Skeleton Programming for Heterogeneous GPU-based Systems

by

Usman Dastgeer
Copyright © 2011 Usman Dastgeer [except Chapter 4]

Copyright notice for Chapter 4: © ACM, 2011. This is a minor revision of the work published in Proceedings of the Fourth International Workshop on Multicore Software Engineering (IWMSE’11), Honolulu, Hawaii, USA, 2011, http://doi.acm.org/10.1145/1984693.1984697. ACM COPYRIGHT NOTICE. Copyright © 2011 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

ISBN 978-91-7393-066-6
ISSN 0280–7971
Printed by LiU Tryck 2011

URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-70234
Skeleton Programming for Heterogeneous GPU-based Systems

by

Usman Dastgeer

October 2011
ISBN 978-91-7393-066-6
Linköping Studies in Science and Technology
Thesis No. 1504
ISBN 0280–7971
LiU–Tek–Lic–2011:43

ABSTRACT

In this thesis, we address issues associated with programming modern heterogeneous systems while focusing on a special kind of heterogeneous systems that include multicore CPUs and one or more GPUs, called GPU-based systems. We leverage the skeleton programming approach to achieve high level abstraction for efficient and portable programming of these GPU-based systems and present our work on SkePU which is a skeleton library for these systems.

We first extend the existing SkePU library with a two-dimensional (2D) data type and accordingly generalized skeleton operations, and implement several new applications that utilize these new features. Furthermore, we consider the algorithmic choice present in SkePU and implement support to specify and automatically optimize the algorithmic choice for a skeleton call, on a given platform.

To show how to achieve high performance, we provide a case-study on an optimized GPU-based skeleton implementation for 2D convolution computations and introduce two metrics to maximize resource utilization on a GPU. By devising a mechanism to automatically calculate these two metrics, performance can be retained while porting an application from one GPU architecture to another.

Another contribution of this thesis is the implementation of runtime support for task parallelism in SkePU. This is achieved by integration with the StarPU runtime system. By this integration, support for dynamic scheduling and load balancing for SkePU skeleton programs is achieved. Furthermore, a capability to do hybrid execution by parallel execution on all available CPUs and GPUs in a system, even for a single skeleton invocation, is developed.

SkePU initially supported only data-parallel skeletons. The first task-parallel skeleton (farm) in SkePU is implemented with support for performance-aware scheduling and hierarchical parallel execution by enabling all data parallel skeletons to be usable as tasks inside the farm construct.

Experimental evaluations are carried out and presented for algorithmic selection, performance portability, dynamic scheduling and hybrid execution aspects of our work.

This work has been supported by EU FP7 project PEPPHER and by SeRC.

Department of Computer and Information Science
Linköping universitet
SE-581 83 Linköping, Sweden
Acknowledgements

First, I would like to express my deep gratitude to my main supervisor Christoph Kessler for always keeping the door open for discussions. Without your kind support and guidance, this thesis would not have been possible. Thanks so much for your endless patience and always being there when I needed.

Special thanks to my co-supervisor Kristian Sandahl for his help and guidance in all matters and for showing trust in me. Thanks also to Johan Enmyren, who, together with Christoph Kessler, started the work on the SkePU skeleton framework that I have based much of my work on.

This work has been financially supported by the EU FP7 project PEPPHER, grant #248481 (www.peppher.eu), and Swedish e-Science Research Center (SeRC). I would very much like to thank all the members of the PEPPHER project, for interesting discussions in the project meetings that I have attended. I have learned a lot from these discussions and many ideas to this research are influenced by our discussions in the project meetings.

Thanks also to all the past and present members of the PELAB and my colleagues at the department of computer and information science, for creating an enjoyable atmosphere. A big thanks to Bodil Mattsson Kihlström, Åsa Kärman and Anne Moe who took care of any problems that I have run into. Thanks to Daniel Cederman and Philippas Tsigas for running some experiments on their CUDA machines.

I would also like to thank my friends and family for their continuous support and encouragement. Especially, I am grateful to my parents and family members for their love and support.
Contents

1 Introduction 1  
  1.1 Motivation .................................................. 1  
  1.2 Skeleton programming ....................................... 3  
  1.3 Problem formulation ......................................... 4  
  1.4 Contributions ................................................ 5  
  1.5 List of publications .......................................... 5  
  1.6 Thesis outline ................................................ 6  

2 Background 7  
  2.1 Programming NVIDIA GPUs ................................. 7  
    2.1.1 CUDA .................................................. 8  
    2.1.2 OpenCL ............................................... 9  

3 SkePU 10  
  3.1 SkePU library ............................................... 10  
    3.1.1 User functions ....................................... 10  
    3.1.2 Containers .......................................... 11  
    3.1.3 Skeletons ........................................... 12  
    3.1.4 Lazy memory copying ................................. 23  
    3.1.5 Multi-GPU support ................................... 23  
    3.1.6 Dependencies ....................................... 24  
  3.2 Application examples ....................................... 24  
    3.2.1 Gaussian blur filter .................................. 24  
    3.2.2 ODE solver .......................................... 25  

4 Auto-tuning SkePU 29  
  4.1 Need for algorithmic selection ............................. 29  
    4.1.1 A motivating example ................................ 29  
  4.2 Execution plan ............................................. 30  
  4.3 Auto-tuning ................................................ 31  
    4.3.1 Prediction framework ................................. 32  
    4.3.2 Prediction accuracy .................................. 36  
  4.4 Evaluation ................................................ 38  
    4.4.1 Tuning ................................................. 38
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.2 Performance portability</td>
<td>41</td>
</tr>
<tr>
<td><strong>5</strong> An optimization case-study</td>
<td>42</td>
</tr>
<tr>
<td>5.1 Background</td>
<td>42</td>
</tr>
<tr>
<td>5.2 GPU optimizations</td>
<td>43</td>
</tr>
<tr>
<td>5.3 Maximizing resource utilization</td>
<td>45</td>
</tr>
<tr>
<td>5.4 Evaluation</td>
<td>46</td>
</tr>
<tr>
<td><strong>6</strong> SkePU StarPU integration</td>
<td>54</td>
</tr>
<tr>
<td>6.1 Need for runtime support</td>
<td>54</td>
</tr>
<tr>
<td>6.2 StarPU</td>
<td>55</td>
</tr>
<tr>
<td>6.2.1 StarPU task-model</td>
<td>56</td>
</tr>
<tr>
<td>6.2.2 Data management</td>
<td>56</td>
</tr>
<tr>
<td>6.2.3 Dynamic scheduling</td>
<td>56</td>
</tr>
<tr>
<td>6.3 Integration details</td>
<td>57</td>
</tr>
<tr>
<td>6.3.1 Asynchronous execution</td>
<td>57</td>
</tr>
<tr>
<td>6.3.2 Abstraction gap</td>
<td>57</td>
</tr>
<tr>
<td>6.3.3 Containers</td>
<td>57</td>
</tr>
<tr>
<td>6.3.4 Mapping SkePU skeletons to StarPU tasks</td>
<td>58</td>
</tr>
<tr>
<td>6.3.5 Data Partitioning</td>
<td>59</td>
</tr>
<tr>
<td>6.3.6 Scheduling support</td>
<td>59</td>
</tr>
<tr>
<td>6.4 Implementation of the Farm skeleton</td>
<td>59</td>
</tr>
<tr>
<td>6.5 Evaluation</td>
<td>62</td>
</tr>
<tr>
<td>6.5.1 Data Partitioning and locality on CPUs</td>
<td>64</td>
</tr>
<tr>
<td>6.5.2 Data-locality aware scheduling</td>
<td>67</td>
</tr>
<tr>
<td>6.5.3 Performance-model based scheduling policies</td>
<td>67</td>
</tr>
<tr>
<td>6.5.4 Static scheduling</td>
<td>68</td>
</tr>
<tr>
<td>6.5.5 Overhead</td>
<td>68</td>
</tr>
<tr>
<td><strong>7</strong> Related Work</td>
<td>71</td>
</tr>
<tr>
<td>7.1 Skeleton programming</td>
<td>71</td>
</tr>
<tr>
<td>7.2 Hybrid execution and dynamic scheduling</td>
<td>73</td>
</tr>
<tr>
<td>7.3 Algorithmic selection and Auto-tuning</td>
<td>74</td>
</tr>
<tr>
<td><strong>8</strong> Discussion and Future Work</td>
<td>76</td>
</tr>
<tr>
<td>8.1 SkePU extensions</td>
<td>76</td>
</tr>
<tr>
<td>8.2 Algorithmic selection and PEPPHER</td>
<td>78</td>
</tr>
<tr>
<td><strong>9</strong> Conclusions</td>
<td>80</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

For more than thirty years, chip manufacturers kept the Moore’s law going by constantly increasing the number of transistors that could be squeezed onto a single microprocessor chip. However in the last decade, the semiconductor industry switched from making microprocessors run faster to putting more of them on a chip [83]. This switch is mainly caused by physical constraints that strongly limit further increase in clock frequency of a single microprocessor. Since 2003, we have seen dramatic changes in the mainstream computing as the CPUs have gone from serial to parallel (called multicore CPUs) and a new generation of powerful specialized co-processors called accelerators has emerged.

The transition from serial to parallel execution on CPUs means that the sequential programming interface cannot be used to efficiently program these parallel architectures [83, 47, 12, 13, 60]. Unlike the sequential programming paradigm, there exists no single unified parallel programming model to program these new multicore architectures. The problem with programming these architectures will inflate even further with introduction of hundreds of cores in consumer level machines, by the end of this decade [28].

Beside multicore CPUs, we have seen a tremendous increase in the usage of a special kind of accelerator called Graphics Processing Units (GPUs) for General Purpose computing (GPGPU) over the last five years. This mainly started with the introduction of Compute Unified Device Architecture (CUDA) by NVIDIA in 2006 [67]; since then numerous applications are ported to GPUs with speed-ups shown up to two order of magnitude over the multicore CPUs execution. The massive floating point performance of modern GPUs along with relatively low power consumption turn them into a compelling platform for many compute-intensive applications. Also, the modern GPUs are becoming increasingly programmable especially with the introduction of L1 cache in the new NVIDIA Fermi GPUs.
The combination of multicore CPUs and accelerators is called a heterogeneous architecture. These architectures have promising prospects for future mainstream computing. One popular example of a heterogeneous architecture which we have focused in our work, is a system consisting of multicore CPUs and one or more programmable GPUs, also called a GPU-based system. Currently, there exist more than 250 million GPU-based systems [49], world-wide. The CPUs are good in low-latency computations and control instructions while GPUs have massive computational power, suited for high-throughput computing. The combination of two offers opportunities for power-efficient computing by exploiting the suitability of a computation to the type of processing device.

Although providing power and cost-efficient computing, these GPU-based systems expose a lot of problems on the software side. These problems can be discussed with respect to three software-related properties:

- **Programmability**: There exists no unified programming model to program such a GPU-based system. The OpenCL standard is an effort in this regard but it is quite low level and the standard is still evolving. Furthermore, many existing parallel programming models do not provide an adequate level of abstraction and they tend to expose concurrency issues to the programmer.

- **Portability**: The choice of programming model can restrict the portability in these systems as programming models are often associated with a particular type of device (e.g. OpenMP for CPUs). Even with a unified programming model such as OpenCL, portability is restricted by device-specific optimizations.

- **Performance**: Getting performance from these systems is a daunting task. The performance currently depends on device-specific optimizations, often hard-coded in the code which limits portability [63]. Moreover, the abundance and quick evolution of these systems can quickly vanish the optimization effort while porting application from a system with one architecture to another system with different architecture or even to the next generation of the same system.

To make matters worse, apparently, there exist tradeoffs between these properties: Optimizing performance often requires coding device-specific optimizations which are very low-level and can restrict the portability to other systems, possibly with different architectural features. A high level of abstraction may yield better programmability support at the expense of performance. Skeleton programming can help to manage these apparent tradeoffs, at least to a certain extent.
1.2 Skeleton programming

A typical structured parallel program consists of both a computation and a coordination part [52]. The computation part expresses the calculation describing application logic, control and data flow in a procedural manner. The coordination part consists of managing concurrency issues such as thread management, deadlocks, race-conditions, synchronization, load balancing and communication between the concurrent threads.

Skeletons, introduced by Cole [31], model computation and coordination patterns that occur frequently in the structured parallel applications and provide abstraction with a generic sequential high-level interface. As a predefined component derived from higher-order functions, a skeleton can be parametrized with user computations. Skeletons are composable in a way that more complex parallel patterns can be modeled by composing existing possibly simpler skeletons. Thus, a program using skeletons can be built by decomposing its functionality into common computational patterns which can be modeled with available skeletons. Writing applications with skeletons is advantageous as parallelism and synchronization, as well as leveraging the target architectural features comes almost for free for skeleton-expressed computations. However, computations that do not fit any predefined skeleton or their combination still have to be written manually. Skeletons can be broadly categorized into data and task-parallel skeletons:

- **Data parallel skeletons**: The parallelism in these skeletons comes from (possibly large amount of) data, e.g., by applying a certain function $f$ independently on each element in a large data structure. The behavior of data parallel skeletons establishes functional correspondences between data and is usually considered as a type of fine-grained parallelism [52].

- **Task parallel skeletons**: The parallelism in task-parallel skeletons comes from exploiting independence between different tasks. The behavior of task parallel skeletons is mainly determined by the interaction between tasks. The granularity of tasks, either fine or coarse, determines the granularity of parallelism.

By arbitrary nesting both task and data parallel skeletons, structured hierarchical parallelism can be built inside a skeleton application. This is often referred to as **mixed-mode parallelism**. In the following, we describe how skeleton programming, in general, addresses the programmability, portability and the performance problem:

- **Programmability**: Skeletons provide a sequential interface to the outside world as parallelism is implicitly modeled based on the algorithmic structure that a skeleton implements. So, an application programmer can use a skeleton just like a sequential component. This allows building a parallel application using skeletons analogously to
the way in which one constructs a structured sequential program. This also means that the low-level concurrency issues such as communication, synchronization, and the load-balancing are taken care of by a skeleton implementation. This frees an application programmer from managing parallelism issues and helps him/her focus on implementing the actual logic of the application.

- **Portability:** A skeleton interface models the description of the algorithmic structure in a platform-independent manner which can subsequently be realized by multiple implementations of that skeleton interface, for one or more platforms. This allows an application written using skeletons to be portable across different platforms for which the skeleton implementation(s) exist. This decoupling between a skeleton description, exposed via a skeleton interface to the application programmer and actual implementations of that skeleton is a key tenet of many modern skeleton programming frameworks [52].

- **Performance:** Although having a generic interface, a skeleton implementation can be optimized internally by exploiting knowledge about communication, synchronization and parallelism that is inherent in the skeleton definition. A skeleton invocation can easily be expanded or bound to an equivalent expert-written efficient implementation for a platform that encapsulates all low-level platform-specific details such as managing parallelism, load balancing, communication, utilization of vector instructions, memory coalescing etc.

### 1.3 Problem formulation

In this thesis, we consider the skeleton programming approach to address the portability, performance and programmability issues for the GPU-based systems. We consider the following questions:

- How can skeleton programming be used to program GPU-based systems while achieving performance comparable to hand written code?

- A skeleton can have multiple implementations, even for a single platform. On a given platform, what can be done to make a selection between different skeleton implementations for a particular skeleton call?

- How can performance be retained (at least to a certain extent) while porting an application from one architecture to another?

- How does dynamic scheduling compare with the static scheduling for a skeleton program execution on GPU-based systems?

- Can we simultaneously use different computing devices (CPUs, GPUs) present in the system, even for a single skeleton call?
1.4 Contributions

Most important contributions of the work presented in this thesis are:

1. The existing skeleton programming framework for GPU-based systems (SkePU) is extended to support two-dimensional data type and skeleton operations. Various applications are implemented afterwards based on this support for two-dimensional skeleton operations (Chapter 3).

2. The concept of an execution plan is introduced to support algorithmic selection between multiple skeleton implementations. Furthermore, an auto-tuning framework using an off-line, machine learning approach is proposed for automatic generation of these execution plans for a target platform (Chapter 4).

3. A case-study is presented for optimizing 2D convolution operations on GPUs using skeleton programming. We introduce two metrics for resource maximization on GPUs and show how to calculate them automatically. We evaluate their performance implications and show how we can use these metrics to attain performance portability between different generations of GPUs (Chapter 5).

4. Dynamic scheduling support is implemented for the SkePU skeleton library. Impact of dynamic and static scheduling strategies is evaluated with different benchmark applications. Furthermore, the first task-parallel skeleton (farm) for the SkePU library is implemented with dynamic load-balancing and performance aware scheduling support. The farm implementation supports data-parallel skeletons as tasks, enabling hierarchical mixed-mode parallel executions (Chapter 6).

5. Support for simultaneous use of multiple kinds of resources for a single skeleton call (known as hybrid execution) is implemented for SkePU skeletons. Experiments show significant speedups by hybrid execution over traditional CPU- or GPU-based execution (Chapter 6).

1.5 List of publications

The main body of this thesis is based on the following publications:


2. Usman Dastgeer, Johan Enmyren, and Christoph W. Kessler. Auto-tuning SkePU: A Multi-Backend Skeleton Programming Framework


The mapping between the preceding publication list and thesis chapters is as follows: Publication 1 and 2 maps to text in Chapter 4; Publication 3 and 4 maps to text in Chapter 6 and Chapter 5 respectively. Publication 5 maps to most of the text in Chapter 3.

