Fachhochschul-Bachelorstudiengang
**SOFTWARE ENGINEERING**
A-4232 Hagenberg, Austria

# GPGPU Computing with OpenCL

Bachelorarbeit
Teil 1

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

**Bernhard Manfred Gruber**

Begutachter: FH-Prof. DI Dr. Heinz Dobler

Hagenberg, September 2013

# Kurzfassung

Diese Arbeit bietet eine Einführung in die Programmierung von GPUs unter Verwendung von OpenCL. Nach einem historischen Überblick über die Entwicklung von Grafikkarten, werden die Besonderheiten von GPU und CPU Hardware erläutert. Basierend auf diesem Wissen wird OpenCL als API vorgestellt, das die Programmierung vor allem dieser Hardwaretypen unterstützt. Ein genauere Betrachtung des OpenCL Ausführungs- und Speichermodels, das die Unterstützung heterogener Hardware ermöglicht, wird durch ein einfaches Codebeispiel abgerundet.

Anschließend fährt die Arbeit mit der Implementierung einiger Standardalgorithmen fort, die mittels OpenCL auf die GPU gebracht werden. Die ausgewählten Problemstellungen beginnen mit Matrixmultiplikation und setzen sich über All-Prefix-Sum und Sortieralgorithmen fort. Aufgrund der Parallelität die bereits in der Natur der ersten Problemstellung liegt, wird im betreffenden Kapitel ein besonderer Fokus auf Performanceanalyse und Optimierung gelegt. Die All-Prefix-Sum und Sortieren sind schwieriger in unabhängige Arbeitspakete aufzuteilen. Entsprechende Methoden werden vorgestellt, um mit dieser Art von Problemen umzugehen.

Jede GPU Implementierung wird einer Laufzeitmessung unterzogen und mit entsprechenden CPU Varianten verglichen. Da CPUs und GPUs sehr unterschiedliche Hardwarearchitekturen aufweisen, wurden entsprechende Algorithmen zur Lösung der jeweiligen Problemstellung ausgewählt, die das Potential der zugrunde liegenden Plattform bestmöglich ausnutzen.

# Abstract

This thesis provides a introduction into programming for GPUs using OpenCL. After a historical overview of how graphic cards have evolved, the peculiarities of GPU and CPU hardware are discussed. Based on this knowledge, OpenCL is introduced as an API supporting all kinds of processing hardware. A deeper look into OpenCL's execution and memory model, which allows handling heterogeneous hardware, is rounded off by a simple, yet full example code.

The thesis then continues with several implementations of standard algorithms for the GPU. The chosen problems start with matrix multiplication and go along with the all-prefix sum and sorting. As the first problem already offers parallelism naturally, performance analysis and optimization is focused during the first implementation chapter. The all-prefix sum and sorting are both problems being more difficult to split into independent pieces of work. Techniques will be discussed to tackle such kind of problems.

Each GPU implementation is benchmarked and compared with one or more traditional CPU approaches. As GPUs and CPUs have different hardware architectures, appropriate algorithms and optimizations have been chosen to solve the problems by exploiting the underlying platform at best.

# Contents

# 1 Introduction

## 1.1 Motivation

For a long time the speed of programs experienced a constant growth through improved processor hardware. Intel co-founder Gordon E. Moore was one of the first persons to describe this trend in 1965. This description is well known by the name Moore's law which initially stated that the number of transistors on an integrated circuit doubles every year (Moore 1965). This trend slowed down and Moore had to change the interval to two years a decade later (Kanellos 2003). The most important consequence however has been realized by one of Moore's co-workers at Intel, David House, who predicted that the performance of processors would double every 18 months (Kanellos 2003). As a result, an increase in computing power could be obtained simply by running the same programs on newer hardware. As a consequence, hardware dominated the growth of programs' performance for a long period of time.

Unfortunately, processing hardware technologies hit a limit at the beginning of the third millennium. Physical bounds, like the size of atoms, prevented a further shrinking of integrated circuits which would allow an increase in clock speed. Therefore, processor vendors like Intel and AMD started to place multiple CPUs on a single chip which still lead to an increase in computational power but in a different way than in the last 40 years. This change is sometimes referred to as the multicore crisis (Warfield 2007). The consequence from a software developer's perspective is that traditional algorithms stopped gaining speed by being run on newer hardware. In fact, improving performance is now up to the programmer, who is responsible for writing concurrent and parallel software that can make full use of the underlying hardware's capabilities.

But multicores were not the only major hardware change that had an impact on the way software is developed recently. Graphics cards, which were initially intended to offload and accelerate 2D and 3D graphic operations from the CPU to a separate hardware device, started to gain popularity in non graphical areas of programming. A GPUs intense floating point processing power and parallel nature by design makes it ideal for uses in several areas of science, business and engineering (General Purpose GPU). When used in the right place, a GPU may compute results several magnitudes faster than the CPU (McClanahan 2010). However, due to the very different hardware architecture, the graphics card is still not suitable to efficiently solve the same problems as a CPU does.

Todays software engineers working in performance focused domains have a hard time designing their applications. With two different types of processors (CPU and GPU), both excellent in their own ways, and a lot of APIs and frameworks around them (OpenMP, CUDA, OpenCL, DirectCompute, C++ AMP, ...), the available hardware and corresponding software is more heterogeneous than ever. Thus, profound knowledge of the available technologies, their performance and restrictions as well as the consequences for a development team is essential for choosing the optimal strategy to tackle todays computational needs.

## 1.2 Goal

The goal of this thesis is to provide the reader with a state of the art comparison of modern GPU and CPU performance in solving traditional problems such as sorting arrays and multiplying

matrices using various approaches. The GPU algorithms will be implemented using the Open Computing Language (OpenCL). Therefore the reader is given a short introduction into OpenCL in order to understand the provided code samples and how programming for a GPU works. These OpenCL implementations are benchmarked against algorithms implemented in C/C++ solving the same problem. It is important to note that the GPU and CPU version do not necessarily have to use the same algorithm, they only have to output the same result. This decision was inevitable due to the highly diverse hardware properties of the CPU and the GPU. The problems chosen for this comparison are sorting an array, multiplying two large matrices and calculating the parallel prefix sum of an array (explained in corresponding chapter 4). These algorithms cover several different aspects that play an important role when implementing software for a GPU such as runtime complexity, memory footprint and memory transfer time vs. computation time.

Eventually, the results of benchmarking the chosen algorithms should give the reader an idea of how much a GPU can accelerate different kinds of algorithms and what a programmer has to pay attention to when developing GPGPU accelerated software. This knowledge is not only valuable during developing but also useful when choosing the right technology for a given problem. This thesis should aid all software engineers in understanding how GPU computing works and where it should be used.

## 1.3 History of GPGPU Computing

Before we go into details about OpenCL and several GPU implementations, some background information about how GPUs have evolved is provided which may help understanding the design and peculiarities of graphics hardware. The following information is taken from a paper survey in 2010 (McClanahan 2010).

In the early 1980's a "GPU" was nothing more than an integrated frame buffer, an additional chip that relied on the CPU for drawing.

One of the first dedicated video cards was the IBM Professional Graphics Controller (PGA) released in 1984. It used an on-board Intel 8088 microprocessor to take care of graphical calculations in order to take off load from the main processor (CPU). Although the Intel 8088 was in fact a conventional CPU, the PGA's separate on-board processing unit marked an important step towards the development of GPUs as co-processors.

In the following years, more graphics orientated features where added like shading, lighting, rasterization, the depth buffer and color blending. With the introduction of the Open Graphics Library (OpenGL) in 1989 by Silicon Graphics Inc. (SGI), the world's first application programming interface (API) for 2D and 3D graphics was released. OpenGL was designed upon a concept called the graphics pipeline which depicts video processing as data traveling through multiple stages of operations. The pipeline begins at the CPU by sending geometry data together with colors, textures etc., which is then transformed from the initial 3D coordinate space to pixel coordinates. Lighting operations use material colors and textures to shade the incoming triangles which are then rasterized into the frame buffer for display. As the programmer was not able to alter the functionality of the pipeline, it is also known under the term "fixed function pipeline". This processing model determined the design of graphics hardware for more then a decade.

In the mid 1990's NVIDIA, ATI and Matrox started to provide graphic controllers for consumers. Famous computer games like Quake and Doom became popular spurring the gaming industry and the interest in GPUs. However, GPUs at this time were only able to output one pixel per clock cycle and CPUs were able to send more triangles than the GPU could handle. This problem lead graphic hardware vendors to adding more pipelines and eventually more cores to their GPUs in order to enable parallel pixel processing and to increase the throughput.

NVIDIA's release of the GeForce 3 in 2001 marked an important step in the evolution of GPUs by loosening the restrictions of the fixed function pipeline. With the ability of writing small programs

for the GPU, which could operate on the vertices traveling through the pipeline, programmers were given a tool to make limited changes to the behavior of the pipeline. These programs were called vertex shaders and were written in an assembly-like language. One year later, the pixel shader followed running on a separate part of the GPU hardware.

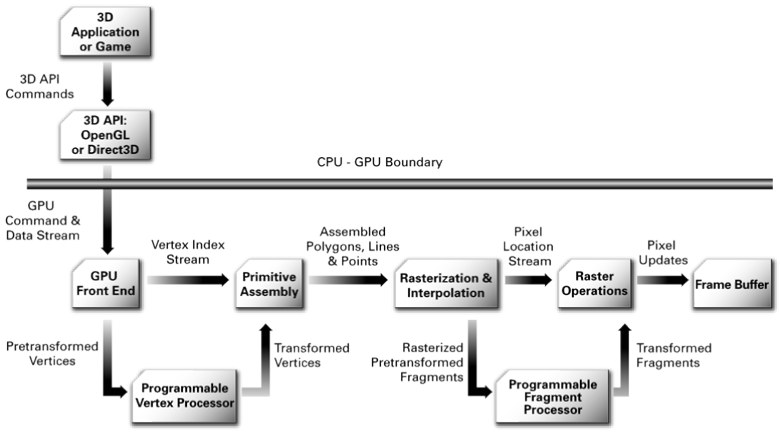


Figure 1: The graphics pipeline with programmable vertex and fragment processor. (Fernando and Kilgard 2003)

With the introduction of the High Level Shading Language (HLSL) with DirectX 9 in 2003, programming GPU hardware became easier than with the previous shaders written in assembler. The first developers started to use the available programmability for non-graphical tasks. The first attempts of GPU computing emerged. A year later, in 2004 Brook and Sh appeared representing the first languages targeting GPGPU computing. Although the workflow of the GPU is still based on a programmable graphics pipeline, the hardware mainly consists of strong, highly parallel floating-point processors with fast memory access.

In 2006 the GeForce 8800 featured the first GPU with a unified programmable processor called a streaming multiprocessor which is used to execute the vertex, pixel and a new geometry shader. The graphics pipeline has become only a software concept used by graphic APIs such as OpenGL and DirectX.

Software GPGPU support was introduced by NVIDIA with the Compute Unified Device Architecture (CUDA). It offers a C like language to create general purpose programs (called kernels) that can be executed on the GPU. ATI followed with the ATI Stream technology and Microsoft introduced compute shaders with DirectX 10.

In 2010 NVIDIA released the first GPU based on their Fermi architecture which was explicitly designed for GPGPU computing. The GTX580 Fermi GPU released later that year contained 16 streaming multiprocessors with 512 CUDA cores and accessed 1.5 GiB GDDR5 RAM with a bandwidth of 192.4 GB/s (NVIDIA Corporation 2013b).

## 1.4 Chapter overview

After this short introduction, chapter 2 will continue with a comprehensive coverage of OpenCL. Beside general information about the API, this chapter focuses on the knowledge required to understand how OpenCL executes kernels on the GPU and what a developer has to pay attention to when programming for graphics hardware. This information is vital for understanding the implemented algorithms in the subsequent chapters.

Chapter 3 will present the first OpenCL implementation of a standard algorithm by tackling multiplications of large floating point matrices. Beside a simply and naive approach, several possibilities

of optimizations are discussed introducing graphic hardware features like shared memory, texture memory and vector types.

Chapter 4 continues with implementations of a prefix sum algorithm (also known as scan) – a ridiculous simple problem for a CPU due to its sequential nature. However, the linearity if the algorithm does not fit the architecture of a GPU. A tree based approach is discussed which is commonly used to partly parallelize linear algorithms.

Chapter 5 focuses sorting as one of the most famous problems in computer science. Several well performing CPU implementations such as C's qsort and C++'s std::sort will be compared with GPU sorting techniques using less popular algorithms such as the bitonic sorting network and radix sort.

Chapter 6 rounds off the thesis with a summary of the covered information, a final conclusion and outlook as well as personal experiences made during the creation of this document.

# 2 OpenCL

## 2.1 What is OpenCL?

OpenCL is specified by the Khronos Group in the OpenCL 1.2 Specification as follows:

> OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms. (Munshi 2012)

The Khronos Group is an industry consortium that maintains OpenCL as an open standard. This means that the Khronos Group offers a specification of the OpenCL API and a detailed description about its functionality and behavior for free. This specification can be downloaded from their Website (Munshi 2012). Maintaining the OpenCL standard consequently means, that the Khronos Group neither provides software development kits (SDKs), drivers that implement OpenCL nor hardware that it can make use of. These concerns are subject to other companies called vendors, which are typically hardware manufacturers providing necessary developing resources and include OpenCL support in their drivers. Examples of such companies are the two famous graphic card vendors NVIDIA and AMD as well as the renowned processor manufacturer Intel.

Unlike other famous APIs of Khronos Group, which are very specific in their usage or the targeted hardware (like the famous 3D API OpenGL used to drive graphic hardware), OpenCL is a general purpose programming framework. It was conceived with universality in mind, offering almost no restrictions on the field of application it may be used in. Portability from its well designed hardware abstraction model, which enables it to run on many different kinds of devices, even ones which may have not been created yet, is one of OpenCL most powerful strengths. Such devices may be classical CPUs and GPUs, but also more uncommon types of hardware like FPGAs (field programmable gate arrays), DSPs (digital signal processors) or Intel's MICs (Many Integrated Core) (Merritt 2011). Moreover, OpenCL may be used to combine multiple available devices into a single heterogeneous platform extending an applications processing resources beyond the bounds of individual pieces of hardware.

In addition to being independent of a specific purpose and decoupled from the underlying hardware, OpenCL is also available across all mayor operations systems including Windows, Linux and Mac OS X.

With the upcoming specification of WebCL (currently available as a working draft), OpenCL will eventually even find its way into Web browsers and the HTML5 technology. Thus making it even independent of an operating system and bringing high performance computing into Web applications.

## 2.2 Components

OpenCL is not only an API. As it allows programs to run on hardware that may have certain restrictions or offer different features than a classical CPU, traditional languages like C++, Java or C# can not be used to write those programs. Therefore, the OpenCL standard includes the

specification of a separate language that is used to write small programs that are executed on a hardware device. These programs are called kernels and are written in the OpenCL C language, which is a restricted version of C99 with extensions for vectorization and handling OpenCL's abstract memory model.

To allow OpenCL to support many different hardware platforms it consists of the following components:

### API

The application programming interface (API) is specified by the Khronos Group, ensuring that every vendor implementation of OpenCL is compatible and exchangeable. The API is provided as a set of C header files that a software developer can download from Khronos' Website. These headers are typically also shipped with an SDK. Khronos additionally specifies a C++ wrapper around the C API. Bindings for other languages exist but they are not maintained by Khronos. The API is used by a conventional program (e.g., written in C++) to run kernels on an OpenCL capable device. This program is called the host application.

### SDK

The software development kit is the collection of tools and resources, a software developer needs to write applications using OpenCL. An SDK is usually provided by a vendor, an implementor of OpenCL. Examples of such SDKs are the NVIDIA CUDA Toolkit and AMD's Accelerated Parallel Processing (APP) SDK. These SDKs typically contain header files to be included by a C/C++ program and static libraries for the linker, which are responsible for binding to the OpenCL driver at runtime. Furthermore, code examples, tutorials, documentation, developing tools etc. may be provided depending on the vendor. With headers and static libraries, an SDK contains all resources necessary to write and compile an OpenCL application.

### OpenCL C language

Kernels are typically written in separate source files using the OpenCL C language. An OpenCL application reads these source files at run time and sends them to the OpenCL compiler to create a binary for one or more available hardware devices where it may be executed. A source file written in the OpenCL C language may consist of several functions, variable definitions, control flow statements, comments, etc., but has to have at least one function prefixed with the `__kernel` attribute. This function may be called by the host application through the OpenCL API and serves as an entry point into a kernel.

### OpenCL C Compiler

OpenCL kernels must be compiled for a specific platform before they can be executed. This compilation is initiated and controlled by the host application through the API. The separate compilation at run time is required to retain OpenCL's portability, as an OpenCL application may be deployed on any kind of machine with a suitable driver. Consequently, the available OpenCL implementation (providing the compiler) and the used hardware device (affecting the compiler's output) may only be determined at runtime.

### Driver

Finally, the driver is OpenCL's core. It implements the OpenCL API and maps the functionality specified in the standard to a vendor's hardware. The host application uses the driver through the API to initialize OpenCL, compile kernels, allocate memory resources, initiate computations and communicate with kernels running on a device. A driver is always specific to a dedicated hardware and must therefore be provided by a vendor. The driver is sometimes also referred to as Installable Client Driver (ICD).

## 2.3 Hardware architectures

One of the greatest advantages of OpenCL and also one of the biggest influences on its design is OpenCL's ability to support devices of many different hardware architectures. These devices can be used through OpenCL by the same common API and may even be used together in a single application or computation, which is sometimes referred to as heterogeneous computing. A closer look into the architectures of the two most common hardware devices, namely CPUs and GPUs, may help the reader to better understand design decisions made by the Khronos Group when OpenCL was conceived. Furthermore, a deeper understanding of graphics hardware will be needed for the implementation chapters of this thesis.

### 2.3.1 Central Processing Unit (CPU)

The CPU is the kind of processors developers are mostly used to. Software written in common languages like C++, Java or C# is directly or indirectly (using a Virtual Machine) executed on this kind of hardware. Figure 2 shows the design of a modern processor by the example of an Intel Ivy Bridge quad-core. One can see the four physical cores of the processor. Thanks to hardware multithreading (Hyper-threading in Intel terms) each physical core can process a pair of threads which are seen as two virtual CPUs by the operation system resulting in 8 available cores for an application. Note the on-chip graphics processor in the Ivy Bridge architecture which can be used as a replacement for a graphics card. It is however not used as general processing device such as the CPU's cores.

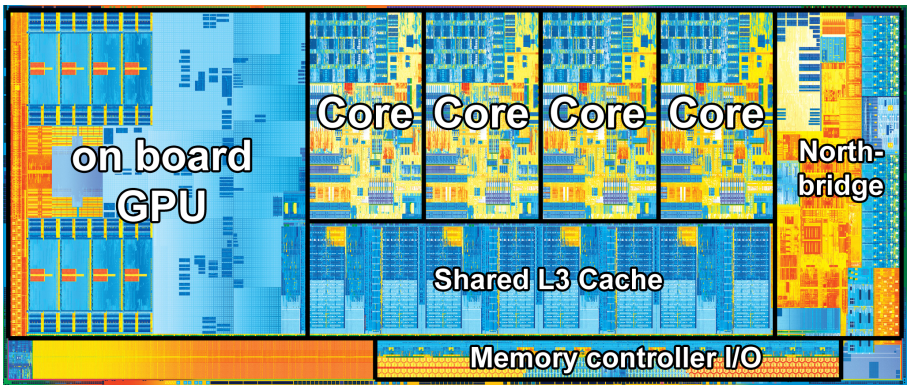


Figure 2: Architectural overview of an Intel Ivy Bridge processor. (Shimpi and Smith 2012)

When it comes down to calculation throughput, and the tackled problem offers some degree of data parallelism (several chunks of data are processed equally), SIMD (Single Instruction Multiple Data) features of the CPU can be used to increase the number of input elements an instruction can process. Intel's Ivy Bridge for example allows the parallel execution (vectorization) of up to eight single precision floating point operations using AVX (Advanced Vector Extensions, the successor of SSE, the Streaming SIMD Extensions) instructions.

