SkelCL – A Portable Multi-GPU Skeleton Library

Michel Steuwer, Philipp Kegel, and Sergei Gorlatch

04/10 - I

SkelCL – A Portable Multi-GPU Skeleton Library

Michel Steuwer, Philipp Kegel, and Sergei Gorlatch

Department of Mathematics and Computer Science
University of Münster, Münster, Germany

November 30, 2010

Abstract

Modern Graphics Processing Units (GPU) are increasingly used as general-purpose processors. While the two currently most widely used programming models for GPUs, CUDA and OpenCL, are a step ahead as compared to extremely laborious shader programming, they still remain effort-demanding and error-prone. Memory management is complex due to the separate memories of CPU and GPU and GPUs support only a subset of common CPU programming languages. Both, CUDA and OpenCL lack support for multi-GPU systems, such that additional challenges like communication and synchronization arise when using multiple GPUs.

We present SkelCL – a novel library for simplifying OpenCL programming of single and multiple GPUs. SkelCL provides for the application programmer two main abstractions:

Firstly, a unified memory management that implicitly transfers data transfers between the system's and the GPU's memory. Lazy copying techniques prevent unnecessary data transfers and increase performance. Secondly, SkelCL provides so-called algorithmic skeletons to describe computations. Algorithmic skeletons encapsulate parallel implementations of recurring programming patterns. Skeletons are configured by providing one or several parameters which adapt a skeleton parallel program pattern according to an application. In addition a set of basic skeletons, SkelCL also provides means to easily compose skeletons, to allow for more complex applications.

An important advantage of SkelCL is that it allows for employing all devices of a multi-GPU systems automatically. The algorithmic skeletons can divide and coordinate the computation among multiple GPUs. SkelCL's memory management facilities ensures that input data is distributed properly among the available GPUs.

Using a Mandelbrot set computation as well as an industrial-strength medical imaging application, we demonstrate that SkelCL greatly alleviates programming of single GPU and especially of multi-GPU systems, while providing competitive performance.

Keywords: SkelCL, GPU Computing, Algorithmic Skeletons, Multi-GPU

Contents

1 Introduction **1**

1.1 Graphics Processing Unit (GPU) 2

1.1.1 Differences to a Central Processing Unit (CPU) 4

1.2 GPU Computing 5

1.2.1 OpenCL 6

1.2.2 Challenges of GPU Computing 10

1.3 Skeletal Programming 11

1.3.1 Skeleton Libraries 12

2 SkelCL **13**

2.1 Overview of SkelCL 13

2.2 Common Classes in SkelCL 15

2.2.1 The Context Class 15

2.2.2 The DeviceInfo Class 15

2.3 Memory Management 16

2.3.1 The Vector Class 16

2.3.2 Behind the Scenes: Implementation of the Vector Class . 17

2.4 Skeletons in SkelCL 18

2.4.1 Unary and Binary Skeletons 19

2.4.2 Creating Skeletons 20

2.4.3 Launching Skeletons 21

2.4.4 Behind the Scenes: Kernel Generation 23

2.4.5 Skeletons in SkelCL: Map, Zip, Reduce, Scan 28

2.4.6 Additional Arguments 36

2.5 Chaining Skeletons 39

2.5.1 The Sequence 40

2.5.2 The Composition 43

2.6 Multi-GPU 48

2.6.1 Challenges of Multi-GPU Computing 48

2.6.2 Multi-GPU Memory Management 48

2.6.3 Multi-GPU Skeletons 51

3 Applications **55**
3.1 Mandelbrot Set 55
3.1.1 Implementation with SkelCL 56
3.1.2 Comparison to Other Implementations 57
3.1.3 Results 59
3.1.4 Conclusion 62
3.2 List-Mode Ordered Subset Expectation Maximization 63
3.2.1 Implementation with SkelCL 65
3.2.2 Comparison to Other Implementations 67
3.2.3 Results 69
3.2.4 Conclusion 70

4 Conclusion **71**
4.1 Related Work 71
4.2 Future Work 75
4.3 SkelCL Summarized 79

Bibliography **81**

List of Figures

1.1 Schematic overview of a GPU connected to a CPU. 3
1.2 The OpenCL platform model. 6

2.1 Schematic overview of SkelCL. 14
2.2 Class diagram of all skeleton classes. 19
2.3 Binary tree built by the reduce skeleton. 32
2.4 The up-sweep phase of the scan algorithm. 35
2.5 The down-sweep phase of the scan algorithm. 35
2.6 Tree representation for the expression `(A+B)*(C+D)`. 43
2.7 Class diagram of all classes related to the composition. 46
2.8 Visualization of a depth-first tree traversal. 47
2.9 An example of the multi-GPU scan algorithm used in SkelCL. . 53

3.1 An area of the mandelbrot set. 56
3.2 Runtime results for the Mandelbrot set. 59
3.3 Runtime results for different work-group sizes. 61
3.4 Runtime results for the best versions. 61
3.5 Runtime results for SkelCL's multi-GPU version. 62
3.6 Runtime results for a single iteration of the list-mode OSEM algorithm. 69
3.7 Speedup for one iteration of the list-mode OSEM algorithm. . . 70

List of Listings

2.1 Exemplary use of the `Vector` class. 16
2.2 A template class definition. 18
2.3 Exemplary use of a function pointer in C or C++. 20
2.4 A skeleton is instantiated by providing a function definition as string. 20
2.5 A skeleton is instantiated by providing a path. 21
2.6 Calculation of the dot product of two vectors. 22
2.7 A simple map skeleton. 24
2.8 Kernel generation. 26
2.9 Function returning the typename as a string for a given type. . . 28
2.10 The map skeleton in SkelCL. 29
2.11 The zip skeleton in SkelCL. 30
2.12 The implementation of the zip skeleton. 31
2.13 The reduce skeleton in SkelCL. 32
2.14 The scan skeleton in SkelCL. 34
2.15 Passing additional arguments to a map skeleton. 36
2.16 The combined source code from the map skeleton and the user-provided function. 37
2.17 The implementation of the map skeleton including the capability of passing additional arguments to the user-provided function. . 38
2.18 Generated source code by SkelCL. Two additional arguments are passed to the kernel and then to the user-provided function. . . 39
2.19 Creation and execution of a sequence. 40
2.20 Generated sequence class for two skeletons. 42
2.21 Building a composition for the expression `(A+B)*(C+D)`. 43
2.24 Declaration of the `+` operator for two vector instances. 44
2.22 A skeleton is instantiated by providing a predefined kernel definition. 44
2.23 Source code for executing the composition built for the expression `(A+B)*(C+D)`. 44
3.1 Launching the same kernel, or skeleton respectively, in CUDA, OpenCL, and SkelCL. 58

3.2 High-level implementation of the OSEM algorithm in pseudo code. 64
3.3 Sequential implementation of the list-mode OSEM algorithm. . 65
3.4 Implementation of the list-mode OSEM algorithm in SkelCL. . . 66
3.5 Code necessary to perform data exchange between multiple devices in CUDA. 68
4.1 A simple map example with SkePU. 74

Chapter 1

Introduction

Modern *graphics processing units* (GPUs) provide large numbers of processing units combined with a high memory bandwidth. While these devices originally have been used for graphics rendering only, they turned into highly efficient parallel processors that also allow for general-purpose computations. Hence, GPUs are increasingly used in a variety of applications to increase performance. In particular, computationally intensive scientific applications benefit from the use of GPUs. New programming models have been developed to alleviate general-purpose computing on the GPU, also known as *GPU computing*. The two most popular approaches to GPU computing are CUDA and OpenCL [NVI06b, THH08]. While both programming models are similar, CUDA is a proprietary software developed by NVIDIA, whereas OpenCL is an open industry standard.

Programming parallel architectures has always been difficult and error-prone. Both, CUDA and OpenCL, are low-level programming models: As GPUs have separate memories, memory allocation and deallocation on the GPU has to be controlled by the programmer. Moreover, data has to be transferred explicitly from the system's main memory to the GPU's memory and back. In OpenCL, a particularly lengthy program initialization is also required, and low-level API functions have to be called to pass parameters to GPU functions. Besides, these functions have to be specified as so-called kernel functions, as OpenCL cannot execute common program functions on a GPU. As a result, GPU programs require a considerable amount of low-level boilerplate code, which renders GPU computing tedious and error-prone. Moreover, CUDA and OpenCL lack specific support for programming multi-GPU systems. Thus, even more programming effort is required when using multiple GPUs.

In this paper, we present *SkelCL* – a library for high-level multi-GPU programming. SkelCL is based on two principal approaches for simplifying GPU programming: Firstly, a unified memory management that simplifies the handling of graphics memory and, secondly, the use of algorithmic skeletons to specify calculations for GPUs in a high-level manner. Furthermore, SkelCL allows for using multi-GPU systems to speed up the calculations even further. SkelCL is build on top of OpenCL. Therefore, it is not bound to a specific hardware or vendor and can be executed on any OpenCL-capable device.

The rest of this paper is structured as follows:

The remainder of this chapter provides a basic introduction to the architecture of GPUs, the OpenCL API, and its application in GPU computing. We also describe the major challenges of programming GPUs. A brief introduction to algorithmic skeletons concludes this chapter.

In chapter two, we present SkelCL: we show program examples that use SkelCL, as well as implementation details of our library. In particular, the use of multiple GPUs is explained in this chapter.

The third chapter features applications implemented with SkelCL. Programs written in native OpenCL and CUDA code are compared to corresponding implementations that use SkelCL. Measurements are shown as well as source code to demonstrate how SkelCL simplifies GPU programming as compared to standard GPU programming models.

We summarize our work in the final chapter, comparing SkeCL to related work and discuss possible future enhancements.

1.1 Graphics Processing Unit

The main objective of a graphics-accelerator card is to render 3D graphics. The whole design of the hardware is laid out to optimize this task. One of the most common operations performed in 3D rendering are matrix multiplications with four-dimensional matrices, which are done entirely in hardware. Several steps have to be performed every single time an image is rendered. Therefore, a pipeline structure is used in the design of graphics hardware. This is called the *rendering pipeline*.

Rendering of more detailed graphics became more important over time. To increase the quality of the images, the so far static rendering pipeline has to become more flexible. The ability to influence intermediate results inside the pipeline was added. At predefined points in the pipeline, small software programs – called *shaders* – are executed. Different types of shaders evolved for different purposes and the graphics hardware changed to allow the execution of shaders. Common to all shaders is the inherent parallelism. For example, one *pixel shader* is executed per pixel in the output image.

The development of the last years led to *unified shading architectures*. All

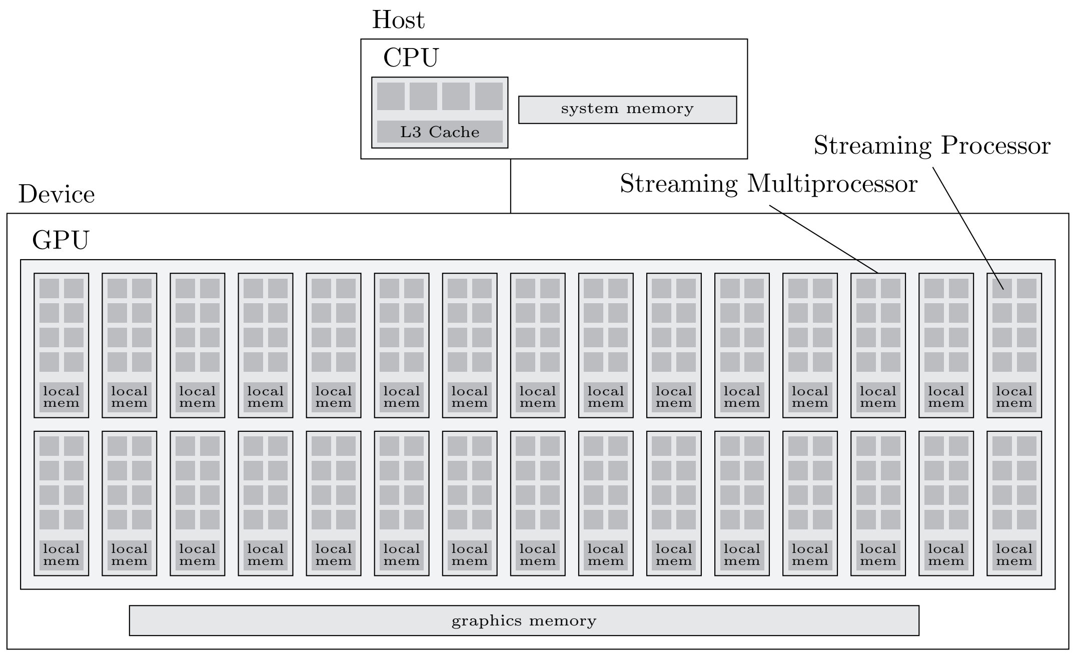


Figure 1.1: Schematic overview of a GPU connected to a CPU (based on [LOS09]).

different types of shaders are executed in the same processing unit on graphics-accelerator cards with a unified shading architecture. This saves space on the graphics hardware and the central processor can operate at full capacity. This processing unit is commonly known as the *graphics processing unit* (GPU). The GPU differs from a *central processing unit* (CPU) in many ways.

A modern GPU by NVIDIA is shown in Figure 1.1. It consist of 240 so called *stream processors* (SP) which are organized in groups of eight. Such a group is called a *streaming multiprocessor* (SM). Each SM has local memory, which can be accessed from all its SPs. The SMs (in Figure 1.1 there are 30) are all connected to the graphics memory. Modern graphics-accelerator cards provide up to several GBytes of graphics memory. The whole graphics hardware is connected to the rest of the system through the PCI Express bus. The graphics hardware is often called the *device* whereas the rest of the system is the *host*. The host is shown at the top of Figure 1.1. It consists of a CPU and the system memory. The host system has more components, like a hard disk, which are less important in GPU computing and thus are omitted here.

One SM acts similar to a *single instruction multiple data* (SIMD) unit, though

		SIMD width	Freq. (GHz)	Bandwidth (GB/sec)	Peak perf. (GFLOPS)
Core i7-960	4 Cores	4	3.2	32	102.4
Tesla T10	30 SMs	8	1.3	141	933.1

Table 1.1: Comparison of the NVIDIA Tesla T10 GPU and the Intel Core i7-960 CPU [LKC+10].

differences to a classical SIMD unit exist. Basically, all SPs inside a SM perform the same operation at the same time. But the execution model of a SM is a bit more flexible because not all SPs have to participate in a certain step. This allows the common execution of code that branches across multiple threads. This is called *single instruction multiple thread* (SIMT) by NVIDIA [NVI10].

Different SMs can perform entirely different tasks. So a GPU can be seen as a collection of multiple SIMD units. Though it is common to execute the same task on different SMs.

In our tests, we used the NVIDIA Tesla S1070 GPU Computing System which consists of four individual Tesla T10 devices. One Tesla T10 is similar to the schematically shown GPU since it has 30 SMs on board, making it a total of 240 SPs. It is built from 1.4 billion transistors and has a nominal single floating point peak performance of 933.1 GFLOPS. In contrast, the Core i7-960, a modern CPU by Intel, consists of four cores and about 700 million transistors are used to build it. Its single floating point peak performance at 102.4 GFLOPS is only about a ninth compared to the GPU. The most important specifications of both processors are shown in Table 1.1. Though, different opinions on the objective performance boost of GPUs compared to CPUs exist [LKC+10]. It seems undeniable that certain data parallel algorithms are rather well suited for GPUs [FB09, vMAF+08, SVGM08].

1.1.1 Differences to a CPU

CPUs and GPUs are build for different purposes, so they differ in various points. CPUs are primarily designed for integer arithmetic, whereas GPUs have been optimized to process floating point numbers. For example, a GPU automatically combines a floating point multiplication and an addition to a single instruction, which is performed in only one step (*fused multiply-add* (FMA) [LOS09]). A major difference between a CPU and a GPU is the handling of *thread stalls*. When executing multiple threads, a thread stall occurs when the running thread needs to wait, for example, for a memory load to finish. On a CPU, switching threads is very expensive because the content of all registers must be saved and the new content must be load into them. This is called a *context switch*. Much effort on a CPU is done to avoid such situations or to minimize the time a thread waits for a memory load by the use of caches. A GPU does not perform a context switch at all. On a GPU much more registers are available, such that the context of multiple threads is stored and it is possible to switch between them without any overhead. A modern GPU has up to 16 384 registers per SM whereas one single x86-64 CPU core has only 16 registers [NVI10, AMD09]. On the other hand, a CPU has a sophisticated cache hierarchy to avoid the occurrence of thread stalls. Most GPUs do not have a cache at all, and if so, the hierarchy is flat and the caches are not as big as on a CPU.

Many concepts common in a CPU environment are entirely missing on a GPU like paging or a virtual address space. Even the memory layout is quite different. On a CPU the memory is divided in a stack and heap segment, but there is nothing similar in GPU memory. Therefore, function calls are handled differently on a GPU as no return address can be stored on a stack. As a result, recursion is currently forbidden in a GPU environment.

1.2 GPU Computing

Programming the GPU is different than programming the CPU. The different architectures require different programming models. Therefore, new programming models were developed with new graphics hardware. At first, the *OpenGL* programming environment was used for general purpose computations. OpenGL is a programming interface for producing 3D computer graphics [SA09]. Data had to be stored inside of textures and shaders were used to operate on them. This approach had limitations because the hardware is used in a way it is not designed for. To overcome these limitations, the *brook stream program language* [BFH$^+$04] and the *sh library* [MQP02] were developed. These were first attempts to provide a universal programming environment on GPUs. The first programming model, which is widely used in research and economy, is *CUDA*. CUDA was announced by NVIDIA at the end of 2006 along with NVIDIAs first generation of GPUs with a unified shading architecture [NVI06b, NVI06a]. The current version is 3.1, which was released in June 2010.

In 2008, Apple proposed to establish an open industry standard for GPU

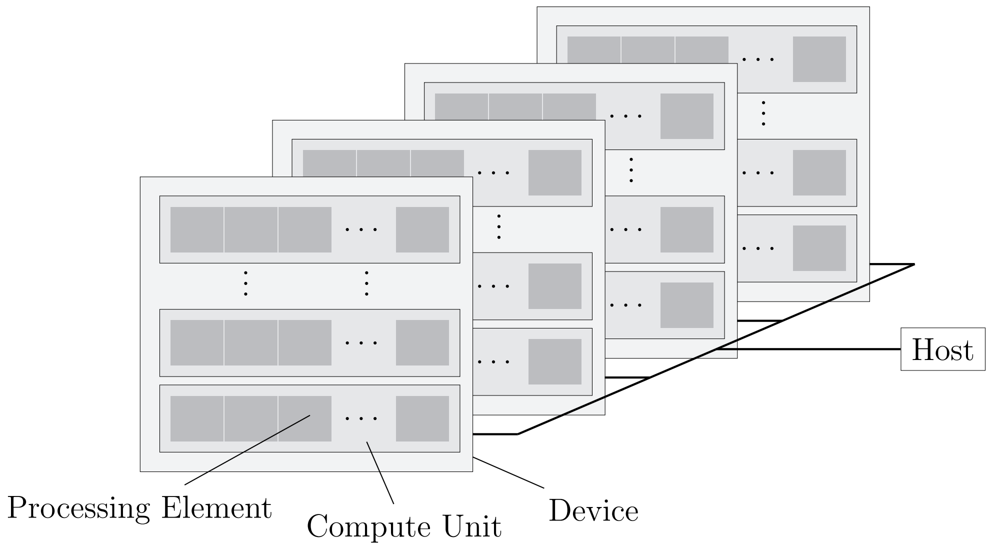


Figure 1.2: The OpenCL platform model.

programming. In cooperation with AMD, IBM, Intel, NVIDIA and others, a working group was formed to develop *OpenCL*.

1.2.1 OpenCL

OpenCL was presented for the first time at the Siggraph Asia in December 2008 by AMD and NVIDIA [THH08]. It is an open industry standard managed by the Khronos Group [Mun10]. The idea of OpenCL is to provide a unified interface to heterogeneous parallel systems. In contrast to CUDA, it is not bound to hardware by a specific vendor. The first official release of the OpenCL specification was published in October 2009. In June 2010 the current version 1.1 was published.

The OpenCL Platform OpenCL distinguishes between the *host* system and *OpenCL devices*. OpenCL applications run on the host and launch special functions, which are executed on the devices. These special functions are called *kernel functions*. The host manages and coordinates the execution of the kernel functions on the devices. The OpenCL platform model is shown in figure 1.2.