### 1.6 Thesis outline

The rest of the thesis is organized as follows:

- Chapter 2 introduces technical background concepts that are important for understanding the remainder of the thesis.
- Chapter 3 presents the SkePU skeleton programming framework for GPU-based systems.
- Chapter 4 presents our work on auto-tuning the algorithmic choice between different skeleton implementations in SkePU.
- Chapter 5 contains a case-study that shows usage of parametric machine models to achieve limited performance portability for 2D convolution operations.
- Chapter 6 describes our work on achieving dynamic scheduling and hybrid execution support for SkePU. It also explains our implementation of the *farm* skeleton with dynamic scheduling support.
- Chapter 7 discusses related work.
- Chapter 8 lists some future work.
- Chapter 9 concludes the thesis.
Chapter 2

Background

This chapter contains a brief description of CUDA and OpenCL programming with NVIDIA GPUs.

2.1 Programming NVIDIA GPUs

Traditionally, GPUs were designed and used for graphics and image processing applications only. This is because of their highly specialized hardware pipeline which suited graphics and similar applications, made it difficult to use them for general-purpose computing. However, with the introduction of programmable shaders and high-level programming models such as CUDA, more and more applications are being implemented with GPUs [67]. One of the big differences between a traditional CPU and a GPU is the difference between how they use the chip-area. A CPU, as a multi-tasking general-purpose processing device, uses lot of its chip area for other circuitry than arithmetic computations, such as caching, speculation and flow control. This helps it in performing a variety of different tasks at a high speed and also in reducing latency of sequential computations. The GPU, on the other hand, devotes much more space on the chip for pure floating-point calculations since it focuses on achieving high throughput by doing massively parallel computations. This makes the GPU very powerful on certain kinds of problems, especially those that have a data-parallel nature, preferably with much more computation than memory transfers [6].

In GPU computing, performance comes from creating a large number of GPU threads, possibly one thread for computing a single value. GPU threads are quite light-weight entities with zero context-switching overhead. This is quite different to CPU threads which are more coarse-grained entities and are usually quite few in numbers.
2.1.1 CUDA

In 2006, NVIDIA released the first version of CUDA [67], a general purpose parallel computing architecture based on ANSI C, whose purpose was to simplify the application development for NVIDIA GPUs. CUDA versions 3.0 or higher support a big subset of C++ including templates, classes and inheritance which makes CUDA programming relatively easier in comparison to OpenCL which is a low level alternative to program heterogeneous architectures.

In CUDA, the program consists of host and device code, potentially mixed in a single file that can be compiled by the NVIDIA compiler nvcc, which internally uses a conventional C/C++ compiler like GCC for compiling the host code. A CUDA program execution starts on a CPU (host thread); afterwards the host thread can invoke the device kernel code while managing movement of application data between host and device memory.

Threads in a CUDA kernel are organized in a 2-level hierarchy. At the top level, a kernel consists of a 1D/2D grid of thread-blocks where each thread block internally contains multiple threads organized in either 1, 2 or 3 dimensions [67]. The maximum number of threads inside a single thread block ranges from 512 to 1024 depending upon the compute capability of a GPU. One or more thread blocks can be executed by a single compute unit called SM (Streaming Multiprocessor). The SMs do all the thread management and are able to switch threads with no scheduling overhead. Furthermore, threads inside a thread block can synchronize as they execute inside the same SM. The multiprocessor executes threads in groups of 32, called warps, but each thread executes with its own instruction address and register state, which allows for separate branching. It is, however, most efficient if all threads in one warp take the same execution path, otherwise the execution in the warp is sequentialized [6]. To measure effective utilization of computational resources of a SM, NVIDIA defined the warp occupancy metric. The warp occupancy is the ratio of active warps per SM to the maximum number of active warps supported for a SM on a GPU.

A CUDA program can use different types of memory. Global device memory is large but high latency memory that is normally used for copying input and output data to and from the main memory. Multiple accesses to this global memory from different threads in a thread block can be coalesced into a single larger memory access. However, the requirements for coalescing differ between different GPU architectures [6]. Besides global memory, each SM has an on-chip read/write shared memory whose size ranges from 16KB to 64KB between different generation of GPUs. It can be allocated at thread block level and can be accessed by multiple threads in a thread block, in parallel unless there is a bank conflict [6]. In the Fermi architecture, a part of the shared memory is used as L1 cache (configurable, either 16KB/48KB or 48KB/16KB L1/shared-memory). Constant memory is a small read-only hardware-managed cache, supporting low latency, high speed access when all threads in a thread block access the same memory location. Moreover, each
SM has 8,192 to 32,768 32-bit general purpose registers depending upon the GPU architecture [6]. The register and shared memory usage by a CUDA kernel can be analyzed by compiling CUDA code using the `nvcc` compiler with the `--ptxas-options -v` option.

### 2.1.2 OpenCL

OpenCL (Open Computing Language) is an open low-level standard by the Khronos group [82] that offers a unified computing platform for modern heterogeneous systems. Vendors such as NVIDIA, AMD, Apple and Intel are members of the Khronos group and have released OpenCL implementations, mainly targeting their own compute architectures.

The OpenCL implementation by NVIDIA runs on all NVIDIA GPUs that support the CUDA architecture. Conceptually, the OpenCL programming style is very similar to CUDA when programming on NVIDIA GPUs as most differences only exist in naming of different concepts [68]. Using OpenCL, developers write compute kernels using a C-like programming language. However, unlike CUDA, the OpenCL code is compiled dynamically by calling the OpenCL API functions. At the first invocation, the OpenCL code is automatically uploaded to the OpenCL device memory. In principle, the code written in OpenCL should be portable (executable) on all OpenCL platforms (e.g. x86 CPUs, AMD and NVIDIA GPUs). However, in reality, certain modifications in the program code may be required while switching between different OpenCL implementations [41]. Furthermore, device-specific optimizations applied to an OpenCL code may negatively impact performance when porting the code to a different kind of OpenCL device [63, 41].
Chapter 3

SkePU

In this chapter, we introduce SkePU - a skeleton programming framework for multicore CPU and multi-GPU systems which provides six data-parallel and one task-parallel skeletons, two container types, and support for execution on multi-GPU systems both with CUDA and OpenCL.

The first version of the SkePU library was designed and developed by En-myren and Kessler [44], with support for one-dimensional data-parallel skeletons only. Since then, we have extended the implementation in many ways including support for a two-dimensional data-type and operations. Here, we present a unified view of the SkePU library based on its current development status.

In Section 3.1, we describe the SkePU library while in Section 3.2, we evaluate SkePU with two benchmark applications.

3.1 SkePU library

SkePU is a C++ template library that provides a simple and unified interface for specifying data- and task-parallel computations with the help of skeletons on GPUs using CUDA and OpenCL. The interface is also general enough to support other architectures, and SkePU implements both a sequential CPU and a parallel OpenMP backend.

3.1.1 User functions

In order to provide a simple way of defining functions that can be used with the skeletons regardless of the target architecture, SkePU provides a macro language where preprocessor macros expand, depending on the target selection, to the right kind of structure that constitutes the function. The SkePU user functions generated from a macro based specification are basically a struct with member functions for CUDA and CPU, and strings
3.1. SkePU library

BINARY_FUNC(plus_f, double, a, b, return a+b;
)  
// EXPANDS TO: =====>
struct plus_f
{
skepu::FuncType funcType;
std::string func_CL;
std::string funcName_CL;
std::string datatype_CL;
plus_f()
{
  funcType = skepu::BINARY;
  funcName_CL.append("plus_f");
  datatype_CL.append("double");
  func_CL.append("double plus_f(double a, double b)\
  {\
    return a+b;
  }\n");
}
double CPU(double a, double b)
{
  return a+b;
}
__device__ double CU(double a, double b)
{
  return a+b;
};

Figure 3.1: User function, macro expansion.

for OpenCL. Figure 3.1 shows one of the macros and its expansion, and Listing 3.1 lists all macros available in the current version of SkePU.

3.1.2 Containers

To support skeleton operations, SkePU includes an implementation for the Vector and Matrix containers. The containers are defined in the skepu namespace.

1D Vector data type

The Vector container represents a vector/array type, designed after the STL container vector. Its implementation uses the STL vector internally, and its interface is mostly compatible with the STL vector. For instance,
skepu::Vector<double> input(100,10);
creates a vector of size 100 with all elements initialized to 10.

2D Matrix data type

The Matrix container represents a 2D array type and internally uses contiguous memory to store its data in a row-major order. Its interface and
behavior is similar to the SkePU Vector but with some additions and variations. It provides methods to access elements by row and column index. Furthermore, it provides an iterator for row-wise access, while for column-wise access, it uses matrix transpose to provide read only access. A 50x50 matrix with all elements initialized to value 10 can be created as follows:

```cpp
skepu::Matrix<double> input(50,50,10);
```

It also provides operations to resize a matrix and split the matrix into sub-matrices.

### 3.1.3 Skeletons

SkePU provides Map, Reduce, MapReduce, MapOverlap, MapArray and Scan skeletons with sequential CPU, OpenMP, CUDA and OpenCL implementations. The task-parallel skeleton (Farm) is currently implemented with the support of the StarPU runtime system (see Chapter 6). A program using SkePU needs to include the SkePU header file(s) for skeleton(s) and container(s) used in the program that are defined under the namespace `skepu`.

In the object-oriented spirit of C++, the skeleton functions in SkePU are represented by objects. By overloading `operator()` they can be made to behave in a way similar to regular functions. All of the skeletons contain member functions representing each of the different implementations, CUDA, OpenCL, OpenMP and CPU. The member functions are called CU, CL, OMP and CPU respectively. If the skeleton is called with `operator()`, the library decides which one to use depending on the execution plan used (see Section 4.2). In the OpenCL case, the skeleton objects also contain the necessary code generation and compilation procedures. When a skeleton is instantiated, it creates an environment to execute in, containing all available

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNARY_FUNC</td>
<td>Unary function with multiple parameters and a function to be applied.</td>
</tr>
<tr>
<td>UNARY_FUNC_CONSTANT</td>
<td>Unary function with multiple parameters and a constant value.</td>
</tr>
<tr>
<td>BINARY_FUNC</td>
<td>Binary function with two parameters and a function to be applied.</td>
</tr>
<tr>
<td>BINARY_FUNC_CONSTANT</td>
<td>Binary function with two parameters and a constant value.</td>
</tr>
<tr>
<td>TERNARY_FUNC</td>
<td>Ternary function with three parameters and a function to be applied.</td>
</tr>
<tr>
<td>TERNARY_FUNC_CONSTANT</td>
<td>Ternary function with three parameters and a constant value.</td>
</tr>
<tr>
<td>OVERLAP_FUNC</td>
<td>Overlap function with multiple parameters and a function to be applied.</td>
</tr>
<tr>
<td>OVERLAP_FUNC_STR</td>
<td>Overlap function with multiple parameters and a stride for access.</td>
</tr>
<tr>
<td>OVERLAP_DEF_FUNC</td>
<td>Overlap function with multiple parameters and default access (not implemented).</td>
</tr>
<tr>
<td>ARRAY_FUNC</td>
<td>Array function with multiple parameters and a function to be applied.</td>
</tr>
<tr>
<td>ARRAY_FUNC_MATR</td>
<td>Array function with multiple parameters and a matrix function to be applied.</td>
</tr>
<tr>
<td>ARRAY_FUNC_MATR_CONST</td>
<td>Array function with multiple parameters and a constant matrix function to be applied.</td>
</tr>
</tbody>
</table>

Listing 3.1: Available macros.
OpenCL or CUDA devices in the system. This environment is created as a singleton so that it is shared among all skeletons in the program.

The skeletons can be called with whole containers as arguments, doing the operation on all elements of the container. Another way to call them is with *iterators*, where a start iterator and an end iterator are provided instead, which makes it possible to only apply the skeleton on parts of a container.

As an example, the following code excerpt

```cpp
skepu::Reduce<plus_f> globalSum(new plus_f);
```

shows how a skeleton instance called `globalSum` is created by instantiating the Reduce skeleton with the user function `plus_f` (as described in Listing 3.3) as a parameter. In the current version of SkePU it needs to be provided both as a template parameter and as a pointer to an instantiated version of the user function (remember that the user functions are in fact structs). Below is a short description of each of the skeletons.

```cpp
/*
 * #include <iostream>
 * #include "skepu/matrix.h"
 * #include "skepu/map.h"
 * UNARY_FUNC(square_f, int, a,
 * return a*a;
 * )
 * int main()
 * {
 *   skepu::Map<square_f> square(new square_f);
 *   skepu::Matrix<int> m(5, 5, 3);
 *   skepu::Matrix<int> r(5, 5);
 *   square(m,r);
 *   std::cout<<"Result: " << r <<"\n";
 *   return 0;
 * }
 * // Output
 * // Result:
 * 9 9 9 9 9
 * 9 9 9 9 9
 * 9 9 9 9 9
 * 9 9 9 9 9
 * 9 9 9 9 9
 */
```

Listing 3.2: A Map example.
Map

Map is a well-known data-parallel skeleton, defined as follows:

- **For vector operands**, every element in the result vector $r$ is a function $f$ of the corresponding elements in one or more input vectors $v_1 \ldots v_k$. The vectors have length $N$. A more formal way to describe this operation is:

$$r[i] = f(v_1[i], \ldots, v_k[i]) \forall i \in \{1, \ldots, N\}$$

- **For matrix operands**, every element in the result matrix $r$ is a function $f$ of the corresponding elements in one or more input matrices $m_1 \ldots m_k$. For matrix operands of size $R \times C$, where $R$ and $C$ are the number of rows and the number of columns respectively, Map is formally defined as:

$$r[i,j] = f(m_1[i,j], \ldots, m_k[i,j]) \forall i \in \{1, \ldots, R\}, j \in \{1, \ldots, C\}.$$ 

In SkePU, the number of input operands $k$ is limited to a maximum of three ($k \leq 3$). An example of Map, which calculates a result matrix as the element-wise square of one input matrix, is shown in Listing 3.2. The output is shown as a comment at the end. A Map skeleton with the name `square` and the user function `square_f` is instantiated and is then applied to input matrix $m$ with result in matrix $r$.

```cpp
#include <iostream>
#include "skepu/matrix.h"
#include "skepu/reduce.h"

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

t main()
{
    skepu::Reduce<plus_f> globalSum(new plus_f);
    skepu::Matrix<float> m(25, 40, (float)3.5);
    float r = globalSum(m);
    std::cout <<"Result: \\n";
    return 0;
}
// Output
// Result: 3500
```

Listing 3.3: An example of a reduction with + as operator.
Reduce

Reduction is another common data-parallel skeleton:

- **For a vector operand**, a scalar result $r$ is computed by applying a commutative associative binary operator $\oplus$ between each element in the vector $v$. Formally:
  \[
  r = v[1] \oplus v[2] \oplus \ldots \oplus v[N].
  \]

- **For a matrix operand**, the reduction is currently implemented for computing a scalar result $r$ by applying a commutative associative binary operator $\oplus$ between each element in the matrix $m$. Formally:
  \[
  r = m[1,1] \oplus m[1,2] \oplus \ldots \oplus m[R,C-1] \oplus m[R,C].
  \]

The future work includes implementation of reduction for a $R \times C$ matrix where an output vector of size $R$ and $C$ is produced instead of a scalar value for row-wise and column-wise reduction respectively.

```c++
#include <iostream>
#include "skepu/vector.h"
#include "skepu/mapreduce.h"

BINARY_FUNC(plus_f, double, a, b, 
    return a+b;
)

BINARY_FUNC(mult_f, double, a, b, 
    return a*b;
)

int main()
{
    skepu::MapReduce<mult_f, plus_f> dotProduct(new mult_f,
    new plus_f);
    skepu::Vector<double> v1(500,4);
    skepu::Vector<double> v2(500,2);
    double r = dotProduct(v1,v2);
    std::cout<<"Result: "$<<r<<"\n";
    return 0;
}

// Output
// Result: 3000
```

Listing 3.4: A MapReduce example that computes the dot product.
Listing 3.3 shows the global sum computation of an input matrix using the Reduce skeleton where reduction is applied using + as operator. The syntax of skeleton instantiation is the same as before but note that when calling the Reduce skeleton in the line float r = globalSum(m) the scalar result is returned by the function rather than returned in a parameter.

MapReduce

MapReduce is basically just a combination of the two above: It produces the same result as if one would first Map one or more operands to a result operand, then do a reduction on that result. The operands can be either vector \((v_1 \ldots v_k)\) or matrix \((m_1 \ldots m_k)\) objects, where \(k \leq 3\) as described above. Formally:

For vectors:

\[
r = f(v_1[1], \ldots, v_k[1]) \oplus \ldots \oplus f(v_1[N], \ldots, v_k[N])
\]

For matrices:

\[
r = f(m_1[1, 1], \ldots, m_k[1, 1]) \oplus \ldots \oplus f(m_1[R, C], \ldots, m_k[R, C])
\]

The \(r\) is output, a scalar value in this case. MapReduce is provided since it combines the mapping and reduction in the same computation kernel and therefore speeds up the calculation by avoiding some synchronization that is needed in case of applying Map and Reduce separately.

The MapReduce skeleton is instantiated in a way similar to the Map and Reduce skeletons, but it takes two user functions as parameters, one for mapping and one for reduction. Listing 3.4 shows computation of the dot product using the MapReduce skeleton for vector operands. A MapReduce skeleton instance with the name \texttt{dotProduct} is created which maps two input vectors with \texttt{mult_f} and then reduces the result with \texttt{plus_f}, producing a scalar value which is the dot product of the two input vectors.

