerators are finished with the first step, they will check that the CPU threads have produced the array of missing elements and then quickly finish their second step. This way it is possible to balance the partition ratio so that the CPU threads finish their second step at the same time as the accelerator, without wasting performance by having one PU wait for the other in the middle of the computation.

An alternative would be to redistribute the work in the second step with a different partition ratio, allowing both steps to be evenly balanced between CPU and accelerator. This would also allow the first CPU thread to participate in the work of the second step. However, this solution was not chosen as the Scan skeleton would not have the same interface with a single partition ratio as the other skeletons. The code complexity would also increase, as the missing values that should be added are not tied to one CPU thread/accelerator anymore. The speedup achieved by using all CPU threads would probably not be significant anyway, as the CPU threads would not be able to utilize the cached data from the first step, due to the new offset of the memory access pattern in the second step.

5.3 Auto-tuning of Skeletons

Because the hybrid backend is implemented on the same abstraction level as the other backends and not on a higher level, all features of the other backends are also applicable for the hybrid backend. This includes the already implemented optimal backend selection tuning. The hybrid backend, however, adds one new level of tuning: tuning of the partition ratio.

The tuning of a skeleton instance is intended to work in three steps:

1. **Machine specific tuning**. First the machine specific backend parameters are tuned. This includes for example maximum number of threads and blocks for the GPU. This type of tuning is implemented in SkePU 1, but not yet ported to SkePU 2. Most parameters tuned by this step are global for the machine, and not skeleton specific.

2. **Hybrid backend tuning**. The hybrid backend for the skeleton instance is tuned by building a model for prediction of the optimal partition ratio. This tuning step is implemented as a part of this thesis and is described in Section 5.4.

3. **Backend selection tuning**. Finally the backend selection tuning finds the best backend for different input sizes. The result of this tuning step is an execution plan. In most cases the optimal execution plan will use CPU for small, OpenMP for medium and CUDA/OpenCL for large input sizes. The backends evaluated in this step will also include the hybrid backend, which will likely be the best option for even larger input sizes. This type of tuning has an experimental implementation in SkePU 2.

Note that the hybrid backend tuning step (2), and the backend selection tuning step (3), have some overlap. The hybrid backend will, beyond tuning the optimal partitioning between the CPU and the accelerators for different input sizes, also fall back to the OpenMP or CUDA/OpenCL backends, if they are considered faster than hybrid execution. This will however come with a small performance overhead compared to running the OpenMP or CUDA/OpenCL backend directly, as the hybrid backend must first figure out which is faster: CPU-only, accelerator-only or hybrid execution. For example, if the hybrid backend finds that falling back to CPU-only execution with OpenMP is faster than using hybrid execution, the backend selection tuning will make benchmarks of two backends (the hybrid and the OpenMP backends), executing the same OpenMP implementation. These would thus result in similar execution times. However, running the OpenMP backend directly would presumably be a little faster, as it does not have the overhead of the hybrid backend. In these cases, the backend selection tuning can help to slightly improve the performance compared to just using the hybrid backend and relying on its fall-back mechanisms.

The user might choose to always use the hybrid backend, tuned with hybrid backend tuning and rely on it to fall back to OpenMP or CUDA/OpenCL for smaller input sizes. But one advantage of also using the backend selection tuning is that it includes the sequential CPU backend. This backend is likely optimal for really small input sizes. So by having both the hybrid tuning and the backend selection tuning, the hybrid backend and its tuning can be optimized for larger input sizes, while the backend selection tuning takes care of switching to a better backend for the really small problem sizes.

5.4 Implementation of Hybrid Backend Tuning

To let SkePU automatically predict the optimal partition ratio based on the input size a hybrid backend tuner was implemented. The tuner builds two execution time models, one for the CPU and one for the accelerator backend. These models are then used to predict the optimal partition ratio for a specific input data size.

Several solutions to the problem of partitioning a workload between heterogeneous PUs has been proposed over the years, including solutions based on theoretical performance [22], empirical performance [33, 41] and a mix of both [1]. The solution implemented in this thesis is based on empirical performance and resembles the implementation made by Luk et al. [33] in their parallel programing framework Qilin.

Because of how general the SkePU framework is, an auto-tuner that works perfectly in every case is almost impossible to implement. The execution time of a skeleton could for instance be bound by the size of the random access containers, or even of the values in the input data. Finding such relations would require either manual input from the user or very sophisticated machine learning algorithms with long training times, neither of which is desirable. Instead the hybrid tuner was implemented to work well for common cases where the execution time is bound by the size of the element-wise arrays. In other cases, the partition ratio can be manually tweaked by the user.

The hybrid tuner solution implemented in this thesis is based on empirical performance; it invokes the skeleton with different input sizes. Each skeleton instance is tuned once and the model built from that tuning can then be reused for all succeeding invocations of the skeleton instance. The hybrid tuner takes as parameters the upper and lower limit to tune for, as well as the number of input sizes to benchmark. The parameters also include the number of CPU threads to use and the number of accelerators to tune for. The specified number of sizes to benchmark are evenly distributed between the given input size limits. Benchmarks are made using the OpenMP backend and the OpenCL/CUDA backend. To avoid temporary fluctuations, five executions are made for each input size and backend and the median execution time is added to the execution time model.

One hybrid backend tuning will only be valid for a specific number of CPU threads and accelerators as measurements are done on the backends and not on individual PUs. The tuning also includes data transfer time for the accelerator backend, but does not separate it from the computation time. The tuner will thus underestimate the accelerator performance in cases where the data is already residing in accelerator memory. In those cases the total execution time of the accelerator will be shorter than the one measured during the tuning, as it only includes computation time and not data transfer time.

Listing 5.2 shows how to use the hybrid backend tuning to tune a skeleton instance with 16 threads and one accelerator. In the example the minimum and maximum limits to tune between as well as the number of tuning steps are defined. Compared to the example of how to use the hybrid backend with a manually set partition ratio seen in Listing 5.1, not much changes are required to let SkePU automatically find the best partition ratio.

```
const int NUM_CPU_THREADS = 16;
const int NUM_ACCELERATORS = 1;
const int MIN_SIZE = 1000;
const int MAX_SIZE = 1000000;
const int STEPS = 10;

// Tune
skepu2::backend::tuner::hybridTune(skeleton_instance, NUM_CPU_THREADS,
                                NUM_ACCELERATORS, MIN_SIZE, MAX_SIZE, STEPS);

// Use auto-tuning
skepu2::Vector<int> in, out;
skeleton_instance(out, in);
```

Listing 5.2: Example of using hybrid backend tuning.

The OpenMP backend is benchmarked with one less CPU thread specified in the call to the hybrid tuner, as one thread will be used to manage the accelerator backend.

5.4.1 Execution Time Model

The execution time is modeled as two linear curves: one for the CPU and one for the accelerator. The curves are built from the execution time samples taken by the hybrid tuner by using least-squares fitting. The result is two linear equations of the form:

$$t = ax + b \tag{5.1}$$

where t is the approximated execution time, x is the input data size in number of elements and a and b are the parameters found by least-squares fitting.

Let N be the problem size and R denote the (CPU) partition ratio, i.e. the fraction of work computed by the CPU. The partition size of the CPU is then NR and the partition size of the accelerator will thus be $N(1 - R)$. The predicted execution time curves can then be written as:

$$t_{cpu} = a_{cpu}NR + b_{cpu} \tag{5.2}$$

$$t_{acc} = a_{acc}N(1 - R) + b_{acc} \tag{5.3}$$

where the first equation is the predicted execution time of the CPU and the second equation is the predicted execution time of the accelerator. The workload is perfectly balanced between the CPU and the accelerators when $t_{cpu} = t_{acc}$. The optimal partition ratio can therefore be calculated by solving the following equation, formed by Equations (5.2) and (5.3):

$$a_{cpu}NR + b_{cpu} = a_{acc}N(1 - R) + b_{acc} \tag{5.4}$$

Solving for the optimal partition ratio R results in the equation:

$$R = \frac{a_{acc}N + b_{acc} - b_{cpu}}{N(a_{cpu} + a_{acc})} \tag{5.5}$$

Solving this equation is equivalent to finding the intersection point between Equations (5.2) and (5.3). In practice the calculated optimal partition ratio R might lie above 100% or below 0%. This indicates that the performance of either the CPU or the accelerator backend receptively is superior to the other, and that the entire workload should be offloaded to that PU. The prediction calculation will limit the value of R between 0 and 1, and let the hybrid backend implementation automatically fall back on the OpenMP or CUDA/OpenCL backend if one of the partitions is empty. Three kinds of predictions are possible: accelerator-only if

$R = 0$, CPU-only if $R = 1$ and hybrid execution if $0 < R < 1$. Falling back will come with a small performance overhead compared to using the OpenMP or CUDA/OpenCL backends directly, as the partition ratio must be predicted first. However, for non-trivial problem sizes, this overhead is negligible.