The host side is not used to perform calculations in OpenCL. Therefore, it is mostly unimportant for OpenCL and, thus, unspecified. In contrast, the structure of the devices is specified. An OpenCL device consists of one or more *compute units* (CUs), that are divided into one or more *processing elements* (PEs). All computation on the device is performed in the PEs. This design is the foundation for the execution model, which describes how kernel functions are executed on an OpenCL device. Different hardware can be used as an OpenCL device, for example, a GPU or the CPU itself. For instance, a dual-core CPU is a single OpenCL device with two CUs and a single PE per CU. The OpenCL Platform model was created with the GPU in mind, so that GPUs can be mapped directly to OpenCL devices. In OpenCL, a GPU appears as one device with one CU per SM. The SPs of a SM are the PEs in OpenCL terms.

The Execution Model A kernel function is a function which is simultaneously executed multiple times on a device. All parts, other than kernel functions, are called the *host program*. The host program manages the execution of kernel functions on OpenCL devices. It has to decide how many instances of a kernel function are launched. Therefore, an index space is specified. For every point in the index space, an instance of the kernel is launched. This single instance of a kernel is called a *work-item*. A work-item can be identified by its position in the index space. This identifier is called the *global ID*. Every work-item executes the same code, but the execution can vary per work-item due to branching according to the global ID. Normally, a kernel function operates on a set of data which implicitly defines the index space. For each element of this set, a work-item is launched which processes this element.

Work-items are organized in so called *work-groups*. When a kernel function is started, the host code specifies how many work-items are launched and how many work-items form a work-group. The work-items must be evenly divisible into work-groups. The host program is responsible to ensure this requirement. Work-groups have a unique *work-group ID* to be distinguishable from each other. Inside a work-group, every work-item has a *local ID*. The IDs are related to each other, the global ID of a work-item can be calculated from the local ID and the work-group ID. Work-groups are useful to split the global work into related parts. Therefore, it can help the programmer to structure the code in a natural way. An important guarantee is also bound to the concept of work-groups. All work-items in one work-group are guaranteed to be executed on the same CU. Synchronizations between all work-items in the same work-group are possible with a barrier function. On the other hand, synchronization between work-items of different work-groups are not possible because there are no guarantees given for the order of execution between multiple work-groups. Therefore, the code has to be written in a way that synchronization is only required in the work-groups, but not between arbitrary work-items. A work-group can have up to 512 work-items. Some, but not all, OpenCL devices support atomic memory operations in addition to synchronization.

The Memory Model As we discussed the details of a GPU in Section 1.1, we saw that a GPU distinguishes between different memory areas. This is also reflected in the OpenCL memory model. OpenCL differs between four different types of memory: *global*, *constant*, *local* and *private* memory.

The global memory is the largest part of the available memory. All work-items in all work-groups can read and write to the global memory at any time. An order of the memory access can not be predicted, so race conditions might occur. To avoid race conditions, atomic operations on global memory are available on some OpenCL devices and can be used with an OpenCL extension. The host can read and write global memory to upload data to the device and fetch the results after the kernel execution.

Constant memory is a special region of global memory which stays unchanged during a kernel execution. Work-items can only read from constant memory, whereas they cannot not write to it. The host can access the constant memory the same way as global memory.

Local memory is a memory region which is only available for work-items of the same work-group. Local memory can be used to exchange data between multiple work-items in one work-group. Memory consistency of the local and global memory across work-items of the same work-group is guaranteed after synchronization, with the `barrier` function. Memory consistency between work-items from different work-groups can not be guaranteed. Local memory is smaller, but can be accessed faster than global or constant memory. The host cannot access local memory directly.

Private memory is private to a work-item. This means, only one work-item can access variables in this private memory.

It is very important to point out that the memory management has to be done explicitly by the programmer. The different memory types do not correspond to different stages in a CPU cache hierarchy. The programmer is responsible to move data between the different memory areas. Typically, the data is first copied from host memory into the global memory of the device . To achieve best performance, the data is copied to the local memory when calculations are performed on the data. Afterward, the result is copied back into global memory. After the kernel finished, the data is copied back to the host.

The OpenCL C Programming Language In contrast to the host code, a special programming language has to be used for writing kernel functions. This language, the OpenCL C programming language, is based on the C99 standard, with additional restrictions and modifications. Hence, it is very familiar to many programmers. Because it is based on C, object oriented concepts from C++ are not supported, like classes for example. Major differences to C99 are:

- Pointers have to be annotated with an address space qualifier to specify the corresponding address space.
- Kernel functions that are launched from the host must be annotated with the function qualifier `__kernel`. Functions called from kernel functions do not have to be annotated.

The most important restrictions are:

- Dynamic memory allocation is not supported. Hence, the length of all arrays must be known a priori.
- None of the C standard headers are supported. Therefore, functions like `malloc` or `printf` are not available.
- Recursion is not supported. Functions on a device cannot call themselves. To achieve a similar behavior iterative constructs can be used or kernel functions can be called multiple times by the host.

OpenCL provides built-in functions to determine local and global IDs as well as work-group IDs. The size of a work-group and the global size can also be accessed through built-in functions. Because no functions from C standard headers are available, most standard mathematics functions are missing, too. To overcome this issue, OpenCL provides a wide range of mathematics functions as built-in functions, like `log` or `sqrt`.

Building Kernel Functions To execute a kernel function, the source code defining the function has to be compiled first to executable binary code. This is done explicitly by passing the source code of a kernel function as a string to the OpenCL driver. Therefore, the executable binary code for the device is build at runtime by the host program. The compiled source code can be written to disk in a binary representation and later be used directly as a kernel function. This can speed up the build process considerably.

OpenCL and Multiple Devices OpenCL provides access to all devices of a system. This allows for programming multiple devices, even if the devices are not made by the same vendor or even not of the same type. For example, a mixed configuration with a CPU and multiple GPUs can be programmed using only OpenCL. In OpenCL, one *command queue* is created per device. Kernels are launched by submitting them to a command queue. Because the command queue is bound to a device, the command queue used dictates the device on which the kernel is started. Multiple kernels can be queued in a command queue. The command queue executes the kernels in-order. An out-of-order execution can be configured if the device supports it. Besides, memory operations like copying data from or to the device are queued in command queues as well. Synchronization functions are used to wait for kernels to finish.

Differences to CUDA

OpenCL is very similar to CUDA. The basic concepts are shared and kernel functions written in CUDA have basically the same restrictions as in OpenCL. This makes it very easy to convert a CUDA program to an OpenCL program and vice versa. Unlike OpenCL, CUDA provides a special compiler called `nvcc` which is used to compile the device code as well as the host code. This allows for a special syntax for the host code, for example, for launching kernel functions.

Multi-GPU programming is possible with OpenCL, though no special support is provided. This is not different to CUDA. In CUDA, one host thread per device has to be created to manage the device. Therefore, CPU thread programming is required for multi-GPU programs whereas such a restriction does not exist in OpenCL.

1.2.2 Challenges of GPU Computing

As shown in Section 1.2.1, GPU computing differs from CPU computing. Most notably the whole programming model is optimized for data-parallel programming. But also abstractions known from the CPU, like a cache, are absent on a GPU. Hence, the most pitfalls come from the different architecture and the unfamiliar environment. This includes the missing recursion on a GPU or the requirement to use the provided IDs to branch between multiple kernels. Launching a kernel function is also inconvenient because the number of kernel instances to be created has to be specified. Furthermore, the kernel instances have to be divided into work-groups. This decision implies which kernel instances can synchronize between each other. Therefore, it is very important to choose the total count of kernel instances as well as the size of the work-groups properly. In addition, the memory management on the GPU with different address spaces is complicated and has to be mastered manually.

Beside these architectural problems, some tasks in GPU computing just require a lot of boilerplate code to be done. In particular, the memory transfers between CPU memory and GPU memory are similar in most applications. The input data is transferred to the GPU before computation and the result is transferred in the opposite direction afterward. The same code has to be written many times with the risk of making mistakes every time. Due to the almost entirely missing debugging features of current GPU programming environments, it is hard to find those mistakes. The initial start-up of the OpenCL system is also lengthy and could be simplified.

1.3 Skeletal Programming

The complexity of parallel programming is not unique to GPUs, since many years different approaches have been proposed to simplify parallel programming. In the late 1980s, Murray Cole developed the concept of skeletal programming [Col89]. It is heavily inspired by functional programming paradigms and separates the distribution and communication necessary for a parallel execution from the actual performed calculation.

Algorithmic skeletons are parallel *higher-order functions*. A higher-order function is a term used in functional programming and describes a function that takes another function as argument or returns a function as return value. Hence, higher-order functions operate on other functions. An algorithmic skeleton takes one or more functions as an argument. It executes them in a parallel manner. Different skeletons exist to describe different kinds of parallelism. The necessary distribution and communication to achieve the parallelism is hidden from the programmer. Solutions for frequent or basic problems can be provided as skeletons. Combinations of skeletons are used to solve more complex algorithm.

A set of well known skeletons exists. These can be divided into *data-parallel* and *task-parallel* skeletons. Task-parallelism aims for performance enhancements by performing different tasks at the same time. In contrast, data-parallelism achieves performances improvements by performing the same calculation on a whole bunch of data at once instead of on a single value. Skeletons can support both models by specifying the necessary distribution and communication.

1.3.1 Skeleton Libraries

Usually, multiple skeletons are grouped together in libraries. Two of these are briefly introduced below. Both of them run on multi-core CPUs or cluster systems, but none on GPUs.

eSkel

Murray Cole and Anne Benoit developed a C-based skeleton library called *eSkel* [BCGH05]. It provides some common task-parallel skeletons. Data-parallel skeletons are not supported. eSkel is implemented using the *Message Passing Interface* (MPI) [MPI09]. This enables eSkel to be run on cluster systems. A basic idea of eSkel is to be an addition to the existing standards C and MPI, but not a replacement. Therefore, MPI code can be mixed with skeletons to achieve maximum flexibility.

Muesli

Muesli is a skeleton library written in C++, developed in Münster [CPK09]. Task-parallel and data-parallel skeletons are available. Muesli offers various distributed data structures, on which data-parallel skeletons operate. Task-parallel skeletons are implemented as independent classes and can be combined and nested arbitrarily. Like eSkel, Muesli is built on top of MPI. Unlike eSkel, Muesli entirely hides the underlying MPI. Furthermore, OpenMP [OMP08] is used to improve performance on multi-core clusters.

Chapter 2

SkelCL

In this chapter, we describe SkelCL. SkelCL is a skeleton library for GPU computing. It is implemented using OpenCL and can, therefore, be used with a wide range of graphics hardware. Furthermore, SkelCL supports multi-GPU systems.

In the first section of this chapter, a general overview of the different components of SkelCL is given. Afterward, the components of SkelCL are described in detail. Of particular interest are the two key abstractions: the memory management and the algorithmic skeletons. The multi-GPU capabilities of SkelCL are described in a separate section.

2.1 Overview of SkelCL

Figure 2.1 shows an overview of the layout of SkelCL. SkelCL can roughly be divided into four parts. The common classes part is the smallest of the four parts. It provides the interface to the OpenCL system which SkelCL is based on. This part is discussed in Section 2.2.

The memory management part provides two classes for memory abstractions: The `Vector` class and the `Scalar` class which is derived from the `Vector` class. The memory management is described in Section 2.3.

Skeletons are an important part of SkelCL since skeletons describe computation and communication patterns. The central class is the `Skeleton` class from which all skeletons are derived. Four different skeletons are implemented in SkelCL. These are the Map, zip, reduce and scan skeleton. Every skeleton has a kernel attached to it. A kernel is – as presented in the first Chapter – the description of the calculation which is performed on the GPU. The skeletons and their implementations are presented in Section 2.4.

The fourth part of SkelCL is entitled "chaining skeletons". The classes in this part can be used to compose skeletons. Composed skeletons are capable

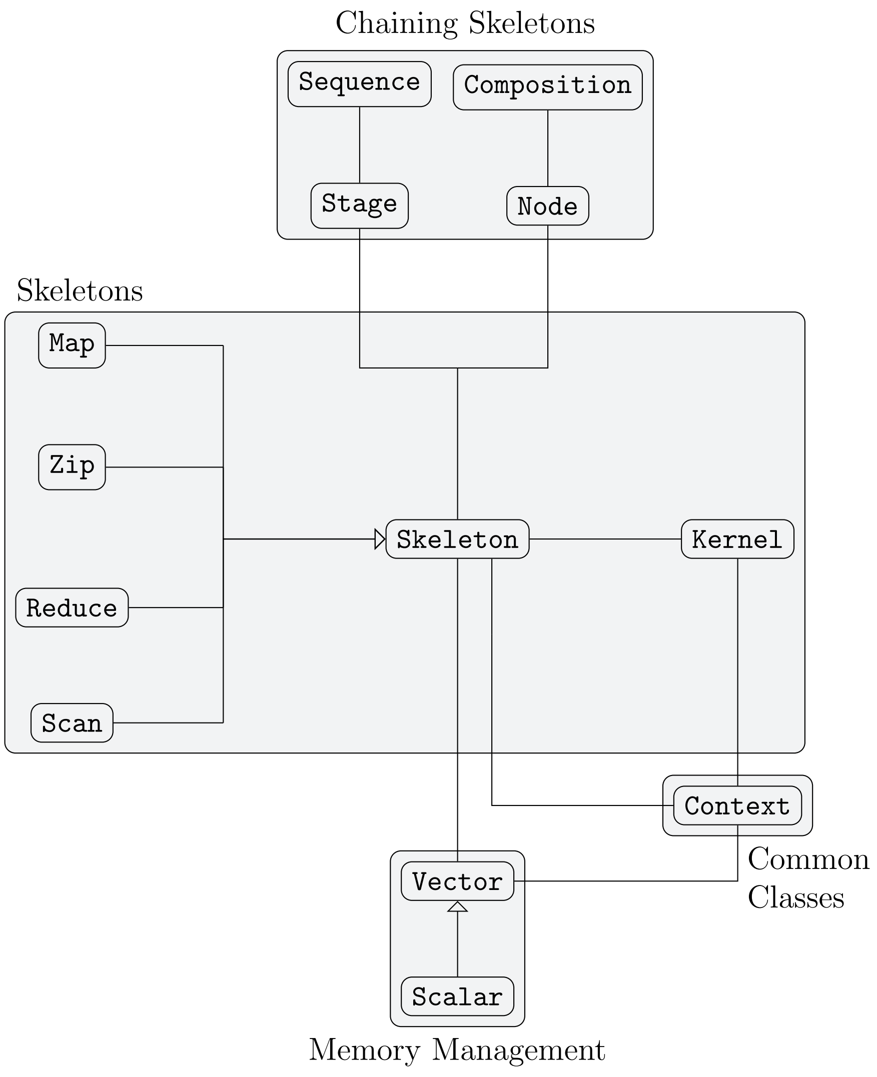


Figure 2.1: Schematic overview of SkelCL. The four different parts of SkelCL are highlighted.

of expressing more complex algorithms, which may not be expressed by simple skeletons. A sequence and a composition are the two supported structures here. Section 2.5 describes the chaining of skeletons.

2.2 Common Classes in SkelCL

The common classes are used by all components of SkelCL since they encapsulate the structures needed by OpenCL. Two classes are noteworthy here: the `Context` class and the `DeviceInfo` class.

2.2.1 The Context Class

OpenCL requires some basic objects to work. These are encapsulated in the `Context` class. With the call of `SkelCL::init`, an instance of this class is created. The user can pass the number of GPUs which should be used by SkelCL. During creation, the basic OpenCL objects are created and initialized.

The `Context` class is implemented as a singleton, so that only a single instance is created. It is the interface to the basic OpenCL structures. Therefore, on initialization, the context instance creates the OpenCL context, as well as a command queue for every OpenCL device (see Section 1.2.1). After the context instance is initialized, OpenCL is set up and ready to execute kernels. The context instance is used by almost all classes in SkelCL, for example, to queue commands for execution. Synchronization functions across multiple devices are also provided by the `Context` class.

2.2.2 The DeviceInfo Class

Applications need information about devices to adopt to a specific GPU. For example, the amount of available memory, or the number of compute units might be required. OpenCL provides a lot of those information. In SkelCL, the `DeviceInfo` class provides access to all information about a certain device as provided by OpenCL. Furthermore, the `DeviceInfo` class caches requested information. This means, that only the first time an information is requested an actual OpenCL call is made to get the information. After that, the cached value is returned. This reduces the time necessary to access information about devices.

2.3 Memory Management

The main goal of SkelCL is to ease GPU-programming. One major challenge in GPU-programming is the memory management. The host and the devices have separate memory. Thus, data needed on the device has to be copied to the device memory explicitly. Because the device can not access the host memory, every data transfer has to be initialized by the host.

To overcome these issues, SkelCL provides a vector class for a unified abstraction of memory.

2.3.1 The Vector Class

The `Vector` class is the unified abstraction for memory used by host and device. Listing 2.1 shows an initialization of a vector by passing a pointer. In this example, the vector is also marked as read-only. As the example shows, the vector is a generic container. Therefore, any primitive C data type, like `float` or `int`, can be stored in a vector as well as user defined structures. User defined structures have to be defined with the keyword `struct`. C++ classes are not supported because OpenCL can not handle them. However, data stored in a C++ object can easily be organized in a C `struct`.

A vector is aware which data is up-to-date. If, for example, a calculation on a device has finished, the data on the device is marked as up-to-date, the one on the host is marked as out-of-date. Before every data transfer, the vector checks whether the data transfer is necessary. Only if the transfer is necessary, the actual data is transferred. This behavior minimizes expensive data transfers between host and device. The avoidance of data transfer benefits especially in situations where multiple calculations are made on the same device, one after the other. Details of how this can be expressed in SkelCL can be found in Section 2.5.

The data transfer is entirely hidden from the user. No data copies have to be initialized explicitly.

The scalar is a special vector with just a single element. It is implemented as a subclass of the `Vector` class. The `Scalar` class offers additional methods,

```
float* input = mallocAndInit(size);
Vector<float> v = Vector<float>(input, size, SCL_READ_ONLY);
```

Listing 2.1: Exemplary use of the `Vector` class.

like a `getValue` method. These are only suitable if a single value is managed.

2.3.2 Behind the Scenes: Implementation of the Vector Class

The `Vector` class provides a unified memory abstraction. This is done by holding a pointer to the host memory and an OpenCL buffer object for the device memory. Memory on the host is associated with memory on a device. An OpenCL buffer object is comparable to a pointer in the host memory. Inside a kernel, the buffer appears as normal pointer to a contiguous memory space.

The vector creates a buffer before the data associated with the host pointer is copied to the device. The buffer is only created once when data is uploaded.

The data transfer between host and device is performed implicitly by SkelCL. To achieve this, SkelCL has to know when a certain vector is required on the device. This is done using the concept of skeletons which are discussed in Section 2.4. Right now, it is only important to know that skeletons describe calculations and that vector objects are passed as inputs and outputs to these skeletons.

When a vector is passed as an input to a skeleton the vector checks whether its data is already on the device. This could be the case if multiple skeletons are executed one after another. If the data is not on the device a data transfer is initialized to copy the data. The vector sets flags to indicate whether the data is present on the device or the host.

The vector also tracks which version of the data is up-to-date. After the execution of a skeleton, the output vector is marked as up-to-date on the device whereas the version on the host is marked as out-of-date. The update of the host data is not performed until it is necessary or requested by the user. Therefore, additional calculations can be performed on the devices using this data without additional data transfers.

Before accessing a vector on the host, a function has to be called to access the data of the vector. The vector checks if the latest version is available on the host. A download is initialized only if the data on the host is not up-to-date.

The `Vector` class is capable of holding data of different types. This is called *parametric type polymorphism.* C++ supports parametric polymorphism with the templates system. SkelCL uses class templates to support different types to be managed by the `Vector` class.

```
1 template <typename T>
2 class wrapper {
3 public:
4   wrapper(const T& pValue) : value(pValue);
5   T value;
6 }
7
8 wrapper<float> w(2.5f);
```

Listing 2.2: A template class definition.

Class templates A *class template* can be used to generalize a specific type which is used inside the class [Str00]. For example, the implementation of a list does not depend on the type of objects which are stored inside the list. Therefore, the C++ language supports writing source code without specifying the exact type used. The class definition is annotated with a special *type parameter* that is used throughout the class definition as replacement for the actual type. Listing 2.2 shows an example of a class definition using class templates. `T` is the type parameter of the class `wrapper` which has a member `value`. When an instance of the class is created, the type which is replaced for `T` has to be specified in angle brackets as shown in line 8. Here the type `float` is specified and replaced in the class definition. Therefore, the constructor expects a reference to a float value and saves the given value in the member `value` which has the type `float`.