MapOverlap

The higher order function MapOverlap is a variation of the Map skeleton:

- \textbf{For vector operands}, each element \(r[i]\) of the result vector \(r\) is a function of several adjacent elements of one input vector \(v\) that reside within a certain constant maximum distance \(d\) from \(i\) in the input vector. The number of these elements is controlled by the parameter \texttt{overlap}(\(d\)). Formally:

\[
r[i] = f(v[i-d], v[i-d+1], \ldots, v[i+d]) \ orall i \in \{1, \ldots, N\}.
\]

The edge policy, how MapOverlap behaves when a read outside the array bounds is performed, can be either cyclic or constant. When cyclic, the value is taken from the other side of the array within distance \(d\), and when constant, a user-defined constant is used. When nothing is specified, the default behavior is constant with 0 as value.
3.1. SkePU library

- For matrix operands, MapOverlap (a.k.a. 2D MapOverlap) can be used to apply two-dimensional filters. To understand the 2D MapOverlap implementation, we first need to know about two-dimensional filters and their types.

Two-dimensional filters: In image processing [42, 51], a two-dimensional filter is specified as a matrix (also known as two-dimensional filter matrix) and can be either separable or non-separable. A two-dimensional filter matrix $F$ is called separable if it can be expressed as the outer product of two vectors, i.e. one row and one column vector ($H$ and $V$ respectively), as follows:

$$ F = H \times V $$

The separability of a two-dimensional filter matrix can be determined by calculating the rank of the filter matrix, as a separable matrix should have a rank equal to 1. If not separable (i.e. the rank of the filter matrix is not equal to 1), the filter is called non-separable and is applied for each element in the filter matrix.

Determining separability of a filter matrix can be important as a separable matrix may require much less computations (i.e. multiply and add operations) to perform while yielding the same result. With a filter matrix $F$ of size $R \times C$, the computational advantage of applying separable filter versus non-separable filter is:
For instance, for a $15 \times 15$ filter, a separable filter can result in 7.5 times less computations than a non-separable filter. For a detailed description of separable filters and how to calculate the two outer product vectors, we refer to [42, 51].

To support implementation of both separable and non-separable filters, we have designed two variations of the 2D MapOverlap skeleton.

2D MapOverlap with separable overlap: It can be used to apply two-dimensional separable filters by dividing the operation into two one-dimensional MapOverlap operations, i.e. row-wise and column-wise overlap. In row-wise overlap, each element $r[i, j]$ of the result matrix $r$ is a function of several row adjacent elements (i.e. in the same row) of one input matrix $m$ that reside at a certain constant maximum distance from $j$ in the input matrix. The number of these elements is controlled by the parameter $\text{overlap}(d)$. Formally:

$$r[i, j] = f(m[i, j - d], m[i, j - d + 1], \ldots, m[i, j + d]) \forall i \in \{1, \ldots, R\}, j \in \{1, \ldots, C\}.$$ 

In column-wise overlap, each element $r[i, j]$ of the result matrix $r$ is a function of several column adjacent elements (i.e. in the same column) of one input matrix $m$ that reside at a certain constant maximum distance from $i$ in the input matrix. The number of these elements is controlled by the parameter $\text{overlap}(d)$. Formally:

$$r[i, j] = f(m[i - d, j], m[i - d + 1, j], \ldots, m[i + d, j]) \forall i \in \{1, \ldots, R\}, j \in \{1, \ldots, C\}.$$ 

There exists several application of this type of overlap, including Sobel kernel and two-dimensional Gaussian filter [51].

The edge policy can be cyclic or constant. In case of cyclic edge policy, for a row-wise (or column-wise) overlap, when a read outside the row (or column) bounds is performed, the value is taken from the other side of that row (or column) within distance $d$. In case of constant edge policy, a user-defined constant is used which is also the default option with 0 as value.

2D MapOverlap with non-separable overlap: It can be used to apply two-dimensional non-separable filters. The non-separable overlap implementation is different in two ways from the separable overlap implementation, as shown in Figure 3.2. First, the non-separable overlap cannot be divided into row-wise or column-wise overlap but rather
it is applied in a single step. A second and more important difference is that non-separable overlap defines the overlap in terms of block neighboring elements, which include diagonal neighboring elements besides row-wise and column-wise neighbors. The overlap is controlled by the parameters \( \text{overlap\_rows}(d_R) \) and \( \text{overlap\_columns}(d_C) \). The overlap can be applied only based on the neighboring elements or by also providing a weight matrix to the neighboring elements. As the overlap logic is defined inside the skeleton implementation, the \text{OVERLAP\_DEF\_FUNC} macro is used which does not require a user function to be passed as a parameter.

The edge policy is defined by the skeleton programmer, in this case, by adding extra border elements in the input matrix \( m \). These border elements can be calculated by e.g. constant and cyclic policy as defined above. For an output matrix of size \( R \times C \) and 2D overlap of size \( d_R \times d_C \), the input matrix \( m \) is of size \( (R + 2d_R) \times (C + 2d_C) \). Hence, each element \( r[i, j] \) of a result matrix \( r \) is calculated as follows:

\[
r[i, j] = f(m[i, j], m[i, j + 1], \ldots, m[i + 2d_R, j + 2d_C]) \quad \forall i \in \{1, \ldots, R\}, j \in \{1, \ldots, C\}.
\]

Listing 3.5: A MapOverlap example.
There exists several application of this type of overlap, including 2D convolution and stencil computations [51].

In the current implementation of SkePU, when using any of the GPU variants of MapOverlap, the maximum overlap that can be used is limited by the shared memory available to the GPU, and also by the maximum number of threads per block. An example program that does a one-dimensional convolution with the help of MapOverlap for vector operands is shown in Listing 3.5. Note that the indexing is relative to the element calculated, $0 \pm \text{overlap}$. A MapOverlap skeleton is instantiated with $\text{over}_f$ as user function and is then called with an input vector $v$ and a result vector $r$. The constant edge policy is specified using the $\text{skepu}::\text{CONSTANT}$ parameter with value $(\text{float})1$.

```cpp
#include <iostream>
#include "skepu/vector.h"
#include "skepu/maparray.h"

ARRAY_FUNC(arr_f, double, a, b,
    int index = (int)b;
    return a[index];
)

int main()
{
    skepu::MapArray<arr_f> reverse(new arr_f);
    skepu::Vector<double> v1(10);
    skepu::Vector<double> v2(10);
    skepu::Vector<double> r;

    // Sets $v_1 = 1\ 2\ 3\ 4...$
    // $v_2 = 9\ 8\ 7\ 6...$
    for (int i = 0; i < 10; ++i)
    {
        v1[i] = i + 1;
        v2[i] = 10 - i - 1;
    }
    reverse(v1, v2, r);

    std::cout << "Result: " << r << "\n";
    return 0;
}
```

Listing 3.6: A MapArray example that reverses a vector
3.1. SkePU library

MapArray

MapArray is yet another variation of the Map skeleton:

- **For two input vectors**, it produces an output vector \( r \) where each element of the result, \( r[i] \), is a function of the corresponding element of one of the input vectors, \( v_2[i] \) and any number of elements from the other input vector \( v_1 \). This means that at each call to the user defined function \( f \), which is done for each element in \( v_2 \), all elements from \( v_1 \) can be accessed. The notation for accessing an element in \( v_1 \) is the same as for arrays in C, \( v_1[i] \) where \( i \) is a number from 0 to \( K - 1 \) where \( K \) is the length of \( v_1 \). Formally:

\[
r[i] = f(v_1, v_2[i]) \forall i \in \{1, \ldots, N\}.
\]

- **For one input vector and one input matrix**, a result matrix \( r \) is produced such that \( r[i,j] \) is a function of the corresponding element of input matrix \( m[i,j] \) and any number of elements from the input vector \( v \). This means that at each call to the user defined function \( f \), which is done for each element in the matrix \( m \), all elements from vector \( v \) can be accessed. Formally:

\[
r[i,j] = f(v, m[i,j]) \forall i \in \{1, \ldots, N\}, j \in \{1, \ldots, M\}.
\]

```cpp
#include <iostream>
#include "skepu/matrix.h"
#include "skepu/scan.h"

BINARY_FUNC(plus_f, int, a, b,
    return a+b;
)

int main()
{
    skepu::Scan<plus_f> prefixSum(new plus_f);
    skepu::Vector<int> v(10, 1);
    skepu::Vector<int> r;
    prefixSum(v, r, skepu::INCLUSIVE);
    std::cout << "Result: " << r << "\n";
    return 0;
}
```

Listing 3.7: A Scan example that computes prefix sum of a vector
Listing 3.6 shows an example of the MapArray skeleton that reverses a vector by using \( v_2[i] \) as index to \( v_1 \). A MapArray skeleton is instantiated and called with \( v_1 \) and \( v_2 \) as inputs and \( r \) as output. \( v_1 \) will be corresponding to parameter \( a \) in the user function \( arr\_f \) and \( v_2 \) to \( b \). Therefore, when the skeleton is applied, each element in \( v_2 \) can be mapped to any number of elements in \( v_1 \). In Listing 3.6, \( v_2 \) contains indexes to \( v_1 \) of the form 9, 8, 7..., therefore, as the user function \( arr\_f \) specifies, the first element in \( r \) will be \( v_1[9] \) the next \( v_1[8] \) etc, resulting in a reverse of \( v_1 \).

**Scan**

Scan (also known as Prefix Sum) is a kernel operation, widely used in many applications such as sorting, list ranking and Gray codes [71]. In Scan skeleton:

- For a given input vector \( v \) with elements \( v[1], v[2], \ldots v[N] \), we compute each of the \( v[1] \oplus v[2] \oplus \cdots \oplus v[k] \) for either \( 1 \leq k \leq N \) (inclusive scan) or \( 0 \leq k < N \) (exclusive scan) where \( \oplus \) is a commutative associative binary operator. For exclusive scan, an initial value needs to be provided.

- For a matrix operand, scan is currently supported row-wise by considering each row in the matrix as a vector scan operation as defined above. A column-wise scan operation is a topic for future work.

Listing 3.3 shows a prefix sum computation using \( + \) as operator on an input vector \( v \). A Scan skeleton with the name \( prefixSum \) is instantiated with a binary user function \( plus\_f \) and is then applied to an input vector \( v \) with result stored in vector \( r \). The scan type is inclusive, specified using the \( \text{skepu::INCLUSIVE} \) parameter.

**Farm skeleton**

Farm is a task-parallel skeleton which allows the concurrent execution of multiple independent tasks, possibly on different workers. It consists of farmer (also called master) and worker threads. The farmer accepts multiple incoming tasks and submits them to different workers available for execution. The overhead of submitting tasks to different workers should be negligible, otherwise the farmer can become the bottleneck. The farmer is also responsible for synchronization (if needed) and for returning the control (and possibly results) back to the caller when all tasks finished their execution. The workers actually execute the assigned task(s) and notify the farmer when a task finishes the execution. A task is an invocation of a piece of functionality with implementations for different types of workers available in the system.\(^2\) Moreover, a task could itself be internally parallel (e.g., a

\(^2\)In our farm implementation, a task could define implementations for a subset of worker types (e.g., a task capable of running only on CPU workers).
3.1. SkePU library

Figure 3.3: A Farm skeleton.

data parallel skeleton) or could be another task-parallel skeleton (e.g., another farm), allowing hierarchical parallel executions. For tasks \( t_1, t_2, \ldots, t_K \), where \( K \) is the total number of tasks, farm could be defined formally as:

\[
\text{farm}(t_1, t_2, \ldots, t_K)
\]

It is also shown in Figure 3.3. The farm implementation for SkePU with a code example is discussed in Chapter 6.

3.1.4 Lazy memory copying

Both SkePU Vector and Matrix containers hide GPU memory management and internally use lazy memory copying to avoid unnecessary memory transfer operations between main memory and device memory. A SkePU container keeps track of which parts of it are currently allocated and uploaded to the GPU. If a computation is done, modifying the elements in a container in the GPU memory, these are not immediately transferred back to the host memory. Instead, the container waits until an element is accessed on the host side before any copying is done (for example through the \( [] \) operator for Vector). This lazy memory copying is of great use if several skeletons are called one after the other, with no modifications of the container data by the host in between. In that case, the payload data of the container is kept on the device (GPU) through all the computations, which significantly improves performance. Most of the memory copying is done implicitly but the containers also contain a flush operation which updates a container from the device and deallocates its memory.

3.1.5 Multi-GPU support

SkePU has support for carrying out computations with the help of several GPUs by dividing the work among them. By default, SkePU will utilize as many GPUs as it can find in the system; however, this can be controlled by defining \texttt{SKEPU\_NUMGPU}. Setting it to 0 makes it use its default behavior i.e. using all available GPUs in the system. Any other number represents the number of GPUs it should use in case the actual number of GPUs present
in the system are equal or more than the number specified. In SkePU, memory transfers between device memories of different GPUs is currently implemented via CPU main memory. With CUDA 4.0, multi-GPUs memory transfers could be done more efficiently with the release of GPU direct 2.0. However, it only works with modern Fermi-based Tesla GPUs.

3.1.6 Dependencies
SkePU is based on C++ and can be compiled with any modern C++ compiler (e.g., GCC). The library does not use any third party libraries except for CUDA and OpenCL. To use either CUDA or OpenCL, their corresponding environments must be present. CUDA programs need to be compiled with Nvidia compiler (NVCC) since CUDA support is provided with the CUDA runtime API. As SkePU is a C++ template library, it can be used by including the appropriate header files, i.e., there is no need to separately compile and link to the library.

3.2 Application examples
In this section, we present two example applications implemented with SkePU. The first example is a Gaussian blur filter that highlights the performance implications of data communication for GPU execution and how lazy memory copying helps in optimizing it. The second application is for a Runge-Kutta ODE solver where we compare an implementation written using SkePU skeletons with respect to other existing implementations and also with respect to a hand-written application.

The following evaluations were performed on a dual-quadcore Intel(R) Xeon (R) E5520 server clocked at 2.27 GHz with 2 NVIDIA GT200 (Tesla C1060) GPUs.

3.2.1 Gaussian blur filter
The Gaussian blur filter is a common operation in computer graphics that convolves an input image with a Gaussian function, producing a new smoother and blurred image. The method calculates the new value of each pixel based on its own and its surrounding pixels’ values.

It can be done either in two dimensions, for each pixel accessing a square halo of neighbor pixels around it, or in one dimension by running two passes over the image, one row-wise and one column-wise. For simplicity, we use here the second approach, which allows to use Vector as container for the image data. When calculating a pixel value, the surrounding pixels are needed but only within a limited neighbourhood. This fits well into the calculation pattern of the MapOverlap skeleton. MapArray (a variant of

\[^{3}\text{The same example is also implemented using the first approach, shown later in Section 6.5.} \]
3.2 Application examples

```c
1 OVERLAP_FUNC(blur_kernel, int, 19, a,
2     return (a[-9] + 18*a[-8] + 153*a[-7] + 816*a[-6] + 3060*a
6     + 18*a[8] + a[9]) >>18;
7 )
```

Listing 3.8: User function used by MapOverlap when blurring an image.

MapOverlap without the restriction to a constant-sized overlap) was also used to restructure the array from row-wise to column-wise data layout. The blurring calculation then becomes: a MapOverlap to blur horizontally, then a MapArray to restructure the image, and another MapOverlap to blur vertically. The image was first loaded into a vector with padding between rows.

Timing was only done on the actual blur computation, not including the loading of images and creation of vectors. For CUDA and OpenCL, the time for transferring the image to the GPU and copying the result back is included. The filtering was done with two passes of a 19-value filter kernel which can be seen in Listing 3.8. For simplicity, only grayscale images of quadratic sizes were used in the benchmark.

The result can be seen in Figure 3.4 where part 3.4a shows the time when applying the filter kernel once to the image, and part 3.4b when applying it nine times in sequence, resulting in heavier blur. We see that, while faster than the CPU variant, CUDA and OpenCL versions are slower than the one using OpenMP on 8 CPU cores for one filtering. This is due to the memory transfer time being much larger than the actual calculation. In Figure 3.4b, however, filtering is done nine times which means more computations and less memory I/O due to the lazy memory copying of the vector. Then the two single GPU variants outperform even the OpenMP version.

Since there is a data dependency in the MapOverlap skeleton when running on multiple-GPUs, we also see that running this configuration loses a lot of performance when applying MapOverlap several times in a row because it needs to transfer data between the GPUs, via the host.

3.2.2 ODE solver

A sequential Runge-Kutta ODE solver was ported to GPU using the SkePU library. The original code used for the porting is part of LibSolve, a library of various Runge-Kutta solvers for ODEs by Korch and Rauber [69]. LibSolve contains several Runge-Kutta implementations, iterated and embedded ones, as well as implementations for parallel machines using shared or distributed memory. The simplest default sequential implementation was
(a) Average time of blurring quadratic greyscale images of different sizes. The Gaussian kernel is applied once to the image.

(b) Average time of blurring quadratic greyscale images of different sizes. The Gaussian kernel is applied nine times to the image.

Figure 3.4: Average time of blurring images of different sizes. Average of 100 runs.
used for the port to SkePU, however other solver variants were used unmodified for comparison.

The LibSolve package contains two ODE test sets. One, called BRUSS2D, is based on the two-dimensional brusselator equation. The other one is called MEDAKZO, the medical Akzo Nobel problem [69]. BRUSS2D consists of two variants depending on the ordering of grid points, BRUSS2D-MIX and BRUSS2D-ROW. For evaluation of SkePU only BRUSS2D-MIX was considered. Four different grid sizes (problem size) were evaluated, 250, 500, 750 and 1000.

