Linköping Studies in Science and Technology

Thesis No. 1504

Skeleton Programming for Heterogeneous GPU-based Systems

by

Usman Dastgeer

Submitted to Linköping Institute of Technology at Linköping University in partial fulfilment of the requirements for degree of Licentiate of Engineering

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

Linköping 2011

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
ISSN 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 Kärrman 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

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 can we 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:

1. Johan Enmyren, Usman Dastgeer, and Christoph W. Kessler. Towards A Tunable Multi-Backend Skeleton Programming Framework for Multi-GPU Systems. *MCC'10: Proceedings of the 3rd Swedish Workshop on Multicore Computing.* Gothenburg, Sweden, Nov. 2010.

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

for Multi-GPU Systems. *IWMSE'11: In Proceeding of the 4th international workshop on Multicore software engineering.* ACM, New York, USA, 2011.

3. Usman Dastgeer, Christoph W. Kessler and Samuel Thibault. Flexible runtime support for efficient skeleton programming on hybrid systems. Accepted for presentation: *ParCo'11: International Conference on Parallel Computing.* Ghent, Belgium, 2011.

4. Usman Dastgeer and Christoph W. Kessler. A performance-portable generic component for 2D convolution on GPU-based systems. Submitted: *MCC'11: Fourth Swedish Workshop on Multicore Computing.* Linköping, Sweden, 2011.

5. Christoph W. Kessler, Sergei Gorlatch, Johan Enmyren, Usman Dastgeer, Michel Steuwer, and Philipp Kegel. Skeleton Programming for Portable Many-Core Computing. In: *S. Pllana and F. Xhafa, eds., Programming Multi-Core and Many-Core Computing Systems*, Wiley-Blackwell, New York, USA, 2011 (to appear).

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 Enmyren 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

```
BINARY_FUNC(plus_f, double, a, b,   struct plus_f
    return a+b;                     {
)                                       skepu::FuncType funcType;
                                        std::string func_CL;
           // EXPANDS TO:  ====>        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)\n"
                                            "{\n"
                                            "   return a+b;\n"
                                            "}\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

```
UNARY_FUNC(name, type1, param1, func)
UNARY_FUNC_CONSTANT(name, type1, param1, const1, func)
BINARY_FUNC(name, type1, param1, param2, func)
BINARY_FUNC_CONSTANT(name, type1, param1, param2, \
const1, func)
TERNARY_FUNC(name, type1, param1, param2, param3, func)
TERNARY_FUNC_CONSTANT(name, type1, param1, param2, \
param3, const1, func)
OVERLAP_FUNC(name, type1, over, param1, func)
OVERLAP_FUNC_STR(name, type1, over, param1, stride, func)
OVERLAP_DEF_FUNC(name, type1)
ARRAY_FUNC(name, type1, param1, param2, func)
ARRAY_FUNC_MATR(name, type1, param1, param2, func)
ARRAY_FUNC_MATR_CONST(name, type1, param1, param2, const1,
    const2, func)
```

Listing 3.1: Available macros.

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:

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

It also provides operations to resize a matrix and split the matrix into submatrices.

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

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

```
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.

```
#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 $v_1 \dots v_k$. The vectors have length N. A more formal way to describe this operation is:

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

- *For matrix operands*, every element in the result matrix r is a function f of the corresponding elements in one or more input matrices $m_1 \dots m_k$. 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:

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

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`.

```
#include <iostream>

#include "skepu/matrix.h"
#include "skepu/reduce.h"