5.5 Implementation of the StarPU Backend

To give a fair comparison between the new hybrid backend and the old StarPU based hybrid execution implementation in SkePU 1, the StarPU library was re-integrated into SkePU 2. The new integration is based on the old version to give a similar performance, but with the more generic SkePU 2 API.

StarPU uses its own data management system, just like SkePU does. To keep SkePU's smart container API intact, the vector and matrix implementations in SkePU automatically transfer the control to the StarPU data management system whenever a data container is used in a skeleton with the StarPU backend. The control is taken back by SkePU once a container is used in a skeleton with one of the other backends. The memory management code of SkePU has not changed much with SkePU 2, allowing the StarPU integration in the vector and matrix containers to be more or less reused from the old integration. The backend implementation of the skeletons, on the other hand had to be rewritten to fit the more flexible meta-programming based interface of SkePU 2. The StarPU implementation was added as a separate backend in SkePU 2, just like the hybrid backend. In conjunction with the automatic data management, this allows the already existing backends to be used alongside the StarPU backend.

The abstraction gap between SkePU and StarPU is something that was already noticed during the integration of StarPU into SkePU 1 [11]. It has since grown even more in SkePU 2 with the increased use of meta-programing and other high-level C++ features. StarPU is implemented in C and uses raw pointers and run-time type casting to pass arguments. In the C++-based SkePU 2 on the other hand, argument lists are built at compile-time using variadic templates and parameter packs. However, thanks to the flexibility of both StarPU and SkePU 2 it was possible to re-integrate them, without making any changes to their APIs.

StarPU defines computational tasks using a structure called *codelet*. A codelet encapsulates multiple implementations variants of a single function. All different implementation variants in a codelet should produce the same result, but might be targeting different architectures or be optimized for certain hardware. Each skeleton in the StarPU backend of SkePU has at least one codelet with two implementation variants: one for multi-core CPU implemented using OpenMP and one for GPU implemented in CUDA. Some skeletons have multiple codelets for different variations of the skeleton (e.g. different codelets for one-dimensional and two-dimensional Reduce). When a skeleton is invoked with the StarPU backend selected, the workload is partitioned into a number of equally sized chunks. All chunks are then submitted as tasks to the StarPU runtime system to be scheduled and executed on a PU. As implementation variants for both CPU and GPU are given, the work can be executed simultaneously on multiple PUs, thus enabling hybrid execution. Because the StarPU library is implemented in C and enforces the use of void pointers to pass the data to a codelet, it was not possible to reuse the already existing OpenMP and CUDA backend implementations of the skeletons. Neither could the old SkePU 1 implementation variants for StarPU be reused because of the redesigns and generalizations made in SkePU 2. Instead new implementation variants had to be made as static member functions, relying on compile-time meta-programming to handle input containers of arbitrary number.

Because the CPU implementation variants of the codelets use OpenMP to form so called *parallel tasks* (i.e. a task encapsulating data parallelism), the regular HEFT schedulers could not be used. Instead the PHEFT (Parallel Heterogeneous Earliest Finish Time) scheduler was used. Each skeleton instance is assigned their own StarPU performance model to predict the performance of a task on different PUs. For the evaluation made in this thesis, StarPU's history based performance model was used. This performance model uses earlier execution time measurements of the same task size to predict the performance. An assumption is thus made that a few problem sizes are executed many times, as the same problem size must have been executed earlier for the model to actually be able to do a prediction. This might not always be the case for SkePU skeletons, but it fits the execution scheme used in the evaluation well, see Section 6.4. The history based performance model can easily be exchanged for any of StarPU's regression based performance models. Similar to the execution time model implemented in this thesis for the hybrid backend, the regression based models fit curves from measured execution times on the different PUs. These models will thus be better suited for applications where the input sizes to the skeletons varies, as they can predict the execution time of a task size it has never encountered.

Due to time constraints only some of the skeletons were implemented for the StarPU backend, namely Map, Reduce and MapReduce. Some variants of the skeletons (such as Reduce for matrices) and special features have not been implemented in the StarPU backend yet. Focus was put on skeletons and features that are frequently used in the SkePU example programs, as this would be enough for a comparison between StarPU and the new hybrid backend in the evaluation. The evaluation uses StarPU version 1.2.4; the latest version of StarPU at the time this thesis was written.

6 Evaluation

After the implementation phase, the new hybrid execution backend was evaluated. First, the correctness was evaluated to ensure that the new implementation of all skeletons gave the same results as the other backends. This evaluation step is described in Section 6.1.

Then, the performance of the new implementation was evaluated, to ensure that it resulted in shorter execution times. The performance of the new hybrid backend was examined on two levels. First on skeleton type level, where a single skeleton was considered at a time, described in Section 6.2. Then on generic applications consisting of one or multiple skeletons solving real world problems, described in Section 6.3. Finally, the new hybrid execution backend was compared to the ported StarPU backend, which is described in Section 6.4.

In the performance evaluations, the SkePU `Timer` class was used. The timing includes upload and download time to accelerators, but does not include the time it takes to allocate SkePU containers in main memory. In repeated benchmarks, new containers were allocated for each benchmark to minimize the impact of cached data.

CPU	2x Intel Xeon E5-2660
CPU base frequency	2.2 GHz
CPU total cores	16
CPU memory	64 GB
GPU	NVIDIA Tesla K20Xm
GPU cores	2688
GPU memory	6 GB
C++ compiler	GCC 4.9.2
CUDA compiler	nvcc 7.5.17

Table 6.1: Specification of evaluation systems.

The evaluation system is described by Table 6.1. The evaluation system consisted of two identical CPUs, working as a single CPU, and a GPU connected to the system via a PCIe (Peripheral Component Interconnect Express) bus connection.

6.1 Evaluation of Correctness

The first step of the evaluation was to verify the correctness of the new hybrid execution scheduling implementation. SkePU already features a sequential implementation of all skeletons to do this. Evaluation was done by comparing the output of the sequential backend and the new hybrid backend implementation for each skeleton type. A test framework was created with tests for each skeleton type. Skeletons with multiple variants for vectors and matrices had separate tests for the variants. Containers filled with randomized input data were used in the tests and for each skeleton to be tested the sequential and hybrid backends were called with these input containers. The hybrid backend was called with a fixed partition ratio to ensure hybrid execution was actually used. By comparing the result of the sequential and the hybrid backend, verifying the same result was produces by both backends, the correctness of the implementation could be ensured for general cases. In addition to this, multiple corner cases were tested to ensure that the fall-back mechanisms worked, as well as special skeleton specific features, such as different edge handling schemes in the MapOverlap skeleton. Once all skeletons had passed the correctness test, the performance of the hybrid backend could be evaluated.

6.2 Evaluation of Single Skeleton Performance

The performance of the hybrid execution implementations was first evaluated for single skeleton calls. One test program was created for each skeleton type, using a typical user function for that skeleton. The skeletons were tested with the OpenMP backend, the CUDA backend and the new hybrid execution backend. The hybrid backend used the hybrid auto-tuner to predict the optimal partition ratio. Tuning was performed at startup of the test program, and the same execution time model was then used for all invocations of the skeleton instance. The execution time of each skeleton and backend was recorded for a number of input sizes, ranging from $100,000$ to $4,000,000$ in increments of $100,000$. To minimize the impact of temporary variations in execution time due to other programs, each input size and backend was executed seven times and the median execution time was recorded.

The user functions (denoted f) used to instantiate the skeletons in this evaluation are presented below.

Map The Map skeleton calculated the sum of squares of two input arrays, with the user function: $f(a,b) = a^2 + b^2$.