The porting was fairly straightforward since the default sequential solver in LibSolve is a conventional Runge-Kutta solver consisting of several loops over arrays sized according to the problem size. These loops were replaced by calls to the Map, Reduce and MapReduce skeletons. The right hand side evaluation function was implemented with the MapArray skeleton.

As mentioned earlier, the benchmarking was done using the BRUSS2D-MIX problem with four different problem sizes (N=250, N=500, N=750 and N=1000). In all tests the integration interval was 0-4 (H=4) and time was measured with LibSolves internal timer functions, which on UNIX systems uses `gettimeofday()`. The different solver variants used in the testing were the following:

- **ls-seq-def**: The default sequential implementation in LibSolve.
- **ls-seq-A**: A slightly optimized variant of ls-seq-def.
- **ls-shm-def**: The default shared memory implementation in LibSolve. It uses pthreads and was run with 8 threads, one for each core of the benchmarking computer.
- **ls-shm-A**: A slightly optimized variant of ls-shm-def, using pthreads and run with 8 threads.
- **skepu-CL**: SkePU port of ls-seq-def using OpenCL as backend and running on one Tesla C1060 GPU.
- **skepu-CL-multi**: SkePU port of ls-seq-def using OpenCL as backend and running on two Tesla C1060 GPUs.
- **skepu-CU**: SkePU port of ls-seq-def using CUDA as backend and running on one Tesla C1060 GPU.
- **skepu-OMP**: SkePU port of ls-seq-def using OpenMP as backend, using 8 threads.
- **skepu-CPU**: SkePU port of ls-seq-def using the default CPU backend.
- **CU-hand**: A “hand”-implemented CUDA variant. It is similar to the SkePU ports but no SkePU code was utilized. Instead, CUBLAS [3] functions were used where applicable, and some hand-made kernels.

The result can be seen in Figure 3.5. The two slowest ones are the sequential variants (ls-seq-def and ls-seq-A), with ls-seq-A of course performing slightly better due to the optimizations. LibSolves shared memory solvers (ls-shm-def and ls-shm-A) show a great performance increase compared to the sequential variants with almost five times faster running time for the largest problem size (N=1000).
We also see that the SkePU CPU solver is comparable to the default LibSolve sequential implementation and the OpenMP variant is similar to the shared memory solvers. The SkePU OpenCL and CUDA ported solvers are however almost 10 times faster than the sequential solvers for the largest problem size. The reason for this is that all the calculations of the core loop in the ODE solver can be run on the GPU, without any memory transfers except once in the beginning and once at the end. This is done implicitly in SkePU since it is using lazy memory copying. However, the SkePU multi-GPU solver does not perform as well; the reason here also lies in the memory copying. Since the evaluation function needs access to more of one vector than what it has stored in GPU memory (in multi-GPU mode, SkePU divides the vectors evenly among the GPUs), some memory transfers are needed: First from one GPU to host, then from host to the other GPU; this slows down the calculations considerably.

Comparing the “hand”-implemented CUDA variant, we see that it is similar in performance to skepu-CU with CU-hand being slightly faster (approximately 10%). This is both due to the extra overhead when using SkePU functions and some implementation differences.

There is also a start-up time for the OpenCL implementations during which they compile and create the skeleton kernels. This time (≈5-10 seconds) is not included in the times presented here since it is considered an initialization which only needs to be done once when the application starts executing.
Chapter 4

Auto-tuning SkePU

In this chapter, we discuss our work on auto-tuning the algorithmic selection for a skeleton call in SkePU. We start by discussing the need for algorithmic selection in Section 4.1, followed by the description of execution plan in Section 4.2 and the auto-tuning and prediction framework in Section 4.3. In Section 4.4, we do the evaluation.

4.1 Need for algorithmic selection

In order to allow skeleton calls to execute on different backends (CPU, GPU), SkePU defines skeleton implementations for different backends (e.g. OpenMP implementation for multicore CPUs and OpenCL for GPUs). In a system containing multiple CPUs and one or more GPUs, multiple implementation variants will become available for a skeleton invocation. Making an optimal or (close-to) optimal choice between these implementation variants for a skeleton invocation is considered a tuning aspect of the SkePU library.

4.1.1 A motivating example

Listing 4.1 shows a vector sum computation using the reduce skeleton. Execution of this computation using different implementations of the reduce skeleton is shown in Figure 4.1. Note that, for technical reasons, it is not possible (without major effort) to mix OpenCL and CUDA code within the same program. As shown in the figure, usage of a single CPU core on this machine yields sub-optimal performance. Furthermore, execution using multiple CPU cores available in the system using OpenMP thread-level parallelism is faster for relatively smaller problem sizes. For larger problem sizes, GPU executions show significant speedups in comparison to execution on CPU cores as the advantage of using GPUs superseded the overhead of data communication. The execution pattern may seem obvious; however,
the transition points when to switch from an implementation for one CPU to an OpenMP parallelization or to code for a single GPU or for multiple GPUs are strongly dependent on the characteristics of the target system and on the computation itself.

This lead to the idea to provide a mechanism for automatic adaptation at run-time to let SkePU select the best implementation variant depending on the actual problem size. The mechanism that is developed and added to SkePU is known as an execution plan.

4.2 Execution plan

An execution plan is a data structure that contains a list of input sizes and adjoining backends which is used to decide which backend to use for a certain input size. In other words, it provides a mapping from an input size to the backend configuration for that input size. A backend configuration includes a certain backend choice and a parameter set for that backend where the parameter set contains tunable parameters for that backend. For now, the parameter set includes the number of threads for the OpenMP backend and grid-size and block-size for GPU backends. For the single-CPU backend,

```cpp
#include <iostream>
#include "skepu/vector.h"
#include "skepu/reduce.h"

BINARY_FUNC(plus_f, double, a, b, return a+b;

int main()
{
  skepu::Reduce<plus_f> globalSum(new plus_f);
  skepu::Vector<double> v0(1000,2);
  // Following call can map to different reduce implementations
  double r = globalSum(v0);
  std::cout<>"Result: " <<r <<"\n";
  return 0;
}

// Output
// Result: 2000
```

Listing 4.1: Vector sum computation using reduction with + as operator.
4.3 Auto-tuning

The tuning of SkePU is based on a prediction framework that enables automatic generation of execution plan(s) with optimal configuration and backend selection.
Figure 4.2: Vector sum with reduce, computed with an empirically determined execution plan on a machine with 2 quad-core Intel(R) Xeon(R) CPU E5520, and two NVIDIA Tesla C1060 GPUs.

4.3.1 Prediction framework

The tuning and prediction framework works as follows:

1. A heuristic algorithm is used to calculate a near-optimal configuration for each backend. The output of this algorithm is the generation of a near-optimal execution plan for a single backend.

2. Estimates for the different components of the execution time on different backends are either calculated off-line using micro-benchmarking or taken from the first step.

3. The optimal configuration information and the off-line calculated estimates for different backends are fed to the prediction framework, which generates an overall optimal execution plan across different backends.

By separating the first and second step in the above list we achieve flexibility in the tuning process. For example, the separation allows for the possibility to tune only for a specific backend by omitting the second step. However, typically the process will be carried out in accordance with the above list where the first and the second step are combined into a big step which helps in avoiding redundancy in the training phase.
4.3. Auto-tuning

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

Figure 4.3: Defining an execution plan and applying it to a skeleton instance, using CPU, OpenMP and OpenCL backends.

**Tuning for optimal configuration of a single backend** A heuristic algorithm is used to calculate an optimized parameter configuration for a certain backend implementation for different problem sizes. For the OpenMP backend, an optimal configuration of the parameter ‘Number of OpenMP threads’ can vary based on different problem sizes and skeleton implementation. Similarly, for GPU backends, different ‘Grid-size’ and ‘Block-size’ combinations may prove optimal depending upon the problem size and the skeleton implementation.

The heuristic algorithm starts with an initial set of candidate configurations of values for the parameter set and iteratively refines through the search space as it proceeds through the training data. The initial parameter set includes multiples of the warp size for the block-size parameter up to the maximum block size allowed by the underlying platform. Similarly, for the grid-size parameter, it includes multiples of some smaller possible value for grid-size (e.g., 512) up to the maximum allowable grid-size. At each iteration, it executes the target kernel (or computationally similar kernel) with different parameter configurations and measures the execution time. The stepwise refinement mainly penalizes those candidates that have never been optimal or even close to optimal for a certain number of previous executions. The heuristic algorithm terminates (exit-criterion) when all items in the training data set have been processed.

The output of the heuristic algorithm is the generation of an execution plan with optimal parameter values for different ranges of problem sizes. It calculates these input ranges by interpolating from the values used in training data. In the following, we show a small part of the execution plan for the OpenCL GPU backend which is automatically generated by the algorithm, where columns denote lower input limit, upper input limit, backend, block-size, and grid-size respectively:

<table>
<thead>
<tr>
<th>lower input limit</th>
<th>upper input limit</th>
<th>backend</th>
<th>block-size</th>
<th>grid-size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>750000</td>
<td>CL_BACKEND</td>
<td>32</td>
<td>32768</td>
</tr>
<tr>
<td>750001</td>
<td>1250000</td>
<td>CL_BACKEND</td>
<td>32</td>
<td>512</td>
</tr>
<tr>
<td>1250001</td>
<td>2250000</td>
<td>CL_BACKEND</td>
<td>128</td>
<td>2048</td>
</tr>
<tr>
<td>2250001</td>
<td>3750000</td>
<td>CL_BACKEND</td>
<td>32</td>
<td>2048</td>
</tr>
<tr>
<td>3750001</td>
<td>4250000</td>
<td>CL_BACKEND</td>
<td>32</td>
<td>16384</td>
</tr>
</tbody>
</table>
To balance the trade-off between accuracy (i.e., optimality of the resulting execution plan for any input range) and overhead (i.e., size of the execution plan representation and time for looking up entries at runtime), we use a threshold parameter \( \theta \) as follows. The entries in the candidate set partition the problem size axis into adjacent intervals. For predicting execution time for any problem size located in such an interval \( i \), we use linear interpolation between the two currently held candidate configurations \( C_i \) and \( C_{i+1} \) defining the interval. Let \( \text{diff}(i+1) \) denote the maximum relative error in execution time prediction for any point in intervals \( i \) and \( i + 1 \) of the problem size axis when using the stored candidate configuration for interval \( i \) and interpolating between \( C_i \) and \( C_{i+2} \), compared to what would be predicted when using the configuration for \( C_{i+1} \) where applicable. If \( \text{diff}(i) \leq \theta \), i.e. the relative prediction error does not exceed the given threshold value (in percent), the intervals \( i \) and \( i + 1 \) can be merged by dropping the candidate configuration \( C_{i+1} \), thus reducing the size of the candidate set by 1. Choosing a threshold value \( \theta \) is usually a tradeoff between the runtime overhead and the precision of the tuning process. Concretely, for our training runs, we obtained the best results for threshold values between 1% and 5%.

Offline calculation of estimates Our auto-tuning mechanism is mainly based on off-line measurements. It starts with exercising a skeleton instance off-line for different sample problem sizes and then records time for different parts of the execution separately. For GPU backends, it keeps track of the following time components:

- **Copy-Up-Time**: time for copying data from host to device.
- **Copy-Down-Time**: time for copying data from device to host.
- **Kernel Execution time**: time for actual execution of a kernel.
- **Overhead time**: the overhead for the skeleton function call.

The overhead time is proved to be insignificant in many cases. For the OpenMP backend, the prediction framework uses:

- **Total time**: overall time for the skeleton function call.
- **Overhead time**: Difference between two successive executions, to account for caching effects.
4.3. Auto-tuning

The time required for estimating these parameters is often negligible. As shown in Table 4.1, communication time (Copy-Up and Copy-Down) time is calculated and tracked (in bytes) for each new platform. We calculate it using micro-benchmarking, but it can also be obtained using the Bandwidth Tests available with CUDA and OpenCL SDK code samples. Kernel execution time depends on the skeleton implementation and on the computational complexity of the user function that the skeleton is parameterized with. However, the OpenMP parameters need to be estimated for each new skeleton implementation in the current implementation.

Table 4.1: Estimates calculation requirements

<table>
<thead>
<tr>
<th>Type</th>
<th>Calculated for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy-Up time</td>
<td>each platform</td>
</tr>
<tr>
<td>Copy-Down time</td>
<td>each platform</td>
</tr>
<tr>
<td>Kernel Execution time</td>
<td>each skeleton implementation</td>
</tr>
<tr>
<td>Total time (OpenMP)</td>
<td>each skeleton implementation</td>
</tr>
<tr>
<td>Overhead time (OpenMP)</td>
<td>each skeleton implementation</td>
</tr>
</tbody>
</table>

Based on the backend and the skeleton being tuned for, we use a formula to divide this time into repetitive time and fixed time. Repetitive time \((R)\) is the time that will be incurred every time a skeleton will be called, even for successive executions with the same data (containers). Fixed time \((F)\) is the time that is incurred once in case of multiple repetitive executions on the same data (containers) for a given (or computationally equivalent) skeleton. For GPU backends, this mainly accounts for the lazy memory copying feature of SkePU which avoids data transfer between device memory and main memory in the case of repetitive executions. For the OpenMP backend, it accounts for loading data into cache which can be reused in later repetitive executions.

Generation of Execution plan. During actual execution of a program, the prediction mechanism uses off-line calculated estimates for different backends available (e.g. OpenCL, OpenCL-Multi, OpenMP) alongside optimal parameter configuration for each backend to provide an optimized schedule considering all available backends and constructing an execution plan. This optimal execution plan guides the overall execution at runtime with minimal overhead. In the following we show an excerpt from an execution plan automatically generated by the prediction framework which shows switching between different backends along with their parameter configurations.

```
1    ---  50000   OMP_BACKEND 8
50001  ---  150000  CL_BACKEND 32  16384
```
Our tuning system considers the number of desired (repetitive) executions (default=0) while choosing the optimal backend for a given problem size. This accounts for the fact that for single (non-repetitive) execution, OpenMP often proves better (mainly because of data transfer overhead to GPUs) while for more repetitive executions, GPUs significantly outperform OpenMP variants (see Section 4.4).

4.3.2 Prediction accuracy

The prediction mechanism uses interpolation and extrapolation to predict for intermediate and larger/smaller problem sizes. We will consider two prediction capabilities to show the accuracy of the prediction framework:

**Prediction for repetitive executions**

Estimates calculated off-line for different problem sizes are categorized into either repetitive time or fixed time. What constitutes the fixed and repetitive part varies for different skeletons and backends and its logic is embedded in the prediction framework. However, once calculated, this execution data can provide accurate prediction for any number of repetitive executions on the target platform using the simple formula

\[ T_r = R \times r + F \]

where \( T_r \) represents the time for \( r \) repetitive executions and \( R \) and \( F \) represent repetitive and fixed time respectively.

As shown in Figure 4.4, predicted execution times closely align with the actual executions for any number of repetitive executions. However, in this case, for repetitive execution with \( r = 2 \) the OpenMP backend was predicted to be optimal by the prediction framework but it shows large fluctuations in actual execution time which can be attributed to the internal cache behavior and/or resource contention as tests were performed on a non-dedicated shared-memory multi-CPU/multi-GPU server (see Section 4.4).

Based on our initial investigation, the variability in OpenMP execution time, even in successive runs, has two reasons:

1. Sharing of resources with other co-running (including background) processes.
2. Differences in mapping of OpenMP threads to available CPU cores by the underlying operating system (e.g. affinity vs custom OpenMP thread scheduling).
4.3. Auto-tuning

Figure 4.4: Vector sum computation - Comparisons of predictions of execution time and actual execution time for multiple repetitive executions using prediction data calculated for single execution. The numbers in titles represent ‘number of repetitions of the skeleton with same data’.

Figure 4.5: Predictions of execution time for a new user function (UF) using the two proposed extrapolation techniques, and comparison to the actual execution time with the new (predicted) user function for a Map computation.
Chapter 4. Auto-tuning SkePU

Prediction for different user-functions

As skeletons are higher order functions, they are parameterized for different functionality by different user functions. For data-parallel skeletons, the complexity of these user functions is normally quite limited, but we considered possibilities to predict for new (often more complex) user functions based on extrapolating from estimates calculated earlier. This is achieved by sampling of kernel execution time of new user functions for different problem sizes and providing this data (pairs of problem size and new execution time) to the tuning framework.\footnote{For this experiment, we have measured the kernel execution time directly, but it could alternatively be estimated using some analytical method such as in [17].} Our tuning framework internally maps these new kernel execution timings to previous executions and tries to extrapolate estimations using the ratios of new values to previous values\footnote{In the presence of similarity between the computational pattern of current and previous user function, less training data can yield more accuracy. However, in other circumstances, it can be easily adjusted to do interpolation based on new estimates only.}.

The extrapolation can use either the overall average ratio or intermediate ratios calculated for the different intervals of problem sizes, as provided in the samples. Accuracy of prediction is largely affected by quantity and quality (e.g., randomly distributed problem sizes) of samples provided for kernel execution time of new user functions. Figure 4.5 shows comparisons of the two proposed ratio-based techniques that can be used with just seven samples provided. As seen in this example, intermediate ratios to interpolation performed better than using the overall average of ratios calculated from all samples.

4.4 Evaluation

To do the evaluation, we consider computations using Map, MapReduce and Reduce skeletons.