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

int 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: " <<r <<"\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.

```
#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 $(v_1 \dots v_k)$ or matrix $(m_1 \dots m_k)$ objects, where $k \leq 3$ as described above. Formally:
For vectors:

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

For matrices:

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

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 `dotProduct` is created which maps two input vectors with `mult_f` and then reduces the result with `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:

- *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 `overlap(`d`)`. Formally:

  $$r[i] = f(v[i-d], v[i-d+1], \dots, v[i+d]) \; \forall i \in \{1, \dots, N\}.$$

  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.

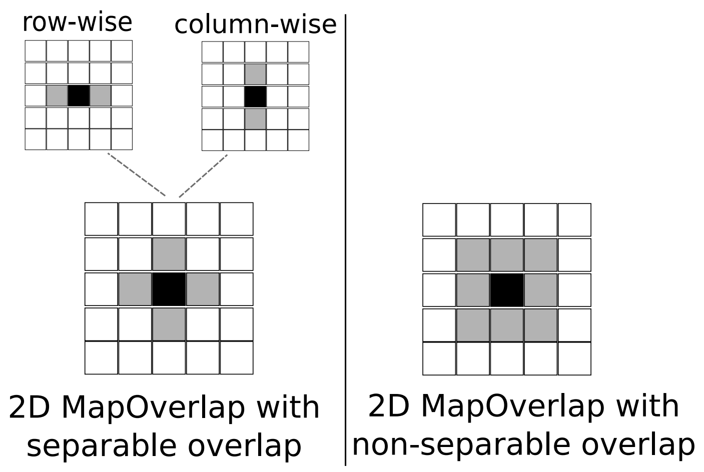

row-wise
column-wise
2D MapOverlap with separable overlap
2D MapOverlap with non-separable overlap

Figure 3.2: Difference between 2D MapOverlap with separable and non-separable overlap.

- *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:

$$RC/(R+C)$$

For instance, for a 15×15 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 `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[1]. The number of these elements is controlled by the parameter `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

[1]The actual access distance between matrix elements could be different; for example, if a matrix is stored row-wise but adjacency is defined in terms of columns.

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 `overlap_rows(`d_R`)` and `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 `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:

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

```
#include <iostream>

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

OVERLAP_FUNC(over_f, float, 2, a,
    return a[-2]*0.4f + a[-1]*0.2f + a[0]*0.1f +
           a[1]*0.2f + a[2]*0.4f;
)

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

    skepu::Vector<float> v(15,10);
    skepu::Vector<float> r;

    conv(v, r, skepu::CONSTANT, (float)1);

    std::cout<<"Result: " <<r <<"\n";

    return 0;
}

// Output
// Result: 7.6 9.4 13 13 13 13 13 13 13 13 13 13 13 9.4 7.6
```

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, $0 \pm \mathit{overlap}$. A MapOverlap skeleton is instantiated with `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 `skepu::CONSTANT` parameter with value `(float)1`.

```
#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 v1 = 1 2 3 4...
    //     v2 = 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;
}

// Output
// Result: 10 9 8 7 6 5 4 3 2 1
```

Listing 3.6: A MapArray example that reverses a vector

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:

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

```
#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;
}

// Output
// Result: 1 2 3 4 5 6 7 8 9 10
```

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 `v1` and `v2` as inputs and `r` as output. `v1` will be corresponding to parameter `a` in the user function `arr_f` and `v2` to `b`. Therefore, when the skeleton is applied, each element in `v2` can be mapped to any number of elements in `v1`. In Listing 3.6, `v2` contains indexes to `v1` of the form 9, 8, 7..., therefore, as the user function `arr_f` specifies, the first element in `r` will be `v1[9]` the next `v1[8]` etc, resulting in a reverse of `v1`.

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 $v[1], v[2], \cdots v[N]$, 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 `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).

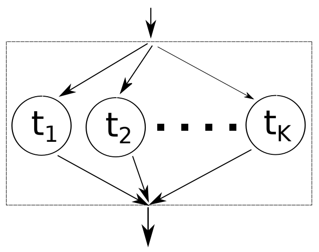


Figure 3.3: A Farm skeleton.

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

$$farm(t_1, t_2, \dots t_K)$$

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 `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[3]. 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]The same example is also implemented using the first approach, shown later in Section 6.5.