Considering memory access, the CPU uses the standard data bus to communicate with the external main memory. As this communication usually takes a while (up to hundreds or even thousands of CPU cycles (Gaster et al. 2011, p.54)), the CPU uses a hierarchical cache system. Each core has it's own level one and level two cache, which have to be kept synchronized to the other cores. All cores share a common level three cache that is further connected to the memory controller.

From a performance concerned developer's perspective, all one has to pay attention to is to have enough parallelism in an application to utilize all cores of the CPU (which is typically 2, 4 or 8 on consumer hardware). If an application has to process large amounts of data using the same operations on elements of the input (e.g. multimedia applications like image or video processing),

vector instructions may be used to speed up execution. Regarding memory, beside small adjustments to improve cache effects, the developer has only limited possibilities to optimize, as the entire cache system is controlled by the CPU and operation system.

### 2.3.2 Graphics Processing Unit (GPU)

When we think of a GPU we think of a highly parallel, graphics orientated processing device. From its introduction with OpenGL in 1989 until the creation of the first GPGPU languages in 2004 (cf. section 1.3), the architecture of a GPU's hardware followed the principles of the graphics pipeline. To fit this purpose, GPUs developed from original desktop processors to (as written in the book Heterogeneous Computing with OpenCL (Gaster et al. 2011)) heavily multithreaded devices employing sophisticated task management directly in hardware. This was necessary to cope with complex task graphs processing vertices, geometry and pixels. These tasks and the data they process are highly parallel and represent an extensive amount of independent work, which is therefore most suitably handled by a device with a great amount of cores employing latency tolerant multithreading[1].


Figure 3: Full chip block diagram of NVIDIA's Kepler GK110 architecture containing 15 streaming multiprocessors (SMX). (NVIDIA Corporation 2013c)

Figure 3 shows the design of a modern GPU by the example of NVIDIA's Kepler architecture. On the top of the block diagram the PCI Express controller is placed. This interface is responsible for the communication of the graphics card with the CPU through the mainboard's PCI Express bus. After data has arrived at the graphics device, it has to be scheduled for processing to one or more of the streaming multiprocessors (SM, or SMX in NVIDIA's Kepler architecture terms). A SM is an aggregation of many cores with additional scheduling logic and therefore representing an extra hierarchical step in a GPU's workflow. This is one of the first, bigger differences in the architecture of modern GPUs when compared with traditional CPUs. SMs are called compute units in OpenCL terms.

Figure 4 shows a detailed view of the structure of a streaming multiprocessors. Once a package

[1]Latency tolerant processing units are characterized by being insensitive to longer lasting memory requests.


Figure 4: Architecture of a Streaming Multiprocessor (SMX) of NVIDIA's Kepler GK110 architecture containing 192 single precision CUDA cores, 64 double precision units, 32 special function units (SFU), and 32 load/store units (LD/ST) (NVIDIA Corporation 2013c).

of work is scheduled to a SM, it is prepared for execution by the Warp Schedulers. A Warp is a block of threads (32 on NVIDIA hardware) executing the same instructions in parallel along cores managed by the Warp Scheduler. This concept is sometimes called SIMT (Single Instruction Multiple Threads). The difference to SIMD (Single Instruction Multiple Data) is that SIMD requires a vector instruction in each thread whereas SIMT only uses scalar code; the vectorization is handled by the hardware (Micikevicius 2012, p.99). A Warp Scheduler is able to manage multiple Warps at the same time (up to 64 on the Kepler GK110 architecture (NVIDIA Corporation 2013c, p.7)), which may be executed interleaved on the cores (similar to the two threads on a hyper-threading Intel core). This allows to hide memory latency (latency tolerance), because when a Warp has to stall because of a memory request, the Warp Scheduler can execute other Warps until the memory request is satisfied.

Another important aspect to notice is that all cores within a SM share the same registers. This plays an important role when choosing the right number of Warps to run concurrently on the SM. Too less Warps may result in unused core cycles whereas a too large number of Warps may request more registers than there are available at the SM. The additionally needed registers are then allocated in the slow global memory. This concern will be discussed later in section 2.6.

The actual work is then performed on the cores itself. The major number of cores (labeled "Core" in green in figure 4) are used for single precision floating point and integer arithmetics. In contrast to older GPUs, modern graphics processing units also include support for double precision floating point arithmetic, which was not needed for graphical calculations in the past, but becomes increasingly important in GPGPU computing. The Kepler architecture serves this need by dedicated cores (labeled "DP Unit"). Additional hardware resources include the load and store unit for register and memory access and the special function unit (labeled "SFU"). On the bottom of figure 4 we find the texture units which are heavily used in 3D graphics. They can also be used by OpenCL via images (more in section 2.7.2).

Finally, beside the way of executing threads on its cores, the memory system is the second component of a GPU that shows significant differences to a CPU. Although using a hierarchical cache system, the GPU lacks caches for each individual core. The first level cache resides within the SM and is used by all cores inside the SM. A specialty of the GPU is the shared memory (also scratch pad memory or local memory), which can be seen as a programmer controllable cache. An OpenCL kernel may use this small block of memory to cache data from the global memory and to limitedly share data with other threads (more on this in section 2.6.2). Furthermore, shared memory is implemented in hardware using multiple memory banks. Special access patterns are required to avoid so-called bank conflicts resulting in slow memory request serialization between threads. Outside the SM (cf. figure 3) the GPU offers a larger level two cache which is then connected to multiple memory controllers to access the global memory. The global memory resides on the graphics card outside the GPU chip (such as the RAM is outside a CPU) and has a larger size of up to several GiB. An important difference of the global memory when compared with the RAM on the mainboard is that RAM can be accessed in almost any pattern[2] without significant performance penalties (apart from cache effects). This is significantly different for a GPU where memory requests should be coalesced[3] and blocked across the threads of a warp to achieve optimal memory system utilization.

Concerning the performance of a GPGPU accelerated application, there are a huge amount of concerns to worry about. Two of them have already been discussed, namely choosing the right number of warps concurrently executing on the SMs and paying attention to memory access. For the latter, the memory access pattern should be coalesced and frequently needed or shared data should be cached in shared memory. Further optimizations exist but would exceed the scope of this thesis. For further reading, NVIDIA has a detailed paper on this subject (Micikevicius 2012).

AMD GPUs are structured similarly as the NVIDIA Kepler architecture presented here. A small difference is that AMDs Ultra-Threaded Dispatch Processor replaces the Warp Schedulers and schedules Wavefronts of 64 threads instead of 32 thread Warps. However, the most fundamental difference is found on instruction level. Whereas NVIDIA's cores only process one instruction at a time across a Warp, AMD processes a full packet of instructions in parallel. These packets are referenced to as Very Long Instruction Words (VLIWs). VLIWs are packets of four (or five on older GPUs) instructions that can be executed in parallel (instruction level parallelism). These packets are generated by the compiler. Due to dependencies between instructions, a VLIW may not always be completely filled and therefore not utilize a VLIW processor completely. Moving instruction level parallelization into the compiler is advantageous for the hardware, as it becomes simpler compared to hardware which tries to parallelize the instruction stream at run time via superscalar or out-of-order execution. Kepler's Warp schedulers for example can issue two instructions in parallel but have to resolve data dependencies between them at runtime via scoreboarding[4] (Kanter

[2] A memory access pattern describes how a program or a part of it organizes multiple and often related memory requests. Examples are random accesses, requests to consecutive addresses (reading a memory block), strided access (e.g., reading only a specific member of each object inside an array.) and accesses to the same address (which are typically cached on a CPU)

[3] A coalesced memory access occurs when all threads of a Warp request data on consecutive memory addresses resulting in a fast memory block transfer instead of individual transfers of smaller chunks. Most GPU hardware architectures have their global memory organized in segments of 64 or 128 bytes where always the full segment has to be transferred regardless of the size of the actually needed data (NVIDIA Corporation 2009, p.13). Coalesced and aligned (guaranteed by compiler) memory access ensures to always transfer the minimum needed segments.

[4] Scoreboarding is a technique of processors where data dependencies of all instructions are tracked and instructions are only issued for execution if all dependencies have been resolved. Therefore, instructions may be reordered and executed in a different order.

2010, p.4). However, static scheduling might not be as efficient as scheduling at runtime as the compiler cannot make any assumptions about the occupancy of different pieces of hardware and thereof resulting delays. For further details about AMD's GPU architectures, David Kanter wrote an article on real world technologies about AMD's Cayman architecture from 2010 with several comparisons to NVIDIA's Fermi architecture (Kanter 2010).

## 2.4 API Overview

The OpenCL API is specified using the C programming language. The Khronos Group defines a set of C functions, types and constants that make up the API a developer can use to write applications using OpenCL. These declarations are cumulated into the cl.h C header file, which can be downloaded from the Khronos' Website and can typically also be found in the include directory of any OpenCL SDK. Although specified in C, OpenCL has an object-oriented design. The UML class diagram in figure 5 shows the objects which can be created, queried and manipulated using corresponding API calls. The Khronos Group also specifies a C++ Wrapper with OpenCL 1.1 built atop the C API which will not be covered in this thesis.

The following sections will give a brief introduction into OpenCL's design and relevant API functions. The sections are based on chapter 2 of the book Heterogeneous Programming with OpenCL (Gaster et al. 2011, p.15-31).

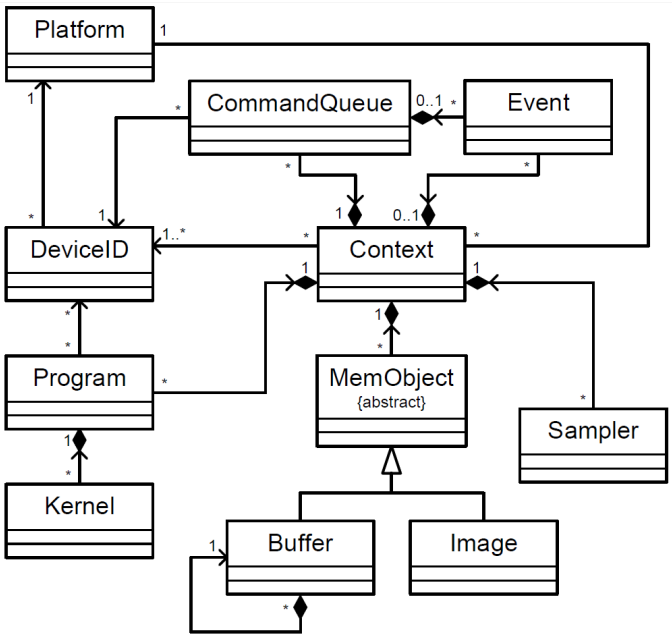


Figure 5: OpenCL class diagram from the OpenCL 1.2 specification. (Munshi 2012)

## 2.5 Platform model

As OpenCL is an open standard it can be adopted by any hardware vendor manufacturing processors powerful enough to support the features specified by the Khronos Group. Many computers contain more than one of such processors; typically at least a CPU and a graphics card, which are both very suitable to be used by OpenCL. However, these devices do not have to share the same OpenCL implementation. On the one hand, someone may have an NVIDIA graphics card using NVIDIA's driver and, on the other hand, have an Intel Core using their OpenCL CPU driver. And then, if somebody has an AMD Radeon graphics card with AMD's Catalysts driver, the CPU can

also be accessed using the same driver. This situation of multiple available OpenCL implementations on the same machine is handled by the platform model. Each OpenCL implementation on the system (a vendor specific driver) represents a platform. The available platforms on the system, which in this context is often called the host, can be queried using the `clGetPlatformIDs` function (for a source code sample see section 2.8). Each platform supports one or more devices (the hardware supported by the driver). Reconsidering the previous scenario, the Intel platform would support one device, which is the CPU, whereas the AMD platform would support both the GPU and the CPU. The available devices of a platform can be retrieved by calling `clGetDeviceIDs` with the desired platform. Devices available on the same platform (e.g., CPU and AMD GPU using the AMD platform) can be used together which is referred to as heterogeneous computing. A device itself consists of multiple compute units being functionally independent from each other (cf. the SMs of a GPU). Each compute unit itself is then eventually divided into processing elements (cf. the cores inside a SM). The number of available compute units on a device and other information like memory capacities can be queried using `clGetDeviceInfo`. Figure 6 visualizes the relation of platforms, devices, compute units and processing elements. (Gaster et al. 2011, pp.19)

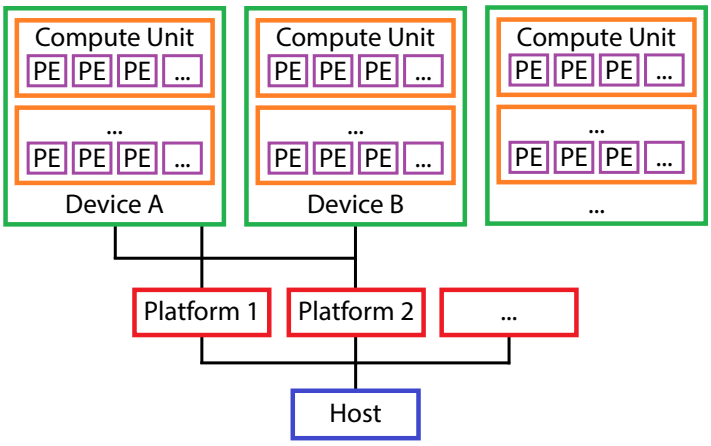


Figure 6: OpenCL's platform model.

## 2.6 Execution model

The central component of OpenCL's execution model are kernels. But before a kernel can be executed on a device, a number of additional resources have to be set up. After a device has been selected, a context has to be created that manages objects like memory buffers or kernels associated with the device. A context is created by calling `clCreateContext` with one or more devices of the same platform as arguments. Additionally, properties may be passed to the function, like a platform to restrict the context to this platform allowing multiple contexts for different platforms to coexist, or a handle to an OpenGL (Open Graphics Library, a 3D API) context to enable graphics interoperability. (Gaster et al. 2011, p.22)

While a context holds objects shared between the host and one or more devices, OpenCL offers an additional mechanism for host device communication. To request action from a device, a command queue needs to be created within a context and associated with this device. This is done by calling `clCreateCommandQueue` with the context and device as arguments. Once a command queue is created, the host can request actions from the device by enqueuing operations like memory buffer writes/reads or kernel executions. Per default these operations are executed asynchronously to the host application, although explicit synchronization can be enforced by calling `clFinish` on the command queue, which blocks until all pending operations have been completed. (Gaster et al. 2011, p.22, 23,26)

### 2.6.1 Creating kernels

After the OpenCL execution environment has been successfully prepared, all that is left is the actual piece of code to execute. OpenCL C code is usually written into text files just as every other code. But in contrast to the host application, OpenCL C code is not compiled before an OpenCL application is deployed. OpenCL C code is compiled at runtime via corresponding API functions. This peculiarity is required as different vendors use individual binary codes that may be executed on their devices. Therefore, an application has to read the OpenCL C code from the source files at runtime and pass it to the `clCreateProgramWithSource` function which returns a program object representing the given piece of code. This program can be compiled and linked for one or more devices by calling `clBuildProgram`. As compile errors can occur (return value is not equal to `CL_SUCCESS`) the compile log may be retrieved by a subsequent call to `clGetProgramInfo`. After the program has been successfully built, kernels can be extracted (a program may contain multiple kernels). This is done by calling `clCreateKernel` with the compiled program and the name of a function with the `__kernel` attribute, which will be used as entry point into the kernel. The obtained kernel is ready for being enqueued on a command queue to a device, the kernel was compiled for. (Gaster et al. 2011, p.26, 27)

### 2.6.2 Kernel execution

A kernel can be seen as C function that is executed in parallel on the device's processing resources by the OpenCL runtime. As elaborated in section 2.3, these resources may be organized very differently. Therefore, OpenCL uses an abstraction model describing the dimensions of the queued work and how it is split up into small pieces. (Gaster et al. 2011, p.16)

When the host application wants to enqueue a kernel, it has to specify the size of the problem as a so-called n-dimensional range (NDRange). This range can be seen as a one-, two- or three-dimensional grid of indexes which typically represent the size of the input or output of a kernel. Each element of the grid represents a single kernel invocation and is called a work item. Work items are identified by their position inside the grid which is named global id. The size of the grid is called the global work size. (Gaster et al. 2011, p.18)

For example: A matrix multiplication with an output matrix of $M * N$ elements might enqueue a kernel with a two-dimensional range of $M * N$ work items. Each work item would then use it's indices to determine the row and column of the input matrices and the position of the calculated output value.

To closer adapt to GPU hardware architectures, the NDRange can be equally divided into work groups. Work items inside a work group benefit from having a dedicated shared memory block available. Access to this memory can be synchronized between the work items of a work group providing a basic method of communication. To address a work item inside a work group, each work item has an additional local id and a group id. The extents of a work group must be of the same dimension as the NDRange and the sizes in each dimension, called local work sizes, must be powers of two and evenly divide the global work sizes. (Gaster et al. 2011, p.18) The maximum number of work items in a work group is implementation dependent and can be queried by a call to `clGetDeviceInfo` with the `CL_DEVICE_MAX_WORK_GROUP_SIZE` constant as argument. Figure 7 shows the relation of the NDRange, work groups and work items.

A kernel is executed on a command queue by calling `clEnqueueNDRangeKernel`. Additional important arguments beside the kernel and the command queue are the NDRange's dimensions, the global work size and the local work size. The local work size is optional. Before a kernel is executed, arguments may be specified for the `__kernel` function according to the function's signature. These arguments are passed to the device when the kernel starts executing and can be accessed via the `__kernel` function's parameters. Arguments are set from the host by calling `clSetKernelArg` with the argument's index in the kernel function's signature as well as the size and value of the argument.

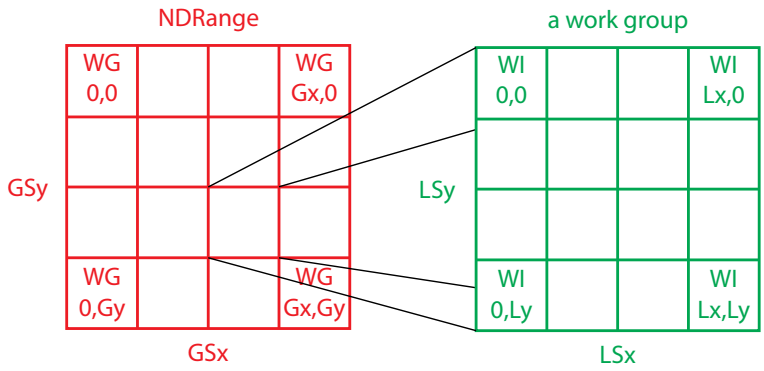


Figure 7: The two-dimensional NDRange consisting of $GSx * GSy$ work groups. Each work group consists of $LSx * LSy$ work items. A work item has two indices in each dimension, the local index (relative to the work group) and a global index (relative to the NDRange), as well as the indices of the group it is part of. The local work sizes are LSx and LSy whereas the global work sizes are $GSx * LSx$ and $GSy * LSy$.

### 2.6.3 Kernel code

The language used to write kernels is OpenCL C, which is derived from C99 and extended by OpenCL specifics like vector data types, built in functions and additional keywords. A kernel function must have the `__kernel` attribute and must have `void` as it's return type. The function's name identifies the kernel to the host. A kernel may have parameters which are declared in the function's signature. An example kernel function for a matrix multiplication is given in listing 1.

```
1 __kernel void matrixMultiplication(int n, int m, __global const float* matrixA,
2     __global const float* matrixB, __global float* output) {
3   // kernel code ...
4 } // matrixMultiplication
```

Listing 1: An example of a kernel function's signature.