Reduce The Reduce skeleton found the maximum odd number in the input array with the user function: $f(a,b) = \begin{cases} max(a,b) & \text{if } odd(a) \land odd(b) \\ a & \text{if } odd(a) \land even(b) \\ b & \text{if } even(a) \land odd(b) \\ 0 & \text{if } even(a) \land even(b) \end{cases}$.

MapReduce The MapReduce skeleton was implemented as a dot product, i.e. with the Map function: $f_M(a,b) = ab$ and the Reduce function: $f_R(a,b) = a + b$.

MapOverlap The MapOverlap skeleton calculated the sum of the elements inside the overlap area. If A denotes the set of values inside the overlap area, the user function can be written as: $f(A) = \sum_{a \in A} a$. The overlap size was set to 5 (i.e. with a total of 11 elements available to the user function). The out-of-bounds edge handling scheme was set to *duplicate*, using the closest value in the array for all out-of-bounds accesses.

Scan The Scan skeleton was implemented as a prefix sum, i.e. with the plus operator user function: $f(a,b) = a + b$.

For Reduce and MapOverlap, only the vector implementations were tested.

Application	Algorithm	Skeletons
CMA	Cumulative moving average	Map<1>, Scan
Dotproduct	Dot product	MapReduce<2>
Gaussian	One-dimensional Gaussian filter	MapOverlap
Mandelbrot	Mandelbrot fractal	Map<0>
PPMCC	Pearson product-moment correlation coefficient	Reduce, MapReduce<1>, MapReduce<2>
PSNR	Peak signal to noise ratio	Map<2>, MapReduce<2>
Taylor	Taylor series expansion of $\log(1+x)$	MapReduce<0>

Table 6.2: List of applications used in the evaluation.

6.3 Evaluation of Generic Application Performance

To evaluate performance in a more realistic context, an application evaluation was performed. In this evaluation seven different test applications, most of them available through the SkePU website [1], was used. These applications included a Mandelbrot fractal generator and Taylor expansion. The full list of example applications used in the evaluation is presented in Table 6.2. In the skeleton column, the number within <> denotes the *arity* of the skeleton instance, i.e. how many element-wise accessed input containers are used. The Gaussian filter application was executed with an overlap of 5 elements on each side of the center element.

Each application was executed with five configurations: four different backends and an *oracle*. As a baseline for speedup calculation, the applications were first executed with the sequential CPU backend. The applications were then executed with the OpenMP, CUDA and hybrid backends, and the speedups over the sequential CPU backend were calculated for each of these backends. Each skeleton instance was tuned with the hybrid auto-tuner and the optimal partition ratio was predicted for each skeleton instance by the models. Finally, as an upper bound for the performance of the hybrid backend, the applications were executed with an oracle. By executing the hybrid backend with an optimally chosen partition ratio, the oracle shows the highest possible speedup achievable with the hybrid backend implementation. This method has been used in several papers to show the upper bound of hybrid execution implementations [22, 33, 41]. The oracle executed the hybrid backend with manually set partition ratios, ranging from 0% to 100% in steps of 5 percentage points for each skeleton instance in the application. This results in 21^k cases per application, where k is the number of skeleton instances used in the application. The best execution time out of these tested was saved as the resulting execution time of the oracle, i.e. an approximation of the best possible execution time of the hybrid backend. This result gives a hint of the maximum possible performance of the hybrid backend including its partitioning, but might miss the true optimal ratio due to its discretization of the tested ratios.

To reduce the impact of temporary execution time fluctuations, each benchmark, including all partition ratio benchmarks tested for the oracle, was executed seven times and the median execution time was used.

6.4 Evaluation of Performance Compared to StarPU

Lastly an evaluation of the performance compared to the dynamic hybrid execution scheduling provided by the StarPU backed was made. Because StarPU uses runtime schedulers that learn over time, a fair comparison should let the StarPU performance models have a chance to train and improve their predictions. To do this, the same skeleton instances were invoked 30 times in a row, giving StarPU a chance to improve and stabilize its performance. The OpenMP and CUDA backends were included in the evaluation as a comparison of single PU performance. The hybrid backend was tuned a single time in the beginning and that execution time model was then used for all repeated invocations. All backends were invoked 30 times in a row and

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

the execution time was measured each time. No filtering of temporary fluctuations was made as in the other performance evaluations. The performance of the StarPU backend is highly dependent on the number of tasks each skeleton call is divided into. The task size was therefore manually chosen for each tested skeleton instance, to give the best possible performance from this backend. StarPU saves the performance models for each skeleton instance on disk for reuse between runs. These were discarded before the test programs were executed to make sure that the performance models were built from scratch and not reused from earlier executions. For each benchmark new containers were allocated to minimize the influence of caches. The same user functions were used as in the single skeleton evaluation, see Section 6.2.

7 Results

In this chapter the results of the performance evaluations are presented. First the single skeleton and generic application results are presented in Sections 7.1 and 7.2 respectively, then the results of the comparison to the StarPU-backend reimplementation are presented in Section 7.3.

The execution times include data transfer time to the accelerators, but do not include time for allocation of data containers in CPU memory. Neither do they include the tuning time for the skeletons.

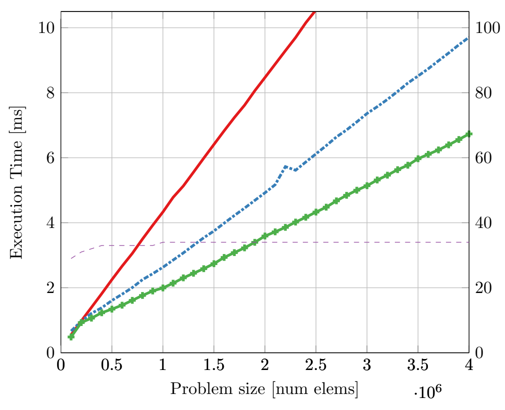
7.1 Single Skeleton Performance

The results of the single skeleton performance evaluation are presented in Figure 7.1. In the graphs the *Hybrid* line is the new hybrid backend, tuned with the auto-tuner, using the CUDA backend for the accelerator partition. The *OpenMP* and *CUDA* lines show the execution time of these backends with 16 CPU threads and one GPU respectively. The predicted partition ratio for all problem sizes is also shown as a separate line. For most skeletons this line shows that the hybrid backend falls back to CPU-only execution for small problem sizes (partition ratio at 100%) and offloads more and more of the computation to the GPU with an increased problem size.

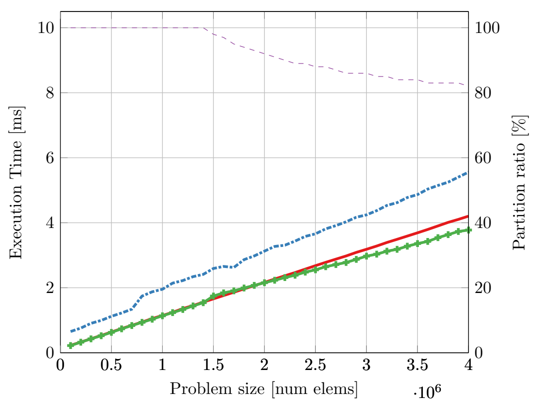
In the graphs we can see that hybrid execution can improve the execution time for all skeletons, at least for larger problem sizes. Among these five tested skeleton instances we can see that hybridization works best for Map, MapReduce and MapOverlap, while the gains are smaller for Reduce and Scan.

7.2 Generic Application Performance

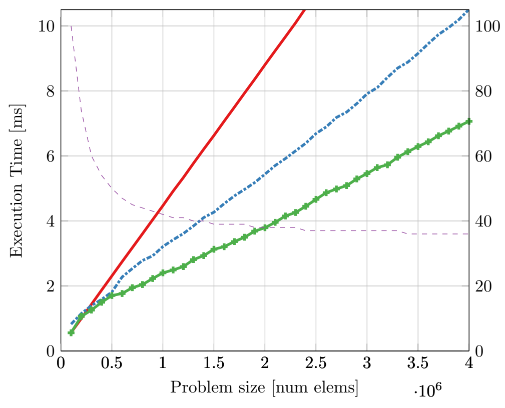
The result of the generic application evaluation is presented in Figure 7.2. The bars show the speedup of the tested backends compared to the sequential CPU implementation. Here *Hybrid* is the hybrid backend tuned with the auto-tuner, using the CUDA backend for the accelerator partition. The *Oracle* bar shows the speedup of the hybrid backend (with the CUDA backend for the accelerator partition), executed with the optimal combination of the 21 partition ratios that were tested per skeleton instance. The *OpenMP* and *CUDA* bars show the execution time of these backends with 16 CPU threads and one GPU respectively.