2.4 Skeletons in SkelCL

SkelCL provides four basic skeletons: map, zip, reduce, and scan. Each skeleton consumes vectors as input and produces vectors as output. They can be executed on one single device or simultaneously on multiple devices.

First, we discuss some basic concepts which are common to all skeletons. It is described how skeletons are created and launched. Then we show how this is implemented in SkelCL. Next, the four provided skeletons are described in detail. Finally, we discuss how additional arguments can be passed to skeletons.

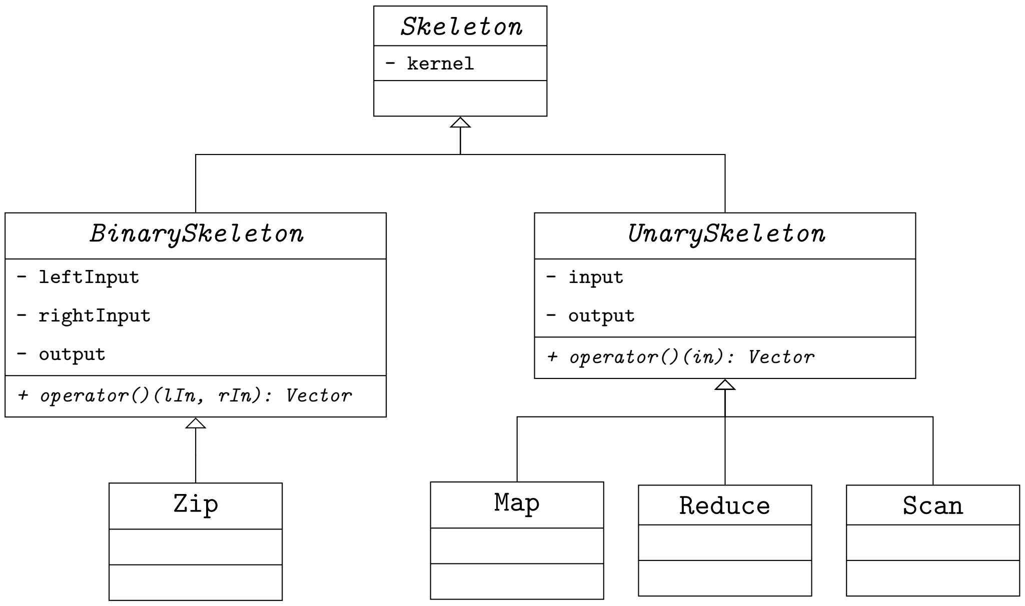


Figure 2.2: Class diagram of all skeleton classes.

2.4.1 Unary and Binary Skeletons

A general distinction of skeletons can be made by the number of vectors which are passed as an input. In SkelCL map, scan and reduce are unary skeletons which take one vector as argument and produce one output vector. The reduce skeleton produces a scalar as output which is a special kind of vector as we saw in Section 2.3.1. The zip is a binary skeleton with two vectors as input. A binary skeleton produces also one output vector. An overview of the skeleton classes in SkelCL and how they are related is shown in Figure 2.2. The `BinarySkeleton` class stores references to a left and right input vector, where the `UnarySkeleton` class only stores a single input. Both classes save a reference to an output vector. This is also reflected in the functions used to execute the skeletons. How skeletons are launched is described in detail in Section 2.4.3.

The main reason to keep binary and unary skeletons apart, is the ability to chain and combine skeletons. For combining skeletons, SkelCL needs to know how many inputs a skeleton takes. We discuss the details of combining skeletons in Section 2.5.

```
bool lessThen(float left, float right) {
  return (left<right);
}

sort(myVector.begin(), myVector.end(), lessThen);
```

Listing 2.3: Exemplary use of a function pointer in C or C++. The variable `myVector` is the vector which contains the data to be sorted.

```
Zip<float> mult("(float x, float y){ return x * y; }");
```

Listing 2.4: A skeleton is instantiated by providing a function definition as string.

2.4.2 Creating Skeletons

Skeletons are higher-order functions [Col89] because they take functions as argument. In C, the technique used to pass functions as arguments are function pointers. A function pointer is a pointer which points to the code of the function. Listing 2.3 shows how a function pointer is passed as a parameter to the sort function provided by the Standard Template Library (STL) [Str00].

A function pointer refers to the address of the function's binary code already compiled for the CPU. Therefore, it is not possible to use a function pointer for kernel functions which should be executed on a GPU. In OpenCL, kernel functions are compiled at runtime specially for the GPU. For that reason, the kernel function has to be accessible as source code. Thus, skeletons in SkelCL expect the function parameter to be a specially formed string. The string is basically the source code for the function to be passed to the skeleton. Listing 2.4 shows a simple example of a skeleton definition in SkelCL.

SkelCL takes the provided function definition and generates, together with additional source code for the skeleton, a valid OpenCL kernel. The combined source code is compiled by OpenCL to a kernel function which is ready for execution. The details of the code generation are described in Section 2.4.4.

As shown in Listing 2.4, the user can omit the return type and the function name. The return type is unnecessary because it is already given by the definition of the skeleton. If a function name is given, it is replaced by SkelCL, so it knows how to call the provided function. Because the function name is replaced during the code generation anyway, it can be omitted entirely.

```
Zip<float> mult("./multiplicationKernel.cl".);
```

Listing 2.5: A skeleton is instantiated by providing a path. The path is relative and points to a file in the same directory with the name “multiplicationKernel.cl”.

If a function definition is several lines long, it is inconvenient to write the definition in a single line. Instead, it is possible to pass a filename of a source file to a skeleton. The content of the given file is then used as the source code. The provided path must be given as string starting with a slash or a period. This string has to specify the path either absolutely or relatively. An example is shown in Listing 2.5.

Function definitions in a separate file are handled differently than those defined inline. The advantages of defining the code in a separate file are:

- `#include` statements can be used to embed code from arbitrary headers
- It is allowed to define and call additional custom functions

Including headers is useful when custom data types are used, like structs which can be defined in a separate header file. The header file can then be included by the host code as well as by the code for the device. A requirement for the function definitions is, that the function which should be called directly by the skeleton have to be named `<f>`. This name is replaced by SkelCL and used to determine the right function to be called. The syntax ensures that no naming conflicts occur.

2.4.3 Launching Skeletons

After creating a skeleton as seen in Section 2.4.2, it may be used in the application. To launch a skeleton, it is required to pass the input data which the skeleton should process. All input or output data processed by skeletons in SkelCL are instances of the provided `Vector` class. Therefore, `Vector` instances have to be created and filled with data before launching a skeleton. However, the skeleton itself can be created up-front without knowing what data the skeleton should process in the future. Listing 2.6 shows a simple fully functional SkelCL program. It calculates the dot product of two vectors. After initializing SkelCL, a reduce and a zip skeleton are created. Even though the skeletons are

```
int main (int argc, char const *argv[]) {
  SkelCL::init(); // Initialize SkelCL.
    // Create zip skeleton with the multiplication
    // as operation.
  Zip<float> mult("(float x, float y){return x*y;}");
    // Reduce skeleton requires the identity
    // of the given operation as second argument.
    // For the addition this is 0.
  Reduce<float> sum("(float x, float y){return x+y;}", "0");

  int size = 1024;
    // Fill pointers a_ptr and b_ptr with random values.
  float* a_ptr = mallocAndInit(size);
  float* b_ptr = mallocAndInit(size);

    // Create vectors A and B.
  Vector<float> A = Vector<float>(a_ptr, size, SCL_READ_ONLY);
  Vector<float> B = Vector<float>(b_ptr, size, SCL_READ_ONLY);

    // Execute the skeletons.
  Scalar<float> C = sum( mult(A, B) );

    // Access the calculated value.
  float c = C.getValue();
}
```

Listing 2.6: Calculation of the dot product of two vectors. The function `mallocAndInit` allocates memory and initializes it with random numbers.

not introduced yet, they can be explained here shortly: the zip skeleton combines two vectors entry-wise whereas the reduce skeleton combines the elements of one vector to a single value. In this case, the two vectors `A` and `B` are first multiplied entry-wise and the result is then reduced to a single value. In the source code, the skeletons are called like normal functions. The return value is accessed by the `getValue` function. This can only be called on a scalar, not on an arbitrary vector.

As we see in the example, a skeleton is launched by using the name of the skeleton as a function and passing the appropriate arguments to it. This behavior is implemented using the C++ operator overloading feature.

As we saw in the first chapter OpenCL requires that launched kernels are organized in work-groups. The work-group size can have a significant impact on the runtime of a kernel. Since SkelCL cannot guess an appropriate work-group size for a specific kernel, a default value for all kernels is used. To override this default value, the user can specify the work-group size by using the `setWorkGroupSize` function.

The example in Listing 2.6 shows that it is possible to chain skeletons. The result of the zip skeleton is used as the input for the reduce skeleton. The syntax to achieve this chaining is simple, the calls of the skeletons are just nested in each other. As already described in the Section 2.3.1, no intermediate results are copied back to the CPU memory. The result of the inner skeleton resides on the devices and are used as the input of the outer skeleton. No data transfers are performed in between at all.

SkelCL provides additional structures to organize the execution of multiple skeletons. These are the sequence and the composition which are described in detail in Section 2.5.

2.4.4 Behind the Scenes: Kernel Generation

As discussed in Section 2.4.2, the user-provided function is passed to a skeleton as a string. SkelCL uses this string to generate the source code for a valid OpenCL kernel. We use the code shown in Listing 2.7 as an easy example. The performed steps are exactly the same for the map and zip skeleton. The scan and reduce skeleton differ a bit. Details are provided in the Sections 2.4.5.

When the code in Listing 2.7 is executed, an instance of the `Map` class is created. Inside the constructor, the kernel is build. Every skeleton has an instance of the kernel class as a member. An object of the kernel class encapsulates an OpenCL kernel. The kernel class also has the capability to combine the user-provided function with the code from the skeleton and build it after-

```
Map<float> m("( float elem ){ return elem * 2; }");
```

Listing 2.7: A simple map skeleton. Every element of the input vector is doubled.

ward. Therefore, the skeleton, in this case the map skeleton, passes several information to its kernel member. These include both the skeleton name as a string and the filename of the file which contains the source code of the skeleton. Furthermore, the names of the types used in the skeleton are determined and passed to the kernel object. How this is done, is described later in this section in an extra paragraph on page 27.

The user-provided function definition is also available to the kernel. This is stored as a copy inside the kernel object. The kernel object continues with combining the kernel function source code. Therefore, the given function definition is parsed to determine whether the definition is specified inline or a filename is provided.

If the function definition is specified inline, it is checked if the return type and function type is specified or not. As already discussed, the return type and the function name can be omitted. SkelCL inserts the placeholders `<Tout>` for the return type and `<f>` for the function name. These placeholders will be replaced later by the actual values. If a function name is provided, it will be replaced here by searching for the opening parenthesis character and replacing the word in front of it. The function name will be `<f>` at this time whether it was replaced or not.

When a filename is provided, the content of the file is loaded. `#include` statements are expanded by SkelCL. Hence, the filename given with the include is used and the file is loaded the same way as the first. As already mentioned in Section 2.4.2 the function which should be called by the skeleton must be named `<f>`.

After this step it is guarantied that exactly one function exists with the name `<f>`. This function will be called by the skeleton.

The source code of the skeleton is loaded exactly the same way as the code of a user-provided function. Every skeleton has an implementation file with the OpenCL code inside. In the example from Listing 2.7, the implementation of the map skeleton is loaded. The OpenCL code for the map skeleton is shown in Listing 2.8b. The function is annotated with a `__kernel` modifier which is required by OpenCL and marks the function as being executed on an OpenCL device. The `__global` modifiers are also required by OpenCL and mark the address spaces in which the input and output data are stored. Due to the organization of work-items in work-groups, more kernel instances can be launched than necessary. Therefore, inside the body of the function, the conditional statement checks that not too many function calls are made. In line 4, the actual user-provided function is called. The right input value is passed as an argument and the result is stored in the output vector at the corresponding location. All placeholders in angle brackets are replaced before the actual code is compiled.

After reading both, the user defined function and the implementation they are combined to one single string. The user defined part is always copied above of the source code provided by SkelCL(see Listing 2.8c). The result of the combination is not valid C code yet and, therefore, not valid OpenCL code either. The placeholders enclosed by angle brackets have to be replaced which happens in the next step.

There are different types of placeholders used in this intermediate form in SkelCL. Two of them can be seen in Listing 2.8c.

Type placeholders start with the letter `T` and have to be replaced by the appropriate type names. As mentioned above the typenames are passed as parameters to the kernel object. The typenames are passed as a vector of strings. Every element in the vector corresponds to a placeholder. The first element in the vector is used as replacement for the placeholder `<T0>` and so on. A special case is the placeholder `<Tout>` which represents the type of the output parameter. This placeholder always correspond to the last value in the vector.

Function placeholders are always named `<f>`. This placeholder is used to guarantee that the right function is called by the skeleton. It is replaced by a fixed name specified inside of SkelCL.

After replacing the placeholders, the source code is ready for compilation. The result is shown in Listing 2.8d. For clarity a simplified source code is shown in the Listings compared to the actual produced one. To avoid naming conflicts, all used variable or function names are prefixed with `SCL` and written entirely in upper case. The Listings here differ only for readability reasons.

The source code is now complete and ready to be built by OpenCL.

Building the source code every time from source is a time consuming task. During our measurements (presented in Chapter 3) the building process took several hundreds of milliseconds. For a small kernel, this can be a huge overhead. Therefore, SkelCL saves already compiled kernels on disk. They can be loaded later if the same kernel is used again. Our test showed that loading

```
Map<float> m("( float elem ){ return elem * 2; }");
```

(a) Simple map skeleton.

```
__kernel void map(const __global <T0>* in,
                  __global <T1>* out,
                  const unsigned int elements) {
  unsigned int id = get_global_id(0);
  if (id < elements)
    out[id] = <f>(in[id]);
}
```

(b) The OpenCL implementation of the map skeleton.

```
<Tout> <f>( float elem ){ return elem * 2; }

__kernel void map(const __global <T0>* in,
                  __global <T1>* out,
                  const unsigned int elements) {
  unsigned int id = get_global_id(0);
  if (id < elements)
    out[id] = <f>(in[id]);
}
```

(c) The result after combining the user-provided function with the skeleton implementation.

```
float f( float elem ){ return elem * 2; }

__kernel void map(const __global float* in,
                  __global float* out,
                  const unsigned int elements) {
  unsigned int id = get_global_id(0);
  if (id < elements)
    out[id] = f(in[id]);
}
```

(d) The final source code of the map skeleton including the user defined function.

Listing 2.8: Kernel generation. The source code of (a) and (b) are combined to (c). After replacing the placeholders the final source code (d) is generated.

the kernels from disk is at least five times faster than building the kernel from source.

It must be guaranteed that only the exact same kernel is loaded from disk as compared to the one to be built. To ensure this, SkelCL uses a cryptographic hash function.

The first time a kernel is going to be built, there is obviously no possibility that the kernel is already built and saved on disk. Therefore, the kernel is built from source by OpenCL. Afterward, a binary representation of the kernel is saved on disk. The filename which is used is the SHA-1 hash of the source code which was compiled. The SHA-1 hash function is a cryptographic hash function which produces the same result only for the same input [FIP08]. If the input varies, the result will change too. When building the same kernel a second time the same SHA-1 hash is calculated. SkelCL checks whether a file with the SHA-1 hash as filename exists. If such a file exists, we can be certain that the file contains a binary representation of the same kernel which is going to be built. Instead of building the kernel from source, we can now take the binary version directly from disk.

After building the kernel from source or loading an already build version from disk, the kernel and thus the skeleton is ready for execution.

How to Get a String Representation of a Type

Skeletons in SkelCL are template classes. This means the type used inside the skeleton is annotated in angle brackets. This allows for generic programming without losing type safety. The annotated type is not only used inside the host code. Instead, the type is also used in the source code for the kernel. Since this source code is built at runtime, the typename must be available as a string. Unfortunately, the user only provided a type, not a string. To overcome this problem, a function is needed which turns a type into a string. SkelCL provides such a function. Listing 2.9 shows a *function template* which returns the typename of a given type as a string. Not only classes but also functions can have template arguments in C++. The shown function can be called like this: `typeName<float>()`. The type is passed as parameter between the angle brackets. Inside the function, the `typeid` keyword of the C++ language is used to determine the given type at runtime [Str00]. Unfortunately, the returned typename is *mangled*. Compilers use the technique of name mangling to ensure that unique names are used during the compilation. To distinguish, for example, two variables with the same name in different namespaces, additional characters are added to the names. Also typenames are changed. For example the compiler

```
template <typename T>
static std::string typeName()
{
  const char* mangledName = typeid(T).name(); // get type name
  int status;
  // unmangle name
  char* name = abi::__cxa_demangle(mangledName,
                                   NULL, NULL, &status);
  ASSERT(status == 0);
  std::string ret(name); free(name);
  return ret;
};
```

Listing 2.9: Function returning the typename as a string for a given type.

gcc 4.2 turns the type `float` into `f` and `unsigned short` into `t`. This process is called name mangling [Lev99]. The mangled typename can not be used inside ordinary source code. First the mangled name has to be *unmangled*. Therefore, the `abi::__cxa_demangle` function is used, provided by the *C++ ABI (application binary interface)* [Cxx01]. It turns `f` back into `float` and `t` into `unsigned short`. This name can be used in source code and is, therefore, useful in SkelCL.

2.4.5 Skeletons in SkelCL

SkelCL provides four basic skeletons: map, zip, reduce and scan. These skeletons are commonly used and implemented in many skeleton libraries [BCGH05, CPK09]. We discuss each of the skeletons in more detail in this sections. We use a unified notation to express the meaning of the skeletons. This is explained first.

The notation for a *function* f that takes an argument of *type* A and returns a result of type B is:

$$f : A \to B$$

An element of type A is written in lower case: a. A *set* or *list* of elements is enclosed in square brackets: $[A]$. If we talk about a special element of the list, we use an index for the position inside the list. a_0 refers to the first element of the list $[A]$, whereas a_i refers to the element on position $i + 1$.

$$f : [A] \to A$$

```
1 Map<float> map("(float elem){return elem * 2.0;}");
2 output = map(input);
```

Listing 2.10: The map skeleton in SkelCL.

This function takes a list of elements of type A and returns a single element of type A.

A function can also be an argument to another function. We use the following syntax for that:

$$f : (A \to B) \to C$$

The argument to the function f is an function of type $A \to B$.

Sometimes it is easier to write a function in *infix notation*. If we do so, we use rounded symbols.

$$A \textcircled{f} B$$

The Map Skeleton

The map skeleton is a very basic skeleton. Equation 2.1 shows a formal definition of the map skeleton.

$$map : (A \to B) \to ([A] \to [B]) \tag{2.1}$$

The map skeleton takes a unary function as argument. We can bind a certain unary function $f : A \to B$ to the map skeleton. This is shown in Equation 2.2.

$$(map\ f) : [A] \to [B] \tag{2.2}$$

The resulting function takes a list of elements as argument and returns a list as result. The type of the elements inside the list is not specified by the map skeleton itself. The provided function f specifies the types. Furthermore the function f describes the calculation which is performed on every element of the list. The map skeleton only ensures that the function f is called for every element of the input list. The number of elements in the result list is exactly the same as in the input list. Therefore, in SkelCL, the input and output vectors must have the same size.

An example of the usage of the map skeleton in SkelCL is shown in Listing 2.10. In line 1 the provided function is bound to the skeleton. The resulting function is then executed in line 2.

```
Zip<float> zip("(float l, float r){return l + r;}");
output = zip(inputLeft, inputRight);
```

Listing 2.11: The zip skeleton in SkelCL.

The implementation of the map skeleton is shown in Section 2.4.4.

The Zip Skeleton

The zip skeleton is a binary skeleton. This means that the function passed to the skeleton takes two arguments instead of one. This is reflected by its formal definition:

$$zip : ((A, B) \to C) \to (([A], [B]) \to [C]) \tag{2.3}$$

A binary function $f : (A, B) \to C$ can be bound to the zip skeleton.

$$(zip\ f) : ([A], [B]) \to [C] \tag{2.4}$$

After binding the provided function, the zip skeleton takes two lists as arguments. The zip skeleton ensures that the given function is called for every pair of inputs (a_i, b_i). The result is stored at the corresponding position in the result list. In SkelCL, the input lists must have the same size as the result list.