## 2.7 Memory

Many applications often have to work with large amounts of data. As the data that can be passed to a kernel function by value is restricted to a small size, OpenCL offers the possibility to allocate larger blocks of memory represented as a memory object which is valid inside a context and moved to a device when necessary. At the very top, OpenCL defines two basic types of memory, buffers and images. Buffers can be seen as a continuous block of memory equivalent to an array in C allocated using `malloc`. Images in contrast are abstract memory objects which can make use of device specific hardware acceleration and optimization when being read from or written to (e.g., texture units on a GPU). Images do not have to be supported by a device. (Gaster et al. 2011, pp.23)

### 2.7.1 Buffers

Buffers are created by the host application via a call to `clCreateBuffer`. In addition to the context and buffer size, several flags may be specified. With these flags (beside further options) the buffer can be declared as read or write only, or OpenCL can be instructed to use or copy a provided block of host memory. Once a buffer is created, read and write operations may be enqueued on a command queue to transfer a buffer's content (or parts of it) between the host and

a device. `clEnqueueReadBuffer` and `clEnqueueReadBuffer` are the corresponding API functions for transferring memory. When needed by a kernel, buffers have to be set as arguments to a kernel and can be accessed via a pointer from inside the kernel function. (Gaster et al. 2011, p.24)

### 2.7.2 Images

Images abstract the underlying storage implementation to allow a device to use hardware optimizations. GPUs for example might use their texture units for improved read and write access as well as providing hardware accelerated interpolation between pixel values of the image. As images do not need to be represented as continuous arrays of a specified data type, a format descriptor is used to determine the size and data type of a pixel value of the image. Images are created by a call to `clCreateImage2D` or `clCreateImage3D` (unified to `clCreateImage` in OpenCL 1.2) with similar parameters as the buffer creating functions. Additionally, the image format descriptor has to be specified and an optional padding between the rows of the image can be set. Images are read and written by the host application via calls to `clEnqueueReadImage` and `clEnqueueWriteImage`. Like buffers, images can be set as arguments to a kernel, but cannot be accessed via pointers inside the kernel function. Special built-in functions like `read_imagef` have to be used in combination with a sampler object which determines out-of-bounds image access, interpolation, and access coordinate handling. (Gaster et al. 2011, pp.25)

### 2.7.3 Memory model

As memory subsystems are highly diverse between different kinds of hardware devices, OpenCL defines an abstract memory model giving vendors the possibility to map OpenCL's abstract types of memory to physical memory units on a device. OpenCL defines four types of memory which are shown in figure 8:

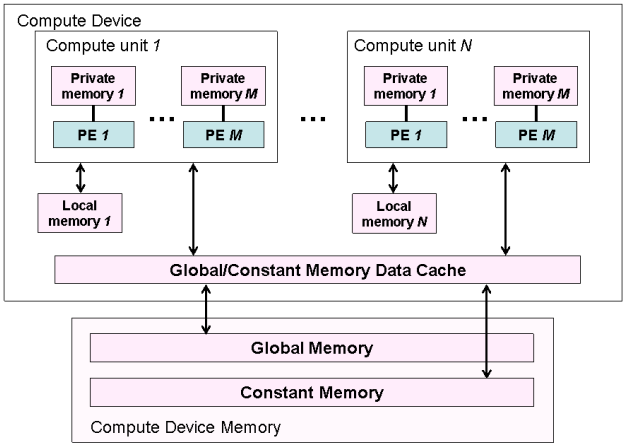


Figure 8: Memory model as defined by the OpenCL 1.2 specification.(Munshi 2012)

**1. Global memory**
Global memory is the largest of all memory regions and visible to all compute units on a device. This memory region is used whenever data is copied to or from a device. Buffers and images are allocated in this memory. All pointer variables pointing into global memory have to be prefixed with the `__global` attribute inside a kernel. (Gaster et al. 2011, p.29)

**2. Constant memory**
Constant memory is typically implemented as part of the global memory and is used to store read only data which is needed by all work items. All pointer variables pointing into constant memory have to be prefixed with the `__constant` attribute inside a kernel. (Gaster et al. 2011, p.30) Newer types of hardware may provide special caches for this kind of memory (cf. the read-only data cache in NVIDIA's Kepler architecture in figure 3).

**3. Local memory**
Local memory (or shared memory) is a small block of memory which is shared between all work items inside a work group. As access to local memory is very fast, it can be seen as a programmer controlled cache to global memory. Local memory is often used for preloading data needed by all work items of a work group or to synchronize work items across a work group. All pointer variables pointing into local memory have to be prefixed with the `__local` attribute inside a kernel. Local memory can be allocated statically inside a kernel by declaring a fixed sized array like `__local float sharedArray[32];` or dynamically with a call to `clSetKernelArg` providing the size of the local memory block requested and `nullptr` as pointer to the arguments value. (Gaster et al. 2011, p.30)

**4. Private memory**
Private memory belongs to each work item itself and is typically stored in registers (on a GPU) except a work group requests more memory than the available register file. In this case, the spilled private memory is mapped to global memory. Private memory cannot be allocated, but pointers to local variables may be used to pass values by reference to functions called by the kernel function. Pointers to variables pointing to private memory can be prefixed with the `__private` attribute inside kernels, but they do not have to as `__private` is the default pointer type. (Gaster et al. 2011, p.30)

## 2.8 Code sample

The following piece of code in listing 2 shows a minimal OpenCL application with a kernel that calculates the sum of two input vectors. To keep the code simple, no error handling is performed, only the error code is retrieved. The codes in the subsequent implementation chapters will completely go without error handling or error code retrieval.

```
1  #include <CL/cl.h>
2  #include <iostream>
3
4  int main(int argc, char* argv[]) {
5    cl_int error;
6
7    cl_platform_id platform;
8    error = clGetPlatformIDs(1, &platform, nullptr);
9
10   cl_device_id device;
11   error = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, nullptr);
12
13   cl_context context = clCreateContext(nullptr, 1, &device, nullptr, nullptr, &error);
14   cl_command_queue queue = clCreateCommandQueue(context, device, 0, &error);
15
16   const char* source = ""
17   "__kernel void vectorAdd(__global float* a, __global float* b, __global float* c)"
18   "{"
19   " size_t id = get_global_id(0);"
20   " c[id] = a[id] + b[id];"
21   "}";
22   cl_program program = clCreateProgramWithSource(context, 1, &source, nullptr, &error);
23   error = clBuildProgram(program, 1, &device, "", nullptr, nullptr);
24
25   cl_kernel kernel = clCreateKernel(program, "vectorAdd", &error);
26
27   const size_t N = 1024;
28   float* a = new float[N];
29   float* b = new float[N];
30   float* c = new float[N];
31
32   for (size_t i = 0; i < N; i++) {
33     a[i] = i;
```

```
34      b[i] = N - i;
35    } // for
36
37    cl_mem bufferA = clCreateBuffer(context, CL_MEM_READ_ONLY,  N * sizeof(float), nullptr, &error);
38    cl_mem bufferB = clCreateBuffer(context, CL_MEM_READ_ONLY,  N * sizeof(float), nullptr, &error);
39    cl_mem bufferC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, N * sizeof(float), nullptr, &error);
40
41    error = clEnqueueWriteBuffer(queue, bufferA, false, 0, N * sizeof(float), a, 0, nullptr, nullptr);
42    error = clEnqueueWriteBuffer(queue, bufferB, false, 0, N * sizeof(float), b, 0, nullptr, nullptr);
43
44    error = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufferA);
45    error = clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufferB);
46    error = clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufferC);
47    error = clEnqueueNDRangeKernel(queue, kernel, 1, nullptr, &N, nullptr, 0, nullptr, nullptr);
48
49    error = clEnqueueReadBuffer(queue, bufferC, true, 0, N * sizeof(float), c, 0, nullptr, nullptr);
50
51    for (size_t i = 0; i < N; i++)
52      std::cout << c[i] << ", ";
53
54    delete[] a;
55    delete[] b;
56    delete[] c;
57
58    error = clReleaseMemObject(bufferA);
59    error = clReleaseMemObject(bufferB);
60    error = clReleaseMemObject(bufferC);
61    error = clReleaseKernel(kernel);
62    error = clReleaseProgram(program);
63    error = clReleaseCommandQueue(queue);
64    error = clReleaseContext(context);
65    error = clReleaseDevice(device);
66
67    return 0;
68  } // main
```

Listing 2: A minimalistic working OpenCL application which calculates the sum vector of two input vectors.

The first step in every OpenCL program is to choose a platform. As platforms correspond to the available OpenCL implementations on the system, multiple platforms can be available. The example code in listing 2 queries the first available OpenCL platform by calling `clGetPlatformIDs` with 1 and a pointer to a variable able to hold one platform id as arguments. The third parameter could be used to retrieve the number of actually available platforms.

After a platform has been chosen, we can continue by choosing a device for this platform which works analogously as selecting a platform. In addition to specifying the platform to query devices for, OpenCL also allows us to define the type(s) of devices we would like to get. In this case, we would like to have the first available GPU on the first available platform by specifying `CL_DEVICE_TYPE_GPU` when calling `clGetDeviceIDs`[5]. To be able to allocate OpenCL resources and enqueue operations on a device, we now have to create a context and a command queue using `clCreateContext` and `clCreateCommandQueue`. Further arguments to both API functions allow specifying further properties which can be looked up in the corresponding documentation.

The program is now ready to create a kernel. OpenCL kernel source code is typically placed in a file and read at runtime. For simplicity, the kernel code of this example is emplaced into the host code as a string literal. The OpenCL kernel simply queries the id of the current work item, loads the values from the buffers `a` and `b` at this index and writes the sum of them to buffer `c` at the same index.

To create an actual kernel object within the context, we first have to create an OpenCL program of the source code which is done by calling `clCreateProgramWithSource` and passing the source code as argument. Furthermore, the program has to be compiled for the chosen device. By calling `clBuildProgram` the source code of the program object is compiled and linked into an executable for the specified device. This step can take up to several seconds. Similar as building programs in other languages, compiling may fail. In this case, an error code is returned and

[5] Calling `clGetDeviceIDs` with a device type as argument may return no devices. The Intel OpenCL platform for example only supports CPUs and therefore does not provide a GPU device. Appropriate error handling is necessary in real world applications.

a compiler log could be retrieved using `clGetProgramBuildInfo`. On a successful build, the kernel object (which can be seen as entry point into the program) can be finally retrieved by calling `clCreateKernel` and specifying the compiled program as well as the name of the `__kernel` function.

Before we can execute the kernel, we have to allocate storage for the input and output data. Three arrays are allocated on the host with a size of `N`. The first two are filled with data. To move the data to the GPU, three buffer objects have to be created using `clCreateBuffer` having the same size as the corresponding host arrays. Note that the two input buffers are created as read only and the output buffer as write only by setting appropriate flags. To initiate the copy operations from the host to the GPU device, buffer write operations have to be enqueued on the command queue using `clEnqueueWriteBuffer`. Arguments are the command queue, the buffer to write to, a boolean specifying if the write operation should be blocking, the offset and length of the region to be written to, a pointer to the host memory from which should be read and several further arguments concerning events which will not be covered. By specifying `false` for the blocking write parameter, the write operation is execute asynchronously to the host program. Therefore, the provided host pointers (`a` and `b`) have to be available at least until the first blocking OpenCL call. Asynchronous memory transfers are advantages, as OpenCL may decide itself when to perform the actual copy. Furthermore, multiple copies may be executed concurrently and even beside kernel executions.

Now it is time to setup and execute the kernel. The three buffer objects, on which the kernel will operate, are set as the kernel's arguments via `clSetKernelArg`. The second parameter specifies the index of the argument in the order they appear in the kernel function's signature. Finally, the kernel can be enqueued to be executed using `clEnqueueuNDRangeKernel`. The problem is one-dimensional with a global work size of `N`. The local work size is not specified and therefore determined by OpenCL. Similar to the buffer writes, the kernel enqueue operation is also executed asynchronously. However, the arguments do not have to be available anymore after `clSetKernelArg` returns as they are copied by the OpenCL implementation. Furthermore, as the kernel depends on the buffer objects as inputs, the kernel is not executed until all buffer operations (memory transfers) on the inputs have completed.

The last step is to read the results back to host memory via `clEnqueueReadBuffer` having the same arguments as the corresponding write function. This time however, the synchronization boolean is set to `true`. Therefore, the function call blocks until the read operation has finished. Afterwards the result on the host may be further processed (e.g., printed).

When OpenCL resources are no longer needed, they should be released using their corresponding `clRelease*` API functions.

# 3 Matrix multiplication

Multiplying large matrices is often part of heavy scientific calculations and therefore an important building block in every mathematics library. Unfortunately, with increasing size the computation of the product becomes desperately slow, due to an asymptotic runtime complexity of $\mathcal{O}(n^3)$ for a naive attempt. Therefore, several improvements have been tried to speed up the calculation. Strassen's algorithm for example achieves a runtime complexity of $\mathcal{O}(n^{\log_2 7}) = \mathcal{O}(n^{2.807})$ (Strassen 1969) which is faster than the naive version, but still does not compute results within a satisfying time for larger matrixes. Nonetheless, modern computing systems can take advantage of one important aspect of the matrix multiplication which is it's embarrassing parallel nature. Each element of the output matrix can be computed completely independent (cf. figure 9). This potential may not only be used by today's CPU's vector extensions like SSE [1] or AVX [2]. Especially a GPU can make excellent use of the large number of independent pieces of work to perfectly utilize its hundreds or thousands of cores.

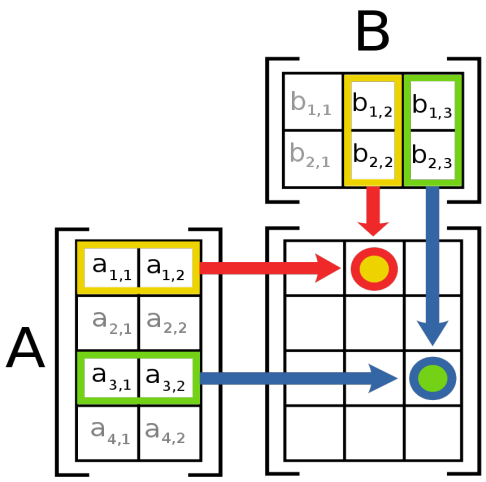


Figure 9: Principle of the matrix multiplication (StefanVanDerWalt 2010).

Beside the implementation and optimization of the matrix multiplication on GPUs, this chapter will also present performance data of GPUs collected by an appropriate profiling tool (AMD's CodeXL will be used). This allows a better understanding about an algorithm's performance and how bottlenecks can be found and eliminated.

The provided benchmarks were created on a notebook with medium hardware components. Details can be found in the appendix section B. Furthermore, the host code may use the utility functions `roundToMultiple` and `roundToPowerOfTwo` for aligning input data to appropriate size. Their respective definitions can be found in appendix chapter A.

This chapter will take a look at different implementations of the matrix multiplication for both, the CPU and the GPU. For simplicity, square matrices are used and stored as one dimensional arrays in row major order.

[1] Streaming SIMD Extensions, a vector instruction set extension by Intel supporting 128 bit wide registers to process 4 single precision floats in parallel.
[2] Advanced Vector Extensions, Intel's latest vector instruction set extension featuring 256 bit wide registers to process 8 single precision floats in parallel.

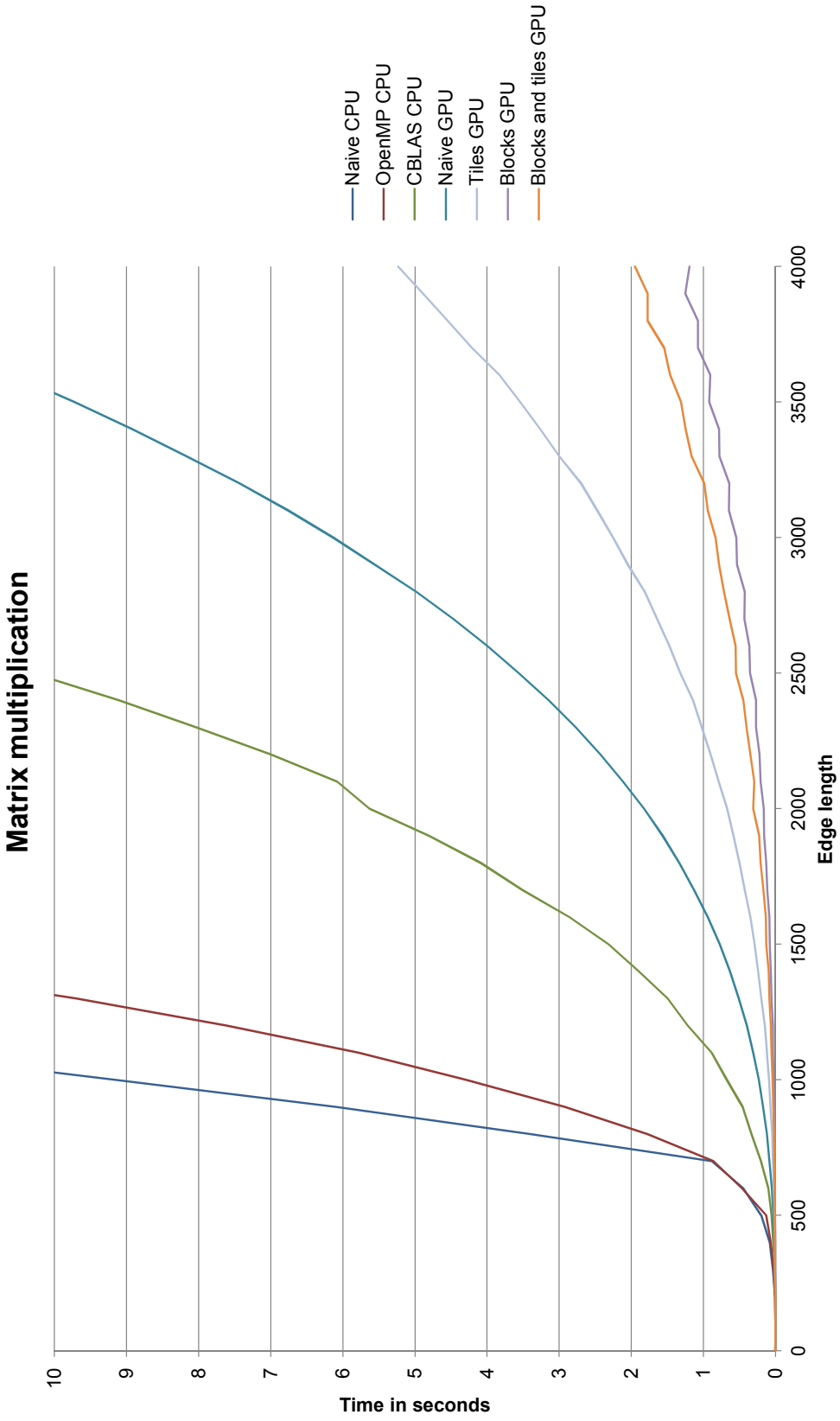


Figure 10: Benchmark of several square matrix multiplication implementations.

## 3.1 CPU Implementation

Although lots of optimized algorithms and their implementations exist, we will first have a look at a simple and naive implementation as the one given in listing 3.

```
1 void multNaiveCPU(float* a, float* b, float* c, size_t n) {
2   for (size_t row = 0; row < n; row++) {
3     for (size_t col = 0; col < n; col++) {
4       c[row * n + col] = 0;
5       for (size_t i = 0; i < n; i++)
6         c[row * n + col] += a[row * n + i] * b[i * n + col];
7     } // for
8   } // for
9 } // multNaiveCPU
```

Listing 3: A simple C++ implementation of a square matrix multiplication for the CPU.

The `naiveCPU` function takes two pointers to the memory where the two input matrices are stored, another pointer to already allocated memory where the output is written to and a final parameter `n` giving the size of the input and output matrices. The algorithm is simple. Every element of the output matrix has to be calculated, therefore the outer two `for` loops run through all of these elements. For each output element the dot product of the `row` in matrix `a` and the `col` in matrix `b` are calculated.