All results except performance portability (Figure 4.7) were performed on a dual-quadcore Intel\textsuperscript{\textcopyright} Xeon\textsuperscript{\textcopyright} E5520 server clocked at 2.27 GHz with 2 NVIDIA GT200 (Tesla C1060) GPUs. For the demonstration of performance portability (Figure 4.7), we have used Tesla C2050 and Geforce GTX280 running with CUDA.

4.4.1 Tuning

Tuning considers the possibilities of single as well as repetitive executions and the effect of lazy memory copying on repetitive executions, as shown in Figure 4.6. The tuned configuration proved better in Figure 4.6(a) than other versions for different backends with grid-size and block-size. In Figure 4.6(b) the tuned configuration is an OpenMP configuration, and for successive runs, it showed some fluctuations in the performance, the possible reasons for which are presented earlier in the text.
4.4. Evaluation

Figure 4.6: Dot product calculation using MapReduce, comparison of Tuned version with other suitable configurations of OpenCL, OpenCL multi, and OpenMP for a) 10 repetitive executions where OpenCL and OpenCL-Multi are close-to-optimal due to lazy memory copying, and b) single execution where the OpenMP backend performs better than the GPU counterparts. The numbers in titles represent ‘gridsize blocksize’ and ‘number of OpenMP threads’ for GPU and OpenMP execution respectively.
Chapter 4. Auto-tuning SkePU

Figure 4.7: Element-wise sum with Map, computed on two CUDA architectures (C2050, GTX280) using offline-tuning to automatically construct execution plans. Numbers in titles represent ‘gridsize blocksize’ combinations used for that execution.
4.4.2 Performance portability

To show the performance portability that can be achieved by using the above-mentioned approach, Figure 4.7 demonstrates a map skeleton on two different CUDA platforms that are different from our main target platform. The reason that the auto-tuned version performs much better than others lies in the fact that it could be difficult to figure out optimal choices statically for different problem sizes and in this case hardcoded static intuitive choices were not optimal. Furthermore, inter-/extrapolation may not yield an optimal solution in each case as the search space for the optimal parameter setting can be non-linear [17]. In Figure 4.7(b), for one problem size, the tuned configuration proved non-optimal due to inherent characteristics of the OpenMP shared-memory platform but also due to the non-linear search space for ‘number of threads’.
Chapter 5

An optimization case-study

Although skeletons provide a generic high-level interface, they can have optimized implementations for different architectures and problem instances. In this chapter, we describe our work on providing an optimized GPU (CUDA/OpenCL) implementation for the 2D MapOverlap skeleton with non-separable overlap. We explain it with the help of a 2D convolution application, implemented using that skeleton. We also present two metrics to maximize GPU resource utilization and show how we can achieve performance portability while moving from one GPU (CUDA/OpenCL) architecture to another one by an automatic calculation of these metrics for the new architecture.

```c++
int filter_size = filter_width * filter_height;
for (int i=0;i<out_height;i++){
    for (int j=0;j<out_width;j++){
        float sum = 0;
        for (int k=0; k<filter_height; k++){
            for (int l=0; l<filter_width; l++){
                sum += in[i+k][j+l] * filter_weight[k][l];
            }
        }
        out[i][j] = sum / filter_size;
    }
}
```

Listing 5.1: An excerpt from simple 2D convolution implementation in C++.

5.1 Background

2D convolution is a kernel widely used in image and signal processing applications. It is a kind of MapOverlap operation where a value for every output pixel is computed based on a weighted average of the corresponding input pixel alongside neighboring input pixels. Listing 5.1 shows an excerpt from a C++ implementation of 2D convolution. In the implementation, the
5.2 GPU optimizations

As a data parallel operation, the MapOverlap computation is well-suited for GPU execution, especially for large problem sizes. The naive CUDA implementation can be defined by assigning one thread for computing one output element. Starting from the naive implementation, the CUDA implementation is further optimized in the following ways.

Usage of constant memory

In our framework, the 2D non-separable convolution can be done with or without usage of a weight matrix. In case the weight matrix is used, as in

1In SkePU, OpenCL implementations are mainly used with GPUs.
the 2D convolution application, the weights are multiplied with the corresponding neighboring elements for each output element. As an element in the weight matrix is accessed in read-only mode by all threads in parallel, it is an ideal candidate for placement in fast constant memory [96]. The performance gains from this optimization depend upon the architecture as it could yield up to 100% performance improvement on non-cache GPUs (e.g., NVIDIA GPUs before the Fermi architecture). However, for NVIDIA Fermi GPUs with explicit L1 cache, the performance improvement with usage of constant memory can be much less due to implicit caching and reuse capabilities of these architectures. For instance, for NVIDIA C2050, performance improvement up to 30% is observed over different filter sizes.

Usage of shared memory

In the naive implementation, every thread in a thread-block loads all the neighboring elements from the GPU device memory, which can be very inefficient especially on GPUs with no explicit L1 cache. Instead, threads in a thread block can use shared memory to store their neighborhood and can subsequently access it from there [96]. This optimization can significantly reduce the global memory bandwidth consumption, especially for large filter sizes. For example, for a thread block of 16 × 32 and filter size of 29 × 29, each thread block loads 430592 values without usage of shared memory. With usage of shared memory, the loads are reduced to (16 + 29 − 1) × (32 + 29 − 1) = 2640 values per thread block, a reduction by a factor of 163. Again, the optimization may not yield that much difference in performance while executing on GPUs with L1 cache such as NVIDIA Fermi GPUs.

Adaptive Tiling

Besides the memory optimizations mentioned above, another optimization that can be applied to this class of applications on modern GPUs is known as $1 \times N$ tiling [86]. A tile refers to a block of input data simultaneously processed by multiple threads in a thread block. $1 \times N$ tiling refers to a technique of increasing workload for a thread block by assigning $N$ tiles to a thread block to process instead of 1 tile. This approach reduces the number of thread blocks by a factor of $N$. Besides reducing the overhead associated with thread blocks (e.g. array index calculations, loading constant values), this technique also decreases the amount of overall neighborhood loads as the number of thread blocks is decreased. Despite of its potential advantages, tiling can also result in increased shared memory usage by a thread block as now each thread block processes $N$ elements instead of 1. Similarly, register usage can also increase as extra registers are normally used to save intermediate results. As shown by van Werkhoven et al. [96], using any fixed tiling factor for an image convolution application can result in sub-optimal performance over different filter sizes. Furthermore, with a fixed tiling factor (e.g. 4), the
program may simply not work on certain GPUs due to resource limitations (e.g. shared memory size, number of registers). Rather, the adaptive tiling introduced by van Werkhoven et al. [96] where the tiling factor is chosen based on different filter sizes and resource limitations can be used.

The dynamic selection of the tiling factor is interesting for several reasons. First, there could be several different mechanisms to determine the tiling factor based on different performance characteristics. Furthermore, an automatic way of determining the tiling factor over different machine and problem configurations can help in attaining performance portability.

5.3 Maximizing resource utilization

Modern GPUs have many types of resources that can affect the performance of an executing kernel. These resources can be broadly categorized into computational resources such as arithmetic-logic-units and storage resources such as registers and shared memory. Effective usage of both kind of resources is important for performance but sometimes a tradeoff exists as both cannot be optimized at the same time.

The adaptive tiling is focused on maximizing utilization of storage resources of a multiprocessor such as shared memory and registers. On the other hand, warp occupancy (also known as occupancy) strives for maximizing the computational resource utilization of a multiprocessor\(^2\). In our work, we consider the tradeoff between these two related but different maximization functions, i.e., maximizing computational resource utilization (i.e. maximizing occupancy) or maximizing storage resource utilization (i.e. maximizing the tiling factor).

Tiling metrics

We define the following two metrics to calculate tiling factors, dynamically:

- In the first metric (\(\Phi_{\text{occupancy}}\)), maximizing occupancy is defined as the primary objective while tiling is maximized as a secondary objective. The objective function first strives to achieve maximum occupancy (possibly 100%) while keeping tiling to 1 and later choose to increase the tiling factor to the maximum level possible without decreasing the already determined occupancy level.

- In the second metric (\(\Phi_{\text{tiling}}\)), we do the other way around by maximizing tiling as the primary objective while keeping occupancy to the minimum (i.e. assuming only one thread-block per multiprocessor).

\(^2\)The warp occupancy, as defined by NVIDIA [2], is the ratio of active warps per multiprocessor to the maximum number of active warps supported for a multiprocessor on a GPU.
Chapter 5. An optimization case-study

The occupancy is considered in case tiling cannot be increased any further (i.e. in our case, we use at most $1 \times 16$ tiling)\(^3\).

The metrics differ in their aggressiveness for tiling. As later shown in Section 5.4, $\Phi_{\text{occupancy}}$ often results in small tiling factors but greater occupancy while $\Phi_{\text{tiling}}$ often results in relatively large tiling factors but lower occupancy.

Performance portability support

Both tiling metrics are implemented in the GPU implementations of the 2D MapOverlap skeleton and an application (e.g. 2D convolution application) that uses the 2D MapOverlap skeleton, can be configured to use either one of these two metrics. The input for these objective functions is input (including filter) dimensions and CUDA architecture parameters. The former are automatically inferred by the program execution while the latter are determined based on the compute capabilities of the CUDA architecture. By defining such metrics which can be calculated automatically with very low overhead, we can calculate the tiling filters for different problem sizes on different architectures. In the next section, we demonstrate how performance portability can be achieved with automatic calculation of tiling factors when moving to a new architecture.

<table>
<thead>
<tr>
<th></th>
<th>C2050</th>
<th>GTX280</th>
<th>8800 GT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute capability</td>
<td>2.0</td>
<td>1.3</td>
<td>1.1</td>
</tr>
<tr>
<td>Number of multiprocessors (SM)</td>
<td>14</td>
<td>30</td>
<td>14</td>
</tr>
<tr>
<td>Number of cores in a SM</td>
<td>32</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Processor Clock (GHz)</td>
<td>1.15</td>
<td>1.2</td>
<td>1.5</td>
</tr>
<tr>
<td>Local memory (KB)</td>
<td>48</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Cache support</td>
<td>yes (16KB)</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Memory bandwidth (GB/sec)</td>
<td>144</td>
<td>141.7</td>
<td>57.6</td>
</tr>
<tr>
<td>Memory interface</td>
<td>384-bit</td>
<td>512-bit</td>
<td>256-bit</td>
</tr>
</tbody>
</table>

Table 5.1: Evaluation setup.

5.4 Evaluation

The experiments are conducted on three different NVIDIA GPUs with different compute capabilities, as shown in Table 5.1. For experiments on C2050, an input image of $4096 \times 4096$ is used with filter dimensions ranging from 3 up to 27. For GTX 280 and 8800 GT experiments, an input image

\(^3\)The limit on the tiling factor is set to allow the secondary objective (i.e. maximizing occupancy) to be considered besides maximizing tiling only.
5.4. Evaluation of $2048 \times 2048$ is used with filter dimensions ranging from 3 up to 25. Furthermore, a fixed thread block size calculated based on GPU capabilities is used which is $16 \times 32$ for C2050 and $16 \times 16$ for GTX280 and 8800 GT.

The C2050 was the development platform while the GTX280 and 8800 GT are used to show performance portability and effect of an L1 cache. To be realistic, the overhead of calculating the tiling factors for a given objective function is also considered as part of the execution time, which proves to be negligible. The following implementations are referred to in the evaluation:

- *naive* implementation: The very simple CUDA implementation without any explicit optimization.
- *optimized* implementation: The naive implementation with constant memory and shared memory usage.
- *tiling-optimized* implementation: The optimized implementation with tiling. The tiling factor could be based upon either $\Phi_{\text{occupancy}}$ or $\Phi_{\text{tiling}}$.

*Filter Width* and *Filter Height* in sub-figures correspond to $2d_C + 1$ and $2d_R + 1$ respectively (see Section 3.1.3). 2D map projection is used to present 3D results in all figures. Besides scales on x- and y-axis, please consider the differences in the grey-scale in each sub-figure for the correct interpretation of results.

**Usage of shared and constant memory**

As mentioned earlier, the effect of applying shared memory and constant memory optimizations is largely influenced by the caching capabilities of a GPU. Figure 5.1 shows performance improvements over different GPU architectures for the optimized implementation over the naive implementation. On a cache based C2050, performance improvements are, on average, a factor of almost 1.5. However, on a GTX280 GPU which has no cache, the average performance difference is by a factor of 3.4. On 8800 GT, the average performance difference is by a factor of 25 which is much higher than for the GTX280. This is because of substantial difference in memory bandwidth of 8800 GT and GTX280 (see Table 5.1) which has a big performance impact for global memory accesses done frequently in the naive implementation.

**Tradeoff between $\Phi_{\text{occupancy}}$ and $\Phi_{\text{tiling}}$**

Table 5.2 highlights the tradeoff between the two metrics on different GPUs. For C2050, when maximizing occupancy ($\Phi_{\text{occupancy}}$), the tiling factor is reduced by a factor of 3.74 to gain the last 67% in occupancy. Similarly, for GTX280, the occupancy was much less when optimizing for tiling ($\Phi_{\text{tiling}}$) in comparison to when optimizing for occupancy. However, for 8800 GT,
Chapter 5. An optimization case-study

Figure 5.1: 2D convolution with *naive* (a,c,e) and *optimized* (b,d,f) implementations over different NVIDIA GPUs. Average GFLOP/s are mentioned in the caption of each sub-figure.
Figure 5.2: Tiling factors chosen when maximizing either occupancy ($\Phi_{\text{occupancy}}$, a,c,e) or tiling ($\Phi_{\text{tiling}}$, b,d,f) over different NVIDIA GPUs.
Chapter 5. An optimization case-study

<table>
<thead>
<tr>
<th></th>
<th>$\Phi_{\text{occupancy}}$</th>
<th>$\Phi_{\text{tiling}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tiling factor</td>
<td>Occupancy</td>
</tr>
<tr>
<td>C2050</td>
<td>3.83</td>
<td>100%</td>
</tr>
<tr>
<td>GTX280</td>
<td>1.63</td>
<td>75%</td>
</tr>
<tr>
<td>8800 GT</td>
<td>7.35</td>
<td>33.34%</td>
</tr>
</tbody>
</table>

Table 5.2: Average tiling factor and occupancy achieved with 2 metrics on different GPUs.

<table>
<thead>
<tr>
<th></th>
<th>C2050</th>
<th>GTX280</th>
<th>8800 GT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK_SIZE_X</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>BLOCK_SIZE_Y</td>
<td>32</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>THREADS_PER_SM</td>
<td>1536</td>
<td>1024</td>
<td>768</td>
</tr>
<tr>
<td>NUM_REGISTERS_PER_SM</td>
<td>32768</td>
<td>16384</td>
<td>8192</td>
</tr>
<tr>
<td>SHARED_MEM_SIZE_BYTES</td>
<td>48800</td>
<td>16384</td>
<td>16384</td>
</tr>
<tr>
<td>THREADS_PER_WARP</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>WARPS_PER_SM</td>
<td>48</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>THREAD_BLOCK_PER_SM</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 5.3: CUDA architecture specific parameters.

The two metrics do not yield any difference. This is because of constraints in maximizing occupancy ($\Phi_{\text{occupancy}}$) any further than what is assumed initially in $\Phi_{\text{tiling}}$ (i.e. 1 thread block per multi-processor). Figure 5.2 shows the different tiling factors chosen by the two metrics ($\Phi_{\text{occupancy}}, \Phi_{\text{tiling}}$) over different GPUs and filter sizes.

**Performance implications ($\Phi_{\text{occupancy}}$ and $\Phi_{\text{tiling}}$)**

The maximization goal may have a profound impact on the tiling factors chosen and consequently on the achieved performance as shown in Figure 5.3. Furthermore, on all GPUs, $\Phi_{\text{tiling}}$ yields better or equal performance than $\Phi_{\text{occupancy}}$ for this particular application.

**Performance portability**

Table 5.3 shows the CUDA architecture specific parameters that are used to calculate the tiling factors. The table also shows the parameter values for C2050, GTX280 and 8800 GT GPUs which can be obtained easily e.g., by querying the device capabilities. Based on these parameters alongside the problem size and filter dimensions, the two metrics generate the tiling factors for their corresponding maximization goal. As the tiling factors are automatically calculated and considered for execution, this gives us performance portability when moving from one GPU architecture to another.
Figure 5.3: 2D convolution when tiling factors are chosen for maximizing either occupancy (\(\Phi_{\text{occupancy}}, \ a,c,e\)) or tiling (\(\Phi_{\text{tiling}}, \ b,d,f\)) over different NVIDIA GPUs. Average GFLOP/s are mentioned in the caption of each sub-figure.
Figure 5.4: Performance (GFLOP/s) for 2D convolution with optimized implementation (a,c,e) and tiling-optimized implementation (Φ_{tiling}, b,d,f) over different NVIDIA GPUs. Average GFLOP/s are mentioned in the caption of each sub-figure.
without requiring manual changes in the implementation.

To illustrate performance portability, we compare performance of our solution with a baseline implementation for a given GPU architecture. For the solution, we have used the tiling-optimized implementation with $\Phi_{\text{tiling}}$ as its maximization metric. For the baseline implementation, we have considered two alternatives:

1. To use a platform-specific optimized implementation for a given GPU architecture.

2. To use a generic fairly optimized implementation that can be executed on different GPU architectures without requiring rewriting the code.