The current version of the zip skeleton can only work with vectors of the same type. It is not possible to pass two vectors of different types to a zip skeleton. This could be extended in further versions of SkelCL. The general definitions shown in Equation 2.3 and 2.4 must, therefore, be altered a bit and the function f must have the signature $f : (A, A) \to A$.

$$zip : ((A, A) \to A) \to (([A], [A]) \to [A]) \tag{2.5}$$

$$(zip\ f) : ([A], [A]) \to [A] \tag{2.6}$$

The zip skeleton used in SkelCL is shown in Listing 2.11. The first line shows the creation of the skeleton where the user defined function is bound to the skeleton. The skeleton is executed in the second line.

The OpenCL implementation of the zip skeleton is shown in Listing 2.12. The implementation is straightforward: it calls the function `<f>` with the appropriate arguments and stores the result in the output vector.

```
__kernel void zip(const __global <T0>* lIn,
                  const __global <T0>* rIn,
                  __global <T0>* out,
                  const unsigned int elements){
  unsigned int id = get_global_id(0);
  if (id < elements)
    out[id] = <f>(lIn[id], rIn[id]);
}
```

Listing 2.12: The implementation of the zip skeleton.

The Reduce Skeleton

The definition of the reduce skeleton (also known as fold) is shown in Equation 2.7.

$$reduce : ((A, A) \rightarrow A) \rightarrow ([A] \rightarrow A) \tag{2.7}$$

The reduce skeleton takes a binary function as an argument. A function $f : (A, A) \rightarrow A$ can be bound to the reduce skeleton.

$$(reduce\ f) : [A] \rightarrow A \tag{2.8}$$

The bound skeleton takes a list of elements and returns a single value of the same type. To describe the behavior of the reduce skeleton we use the infix notation. The result value r is produces as shown in Equation 2.9. The input is a list with elements a_i.

$$r = (((a_0 \textcircled{f} a_1) \textcircled{f} a_2) \textcircled{f} \ldots) \tag{2.9}$$

The elements are combined successively by calling the given function on a pair of elements. The intermediate results are also used as arguments to the provided function.

We take the list

$$[1\ 2\ 3\ 4\ 5]$$

for an example and the multiplication as operator. The result would be the product of all elements of the input list: 120.

To execute this algorithm in parallel, the provided function has to be associative. This means the order of applying the function to the list elements is not important for calculating the result. Most operators are associative, like addition, multiplication or even matrix multiplication. SkelCL guarantees that the position of the operands do not change during execution. Therefore, the

```
Reduce<int> r("(int l, int r){ return l * r; }", "1");
output = r(input);
```

Listing 2.13: The reduce skeleton in SkelCL.

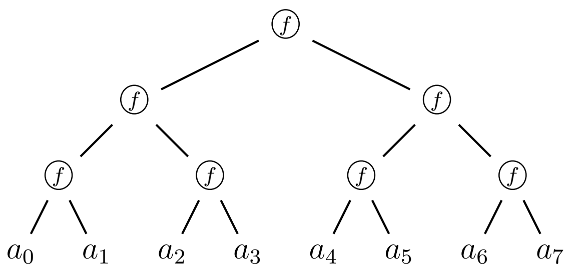


Figure 2.3: The reduce skeleton builds a binary tree. The tree can be processed in parallel.

given function does not have to be commutative.

In contrast to the definition shown in Equation 2.8, in SkelCL it is requires to pass an additional value to the reduce skeleton. This value has to be the identity for the given function. This is for example zero for the addition or one for the multiplication.

How to use the reduce skeleton in SkelCL, is shown in Listing 2.13. In the first line, the function is passed to the skeleton as well as the identity. The second line shows how the reduce skeleton is executed.

The implementation of the reduction used by SkelCL is based on example code provided by Apple [App09]. It was adapted to the environment of SkelCL and enhanced for multiple GPUs as we show in Section 2.6.3. Instead of showing the implementation here, an overview over the used techniques and the algorithms is given.

The basic idea of the parallel reduce algorithm is to build a binary tree which can be processed in parallel. Equation 2.10 shows a reduce calculation for a list with eight elements $a_0, \ldots a_7$.

$$r = a_0 \circled{f} a_1 \circled{f} a_2 \circled{f} a_3 \circled{f} a_4 \circled{f} a_5 \circled{f} a_6 \circled{f} a_7 \tag{2.10}$$

A corresponding tree is shown in Figure 2.3. Because the given operator is associative, the tree and the expression describe exactly the same calculation. Depending on the size of the input vector, different trees are build in SkelCL.

For best performance, the fast local memory is used for the reduce calculation. Unlike other skeletons, the kernel of reduce is not built when the skeleton is created. Instead, the kernel is built dynamically right before the skeleton is launched, depending on the size of the input vector. The size of the input vector dictates how the actual algorithm is executed. Usually, multiple kernels are build. These are launched one after the other. Each kernel launches with multiple work-groups where each work-group processes one subtree of the global tree. The reduction of such a subtree is done entirely in the local memory which is only available inside a work-group. Therefore, the kernel returns as many temporary results as work-groups were launched. After the first kernel is launched, a next kernel is launched which processes the results of the first step as inputs and so on. The last kernel launched launches with only a single work-group producing the final value as result.

The Scan Skeleton

The scan skeleton (also known as prefix sum or prefix reduction) can be defined as shown in Equation 2.11.

$$scan : (((A, A) \to A), id) \to ([A] \to [A]) \tag{2.11}$$

A binary function $f : (A, A) \to A$ and a value id is bound as argument to the scan skeleton. The value is the identity for the given operator.

$$(scan\ (f\ id)) : [A] \to [A] \tag{2.12}$$

The bound scan skeleton takes a list as argument and produces a list of the same size as output. Similar to the notation used for the reduce skeleton, we use an infix notation for the provided function. Applying scan to a vector $[a_0, a_1, \ldots, a_{N-1}]$ produces the following result:

$$[id, a_0, (a_0 \circled{f} a_1), \ldots, (a_0 \circled{f} a_1 \circled{f} \ \ldots \ \circled{f} a_{N-2})] \tag{2.13}$$

This version of the scan algorithm is often called an *exclusive scan* because the element at position j in the output vector is the combination of all input elements up to but not including the input element j.

If we take the list

$$[1\ 2\ 3\ 4\ 5\ 6]$$

```
Scan<int> s("(int l, int r){ return l + r; }", "0");
output = s(input);
```

Listing 2.14: The scan skeleton in SkelCL.

as an easy example and the addition as the binary operator we would get

$$[0\ 1\ 3\ 6\ 10\ 15]$$

as result.

Some of the possible applications of the scan skeleton are stream compaction or a radix sort implementation. Details can be found at Harris et al. [HSO07].

In SkelCL, a reduce skeleton can be created and launched like the other skeletons as shown in Listing 2.14.

The implementation of the reduce algorithm used by SkelCL was developed by Sengupta et al. [SHZO07]. It is highly optimized and makes heavy use of local memory as well as it tries to avoid bank conflicts which can occur when multiple threads access the same memory location. This implementations has been adapted for SkelCL and changed to use multiple GPUs. This is described in Section 2.6.3 in detail.

When creating a scan skeleton, multiple kernels are generated. These different kernels are optimized for different situations. For example, a special kernel is provided if the size of the input vector is a power of two.

The scan algorithm used in SkelCL is divided in two main phases, an up-sweep and a down-sweep phase. The first phase builds a tree like in the reduce skeleton after that the second phase traverse the tree down from the root to the leafs to process the intermediate results. The up-sweep phase is shown in Figure 2.4. As we can see, the algorithm builds the tree not actual in memory. The data stays in an array. After the first phase, intermediate results are spread over the array. These are used in the down-sweep phase to combine the final result. The down-sweep phase is shown in Figure 2.5. By replacing the top value with the identity and then moving this value down the array, the actual results are generated. This algorithm was presented already in 1989 by Blelloch [Ble89].

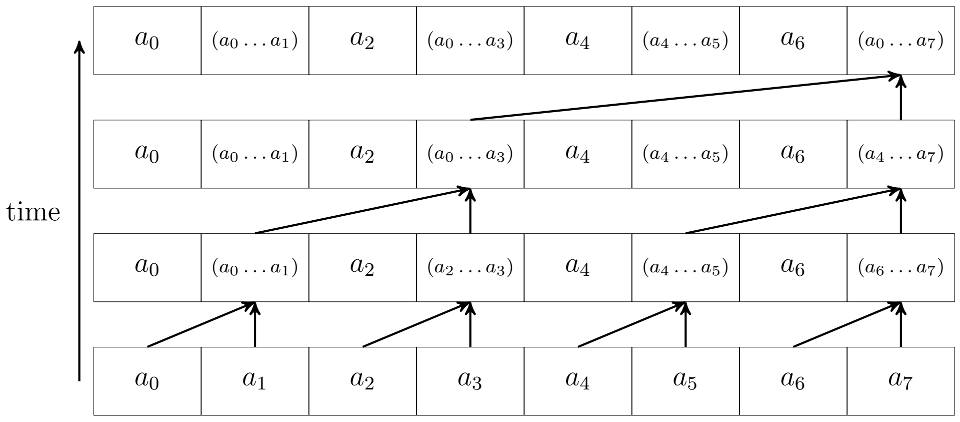


Figure 2.4: The up-sweep phase of the scan algorithm. The paths show the performed calculations. This is an example for a list with eight elements. (Based on [HSO07].)

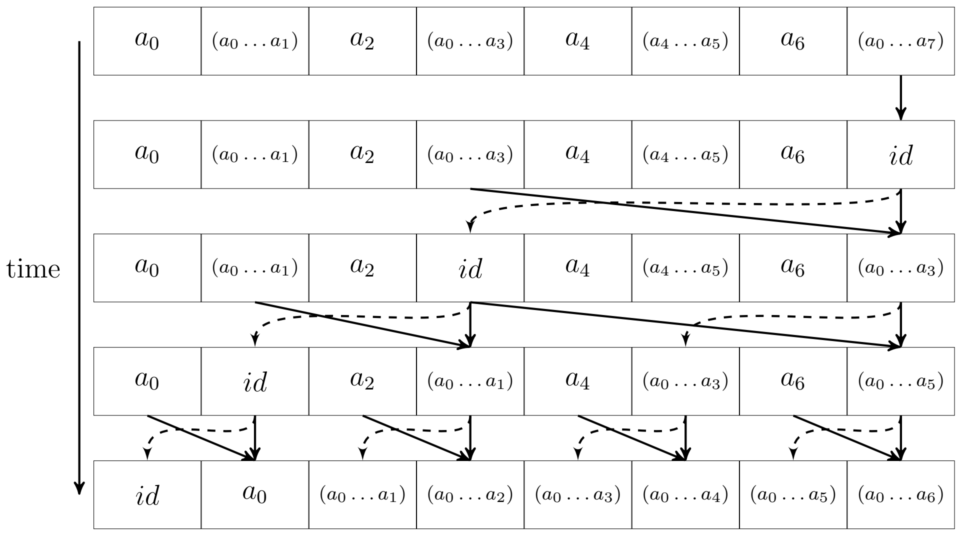


Figure 2.5: The down-sweep phase of the scan algorithm. The solid paths show the actual calculation, the dashed paths show data movements. This is an example for a list with eight elements. (Based on [HSO07].)

```
Map<float> m("float f( float input,
                        __global float* additionalInput,
                        int count ){ ... }");

Arguments arguments;
arguments.push(additionalInput);
arguments.push(sizeof(int), &count);

m(input, arguments);
```

Listing 2.15: Passing additional arguments to a map skeleton.

2.4.6 Additional Arguments

The skeletons described in Section 2.4.5 dictate how many input arguments can be passed to the skeleton. The map skeleton, for example, specifies that the provided function has to be a unary function which processes only one argument. Therefore, only one input argument can be passed to the skeleton. This is a problem, since not all algorithms fit in the strict boundaries of skeletons. We present an example application in Chapter 3 which could not be built without the use of additional arguments.

In Muesli, this problem is solved in an object oriented style [CPK09]. The object, called a functor, which implements the provided function and is passed to the skeleton can hold additional data. Therefore, arbitrary data can be passed to the function as members of the object which actually defines the function. This is not possible in SkelCL. As described in Section 2.4.2 it is not possible to define a kernel with a function pointer or even a C++ object.

SkelCL provides a way to pass arbitrary additional arguments to every skeleton. It takes advantage of the fact, that function definitions are passed as string to SkelCL. It should be noticed that simple constant values can be passed to a skeleton using a simple `#define` statement inside the string defining the provided function. This does not require additional arguments at all.

Additional Arguments and Skeletons

SkelCL can pass arbitrary arguments to the function which is called inside a skeleton. Two steps are required to achieve this. First, the function definition must be changed so that it expects the additional arguments. Second, the additional arguments have to be passed along when the skeleton is launched.

A simple example is presented in Listing 2.15 showing both major steps. The

```
1  float f( float elem ){ return elem * 2; }
2
3  __kernel void map(const __global float* in,
                     __global float* out,
                     const unsigned int elements) {
4    unsigned int id = get_global_id(0);
5    if (id < elements)
6      out[id] = f(in[id]);
7  }
```

Listing 2.16: The combined source code from the map skeleton and the user-provided function.

function definition takes three instead of just one argument which would be normal for a map skeleton. In this example, an additional vector (`additionalInput`) is passed to the skeleton. The vector can be accessed as a normal pointer inside the provided function. The `__global` modifier specifies the address space for the argument. This is only required for pointer types as described in Section 1.2.1. Therefore, the modifier is required when passing a vector as additional argument. The third argument `count` is a simple scalar value. The example shows that all additional arguments are grouped together in an `Arguments` object. The `Arguments` object is then passed to the skeleton which is responsible to pass the arguments to all launched kernel instances.

Arbitrary types can be passed as arguments. It is especially easy to pass vectors as arguments because no additional information have to be provided. When passing in other data types, the size of the type has to be provided.

The order in which arguments are pushed into the `arguments` object is important. It must be the same as the order in the function declaration.

Behind the Scenes: Implementation of Additional Arguments

Building a Valid Kernel As described in the Section 2.4.4, SkelCL combines the source code of the provided function with the implementation of the skeleton. String replacements are made to ensure that the implementation of the skeleton calls the right function. The major challenge to allow for additional arguments is to generate the source code which is required to call the provided function. The Listing 2.16 shows the source code which is generated by SkelCL after combining the user-provided function in line 1 and the implementation of the map skeleton in lines 3–7. SkelCL has carefully ensured that the name of the provided function match with the name called by the skeleton. The argu-

```
__kernel void map(const __global <T0>* in,
                  __global <T1>* out,
                  const unsigned int elements
                  <args>) {
  unsigned int id = get_global_id(0);
  if (id < elements)
    out[id] = <f>(in[id] <apply>);
}
```

Listing 2.17: The implementation of the map skeleton including the capability of passing additional arguments to the function `<f>`.

ments passed to the provided function are exactly arranged by the definition of the skeleton. The implementation of the map skeleton will always try to call the provided function with one argument, no matter how many arguments are needed or defined. If more than one argument is defined by the user the build process will fail.

To overcome this problem, SkelCL parses the provided function definition and extracts a list of the defined parameters. This list is then used to generate a valid function call of the provided function.

To extract the function parameters from the source code, SkelCL looks for the function declaration. A list of parameters is extracted from there. Depending on the skeleton, the first or the first two arguments are ignored, because we are only interested in the additional arguments, not in the obligatory ones. During the process of generating the source code for the kernel, the list of additional arguments is processed and a valid function call is generated.

Two additional placeholders beside the type and function placeholder are used for this purpose. The `<args>` placeholder is replaced with the declaration of additional arguments. That is the combination of the type and the name of a parameter. This placeholder is used in the declaration of the kernel function itself. The second placeholder is the `<apply>` placeholder which is replaced only with the names of the parameters. This is used to generate the actual code for calling the provided function.

The use of both placeholders and, therefore, the changed implementation of the map skeleton is shown in Listing 2.17. If no additional arguments are given, the placeholders `<args>` and `<apply>` are removed, leaving the code correct.

The example from Listing 2.15 with two additional arguments is used to show the generated source code by SkelCL with the additional arguments. This is shown in Listing 2.18. The kernel function has additional arguments as well as the function `f`. The function call of `f` is alter so that the additional arguments

```
float f(float input,
        __global float* additionalInput,
        int count){ ... }

__kernel void map(const __global float* in,
                  __global float* out,
                  const unsigned int elements,
                  __global float* additionalInput,
                  int count) {
  unsigned int id = get_global_id(0);
  if (id < elements)
    out[id] = f(in[id], additionalInput, count);
}
```

Listing 2.18: Generated source code by SkelCL. Two additional arguments are passed to the kernel and then to the function `f`.

of the kernel are passed through.

Launching the Kernel After building a valid kernel, it is required that the actual arguments are passed to the kernel at launch time. Therefore, the skeleton processes the additional arguments passed into it.

A distinction is made between two different kinds of additional arguments: vectors and scalar values. If an additional argument is an instance of the `Vector` class, it is handled exactly as an input vector. It is checked whether the data has to be copied to the device and only if this is the case, the data transfer is performed. The distribution of the vector is respected. An arbitrary other argument is just copied to device as a normal OpenCL argument.

2.5 Chaining Skeletons

It is fairly uncommon that an entire algorithm can be expressed with a single skeleton. Hence, it is important to provide the possibility to execute multiple skeletons. It might be essential to an algorithm that skeletons can coordinate their executions properly.

SkelCL offers two structures to chain and combine skeletons: sequences and compositions.

```
Sequence<int, float> sequence(
  Map<int>          ("(int   elem){return elem * 2;    }"),
  Map<int, short>   ("(int   elem){return elem + 5;    }"),
  Map<short, float>("(short elem){return elem / 50.0f;}"
);

Vector<float> output = sequence.execute(input);
```

Listing 2.19: Creation and execution of a sequence.

2.5.1 The Sequence

Skeletons can be chained in a sequence structure provided by SkelCL. After filling the sequence with skeletons, the sequence as a whole is executed by specifying a vector as input for the sequence. The sequence passes the input for execution to the first skeleton. The result of this calculation is taken as the input for the next skeleton in the sequence and so on. The final result of the sequence is the result of the last skeleton inside the sequence.

Listing 2.19 shows how a sequence is created and executed in SkelCL. The specified sequence processes the vector `input` as follows: Every element is first multiplied by 2, than 5 is added and finally the result of that is divided by 50.

Because the result of each skeleton inside the sequence is used as the input for the next skeleton, every skeleton inside the sequence can only process one input argument. Therefore, only unary skeletons can be used inside the sequence, but no binary ones. This distinction was already made in Section 2.4.1. The unary skeletons in SkelCL are: map, reduce, and scan. The zip skeleton is the only binary skeleton and can, therefore, not be used in a sequence. The use of the reduce skeleton inside a sequence is limited: since the reduce skeleton produces a single value as result it is not reasonable to pass this result as an input to another skeleton even though it is possible in SkelCL.

Listing 2.6 showed, that skeletons can be executed in succession by nesting the skeletons in each other. This is different to building a sequence, because this can only be done when launching skeletons. When nesting skeletons the input data is provided in the same step. Therefore, the combination of the skeletons can not be saved and used a second time. A sequence is build up-front independently of the input data. It can be saved and process different data at different times.

Behind the Scenes: Implementation of a Type Safe Sequence.

Listing 2.19 shows a sequence that is parametrized with the `int` and `float` data type using the template mechanism provided by C++. Thus, the input vector has to be of type `int` and the result vector of type `float`. The skeletons inside the sequence have their own type parameters which match with the ones made for the sequence: The first skeleton expects an `int` vector as input and the final skeleton has a `float` vector as result. The other types of the skeletons, like the type `short`, are not important to the sequence so they do not have to be annotated.

The implementation of this behavior in C++ is difficult. Intermediate results have to be stored. Therefore, temporary vectors have to be created. For these temporary vectors, a type has to be specified in the source code. Since the sequence only specifies the input and output type, the types of the intermediate results are unknown to the sequence and can not be used inside the code. Furthermore, it should be possible to group an arbitrary number of skeletons in a sequence. These problems can not be solved with ordinary C++ techniques.

To overcome these issues, a combination of different methods is used to achieve metaprogramming. The two essential techniques are the preprocessor and the template mechanism.

As already mentioned in Section 2.3.2, templates provide a way to write generic code without knowing the exact types of all participating objects. The types have to be annotated when the class (or function in case of a *function template*) is created (or called respectively).

The preprocessor is the first stage in the compilation process. A special syntax is used to define so-called macros which are replaced by the preprocessor before code is passed to the compiler. The preprocessor just replaces text, so it can build source code. To use the full power of the preprocessor, a library is used by SkelCL for this purpose: the boost preprocessor library [AG05]. It offers a lot of predefined macros which can be used to build source code by the preprocessor.