The performance of this implementation is conceivably slow as one can see in figure 10. Multiplying matrices with an edge length of up to 700 elements can be done in a short time (below one second), but larger matrices require a huge amount of time which is unsatisfying in most cases. Also a multithreaded version of the code provided in listing 3 (e.g., by using OpenMP and place a `# pragma omp parallel for` above the outer for loop) does not bring substantial improvements. The optimized Fortan77 BLAS library (*BLAS (Basic Linear Algebra Subprograms)* 2011) achieves a significant better performance when compared with the simple CPU implementation provided here.

## 3.2 Naive GPU implementation

A working GPU implementation can be directly derived from the simple CPU implementation given in the previous section 3.1. As we can see in figure 9, each output element is calculated independently and the calculation of an output element corresponds to the loop body of the second `for` loop in the CPU implementation in listing 3. Therefore, both outer `for` loops offer a good starting point for parallelization.

The naive GPU implementation uses the same principle as the initial CPU version. But before we can jump into GPU code, some setup on the host is required which is shown in listing 4. To simplify the code, context and command queue have already been created and are provided as input to the `multNaiveGPU` function.

```
1  void multNaiveGPU(float* a, float* b, float* c, cl_uint n,
2      cl_context context, cl_command_queue queue, cl_kernel kernel, size_t workGroupSize) {
3    size_t bufferSize = n * n * sizeof(float);
4
5    cl_mem aBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY, bufferSize, nullptr, nullptr);
6    cl_mem bBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY, bufferSize, nullptr, nullptr);
7    cl_mem cBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bufferSize, nullptr, nullptr);
8
9    clEnqueueWriteBuffer(queue, aBuffer, false, 0, bufferSize, a, 0, nullptr, nullptr);
10   clEnqueueWriteBuffer(queue, bBuffer, false, 0, bufferSize, b, 0, nullptr, nullptr);
11
12   clSetKernelArg(kernel, 0, sizeof(cl_mem), &aBuffer);
13   clSetKernelArg(kernel, 1, sizeof(cl_mem), &bBuffer);
14   clSetKernelArg(kernel, 2, sizeof(cl_mem), &cBuffer);
15   clSetKernelArg(kernel, 3, sizeof(cl_uint), &n);
16   size_t adjustedWS = roundToMultiple(n, workGroupSize);
17   size_t globalWS[] = { adjustedWS, adjustedWS };
18   size_t localWS[] = { workGroupSize, workGroupSize };
```

```
19    clEnqueueNDRangeKernel(queue, kernel, 2, nullptr, globalWS, localWS, 0, nullptr, nullptr);
20
21    clEnqueueReadBuffer(queue, cBuffer, true, 0, bufferSize, c, 0, nullptr, nullptr);
22
23    clReleaseMemObject(aBuffer);
24    clReleaseMemObject(bBuffer);
25    clReleaseMemObject(cBuffer);
26  } // multNaiveGPU
```

Listing 4: Host code for a matrix multiplication implementation using OpenCL.

For the three pointers to arrays holding the two input matrices and the output matrix, OpenCL buffers have to be created. On creation, additional flags may be provided to allow the underlying OpenCL implementation to optimize memory access to these buffers. Therefore, the two input matrix buffers are set to `CL_MEM_READ_ONLY` and the output buffer to `CL_MEM_WRITE_ONLY`. These flags only affect access to the memory object from the kernel code and do not restrict the host application to read or write buffers. After the buffers have been created, a write operation is enqueued on the command queue to the GPU for both input matrices. Note, that the third parameter of the calls to `clEnqueueWriteBuffer` specifies whether the function blocks until the write operation has completed or not. This parameter is set to `false` allowing OpenCL to transfer the memory blocks asynchronously to the running host application and even in parallel. Before the kernel can be executed, the two input buffers, the output buffer and the size of the matrix are set as arguments to the `__kernel` function. Furthermore, the global and local work size for the kernel invocation have to be determined. The global work size must be a multiple of the used work group size (local work size). The chosen size of the work groups affects GPU utilization as a too small size may lead to wasted processing power on the SMs and decreases latency tolerance. A too large work group size may cause the kernel to fill up the available register file on the SM. In this case local data is moved to global memory causing a serious drop in performance. Finding the optimal work group size is hardware dependent and often boils down to experimentation. If the kernels are small, it is usually a good idea to try the highest possible work group size (256 on the used GPU). Note, that choosing a work group size which does not evenly divide the global work size and therefore rounding the global work size to the next multiple of the work group size causes the GPU to execute the kernel for additional unneeded work items. Concerning the matrix multiplication example, the kernel would be executed for a bigger matrix with a size evenly dividable by the work group size. This has to be kept in mind when, e.g., accessing buffers inside the kernel. After global and local work sizes have been chosen, the kernel can be enqueued to the command queue to be executed by calling `clEnqueueNDRangeKernel`. The call immediately returns as the kernel is executed asynchronously when the command queue is flushed (`clFlush`) or a blocking command is enqueued, which is the case on the subsequent read operation. The call to `clEnqueueReadBuffer` reads the result from the device to the host application's array. Note, that the third parameter is set to `true` indicating a blocking operation. All previously enqueued commands are ensured to be executed and finished before the read operation takes place, which returns after all data has been successfully copied to client memory.

The last missing component of the naive GPU implementation is the OpenCL kernel itself, shown in listing 5.

```
1   __kernel void MultNaive(__global float* a, __global float* b, __global float* c, uint n) {
2     uint col = get_global_id(0);
3     uint row = get_global_id(1);
4
5     if (row >= n || col >= n)
6       return;
7
8     float sum = 0.0f;
9     for (uint i = 0; i < n; i++)
10      sum += a[row * n + i] * b[i * n + col];
11
12    c[row * n + col] = sum;
13  } // MultNaive
```

Listing 5: OpenCL Kernel code calculating a single element of the output matrix per work item.

The `__kernel` function closely resembles the loop body of the second `for` loop of the CPU implementation in listing 3. The purpose of a single invocation of this kernel is to calculate one

element of the output matrix. By using OpenCL's built-in function `get_global_id`, with the dimension as argument, the work item can retrieve it's position inside the NDRange which equals the size of the output matrix (plus extra space due to work group size rounding). Therefore, the retrieved position has to be checked against the matrices' size as the NDRange may be larger than the original matrix. If the coordinates identify a valid matrix position, the output element is determined by calculating the dot product of the `row` in matrix `a` and the `col` in matrix `b` and eventually written to the output matrix buffer `c`.

When run on the GPU, the performance of this naive OpenCL implementation is amazingly fast, when we have a look at the benchmark in figure 10. For $800 \times 800$ elements, the size where the CPU implementation's run time started to exceed several seconds, the GPU variant still runs at around 200 milliseconds. Additionally, the curve raises far slower than the CPU's one.

However, when we have a look at the measured performance counters of the naive approach in table 1 we notice several things: Despite the perfect 100 % occupancy of the kernel (as a result of no local memory usage and low vector register footprint), the algorithm performs a relatively high number of fetch instructions (cf. the ALU fetch ratio of 2.5). This can also be seen at the time the fetch unit was busy, which was 99.9 %. This means the algorithm is highly memory bound. Furthermore, elements of the input matrices are accessed multiple times (edge length of output matrix times). Although this explains the nice cache behavior of 57.5 % hits, here is definitely room for improvement. Another approach to reduce the total number of memory requests would be to query larger elements per work item. This would lead to fewer but larger memory requests which might be handled more efficiently by the memory system. Furthermore, fewer work items would be required.

## 3.3 Optimization using local memory

Based on the performance analysis of the naive GPU approach, the first and most obvious optimization is to reduce the number of memory requests to global memory. Ideally, each input matrix element should be accessed only once. This scenario cannot be achieved, but the number of memory requests to a single item can be reduced to one inside smaller subregions by subdividing the input matrices into tiles as shown in figure 11. Instead of each work item accessing all needed input elements like in the naive version, each work item of a work group (which is equally sized as a tile) copies one element of a tile from each input matrix to local memory. After the two tiles have been copied completely, the work items than access the corresponding elements from the tiles in local memory (avoiding redundant global memory requests), multiply and add the values to the output sum of each work item (element of the output matrix). Then the next two tiles are loaded. The algorithm repeats until all tiles have been processed.

The host code for this algorithms looks very similar to the one of the naive version. The major difference is that the size of the input matrix must now be a multiple of the tile size. To avoid complex handling of smaller matrices in the kernel, we now create fitting matrices beforehand. This can be achieved by allocating slightly bigger buffers (whose size is a multiple of the tile size) and initialize them with zeros. The actual input matrices can then be copied into the upper left rectangle of the larger buffer. OpenCL offers corresponding functions to achieve this task. Listing 6 presents the host code for the optimized matrix multiplication using tiles in local memory.

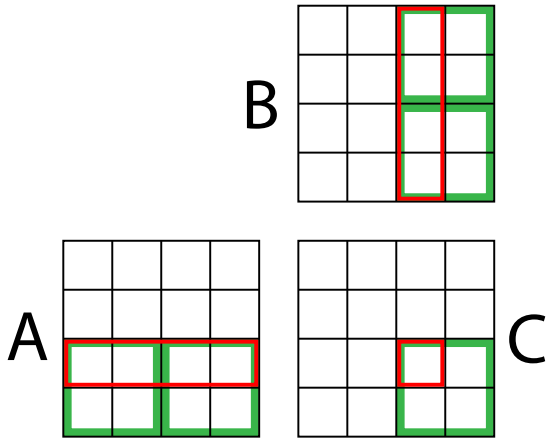


Figure 11: Optimization of the matrix multiplication by subdivision into tiles. Here $4 \times 4$ matrices are divided into tiles of $2 \times 2$ work items.

```
1  #define TILE_SIZE 16
2
3  void multTilesGPU(float* a, float* b, float* c, cl_uint n,
4      cl_context context, cl_command_queue queue, cl_kernel kernel) {
5    cl_uint adjustedSize = roundToMultiple(n, TILE_SIZE);
6    size_t bufferSize = adjustedSize * adjustedSize * sizeof(float);
7    cl_mem aBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY, bufferSize, nullptr, nullptr);
8    cl_mem bBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY, bufferSize, nullptr, nullptr);
9    cl_mem cBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bufferSize, nullptr, nullptr);
10
11   size_t bufferOffset[] = {0, 0, 0};
12   size_t hostOffset[]   = {0, 0, 0};
13   size_t sizes[]        = {n * sizeof(float), n, 1};
14
15   if (adjustedSize != n) {
16     float zero = 0.0f;
17     clEnqueueFillBuffer(queue, aBuffer, &zero, sizeof(float), 0, bufferSize, 0, nullptr, nullptr);
18     clEnqueueFillBuffer(queue, bBuffer, &zero, sizeof(float), 0, bufferSize, 0, nullptr, nullptr);
19     clEnqueueWriteBufferRect(queue, aBuffer, false, bufferOffset, hostOffset, sizes,
20         adjustedSize * sizeof(float), 0, n * sizeof(float), 0, a, 0, nullptr, nullptr);
21     clEnqueueWriteBufferRect(queue, bBuffer, false, bufferOffset, hostOffset, sizes,
22         adjustedSize * sizeof(float), 0, n * sizeof(float), 0, b, 0, nullptr, nullptr);
23   } else {
24     clEnqueueWriteBuffer(queue, aBuffer, false, 0, bufferSize, a, 0, nullptr, nullptr);
25     clEnqueueWriteBuffer(queue, bBuffer, false, 0, bufferSize, b, 0, nullptr, nullptr);
26   } // if
27
28   clSetKernelArg(kernel, 0, sizeof(cl_mem), &aBuffer);
29   clSetKernelArg(kernel, 1, sizeof(cl_mem), &bBuffer);
30   clSetKernelArg(kernel, 2, sizeof(cl_mem), &cBuffer);
31   clSetKernelArg(kernel, 3, sizeof(cl_uint), &adjustedSize);
32   size_t globalWS[] = { adjustedSize, adjustedSize };
33   size_t localWS[] = { TILE_SIZE, TILE_SIZE };
34   clEnqueueNDRangeKernel(queue, kernel, 2, nullptr, globalWS, localWS, 0, nullptr, nullptr);
35
36   if (adjustedSize != n)
37     clEnqueueReadBufferRect(queue, cBuffer, true, bufferOffset, hostOffset, sizes,
38         adjustedSize * sizeof(float), 0, n * sizeof(float), 0, c, 0, nullptr, nullptr);
39   else
40     clEnqueueReadBuffer(queue, cBuffer, true, 0, bufferSize, c, 0, nullptr, nullptr);
41
42   clReleaseMemObject(aBuffer);
43   clReleaseMemObject(bBuffer);
44   clReleaseMemObject(cBuffer);
45 } // multTilesGPU
```

Listing 6: Host code for a matrix multiplication implementation using OpenCL, where the input and output matrices' sizes are rounded up to be a multiple of `TILE_SIZE`.

At first and most importantly a `TILE_SIZE` hast to be defined. This parameter defines the work group size and the size of the square tiles that are allocated in local memory and used to cache values from global memory. As the maximum allowed work group size is 256 on the used GPU, tiles of $16 \times 16$ elements fit perfectly. The buffers are again allocated with `clCreateBuffer`. If the adjusted buffer size is larger than the actual matrix size, the input buffers are initialized

to zero using `clEnqueueFillBuffer`. Afterwards, each input matrix is copied to a rectangular region (as large as the input matrix) in the top left corner of the allocated buffer using `clEnqueueWriteBufferRect`. For a detailed explanation of the arguments have a look at the corresponding entry in the OpenCL specification (Munshi 2012, p.76). Apart form the kernel's arguments, which equal the ones of the naive version, the work group size must be equal to the defined `TILE_SIZE`. Finally, the results are retrieved from the output buffer following the same principle as when the inputs where copied to the input buffers.

Listing 7 presents the kernel code of the optimized matrix multiplication using tiles in local memory.

```
1  #define TILE_SIZE 16
2
3  __kernel void MultTiles(__global float* a, __global float* b, __global float* c, uint n) {
4    uint col     = get_global_id(0);
5    uint row     = get_global_id(1);
6    uint localX  = get_local_id(0);
7    uint localY  = get_local_id(1);
8
9    uint posA    = row * n + localX;
10   uint posB    = localY * n + col;
11   uint stepA   = TILE_SIZE;
12   uint stepB   = TILE_SIZE * n;
13   uint endA    = posA + n;
14   uint tilePos = localY * TILE_SIZE + localX;
15
16   float sum = 0.0f;
17
18   while (posA < endA) {
19     __local float tileA[TILE_SIZE * TILE_SIZE];
20     __local float tileB[TILE_SIZE * TILE_SIZE];
21
22     tileA[tilePos] = a[posA];
23     tileB[tilePos] = b[posB];
24     barrier(CLK_LOCAL_MEM_FENCE);
25
26     for (uint k = 0; k < TILE_SIZE; k++)
27       sum += tileA[localY * TILE_SIZE + k] * tileB[k * TILE_SIZE + localX];
28     barrier(CLK_LOCAL_MEM_FENCE);
29
30     posA += stepA;
31     posB += stepB;
32   } // while
33
34   c[row * n + col] = sum;
35 } // MultTiles
```

Listing 7: OpenCL Kernel code calculating a single element of the output matrix per wok item, where the input matrixes are split into tiles which are cached in local memory. (Tompson and Schlachter 2012)

When compared with the first, naive OpenCL kernel, the optimization using locally cached tiles achieves a speedup of 2.63 for a problem size of 4000 as we can see in figure 10. We can also see further improvements on the performance counters in table 1. Although the general thread management overhead remains constant (equal number of work items, work groups and therefore wavefronts) and the kernel occupancy still stays at perfect 100 %, we notice a major decrease in general ALU instructions. Moreover and most importantly, fetch instructions to global memory have been reduced to 1/16 (due to the used `TILE_SIZE` of 16). Despite this reduction, the eventually fetched data from global memory could only be reduced by a factor of 2.4 due to worse cache behavior. In addition to fewer memory requests and ALU instructions a third and probably unnoticed optimization occurred: As `TILE_SIZE` is a compile time constant, the compiler is now able to unroll and vectorize the inner for loop which results in more independent instructions available to the compiler to generate better filled VLIWs. This can be seen by an almost doubling of the ALU packing counter's value.

## 3.4 Optimization using vectorization

Instead of using local memory to decrease the number of redundant memory requests, this approach focuses on a better utilization of the memory system by using vector types and multiple fetches and writes per work item. Therefore, one work item does not only fetch one element but a block of elements. The access pattern of the algorithm itself works conceptually in the same way to the local tile version of the previous section 3.3. The tiles are called blocks and each work item fetches two whole blocks itself. No local memory is used, as the fetched blocks are stored in each work item's registers. Each work item then also produces an output block which is written to global memory.

The host code for this matrix multiplication version is almost identical to the one using tiles cached in local memory in listing 6. The differences are shown in listing 8.

```
1  #define BLOCK_SIZE 4
2    ...
3    cl_uint adjustedSize = roundToMultiple(n, BLOCK_SIZE);
4    ...
5    size_t adjustedWS = roundToMultiple(adjustedSize, workGroupSize * BLOCK_SIZE);
6    size_t globalWS[] = { adjustedWS / BLOCK_SIZE, adjustedWS / BLOCK_SIZE };
7    size_t localWS[] = { workGroupSize, workGroupSize };
8    ...
```

Listing 8: Differences of the host code for the matrix multiplication using vector types when compared with the one using tiles cached in local memory from listing 6.

Most importantly, a `BLOCK_SIZE` is defined, that sets the size of the block of elements that is fetched from global memory per work item. This parameter must be one of the OpenCL supported vector type widths (which are 2, 4, 8 or 16). A larger `BLOCK_SIZE` means more and bigger elements are fetched from global memory (better bus utilization) per work item and less work items will be needed to run the calculation (lesser thread management overhead). Furthermore, multiple independent global memory fetches can be pipelined. Additionally, as the number of elements a work item calculates increments, the number of independent instructions (instruction level parallelism) of the kernel increases allowing better ALU utilization through mostly full VLIWs (super scalar execution of the kernel). However, as a consequence, the required number of registers per work item rises which allows fewer work groups to run concurrently on a SM. For further reading, Vasily Volkov held a great talk about this kind of tweaking (Volkov 2010). The determination of the optimal `BLOCK_SIZE` is probably different for various GPUs and subject to corresponding benchmarks. The implementation in listing 9 will use a value of four.