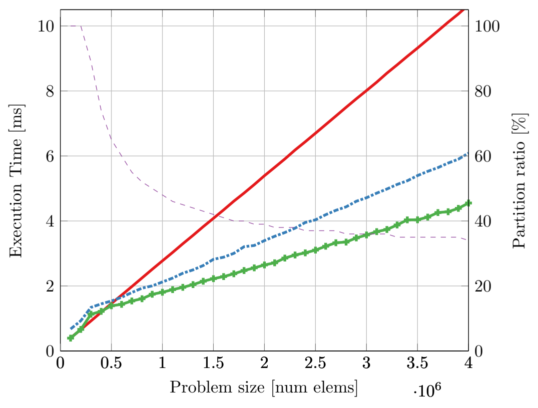
(a) Map skeleton



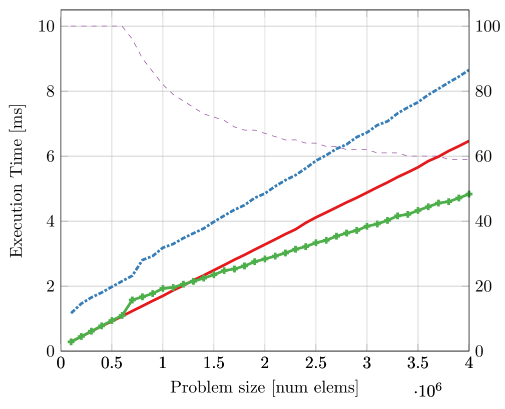
(b) Reduce skeleton



(c) MapReduce skeleton



(d) MapOverlap skeleton



(e) Scan skeleton

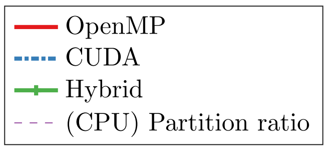


Figure 7.1: Execution time of individual skeletons.

In the diagram we can see that the hybrid backend improves upon, or at least matches the performance of the OpenMP and CUDA backends in all cases except for the PSNR application. The oracle confirms that the overhead of the hybrid backend is negligible in cases when the backend falls back to CPU-only or GPU-only execution, something that can be clearly seen in the bars for the PSNR and Taylor applications.

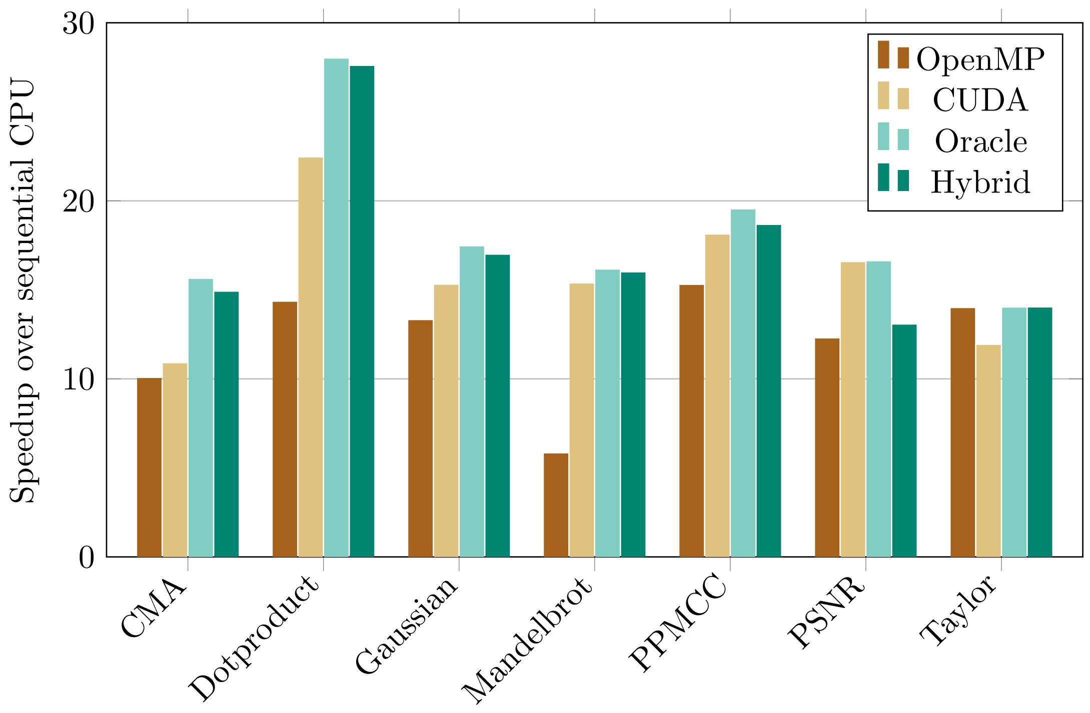
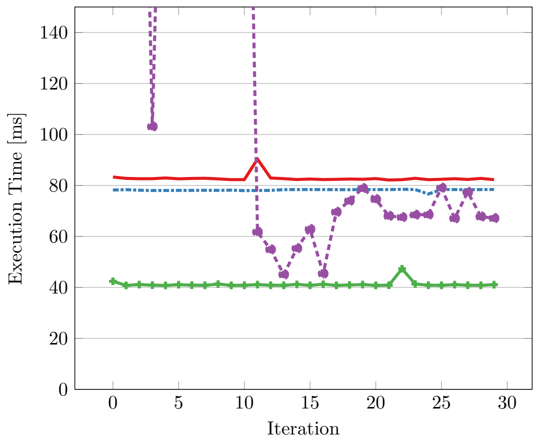


Figure 7.2: Speedup comparison of generic applications.

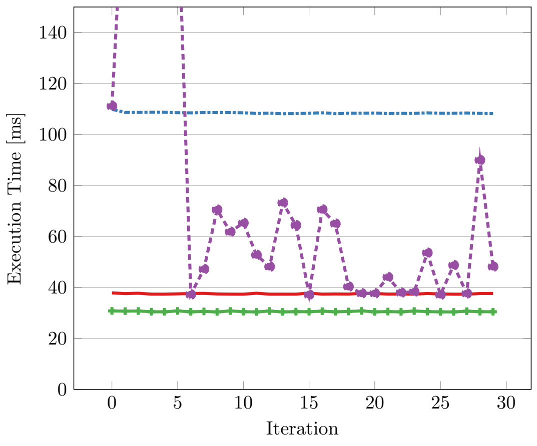
7.3 Comparison to StarPU Performance

The results of the comparison between the new hybrid backend and the StarPU backend are presented in Figure 7.3. The graphs show the execution time of the four tested backends after repeated invocations of the same skeleton instance with the same input size. Both the hybrid backend and the StarPU backend divides the work between the CPU and the GPU. The optimal number of tasks to divide the skeleton workload into for the StarPU backend was manually found for each skeleton instance and was: 6 for Map, 3 for Reduce and 14 for MapReduce.

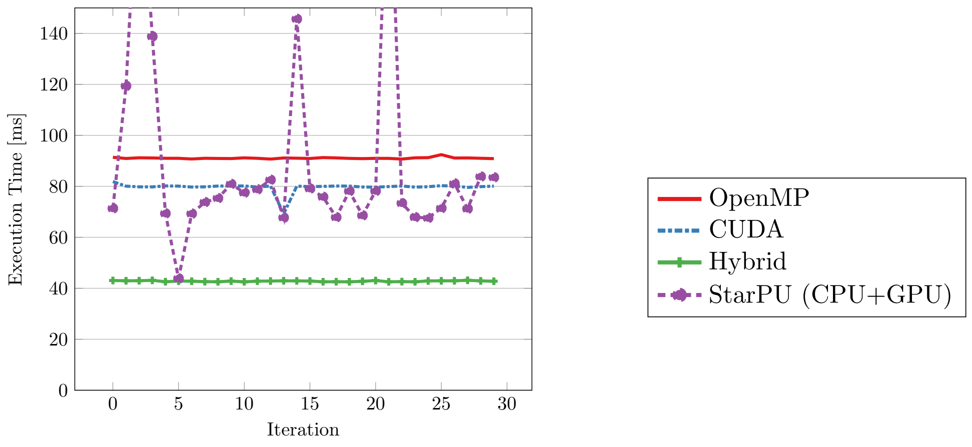
In the graphs the OpenMP, CUDA as well as the new hybrid backend show an even execution time for repeated invocations. The StarPU line however, shows more variation between invocations, although the execution time improves and stabilizes somewhat with time. Hybrid execution with StarPU does not pay off until after five or ten repeated executions in some of the tests.