To offer sequences for different numbers of skeletons, the preprocessor is used to build sequence classes that contain two to ten skeletons. Listing 2.20 shows such a generated class for two skeletons. This class is entirely build with macro calls by the preprocessor. The shown class has two skeletons as members which are called `stage0` and `stage1`. These are instances of the `Stage` class, which act as a proxy for a unary skeleton. When a stage is executed, the encapsulated skeleton is executed and the result of the skeleton is returned. The execute method defines how the two stages are executed after each other and how the

```
template<typename T0, typename T1, typename T2>
class Sequence2 : public AbstractSequence<T0, T2> {
  Stage<T0, T1> stage0;
  Stage<T1, T2> stage1;
public:
  Sequence2(const Stage<T0, T1> &pStage0,
            const Stage<T1, T2> &pStage1)
      : stage0(pStage0), stage1(pStage1) {}

  Vector<T2> execute(Vector<T0>& value0) {
    Vector<T1>& value1 = stage0(value0);
    Vector<T2>& value2 = stage1(value1);
    return value2;
  }
};
```

Listing 2.20: Generated sequence class for two skeletons.

result of `stage0` is passed as input to `stage1`. Important to notice is that three types are provided for the template class: `T0`, `T1` and `T2`. These types are all required since they guarantee that only skeletons with fitting types can be used inside the sequence. This is called *type safety*. The code dictates that the result type of `stage0` has to be of type `T1`, exactly the same type as used as input of `stage1`.

Thus, the specified types are indispensable for the type safety of the sequence. Nevertheless, if all intermediate types have to be specified every time, it is very inconvenient. Furthermore, as we see in Listing 2.19, all types are already specified by the skeletons. In addition, all classes built by the preprocessor must have different names (`Sequence2`, ...) for the number of skeletons inside the sequence.

For convenience handling, a wrapper class is used. The `Sequence` class only requires two template arguments, the type of the input vector, and the type of the result vector. It references to one of the classes built by the preprocessor, for example the `Sequence2` class. By choosing different constructors, different classes are created. The constructors are build using function templates and preprocessor macros. Therefore, different constructors exist depending on the number of skeletons passed as arguments. The template mechanism finally ensures that the right types are replaced for the type parameters at compile time.

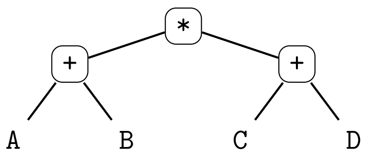


Figure 2.6: Tree representation for the expression `(A+B)*(C+D)`.

```
Zip<float> add("(float l, float r){ return l + r; }");
Zip<float> mult("(float l, float r){ return l * r; }");

Composition<float> comp(mult);
comp.insertLeft(add);
comp.insertRight(add);

Vector<float> E = comp.execute(A, B, C, D);
```

Listing 2.21: Source code for building a composition for the expression `(A+B)*(C+D)`.

2.5.2 The Composition

The most powerful way to organize the execution of skeletons in SkelCL is the composition structure. A composition is a good way to express dependencies between calculations. In a composition skeletons are composed in a tree-like manner.

Trees are also a very common concept in computer science. The composition in SkelCL uses a binary tree to organize the execution of skeletons.

To express the calculation `(A+B)*(C+D)` in SkelCL, a composition can be built with the source code shown in Listing 2.21. A tree representation for this expression is shown in Figure 2.6.

First the skeletons are defined. Then the composition is created and build. The multiplication skeleton is used to create the composition and the additions are added as the left and right child of it. This builds a composition with a corresponding tree like the one shown in Figure 2.6.

The built composition can be executed by passing one vector per leaf as input. In the given example, four leafs are present in the corresponding tree, so four vectors are required as inputs. The inputs are assigned from left to right to the leafs of the corresponding tree.

Unary and binary skeletons can be used to build a composition. Unlike the sequence, the composition has no restrictions regarding the type of the skeletons

```
template <typename T>
Composition<T> operator+(Vector<T>& lhs, Vector<T>& rhs);
```

Listing 2.24: Declaration of the + operator for two vector instances.

it uses.

Instead of writing the source code for the kernel in Listing 2.21, a predefined definition can be used like shown in Listing 2.22.

```
Zip<float> add(Predefined::ADD);
```

Listing 2.22: A skeleton is instantiated by providing a predefined kernel definition.

SkelCL currently offers five predefined definitions for skeletons. These are shown in Table 2.1. Of course, the predefined source code can be used as a shortcut anywhere, not only in the composition structure.

SkelCL uses the C++ operator overloading feature to provide the ability to build compositions with the four basic operations: `+`, `-`, `*` and `/`. Instead of building the composition and executing it as shown in Listings 2.21, the user can simply write the expression shown in Listing 2.23.

```
Vector<float> E = (A + B) * (C + D);
```

Listing 2.23: Source code for executing the composition built for the expression `(A+B)*(C+D)`.

Behind the Scenes: Operator Overloading

Many programming languages offer the ability to define custom behavior for operators. Operators are predefined symbols usually used in an infix notation. Common operators are `+` or `*`, for example.

C++ offers the feature of operator overloading which allows for custom classes using common operators. Table 2.1 shows the operators overloaded in SkelCL. To overload an operator for custom classes, a special function has to be provided. The two arguments of the function can be seen as the operands of the operator. The declaration of a function defining the `+` operator is shown in Listing 2.24. With the shown function it is possible to use the `+` operator between two vector instances. It is also possible to overload an operator for

Operator	Predefined Kernel Source Code
`-A`	`Predefined::NEG`
`A + B`	`Predefined::ADD`
`A - B`	`Predefined::SUB`
`A * B`	`Predefined::MULT`
`A / B`	`Predefined::DIV`

Table 2.1: Operators overloaded in SkelCL an the corresponding predefined source code.

operands with two different types. This is used in SkelCL to support nested expressions.

To build a composition with the basic operators, the `Vector` class overloads all of them. If two vectors are added up with the operator `+`, a composition object is created with a binary node. The binary node stores a zip skeleton with the addition as provided operation. The operator returns the created composition object. This composition can be combined with additional vectors or composition objects to more complex compositions. To accomplish this behavior, the basic operators are overloaded. SkelCL defines operators for two vectors as well as for of a vector and a composition or two compositions. If a composition is assigned to a vector the composition is executed and the result is stored inside the vector. While building the composition this way, the vectors used in the expression are used as the corresponding inputs right away. This is done differently when building a custom composition without the operators and described in the next section.

Behind the Scenes: Implementation of the Composition.

The composition structure in SkelCL is built of several classes shown in Figure 2.7. The composition class stores all nodes of the corresponding tree and holds a special reference to the root node of the corresponding tree. Every node inside the composition is connected to its parent node and up to two child nodes. This makes the corresponding tree for the composition a binary tree. The `Node` class is abstract. Actual instances of nodes are either of the `UnaryNode` or `BinaryNode` class. An unary node stores a `UnarySkeleton` where a binary node stores an `BinarySkeleton` respectively. If a child node is missing for a given node the corresponding pointer is set to `null`.

When building a composition like shown in Listing 2.21, the node objects

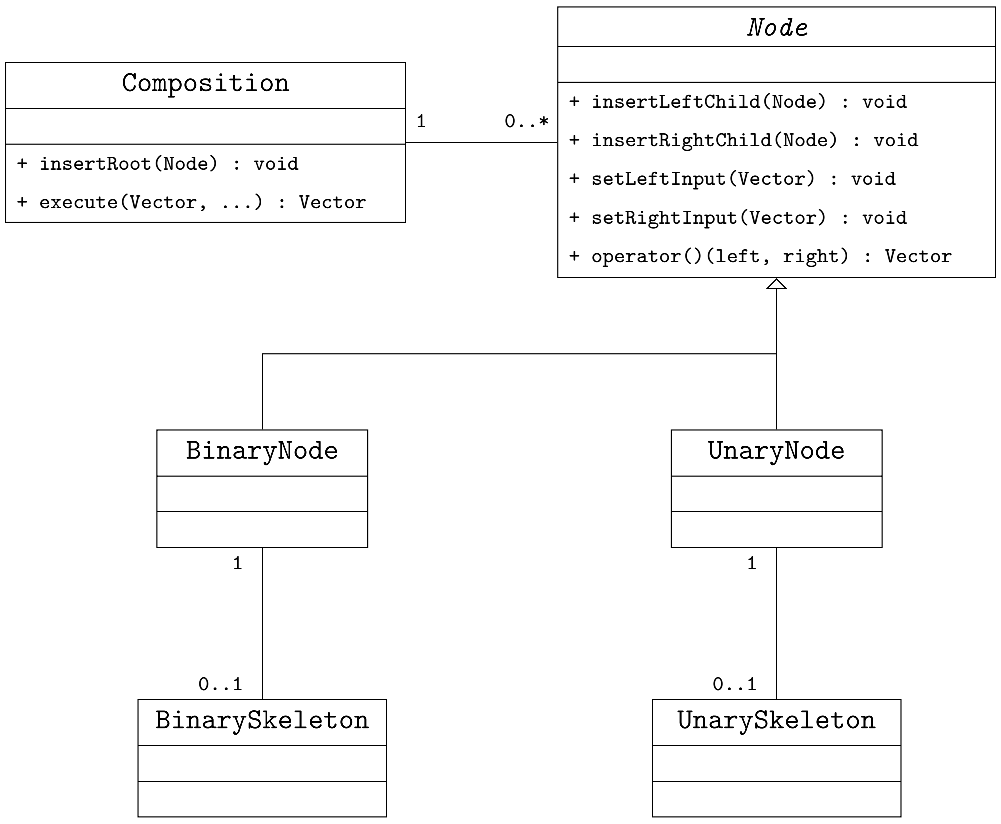

Node
+ insertLeftChild(Node) : void
+ insertRightChild(Node) : void
+ setLeftInput(Vector) : void
+ setRightInput(Vector) : void
+ operator()(left, right) : Vector
Composition
+ insertRoot(Node) : void
+ execute(Vector, ...) : Vector
1
0..*
BinaryNode
UnaryNode
1
0..1
1
0..1
BinarySkeleton
UnarySkeleton

Figure 2.7: Class diagram of all classes related to the composition.

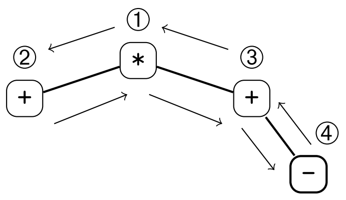


Figure 2.8: Visualization of a depth-first tree traversal. The numbers and arrows indicate the order in which the nodes are visited.

are created and inserted into the composition by making the right connections. Nodes are inserted as children of other nodes, so they get connected to their parents and vice versa.

Assigning input vectors to leafs During execution, the input vectors are distributed to the corresponding leafs before the composition is executed.

The composition class provides many overloaded versions of the execute function. These versions differ in the number of vectors which can be passed as arguments. The boost preprocessor Liberia, described in Section 2.5.1, is used to build these versions at compile time.

To execute a composition properly, one input vector per leaf of the corresponding tree has to be provided. If this is not the case, the execute function fails at runtime. If the correct version is called, the composition is traversed to distribute the vectors to the leafs. Therefore, a *depth-first tree traversal* is performed. This means every node in the corresponding tree is visited after the following scheme: starting at the root node, first the left subtree is visited. After the left subtree returns, the right subtree is visited. The process is visualized in Figure 2.8. This ensures that the leafs of the corresponding tree are visited from left to right. Every node checks whether it has nodes as left and right children. If this it not the case, a reference to the input is stored in that node and so on.

This step is separated from the execution because this step is optional. If the composition is built using the operators, the input vectors are assigned directly to the leafs as the composition is built. In that case, this step is skipped and the execution is performed right away.

Executing the Composition After an input vector has been assigned to every leaf of the composition, the composition is executed.

The execution is performed similarly to the distribution of the inputs: a depth-first tree traversal is performed on the corresponding tree. Therefore, for every node, first the left subtree is executed, then the right subtree. Both results are temporarily stored and used as input to the skeleton associated with the node. This result is then passed to the parent node. For unary nodes, only one subtree to wait for exists, of course. The result of the root node is then returned by the execute function.

2.6 Multi-GPU

In this section, we discuss how all these features work when SkelCL exploits a multi-GPU environment. Before we go into the details, the major challenges of multi-GPU computing are described. Then we discuss how multiple GPUs can be used with SkelCL and finally how this is implemented.

2.6.1 Challenges of Multi-GPU Computing

In Chapter 1, we discussed the major challenges of GPU computing. In a multi-GPU environment, additional challenges occur. OpenCL offers no extra support to program multiple devices. Basically, every GPU is available to OpenCL as a separate device. Communication between these devices has to be done by hand. Because no direct data transfer between two OpenCL devices is possible, communication between devices has always to be implemented across the host [SEN10]. Data to be exchanged between two devices first has to be downloaded to the host before it can be uploaded to other device. The host application must coordinate and perform synchronization and data exchange explicitly. The source code for performing those exchanges contains a lot of boilerplate code. Furthermore, the code is error-prone because many low level details have to be taken in account like calculating offsets and length or the handling with pointers. To overcome these problems, SkelCL's key abstractions – the skeletons and the memory management – can handle multiple GPUs. In the following section, we discuss how this helps to ease multi-GPU computing.

Time-consuming application can often benefit from the use of multiple GPUs, even with multiple synchronizations during a calculation [Sch09].

2.6.2 Multi-GPU Memory Management

As described in Section 2.3, SkelCL provides a unified memory abstraction with the `Vector` class. The vector connects memory on the host with memory on the device and implicitly copies data between them, if needed. To support multi-GPU computing, the vector is, furthermore, capable of managing memory on multiple devices. To control on which devices the data is actually stored SkelCL offers the concept of *distribution*. A distribution describes how the data is distributed among the available devices. Every vector is set to a specified distribution which describes how the data is stored on multiple devices. Four different distributions exists in SkelCL so far: *none*, *single*, *block* and *copy*.

After creation a vector is set to none distribution. As soon as the vector is used by a skeleton as input or output a real distribution is set. If a vector, set to none distribution, is used as an output vector it adopts the distribution from the input vector. This is the main motivation for this distribution. Once a vector has been changed from none distribution to a real one, it can not be changed back to the none distribution.

If a vector is set to the single distribution, only one device is used to store the whole data. The vector acts as if only one device were available. The user can specify on which device the data should be stored.

A vector set to block distribution is distributed evenly among over all available devices. Every device gets a continuous part of the whole data.

The copy distribution copies the data of the vector to every device available. The entire data is copied to every device.

The distribution can be changed during the runtime of an application. This is an important principle since data exchange between multiple device can be done by changing the distribution. The data transfers necessary to complete the change from one to another distribution is done implicitly. We see how this can be used in a real world example in Chapter 3 and how it is implemented in the next section.

When choosing a new distribution, in some cases, additional information have to be provided for SkelCL. If the new distribution is the single distribution, a device number has to be provided to specify the device on which the data should be copied. If no number is given, the first device with the number zero is taken. The second case where additional information have to be provided is when the distribution is changed from a copy distribution. Since in the copy distribution every device has its own copy of the data, different versions can occur if data is modified on the devices. These different versions must be combined to a new version. Therefore, a function has to be provided which is used to combine these different version. If no function is provided the data from the first device is taken as the new version and the data on the other devices gets lost.

Behind the Scenes: Implementation of the Distributed Vector Class

In a single-GPU environment we already saw the need for a memory abstraction. Distinct address spaces between the host memory and the device memory are hidden. In a multi-GPU environment, the address spaces of the different devices are distinguished too. Therefore, the `Vector` class is able to handle this additional address spaces as well.

The vector holds one OpenCL buffer object per device. In addition, for every device, the size of the data attached to the device is stored.

When the single distribution is selected, only one buffer is created and the size for this device is set to the size of the whole data. The sizes for the other devices are set to zero.

The block distribution splits the data into evenly sized chunks which are distributed over the available devices. If the size of the vector cannot be divided into perfectly even chunks the last chunk is increased by the remainder. The data is split canonically: the first chunk is sent to the first device and so on. All necessary data transfers are now performed for all devices. Appropriate offsets are used to ensure that the correct data is sent to the devices and that the vector is put together correctly when data is downloaded from the devices.

Finally, the copy distribution is like a mix of the single and the block distribution. The data is not divided like in the single distribution, but all devices hold a copy of the data like in the block distribution. Therefore, one buffer per device is created and the size per device is set to the size of the whole data. Of course, all data transfers have to be performed for all devices. As described earlier, a special problem occurs if the data on the devices were modified and should now be downloaded to the host. Every device has a different version of the data. A function has to be provided to specify how this versions are combined to a new one. The function first processes the versions of the first and the second device. Therefore, it is applied for every pair (a_i, b_i), where a_i and b_i are the elements at position i from the version of device one and two respectively. Afterward, the function is applied to the calculated intermediate result and the version of device three and so on. In this way, a new version of the vector is calculated which is then used as the data of the vector.

Changing the Distribution One of the major advantages of the distribution feature is the ability to change it at runtime. The advantage is because data transfers are performed automatically. The user just changes the model how the data is distributed but the data transfers are done in the background.

Not every possible combination of changing from one type of distribution to another leads to data transfers. Different combinations are handled differently.

If the distribution is not changing at all, that is the old and new distribution are the same, nothing has to be done. Only when changing from the single distribution to the single distribution and switching the device at the same time a data transfer is performed. This data transfer moves the data from one device to the other.

If the distribution is changing from the copy distribution, the intermediate step of combining the data is performed. This step is not necessary when changing from another distribution.

Beside changing of the distribution a synchronization can be performed if the distribution is set to copy. During this synchronization the different versions of the vector are combined and uploaded to the devices afterward. The distribution is not changed here, but data exchange is performed.

It should be noticed, that the final step of copying data to the devices is not performed directly. This is done lazily such that the data is copied only when the data is actually needed on the device.

2.6.3 Multi-GPU Skeletons

This section describes the multi-GPU capabilities of all four skeletons in SkelCL and how they are implemented.

The Multi-GPU Map Skeleton

The multi-GPU capability of the map skeleton is straightforward. The map skeleton can operate on vectors, no matter what distribution is selected. If no distribution is selected at all, the block distribution is taken as default. If the input data is distributed among different devices, the kernel is launched on all of those devices. The output vector is set to the same distribution as the input vector. So a map on an input vector with block distribution leads to an output vector with block distribution and so on.

The Multi-GPU Zip Skeleton

As the map skeleton, the zip skeleton can operate on vectors with all distributions. An important requirement for the actual calculation is that both input vectors must have the same distribution. Furthermore, if the single distribution is chosen, the vectors must also be stored on the same device. The zip skeleton itself changes the distribution of the input vectors if this requirement is not satisfied. The following rules apply in this case: it is first checked if one of the vectors has no distribution. If this is the case, this vector takes the distribution of the other vector. If both vectors have a distribution set, but they differ, both are set to the block distribution. This is also the default behavior if both vectors have set no distribution at all. The kernel launch is performed for multiple devices if the distribution of the input vectors is not the single distribution. As with the map skeleton, the output vector takes the same distribution as the input vectors.

The Multi-GPU Reduce Skeleton

If the input vector is set to single distribution or copy distribution, the single-GPU algorithm described in Section 2.4.5 is used. The more interesting case is if the input vectors is set to the block distribution.

Since the reduce algorithm in SkelCL is using a tree, the calculation of subtrees can be done independently on different devices. Therefore, every device executes the reduce algorithm for a part of the whole data. In a second step, these intermediate results are reduced further by a single device. Here data transfer between multiple devices and the host is required.

The last step is normally small, because only a few intermediate results have to be reduced further. A reduction on the host site would reduce the time spend for transferring data. This is not possible in SkelCL, since the skeleton is only available to OpenCL devices not the CPU. On some systems the CPU itself may be an OpenCL device, but SkelCL can not rely on this. Therefore, the final step is also executed on one device.

The output vector is a scalar and can, therefore, not be distributed meaningfully among multiple devices. Hence, the output vector of the reduce skeleton is always set to a single distribution.

The Multi-GPU Scan Skeleton

So far, modifying a skeleton for multi-GPU was uncomplicated. To make the scan skeleton multi-GPU capable, a more complex approach is required.

The distribution of the input vector dictates which variant of the scan algorithm is used. An input vector with the single or copy distribution is handled like described in the single-GPU case in Section 2.4.5. Again, if the input vector is distributed with the block distribution, the algorithm has been modified.