Listing 9 presents the kernel code of the optimized matrix multiplication using vector types and multiple reads/writes to global memory per work item.

```
1  #define BLOCK_SIZE 4
2
3  __kernel void MultBlocks(__global float4* a, __global float4* b, __global float4* c, uint n) {
4    uint col = get_global_id(0);
5    uint row = get_global_id(1);
6    uint n4  = n / BLOCK_SIZE;
7
8    if (row >= n4 || col >= n4)
9      return;
10
11   float4 sum0 = (float4)(0.0f);
12   float4 sum1 = (float4)(0.0f);
13   float4 sum2 = (float4)(0.0f);
14   float4 sum3 = (float4)(0.0f);
15
16   for (uint i = 0; i < n4; i++) {
17     float4 blA0 = a[(row * BLOCK_SIZE + 0) * n4 + i];
18     float4 blA1 = a[(row * BLOCK_SIZE + 1) * n4 + i];
19     float4 blA2 = a[(row * BLOCK_SIZE + 2) * n4 + i];
20     float4 blA3 = a[(row * BLOCK_SIZE + 3) * n4 + i];
21     float4 blB0 = b[(i * BLOCK_SIZE + 0) * n4 + col];
22     float4 blB1 = b[(i * BLOCK_SIZE + 1) * n4 + col];
23     float4 blB2 = b[(i * BLOCK_SIZE + 2) * n4 + col];
24     float4 blB3 = b[(i * BLOCK_SIZE + 3) * n4 + col];
25
26     sum0.x += blA0.x * blB0.x + blA0.y * blB1.x + blA0.z * blB2.x + blA0.w * blB3.x;
```

```
27      sum0.y += blA0.x * blB0.y + blA0.y * blB1.y + blA0.z * blB2.y + blA0.w * blB3.y;
28      sum0.z += blA0.x * blB0.z + blA0.y * blB1.z + blA0.z * blB2.z + blA0.w * blB3.z;
29      sum0.w += blA0.x * blB0.w + blA0.y * blB1.w + blA0.z * blB2.w + blA0.w * blB3.w;
30      sum1.x += blA1.x * blB0.x + blA1.y * blB1.x + blA1.z * blB2.x + blA1.w * blB3.x;
31      sum1.y += blA1.x * blB0.y + blA1.y * blB1.y + blA1.z * blB2.y + blA1.w * blB3.y;
32      sum1.z += blA1.x * blB0.z + blA1.y * blB1.z + blA1.z * blB2.z + blA1.w * blB3.z;
33      sum1.w += blA1.x * blB0.w + blA1.y * blB1.w + blA1.z * blB2.w + blA1.w * blB3.w;
34      sum2.x += blA2.x * blB0.x + blA2.y * blB1.x + blA2.z * blB2.x + blA2.w * blB3.x;
35      sum2.y += blA2.x * blB0.y + blA2.y * blB1.y + blA2.z * blB2.y + blA2.w * blB3.y;
36      sum2.z += blA2.x * blB0.z + blA2.y * blB1.z + blA2.z * blB2.z + blA2.w * blB3.z;
37      sum2.w += blA2.x * blB0.w + blA2.y * blB1.w + blA2.z * blB2.w + blA2.w * blB3.w;
38      sum3.x += blA3.x * blB0.x + blA3.y * blB1.x + blA3.z * blB2.x + blA3.w * blB3.x;
39      sum3.y += blA3.x * blB0.y + blA3.y * blB1.y + blA3.z * blB2.y + blA3.w * blB3.y;
40      sum3.z += blA3.x * blB0.z + blA3.y * blB1.z + blA3.z * blB2.z + blA3.w * blB3.z;
41      sum3.w += blA3.x * blB0.w + blA3.y * blB1.w + blA3.z * blB2.w + blA3.w * blB3.w;
42    } // for
43
44    uint posC = (row * BLOCK_SIZE) * n4 + col;
45
46    c[posC + 0 * n4] = sum0;
47    c[posC + 1 * n4] = sum1;
48    c[posC + 2 * n4] = sum2;
49    c[posC + 3 * n4] = sum3;
50  } // MultBlocks
```

Listing 9: OpenCL Kernel code calculating a $4 \times 4$ block of elements of the output matrix per work item using four float4 vectors, where the input matrices are accessed directly from global memory. (Advanced Micro Devices, Inc. 2013a)

Compared with the naive GPU version, the block version achieves a speedup of 11.95 at a problem size of 4000 when looking at the chart in figure 10. The kernel also performs 4.54 times faster than the optimized version using locally cached tiles in section 3.3. Also, the performance counters in table 1 show interesting changes. Most notable is the change of global work size from $4000 \times 4000$ to $1008 \times 1008$ which accounts mostly for the achieved speedup. Although the kernel now processes 16 elements instead of one, the number of instructions executed per work item only increased by a factor of 1.19 (from 20019 to 23782). Furthermore, the number of total fetch instructions per work item remained constant (a work item now issues four instructions but fetches four times more data per instruction) resulting in equal management work for the fetch unit (which is still busy at 99.4 %) but higher throughput. Additionally, as each work item processes four independent output elements, the amount of independent instructions increased leading to an almost doubling in ALU packing performance. The major drawback of this version is that the high number of needed registers (VGPRs) limits the amount of wavefronts that can be concurrently executed on a SM. This can be seen at the lower occupancy of only 37.5 %. As a result, memory latency may not be hidden well enough (by scheduling wavefronts not waiting for memory requests). However, this problem is specific to the number of available registers which is likely to increase as GPGPU computing becomes an important design factor for GPU hardware vendors.

## 3.5 Combining both approaches

The optimization attempts presented sections 3.3 and 3.4 focus on two aspects.

1. The first one tries to reduce the number of redundant global memory requests by subdividing the input matrices into tiles which are read once by each work group and cached in local memory.

2. The second approach tries to improve the general memory system and ALU utilization by fetching larger chunks of memory per work item and processing them using vector types.

The final kernel presented in this chapter will combine both ideas. As a consequence, the matrices will be subdivided in two levels. On the lower level, elements will be organized into small blocks which are always loaded completely by a work item. On the higher level, the blocks are grouped into tiles which can be cached in local memory and are always retrieved by a whole work group.

The host code for this matrix multiplication version is almost identical to the one using tiles cached in local memory in listing 6. The only differences are shown in listing 10.

```
#define TILE_SIZE 16
#define BLOCK_SIZE 4
  ...
  cl_uint adjustedSize = roundToMultiple(n, BLOCK_SIZE * TILE_SIZE);
  ...
  size_t globalWS[] = { adjustedSize / BLOCK_SIZE, adjustedSize / BLOCK_SIZE };
  ...
```

Listing 10: Differences of the host code for the matrix multiplication using vector types and locally cached tiles when compared with the one using only tiles cached in local memory in listing 6.

The multiple of `BLOCK_SIZE * TILE_SIZE`, the matrix sizes are round up to, is required to ensure coalesced memory access and to avoid complicated bounds checking (resulting in branch divergence) inside the kernel. This value is significantly higher than in the previous approaches and may account for a notable memory overhead on larger matrices (e.g., a matrix of size 4033 is rounded up to 4096 creating 512127 padding elements which are 3 % of the processed data).

Listing 11 shows the kernel code of the optimized matrix multiplication using vector types, multiple reads/writes to global memory per work item and locally cached tiles.

```
#define TILE_SIZE 16
#define BLOCK_SIZE 4

__kernel void MultBlocksAndTiles(__global float4* a, __global float4* b,
    __global float4* c, uint n) {
  uint col     = get_global_id(0);
  uint row     = get_global_id(1);
  uint localX  = get_local_id(0);
  uint localY  = get_local_id(1);
  uint n4      = n / BLOCK_SIZE;

  uint posA    = (row * BLOCK_SIZE) * n4 + localX;
  uint posB    = (localY * BLOCK_SIZE) * n4 + col;
  uint stepA   = TILE_SIZE;
  uint stepB   = TILE_SIZE * BLOCK_SIZE * n4;
  uint endA    = posA + n4;
  uint tilePos = localX + (localY * BLOCK_SIZE) * TILE_SIZE;

  float4 sum0  = (float4)(0.0f);
  float4 sum1  = (float4)(0.0f);
  float4 sum2  = (float4)(0.0f);
  float4 sum3  = (float4)(0.0f);

  while (posA < endA) {
    __local float4 aTile[TILE_SIZE * TILE_SIZE * BLOCK_SIZE];
    __local float4 bTile[TILE_SIZE * TILE_SIZE * BLOCK_SIZE];

    aTile[tilePos + 0 * TILE_SIZE] = a[posA + 0 * n4];
    aTile[tilePos + 1 * TILE_SIZE] = a[posA + 1 * n4];
    aTile[tilePos + 2 * TILE_SIZE] = a[posA + 2 * n4];
    aTile[tilePos + 3 * TILE_SIZE] = a[posA + 3 * n4];
    bTile[tilePos + 0 * TILE_SIZE] = b[posB + 0 * n4];
    bTile[tilePos + 1 * TILE_SIZE] = b[posB + 1 * n4];
    bTile[tilePos + 2 * TILE_SIZE] = b[posB + 2 * n4];
    bTile[tilePos + 3 * TILE_SIZE] = b[posB + 3 * n4];

    barrier(CLK_LOCAL_MEM_FENCE);

    for (uint k = 0; k < TILE_SIZE; k++) {
      float4 blA0 = aTile[k + (localY * BLOCK_SIZE + 0) * TILE_SIZE];
      float4 blA1 = aTile[k + (localY * BLOCK_SIZE + 1) * TILE_SIZE];
      float4 blA2 = aTile[k + (localY * BLOCK_SIZE + 2) * TILE_SIZE];
      float4 blA3 = aTile[k + (localY * BLOCK_SIZE + 3) * TILE_SIZE];
      float4 blB0 = bTile[localX + (k * BLOCK_SIZE + 0) * TILE_SIZE];
      float4 blB1 = bTile[localX + (k * BLOCK_SIZE + 1) * TILE_SIZE];
      float4 blB2 = bTile[localX + (k * BLOCK_SIZE + 2) * TILE_SIZE];
      float4 blB3 = bTile[localX + (k * BLOCK_SIZE + 3) * TILE_SIZE];

      sum0.x += blA0.x * blB0.x + blA0.y * blB1.x + blA0.z * blB2.x + blA0.w * blB3.x;
      sum0.y += blA0.x * blB0.y + blA0.y * blB1.y + blA0.z * blB2.y + blA0.w * blB3.y;
      sum0.z += blA0.x * blB0.z + blA0.y * blB1.z + blA0.z * blB2.z + blA0.w * blB3.z;
      sum0.w += blA0.x * blB0.w + blA0.y * blB1.w + blA0.z * blB2.w + blA0.w * blB3.w;
      sum1.x += blA1.x * blB0.x + blA1.y * blB1.x + blA1.z * blB2.x + blA1.w * blB3.x;
      sum1.y += blA1.x * blB0.y + blA1.y * blB1.y + blA1.z * blB2.y + blA1.w * blB3.y;
      sum1.z += blA1.x * blB0.z + blA1.y * blB1.z + blA1.z * blB2.z + blA1.w * blB3.z;
```

```
56        sum1.w += blA1.x * blB0.w + blA1.y * blB1.w + blA1.z * blB2.w + blA1.w * blB3.w;
57        sum2.x += blA2.x * blB0.x + blA2.y * blB1.x + blA2.z * blB2.x + blA2.w * blB3.x;
58        sum2.y += blA2.x * blB0.y + blA2.y * blB1.y + blA2.z * blB2.y + blA2.w * blB3.y;
59        sum2.z += blA2.x * blB0.z + blA2.y * blB1.z + blA2.z * blB2.z + blA2.w * blB3.z;
60        sum2.w += blA2.x * blB0.w + blA2.y * blB1.w + blA2.z * blB2.w + blA2.w * blB3.w;
61        sum3.x += blA3.x * blB0.x + blA3.y * blB1.x + blA3.z * blB2.x + blA3.w * blB3.x;
62        sum3.y += blA3.x * blB0.y + blA3.y * blB1.y + blA3.z * blB2.y + blA3.w * blB3.y;
63        sum3.z += blA3.x * blB0.z + blA3.y * blB1.z + blA3.z * blB2.z + blA3.w * blB3.z;
64        sum3.w += blA3.x * blB0.w + blA3.y * blB1.w + blA3.z * blB2.w + blA3.w * blB3.w;
65      } // for
66      barrier(CLK_LOCAL_MEM_FENCE);
67
68      posA += stepA;
69      posB += stepB;
70    } // while
71
72    uint posC = (row * BLOCK_SIZE) * n4 + col;
73    c[posC + 0 * n4] = sum0;
74    c[posC + 1 * n4] = sum1;
75    c[posC + 2 * n4] = sum2;
76    c[posC + 3 * n4] = sum3;
77  } // MultBlocksAndTiles
```

Listing 11: OpenCL Kernel code calculating a $4 \times 4$ block of elements of the output matrix per work item using `float4` vectors, where the input matrices are split into tiles of blocks which are cached in local memory. (Advanced Micro Devices, Inc. 2013a)

Although this approach can be seen as an optimization of the block only version from section 3.4 by using locally cached tiles, this kernel performs, probably contrary to expectations, slower than the original block only version (cf. benchmark chart, figure 10). The reason for this at first glance confusing behavior is easily detected when we have a look at the corresponding performance counters in table 1. First of all, the number of work items is equal to the block only approach and therefore the number of resulting wavefronts the hardware has to process stays the same. The higher number of index calculations of the kernel becomes notable in the huge increase of processed ALU instructions per work item which is almost twice as much as in the block only variant (44214 vs 23782). However, the use of local memory as a cache works successfully as can be seen at the number of fetches to global memory which has been reduced by a factor of `TILE_SIZE` and the total number of fetched KiB which has been decreased to a fourth. So far, the counter values indicate an increase in performance. The big penalty, however, comes from the high register and moreover the local memory consumption of the kernel which only allows one work group (= four wavefronts) to run concurrently on a SM (the local memory usage of 32 KiB fills the entire available scratch pad memory). As a result, the kernel can only run 12. 5% of the hardware's supported number of wavefronts concurrently (occupancy). Memory request latency will therefore account for most of the consumed runtime. By reducing the work group size, the local memory consumption can be decreased. This would allow more wavefronts to run concurrently on the cost of more redundant global memory requests. However, this idea has not proven to be beneficial during benchmarks. Nevertheless, as hardware advances, more registers and local memory will be available in the future, maybe allowing this kernel to develop its full potential.

| Kernel | Global size | Work group | Time | Local m. | VGPRs | Occ. | Wavefronts |
|---|---|---|---|---|---|---|---|
| Mult | $4000 \times 4000$ | $16 \times 16$ | 14390 | 0 | 6 | 100 | 84000 |
| MultLocal | $4000 \times 4000$ | $16 \times 16$ | 5132 | 2048 | 5 | 100 | 84000 |
| MultBlock | $1008 \times 1008$ | $16 \times 16$ | 1074 | 0 | 16 | 37.5 | 15914 |
| MultBlockLocal | $1008 \times 1008$ | $16 \times 16$ | 1823 | 32768 | 31 | 12.5 | 15914 |

| Kernel | ALU insts | Fetches | ALU fetch ratio | Writes | LDS fetchs | LDS writes |
|---|---|---|---|---|---|---|
| Mult | 20019 | 8000 | 2.5 | 1 | 0 | 0 |
| MultLocal | 7767 | 500 | 15.5 | 1 | 4000 | 500 |
| MultBlock | 23782 | 7918 | 3.0 | 4 | 0 | 0 |
| MultBlockLocal | 44214 | 503 | 87.9 | 4 | 18352 | 2011 |

| Kernel | ALU busy | ALU packing | Fetch unit busy | Cache hit | Fetch size |
|---|---|---|---|---|---|
| Mult | 62.5 | 36.0 | 99.9 | 57.5 | 23149005 |
| MultLocal | 68.1 | 59.3 | 17.5 | 7.3 | 9729495 |
| MultBlock | 74.6 | 62.8 | 99.4 | 31.2 | 21510158 |
| MultBlockLocal | 69.8 | 69.6 | 4.2 | 37.1 | 5030126 |

**Global size** The global work size as specified to `clEnqueueNDRangeKernel`.
**Work group** The size of the work groups as specified to `clEnqueueNDRangeKernel`.
**Time** The time spent executing the kernel on the GPU in milliseconds. This value is measured using the on-board GPU clock.
**Local m.** The amount of local memory needed by a work group in bytes.
**VGPRs** The number of Vector General Purpose Registers needed per work item.
**Occ.** The kernel occupancy. This is an estimate of the number of in-flight wavefronts on a compute unit as a percentage of the theoretical maximum number of wavefronts that the compute unit can support.
**Wavefronts** The total number of wavefronts that where executed on all compute units. A wavefront (on an AMD GPU) is a packet of 64 hardware threads running simultaneously on a SM.
**ALU insts** The average number of ALU instructions executed per work item.
**Fetches** The average number of fetch instructions to the global memory executed per work item.
**Writes** The average number of write instructions to the global memory executed per work item.
**LDS fetches** The average number of fetch instructions to the local memory (local data share) executed per work item.
**LDS writes** The average number of write instructions to the local memory (local data share) executed per work item.
**ALU busy** The percentage of GPU time ALU instructions are processed.
**ALU fetch ratio** The ratio of ALU instructions to fetch instructions.
**ALU packing** This value indicates how well the shader compiler packs the scalar and vector instructions of the OpenCL program into 5-way VLIW instructions for the GPU.
**Fetch unit busy** The percentage of GPU time the fetch unit is busy.
**Cache hit** The percentage of fetch instructions to the global memory which could be served from the cache.
**Fetch size** The total number of kilobytes eventually fetched from global memory.

Table 1: Selected performance counters of all matrix multiplication kernels gathered using AMD CodeXL's GPU profiler (Advanced Micro Devices, Inc. 2013c). Some values have been rounded. The descriptions of the counters are based on the tool tips provided by the CodeXL Visual Studio integration.

## 3.6 Further implementations

Besides the presented kernels, further approaches have been implemented and benchmarked which have not been covered. However, some experiences may help future developers.

**1D vs 2D problem dimensions**
For the naive approach from section 3.2, it does not make a difference whether the kernel is run with a 1D or 2D problem dimension. 2D seems more natural for matrices but does not affect performance.

**Using images and the texture units**
Using images instead of buffers as memory objects (without further specialization of access patterns) does not increase performance. Images may benefit from special hardware capabilities (e.g. texture units on a GPU provide different cache mechanisms).

**Vector types vs arrays**
Using vector types over an array (e.g. `float4 a` over `float a[4]`) does make a big difference. The blocked approach from section 3.4 has been implemented using arrays instead of vector types. Performance dropped by a factor of four. The reason for this is the size of a GPU's vector registers which is exactly large enough to hold a `float4` value. Components of the vector type can only be addressed at compile time (via, e.g., `.y`). Arrays however can be addressed at runtime and therefore the compiler has to ensure that every array element is addressable. As a result, an array of float values is placed into a consecutive block of registers where each element occupies only the first vector component of each register. This ensures fast array element access but on the cost of quadrupled variable size.

Furthermore, several libraries exist supporting GPGPU accelerated matrix multiplication.

**clAmdBlas (Advanced Micro Devices, Inc. 2013b)**
A BLAS (Basic Linear Algebra Subprograms) library provided by AMD achieving similar results when compared with the blocks kernel of section 3.4.

**cuBlas (NVIDIA Corporation 2013a)**
A BLAS (Basic Linear Algebra Subprograms) library provided by NVIDIA shipped with their CUDA Toolkit. It uses CUDA as GPU Computing language and runs on NVIDIA GPUs only.

## 3.7 Summary and conclusion

Multiplying matrices is an important part of many scientific applications and should therefore focus on performance. Although several extremely tuned CPU libraries have established themselves, there are still very few GPU accelerated routines available. We have seen that the matrix multiplication is embarrassingly easy to implement for the GPU as the parallelism is already part of the problem's nature. However, there is still as much room for improvements on a GPU as there is on the CPU's side. By the help of profiling information (performance counters) bottlenecks of the presented implementations could be detected and tackled.

We have seen the importance of avoiding redundant global memory requests by using local memory as a programmer controlled cache. Furthermore, independent instructions are beneficial to allow for better ALU utilization by instruction level parallelism. Additionally, vector types and larger memory fetches and processed elements per work item reduce the general thread management overhead and may achieve a better utilization of the global memory subsystem. Finally, a fast GPU accelerated matrix multiplication could be implemented beating the already optimized BLAS library (CPU) by factor of 36 (43.49 vs 1.21 seconds). This result definitely shows the potential of modern GPU hardware.

# 4 Prefix sum

The all-prefix-sums operation (also referred to as prefix sum or simply scan) is a simple calculation which is most often found as part of larger and more complex routines. It serves as fundamental building block for implementing well-known algorithms such as radix sort (covered in section 5.2.3), stream compacting, minimum spanning tree and many more (cf. Blelloch's papers on prefix sum (Blelloch 1989) (Blelloch 1990) and the GPU Gems 3 book section 39.1 (Harris, Sengupta, and Owens 2008)).