We opted for the second alternative as the first one would require a lot of extra effort for writing optimized implementations for each of the three GPUs that we have considered. Following the second alternative, we have chosen our optimized implementation as the baseline implementation for comparison. The choice is justified as the optimized implementation provides significant speedups over naive implementation for different class of GPU architectures (see Figure 5.1) and it is also fairly generic as it can run on any modern GPU with a shared memory support.

We define relative performance on a platform as ratio between the average performance of solution and the baseline implementation. By measuring this ratio, we consider our solution as performance portable if it can retain the relative performance to a potentially higher level (at least $>1$, i.e., better than the baseline implementation for every invocation).

Figure 5.4 compares the performance of 2D convolution with solution over baseline implementation. The relative performance is $3.6$, $2.3$, and $2.4$ on average for C2050, GTX280 and 8800 GT GPUs respectively which is much higher than our threshold value i.e., $1$.

**Concluding Remarks** Based on our findings, we conclude that maximizing the tiling factor at the expense of warp occupancy gives better performance for 2D MapOverlap computations on the three GPUs considered. Furthermore, we have shown how to retain performance while porting the application between different GPU architectures by automatic calculation of tiling metrics for the new architecture.

Our work is different from van Werkhoven et al. [96] in three aspects. First, our work is more about designing a generic skeleton implementation that allows e.g., convolution without usage of a weight matrix. Secondly, we present and evaluate the tradeoffs between different resource utilization criteria to calculate tiling factors with the help of separate metrics. Another important difference is that in our work, we show how to achieve performance portability by automatic calculation of these metrics while moving between different GPU architectures.
Chapter 6

SkePU StarPU integration

In this chapter, we present our work on integration of the SkePU skeleton library with the StarPU runtime system and the implementation of the farm skeleton. We start by describing the need for the integration in Section 6.1, followed by a brief description of the StarPU runtime system in Section 6.2. We then discuss different aspects of the integration work and the implementation of the farm skeleton in Section 6.3 and Section 6.4 respectively. Evaluation of the integration from different performance aspects with the help of different applications is presented in Section 6.5.

6.1 Need for runtime support

SkePU provides several implementations for each skeleton type including an OpenMP implementation for multi-core CPUs and CUDA and OpenCL\(^1\) implementations.

\[^1\]The OpenCL implementation can be used with CPUs as well but it is optimized for GPUs in our case.

\begin{verbatim}
... 
1 s1(v1, v2, out1); // MapArray skeleton call 
2 while (...) 
3 { 
4   s2(out1, v1); // Map call 1 
5   for (...) 
6   s3(v1, v3); // Map call 2 
7   res = s4(v2) // Reduce call 
8   s5(out1,v5, out2); // Map call 3 
9 } 
10 res2 = s6(out1, out2); //MapReduce call 
11 ... 
\end{verbatim}

Listing 6.1: Simplified example code (ODE solver) showing composition of different skeletons
implementations for single as well as multiple GPUs. In a platform containing both CPUs and GPUs, several \textit{implementation variants} for a given skeleton call are applicable and an optimal variant should be selected for each skeleton call. Selecting an offline optimal variant for an independent SkePU skeleton call is implemented in SkePU itself using offline measurements and machine learning (see Chapter 4). However, an application composed of multiple skeleton calls will most likely have data flow based constraints between different skeleton calls.

Listing 6.1 shows an example of such an application. Tuning such a composed application that has arbitrary nesting of different types of skeletons and also different variations of the same skeleton (e.g. different user functions) requires runtime information about resource consumption and data locality. For example, the decision made for the \texttt{s1} call in Listing 6.1 can affect the optimal choices for later skeleton calls due to data-dependency. Likewise, knowledge about subsequent skeleton calls along with their execution frequency can affect the optimality of the decision at the \texttt{s1} call. As the implementation variants in SkePU are mainly for different types of processing devices (backends), the algorithmic selection between these implementation variants can be modeled as a scheduling problem. One possibility is to determine an optimal schedule e.g. by an ILP formulation. However, some skeleton programs can be difficult to formulate in ILP due to conditional dependencies and looping between different skeleton calls. Also, an optimal solution obtained from ILP is optimal only for that particular execution environment and may prove sub-optimal on other execution environments. A viable solution in this case could be to use dynamic scheduling techniques which can consider the runtime information about data placement and locality to make their decision. Such dynamic scheduling policies are often heuristics as they consider information about the current call along with the current resource usage and data placement information to make the scheduling decision.

Besides lack of dynamic scheduling support, the original SkePU implementation has no efficient means to use multiple kinds of resources (CPUs and GPUs) simultaneously for a skeleton execution (also known as \textit{hybrid execution}). Simultaneous use of multiple computational resources present in a system can significantly increase the performance especially for compute intensive applications. To achieve both dynamic scheduling and hybrid execution capabilities, we integrate our SkePU library with the StarPU runtime system.

6.2 StarPU

StarPU \cite{15, 14} is a C-based unified runtime system for heterogeneous multicore platforms with generic scheduling facilities. Three main components of StarPU are task and codelet abstraction, data management, and dynamic scheduling framework.
6.2.1 StarPU task-model

StarPU uses the concept of codelet, a C structure containing different implementations of the same functionality for different computation units (e.g. CPU and GPU). A StarPU task is then an instance of a codelet applied to some data. A SkePU skeleton call can map to one or more StarPU tasks where different implementation variants of a SkePU skeleton call correspond to different implementations of the StarPU codelet, as discussed in Section 6.3.4. When using StarPU, the programmer has to explicitly submit all tasks and register all the input and output data for all tasks. The submission of tasks is asynchronous and termination is signaled through a callback\(^2\). This lets the application submit several tasks, including tasks which depend on others. Dependencies between different tasks can be found either by StarPU implicitly, by considering data dependencies (RAW, WAR, WAW) between submitted tasks, and/or can be explicitly specified by the programmer using integers called tags.

6.2.2 Data management

StarPU provides a virtual shared memory subsystem and keeps track of data across different memory units in the machine by implementing a weak consistency model using the MSI coherency protocol. This allows StarPU to avoid unnecessary data movement when possible. Moreover, the runtime can estimate data transfer cost and can do prefetching to optimize the data transfers. The runtime can also use this information to make better scheduling decisions (e.g. data aware scheduling policies).

StarPU defines the concept of filter to partition data logically into smaller chunks (block- or tile-wise) to suit the application needs. For example, the filter for 1D vector data is block filter which divides the vector into equal-size chunks, while for a dense 2D matrix, the filters include partitioning the matrix into horizontal and/or vertical blocks. Multiple filters can be applied in a recursive manner to partition data in any nested order (e.g. dividing a matrix into \(3 \times 3\) blocks by applying both horizontal and vertical filters).

6.2.3 Dynamic scheduling

Mapping of tasks to different execution units is provided in StarPU using dynamic scheduling. There are several built-in scheduling strategies including greedy (priority, no-priority), work-stealing, and several performance-model aware policies (e.g. based on historical execution records).

\(^2\)Task execution can be made synchronous as well, by setting the synchronous flag for a StarPU task. This makes the task submission call blocking and returns control after the submitted task finishes its execution.
6.3 Integration details

StarPU is implemented as another possible backend to the SkePU skeleton calls. The main objective of the integration was to keep the generic, high-level interface of SkePU intact while leveraging the full capabilities of StarPU underneath that interface. Some minimal changes in the skeleton interfaces were required to allow for the desired flexibility at the programmer level, but these interface extensions are kept minimal (and optional e.g. by using default parameter values) to most extent, allowing previously written SkePU programs to smoothly run with this new backend in many situations while requiring very small syntactic changes in other cases.

6.3.1 Asynchronous execution

Like most task-based runtime systems, StarPU relies on abundance of submitted (potentially independent) tasks to achieve parallelism and performance. To provide a maximum amount of task level parallelism, most SkePU skeletons\(^3\) are programmed to support asynchronous execution. The asynchronous execution allow execution of multiple skeleton calls in parallel on different backends if these skeleton calls do not have any data dependency.

6.3.2 Abstraction gap

SkePU heavily relies on features of C++ (e.g. templates, functors, classes and inheritance) while StarPU is a pure C based runtime system (e.g. using function and raw pointers). The integration is achieved by wrapping the skeleton code in static member functions which can be passed to StarPU as a normal C function pointer. The possibility of asynchronous SkePU skeleton executions resulted in severe concurrency issues (e.g. race conditions, data consistency problems). For instance, two or more skeleton calls of a single skeleton instance can be executed concurrently if they do not have any data dependency. In this case, concurrent access to the skeleton instance data, shared between multiple skeleton calls of a single skeleton instance can lead to data consistency problems. Certain programming techniques such as data-replication are used to resolve these issues while avoiding mutual exclusion. It helped in improving the performance of integration without compromising the high-level interface of SkePU.

6.3.3 Containers

The data management feature (including lazy memory copying) in the SkePU containers was overlapping with StarPU data management capabilities. To enable the integration, that data management part of the SkePU containers

\(^3\)Only Reduce and MapReduce skeletons are limited to synchronous executions as they return their result as a scalar value. However, these skeletons can also be executed asynchronously using the SkePU farm skeleton.
6.3.4 Mapping SkePU skeletons to StarPU tasks

In StarPU, the unit of execution is a task. An application needs to explicitly create and submit tasks to the runtime system. However, in our case, the creation and submission of tasks is transparently provided behind the skeleton interface. The mapping technique is illustrated in Figure 6.1a and explained below.

A SkePU skeleton call $S$ with $k$ operands $v_i$, where $1 \leq i \leq k$, each of size $N$, can be translated into one or more StarPU tasks. In direct (1:1) mapping, it translates to 1 StarPU task $t$ with $k$ operands $d_i$, each of size $N$ and $d_i = v_i \forall i$. In 1:$m$ mapping, $m$ StarPU tasks $t_j$ where $1 \leq j \leq m$ are generated, each taking $k$ operands $d_{ij}$ where $1 \leq i \leq k$ and $1 \leq j \leq m$, each of size $N'$. In our case, $N' \leq N/m$ as we divide a skeleton call into equally partitioned tasks based on operand data, considering the fact that computational complexity of data-parallel skeletons is usually the same for individual elements. For the MapArray skeleton, partitioning works as

---

4In CUDA with GT200 or newer GPUs, memory allocated through cudaHostAlloc becomes directly accessible asynchronously from the GPU via DMA and is referred to as page locked memory.
5. Implementation of the Farm skeleton described above except for the first argument which is not partitioned. The $1:m$ task-mapping is achieved with the help of data partitioning.

6.3.5 Data Partitioning

The filters in StarPU are a powerful (logical) data partitioning feature that can help in many ways. For example, it can help in dividing a bigger task into smaller independent sub-tasks, by dividing the operand data using filter(s). In this way, filters help to increase task level parallelism by creating several smaller sub-tasks which all can be executed in parallel, potentially on different execution units. It also allows executing tasks of much larger size on GPUs than what can actually fit in the GPU memory. Furthermore, filters also allow to logically layout the data according to what best suits the application needs and can significantly improve cache behavior.

Partitioning support is implemented using StarPU filters in all existing SkePU data-parallel skeletons except the Scan skeleton, by adding an optional last argument to the skeleton call, specifying the number of desired partitions for that skeleton call (defaults to 1, no-partition). The layout of partitioning depends upon the skeleton type (e.g. partition a Matrix horizontally and vertically for 2D MapOverlap row-wise and column-wise respectively).

6.3.6 Scheduling support

StarPU pre-defines multiple scheduling policies, including some based on performance models. The performance models could be either execution history based or some application specific parametric models (e.g. $an^3 + bn^2 + cn + d$). With the history based performance model, StarPU keeps track of the execution time of a task from its actual executions to guide the scheduling decisions in future [14]. We have configured all skeletons to use the history based performance model. This could be enabled, just by defining the USE_HISTORY_PERFORMANCE_MODEL flag in the application. The actual scheduling policy can later be altered at execution time using the STARPU_SCHED environment variable.

6.4 Implementation of the Farm skeleton

The current Farm implementation in SkePU uses the StarPU runtime system underneath. A task is defined either as an object overloading the call operator (i.e. functors in C++) or as a normal C/C++ function. To allow any number of tasks, the C++0X variadic template feature is used. Farm itself does not have any mechanism for defining ingoing and outgoing operands as this information is inherited from the task. With the current

---

5This is due to the semantics of the MapArray skeleton where for each element of the second operand, all elements of the first operand should be available.
Chapter 6. SkePU StarPU integration

Figure 6.2: The a) normal and b) asynchronous execution modes of the Farm skeleton. The grey part shows the execution while the white one shows idle time (either blocked or finished execution).

implementation, any SkePU skeleton (including Farm) can be used as a task. Furthermore, any task implementation using the StarPU runtime system can be wrapped into a Farm task. In the following, the implementation is discussed with respect to load-balancing, synchronization and communication aspects.

Load balancing: The ability to dynamically handle load imbalances between its tasks is an inherent feature to the semantics of the Farm skeleton. Usually, dynamic scheduling techniques are devised to solve this problem. The situation can happen because of one or more of the following:

- Submitted tasks could be of different sizes.
- A task can take different execution time depending on the worker assigned. This is highly likely in modern heterogeneous architectures that consist of both CPU and GPU workers.
- The workers can have different computational behavior and suitability for certain computational patterns (e.g. CPU workers are normally better for control decisions while modern GPU workers are well-suited for arithmetic computations).

While dynamic greedy scheduling policies (e.g., work stealing) may manage load-balancing issues in certain situations, performance model aware scheduling policies are needed to handle more complex cases (e.g., where a task is suitable to a certain worker type). With the help of StarPU, the current Farm implementation uses such dynamic performance-aware scheduling policies to manage the load balancing issues.

Synchronization: In normal execution as shown in Figure 6.2a, the Farm skeleton blocks the calling program until all spawned tasks finish their execution. This ensures that the results of all spawned tasks are available in the code immediately following the farm call. This behavior is desirable in
6.4. Implementation of the Farm skeleton

many situations; however it could be overly restrictive in other situations (e.g., where one wants to nest farm calls within other skeletons, including other farms) as no other processing can concurrently happen on the calling thread. To circumvent this issue, we also define an asynchronous mode of execution for the Farm skeleton in which the farm call submits tasks to workers and immediately returns to the calling program, as shown in Figure 6.2b. This behavior enables concurrent execution on the calling program alongside the farm execution. However, in this case, the synchronization needs

```cpp
#include <iostream>
#include "skepu/vector.h"
#include "skepu/map.h"
#include "skepu/reduce.h"
#include "skepu/farm.h"

BINARY_FUNC(plus_f, float, a, b, return a+b;

UNARY_FUNC(square_f, float, a, return a*a;

int main()
{
  skepu::Vector<float> v0(20, (float)10);
  skepu::Vector<float> v1(20, (float)5);
  float result;
  skepu::Reduce<plus_f, float> globalSum_t(new plus_f,
          &v0, &result);
  skepu::Map<square_f, float> square_t(new square_f,
          &v0, &v1);

  // 1. Normal mode
  skepu::farm(square_t, globalSum_t);

  // 2. Asynchronous mode
  skepu::setFarmNoWait(true); // enter asynchronous farm mode
  skepu::farm(globalSum_t, square_t); // farm call 1
  // other code..
  skepu::setFarmNoWait(false); // exit asynchronous farm mode
  ...
  skepu::join_farm_all(); // explicit synchronization.

  std::cout << "v1: " << v1 << "\n";
  std::cout << "result: " << result << "\n";
  return 0;
}
```

Listing 6.2: Farm skeleton example illustrating the two execution modes (normal, asynchronous).
to be explicitly enforced by the programmer using special constructs before accessing the results. Listing 6.2 shows the usage of Farm skeleton for calculating element-wise square (a Map) and global sum (a Reduce) of a vector in parallel. It also shows both execution modes (normal, asynchronous).

Communication: Many heterogeneous systems contain separate address spaces for different worker types (e.g., GPU device memory for GPU workers). The distributed memory model requires explicit communication of data across different memory units. However, the Farm skeleton lifts this communication burden from the programmer and provides data management for task operand data across different workers’ memories. Technically, this is based on the data management API of StarPU. Furthermore, to facilitate data-access, smart containers (1D Vector and 2D Matrix container, see Section 6.3.3) are defined which transparently handle the communication with the StarPU. They provide a high level interface to access their contents from the main application while ensuring data consistency when data resides in different memory units.

6.5 Evaluation

In this section, we present an evaluation of different aspects of the integration using the following four benchmark applications:

- The Separable Gaussian blur is a common operation in computer graphics that produces a new smoother and blurred image by convolving the image with a Gaussian function. The method basically calculates the new value of the pixel based on its own and its surrounding pixel values. It can be performed by running two passes over the image, one row-wise and one column-wise. Listing 6.3 shows an implementation of Gaussian blur using the SkePU 2D MapOverlap skeleton with separable overlap.

- The Coulombic potential [97] measures the interaction between water molecules. It is widely used in molecular modeling applications and analysis tools such as VMD [1]. A water molecule contains non-uniformly distributed positive and negative charges even though it is electrically neutral. The Coulombic potential is used for point charges to estimate the forces between the charged portions of each water molecule and the charged parts of its neighbors. It is implemented using the SkePU Map and MapArray skeletons.

- The iterative Successive Over-Relaxation (SOR) is a method of solving a linear system of equations in the numerical linear algebra domain [57]. It uses the extrapolation of the Gauss-Seidel method with a weighted average between the previous iterate and the computed Gauss-Seidel iterate successively for each component over $k$ iterations,
6.5. Evaluation

\[ x_i^{(k)} = w \bar{x}_i^{(k)} + (1 - w)x_i^{(k-1)} \]

where \( \bar{x} \) denotes a Gauss-Seidel iterate and \( w \) is the extrapolation factor [24]. It is implemented using the SkePU Map and 2D MapOverlap skeletons.