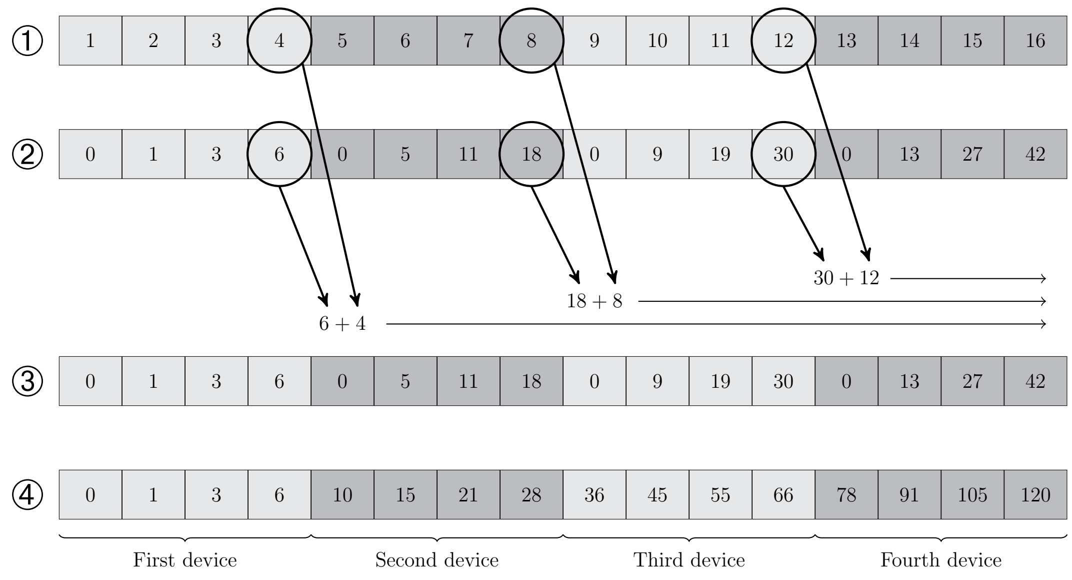


Figure 2.9: An example of the multi-GPU scan algorithm used in SkelCL.

Figure 2.9 shows a example of the multi-GPU algorithm for four devices and a vector of length 16. The operation used in the example is the addition.
① shows the initial input vector with the values from 1 up to 16. This vector is divided evenly among all four devices which are highlighted in different gray shades. Every device performs the normal scan algorithm for its chunk of data. The result of this is shown in ②. Because the implemented algorithm is an exclusive scan, as discussed in Section 2.4.5, the first value in every chunk is the identity. In the case of the addition, this is 0. To complete the scan on the entire vector, the marked values in ② are downloaded to the host and used together with the marked values from the original input vector ① to build map skeletons. These map skeletons are then executed on all devices but the first one. This is shown in ③. The values 4 and 6 have to be added to all values starting at the first value of the second device. The values 8 and 18 are added to the values on device three and four and the values 12 and 30 are only added to the values on device four. Therefore, for every device a different map skeleton is built with the appropriate values. The final result is shown in ④. The output vector is – like the input vector – distributed over all devices, with the block distribution.

Chapter 3

Applications

In this chapter, two applications implemented with SkelCL are presented.

The first application is the calculation of the Mandelbrot set. This example is a pure synthetic benchmark to evaluate the performance possible with SkelCL.

The second example is a real work application from the field of medical imaging. It is used in medical image reconstruction to calculate an image out of data provided by a scanner. This algorithm is used in clinics day-to-day.

3.1 Mandelbrot Set

The calculation of the Mandelbrot set is a time-consuming task. An effective parallel implementation can be written easily so it is a good benchmark to compare the raw performance of different architectures or computing models.

The Mandelbrot set is a set of points in the complex numbers plane. If drawn into an image, the boundary of the Mandelbrot set forms a fractal. It was discovered by the mathematician Benoît Mandelbrot in the early 1980s [Man80].

The Mandelbrot set are all complex numbers c for which the sequence

$$z_{i+1} = z_i^2 + c \tag{3.1}$$

with $i \in \mathbb{N}$, starting with $z_0 = 0$ does not escape to infinity. If the sequences does escape to infinity, the complex number c is not inside the Mandelbrot set.

When calculating an image of the Mandelbrot set with a computer, the given sequence is calculated for every pixel. If a threshold is crossed, it is presumed that the sequence will escape to infinity and that the pixel is not inside the Mandelbrot set. If the threshold is not crossed for a given number of steps in the sequence, the pixel is taken as a member of the Mandelbrot set.

An area of an image of the Mandelbrot set is shown in Figure 3.1. The black parts at the lower left and right are inside the Mandelbrot set where the gray

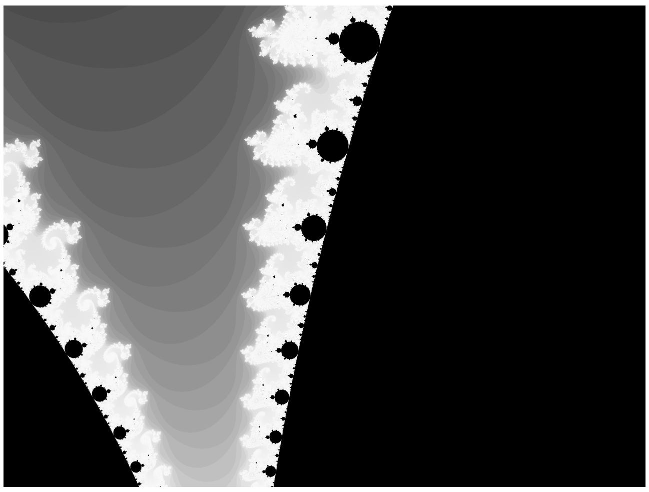


Figure 3.1: An area of the mandelbrot set.

shaded part is outside. The actual shading denotes how many steps of the sequence from Equation 3.1 where calculated before the threshold was crossed.

3.1.1 Implementation with SkelCL

To calculate the Mandelbrot set, the color of every pixel in the output image has to be determined. A map skeletons fits well to this requirement, since it executes a given function for every element of the input vector. A vector with the positions of the individual pixels is used as input in this case. The output is a vector of pixels describing the output image.

The positions and pixels are provided as C structs. The position data type is a C struct storing two values with the names x and y, whereas the pixel struct stores three values named: red, green, and blue.

These data structures are used on both sides, the host side and the device side as well. As described in Chapter 2, SkelCL supports including header files in kernel function definitions. Hence, a common header was made with the definition of these data types which is included in the kernel function and in the host code.

The kernel function is straightforward. It reads the position passed as an input, performs the iterative calculation for this position and then returns a pixel struct with the right coloring set.

3.1.2 Comparison to Other Implementations

Three different versions are compared: an OpenCL, a CUDA and a SkelCL version. The same parallelization strategy is used in all three versions. Therefore, the kernel function is similar in all versions. The host code is quite different across the versions. The most notable difference is that SkelCL uses a skeleton, whereas CUDA and OpenCL are using plain kernels.

The OpenCL version is, by far, the longest version in terms of lines of code. The setup process of OpenCL requires a lengthy creation and initialization of different data structures. This is a lot shorter in the CUDA version as well as in the SkelCL version. In both versions, only one line is required to initialize the whole system.

The kernel function is launched differently in CUDA and OpenCL. In CUDA, the kernel is called like a ordinary function. Only the number of threads to execute this kernel has to be specified. In contrast, OpenCL requires a function call for each argument of a kernel. After passing all arguments, the kernel can be launched by specifying the number of kernels that should be launched. The way to launch kernels in OpenCL is, therefore, inconvenient. Since no kernel functions are launched directly in SkelCL, the inconvenience of OpenCL is hidden. Launching skeletons is as convenient as calling a function. Though, additional arguments must be passed to an arguments object which is then passed to the skeleton (see Section 2.4.6).

The three different versions of launching a calculation are shown in the Figures 3.1.

This example does not require to upload data to the device before the calculation is started. Only the results have to be copied back to the host. In the OpenCL and CUDA versions, this has to be done manually. In SkelCL, a method on the vector has to be invoked to access the data. Even though it is not required by the algorithm, SkelCL does upload data to the device. This eases the modeling of the problem with skeletons, since an input vector is required by the map skeleton. This vector is used to pass indices to the skeleton, instead of calculating them on the device. Therefore, the information which kernel is responsible for which pixel is passed explicitly to the device. In the OpenCL and CUDA versions, this information is passed implicitly to the device. When launching many kernels, every kernel instance is provided an ID that identifies the instance in the set of launched kernels. This information is used to determine which pixel is calculated by which kernel.

Most notable in this example is that the SkelCL version is able to use multi-GPU systems by design. To achieve this with the other versions additional effort

```
dim3 threads(16, 16);
dim3 blocks(width / 16, height / 16);

mandelbrotKernel<<<blocks, threads>>>(output, startX, startY,
                                      dx, dy, width, height);
```

(a) CUDA version. The arguments in the angle brackets specify how many threads are launched.

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *) &output);
clSetKernelArg(kernel, 1, sizeof(float),  (void *) &startX);
clSetKernelArg(kernel, 2, sizeof(float),  (void *) &startY);
clSetKernelArg(kernel, 3, sizeof(float),  (void *) &dx    );
clSetKernelArg(kernel, 4, sizeof(float),  (void *) &dy    );
clSetKernelArg(kernel, 5, sizeof(int),    (void *) &width );
clSetKernelArg(kernel, 6, sizeof(int),    (void *) &height);

size_t  localWorkSize[] = {16, 16};
size_t globalWorkSize[] = {width, height};

clEnqueueNDRangeKernel(commandQueue, mandelbrotKernel, 2, NULL,
                       globalWorkSize, localWorkSize,
                       0, NULL, NULL);
```

(b) OpenCL version. Every argument is passed one at a time with a function call. It is specified how many kernel functions are actually launched, which is similar to the CUDA implementation.

```
Arguments arguments;
arguments.push(sizeof(float), &startx);
arguments.push(sizeof(float), &starty);
arguments.push(sizeof(float), &dx    );
arguments.push(sizeof(float), &dy    );

mandelbrotMap(input, output, arguments);
```

(c) SkelCL version. The additional arguments are passed as a single one. The height and width argument passed in the other versions can be omitted since it is used there to do boundary checks and offset calculations which are not necessary in this version.

Listing 3.1: Launching the same kernel, or skeleton respectively, in CUDA, OpenCL, and SkelCL.

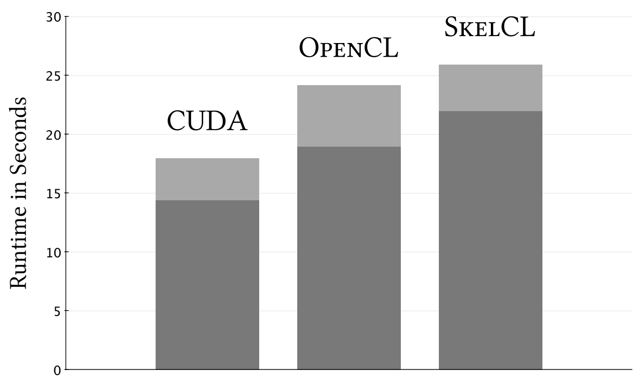


Figure 3.2: Runtime results for the Mandelbrot set. The dark part is the computation time spent on the GPU. The lighter part is executed on the CPU.

has to be done. Even though, the SkelCL version is not much shorter than the CUDA version, this advantage is remarkable. The OpenCL version is in total more than twice as long as the CUDA and SkelCL versions in terms of lines of code. In addition, OpenCL and CUDA are written in C whereas SkelCL is written in C++. Therefore, SkelCL offers a more convenient interface for most C++ programmers.

3.1.3 Results

In the last section, we compared the implementations in terms of programming style and effort. In this section, we focus on the performance of the different versions.

Figure 3.2 shows an comparison of the three different versions. We used a Tesla T10 GPU for these tests. The fastest implementation is the CUDA-based, the calculation took about 18 seconds. The OpenCL implementations took less than 25 seconds and SkelCL roughly under 26 seconds. Since SkelCL is build on top of OpenCL, the difference between these two versions can be regarded as the overhead introduced by SkelCL. This is only an overhead of 4.6%. The additional abstractions and the enormous reduction of code in comparison to the OpenCL implementation justify this small amount of extra time. In comparison to the CUDA implementation, the OpenCL implementation is 38.0% slower and the SkelCL version 44.4%. Kong et al. [KDY$^+$10] also compared the runtime of CUDA and OpenCL and made similar observations. They present measurements where the runtimes of CUDA and OpenCL vary, though CUDA is always faster than OpenCL. The runtimes vary between small differences up to one case where CUDA is seven times faster than OpenCL.

Figure 3.2 shows that the CPU time (shown as lighter parts) of the OpenCL implementation is worse than the SkelCL version. This can be explained by the fact that in the OpenCL version the kernel is built from source every time, whereas it is loaded from disk in the SkelCL version.

SkelCL offers the opportunity to adjust the size of the work-groups which are launched during a skeleton is executed. This information has always to be provided in CUDA and OpenCL while it is optional in SkelCL. The results above are achieved with the default value (work-group size of 256) for SkelCL and work-groups of size 16 times 16 for the CUDA and OpenCL measurements. Since the work-group size can have a huge impact on the performance, it is reasonable to try to optimize this value. This has been done for the Mandelbrot set example. The runtime using various work-group sizes is shown in Figure 3.3. Only multiples of 32 are chosen since this fits well in the graphics architecture of the NVIDIA graphics card [NVI10]. In SkelCL, the work-group size is always one-dimensional while it can be up to three-dimensional in OpenCL and CUDA. In this example, the work-groups are aligned two-dimensionally in the OpenCL and CUDA versions. This fits well with the two-dimensional image which is calculated. On the other hand, this restricts the number of valid possibilities for the work-group size. Furthermore, in this example, the image size must be evenly dividable by the work-group size. The image size is fix with a width of 4096 and 3072 in height. The x-axis in Figure 3.3 is, therefore, annotated with two scales. The numbers indicate what work-group size is chosen for SkelCL and the pairs, like 8 x 8, are the sizes chosen for OpenCL and CUDA. For SkelCL the slowest runtime (with a work-group size of 32) is 43.9% slower than the fastest with a work-group size of 160. The fastest version is clearly faster than the best OpenCL implementation and only about 17% slower than the fastest CUDA version. Notably, only the SkelCL version can gain a significant speedup from changing the work-group size. This might be a side effect from the used work-group sizes. Where the CUDA and OpenCL versions use two-dimensional work-groups SkelCL always uses one-dimensional work-groups. This seems to have an effect on how work-groups are schedules on the device. In Figure 3.4 the best results of each version are compared.

As already pointed out in the previous section, the SkelCL version is able to use multi-GPU systems by design. Therefore, the unmodified version was measured with two and four Tesla T10 GPUs as well. The results are shown in Figure 3.5. The achieved speedup is 1.5 for two GPUs and 2.2 for four GPUs. Looking at the GPU part unveils a much better speedup of 1.7 or 3.1 respec-

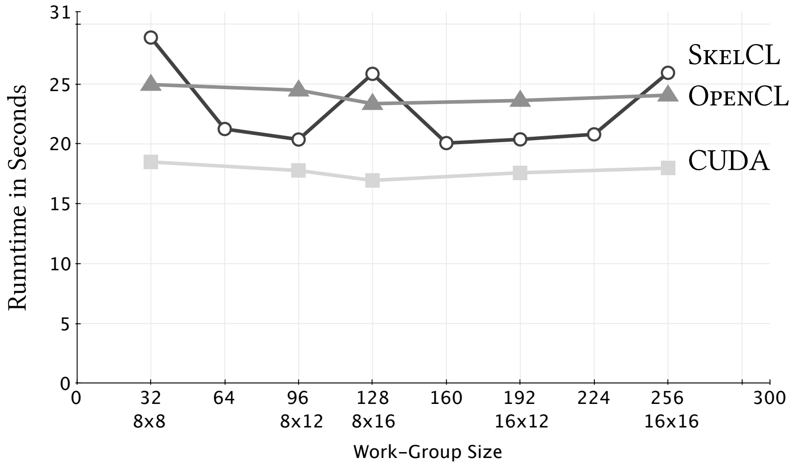


Figure 3.3: Runtime results for different work-group sizes.

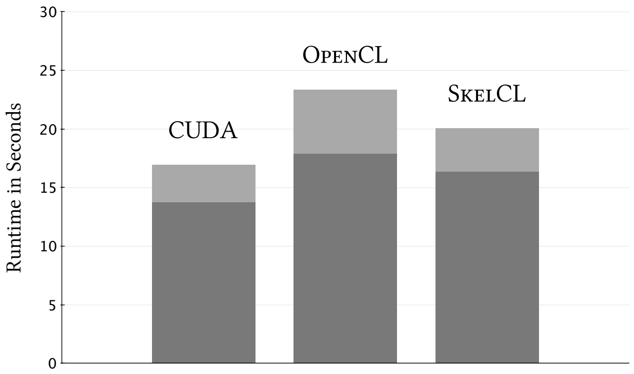


Figure 3.4: Runtime results for the best versions.

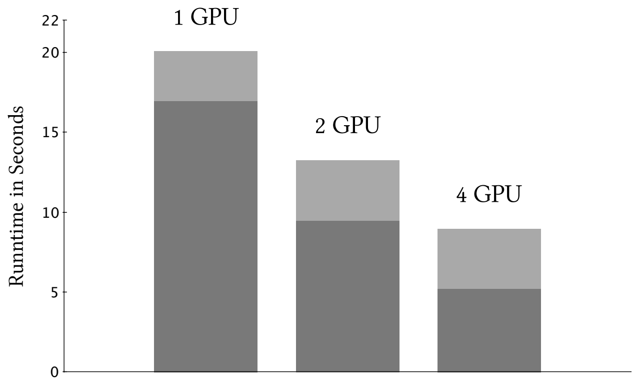


Figure 3.5: Runtime results for SkelCL's multi-GPU version.

tively. Therefore, this example would benefit enormously from optimizations of the host code.

3.1.4 Conclusion

This simple example clearly shows that the performance of SkelCL is more than satisfying. An overhead of less than 5% with the default values, as compared to OpenCL, is neglectable. It is even possible to outperform the OpenCL implementation by tuning the work-group size properly. Furthermore, the presented implementation was multi-GPU ready from start. Even in such a simple example, adding multi-GPU to the application by hand needs much afford and requires many more lines of code. To achieve multi-GPU support for the CUDA implementation, multiple threads on the host side have to be managed. Every thread manages one device. Synchronization between these host threads might be required. Then, the work has to be distributed properly over all devices and the results must be reassembled on the host side. Similar steps have to be done in OpenCL as well.

3.2 List-Mode Ordered Subset Expectation Maximization

The *List-Mode Ordered Subset Expectation Maximization algorithm* (list-mode OSEM) is a numerical algorithm for image reconstruction. It is used to reconstruct three-dimensional images from data recorded by *positron emission tomography* (PET) scanners.

PET is a popular technology for medical imaging. A radioactively marked substance (*tracer*) is injected into the patient. This substance accumulates at specific tissues. When a radioactive particle decays, two positrons are emitted in opposite directions. The scanner surrounds the patient and detects both positrons. The detection of such a pair of positrons is called an *event*. From this data it can be assumed that the decay must have happened on a line between the detected positions. This is called the *line of response* (LOR).

These events are the input data for the list-mode OSEM algorithm. The list-mode OSEM cuts the enormous input data into subsets which are processed iteratively one after another. For each subset, a new version of the reconstructed image is calculated. The computation shown in Equation 3.2 is performed during one iteration.

$$f_{l+1} = f_l c_l \qquad \text{with} \qquad c_l = \sum_{i \in S_l} (A_i)^t \frac{1}{A_i f_l} \tag{3.2}$$

f is the three-dimensional image which is improved in each iteration. S_l is the set of all events to be processed in iteration l. A is the so-called system matrix where each row A_i represents the intersection of one LOR with the image f. This is called the LOR's *path*. Every element a_{ik} of a path describes the length of intersection of the LOR for the given event i with voxel k of the reconstructed image. The path is calculated for every event with Siddon's algorithm [Sid85]. The calculation of c_l is referred to as the *forward projection*, whereas the calculation of f_{l+1} is called the *update*.

A high-level description of the calculation in Equation 3.2 is shown in Listing 3.2. The system matrix A is calculated for every subset as a whole. This can be done, because inside the summation no dependencies between the different rows A_i of the system matrix exists, the order of the execution is not important. Therefore, it is abstracted from the order of the summation suggested by Equation 3.2. A combination of nested skeletons computes c_l. Finally, the new f_{l+1} is calculated.