The all-prefix-sums operation uses a binary, associative operator $\oplus$ with identity element $I$ to transform an input array of $n$ elements

$$[a_0, a_1, \ldots, a_{n-1}]$$

into an output array of $n$ elements where

a) each output element is the sum of all elements preceding the corresponding input element. This is known as an exclusive scan. (Harris, Sengupta, and Owens 2008)

$$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2})]$$

b) each output element is the sum of all elements preceding the corresponding input element and the input element itself. This is known as an inclusive scan. (Harris, Sengupta, and Owens 2008)

$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})]$$

Contrary to the matrix multiplication of chapter 3, the all-prefix-sums operation does not offer similarly trivial parallelism. Scanning an array is naturally sequential with a complexity of $\mathcal{O}(n)$. Although each output element could be calculated independently to gain parallelism, a lot of redundant work would be necessary raising the overall complexity to $\mathcal{O}(n^2)$ with the last element still taking $\mathcal{O}(n)$ time to calculate. This chapter will focus on efficient and parallel implementations of exclusive scan (except otherwise noted) using addition as operator on an input array of signed 32 bit integers. This chapter is orientated towards the Parallel Prefix Sum article from GPU Gems 3 (Harris, Sengupta, and Owens 2008).

## 4.1 CPU Implementation

Implementing a sequential, single threaded scan for the CPU is simple. The first output element is initialized to zero. We than loop over the remaining output elements and set each one to the value of its predecessor plus the corresponding input element. Listing 12 presents an example of a scan implementation.

```
1  void scanCPU(int* data, int* result, size_t n) {
2    result[0] = 0;
3    for (size_t i = 1; i < n; i++)
4      result[i] = result[i - 1] + data[i - 1];
5  } // scanCPU
```

Listing 12: A simple C++ implementation of an exclusive scan for the CPU.

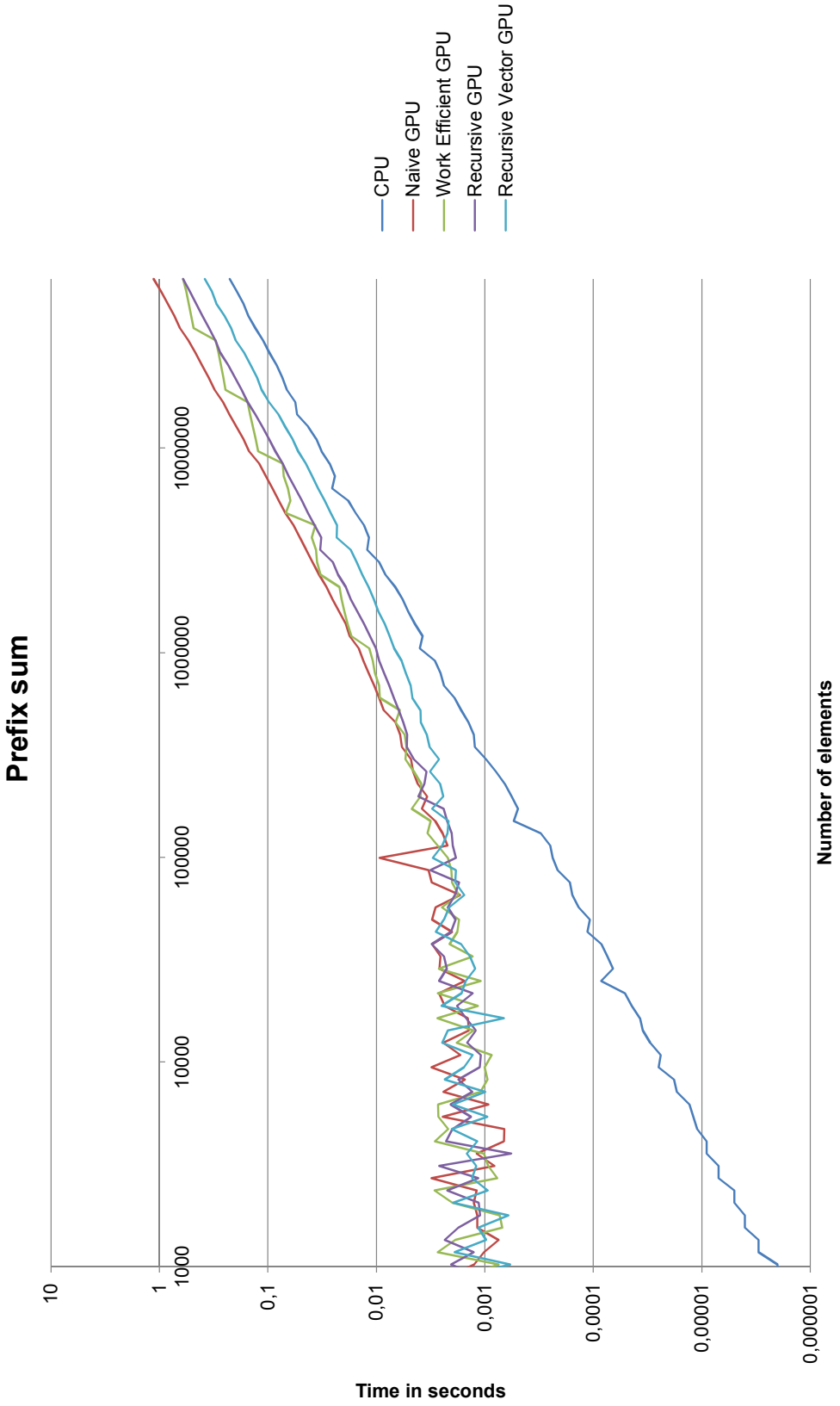


Figure 12: Benchmark of several prefix sum implementations, where both axes are of logarithmic scale.

This code performs exactly $n - 1$ additions which is the minimum number of additions required to produce an exclusive scan of an array with $n$ elements. Concerning the following parallel scan implementations later in this chapter, we would like them to be work-efficient. This means that the parallel implementation should have the same work complexity of $\mathcal{O}(n)$ as the sequential one.

> A parallel computation is work-efficient if it does asymptotically no more work (add operations, in this case) than the sequential version (Harris, Sengupta, and Owens 2008).

The benchmark of this algorithm in figure 12 confirms the linearity of scan. Furthermore, we can also see that scanning is a quite fast operation (when, e.g., being compared to the matrix multiplication of chapter 3). The CPU implementation manages to scan $2^{26}$ elements (256 MiB of data) in 225 ms.

## 4.2 Naive GPU implementation

The first GPU implementation is base on the article Data Parallel Algorithms (Hillis and Guy L. Steele 1986). The discussed approach is to compute an inclusive (!) scan is shown in figure 13. The algorithm uses several passes to compute the final output array in place. In each pass the value of a predecessor is added to an element. The offset from each element to its predecessor is determined by the pass index and is $2^{d-1}$ where d is the number of the pass starting with 1.

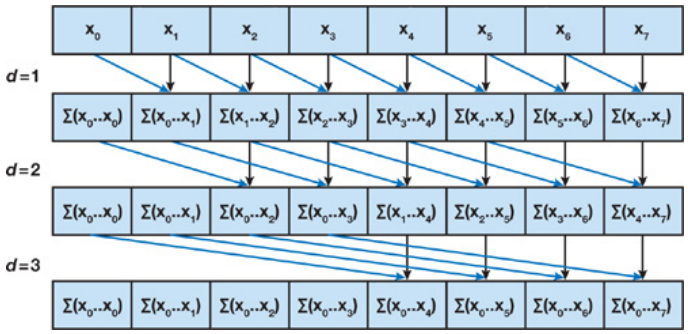


Figure 13: A naive approach for a parallel scan. (Harris, Sengupta, and Owens 2008)

By now the algorithm assumes that in each pass all input elements are read before any output elements are written. This can only be achieved if this algorithm is run on a device with as many cores as input elements to ensure correct read and write ordering. This is usually not the case for larger arrays (current GPUs have around a few thousands cores. cf. NVIDIA Kepler GK110 in section 2.3.2). A solution to this problem is double buffering. Instead of computing the partial sums in place inside the input array, a second, equally sized buffer is created. In each pass input data is read from one of the buffers and written to the other. Before the next pass the buffers are swapped. Listing 13 shows an example host code implementing this approach.

```
1  void scanNaiveGPU(int* data, int* result, cl_uint n,
2      cl_context context, cl_command_queue queue, cl_kernel kernel, size_t workGroupSize) {
3    size_t bufferSize = n * sizeof(int);
4    cl_mem src = clCreateBuffer(context, CL_MEM_READ_WRITE, bufferSize, nullptr, nullptr);
5    cl_mem dst = clCreateBuffer(context, CL_MEM_READ_WRITE, bufferSize, nullptr, nullptr);
6
7    clEnqueueWriteBuffer(queue, src, false, 0, bufferSize, data, 0, nullptr, nullptr);
8
9    for (cl_uint offset = 1; offset < n; offset <<= 1) {
10     clSetKernelArg(kernel, 0, sizeof(cl_mem), &src);
11     clSetKernelArg(kernel, 1, sizeof(cl_mem), &dst);
12     clSetKernelArg(kernel, 2, sizeof(cl_uint), &offset);
13     clSetKernelArg(kernel, 3, sizeof(cl_uint), &n);
14     size_t globalWS[] = { roundToMultiple(n, workGroupSize) };
15     size_t localWS[]  = { workGroupSize };
16     clEnqueueNDRangeKernel(queue, kernel, 1, nullptr, globalWS, localWS, 0, nullptr, nullptr);
17
```

```
18      std::swap(src, dst);
19    } // for
20
21    clEnqueueReadBuffer(queue, src, true, 0, bufferSize, result, 0, nullptr, nullptr);
22
23    clReleaseMemObject(src);
24    clReleaseMemObject(dst);
25  } // scanNaiveGPU
```

Listing 13: Host code for the naive scan algorithm.

At first, two buffers with the size of the input are created. Both of them have to be read- and writable as they are read from and written to alternatingly when executing the passes. The source buffer is filled with the input data. Although this algorithm is independent from the chosen work group size, we have to round the number of enqueued work items (one for each input element) up to be a multiple of the work group size, which will be the size of the enqueued ND range. After this short setup the passes are executed. Each pass corresponds to a power of two (loop variable `offset`, cf. figure 13) which corresponds to the offset of an element to the predecessor that should be added to it. This offset is raised to the next power of two each pass until it is larger than the problem size. The kernel is executed once for each pass, given the source and destination buffer, the offset and the original problem size as arguments. At the end of a pass the source and destination buffers are swapped (only the handles, not the actual contents). After the last pass has been executed, the result is read from the source buffer (the last pass wrote to the destination buffer which was swapped with the source buffer at the end of the loop). Listing 14 shows the kernel code corresponding to the host code from listing 13.

```
1   __kernel void ScanNaive(__global int* src, __global int* dst, uint offset, uint n) {
2     size_t id = get_global_id(0);
3
4     if (id >= n)
5       return;
6
7     if (id >= offset)
8       dst[id] = src[id] + src[id - offset];
9     else if (id >= (offset >> 1))
10      dst[id] = src[id];
11  } // ScanNaive
```

Listing 14: OpenCL Kernel code for the naive scan algorithm.

The kernel starts by querying the id of the current element. If this id is larger than the actual problem size, the kernel returns. This case can happen when the problem size has been rounded up to be a multiple of the chosen work group size. If the id addresses a valid input element, we determine if this element has a predecessor at the current pass' offset. If this is the case, the input element is read from the source buffer, added to its predecessor (also read from the source buffer) and written to the destination buffer. If the predecessor offset is to large, the input element remains the same, but has to be copied if it was just calculated in the last pass (to keep the buffers consistent).

When we have a look at the benchmark of this algorithm and compare the results with the CPU implementation, we can clearly see that this approach does not profit very well from the large computational power GPUs offer. With 1132 ms at $2^{26}$ elements the naive GPU version is five times slower than the CPU version. The low performance has basically two reasons. The first is the high number of kernel invocations necessary to compute the final result. For $2^{26}$ input elements to scan, 26 passes are necessary each consisting of $2^{26}$ work items mostly performing one addition. Hence, leading to a runtime/work complexity of $\mathcal{O}(nlogn)$. Compared with the complexity of the original CPU implementation, which was $\mathcal{O}(n)$, this algorithm is not work-efficient. The second flaw of this implementation is the high rate of global memory access. Both buffers are accessed multiple times at the same locations throughout the passes. As the scan operation using a simple addition is more memory bound than computational, a lot of time is wasted on waiting for global memory transactions. Fortunately, both problems can be tackled which will be subject to the following sections.

## 4.3 Work efficient GPU implementation

In 1990 Blelloch presented a more efficient version of the all-prefix-sums operation in his article Prefix Sums and Their Applications (Blelloch 1990). He presented a tree-based approach consisting of three phases.

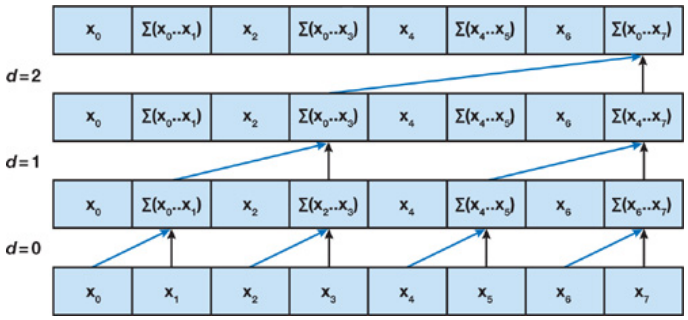


Figure 14: The up-sweep phase of a work efficient parallel scan (Harris, Sengupta, and Owens 2008)

The first phase is called the reduce or up-sweep phase and is illustrated in figure 14. It takes an input array whose length must be a power of two. All elements of the input array are leaves of the tree. The algorithm than takes adjacent pairs of elements and adds them together forming a node holding the sum of it's two child nodes. This step forms a pass of the up-sweep phase and is repeated for the created parent nodes until the root of the tree which than holds the sum of all values. As the values of the right child nodes are not needed anymore after their parent nodes have been calculated, the tree can be built in-place.

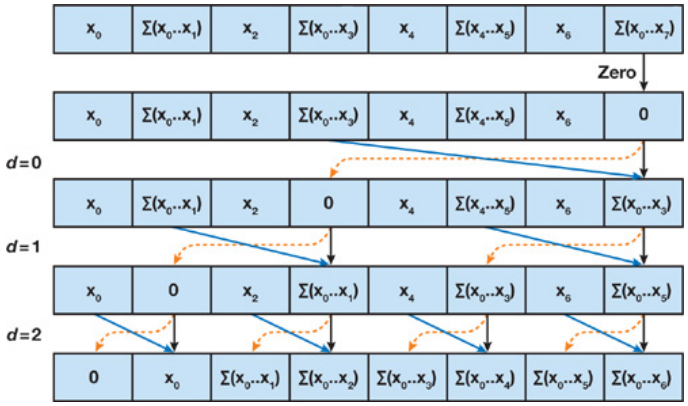


Figure 15: The down-sweep phase of a work efficient parallel scan (Harris, Sengupta, and Owens 2008)

The second phase sets the value of the root node to zero to prepare for the third phase, the down-sweep phase, which is illustrated in figure 15. The down-sweep phase again consists of several passes starting at the root node and repeating along all levels of the tree until the leaves. In each pass the values of the left child nodes are read and temporarily stored. The left child nodes are than replaced by the value of their parent nodes. The previous, temporarily stored values of the left child nodes are then added to the parent nodes' values and written to the right child node. As the parent nodes are not needed anymore after the sum for the right child has been calculated, the tree can again be traversed downwards in-place. Overall, for an input array of $n$ elements, this algorithm performs $n-1$ adds during the up-sweep phase and the same amount of adds during the down-sweep phase[1]. Although almost twice as much work is performed when compared with the sequential CPU implementation, this algorithm can still be considered work-efficient according to

[1] A perfect binary tree with $n$ leaves has a total of $2n-1$ nodes. As only the non-leaf nodes perform additions, the number of leaf nodes can be subtracted resulting in $2n-1-n=n-1$ nodes which perform an addition.

the definition given in section 4.1. Listing 15 shows the host code implementing the work-efficient tree based approach after Blelloch (Blelloch 1990).

```
1  void scanWorkEfficientGPU(int* data, int* result, cl_uint n,
2      cl_context context, cl_command_queue queue, cl_kernel upSweep, cl_kernel downSweep,
3      size_t workGroupSize) {
4    size_t adjustedSize = roundToPowerOfTwo(n);
5    cl_mem buffer = clCreateBuffer(context, CL_MEM_READ_WRITE,
6        adjustedSize * sizeof(int), nullptr, nullptr);
7
8    clEnqueueWriteBuffer(queue, buffer, false, 0, n * sizeof(int), data, 0, nullptr, nullptr);
9
10   // upsweep
11   size_t nodes = adjustedSize >> 1;
12   for (cl_uint offset = 1; offset < adjustedSize; offset <<= 1, nodes >>= 1) {
13     clSetKernelArg(upSweep, 0, sizeof(cl_mem), &buffer);
14     clSetKernelArg(upSweep, 1, sizeof(cl_uint), &offset);
15     size_t globalWS[] = { nodes };
16     size_t localWS[] = { std::min(workGroupSize, nodes) };
17     clEnqueueNDRangeKernel(queue, upSweep, 1, nullptr, globalWS, localWS, 0, nullptr, nullptr);
18   } // for
19
20   // set last element to zero
21   cl_uint zero = 0;
22   clEnqueueWriteBuffer(queue, buffer, false, (adjustedSize - 1) * sizeof(cl_uint), sizeof(cl_uint),
23       &zero, 0, nullptr, nullptr);
24
25   // downsweep
26   nodes = 1;
27   for (cl_uint offset = adjustedSize >> 1; offset >= 1; offset >>= 1, nodes <<= 1) {
28     clSetKernelArg(downSweep, 0, sizeof(cl_mem), &buffer);
29     clSetKernelArg(downSweep, 1, sizeof(cl_uint), &offset);
30     size_t globalWS[] = { nodes };
31     size_t localWS[] = { std::min(workGroupSize, nodes) };
32     clEnqueueNDRangeKernel(queue, downSweep, 1, nullptr, globalWS, localWS, 0, nullptr, nullptr);
33   } // for
34
35   clEnqueueReadBuffer(queue, buffer, true, 0, n * sizeof(int), result, 0, nullptr, nullptr);
36
37   clReleaseMemObject(buffer);
38 } // scanWorkEfficientGPU
```

Listing 15: Host code for the work-efficient scan algorithm.

Contrary to the previous algorithms, this implementation will use two kernels, one for the up-sweep and one for the down-sweep phase. To start with, the number of input elements has to be rounded up to be a power of two. This size is then used to create a buffer which is filled with the input array. After these initialization steps we can begin with the up-sweep phase. The loop variable for the passes is the offset of the buffer index between two adjacent nodes on the same tree level. This value starts with one for the first pass and is raised to the next power of two for each subsequent pass until the size of the input buffer has been reached. For each pass the up-sweep kernel is enqueued with the buffer and the current pass' offset as arguments. The number of work items required for a pass is equal to the number of parent nodes calculated on the current pass' level. This value is a half of the buffer size for the first pass and halves itself after each pass. Although this algorithm is still independent of the chosen work group size, we have to make sure that it is not larger than the global number of work items.
After the up-sweep phase has been completed, the last element corresponding to the tree's root node will be set to zero. This is easily accomplished by enqueuing a write operation writing the value zero to the last buffer element.
Finally the down-sweep phase can finish the scan computation. Similar to the up-sweep phase several passes are executed given the offset between two adjacent tree nodes on the same level and the buffer as argument. In contrast to the up-sweep phase, the tree is now traversed top-down meaning the offset starts with a half of the buffer size and is set to the next lower power of two in each pass until one has been reached. Analogous, the number of nodes to process (the global work size) starts with one and doubles every pass. When the down-sweep phase has completed, the buffer holds the final scan result which can than be read back to host memory. Listing 16 shows the corresponding OpenCL kernel code for the work-efficient scan implementation.

```
1  __kernel void UpSweep(__global int* buffer, uint offset) {
2    uint stride = offset << 1;
3    uint id = (get_global_id(0) + 1) * stride - 1;
4
5    buffer[id] += buffer[id - offset];
```

```
6  } // UpSweep
7
8  __kernel void DownSweep(__global int* buffer, uint offset) {
9    uint stride = offset << 1;
10   uint id = (get_global_id(0) + 1) * stride - 1;
11
12   int val = buffer[id];
13   buffer[id] += buffer[id - offset];
14   buffer[id - offset] = val;
15 } // DownSweep
```

Listing 16: OpenCL Kernel code for the work efficient scan algorithm.