```
OVERLAP_FUNC(blur_kernel, int, 19, a,
  return (a[-9] + 18*a[-8] + 153*a[-7] + 816*a[-6] + 3060*a
      [-5]
          + 8568*a[-4] + 18564*a[-3] + 31824*a[-2] + 43758*a
              [-1]
          + 48620*a[0] + 43758*a[1] + 31824*a[2] + 18564*a[3]
          + 8568*a[4] + 3060*a[5] + 816*a[6] + 153*a[7]
          + 18*a[8] + a[9])>>18;
)
```

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.

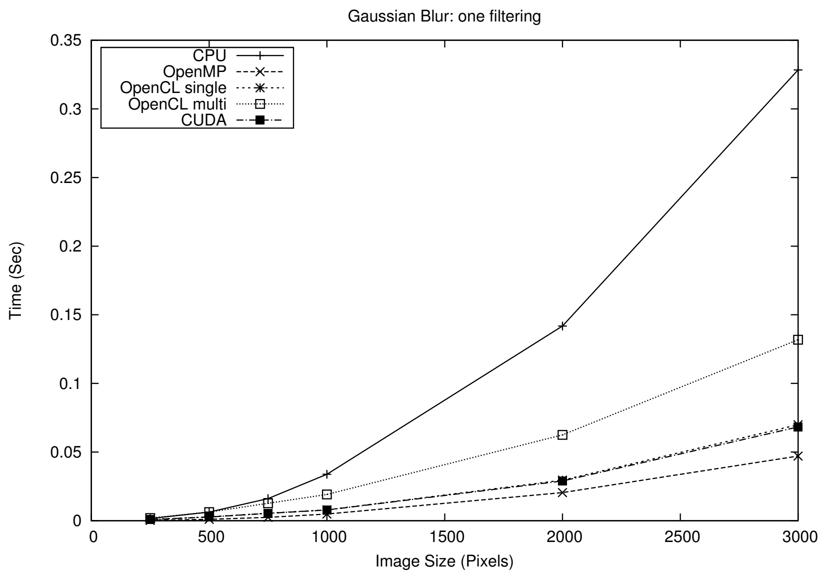
Timing was only done on the actual blur computation, not including the loading of images and creation of vectors. For CUDA and OpenCL, the time for transferring the image to the GPU and copying the result back is included. The filtering was done with two passes of a 19-value filter kernel which can be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

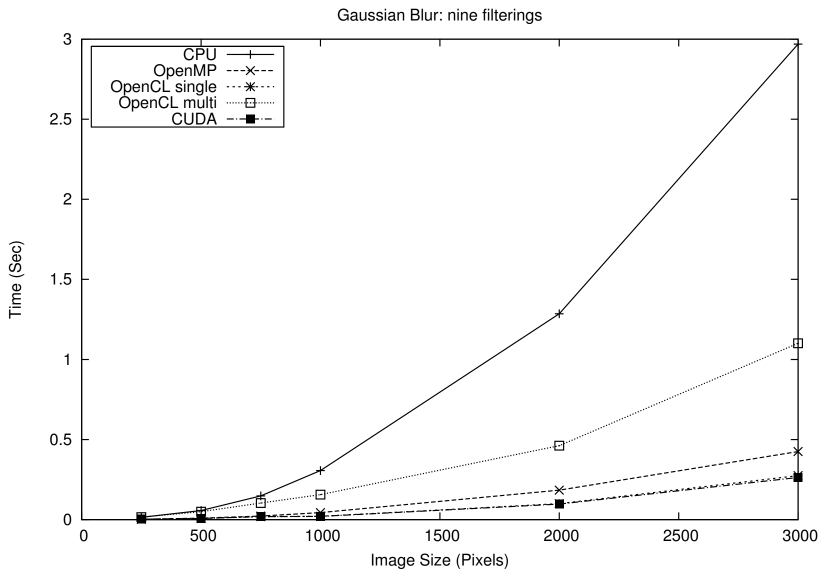
Since there is a data dependency in the MapOverlap skeleton when running on multiple-GPUs, we also see that running this configuration loses a lot of performance when applying MapOverlap several times in a row because it needs to transfer data between the GPUs, via the host.

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).

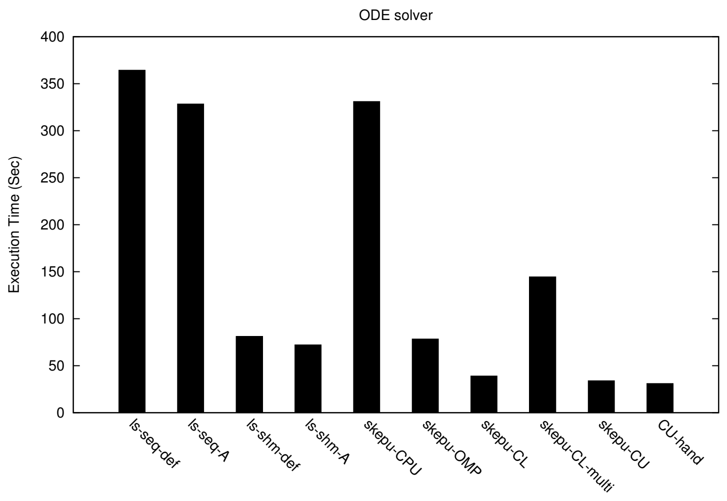


Figure 3.5: Execution-times for running different LibSolve solvers, averaged over four different problem sizes (250,500,750 and 1000) with the BRUSS2D-MIX problem.

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.