(a) Map skeleton, $20 \cdot 10^6$ elements.



(b) Reduce skeleton, $90 \cdot 10^6$ elements.



(c) MapReduce skeleton, $20 \cdot 10^6$ elements.

Figure 7.3: Execution time of repeated invocations of the same skeleton.

8 Discussion

This chapter discusses the findings of the thesis, as well as the methodology that was used. The chapter is divided as follows: the results of the evaluations are discussed in Section 8.1, then the methodology is analyzed in Section 8.2 and finally the work is discussed from a wider perspective in Section 8.3.

8.1 Results

During the development and correctness testing of the hybrid backend, many bugs and inconsistencies were found in the already existing backends. Bugs are expected as the SkePU version this work was based on is in a preview state after an extensive code refactoring. Bug reports and fixes were submitted to the SkePU project and most of these bugs will be fixed in the next public release of SkePU. The majority of bugs were found in the MapOverlap implementation and is the result of the many corner cases of this skeleton. An exhaustive regression test suite for SkePU is hard to create due to all implementation variants, all possible user functions and their flexibility with input arguments, but many of the bugs encountered could have been found by common test cases.

A small test framework for the hybrid backend was created and the hybrid implementation passed all tests. Some tests failed, however, when a specific accelerator backend with a specific number of devices was executed, due to a few bugs still residing in the CUDA/OpenCL multi-device implementations.

8.1.1 Single Skeleton Performance

From the single skeleton performance evaluation it is apparent that performance can be gained by using hybrid execution. Hybrid execution proves to have better performance than the OpenMP and CUDA backends for all five skeletons, at least for larger input sizes. For smaller input sizes the kernel launch overhead and memory copying time of the GPU makes up a larger fraction of the execution time, reducing the gains. For most skeletons the hybrid backend even manages to match the performance of the fastest of the OpenMP and CUDA backends for small input sizes, by automatically switching to CPU-only or GPU-only execution. For the Scan skeleton however, the hybrid curve takes a leap and becomes slower than the OpenMP and CUDA curves, as the prediction algorithm overestimates the GPU performance. This is likely due to the differences in the OpenMP/CUDA and hybrid implementations of Scan, causing the performance of the individual backends and the performance of the partitions in the hybrid backend to differ slightly. In particular the accelerator part of the hybrid backend where the division of the computation into two steps leads to an increased overhead.

Hybrid execution seems to work better for embarrassingly parallel problems, such as the Map skeleton, than for skeletons with more dependencies, like the Reduce and the Scan skeletons.

8.1.2 Generic Application Performance

The result of the generic application evaluation shows that hybrid execution gives a speedup over the performance of the OpenMP and CUDA backends for most of the applications. PSNR and Taylor are the only exceptions where hybrid execution does not gain any performance (at least not for the problem sizes that were tested).

By comparing the upper bound of the oracle to the hybrid backend bar we can see that the auto-tuning manages to find good partition ratios, but there is still some room for improvement when comparing to the oracle.

For the Mandelbrot application the speedup of the OpenMP backend is low (in the optimal case it should be equal to the number of CPU cores, that is 16) compared to the other applications. This is because the application was executed with a relatively small input size, where the overhead of multi-core execution of the OpenMP backend can be noticed. Mandelbrot is a very GPU-friendly application due to the high workload per user function invocation and because no input data is required. Increasing the problem size would result in massive speedups for the GPU, around 100 to 1000 times faster than the sequential execution. In these cases using hybrid execution will not make much of a difference in performance due to the limited speedup of the OpenMP backend. Therefore a smaller problem size was used in the experiment, showing that performance can be gained with hybridization, even for smaller input sizes.

In the applications PSNR and Taylor either CPU-only or GPU-only execution is the fastest, according to the oracle. For the Taylor application, the hybrid backend finds CPU-only execution to be the fastest and therefore matches the performance of the top performing single PU backend. However, for the PSNR application the speedup of the hybrid backend is lower than for the fastest single PU backend—the CUDA backend. In this case the hybrid tuning predicts that the optimal partition ratio is close to 40% for the Map skeleton used the application, and 100% for the MapReduce skeleton. While this might be the optimal choice for the individual skeletons, it is not the optimal solution when both skeletons are executed consecutively. The oracle proves this. It choses the same partition ratio (0%, GPU-only execution) for both skeletons, as this will make the data partition sizes equal and removes the need for the data transfer between the skeleton invocations. As the data transfer time takes up a large proportion of the overall execution time, this leads to a better result when the two skeletons are executed after one another.

8.1.3 Comparison to StarPU Performance

The comparison to the StarPU backend reveals the problem reported with StarPU from the SkePU 1 implementation: the performance is unreliable. For the Map and MapReduce skeletons, the StarPU backend improves over the performance of the OpenMP and CUDA backends. The StarPU backend is never faster than the hybrid backend, although a few executions of the StarPU backend are very close the the execution time of the hybrid backend. For the Reduce skeleton, the StarPU backend only manages to match the performance of the OpenMP backend, but not improve upon it. The Reduce skeleton runs much faster on CPU than on GPU, which makes the possible speedups of hybrid execution small, as can be seen for the hybrid backend. The overhead of the scheduling in StarPU makes it even harder for this backend to gain any performance by hybrid execution.

It is expected that the StarPU backend should have a higher execution time than the hybrid backend, as the runtime scheduler in StarPU comes with a substantial overhead. In the case of Map, the execution time of the OpenMP and CUDA backends are almost equal, indicating that the optimal partition ratio for the hybrid backend is close to 50%. The shortest execution time that can be expected when dividing the workload between two PUs is half the execution time of a single PU. As the execution time of the hybrid backend is close to half the execution time of the single PU, the speedup of the hybrid backend is close to the highest possible speedup with hybrid execution using two PUs in this case. The StarPU backend should not be able to improve upon this execution time for this skeleton instance.

The graphs show to some extent the mentioned need for warm-up runs; the StarPU performance is very bad at the first few invocations and then improves. But the execution time is not stable even after 30 repeated executions. The reason for this might be due to the performance models not being trained enough yet, but is more likely because of the task size. Since the manually chosen number of tasks to divide the skeleton invocations into (chosen as the one that minimized the best execution time) is quite low—only between 3 and 14—the workload of each task is relatively large. If the scheduler makes a single misprediction the result might be a significant load imbalance, where one PU goes idle for a long time at the end of the computation. A higher number of tasks should to some extent give better execution times, but when tried, it proved to increase overheads and give longer execution times instead. The number of tasks is remarkably small for the Reduce skeleton, something that might explain the spikes in execution time for this skeleton. Increasing the number of tasks from three to four results in a much slower execution time. There might still be details in the implementation of the StarPU backend for this skeleton that can be improved.

The performance of the StarPU backend might gain from even larger problem sizes than the ones used in the evaluation, as the scheduling time would be reduced in relation to the computation time. In that case, more tasks per skeleton could be used to make the scheduling easier and minimize the risk of one PU going idle for a long time at the end of the computation. It should also be noted that these skeleton instances are well suited for the hybrid backend because the workload of the user function is identical for each user function invocation. In cases where the computational work of the user function differs between each invocation, a dynamic scheduling approach might be preferable as it can better adapt to the uneven workload. In these cases the StarPU backend might still be an interesting option, as long as the input size is large enough to hide the overhead of the dynamic scheduling.

The performance of the StarPU backend is highly dependent on finding the best number of tasks to divide the skeleton invocation into; a bad choice results in drastic performance drops. This need for hand-tuning parameters is a major drawback of the StarPU backend in comparison with the new hybrid backend. Hand tuning of the number of tasks is far more time consuming than the implementation of the application.