The `UpSweep` kernel starts by computing the value of `stride` which is the distance of two parent nodes in the current pass. This value is used to calculate the buffer index (`id`) of the current parent node (equal to the right child node) from the work items global id. The input value at `id` is read (right child) together with the corresponding neighbor node at the given offset (left child). The computed sum is then written over the right child's location.
The `DownSweep` kernel initializes the same way as it's preceding one by determining the current pass' stride and the parent node's index. Then, the value of the parent node is read and temporarily stored. The value of the left child node (given by `offset`) is added the parent node (which becomes the right child node). Finally the previous, temporarily stored value of the parent node is passed down to the left child node.

When having a look at the benchmark results for this algorithm in figure 12 we can see that our efforts have payed off. The initial 1132 ms of the naive GPU implementation have shrunk to 604 ms on an array of $2^{26}$ elements. Furthermore, this implementation is now work-efficient and therefore avoiding unnecessary additions. Additionally, as all intermediate results are stored in-place, the algorithm does not waste memory by double buffering, such as the naive approach. However, the algorithm requires the input to be a power of two, which becomes more disadvantages with larger input sizes. This can be clearly seen at the stepped shape of the runtime curve which rises at every new power of two as the time required for the calculation is equal for all problem sizes that are round up to the same power of two.

## 4.4 Recursively scanning blocks in local memory

The previous work-efficient implementation in section 4.3 does already perform quite well for an initially sequential problem. However, the input buffer size restriction is undesirable. Furthermore, no local memory is used which might be useful for computing and storing intermediate results.

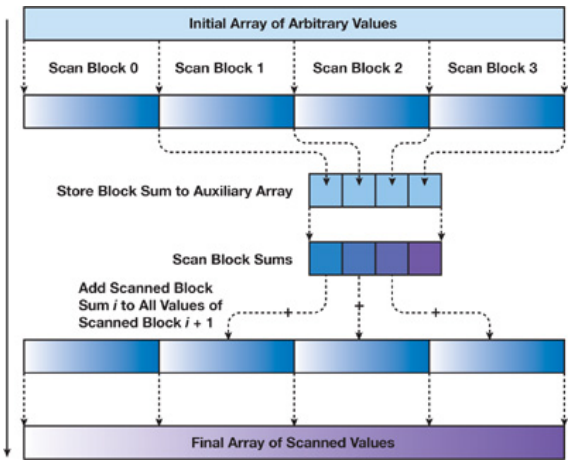


Figure 16: Scanning larger arrays of values recursively (Harris, Sengupta, and Owens 2008).

GPU Gems 3 Chapter 39 (Harris, Sengupta, and Owens 2008) shows a different approach for computing all prefix sums of an input array. The method is shown in figure 16. The input data

is rounded up to be a multiple of a block size (which is two times the chosen work group size in their implementation). Each work group then loads a block from global memory to local memory. Each block is then scanned using Blelloch's work-efficient scan algorithm (Blelloch 1990), but in local memory inside a work group. After the up-sweep phase is finished, the root element of each tree in a block is copied into a smaller, temporary buffer. This buffer is then recursively scanned again. The scan result of the temporary buffer can then be used to offset the scanned blocks of the original buffer to build the final scan. Although this algorithm is not tied to a specific work group size it performs better the larger the work group size is chosen, as the work group sizes determines the factor by which the input data is reduced in each recursion. Listing 17 shows the host code of this recursive scan algorithm.

```
1  void scanRecursiveGPU_r(cl_mem values, cl_uint n,
2      cl_context context, cl_command_queue queue, cl_kernel scanBlocks, cl_kernel addSums,
3      size_t workGroupSize) {
4    size_t sumCount = roundToMultiple(n / (workGroupSize * 2), workGroupSize * 2);
5    cl_mem sums = clCreateBuffer(context, CL_MEM_READ_WRITE, sumCount * sizeof(int), nullptr, nullptr);
6
7    clSetKernelArg(scanBlocks, 0, sizeof(cl_mem), &values);
8    clSetKernelArg(scanBlocks, 1, sizeof(cl_mem), &sums);
9    clSetKernelArg(scanBlocks, 2, sizeof(int) * 2 * workGroupSize, nullptr);
10   size_t globalWS[] = { n / 2 };
11   size_t localWS[] = { workGroupSize };
12   clEnqueueNDRangeKernel(queue, scanBlocks, 1, nullptr, globalWS, localWS, 0, nullptr, nullptr);
13
14   if (n > workGroupSize * 2) {
15     scanRecursiveGPU_r(sums, sumCount, context, queue, scanBlocks, addSums, workGroupSize);
16
17     clSetKernelArg(addSums, 0, sizeof(cl_mem), &values);
18     clSetKernelArg(addSums, 1, sizeof(cl_mem), &sums);
19     clEnqueueNDRangeKernel(queue, addSums, 1, nullptr, globalWS, localWS, 0, nullptr, nullptr);
20   } // if
21
22   clReleaseMemObject(sums);
23 } // scanRecursiveGPU_r
24
25 void scanRecursiveGPU(int* data, int* result, cl_uint n,
26     cl_context context, cl_command_queue queue, cl_kernel scanBlocks, cl_kernel addSums,
27     size_t workGroupSize) {
28   size_t adjustedSize = roundToMultiple(n, workGroupSize * 2);
29   cl_mem buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, adjustedSize * sizeof(int), nullptr,
         nullptr);
30
31   clEnqueueWriteBuffer(queue, buffer, false, 0, n * sizeof(int), data, 0, nullptr, nullptr);
32   scanRecursiveGPU_r(buffer, adjustedSize, context, queue, scanBlocks, addSums, workGroupSize);
33   clEnqueueReadBuffer(queue, buffer, true, 0, n * sizeof(int), result, 0, nullptr, nullptr);
34
35   clReleaseMemObject(buffer);
36 } // scanRecursiveGPU
```

Listing 17: Host code for the recursive scan algorithm.

This implementation again uses two kernels, one for scanning the blocks in local memory and one for applying the offsets from the scanned temporary buffer to the original one. The host code consists of two parts, the setup code and the actual recursion. The setup is performed in `recursiveGPU` which starts by rounding the input data size up to be a multiple of twice the size of a work group, because each work item processes two values. Then a buffer is created and the input data written to it. This buffer is then passed to the recursive scan procedure `recursiveGPU_r`. The recursion starts by computing the size of the temporary buffer `sums` that will hold the values of the root nodes (the sums over each block). As each thread processes two input elements and each work group computes one sum over a block, this temporary buffer's size is the problem size divided by two times the work group size. The result is then rounded up to be a multiple of this value, so it can be recursively scanned again with this algorithm. After the size has been determined, the `sums` buffer can be created. No initialization or memory transfer is required as the buffer is entirely accessed by the kernels. Afterwards, the block scan kernel can be set up. It takes the original input buffer as well as the buffer for the sums as arguments. Furthermore local memory is allocated to cache the block calculated by each work group. As each work item reads two input elements, the global work size is half the number of input elements. After the kernel has been enqueued, we have to check whether the input data for this recursion consisted of more than one block. If this is the case we scan the temporary buffer (containing more than 1 value) by recursively calling `recursiveGPU_r` on this buffer. After scanning the temporary `sums` buffer has completed, the sums can be applied to the original buffer. Therefore

another kernel is enqueued given both buffers as arguments. The global and local work size are equal to the previous kernel on the same recursion level. Listing 18 shows the kernel code for the recursive scan implementation.

```
1  __kernel void ScanBlocks(__global int* buffer, __global int* sums, __local int* shared) {
2    uint globalId = get_global_id(0);
3    uint localId  = get_local_id(0);
4    uint n        = get_local_size(0) * 2;
5
6    uint offset = 1;
7
8    shared[2 * localId + 0] = buffer[2 * globalId + 0];
9    shared[2 * localId + 1] = buffer[2 * globalId + 1];
10
11   // build sum in place up the tree
12   for (uint d = n >> 1; d > 0; d >>= 1) {
13     barrier(CLK_LOCAL_MEM_FENCE);
14     if (localId < d) {
15       uint ai = offset*(2*localId+1)-1;
16       uint bi = offset*(2*localId+2)-1;
17       shared[bi] += shared[ai];
18     } // if
19     offset <<= 1;
20   } // for
21   barrier(CLK_LOCAL_MEM_FENCE);
22
23   // save sum and clear the last element
24   if (localId == 0) {
25     sums[get_group_id(0)] = shared[n - 1];
26     shared[n - 1] = 0;
27   } // if
28
29   // traverse down tree & build scan
30   for (uint d = 1; d < n; d <<= 1) {
31     offset >>= 1;
32     barrier(CLK_LOCAL_MEM_FENCE);
33     if (localId < d) {
34       uint ai = offset*(2*localId+1)-1;
35       uint bi = offset*(2*localId+2)-1;
36
37       int t = shared[ai];
38       shared[ai] = shared[bi];
39       shared[bi] += t;
40     } // if
41   } // for
42   barrier(CLK_LOCAL_MEM_FENCE);
43
44   buffer[2 * globalId + 0] = shared[2 * localId + 0];
45   buffer[2 * globalId + 1] = shared[2 * localId + 1];
46 } // ScanBlocks
47
48 __kernel void AddSums(__global int* buffer, __global int* sums) {
49   uint globalId = get_global_id(0);
50
51   int val = sums[get_group_id(0)];
52
53   buffer[globalId * 2 + 0] += val;
54   buffer[globalId * 2 + 1] += val;
55 } // AddSums
```

Listing 18: OpenCL Kernel code for the recursive scan algorithm.

The `ScanBlocks` kernel starts with querying some values from OpenCL. Beside the global and local id, also the size `n` of the block of the current work group is determined. Each thread than loads two values of the block into local memory. The following code then implements the tree based scan approach by Blelloch which has already been discussed in the previous section 4.3. The three phases (up-sweep, set-last-zero and down-sweep) are executed across all threads of the work group on the block in local memory. The only difference is, that the computed value of the root node after the up-sweep phase is moved to the `sums` buffer at the index of the current work group. After the block has been scanned completely, it is copied back from local to global memory.

The `AddSums` kernel is executed with the same global and local sizes as passed to the `ScanBlocks` kernel on the same recursion level. The `sums` buffer now contains a scan of the sums of all blocks scanned in this recursion. Therefore, the position where the `ScanBlocks` kernel wrote the root node's value, now contains the sum of all elements (of all blocks) preceding the block of the current work group. This value is retrieved and added to all values of the current block finishing the scan.

Concerning the performance benchmark in figure 12, this implementation scales better with the problem size although it is not faster than the global work-efficient scan of the previous section (606 vs. 604 ms on $2^{26} elements$). Furthermore, the recursive implementations' kernels are enqueued far less often than the one's of the work-efficient implementation. This can be explained by the reduction factor in each algorithm. The work efficient implementation reduces the number of work items by a factor of two each iteration while the recursive algorithm reduces by the work group size (which was 256 for the benchmarks) times two. In return, a lot more work is done inside the recursive algorithm's kernels. Also the number of additions required to compute the final result increased as extra adds a needed to apply the sums from the temporary buffer to the input buffer. However, this implementation showed a different concept of how a problem can be broken down into smaller parallel pieces of work. The next implementation will follow up on this idea.

## 4.5 Optimization using vector types

The final implementation shown in this chapter takes the idea of the recursive scan approach from the previous chapter one step further (Harris, Sengupta, and Owens 2008, ch.39.2.5). Instead of loading only two elements in each work item we will load two vectors of elements. These are then scanned inside each work item. The sums of the vectors (root node after the up-sweep phase) is then copied to local memory. The algorithm then continues just as the previous chapter's version by scanning the blocks in local memory. The results can then be used to offset the vectors of each work item just as the `AddSums` kernel on a higher level. The remaining part of the implementation basically stays the same. With this modification, each work group does not only load a block of two times the work group size of elements but a block of two times the work group size times the chosen vector width. As a result the input array is reduced faster by a factor of the chosen vector width in each recursion. The consumed local memory stays the same as only the sums of both vectors are placed into local memory. Only the consumed registers will increase. Listing 19 shows the differences of the host code of the this vector type implementation compared with the previous one.

```
1  #define VECTOR_WIDTH 8
2  ...
3    size_t sumCount = roundToMultiple(n / (workGroupSize * 2 * VECTOR_WIDTH),
4        workGroupSize * 2 * VECTOR_WIDTH);
5    ...
6    size_t globalWS[] = { n / (2 * VECTOR_WIDTH) };
7    ...
8    if (n > workGroupSize * 2 * VECTOR_WIDTH) {
9      scanCLRecursiveVector_r(sums, sumCount, context, queue, scanBlocks, addSums, workGroupSize);
10     ...
```

Listing 19: Differences to the host code shown in listing 18 for the recursive scan algorithm using vector types.

The first and most important step is to define the width of the vectors used. OpenCL supports vector types with the lengths of 2, 4, 8 or 16. The larger this `VECTOR_WIDTH` is chosen, the larger is the reduction of the input buffer in each recursion. However, as a side effect the consumed registers per work item increase leading to a lower occupancy of the kernel. The ideal vector width may be hardware specific and is subject to corresponding benchmarks. This implementation will use a `VECTOR_WIDTH` of 8. The remaining changes to the host code are straight forward. As the blocks get bigger by a factor of `VECTOR_WIDTH`, the size of the temporary `sums` buffer decreases by and has to be rounded up to the same factor. Furthermore, the global work size shrinks. Finally, the check whether the input array consisted of more than one block has to be adapted to the new block size. Although the changes to the host code are quite small, the kernel needs a few more extensions as shown in listing 20.

```
#define CONCAT(a, b) a ## b           // concat token a and b
#define CONCAT_EXP(a, b) CONCAT(a, b) // concat token a and b AFTER expansion

#define UPSWEEP_STEP(left, right) right += left

#define UPSWEEP_STEPS(left, right) \
  UPSWEEP_STEP(CONCAT_EXPANDED(val1.s, left), CONCAT_EXPANDED(val1.s, right)); \
  UPSWEEP_STEP(CONCAT_EXPANDED(val2.s, left), CONCAT_EXPANDED(val2.s, right))

#define DOWNSWEEP_STEP_TMP(left, right, tmp) \
  int tmp = left; \
  left = right; \
  right += tmp

#define DOWNSWEEP_STEP(left, right) \
  DOWNSWEEP_STEP_TMP(left, right, CONCAT_EXPANDED(tmp, __COUNTER__))

#define DOWNSWEEP_STEPS(left, right) \
  DOWNSWEEP_STEP(CONCAT_EXPANDED(val1.s, left), CONCAT_EXPANDED(val1.s, right)); \
  DOWNSWEEP_STEP(CONCAT_EXPANDED(val2.s, left), CONCAT_EXPANDED(val2.s, right))

__kernel void ScanBlocksVec(__global int8* buffer, __global int* sums, __local int* shared) {
  uint globalId = get_global_id(0);
  uint localId  = get_local_id(0);
  uint n        = get_local_size(0) * 2;

  uint offset = 1;

  int8 val1 = buffer[2 * globalId + 0];
  int8 val2 = buffer[2 * globalId + 1];

  // upsweep vectors
  UPSWEEP_STEPS(0, 1);
  UPSWEEP_STEPS(2, 3);
  UPSWEEP_STEPS(4, 5);
  UPSWEEP_STEPS(6, 7);

  UPSWEEP_STEPS(1, 3);
  UPSWEEP_STEPS(5, 7);

  UPSWEEP_STEPS(3, 7);

  // move sums into shared memory block and clear last elements
  shared[2 * localId + 0] = val1.s7;
  shared[2 * localId + 1] = val2.s7;

  val1.s7 = 0;
  val2.s7 = 0;

  // downsweep vectors
  DOWNSWEEP_STEPS(3, 7);

  DOWNSWEEP_STEPS(1, 3);
  DOWNSWEEP_STEPS(5, 7);

  DOWNSWEEP_STEPS(0, 1);
  DOWNSWEEP_STEPS(2, 3);
  DOWNSWEEP_STEPS(4, 5);
  DOWNSWEEP_STEPS(6, 7);

  // scan block in local memory
  ...

  // apply the sums
  val1 += shared[2 * localId + 0];
  val2 += shared[2 * localId + 1];

  buffer[2 * globalId + 0] = val1;
  buffer[2 * globalId + 1] = val2;
} // ScanBlocksVec

__kernel void AddSumsVec(__global int8* buffer, __global int* sums) {
  uint globalId = get_global_id(0);

  int val = sums[get_group_id(0)];

  buffer[globalId * 2 + 0] += val;
  buffer[globalId * 2 + 1] += val;
} // AddSumsVec
```

Listing 20: OpenCL Kernel code for the recursive scan algorithm using vector types with a width of eight.

First of all, several macros are defined for the `ScanBlocksVec` kernel to avoid redundant code for the up-sweep and down-sweep implementation on the vector types. The kernel begins, after querying topological informations, by copying two `int8` vectors into registers. The up-sweep phase is completely executed on the work item's registers on both vectors resulting in a lot of independent instructions. After the up-sweep phase, the last vector elements hold the sum over both vectors which is then copied into the shared memory block. After the last elements are reset to zero, the down-sweep phase continues, finalizing the scan on the two vectors stored in registers. The remaining part of the kernel is equal to the one of the previous implementation in listing 18 which calculates the scan of the block in shared memory and copies the block's sum into the `sums` buffer. A final small addition is to offset the vectors by the sum of all previous vectors of preceding work items in this block. Notice that the value loaded from local memory is added to each vector element. Finally the vectors are written back to global memory.
The `AddSums` kernel only changes by the type of the buffer which now consists of `int8`s. Also notice here, that the value of `val` loaded from the `sums` is added to all vector elements of the two vector elements of `buffer`.

When having a look at the benchmark results in figure 12 we can clearly see a difference to the previous, non-vector implementation. The 604 ms of the non-vector implementation could be reduced by 37% to 383 ms for a problem size of $2^{26}$ input elements. This is already an amazing result for a sequential problem such as scan. Nevertheless, the initial CPU scan time of 225 ms is still out of reach. However, if one has a closer look at the benchmark data one can see that the final recursive vector scan implementation actually beat the CPU implementation when it comes down to run time. It took only 87 ms to scan the $2^{26}$ elements on the GPU. Upload and download have the larger shares of the total time required of the GPU scan, contributing 154 and 141 ms for transferring the 256 MiB to and from the GPU's main memory. This means that 77 % of the total run time is wasted on memory transfers, which is a huge problem for many memory intensive algorithms like scan. Figure 17 shows a detailed view on the upload, run and download phase of the recursive vector scan implementation.

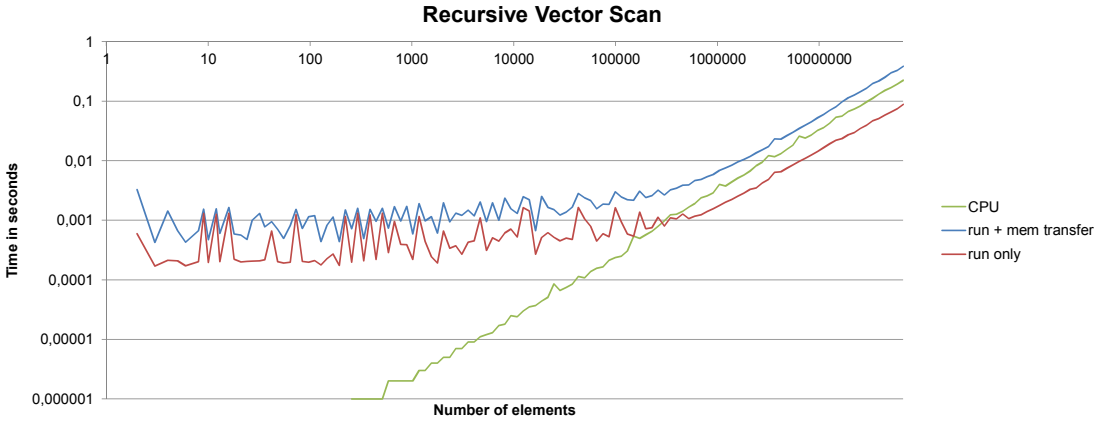


Figure 17: A closer look to the benchmark of the recursive vector scan from section 4.5. It points out how much time is wasted by memory transfers. Note that both axis are of logarithmic scale.