```
 1 foreach subset {
 2   events = readEvents();
 3   // compute $A$
 4   A = map(events, computePath);
 5   // compute $(A)^t \cdot (Af_l)^{-1}$, then sum up
 6   cl = reduce( zip(A, fl, computeCl), $\sum$);
 7
 8   // compute $f_{l+1}$
 9   fl = zip(fl, cl, computeNewF);
10 }
```

Listing 3.2: High-level implementation of the OSEM algorithm in pseudo code.

Unfortunately, major problems exist with the provided high-level solution. Computing the whole system matrix A all at once requires several gigabytes of memory. Therefore, existing implementations of the list-mode OSEM algorithm avoid this problem by calculate the rows of the matrix A (A_i) one by one. Hence the name of the the algorithm, because the events are organized in a list so that one event after another is processed and the rows are generated in succession. Each row has to be processed immediately such that the result of the further calculation can be added to c_l. In this scenario, the rows do not have to be stored at all. This saves a huge amount of memory.

The Muesli implementation of the list-mode OSEM algorithm uses such an approach, as shown by Ciechanowicz et al. [CKS$^+$10], even so this data parallel implementation performed poorly. This approach still requires the nesting of skeletons, like shown in Listing 3.2, line 6 which is currently not supported in SkelCL.

Therefore, SkelCL orientates more on the existing high-performance sequential and parallel implementations of the algorithm. A simplified version of the existing sequential algorithm is shown in Listing 3.3. To parallelize this version, the loop calculating c_l (line 3—10) can be divided among all participating compute units. Furthermore, the update of the image f performed in lines 11—13 can also be processed in parallel.

This parallelization strategy has been implemented with OpenMP and MPI as well as with CUDA and provides high performance [Sch09, KSG09, MSG09, SVG08, SVGM08, CKS$^+$10].

```
for (l=0; l < numberOfSubsets; l++) {
  events = readEvents();                   // read subset
  for (i=0; i < numberOfEvents; i++) {     // compute $c_l$
    path = computePath(events[i]);         // compute $A_i$
    fp = 0;                                // compute $fp = A_i f_l$
    for (m=0; m < path.length; m++)
      fp += f[path[m].coord] * path[m].length;
    for (m=0; m < path.length; m++)        // add $(A_i)^t fp^{-1}$ to $c_l$
      c[path[m].coord] += path[m].length / fp;
  }
  for (j=0; j < imageSize; j++)            // compute $f_{l+1}$
    if (c[j] > 0.0)
      f[j] *= c[j];
}
```

Listing 3.3: Sequential implementation of the list-mode OSEM algorithm.

3.2.1 Implementation with SkelCL

SkelCL uses a map and a zip skeleton to organize the calculations. The implementation of the main loop is shown in Listing 3.4. The forward projection is provided as a map skeleton. It calculates the correction image c_l. To save memory, the map is not applied to the event vector itself. The event vector stores about one million events for each iteration. Therefore, a smaller vector `index` is used. This has only a length of 512 and stores the numbers of 0 to 511 in it. The map skeleton is applied to this index vector such that the provided function processes a whole bunch of events. The number from the index vector is used to calculate the necessary offset to decide which event is processed next. The provided function processes a bunch of events by iterating over them. For a single event the same steps as in the sequential implementation are performed: a path is computed, `fp` is calculated from the path and `f`, and finally `fp` is added to `c`. Therefore, the image `f` and the correction image `c` are provided as arguments.

After the forward projection, the update is performed. This takes f and c and stores the updated image in f again.

An important feature of this implementation is the distribution of the different vectors among the devices. This enables the multi-GPU capabilities of SkelCL. During the forward projection, the parallelism is achieved by splitting the event vector across the devices. The image `f` and the correction image `c` are required on every device as a whole, since every element can be accessed

```
for (l=0; l < numberOfSubsets; l++) {
    // read events for subset
  events = readEvents();
  Vector<Event> eventVector = Vector<Event>(events);
    // split event vector over the devices
  eventVector.setDistribution(Distribution::block);

    // every device gets a full copy of the image f
    // and the correction image c
  c.setDistribution(Distribution::copy);
  f.setDistribution(Distribution::copy);

    // prepare arguments for the forward projection
  SkelCL::Arguments arguments;
  arguments.push(eventVector);
  arguments.push(eventVector.counts());
  arguments.push(paths); // provide memory for the paths (A_i)
  arguments.push(f);
  arguments.push(c);

    // launch map skeleton for the forward projection
    // (compute c_l)
  forwardProjection(index, arguments);

    // signal that c has been modified in the last call
  c.dataOnDevicesModified();
    // split c and f over the devices
    // for c the function add describes how the
    // different versions are combined to one
  c.setDistribution(Distribution::block, add);
  f.setDistribution(Distribution::block);

    // launch zip skeleton for the update
    // (compute f_{l+1})
    // f is also the output of this skeleton
  update(f, c, f);
}
```

Listing 3.4: Implementation of the list-mode OSEM algorithm in SkelCL.

by every device. Right after the forward projection, `c` is marked as modified because it has been altered during the forward projection. This modification can not be detected automatically by SkelCL because `c` has been passed as an additional argument to the map skeleton. For the update stage, the distribution of `f` and `c` is changed. The parallelism is achieved by distributing the image across multiple devices. This can be done since the update does not require communication between the different parts of the image. The additional commands for changing the distribution make the implementation fully multi-GPU ready.

3.2.2 Comparison to Other Implementations

Due to the memory problems, the presented implementation is similarly structured as the CUDA and OpenCL implementation. Though, simplifications are used, for example, by using the `Vector` class. The memory transfers are performed implicitly in SkelCL whereas they have to be explicitly programmed in CUDA or OpenCL.

By far the most significant improvement in terms of programming style is the multi-GPU programming. To achieve the same dataflow, as easily described in SkelCL, requires a lot of source code in pure CUDA or OpenCL. Two different strategies for data exchange are necessary in CUDA. Both are shown in Listing 3.5. The data exchange necessary after the forward projection is shown in Listing 3.5a. CUDA requires a strict one-to-one mapping of a GPU and a thread on the host side. Therefore, POSIX threads [IEE96] are used in this implementation. After copying data from the different devices, this data is summed up to a single version which is then uploaded back to the devices. The shown version is already optimized because it used all available threads to perform the summation of the data. Especially, the performed offset calculations are perfectly balanced which is complicated and error-prone.

The data exchange following the update stage requires a different strategy and is shown in Listing 3.5b. No combination has to be performed since, in this case, the data should only be transferred to the other devices. The shown optimized version uploads only that part to a device which is not already stored on it. This again requires precise length and offset calculations.

The code shown in Listing 3.5 is required just to perform the data exchange for two special cases. Additional code is required to perform the normal data transfer to upload the data before the computation and to download the results. The kernel launches are not shown here either. In SkelCL all this is done implicitly by changing the distribution of the corresponding vectors. This is

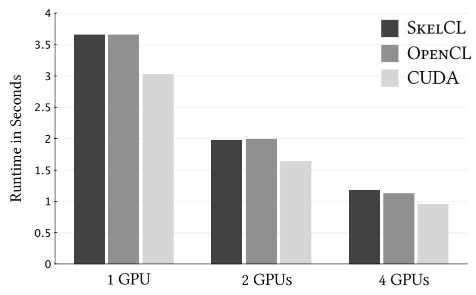
```
  // every host thread copies the image from it's device
cudaMemcpy(&h_temp[device * SIZE], d_c,
           sizeof(float)*SIZE, cudaMemcpyDeviceToHost);
pthread_barrier_wait(&barrier); // host thread synchronisation
  // sum up the image with multiple host threads in parallel
for (int i = device * SIZE / NUM_GPU;
     i < (device + 1)*SIZE / NUM_GPU; i++) {
  for (int j = 1; j < NUM_GPU; j++) {
      // addition of value i from image j
    h_temp[i] += h_temp[j * SIZE + i]; }
}
pthread_barrier_wait(&barrier);
  // every host thread copies the image back to it's device
cudaMemcpy(d_c, h_temp, sizeof(float)*SIZE,
           cudaMemcpyHostToDevice);
```

(a) Data exchange after the forward projection.

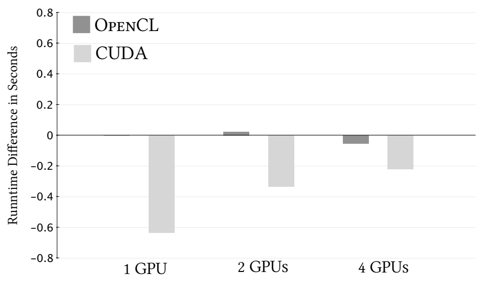
```
  // every host thread copies only a part from it's device
cudaMemcpy( &h_f[device * SIZE/NUM_GPU],
            &d_f[device * SIZE/NUM_GPU],
            sizeof(float) * SIZE/NUM_GPU,
            cudaMemcpyDeviceToHost);
pthread_barrier_wait(&barrier); // host thread synchronisation
  // copy data from all lower devices
if ( device != 0 ) {
  cudaMemcpy( d_f, h_f,
              sizeof(float) * device * SIZE/NUM_GPU,
              cudaMemcpyHostToDevice) ); }
  // copy data from all upper devices
if ( device != (NUM_GPU-1) ) {
  cudaMemcpy( &d_f[device+1 * SIZE/NUM_GPU],
              &h_f[device+1 * SIZE/NUM_GPU],
              sizeof(float) * (NUM_GPU-1-device) * SIZE/NUM_GPU,
              cudaMemcpyHostToDevice); }
```

(b) Data exchange after the update.

Listing 3.5: Code necessary to perform data exchange between multiple devices in CUDA.



(a) Runtime of the OpenCL, CUDA, and SkeCL implementation using one, two, and four GPUs.



(b) Runtime difference of SkelCL as compared to the OpenCL and CUDA implementations.

Figure 3.6: Runtime results for a single iteration of the list-mode OSEM algorithm.

clearly a huge alleviation for the programmer: the provided abstraction help to avoid handling low-level details and concentrates on the essential: the dataflow.

3.2.3 Results

In the sequel, an OpenCL, a CUDA, and a SkelCL version are compared regarding of performance. We used the same setup as in the Mandelbrot set example: a NVIDIA S1070 multi-GPU system consisting of four Tesla T10 GPUs.

Figure 3.6a shows the runtime results of the three different versions for one, two, and four GPUs. The chart shows the runtime for one iteration of the algorithm. Using one GPU, the iteration is performed fastest by the CUDA implementation (3.03 seconds). The OpenCL implementation (3.66 seconds) and the SkelCL implementations (3.66 seconds) are somewhat slower. Though, the results are less different than in the Mandelbrot set example. The worst version (OpenCL) is only about 20% slower than the CUDA version.

With two GPUs, the OpenCL version is equally fast as the SkelCL version, while the CUDA version is faster than the other ones (about 20%). With four GPUs, the CUDA version again outperforms all other versions. It is 23% faster than the SkelCL version an 17% faster than the OpenCL version. Using four GPUs, the OpenCL version is only 5% faster than the SkelCL version.

The differences of the OpenCL and CUDA version compared to the SkelCL version is shown in Figure 3.6b. The speedup is shown in Figure 3.7. For four GPUs, SkelCL provides a speedup of 3.1, while OpenCL and CUDA provide speedups of 3.24 and 3.15 respectively.

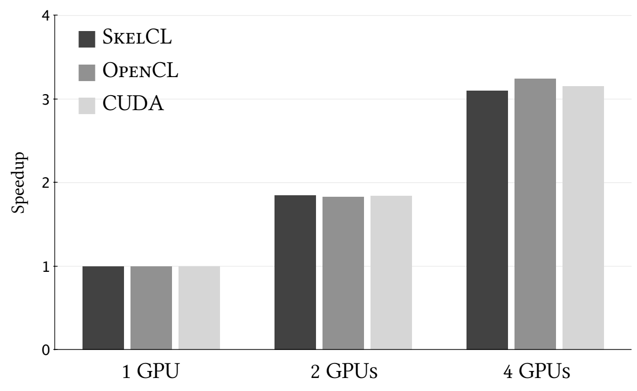


Figure 3.7: Speedup for one iteration of the list-mode OSEM algorithm.

3.2.4 Conclusion

This example shows that SkelCL can be used to implement a real word application while providing high performance. In addition the integration into a sequential version is easy. SkelCL shows a competitive speedup indicating a good scalability. Especially, the memory management and first of all the multi-GPU capabilities of SkelCL provide a higher-level of abstraction as compared to CUDA and OpenCL. Much boilerplate code which is common to multi-GPU programming can saved and the data-flow is described in a high-level manner.

Chapter 4

Conclusion

In this final chapter, we take a closer look at related projects. Besides, future improvements for SkelCL are discussed as well as possible directions of long-term research. We conclude by summing up the capabilities of SkelCL.

4.1 Related Work

Several other projects have tried to overcome different limitations of GPU computing and are presented briefly in the sequel. All of the following projects – except SkePU – work only with CUDA so they are restricted on GPUs by NVIDIA. This is due to the fact, that most of the projects existed before OpenCL was released.

CuPP

The project *CuPP* tries to provide an easy integration of CUDA code into C++ applications [Bre09]. A data structure is provided, that does not have to be exchanged explicitly between the main memory and the memory on the graphics accelerator. Also it is possible to pass C++ classes as parameters to kernels. Multi-GPU applications are feasible with CuPP but no specific support for that is provided.

The MapReduce Framework on GPUs

The *MapReduce* Framework was introduced in 2004 by Google to speed up parallel computing on large data [DG04]. It can be divided into two stage: the map and the reduce stage. During the map stage, a given function is executed on every element of the data set. In the reduce stage, the results of the map stage are grouped together and combined to produce information of interest. The framework can, for example, be used to perform statistic analysis on large data sets. Two projects implement the framework on GPUs. The Hong Kong University of Science and Technology developed a project called *Mars* [HFL+08], and a *Map Reduce Framework* was developed by the University of California, Berkley [CSK08]. This is an example of projects encapsulating a certain, commonly used, algorithm on GPUs. MapReduce is one special algorithmic skeleton which is not supported by SkelCL. The usage of these two projects is limited to this single skeleton. Multi-GPU versions of the skeleton are not available.

CUDPP

CUDPP (CUDA Data Parallel Primitives Library) is a C++ library based on CUDA, which provides data-parallel algorithm *primitives* [SHZO07]. A primitive defines a parallel algorithm, which can be configured by choosing from a set of operations. For example, to calculate a prefix sum, `SCAN` is chosen as algorithm and `ADD` as operation. This configured primitive is then executed on a set of data. CUDPP does not ease the data management, because data still has to be copied explicitly to the graphics accelerator before execution. SkelCL adopted the scan algorithm of CUDPP. But while CUDPP only supports a fix set of operations, arbitrary binary functions can be passed to the scan skeleton in SkelCL. CUDPP is not able to use multiple GPUs either.

GPUSs

An implementation of the Star Superscalar (StarSs) programming model for multi-GPU systems, called *GPUSs* has been developed by the Barcelona Supercomputing Center [ABI+09]. It is a successor to implementations for grid computing (GRIDSs), the Cell B. E. (CellSs), and multi-core processors (SMPSs). The basic idea is to add annotations to the source code such that a source-to-source compiler is able to generate GPU code. The GPUSs version primarily aims for simplifying the handling of multiple GPUs. Tasks can be specified that are scheduled to available GPUs or CPU-cores. The task can have dependencies among themselves which are respected by the scheduler. The system aims for providing simple task-parallelism whereas SkelCL offers concepts to simplify data-parallel applications. The data management is simplified a bit in GPUSs since a simple annotation can be made to indicate a data transfer. SkelCL offers a higher-level memory abstraction because the data movement is described, not the single data transfers. GPUSs minimizes data transfers between the GPU and the CPU by implementing a software cache system. Data resides on the

GPU as long as possible to reduce the need of uploading the same data a second time. SkelCL implements a similar scheme inside the vector class.

StarPU

StarPU is quite similar to GPUSs [ATNW09]. A task scheduler for a heterogeneous environment is provided. Tasks are described as *codelets*. A codelet encapsulated various implementations for different architectures, like one for CUDA and one for multi-core CPUs. Dependencies between different codelets can also be defined. Hence, StarPU offers support for easy task-parallel execution, like GPUSs. In StarPU, predefined as well as user-defined scheduling strategies can be used. SkelCL offers no task-parallel concepts but only data-parallel ones, with the use of the algorithmic skeletons. Therefore, StarPU and SkelCL use multi-GPU systems entirely different.

Thrust

Thrust is an open source library, developed by NVIDIA [HB09]. It provides parallel data structures and algorithms similar to those available in the C++ STL. Especially, container classes for dynamic arrays are available. Unlike SkelCL, Thrust differentiates between containers for data in the main memory and containers for data in the memory on the graphics accelerator. In SkelCL, a unified memory management is provided which connects host and device memory. The algorithms supplied in Thrust cover for example, sort and search algorithms. To apply an algorithm, a functor object is passed as parameter which defines the behavior, along with data on which the calculation is performed. The functions defined in the functor objects are then executed on the GPU. Some of the skeletons provided in SkelCL are also implemented in Thrust. So is the map skeleton in SkelCL implemented as the transform function in Thrust. Thrust is not multi-GPU ready – in contrast to SkelCL.

SkePU

SkePU is a relatively new project developed at the University of Linköping in Sweden [EK10]. It uses algorithmic skeletons to ease GPU computing similar to SkelCL. Many of the ideas of SKePU are also implemented in SkelCL or the other way round. Even though both projects were developed independently at the same time.

The skeletons map and reduce are implemented in both libraries whereas SkePU supports additional variants of the map skeleton as well as a map-reduce