- A Runge-Kutta ODE solver from the LibSolve library of various Runge-Kutta solvers for ODEs [69] is ported to SkePU [44]. The LibSolve package contains two ODE test sets, one called BRUSS2D which is based on the two-dimensional brusselator equation. The other one is called MEDAKZO, the medical Akzo Nobel problem [69]. The BRUSS2D-MIX variant of BRUSS2D is implemented using SkePU Map, Reduce, MapReduce, and MapArray skeletons. Four different grid sizes (problem size) were evaluated, 250, 500, 750 and 1000.

The initial placement of data can have a profound impact on the achieved performance. So, it is important to consider the overhead of data communication as part of the execution time when comparing CPU and GPU executions, as discussed by Gregg et al. [53]. For experiments, we assume that data initially resides in the main memory. Thus, the overhead of data

```
#include <iostream>
#include "skepu/vector.h"
#include "skepu/matrix.h"
#include "skepu/mapoverlap.h"

OVERLAP_FUNC_STR(over_f, int, 2, a, stride,
    return (31824*a[-2*stride] + 43758*a[-1*stride] + 48620*a[0]
            + 43758*a[1*stride] + 31824*a[2*stride]) >>4;)

int main()
{
    skepu::MapOverlap<over_f, skepu::SKEPU_MATRIX> conv(new over_f);

    skepu::Matrix<int> m0(10, 10, (int)10);
    skepu::Matrix<int> r(10,10, (int)0);

    // Applies overlap first row-wise then col-wise
    conv(m0, r, skepu::CONSTANT, (int)0,
         skepu::OVERLAP_ROW_COL_WISE);

    std::cout << "r: " << r << "\n";
    return 0;
}
```

Listing 6.3: Separable Gaussian blur using the MapOverlap skeleton.
Figure 6.3: Speedups when applying Gaussian blur column-wise using MapOverlap column-wise skeleton on one or more CPUs. Results represent OpenMP as well as both fine partitioning (FP, 20 columns per task) and coarse partitioning (CP, 100 columns per task) version. The speedups are calculated in comparison to a sequential version executed on a single CPU without partitioning. (Compiler: gcc with -O3 optimization switch)

transfers to and from the GPU device memory is considered as part of the execution time.

**Platform:** All experiments are carried out on a GPU server with dual-quadcore Intel(R) Xeon(R) E5520 server clocked at 2.27 GHz with 2 NVIDIA Tesla M2050 GPUs.

### 6.5.1 Data Partitioning and locality on CPUs

The efficiency of the data partitioning (intra-skeleton task-parallelism) approach in comparison to OpenMP is shown in Figure 6.3 for applying Gaussian blur column wise, using a MapOverlap skeleton with 2D SkePU Matrix. The column-wise Gaussian blur is chosen as it accesses matrices (both input and output) column-wise, which is cache inefficient for regular C matrices that are stored row-wise. The OpenMP implementation is optimized to distribute work column-wise, using the static scheduling policy, as the dynamic policy performs poorly in this regular work load case. The partitioning using filters is also done vertically for column-wise MapOverlap operation and multiple independent sub-tasks are created. Two task partitioning schemes, fine partitioning (FP) and coarse partitioning (CP) with each sub-task processing 20 and 100 columns respectively, are evaluated. The baseline version is the sequential version, without any partitioning, executed on a single CPU. As shown in the figure, we are able to achieve super-linear speedups while
using multiple CPUs and also with a single CPU by partitioning a data-parallel task into sub-tasks, thanks to improved cache usage. The OpenMP versions are rewritten from a sequential version to divide work column-wise and achieve linear speedups even for smaller matrix sizes. However, the partitioning based approach was using the same sequential code (baseline), written for a single CPU and achieved better speedups than OpenMP without any significant tuning effort (no tuning of partitioning granularity).

**Hybrid execution**

With intra-skeleton task-parallelism (1:m mapping in Figure 6.1a), a skeleton operation can be executed simultaneously by multiple compute devices present in the system, including CPUs and GPUs by dividing the work between them. This allows to effectively use the system resources instead of optimizing usage of any one of them. In the following, we evaluate the support for hybrid execution with two applications.

First, we consider the Coulombic potential grid benchmark. Figure 6.4 shows the improvements from using multiple resources present in the system for the Coulombic application. This shows that usage of CPUs along the GPU yielded better performance even for an application that is known for its suitability to GPU execution [55].

Secondly, a reduction application, where global sum is calculated over
Figure 6.5: Global sum calculation for a vector over different vector sizes using the Reduce skeleton.

Figure 6.6: Work distribution between CPU and a GPU and effect of data-aware scheduling policy for the Coulombic application ($24K \times 24K$ matrix) with 3 successive executions. The upper two diagrams show the CPU and GPU usage respectively while the last one shows the overall amount of work in terms of ready tasks. The figure shows that the data-aware scheduling policy favored more computation on the GPU considering data-locality in the later part of the execution when data was available on the GPU.
6.5. Evaluation

<table>
<thead>
<tr>
<th>Scheduling Policy</th>
<th>Avg Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>heft-tm (HEFT based on Task duration Models)</td>
<td>1.38055</td>
</tr>
<tr>
<td>heft-tm-pr (heft-tm with data PRefetch)</td>
<td>1.38422</td>
</tr>
<tr>
<td>heft-tmdp (heft-tm with remote Data Penalty)</td>
<td>2.07955</td>
</tr>
<tr>
<td>heft-tmdp-pr (heft-tmdp with data PRefetch)</td>
<td>2.13410</td>
</tr>
</tbody>
</table>

Figure 6.7: Iterative SOR execution on a hybrid platform (1 GPU, 4 CPUs).

different vector sizes by using the Reduce skeleton. The reduction operation on a modern CUDA GPU is much faster than a sequential CPU execution [44]. However, still we can get some improvements with usage of one or more CPUs in conjunction with the powerful GPU, as shown in Figure 6.5. As the overhead of data transfer is considered for the GPU execution, dividing the work between GPU and one or more CPUs not only divide the computation but also decreases the data communication to GPU.

The performance gains shown above from the hybrid execution could become even more significant for system containing a powerful CPU and a low-end GPU which is a common configuration in many laptop devices.

6.5.2 Data-locality aware scheduling:

Figure 6.6 shows how a data-aware scheduling policy improves by learning at the runtime. The data-aware scheduling policy considers the current data location and expected data transfer costs between different memory locations, while scheduling a task for execution [14]. The estimate of a task execution time is also used for scheduling which is obtained from earlier executions. This is shown in Figure 6.6, while using 1 CPU and 1 GPU for three consecutive Coulombic calculations using the data-aware scheduling policy. In the first execution, the data was not available on the GPU and the estimates of earlier executions were also not there, hence work is divided across CPU and GPU in some greedy fashion. However, as time passes, more input data become available on the GPU and execution time estimates become available, resulting in more performance-aware decisions in the later part of the execution. The scheduling decisions improved significantly over successive executions as the execution time reduced by almost 7 times in the third execution in comparison to the first execution.

6.5.3 Performance-model based scheduling policies

There are several performance-aware scheduling policies available in StarPU that try to minimize the make-span of a program execution on a heterogeneous platform (known as Heterogeneous Earliest Finish Time (HEFT) scheduling policies). Figure 6.7 shows execution of Iterative SOR with such performance-aware scheduling policies available in StarPU. Average
speedups are calculated over different matrix sizes with respect to a static scheduling (CUDA execution) policy. Result shows that usage of scheduling policies with data prefetching support yielded significant performance gains.

### 6.5.4 Static scheduling

The performance gains while using dynamic scheduling capabilities of StarPU in a hybrid execution platform are shown above for different applications. These performance gains come from the task-level parallelism which depends on inter(or intra)-skeleton independence (1:1 and 1:m mapping in Figure 6.1a). Now, we consider an extreme example where static scheduling on a powerful CUDA GPU supersedes any known dynamic scheduling configuration using CPUs in conjunction.

An application with strong data dependency across different skeleton calls and small computational complexity of each skeleton call can limit performance opportunities for the runtime system to exploit. The ODE solver is such an application, containing lot of repetitive, simple mathematical operations, represented as skeleton calls. The tight data dependency between these skeleton calls allows almost no inter-skeleton parallelism. Furthermore, as tasks are computationally small, the overhead of creation of sub tasks using data partitioning to exploit intra-task parallelism limits the potential performance gains.

Figure 6.8a shows execution of the ODE solver for a fixed number of iterations on 4 CPUs. Unlike Figure 6.3, the data partitioning performed much worse in this case than the OpenMP implementation. This is due to the lack of coarse-grained parallelism in the ODE solver application, as mentioned above.

Figure 6.8b compares execution of the ODE solver with static scheduling on a CUDA GPU with a performance-aware dynamic scheduling policy. The dynamic scheduling policy tries to distribute work across hybrid execution platform which in this case consists of 2 CPUs and 1 CUDA GPU. Although static scheduling proved better for this application but a performance-aware dynamic scheduling comes quite close to it. This shows that even for such an extreme scenario, using dynamic scheduling comes quite close to static scheduling including all the overhead.

### 6.5.5 Overhead

To test the efficiency of our integration, we compare the overhead of using our integrated skeleton framework with StarPU translation to:

- **Hand-coded solution**: Different types of skeletons available in SkePU are executed for basic computations for different problem sizes, each repeated 100 times for the measurement. We do the hard comparison, by disabling the asynchronous task execution and data partitioning
6.5. Evaluation

Figure 6.8: Sequential Runge-Kutta ODE solver with limited parallelism.
feature in our framework that normally results in significant performance improvements. Also, the cost of data registration to StarPU is also included in the measurements. On the other hand, the hand-coded versions of equivalent computations were written specifically for that computation, meaning no overhead of genericity of skeletons that we bear in SkePU skeletons. Even with such a hard comparison, we have observed only 19% overhead, averaged over all executions of different skeleton types and problem sizes, for using our skeleton approach instead of a hard-coded program for a specific computation. The results are encouraging as in most applications, parallelism can be exploited both inside a skeleton call using data partitioning and between different skeleton calls by using asynchronous task execution, which can compensate for the overhead and can significantly improve the performance.

- **Original SkePU solution:** Figure 6.9 shows the execution of the ODE solver application on SkePU-StarPU combination and on original SkePU implementation while using CUDA backend (static scheduling). The overhead of original SkePU implementation in comparison to hand-coded solution is already shown to be negligible for this application (see Section 3.2.2). In this case, the SkePU-StarPU combination incurs some overhead in comparison to original SkePU implementation. This mainly accounts to task-creation overhead for large number of smaller (computation-wise) tasks with no inter-task parallelism.
Chapter 7

Related Work

7.1 Skeleton programming

Cole [30, 31] is normally considered as pioneer of the skeleton programming concept. However, also other work on identifying basic data and task parallel constructs in parallel applications has been reported during the same period. This includes initial formulations of certain task-parallel skeletons such as farm [23, 88], pipeline [64], divide and conquer [65], and branch and bound [50] and some early work on data-parallel skeletons [70] such as scan [25]. In earlier work on skeleton programming, the main focus was on achieving abstraction by building skeleton applications [84] from a high level representation. For this purpose, some suggested the usage of activity graphs [33] to model skeleton computations while others introduced high-level coordination languages [37] such as Skeleton Coordination Language (SCL) [38], the Skeleton Imperative Language (Skil) [26], the Pisa Parallel Programming Language (P3L) [16], the lle language [40], and the Single Assignment C language (SAC) [54].

Functional paradigm

Due to focus on abstraction, earlier realization efforts of skeleton programming are mostly made in functional languages including syntactic extensions to parallel Haskell such as Higher-order Divide-and-Conquer language (HDC) [59] and Eden [76]. Usage of existing functional language constructs such as functors to introduce skeleton support in existing functional languages is reported for Concurrent Clean [62], ML [79], OCamlP3l [46], Skipper [90], and the Hope [36] language. All these implementations share a common drawback of functional programming, i.e., trading performance for achieving easier and shorter programming style. Due to this reason, Loidl et al. [75] showed that a program written using a functional language based skeleton framework can perform up to an order of magnitude worse than
the hand-coded implementation in C/MPI.

**Object-oriented paradigm**

SkeTo [4] provides data parallel skeletons for distributed data structures such as lists (arrays) [92], matrices (2D arrays), and trees. As a C++ template library, it has limited support for fusing consecutive skeleton calls into one call. The fusion optimization can increase performance when applicable by removing intermediate data and synchronization overhead; however it only works with nested skeleton expressions for the list data-structure. The Munster Skeleton Library Muesli [29] supports both data-parallel (for 1D arrays and 2D dense and sparse matrices) and task parallel (Pipeline, Farm, BranchAndBound, and DivideAndConquer) skeletons. It also supports parallelism inside a single MPI-node using OpenMP. eSkel [32, 22, 21] is a quite low-level skeleton library that exposes certain parallelism concerns related to distribution directly to the application programmer. It provides a pipeline and a deal skeleton which is a variation of the pipeline skeleton with stage replication and fusion capabilities. eSkel focuses on nesting of skeletons and provides two nesting modes, transient mode where skeleton objects are destroyed immediately after invocation or a more lasting persistent mode. MALLBA [45] focuses on the combinatorial optimization problem domain. It defines skeletons such as Dynamic Programming, Simulated Annealing, Tabu Search, and Genetic Algorithms. All these skeleton libraries (SkeTo, Muesli, eSkel, MALLBA) are implemented in C/C++ with MPI support for distributed memory systems consisting of multiple MPI nodes.

There exist also several Java-based skeleton libraries including Calcium [27], JaSkel [11], Lithium [9], Muskel [8, 7], Quaff [48], CO2P3S [78] and Skandium [72]. These frameworks mainly differ in their skeleton types and the skeleton composition techniques they imply. Most of them have support for distributed memory systems while Skandium is mainly designed for multicore CPUs with shared memory support.

SkePU differs from all the skeleton frameworks mentioned above in its focus on GPU-based heterogeneous systems. As GPUs are suitable for more compute-intensive applications, SkePU skeletons are designed to leverage these computational capabilities while minimizing the communication overhead. Moreover, it differs in its support for dynamic scheduling, algorithmic selection of skeleton implementation and simultaneous usage of heterogeneous resources including CPUs and GPUs.

**GPU computing**

There exist some related solutions in the GPU computing domain. Thrust [61] is a C++ template library for CUDA that implements functionality like transform (map), reduction, prefix-sum (scan), sorting etc. It became part of the NVIDIA SDK distribution from CUDA version 4.0 onwards. CUDPP is a library of data-parallel algorithm primitives such as parallel prefix-sum,
parallel sort and parallel reduction [58]. It does not however provide higher-order functions which can take any user defined function as an input.

SkelCL [89] is a skeleton library implemented using OpenCL and is similar to SkePU in functionality. Currently, it implements four data parallel skeletons named Map (for one input operand), Zip, Reduce and Scan where Zip is basically a Map operation with two input operands. The SkelCL skeletons operate on a one-dimensional vector data type which provides similar capabilities as the SkePU vector, e.g., the memory management across main memory and GPUs device memories. However, one notable difference exists between SkePU and SkelCL in the specification of data distribution for a container across difference device memories. SkelCL requires the application programmer to explicitly specify the data distribution for its vector container. In this way, the application programmer controls how a skeleton execution maps to different devices present in the system. In SkePU, data distribution for both vector and matrix containers is handled transparently to the application programmer. This gives freedom to the SkePU framework on the actual execution of a skeleton call, including selection of a skeleton implementation to use and the ability to partition the work between different devices in a transparent way. Furthermore, SkelCL has no support for operations like MapOverlap and MapArray. Also it does not have support for two-dimensional skeleton operations and task-parallelism.

One of the main differences between SkePU and the skeleton libraries described in this section is that SkePU can be compiled with several back-ends for both CPU and GPU which opens for the possibility of tuning back-end selection. Most other works have used either NVIDIAs CUDA framework or OpenCL for their implementations. SkePU also seamlessly integrates multi-GPU support and provides hybrid CPU-GPU execution functionality in its implemented skeletons.

7.2 Hybrid execution and dynamic scheduling

OpenCL provides a hybrid execution capability for GPU-based systems. In [55], Grewe and O’Boyle propose a static scheme for partitioning the work of an OpenCL program between CPUs and a GPU. They use static analysis to extract different code features from an OpenCL program and use machine-learning to build and train a model that maps code features to partitions. Once trained, the model can predict (near-)optimal partitioning for new, unseen OpenCL programs. However, using their solution for systems containing NVIDIA GPUs and x86 CPUs is not obvious. This is due to inherent limitations in the NVIDIA OpenCL implementation as it cannot be used for x86 CPUs. Using two different OpenCL implementations, one for NVIDIA GPUs and one for x86 CPUs is possible but it still does not seamlessly allow hybrid execution, considering the fact that they are considered two different OpenCL platforms with separate contexts and queues.
Besides OpenCL, there exist several task-based mapping approaches for dividing the work between CPU and GPU workers. Qilin [77] is a heterogeneous programming system that relies on offline profiling to create performance models for each task on different devices which is later used to calculate the partitioning factor across the devices. However, unlike dynamic scheduling capabilities achieved in SkePU, the performance models in Qilin are calibrated through training executions. For a generalized reduction pattern (also known as MapReduce [39]), work-sharing based dynamic scheduling strategies are implemented, for single-node [85] as well as clusters [87] of GPU-based systems. In [94], Wang et al. propose a dynamic scheduling scheme for optimizing energy consumption for GPU-based systems, by coordinating inter-processor work distribution and per-processor frequency scaling. The work on the reduction pattern is applicable to applications that follow this type of computational pattern and thus cannot be generalized for other applications.