Some variations in execution time of the other backends can be seen, especially for the OpenMP and hybrid backends. The performance of the GPU is more even, suggesting that the variations come from other operating system processes running on the CPU. A slightly higher execution time can be seen for the first measurement of the CUDA backend. This is because the CUDA runtime uses lazy initialization and starts initializing the runtime at the first CUDA call. This overhead is relatively small and will be negligible for even larger problem sizes. In the single skeleton evaluation this effect is filtered out from the other evaluations because of the use of median execution time. One can also notice that the first few execution time measurements of the StarPU backend are especially high, at least for some skeletons. This is because StarPU as a default does not use the PHEFT scheduler until the performance model has been stabilized. For the history based performance model used in this integration, this requires at least 10 measurements on a particular input size to call the prediction for that input size reliable. For the Map skeleton a significant improvement can be seen after 10 invocations when StarPU switches to the PHEFT scheduler. Before the performance model has been stabilized, the much less sophisticated *eager* scheduler is used. The eager scheduler is a greedy type of scheduler where each task is assigned to a worker as soon as the worker is done with its last task.

8.2 Method

The hybrid backend is implemented using static partitioning and static scheduling. This was chosen as many related frameworks have proven it to be a well working choice and because the old StarPU integration with its dynamic task based approach had problems with its overheads. A static workload partitioning scheme works well as long as the execution time of each user function call is approximately the same. When more work is required for some calls, the auto-tuning will result in load imbalance between the CPU and the accelerator. However, in cases where the imbalanced workload of each user function call is already known at compile time, the programmer can manually set a good partition ratio as long as the problem size is constant.

No changes were required in the data management code, as it already had support for making copies of subsections of containers. However, the implementation has one limitation that could not be easily solved: when hybrid execution is used, writing to random access containers will result in undefined behavior as the write operations can not be synchronized between the PUs. The elements written by the CPU will be overwritten in these cases by the elements written by the accelerator (or vice versa), as the smart containers can not check the validity of single data elements. In reality this is not a big problem, as the write order can not be guaranteed even for single PU execution. However, skeleton instances can be implemented such that each element of a random access container is only written to (at most) once. To guarantee correctness in these cases, the programmer must use single PU execution instead of hybrid execution.

8.2.1 Design Choices for the Auto-tuning

For the auto-tuning implementation linear execution time models were used. As all data-parallel skeletons in SkePU scales linearly with the element-wise input size (assuming an $\mathcal{O}(1)$ user function), this was deemed sufficient. The execution time curves seen in Figure 7.1 further confirms that this is a good approximation. The tuner does not distinguish between computation time and data transfer time for the accelerator backend. This might lead to underutilization of the accelerators in cases where data transfer is not needed. The tuner might be extended with this in the future while still using the same partitioning implementation.

8.2.2 Interpretation of Number of Threads in the Hybrid Backend

One design choice made in the hybrid backend was how to interpret the number of CPU threads the programmer explicitly requests. As one CPU thread will be used for the accelerator backend, there are two possible interpretations of using the hybrid backend with N threads. This could ether be including the accelerator thread, meaning that N threads are actually used and N is generally chosen to be the same as the number of processor cores. This will—somewhat counterintuitive—result in that only $N - 1$ threads are used for the CPU partition of the workload. The other choice, to exclude the accelerator thread, means that N requested threads gives N threads for the CPU partition of the workload plus one extra thread for the accelerator partition. This means that if the user requests the same number of threads as the number of processor cores, the hybrid backend will actually use one more thread than the number of cores, resulting in decreased performance. In the end, the first alternative was chosen as it was more coherent with the OpenMP backend. The best performing number of threads for the OpenMP and hybrid backends will then be the same, i.e. the same as the

```
// Setup for OpenMP backend
skepu2::BackendSpec specCPU(skepu2::Backend::Type::OpenMP);
specCPU.setCPUThreads(8); // Uses eight CPU threads
skeleton_instance.setBackend(specCPU);

// Setup for hybrid backend
skepu2::BackendSpec specHy(skepu2::Backend::Type::Hybrid);
specHy.setDevices(1);
specHy.setCPUThreads(8); // Uses eight CPU threads, one for accelerator partition,
    seven for CPU partition
skeleton_instance.setBackend(specHy);
```

Listing 8.1: Example of setting optimal number of threads on a eight-core machine.

number of processor cores. This also means that the number of threads does not need to be updated when changing backend from OpenMP to hybrid or vice versa. Listing 8.1 shows that the optimal choice for computation on a eight-core machine is coherent between the OpenMP and hybrid backends, as eight threads is specified for both backends.

8.2.3 Blocking Accelerator Calls

As discussed earlier in this thesis, the skeleton calls in the CUDA and OpenCL backends are made as blocking calls, locking one CPU thread to handle the accelerator backend until its computation is done. Performance could have been increased somewhat if the accelerator backends had been rewritten to use asynchronous calls, allowing the CPU thread to be used in the computation of the CPU partition once the data was uploaded to the accelerator memory and it was busy working. However, this would create a load balancing problem on the CPU side, as this extra thread would not have as much time to do computations as the other CPU threads. Performing load balancing can easily be activated with OpenMP, but in the end the performance gains would probably not be significant anyway, especially for CPUs with many cores.

8.3 The Work in a Wider Context

Heterogeneous computing is essential for the future in high performance computation and the parallel programming frameworks must therefore have better support for it. With the hybrid execution support in SkePU, programmers will have a user friendly framework for developing applications on heterogeneous architectures. The framework hides the details of accelerator programming and workload partitioning, and the automatic tuning takes care of all system specific settings. This lets the programmer focus on the application rather than the optimizations for a specific hardware. It also lets the programmer view the application as a simple sequential program, rather than a complex parallel program with hybrid execution featuring different kinds of hardware. Hiding the complex parts of the applications will reduce the risk of hard-to-find bugs and errors. The raised abstraction bar will also make heterogeneous computing open for many more programmers, without really requiring any prior knowledge in accelerator or multi-core programming.

The main goal of this thesis is to reduce the execution time and improve usage of the available hardware in SkePU. Apart from the obvious gains of shorter computation times and the possibility of running larger computations within the same time constraints, this might also result in lower power usage for the same computation due to better hardware utilization. More efficient algorithms and better utilization of the hardware can reduce the need for hardware upgrades and extend the lifetime of the computer's components. But one must also consider that more efficient solutions might lead to the opposite effect; in our case, that more people wants to do more large scale computations, as they can be done in shorter time. This will in turn lead to a higher demand of new hardware. Although, if that hardware will have a longer lifetime, the gains might still be better in the long run.

9 Future work

Although this work has made a significant contribution to the SkePU project, there are many things that can still be improved upon. Some topics left for future work is presented in this chapter.

9.1 Auto-tuning for Multiple Accelerators

Despite not being shown in this thesis, there is already working support for hybrid execution with multiple accelerators because the CUDA/OpenCL backends are reused by the hybrid backend. But the workload partitioning in these backends is still very limited. Workload is divided evenly between the accelerators assuming they are of the same model or at least have similar performance. This is not always the case. The tuning ought to be extended to support tuning of multiple accelerators to find the optimal partitioning between them. The multi-accelerator tuning should preferably be done in a new tuning step before the hybrid tuning step. This new multi-accelerator tuning step could find the optimal partition ratio between the accelerators, and let this optimal partitioning between the accelerators to be used in the hybrid tuning step to tune between the CPU and the accelerator backend, just as it is done now. This way, the hybrid tuner would not need to be changed and would not need to know about the actual number of accelerators, as they are treated as one.

9.2 Hybrid Tuning of Skeletons with Custom Data Property Requirements

The hybrid tuning implemented in this thesis does not support tuning of skeleton instances where the data in the containers must fulfill certain properties. Containers used during the tuning are always instantiated with randomized data. Skeletons could, however, assume that the values of the input arrays are always within a specific range. Skeleton invocations where the input data fails to fulfill these properties result in undefined behavior and in the worst case the application will crash during tuning.