```
// marco creates a functor, this is passed to a skeleton
BINARY_FUNC(plus, double, a, b,
  return a+b;
)

int main() {
skepu::Map<plus> sum(new plus); // create the map skeleton

// create vectors of length 10 with 10 or 5 as values
skepu::Vector<double> v0(10, 10);
skepu::Vector<double> v1(10, 5);
skepu::Vector<double> r;

sum(v0, v1, r); // launch the map skeleton

std::cout << "Result: " << r << "\n";
// Output: "Result: 15 15 15 15 15 15 15 15 15 15"
}
```

Listing 4.1: A simple map example with SkePU. Based on [EK10].

skeleton. SkelCL supports the zip skeleton and the scan skeleton which are not implemented in SkePU.

In SkePU a special macro syntax is used to simplify the creation of skeletons. Listing 4.1 shows a simple example how SkePU is used. With the help of a provided macro a function is defined. The macro expands to a C++ class which encapsulates the defined function. This is later passed to the map skeleton.

This process is different to SkelCL where the source is not passed to a macro, but as a string to the skeleton directly.

An advantage of the method chosen by SkePU is that code for different architectures is generated. The macros expand to classes which currently contain three different variants of the user-provided source code. An OpenCL, a CUDA, and a CPU version is generated by the macro. The macro is evaluated before the actual compiling stage is executed. This allows for generation of code used for CUDA and the CPU as well as strings which are used for the OpenCL. CUDA and CPU code can not be generated by SkelCL, because code is only available as a string.

A disadvantage of generating different versions from the same macro is, that these versions can not be adopted for the different architectures. But this is required even for simple applications: in CUDA and OpenCL it is essential to use the provided IDs of the current work-item. These have to be accessed in different ways in CUDA and OpenCL. On a CPU no such information is available at all. With SkePU, accessing these IDs using, for example, the provided OpenCL function, brakes the CUDA versions and the other way round.

Beside the skeletons, a container class for managing memory and especially for transferring data between CPU memory and GPU memory is provided by SkePU. Similar to SkelCL, the container class is designed as a unified memory abstraction.

From all the other projects presented, SkePU is the only project supporting data-parallel calculations on multi-GPU systems. The skeletons provided by SkePU can be executed on multiple GPUs. Therefore, the input vector is divided equally amongst the participating devices. The multi-GPU capabilities can be used with CUDA and OpenCL. But according to Enmyren and Kessler [EK10], the CUDA version is currently not very efficient, since new threads are spawned every time a skeleton is launched and destroyed afterward.

The multi-GPU support in SkelCL is more sophisticated than the one in SkePU. Different distributions of the vector classes can be chosen and the dataflow inside an application can be modified by changing the distribution at runtime. The multi-GPU capabilities available in SkePU can also be reproduced with SkelCL.

4.2 Future Work

SkelCL in its current form is a prototype of a high-level GPU computing programming environment. Using memory management and a high-level abstractions based on algorithmic skeletons, especially for multi-GPU programming, SkelCL introduces major improvements to the current programming style on GPUs. In this section, further improvements to SkelCL and general GPU computing are discussed. This section is divided into possible short-term, mid-term and long-term enhancements. The short-term enhancements are relatively simple changes to SkelCL, whereas the mid-term enhancements are may require major changes in the design of SkelCL. In the long-term enhancements section, possible directions of research based on the results achieved with SkelCL are described.

Short-Term Enhancements

The experiments described in Sections 3.1.3 and 3.2.3 have been conducted exclusively on hardware made by NVIDIA. It would be interesting to compare the performance of SkelCL executed on similar devices build by other vendors, like AMD.

The list-mode OSEM application shows that it is possible to implement a real world application with SkelCL. Additional applications have to be implemented with SkelCL to analyze weaknesses and strongs further. Additional skeletons can be identified by implementing real world applications from different areas.

The vector class provided by SkelCL could be enhanced to fit more naturally into C++ applications. This would allow for easy porting existing applications currently using the C++ STL. In the current version, a C-style pointer is used to pass data to the vector. This could be replaced by the use of iterators, which is common for container classes in the C++ STL. In addition, the access to elements in the vector could be eased by overloading the index operator.

The sequence structure, provided by SkelCL, is type-safe the composition structure is not. The type-safety could be added to the composition structure in the future. Possible type errors would be detected by the compiler, instead of producing crashes at runtime. This simplifies the debugging process, and assists to write correct code.

A common pattern in parallel computing is the need to distribute data across multiple computing devices. This is provided in SkelCL by the ability to distribute a vector among multiple devices. Currently, three different distributions are offered. Additional distribution schemes might be required by other applications. For example, image processing applications often require to access data nearby the current location in memory. If the data is distributed, the data might be stored on a different device. To overcome this problem, a new distribution scheme could be added which provides overlapping memory regions. These regions would be copied to multiple devices so that the required memory can be accessed on every device in parallel. Appropriate synchronization mechanisms are already implemented in SkelCL and could be enhanced for this distribution.

SkelCL is based on OpenCL and, therefore, vendor independent by design. Though, different architectures require different optimizations. For example, the work-group size can have a huge impact on the runtime as we saw in the Mandelbrot example (see Section 3.1). Parameter files describing different GPU architectures could be introduced. SkelCL should read these files at runtime and choose the best optimization for the given architecture. The optimizations could range from the work-group size to special adopted implementations of the skeletons. A format to describe an architecture should be developed and specified. With OpenCL, it is even possible to detect devices at start-up time and load a compatible parameter file automatically.

Mid-Term Enhancements

In general, OpenCL allows for executing kernels on both, CPU and GPU. But most current OpenCL drivers only provide access to a limited set of a system's devices, mostly GPUs. Hence, SkelCL only uses GPUs for kernel execution. A recent extension to the OpenCL API, namely the installable client driver loader (ICD), allows for using multiple OpenCL implementations concurrently. Using this extension it is much more likely to provide access to all of a system's OpenCL devices by installing multiple OpenCL drivers. Thus, SkelCL can use CPUs and GPUs for kernel execution, which will allow for a more efficient implementation of multi-GPU skeletons, e.g., the reduce skeleton 2.6.3.

Currently, the implementations of the map and the zip skeleton do not use the local memory available on the devices. Versions of these skeletons could be developed using this additional resource. Though the implementations is not straightforward, because the skeletons are designed to work with arbitrary data types. This complicates the use of the local memory because the size of this types can vary and the amount of local memory is strictly limited. Especially, a possible skeleton supporting direct access to near-by memory regions could benefit from the local memory.

Other applications implemented with SkelCL could lead to additional skeletons. Even the list-mode OSEM could be described in a higher-level manner by the use of specialized skeletons, like a combined map-reduce skeleton. For example, a map-reduce skeleton can be optimized to use much less memory than a combination of a map and multiple reduce skeletons. A decision has to be made if arbitrary specialized skeletons should be supported in SkelCL or if only a small key set of skeletons should be provided. If real nesting could be supported, many special skeletons can be expressed as a combination of less complex skeletons. This idea is implemented in Muesli.

Beside the vector class, additional datatypes could be added to SkelCL. The vector class can be used like a vector in mathematics as an operand for common operators because of the overloaded operators. This could also be provided for two or three-dimensional matrices. For example, an efficient matrix-matrix multiplication could be provided. Simple calculations only using the provided operators could be expressed with an even simpler syntax than the one provided with the skeletons. The composition structure supported by SkelCL can be used to chain and organize such operations. Implementations of sparse data structures along with optimized skeletons for these data structures would would ease programming of applications that, e.g., use sparse linear algebra.

Currently, the composition and sequence structures can be used to organize skeleton executions. In combination with the provided operators of the vector class, an even simpler syntax to specify computations can be used. In future versions of SkelCL, the composition and sequence structure could combine the kernels of multiple skeletons to one. The source code specifying skeletons could be used to generate one bigger kernel out of multiple skeleton definitions. This would reduce the overhead of launching multiple kernels one after another. A complicated calculation, for example, described by multiple skeletons, could be merged to a single kernel. This kernel would then be executed instead of multiple ones. This could not only save the time necessary to launch multiple skeletons but would also save memory since intermediate results can be processed right away instead of writing them to global memory.

Long-Term Enhancements

In a long-term perspective, the ability to nest skeletons would extensively advance the power of SkelCL. In Muesli, arbitrary nesting of skeletons is one of the key features and the main reason for the flexibility of Muesli. In GPU computing, nesting is difficult. The main reason for that is, that no additional threads can be launched inside GPU code. But, for example, nesting two map skeletons would require this. Additional concepts have to be developed to overcome this challenge. Since the source code of all participating skeletons is available, this code could be analyzed and new code could be generated. For example nested skeletons could be combined into a single kernel. Among other things, an advanced understanding of the source code and memory restrictions are required for this.

Macro definitions would provide an alternative approach for passing kernel definitions in SkelCL. Some of the advantages and disadvantages have already been discussed in Section 4.1. The prominent advantage of this approach (implemented, e.g., by SkePU [EK10]) is that source code for CUDA- and OpenCL-based devices as well as the CPU code can be generated from a single source. On the other hand, the user cannot specify special code for a certain architecture. An advanced marco definition language could be developed to overcome this disadvantage. The most commonly used architecture-dependent adaptions in GPU computing is the use of built in functions and constants.

CUDA and OpenCL both provide similar concepts such that macro definitions could abstract from these differences. On a CPU, a similar environment could be provided with appropriate replacements for missing functions or constants.

A more thorough redesign of SkelCL could render it a fully integrated programming language for GPU computing. By using a source-to-source compiler, the traditional syntax could be extended. SkelCL could be freed from the syntactical restrictions implied by OpenCL. The compiler could generate valid OpenCL or CUDA code from a common syntax. Functions executing on the GPU could be annotated and then the code necessary for memory transfers should be generated automatically. Current restrictions in GPU code could be avoided by generating necessary host code aside with the GPU code. Finally, a unified programming model for high-level programming could be provided. The compiler could unite the distinct memory regions on CPU and GPU, decreasing the complexity of GPU computing. Ideas from functional programming languages could also be adapted. Especially, the concept of *anonymous functions* (also known as *lambda expressions*) is very powerful. An anonymous function is a function defined without a name and getting assigned to a variable. This variable can then be passed around, for example, as parameter of another function. The function is applied by using the current variable name like a usual function name. In SkelCL, this concept – currently missing in C++ – would have helped to drastically simplify the implementation and usage of skeletons. This concept is common to almost every functional language and also included in the new C++ standard [ISO10]. A unified programming model for GPU and CPU computing would render the GPU a true coprocessors to the CPU.

4.3 SkelCL Summarized

We presented SkelCL, a portable multi-GPU skeleton library based on OpenCL. Its key features are an abstraction for memory management and the use of skeletons.

Four basic skeletons are provided with SkelCL: map, zip, reduce and scan. All can be used either with a single-GPU or with multiple GPUs. Skeletons can be combined with the sequence and the composition structures. The composition offers a convenient and rather flexible way to structure calculations.

A unified memory abstraction is given by the vector class. It connects memory on the host side with memory on one or more GPUs. Memory transfers are performed automatically and only if needed. This lazy data copying allows for high performance implementations while providing a high-level of abstraction.

One of the most important features of SkelCL is the multi-GPU capabilities of the vector class. A vector can be distributed among multiple GPUs in different ways. By changing the distribution, dataflow can be described. This mechanism is used to perform data synchronization and movement implicitly. The user is freed from low-level details entirely.

Experimental results show that SkelCL can be used to implement high-performance applications. By using a real word example, it is demonstrated that SkelCL can easily be integrated into an existing sequential application. Its performance is close to that of CUDA and OpenCL.

Bibliography

[ABI+09] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. *An Extension of the StarSs Programming Model for Platfroms with Multiple GPUs.* In H. J. Sips, D. H. J. Epema, and H. Lin, editors, *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009 Proceedings*, volume 5704 of *Lecture Notes in Computer Science*, pages 851–862. Springer, 2009.

[AG05] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.* The C++ In-Depth Series. Addison-Wesley, 2005.

[AMD09] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*, November 2009. Revision 3.15. Available from: `http://support.amd.com/us/Processor_TechDocs/24594.pdf`.

[App09] Apple Inc. *OpenCL Parallel Reduction Example*, 2009. Version 1.3. Available from: `http://developer.apple.com/mac/library/samplecode/OpenCL_Parallel_Reduction_Example/index.html`.

[ATNW09] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. *StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures.* In H. J. Sips, D. H. J. Epema, and H. Lin, editors, *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009 Proceedings*, volume 5704 of *Lecture Notes in Computer Science*, pages 863–874. Springer, 2009.

[BCGH05] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. *Flexible Skeletal Programming with eSkel.* In J. C. Cunha and P. D. Medeiros, editors, *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Proceedings*, volume 3648 of *Lecture Notes in Computer Science*, pages 761–770. Springer, 2005.

[BFH$^+$04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. *Brook for GPUs: stream computing on graphics hardware. ACM Transactions on Graphics*, 23(3):777–786, August 2004.

[Ble89] G. E. Blelloch. *Scans as Primitive Parallel Operations. IEEE Transactions on Computers*, 38:1526–1538, 1989.

[Bre09] J. Breitbart. *CuPP - A framework for easy CUDA integration.* In *IPDPS*, pages 1–8. IEEE, 2009.

[CKS$^+$10] P. Ciechanowicz, P. Kegel, M. Schellmann, S. Gorlatch, and H. Kuchen. *Parallelizing the LM OSEM Image Reconstruction on Multi-Core Clusters.* In B. Chapman, F. Desprez, G. R. Joubert, A. Lichnewsky, F. Peters, and T. Priol, editors, *Parallel Computing: From Multicores and GPU's to Petascale*, volume 19 of *Advances in Parallel Computing*, pages 169–176. IOS Press, 2010.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.

[CPK09] P. Ciechanowicz, M. Poldner, and H. Kuchen. *The Münster Skeleton Library Muesli – A Comprehensive Overview.* Technical report, University of Münster, 2009.

[CSK08] B. Catanzaro, N. Sundaram, and K. Keutzer. *A Map Reduce Framework for Programming Graphics Processors.* presented at the Third Workshop on Software Tools for MultiCore Systems, Boston, Massachusetts, 2008.

[Cxx01] *Itanium C++ ABI* [online]. 2001. Available from: `http://www.codesourcery.com/cxx-abi/` [cited 2010-08-05].

[DG04] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters.* In *6th Symposium on Operating Systems Design & Implementation (OSDI '04), Proceedings*, pages 137—150, 2004.

[EK10] J. Enmyren and C. Kessler. *SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems.* In *Proceedings 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010) (to appear)*, Baltimore, MD, USA, September 2010.

[FB09] Q. Fang and D. A. Boas. *Monte Carlo simulation of photon migration in 3D turbid media accelerated by graphics processing units. Opt. Express*, 17(22):20178–20190, 2009. Available from: `http://www.opticsexpress.org/abstract.cfm?URI=oe-17-22-20178`.

[FIP08] *Secure Hash Standard*, 2008. Federal Information Processing Standard 180-3. Available from: `http://csrc.nist.gov/publications/fips/`.

[HB09] J. Hoberock and N. Bell. *Thrust: A Parallel Template Library*, 2009. Version 1.1. Available from: `http://www.meganewtons.com`.

[HFL$^+$08] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. *Mars: A MapReduce framework on graphics processors.* In A. Moshovos, D. Tarditi, and K. Olukotun, editors, *17th International Conference on Parallel Architecture and Compilation Techniques (PACT 2008)*, pages 260–269. ACM, 2008.

[HSO07] M. Harris, S. Sengupta, and J. D. Owens. *Parallel Prefix Sum (Scan) with CUDA.* In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[IEE96] Technical Committee on Operating Systems and Application Environments of the IEEE. *POSIX, Part 1: System API, ANSI/IEEE Std 1003.1c, Amendment 2: Threads Extension*, 1996.

[ISO10] ISO/IEC. *Programming Languages – C++ (N3092)*, March 2010. Available from: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf`.

[KDY$^+$10] J. Kong, M. Dimitrov, Y. Yang, J. Liyanage, L. Cao, J. Staples, M. Mantor, and H. Zhou. *Accelerating MATLAB Image Processing Toolbox functions on GPUs.* In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 75–85, New York, NY, USA, 2010. ACM. Available from: `https://sites.google.com/site/iptatiproject/`.

[KSG09] P. Kegel, M. Schellmann, and S. Gorlatch. *Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-cores.* In H. J. Sips, D. H. J. Epema, and H. Lin, editors, *Euro-Par*

2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009 Proceedings, volume 5704 of *Lecture Notes in Computer Science*, pages 654–665. Springer, 2009.

[Lev99] J. R. Levine. *Linkers and Loaders.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[LKC$^+$10] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. *Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. SIGARCH Comput. Archit. News*, 38(3):451–460, 2010.

[LOS09] N. Leischner, V. Osipov, and P. Sanders. *Fermi Architecture White Paper.* NVIDIA, 2009. Available from: `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[Man80] B. B. Mandelbrot. *Fractal Aspects of the Iteration of $z \mapsto \lambda z(1-z)$ for complex λ and z. Annals of the New York Academy of Sciences*, 357:249–259, December 1980.

[MPI09] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 2009. Version 2.2. Available from: `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`.

[MQP02] M. D. McCool, Z. Qin, and T. S. Popa. *Shader Metaprogramming.* In T. Ertl, W. Heidrich, and M. Doggett, editors, *SIGGRAPH Eurographics Workshop on Graphics Hardware*, pages 57–68. Eurographics Association, 2002.

[MSG09] D. Meiländer, M. Schellmann, and S. Gorlatch. *Implementing a Data-Parallel Application with Low Data Locality on Multicore Processors.* In Karl-Erwin Groß-pietsch, Andreas Herkersdorf, Sascha Uhrig, Theo Ungerer, and Jörg Hähner, editors, *International Conference on Architecture of Computing Systems - Workshop Proceedings*, pages 57–64, Delft, NL, March 2009.

[Mun10] A. Munshi. *The OpenCL Specification.* Beaverton, OR, 2010. Version 1.1, Document Revision: 33.

[NVI06a] NVIDIA. *New NVIDIA Products Transform the PC Into the Definitive Gaming Platform.* Press Release, 2006. Available from: `http://www.nvidia.com/object/IO_37234.html`.

[NVI06b] NVIDIA. *NVIDIA Unveils CUDA – The GPU Computing Revolution Begins.* Press Release, 2006. Available from: `http://www.nvidia.com/object/IO_37226.html`.

[NVI10] NVIDIA. *NVIDIA CUDA C Programming Guide*, version 3.1.1 edition, July 2010. Available from: `http://developer.nvidia.com/object/cuda_3_1_downloads.html`.

[OMP08] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 2008. Version 3.0. Available from: `http://www.openmp.org/mp-documents/spec30.pdf`.

[SA09] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification.* Beaverton, OR, 2009. Version 3.2.

[Sch09] M. Schellmann. *Efficient PET Image Reconstruction on Modern Parallel and Distributed Systems.* PhD thesis, Westfälische Wilhelms-Universität Münster, 2009.

[SEN10] D. G. Spampinato, A. C. Elster, and T. Natvig. *Parallel Computing: From Multicores and GPU's to Petascale*, volume 19 of *Advances in Parallel Computing*, chapter Modelling Multi-GPU Systems, pages 562–569. IOS Press, 2010.

[SHZO07] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. *Scan Primitives for GPU Computing.* In *Graphics Hardware 2007*, pages 97–106. ACM, August 2007.

[Sid85] R. L. Siddon. *Fast calculation of the exact radiological path for a three-dimensional CT array. Med Phys*, 12(2):252–5, 1985.

[Str00] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[SVG08] M. Schellmann, J. Vörding, and S. Gorlatch. *Systematic Parallelization of Medical Image Reconstruction for Graphics Hardware.* In *Euro-Par 2008 - Parallel Processing*, volume 5168 of *LNCS*, pages 811–821. Springer, 2008. ISBN 978-3-540-85450-0.

[SVGM08] M. Schellmann, J. Vörding, S. Gorlatch, and D. Meiländer. *Cost-Effective Medical Image Reconstruction: From Clusters to Graphics Processing Units.* In *Proceedings of the 2008 Conference on Computing frontiers*, pages 283–292. ACM, 2008. ISBN 978-1-60558-077-7.

[THH08] N. Trevett (NVIDIA), J. Hensley (AMD), and M. Harris (NVIDIA). *OpenCL The Open Standard for Heterogeneous Parallel Programming*, 2008.

[vMAF$^+$08] J. A. van Meel, A. Arnold, D. Frenkel, Portegies S. F. Zwart, and R. G. Belleman. *Harvesting graphics power for MD simulations. Molecular Simulation*, 34(3):259–266, 2008.

Preprints
"Angewandte Mathematik und Informatik"

05/08 - S	G. Alsmeyer, G. Hölker: Asymptotic Behavior of Ultimately Contractive Iterated Lipschitz Functions
06/08 - N	E. Pekalska, B. Haasdonk: Kernel Quadratic Discriminant Analysis with Positive Definite and Indefinite Kernels
07/08 - S	M. Meiners: Weighted Branching and a Pathwise Renewal Equation
08/08 - S	M. Ebbers, M. Löwe: Torpid Mixing of the Swapping Chain on Some Simple Spin Glass Models
09/08 - I	T. Ropinski, I. Viola, M. Biermann, F. Lindemann, R. Leißa, H. Hauser, K. Hinrichs: Multimodal Closeups for Medical Visualization
01/09 - I	J. Mensmann, T. Ropinski, K. Hinrichs: An Evaluation of the CUDA Architecture for Volume Rendering
02/09 - N	P. Henning, M. Ohlberger: Advection-diffusion problems with rapidly oscillating coefficients and large expected drift. Part 1: Homogenization – existence, uniqueness and regularity
03/09 - N	P. Henning, M. Ohlberger: Advection-diffusion problems with rapidly oscillating coefficients and large expected drift. Part 2: The heterogeneous multiscale finite element method
04/09 - I	J. Meyer-Spradow, T. Ropinski, J. Mensmann, K. Hinrichs: Rapid Prototyping of Volume Visualization in Collaboration with Domain Experts
05/09 - N	K. Mikula, M. Ohlberger: A New Level Set Methof for Motion in Normal Direction Based on a Forward-Backward Diffusion Formulation
06/09 - I	T. Ropinski, S. Diepenbrock, S. Bruckner, K. Hinrichs, E. Gröller: Volumetric Texturing
07/09 - I	J.-S. Praßni, J. Mensmann, T. Ropinski, K. Hinrichs: Shape-based Transfer Functions for Volume Visualization
08/09 - N	A. Dedner, R. Klöfkorn, M. Nolte, M. Ohlberger: A generic interface for parallel and adaptive scientific computing: Abstraction principles and the DUNE-FEM module
09/09 - N	P. Henning, M. Ohlberger: A-posteriori error estimate for a heterogeneous multiscale finite element method for advection-diffusion problems with rapidly oscillating coefficients and large expected drift
01/10 - N	K. Mikula, M. Ohlberger: A New Inflow-Implicit/Outflow-Explicit Finie Volume Method for Solving Variable Velocity Advection Equations
02/10 - N	M. Drohmann, B. Haasdonk, M. Ohlberger: Reduced Basis Approximation for Nonlinear Parametrized Evolution Equations based on Empirical Operator Interpolation
03/10 - N	M. Ohlberger, K. Smetana: A new problem adapted hierarchical model reduction technique based on reduced basis methods and dimensional splitting
04/10 - I	M. Steuwer, P. Kegel, S. Gorlatch: SkelCL – A Portable Multi-GPU Skeleton Library