In [56], Grossman et al. suggest a declarative and implicitly parallel coordination language called CnC-CUDA, based on Intel’s Concurrent Collections (CnC) programming model. The proposed extensions allows to model control and data flow and can help generate the hybrid CPU/GPU execution. This is different from our work as it exposes a new programming model while automatically generating the GPU execution code which can be far less optimal, considering the rapid evolution of GPU architectures.

7.3 Algorithmic selection and Auto-tuning

Algorithmic selection is addressed by many in the context of modern homogeneous and heterogeneous systems. In [66], Kessler and Löwe discuss algorithmic selection for explicitly parallel software components for a multi-processor machine where a component has multiple implementation variants. Each implementation variant of a component contains metacode for each of its performance-aware methods $f$ that is used to predict the execution time of the method $f$, based on problem and processor group sizes. A component together with such performance-prediction metacode supplied by the component provider is called a performance-aware component. The composition tool works by offline calculating dispatch tables using an interleaved dynamic programming algorithm that are looked up at the runtime to find the expected best implementation variant, processor allocation and schedule for a given problem and processor group sizes. Their approach works best for divide and conquer algorithms where variant and schedule selection for each component invocation in the call tree often results in significant performance gains while comparing to any fixed variant execution.

PetaBricks [10] is an implicitly parallel language and compiler framework that addresses performance portability with the help of auto-tuning, mostly across homogeneous multi-core architectures. The Merge framework [74] targets a variety of heterogeneous architectures, while focusing on MapRe-
duce [39] as a unified, high-level programming model. Thomas et al. [93] implements the algorithmic selection for the Standard Template Adaptive Parallel Library (STAPL) where offline machine learning is used to generate selection tests whereas the actual selection is done at the runtime, based on the execution of the selection tests. Similarly, automated selection among different algorithmic variants of reductions on shared-memory multiprocessors has been considered by Yu and Rauchwerger [98].

Besides algorithmic selection, auto-tuning can be used to fill-in values for tunable parameters. The auto-tuning framework can internally use model-driven optimization and/or empirical optimization. The model-driven auto-tuning approaches are typically applied in the compiler domain where analytical models are often used to determine values for tunable parameter such as determining the loop tiling and unrolling factors. We have used this approach to determine the tiling factor for the 2D convolution application where GPU device capabilities and problem sizes are used to determine the appropriate tiling factor. When applicable, this approach can give portable performance without incurring any overhead associated with empirical auto-tuning. On the other hand, the auto-tuning frameworks based upon empirical optimizations rely on generating a large number of parametrized code variants for a given algorithm and do actual executions on a given platform to discover the variant that gives the best performance. Our work on determining thread, block sizes and number of threads for GPU and OpenMP skeleton implementations respectively uses empirical auto-tuning. Several works has been reported using empirical auto-tuning for GPU based kernels such as [73, 35, 91]. All these works rely on code generation to generate several code variants filled with different values of the tunable parameters, and later do the training executions to make the final selection. Furthermore, they employ knowledge in the form of GPU device capabilities and application hints to reduce the search space for parameters. The solution space, in our case was small and the search space pruning was mainly carried out by the heuristic algorithm where non-optimal choices are eliminated as the selection process proceeds.
Chapter 8
Discussion and Future Work

There are of course many ways to improve and extend our work. We will describe here what we think are interesting directions for future research.

8.1 SkePU extensions

SkePU is a work in progress and can be extended in many ways. The main idea that should drive the future extensions in SkePU is the ability to effectively implement more and more applications using the SkePU skeletons. This could help in showing effectiveness and broad applicability of skeleton programming approach for GPU-based heterogeneous systems.

Two-dimensional skeleton operations

The implementation of remaining two-dimensional dense matrix skeleton operations such as row-wise and column-wise reduction needs to be done. Similarly, more variations of Map, MapReduce and MapArray skeletons can be designed to allow implementation of a broader set of applications using these skeleton types. This could enable building more application using SkePU skeletons, for example, from dense linear algebra such as dense matrix multiplication and LU decomposition. These applications are well-suited for GPU-execution and thus can be a good prospects for showing effectiveness of skeleton programming, for GPU-based systems. There is also a possibility to implement a sparse matrix data type and related skeleton operations [18, 81]. The operations on sparse matrices such as sparse matrix-vector multiplication are widely used in solving sparse linear systems and eigenvalue problems [43, 80, 95]. However, unlike dense matrix operations, sparse matrix operations are typically much less regular in their access patterns and can be tricky to efficiently implement on throughput oriented processors, such as GPUs [19].
Task parallel skeletons

Currently, the farm skeleton is implemented using the StarPU runtime system. This adds an external system dependency on the StarPU runtime system to use the SkePU farm skeleton. In future, we plan to implement the farm and other task parallel skeletons in the SkePU library without any external system support. This would require to implement runtime support within the SkePU library, defining the notion of task, worker, queues and scheduler. The memory management functionality in SkePU that currently works for SkePU containers only, would then needs to be extended to support other data types.

There exist several task-parallel skeletons that can be interesting for future work. Parallel_for skeleton allows applying the same function independently on multiple data items. One classical example of the Parallel_for skeleton is dense matrix multiplication which could be divided into $n$ sub matrix multiplication operations, each calculating a sub-set of the output matrix. The Pipeline skeleton supports both task and data-parallelism by running multiple operations (called stages) in parallel, where operations are applied to each data-item in an ordered fashion. Other skeleton choices include Branch-and-bound and Divide-and-conquer.

Porting existing applications

The computation and communication patterns that are modeled by SkePU skeletons may exist in many more applications than what we possibly know today. One example of such a pattern is the MapOverlap pattern which was initially found in image processing applications such as image convolution. Recently, we have found out that this pattern is present in other, much larger applications such as unstructured grid-based solvers for computational fluid dynamics (CFD) [34]. Such kind of studies where new applications are investigated and implemented with the existing skeleton-set can be important for showing broad applicability of computational patterns modeled by the existing skeletons.

Porting SkePU to further platforms

Currently, SkePU is implemented for single-node GPU-based systems containing multicore CPUs and one or several GPUs. A possibility for future work include porting the SkePU library to further platforms. One option is implementing support for MPI-based clusters where each MPI node may possibly contain one or more GPUs. This would require besides other things, changing the data-management API for allowing lazy memory copying work across different MPI nodes while keeping data in a consistent state. Another possibility is to port it to the 48-core Intel SCC (Single-chip Cloud Computer) architecture that has an on-chip network and uses message passing to communicate between different processing cores on the chip [5].
8.2 Algorithmic selection and PEPPHER

A topic for future work is to further investigate the selection between multiple implementations of a single functionality, on a given platform. In SkePU, the algorithmic choice exists mainly for different kinds of computational resources present in a GPU-based system (sequential CPU, multi-core OpenMP, GPU). However, we can imagine other scenarios where the algorithmic choice is more pervasive, such that multiple implementations exist for a single computational resource. These implementations can be explicitly provided by the programmer or can be generated automatically by a composition tool e.g., by providing values of tunable parameters in a generic implementation. One simple example is the sorting functionality where multiple sorting implementations can exist for sorting on a sequential CPU. This kind of algorithmic choice is a topic of current and future research in our project, PEPPHER.

PEPPHER

PEPPHER [20] is a 3-year European FP7 project that addresses Performance Portability and Programmability of Heterogeneous many-core architectures. PEPPHER mainly focuses on the GPU-based systems as a heterogeneous architecture and addresses the performance and programmability problem on a given GPU-based system, and code and performance portability between different GPU-based systems. The PEPPHER approach is pluralistic and parallelization and language agnostic, aiming to support multiple parallel languages and parallelism frameworks. It addresses performance portability at different levels of the software stack, proposing a flexible and powerful component model for specifying performance-aware implementation variants, data structures and adaptive auto-tuned algorithm libraries, portable compilation techniques, and a flexible resource aware run-time system.

The fundamental premise of PEPPHER for enabling performance portability is to maintain multiple, tailored implementation variants of performance-critical components of the application and schedule these variants efficiently either dynamically or statically across the available CPU and GPU resources. The PEPPHER component model allows the representation of multiple implementations of a function behind a single interface, which is called a component. To assist in algorithmic selection, a component implementation need to expose its performance-relevant properties in an XML-based component descriptor. The component descriptor includes information about a component’s dependencies, computation and communication resource requirements, performance-prediction functions and tunable parameters. As part of the PEPPHER project, SkePU skeletons can be considered as predefined components for frequently occurring computations that do not need XML-based component descriptors, and are readily available, as part of the PEPPHER library.
Algorithmic selection in PEPPHER

The algorithmic selection between multiple implementation variants is a topic of current and future research in PEPPHER. Currently, the limited dynamic algorithmic selection is supported in the PEPPHER runtime system by considering one implementation per computational resource, similar to what we have in SkePU. This needs to be extended in future, by a static algorithmic selection phase where information from component descriptors along with the training and historical execution records can be used to statically make the selection between different implementation variants of a component.
Chapter 9

Conclusions

In this thesis, we have investigated the skeleton programming approach, with the help of SkePU skeleton programming framework, for programming modern GPU-based heterogeneous systems (CPU/GPU) in an efficient and portable manner.

First, we presented the extensions in the SkePU library to support two-dimensional operations. This allowed implementation of several new applications. The implementation work on the SkePU library mainly targets close-to hand-written performance while providing it behind a generic interface. As shown in the experiments, a skeleton program written using SkePU can achieve performance comparable to hand-written code, thanks to optimized skeleton implementations.

Any high-level programming approach for GPU-based systems must not sacrifice on performance. We have presented a case-study on optimizing a GPU-based skeleton implementation for 2D convolution computations while keeping the generic skeleton interface. Furthermore, to address the issue of retaining performance while porting the application, we have introduced two metrics to maximize either computational or storage resource utilization on a GPU. Experiments have shown that by automatic calculation of these two metrics for an application, performance can be retained while porting the application from one GPU architecture to another. The experiments also suggest that focusing on the optimal usage of storage resources such as registers and local memory rather than computational resources performs better over different architectures for the convolution computation.

In GPU-based systems, multiple types of computational devices exist. As a SkePU skeleton call has implementations available for different computational devices, it can be scheduled to execute on any of them or a combination of them by dividing the work between them. The former is a scheduling (or algorithmic selection\(^\ddagger\)) problem while the latter is known

\(^\ddagger\)As algorithmic choice in the SkePU library is mainly for different type of backends (CPU, GPU), the algorithmic selection can be modeled as a task scheduling problem.
as the hybrid execution capability. The technique used before in SkePU was static scheduling where the backend choice is specified by the programmer during the compilation time. We have extended it in two ways: First, a machine learning based auto-tuning framework is implemented to support automatic selection of (near-)optimal algorithmic choice, for a given platform. The framework is demonstrated to successfully predict the (near-)optimal backend for any number of repetitive executions for a single skeleton call. Secondly, for more complex skeleton program executions, we have implemented the dynamic scheduling support. One advantage of dynamic scheduling is load-balancing support that is critical especially for hybrid executions where the work needs to be divided across different types of computational devices with different architectures. Coming up with an optimal partitioning factor for a computation before the actual execution is hardly possible. The dynamic scheduling can better handle these issues; especially, a performance-aware dynamic scheduling policy that considers the historical execution record and data-locality while making the scheduling decisions. Furthermore, the hybrid execution capability showed significant speedups over usage of any single computational device even for computations that are previously known to be highly suited for that type of computational device.

SkePU initially supported only data-parallel skeletons. The first task-parallel skeleton (farm) in SkePU is implemented with support for performance-aware scheduling and hierarchical parallel execution by enabling all data parallel skeletons to be usable as tasks inside the farm construct.

We have also described topics for future work, both for SkePU and for algorithmic selection in our EU FP7 project PEPPHER.

where a skeleton call maps to a task containing different implementations, that needs to be scheduled on a specific backend.
Bibliography


In this thesis, we address issues associated with programming modern heterogeneous systems while focusing on a special kind of heterogeneous systems that include multicore CPUs and one or more GPUs, called GPU-based systems. We leverage the skeleton programming approach to achieve high level abstraction for efficient and portable programming of these GPU-based systems and present our work on SkePU which is a skeleton library for these systems.

We first extend the existing SkePU library with a two-dimensional (2D) data type and accordingly generalized skeleton operations, and implement several new applications that utilize these new features. Furthermore, we consider the algorithmic choice present in SkePU and implement support to specify and automatically optimize the algorithmic choice for a skeleton call, on a given platform.

To show how to achieve high performance, we provide a case-study on an optimized GPU-based skeleton implementation for 2D convolution computations and introduce two metrics to maximize resource utilization on a GPU. By devising a mechanism to automatically calculate these two metrics, performance can be retained while porting an application from one GPU architecture to another.

Another contribution of this thesis is the implementation of runtime support for task parallelism in SkePU. This is achieved by integration with the StarPU runtime system. By this integration, support for dynamic scheduling and load balancing for SkePU skeleton programs is achieved. Furthermore, a capability to do hybrid execution by parallel execution on all available CPUs and GPUs in a system, even for a single skeleton invocation, is developed.

SkePU initially supported only data-parallel skeletons. The first task-parallel skeleton (farm) in SkePU is implemented with support for performance-aware scheduling and hierarchical parallel execution by enabling all data parallel skeletons to be usable as tasks inside the farm construct.

Experimental evaluations are carried out and presented for algorithmic selection, performance portability, dynamic scheduling and hybrid execution aspects of our work.

Title
Skeleton Programming for Heterogeneous GPU-based Systems

Author
Usman Dastgeer

Abstract
In this thesis, we address issues associated with programming modern heterogeneous systems while focusing on a special kind of heterogeneous systems that include multicore CPUs and one or more GPUs, called GPU-based systems. We leverage the skeleton programming approach to achieve high level abstraction for efficient and portable programming of these GPU-based systems and present our work on SkePU which is a skeleton library for these systems.

We first extend the existing SkePU library with a two-dimensional (2D) data type and accordingly generalized skeleton operations, and implement several new applications that utilize these new features. Furthermore, we consider the algorithmic choice present in SkePU and implement support to specify and automatically optimize the algorithmic choice for a skeleton call, on a given platform.

To show how to achieve high performance, we provide a case-study on an optimized GPU-based skeleton implementation for 2D convolution computations and introduce two metrics to maximize resource utilization on a GPU. By devising a mechanism to automatically calculate these two metrics, performance can be retained while porting an application from one GPU architecture to another.

Another contribution of this thesis is the implementation of runtime support for task parallelism in SkePU. This is achieved by integration with the StarPU runtime system. By this integration, support for dynamic scheduling and load balancing for SkePU skeleton programs is achieved. Furthermore, a capability to do hybrid execution by parallel execution on all available CPUs and GPUs in a system, even for a single skeleton invocation, is developed.

SkePU initially supported only data-parallel skeletons. The first task-parallel skeleton (farm) in SkePU is implemented with support for performance-aware scheduling and hierarchical parallel execution by enabling all data parallel skeletons to be usable as tasks inside the farm construct.

Experimental evaluations are carried out and presented for algorithmic selection, performance portability, dynamic scheduling and hybrid execution aspects of our work.

Keywords
skeleton programming, GPU programming, SkePU, performance, portability
No 17  Vojin Plavsic: Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)
No 73  Ola Strömfors: A Structure Editor for Documents and Programs, 1986.
No 177  Peter Åberg: Design of a Multiple View Presentation and Interaction Manager, 1989.
No 181  Henrik Eriksson: A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.
No 380 Johan Ringström: Compiler Generation for Parallel Languages from Denotational Specifications, 1993.

FHS 3/94

FHS 4/94

No 483 Eva Toller: Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 533 Staffan Flodin: Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 546 Magnus Werner: Multidatabase Integration using Polymorphic Queries and Views, 1996.
No 558 Patrik Nordling: The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.


Charlotte Björkergen: Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.


Svein Bergum: Managerial communication in telework, 2000.


Jean Paul Meynard: Control of industrial robots through high-level task programming, 2000.


Magnus Kald: The role of management control systems in strategic business units, 2000.


Fredrik Elg: Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Mono, 2002.


Andreas Kill: Översättningar av en managementstend - En studie av införandet av Balanced Scorecard i ett landsting, 2005.

He Tan: Aligning and Merging Biomedical Ontologies, 2006.


FiF-a 90 Amra Hailovic: Ett praktikperspektiv på hantering av mjukvarukomponenter, 2006.


FiF-a 91 Hanna Broberg: Verksamhetsanspassade IT-praktikperspektiv på hantering av mjukvarukomponenter, 2006.


Jiri Trnka: Prerequisites for data sharing in emergency management, 2007.


Erik Kuiper: Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008.


Martin Karresand: Completing the Picture - Fragments and Back Again, 2008.


Mirko Thorstensson: Using Observers for Model Based Data Collection in Distributed Tactical Operations, 2008.


Mohammad Saifullah: Exploring Biologically Inspired Interactive Networks for Object Recognition, 2011.
No 1468  **Qiang Liu**: Dealing with Missing Mappings and Structure in a Network of Ontologies, 2011.
No 1481  **Anna Vapen**: Contributions to Web Authentication for Untrusted Computers, 2011.
No 1504  **Usman Dastgeer**: Skeleton Programming for Heterogeneous GPU-based Systems, 2011.