Even more problematic are skeletons with custom defined structs as input data. Structs are not randomized at construction in SkePU, but rather default initialized. If the struct does not provide its own default constructor, the struct members will contain the data that

```cpp
struct Particle {
    float x, y, z;
    float vx, vy, vz;
    float mass;
};

constexpr float G [[skepu::userconstant]] = 6.674e-11;

// User function for updating a single particle
Particle update(Index1D index, Particle p1, const Vec<Particle> particles) {
    for(size_t j = 0; j < particles.size; ++j) {
        Particle p2 = particles[j];

        float dist = calculateDist(p1, p2);

        float force = G * p1.mass * p2.mass / (dist*dist); // Assumes dist != 0
        float accel = force / p1.mass; // Assumes p1.mass != 0
        ...
    }
    ...
}
```

Listing 9.1: Example of N-body simulation with custom data property requirements.

happened to reside in that memory cell, which is likely to be a zero. In physical computing this is a problem as many computations assume some values to be non-zero. A typical example is N-body simulation. A small example of this is shown in Listing 9.1, adapted from the example implementation provided with the SkePU 2 source code. In this example, two assumptions are made about the input data. First that no two particles share the exact same position, causing the distance between them to be zero. A zero distance might result in a crash due to a division by zero. Using randomized input data this is highly unlikely, but for default initialized data this is a problem as some particles are likely to be positioned in origin where $x = y = z = 0$. The second assumption is that the mass is non-zero, which is also unlikely for randomized input, but a problem for default initialized structs where a zero value is likely. In addition to this, the skeleton could have even more hard-to-meet requirements. For example that the initial kinematic energy should be at a certain level, putting a requirement not only on the members of every individual struct, but on all structs together.

In the future, the programmer should be given the opportunity to provide custom benchmarking functions to set up the containers and execute a skeleton instance, to allow for tuning of skeletons with custom data property requirements.

9.3 Performance Improvements of Subsequent Skeleton Invocation

The data management implementation of SkePU lacks some functionality, as it was not intended to be used for hybrid execution at its initial implementation. The major point is the performance of sequential invocations. When the hybrid backend starts the execution of a skeleton, the containers used in the skeleton must be updated in the CPU memory by calling the `updateHost()` member function. This function forces the *entire* container to be updated in the CPU memory, even the part of the container that will only be accessed by the accelerator. Sequential invocations of a skeleton result in performance penalties due to this behavior. For example: consider the case when two skeletons are used and the output data of the first skeleton is used as input to the second skeleton. If the partition ratio is 50% for both skeletons, there should be no need to synchronize the data as the half of the output produced by the accelerator in the first skeleton is already in the accelerator memory before execution of the second skeleton. But since the `updateHost()` member function cannot define an interval to update, the entire container must first be updated in CPU memory, invalidating the half residing in accelerator memory. When execution of the second skeleton starts, half of the container must be re-uploaded to the accelerator even if its content has not been changed in CPU memory. Optimally, only the part needed for the CPU should be updated, causing no memory transfers in this example. The reduced performance of this extra and unnecessary data copying can be noticed in the PSNR application in the evaluation.

9.4 Improved Tuning of Matrix Skeletons

Auto-tuning of skeletons with matrices as input can be improved upon. The auto-tuning implementation for matrices is limited today as it only trains on square matrices using the total size of the matrix as the problem size. In reality tuning of matrices is a two-dimensional problem, as matrices can have any height-width ratio. This would require a hyperplane to describe the execution time, which complicates the execution time model considerably. The current solution works well for the Map skeleton, as it uses the same implementation for vectors and matrices (and because matrices are internally implemented as a sequential array of elements). The workload can be split mid-row between the CPU and the accelerator. This means that the skeleton will have the same execution time for a $M \times N$ matrix as for a $N \times M$ matrix when the same partition ratio is used, because the partitions in the hybrid backend will be of the same sizes for both matrices. For the MapReduce and MapOverlap skeletons this is not the case, as they split the workload row-wise between the CPU and the accelerators. For performance reasons they only operate on whole rows. In that case the performance will be very different for a $M \times N$ and a $N \times M$ matrix when using the same partition ratio. Optimally, in the future this should be taken into consideration somehow by the auto-tuner.

9.5 Adaptive Tuning

Auto-tuning of the hybrid backend uses the OpenMP and CUDA/OpenCL backends for benchmarking. This might not give the best predictions of the partition ratio, as the performance of the partitions in the hybrid backend not always matches the performance of the individual backends. This is especially the case for the Scan skeleton, as it required a different partitioning approach compared to the other backends. One way to improve this would be to let each execution of the hybrid backend take timing measurements of the CPU and accelerator partitions and update the execution time model with those real execution times. The tuning would thus create an initial model that would be further refined the more times the skeleton was invoked. One might have to be careful not to over-fit the model in the case of many executions of the same input size.

10 Conclusion

In this thesis a new hybrid execution backend for the skeleton programming framework SkePU 2 is presented. The backend is capable of dividing the workload of a SkePU skeleton between any number of CPU cores and any number of accelerators to let them simultaneously execute the skeleton. A new auto-tuner is implemented to predict how to partition a skeleton workload between the processing units by benchmarking the skeleton instance. Gains in execution time is shown for all skeletons, proving the usability of hybrid execution in SkePU. This thesis also compares the new hybrid backend to the old, experimental hybrid execution implementation from SkePU 1 that was based on the StarPU runtime system. It is shown that the new hybrid backend is superior to the old one, due to its lower overhead and a more stable and predictable execution time.

Even if the auto-tuner presented in this thesis has some drawbacks, this work still serves as a good foundation for future experiments with hybrid execution in SkePU. This thesis has proven that the workload partitioning implementations work well for all skeletons and future improvements to the auto-tuning can thus keep using the same partitioning schemes that were implemented in this thesis.

We can now revisit and answer the research questions formulated in Section 1.3.

1. *How can the workload of the skeletons in SkePU be partitioned for execution on heterogeneous processing units?*

   Partitioning of the workload is done according to a single parameter: the *partition ratio.* It defines how large proportion of the workload should be computed by the CPU with the rest being computed by the accelerators. The work is executed simultaneously on heterogeneous processing units by letting one CPU thread compute the accelerator partition using the already existing CUDA/OpenCL backends and letting the rest of the CPU threads work on the CPU partition. The details of the workload partitioning is further discussed in Sections 5.1 and 5.2.

2. *How can the optimal workload partitioning in the new hybrid execution backend be predicted for different types of processing units?*

   The optimal partitioning is predicted for a skeleton instance by measuring the execution time of the CPU and the accelerators using the already implemented backends on randomized input data of different input sizes. During this tuning, two linear performance models are built that are used to find the theoretically optimal partitioning between the CPU and the accelerators. This estimation works as the implementation of the hybrid backend reuses the CUDA/OpenCL backends and borrows from the OpenMP backend implementation. The tuning is further discussed in Section 5.3.

3. *How can the overhead of the hybrid execution implementation be kept low and without the need for warm-up runs?*

   The overhead is kept low by using a static load balancing approach with a minimal set of partitioning parameters. The warm-up runs are exchanged for an explicit, one-time auto-tuning per skeleton instance, where the performance of the PUs are measured. The advantages of this are clearly shown in the comparison against the StarPU backend, see Section 7.3.

10.1 Presentation of Results

The new implementation of hybrid execution in SkePU 2, including the hybrid backend and the hybrid auto-tuning presented in this thesis, will be released as open source in the upcoming version of SkePU. The release is planned to a couple of weeks after the publication of this thesis. The StarPU reintegration will be handed over to the SkePU development team to be completed and released in some forthcoming version of SkePU.

A paper on the implementation of the new hybrid backend and the auto-tuner has been accepted for HLPP 2018 (11th International Symposium on High-Level Parallel Programming and Applications) and will be presented in Orléans, France 13th July 2018 [38].

Bibliography

[1] Fernando Alexandre, Ricardo Marques, and Hervé Paulino. "On the support of task-parallel algorithmic skeletons for multi-GPU computing". In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM. 2014, pp. 880–885.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures". In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198.

[3] Stefan Breuer, Michel Steuwer, and Sergei Gorlatch. "Extending the SkelCL skeleton library for stencil computations on multi-GPU systems". In: *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*. 2014, pp. 15–21.

[4] Antal Buss, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger, et al. "STAPL: Standard Template Adaptive Parallel Library". In: *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. ACM. 2010, p. 14.

[5] Linchuan Chen, Xin Huo, and Gagan Agrawal. "Accelerating MapReduce on a coupled CPU-GPU architecture". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press. 2012, p. 25.

[6] Eric S. Chung, Peter A. Milder, James C Hoe, and Ken Mai. "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?" In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2010, pp. 225–236.

[7] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[8] Sylvain Contassot-Vivier and Stéphane Vialle. "Algorithmic scheme for hybrid computing with CPU, Xeon-Phi/MIC and GPU devices on a single machine". In: *Parallel Computing: On the Road to Exascale* 27 (2016), p. 25.

[9] *CUDA C Programming Guide*. Version 9.1. NVIDIA. Mar. 2018, pp. 8, 70–84, 172.

[10] Usman Dastgeer and Christoph Kessler. "Smart containers and skeleton programming for GPU-based systems". In: *International journal of parallel programming* 44.3 (2016), pp. 506–530.

[11] Usman Dastgeer, Christoph Kessler, and Samuel Thibault. “Flexible Runtime Support for Efficient Skeleton Programming on Heterogeneous GPU-based Systems.” In: *PARCO*. 2011, pp. 159–166.

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

[13] Usman Dastgeer, Lu Li, and Christoph Kessler. “The PEPPHER composition tool: performance-aware composition for GPU-based systems”. In: *Computing* 96.12 (2014), pp. 1195–1211.

[14] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[15] Romain Dolbeau, Stéphane Bihan, and François Bodin. “HMPP: A hybrid multi-core parallel programming environment”. In: *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*. Vol. 28. 2007.

[16] Alejandro Duran and Michael Klemm. “The Intel® many integrated core architecture”. In: *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE. 2012, pp. 365–366.

[17] Johan Enmyren. “A Skeleton Programming Library for Multicore CPU and Multi-GPU Systems”. LIU-IDA/LITH-EX-A–10/037–SE. MA thesis. Linköping, Sweden: Linköping University, 2010.

[18] Johan Enmyren and Christoph Kessler. “SkePU: a multi-backend skeleton programming library for multi-GPU systems”. In: *Proceedings of the fourth international workshop on High-level parallel programming and applications*. ACM. 2010, pp. 5–14.

[19] Steffen Ernsting and Herbert Kuchen. “Data parallel algorithmic skeletons with accelerator support”. In: *International Journal of Parallel Programming* 45.2 (2017), pp. 283–299.

[20] August Ernstsson. “SkePU 2: Language Embedding and Compiler Support for Flexible and Type-Safe Skeleton Programming”. LIU-IDA/LITH-EX-A–16/026–SE. MA thesis. Linköping, Sweden: Linköping University, 2016.

[21] Thomas L. Falch and Anne C. Elster. “ImageCL: Language and source-to-source compiler for performance portability, load balancing, and scalability prediction on heterogeneous systems”. In: *Concurrency and Computation: Practice and Experience* (2017).

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

[23] Khronos Group. *OpenCL Overview*. `https://www.khronos.org/opencl/`. Accessed: 2018-05-28.

[24] Matt J. Harvey and Gianni De Fabritiis. “Swan: A tool for porting CUDA programs to OpenCL”. In: *Computer Physics Communications* 182.4 (2011), pp. 1093–1099.

[25] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. “MapCG: writing parallel program portable between CPU and GPU”. In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM. 2010, pp. 217–226.

[26] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. “CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application”. In: *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE. 2013, pp. 136–143.

[27] John R. Humphrey, Daniel K. Price, Kyle E. Spagnoli, Aaron L. Paolini, and Eric J. Kelmelis. “CULA: hybrid GPU accelerated linear algebra routines”. In: *SPIE Defense and Security Symposium (DSS)*. Vol. 7705. 2010.

[28] Intel. *Thread Building Blocks*. `https://www.threadingbuildingblocks.org/intel-tbb-tutorial`. Accessed: 2018-05-28.

[29] *Programming Languages – Technical Specification for C++ Extensions for Parallelism*. Standard. International Organization for Standardization, ISO/IEC JTC1/SC22, Dec. 2015.

[30] Herbert Kuchen. “A skeleton library”. In: *European Conference on Parallel Processing*. Springer. 2002, pp. 620–629.

[31] Herbert Kuchen and Murray Cole. “The integration of task and data parallel skeletons”. In: *Parallel Processing Letters* 12.02 (2002), pp. 141–155.

[32] Lu Li, Usman Dastgeer, and Christoph Kessler. “Pruning strategies in adaptive off-line tuning for optimized composition of components on heterogeneous systems”. In: *Parallel Computing* 51 (2016), pp. 37–45.

[33] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. “Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping”. In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2009, pp. 45–55.

[34] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. “Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption”. In: *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. ACM. 2017, pp. 1–6.

[35] Sparsh Mittal and Jeffrey S. Vetter. “A survey of CPU-GPU heterogeneous computing techniques”. In: *ACM Computing Surveys (CSUR)* 47.4 (2015), p. 69.

[36] *MPI: A Message-Passing Interface Standard*. Version 3.1. Message Passing Interface Forum. June 2015, p. 1.

[37] NVIDIA. *CUDA Zone*. `https://developer.nvidia.com/cuda-zone`. Accessed: 2018-05-28.

[38] Tomas Öhberg, August Ernstsson, and Christoph Kessler. *Hybrid CPU-GPU execution support in the skeleton programming framework SkePU*. Accepted for 11th International Symposium on High-Level Parallel Programming and Applications (HLPP 2018), Orléans, France 12-13 July 2018.

[39] *OpenACC Programming and Best Practices Guide*. openacc-standard.org. June 2015, pp. 20–21.

[40] *OpenMP Application Programming Interface*. Version 4.5. OpenMP Architecture Review Board. Nov. 2015, pp. 14–27.

[41] Jie Shen, Ana Lucia Varbanescu, Yutong Lu, Peng Zou, and Henk Sips. “Workload partitioning for accelerating applications on heterogeneous platforms”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.9 (2016), pp. 2766–2780.

[42] Oskar Sjöström. “Parallelizing the Edge application for GPU-based systems using the SkePU skeleton programming library”. LIU-IDA/LITH-EX-A–15/001–SE. MA thesis. Linköping, Sweden: Linköping University, 2015.

[43] Fábio Soldado, Fernando Alexandre, and Hervé Paulino. “Execution of compound multi-kernel OpenCL computations in multi-CPU/multi-GPU environments”. In: *Concurrency and Computation: Practice and Experience* 28.3 (2016), pp. 768–787.

[44] Michel Steuwer and Sergei Gorlatch. “SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems”. In: *International Conference on Parallel Computing Technologies*. Springer. 2013, pp. 258–272.

[45] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. *Top 500 List - November 2017*. `https://www.top500.org/list/2017/11/`. Nov. 2017.

[46] Patricia Sundin. "Adaptation of algorithms for underwater sonar data processing to GPU-based systems". LIU-IDA/LITH-EX-A–13/029–SE. MA thesis. Linköping, Sweden: Linköping University, 2013.

[47] *SYCL Specification*. Version 1.2.1. Khronos OpenCL Working Group — SYCL subgroup. Dec. 2017, pp. 15–16.

[48] *The OpenACC Application Programming Interface*. Version 2.6. OpenACC-Standard.org. Nov. 2017, pp. 7–17.

[49] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. "Towards dense linear algebra for hybrid GPU accelerated manycore systems". In: *Parallel Computing* 36.5 (2010), pp. 232–240.

[50] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *IEEE transactions on parallel and distributed systems* 13.3 (2002), pp. 260–274.

[51] Josep Torrellas, Monica S. Lam, and John L. Hennessy. "False sharing and spatial locality in multiprocessor caches". In: *IEEE Transactions on Computers* 43.6 (1994), pp. 651–663.

[52] Mario Vestias and Horácio Neto. "Trends of CPU, GPU and FPGA for high-performance computing". In: *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. IEEE. 2014, pp. 1–6.

[53] Fabian Wrede and Steffen Ernsting. "Simultaneous CPU–GPU execution of data parallel algorithmic skeletons". In: *International Journal of Parallel Programming* 46.1 (2018), pp. 42–61.

[54] Yongbing Zhang, Hisau Kameda, and Sheung-Lun Hung. "Comparison of dynamic and static load-balancing strategies in heterogeneous distributed systems". In: *IEE Proceedings-Computers and Digital Techniques* 144.2 (1997), pp. 100–106.