## 4.6 Further implementations

Beside the presented kernels a further approach has been implemented and benchmarked which has not been covered. However some experiences may help future developers.

**Bank conflict avoidance**
The GPU Gems 3 chapter about scan (Harris, Sengupta, and Owens 2008) also talks about avoiding bank conflicts occurring in the kernel in chapter 4.3 where blocks are scanned in local memory. Both recursive scan implementations (chapters 4.4 and 4.5) have been implemented using the presented approach to avoid bank conflicts. However, the time required to calculate the conflict free offsets to local memory compensates for the time won by faster local memory access.

Furthermore, several libraries exist supporting GPGPU accelerated scans.

**clpp (krys@polarlights.net 2011)**
The OpenCL Data Parallel Primitives Library is an open source and freely available library offering a few GPU primitives such as a scan and sort implementation. However, the project seems to be inactive (last release in July 2011).

**ArrayFire (AccelerEyes 2013)**
ArrayFire is a commercial GPU software acceleration library provided by AccelerEyes. It provides a lot of GPGPU accelerated algorithms using CUDA and OpenCL including scan.

## 4.7 Summary and conclusion

In this chapter we have seen how a sequential algorithm like the all-prefix-sum can be parallelized for GPUs. A tree based approach using several passes delivered good results. Also the idea of solving the problem in smaller sub groups which are than merged into the global result proofed to be successful. Both concepts are common patterns when trying to port sequential CPU code to a GPU. However, we did also learn that using a GPU does not always result in a performance boost, despite their enormous power. So does the final scan only perform three times faster than the CPU implementation excluding memory transfer. If the time for copying the data to and from the GPU's main memory is included, we would be better off staying with the CPU. This is a problem for many fast algorithms operating on larger chunks of memory. Nevertheless, a fast GPU implementation of such primitives can still be needed in cases where the input and output data is already present or consumed on the GPU. This is the case when input data is produced on the GPU (e.g., scanning image data from a rendering process) or present as input for other purposes (other algorithms, data for rendering). Also the output may be directly used by the GPU (e.g using scan as part of a filtering routine generating a list of objects to render). Even if a GPU algorithm would be slower in run time than a CPU version it might still outperform the CPU variant including memory transfer to and from the systems main memory.

# 5 Sorting

One of the most fundamental operations in computer science is sorting a sequence of elements. And as old as the problem is, as numerous are the algorithms to solve it. These sorting algorithms differ in various aspects such as best, average and worst runtime or memory complexity, stability, the number of comparisons and swaps and whether the algorithm is a comparison based sort or not.

The sort implementations provided by today's standard libraries are highly tuned and achieve an optimal asymptotic runtime complexity of $\mathcal{O}(n \, log \, n)$ for comparison based sorts. Examples are variations of quick sort (C), merge sort (C++), intro sort (C++, .NET) or Timsort (Java, Python). All these algorithms are comparison based sorts, thus requiring a method of comparing two elements of the input sequence. This comparison is often provided either by the language or standard library (e.g., $<$ operator) or by the programmer via a custom comparison function, giving the flexibility to compare and sort any kind of elements.

Another class of sorting methods are integer sorting algorithms. These algorithms do not use comparisons to determine the order of elements. They rely on more flexible integer arithmetic applied to the keys which have to be sorted. Therefore, they have a better asymptotic runtime complexity than comparison based ones. Popular algorithms of this kind are radix sort, counting sort and bucket sort. All of them running with $\mathcal{O}(n + k)$ or $\mathcal{O}(n * k)$ (where $k$ is a constant) in linear time. However, despite their limitation on the sort key, integer sorting algorithms also work on other kind of types as long as they can be represented as integers in binary (e.g., strings can be seen as byte array forming a (larger) integer).

Considering parallelizability and an eventual GPU implementation, sorting lies between matrix multiplication and prefix sum offering some degree of parallelism depending on the chosen algorithm. This chapter will focus on the implementation of two widely chosen algorithms for GPU sorting. These are the comparison based bitonic sorting network and the integer sorting algorithm radix sort. To reduce code complexity (especially of the latter), the input array consists of unsigned 32 bit integers.

## 5.1 CPU Implementations

Before stepping into the details of the two chosen GPU algorithms, bitonic and radix sort, a set of CPU sorting algorithms is discussed. Their resulting run times are used as a reference and compared with the GPU implementations in the benchmarks.

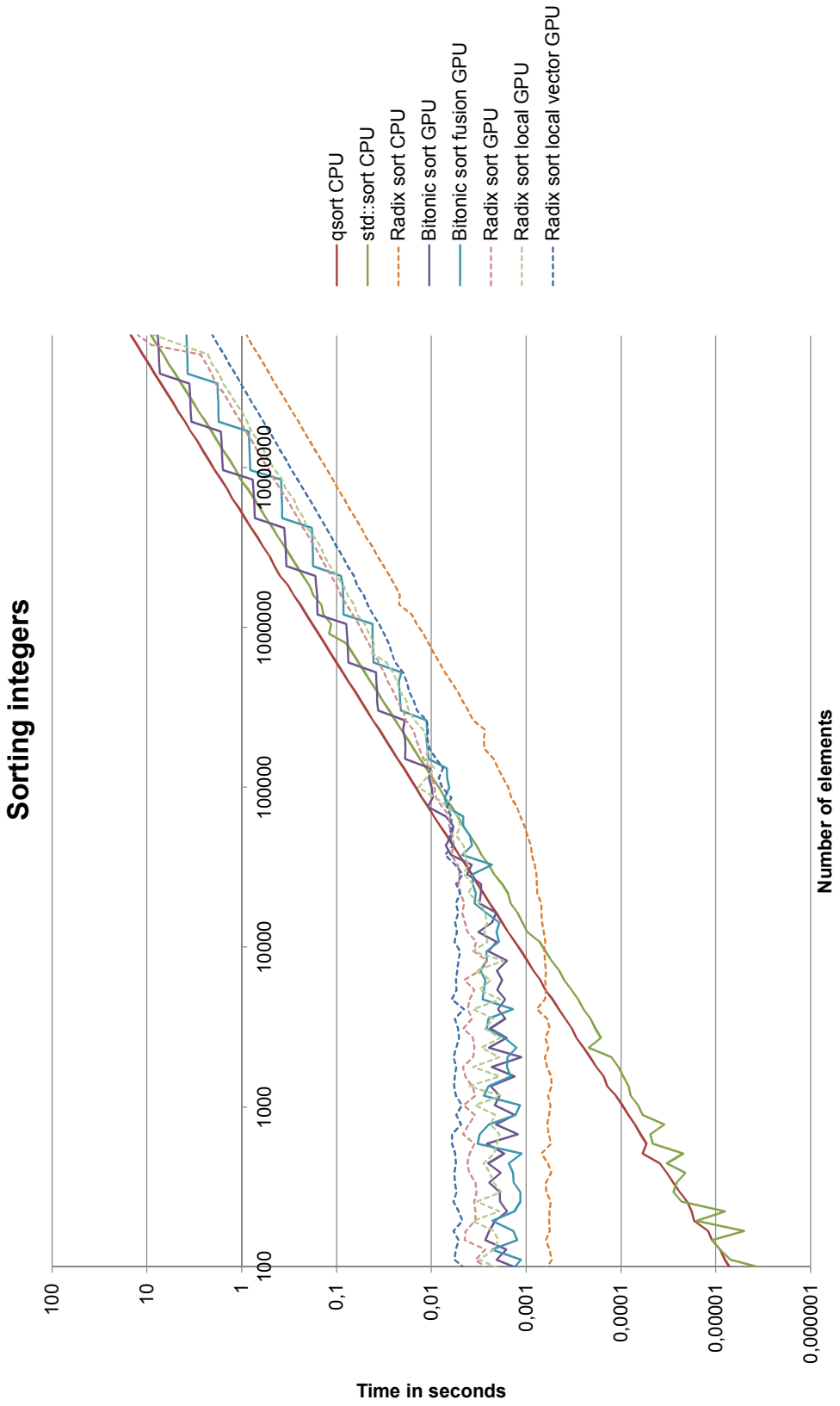


Figure 18: Benchmark of several sort implementations. Both axis are of logarithmic scale. Integer sorting algorithms (radix sort) are drawn in dashed lines.

### 5.1.1 C/C++ standard library routines

As the later GPU implementations should be compared with well implemented and wide-spread CPU ones, the first step is to measure the performance of the standard library's routines. Listing 21 shows a typical usage of the `qsort` function provided by the stdlib header from C.

```
1  typedef uint32_t uint;
2
3  void qsortCPU(uint* data, size_t n) {
4    qsort(data, n, sizeof(uint), [] (const void* a, const void* b) -> int {
5      if (*((uint*)a) < *((uint*)b))
6        return -1;
7      else if (*((uint*)a) > *((uint*)b))
8        return 1;
9      return 0;
10   });
11 } // qsortCPU
```

Listing 21: Sorting an array of unsigned integers using `qsort` from the C stdlib header. The provided comparison lambda function uses comparisons instead of simply subtracting the input values (`a - b`) as this may cause unwanted behaviour due to overflows (e.g., `1u - MAX_UINT` is 2 instead of a negative number). Subtraction does work for signed types.

As `qsort` is usually precompiled and part of the runtime library, a compare function has to be provided which is called for every comparison. This will probably slow down the run time when compared with the C++ `std::sort` which can use an existing overload of the $<$ operator or inline a provided compare function. Listing 22 shows a typical call to the `std::sort` function template from the C++ algorithm header.

```
1  void stdsortCPU(uint* data, size_t n) {
2    std::sort(data, data + n);
3  } // stdsortCPU
```

Listing 22: Sorting an array of unsigned integers using `std::sort` from the C++ algorithm header.

As the $<$ operator is defined for unsigned integer types no custom compare function has to be provided and no additional overhead is created for comparing elements. The difference in the benchmark chart in figure 18 is evident. It takes `qsort` 14.746 seconds to sort a sequence of $2^{26}$ elements while `std::sort` only requires 9.039 seconds.

### 5.1.2 Radix sort

In contrast to the comparison based `qsort` and `std::sort`, radix sort operates on the binary representation of the input elements in several passes. The input values are therefore seen as numbers of a numeral system of a chosen base (= radix). In each pass, a digit at the same position (starting with the least significant) is selected from all input elements. These input elements are then split into buckets according to the value of the current pass' digit. Elements with the same digit value retain their order.

If the radix is two, in each pass a bit (digit) of each input element (number represented in the binary system) is selected (starting with the least significant bit). The input elements are then split into two sequences (buckets) according to the selected bit. This is done by creating a histogram holding the number of elements having the same bit value for each bit combination (0 or 1). After the histogram has been created, it is exclusively scanned. The scanned histogram now holds the start index for each bucket. The input elements are permuted and moved into their corresponding buckets according to the selected bit. Elements with an equal bit are written into the same bucket in the same order as they appeared in the pass' input (each pass is stable). An auxiliary array is usually needed for this permutation step. This procedure is repeated for every bit of the input elements.

By only selecting one bit in each pass, 32 passes are required to sort the input sequence (elements are 32 bit unsigned integers). In each pass the input array is iterated over two times and the histogram has to be scanned once. This accumulates to 64 iterations over the input sequence and 32 scans of a two element histogram making radix sort a linear algorithm but with a quite significant constant factor. This factor can be reduced by selecting several bits of each input element at once in each pass (increasing the radix to a larger power of two). As a result, the number of passes decreases linearly with the number of selected bits (which is called radix and gives the algorithm its name). As a trade-off, the size of the histogram increases by a power of two. The CPU radix sort implementation used in the benchmarks is provided in listing 23.

```
1  #define RADIX 16
2  #define BUCKETS (1 << RADIX)
3  #define RADIX_MASK (BUCKETS - 1)
4
5  void radixSortCPU(uint* data, size_t n) {
6    size_t histogram[BUCKETS];
7    uint* aux = new uint[n];
8
9    uint* src = data;
10   uint* dst = aux;
11   for (size_t bits = 0; bits < sizeof(uint) * 8 ; bits += RADIX) {
12     memset(histogram, 0, BUCKETS * sizeof(size_t));
13
14     // calculate histogram
15     for (size_t i = 0; i < n; ++i) {
16       uint element = src[i];
17       uint pos = (element >> bits) & RADIX_MASK;
18       histogram[pos]++;
19     } // for
20
21     // scan histogram (exclusive)
22     size_t sum = 0;
23     for (size_t i = 0; i < BUCKETS; ++i) {
24       size_t val = histogram[i];
25       histogram[i] = sum;
26       sum += val;
27     } // for
28
29     // permute
30     for(size_t i = 0; i < n; ++i) {
31       uint element = src[i];
32       uint pos = (element >> bits) & RADIX_MASK;
33       size_t index = histogram[pos]++;
34       dst[index] = src[i];
35     } // for
36
37     std::swap(src, dst);
38   } // for
39
40   if(dst != data)
41     memcpy(data, dst, n * sizeof(uint));
42
43   delete[] aux;
44 } // radixSortCPU
```

Listing 23: Sorting an array of unsigned integers using radix sort. The algorithm uses two passes analyzing 16 bits each. The implementation is based on the radix sort sample shipped with AMD APP SDK (Advanced Micro Devices, Inc. 2013a).

The implementation uses a relatively high radix of 16 (`RADIX`) which allows radix sort to finish in two passes. As a consequence, the histogram consists of $2^{16}$ elements (`BUCKETS`) occupying 256 KiB of memory but has to be scanned only two times.

The performance difference of radix sort to the previously presented comparison based algorithm is significant as seen in figure 18. Although the high radix of 16 causes a large overhead on small input sizes (as the full $2^{16}$ element histogram has to be scanned independently of the input size), radix sort shows its strength on larger inputs. It catches up to `std::sort` at approximately 10,000 elements and then outperforms the library routine largely sorting $2^{26}$ elements in 0.884 seconds (compared to 9.039 seconds of `std::sort`).

## 5.2 GPU Implementations

After the presented CPU sorting routines in the previous sections, the following sections focus on creating efficient bitonic and radix sorting implementations for the GPU using OpenCL.

### 5.2.1 Bitonic Sort

Many regular sorting algorithms do not fit into a GPU's massively parallel programming model by offering insufficient parallelism, requiring non-array data structures or begin based on recursion. However, a small subgroup of sorting approaches exists that fits the requirements of parallel hardware perfectly: sorting networks. A sorting network is a network of wires (one for each input) and comparators along these wires. Comparators connect two wires, compare their values and might swap them. Finding optimal sorting networks for a given number of inputs is difficult and still subject to research. However, in 1968 Ken Batcher presented (beside others) an approach to create sorting networks achieving reasonable results (Batcher 1968). His idea is based on efficiently merging a bitonic sequence into a sorted sequence, hence the name bitonic sorter. A bitonic sequence is a sequence of elements which consists of two equally long subsequences, where one is sorted ascending and the other descending. As two arbitrary elements form a bitonic sequence, they can be merged into a sorted sequence. As two sequences sorted in opposite order also form a bitonic sequence they can again be merged into a sorted sequence. This allows us to create sorting networks for any power of two sized inputs.

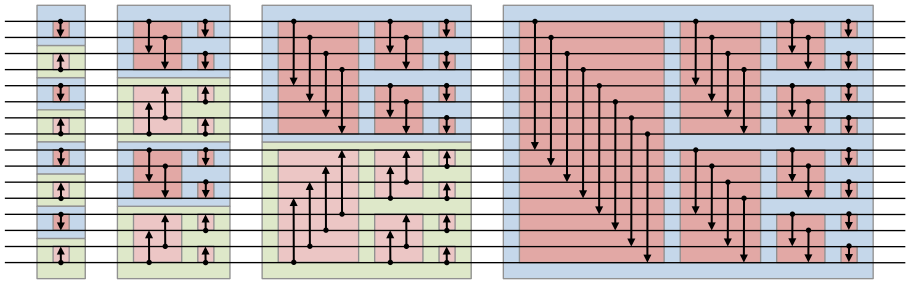

Figure 19: Example of a bitonic sorting network for 16 inputs (Bitonic 2012).

Figure 19 shows a bitonic sorting network for 16 input elements. The input elements start on the left side of the network and travel through it to the right. Each arrow represents a comparator which compares the elements of the connected wires and swaps them if necessary, so that the larger element is swapped to the wire the arrow points at. Each blue or green box is a separate bitonic merge receiving a bitonic sequence as input and outputting a sorted one (blue = ascending, green = descending). The red boxes contain the comparators and are equally structured. They compare the top half of the input (bitonic) sequence against the bottom half and create two bitonic sequences where each element of the upper bitonic sequence is smaller than or equal to every element in the bottom one (vice versa in green boxes). Further red boxes are than recursively applied to the two outputs until the sequence is sorted.

The created networks consist of $\mathcal{O}(n\ log^2\ n)$ comparators. As $\frac{n}{2}$ comparators can be executed in parallel (cf. figure 19), the runtime complexity of a bitonic sorter would be $\mathcal{O}(log^2\ n)$ on a computer with at least $\frac{n}{2}$ processing elements.

Sorting inputs of any length is also possible with a bitonic sorting network. The network is constructed for the next larger power of two and all comparators affecting a wire of a (possible) input element larger than the actual input are omitted. This approach and a sample implementation is discussed in more detail in an online article by Hans Lang (Lang 1998). To keep the code simple, the bitonic sorter implementation presented in this chapter will only focus on inputs with a power of two length. Listing 24 shows the host code of a bitonic sorter implementation.

```
1  void bitonicSortGPU(uint* data, cl_uint n,
2      cl_context context, cl_command_queue queue, cl_kernel kernel, size_t workGroupSize) {
3    size_t adjustedSize = roundToPowerOfTwo(n);
4    cl_mem buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, adjustedSize * sizeof(uint),
5        nullptr, nullptr);
6
7    clEnqueueWriteBuffer(queue, buffer, false, 0, n * sizeof(uint), data, 0, nullptr, nullptr);
8    if (adjustedSize != n) {
9      cl_uint max = numeric_limits<uint>::max();
10     clEnqueueFillBuffer(queue, buffer, &max, sizeof(uint), n * sizeof(uint),
11         (adjustedSize - n) * sizeof(uint), 0, nullptr, nullptr);
12   } // if
13
14   for (cl_uint boxwidth = 2; boxwidth <= adjustedSize; boxwidth <<= 1) {
15     for (cl_uint inc = boxwidth >> 1; inc > 0; inc >>= 1) {
16       clSetKernelArg(kernel, 0, sizeof(cl_mem), &buffer);
17       clSetKernelArg(kernel, 1, sizeof(cl_uint), &inc);
18       clSetKernelArg(kernel, 2, sizeof(cl_uint), &boxwidth);
19       size_t threads = adjustedSize / 2;
20       size_t globalWS[] = { threads };
21       size_t localWS[] = { min(workGroupSize, threads) };
22       clEnqueueNDRangeKernel(queue, kernel, 1, nullptr, globalWS, localWS, 0, nullptr, nullptr);
23     } // for
24   } // for
25
26   clEnqueueReadBuffer(queue, buffer, true, 0, n * sizeof(uint), data, 0, nullptr, nullptr);
27
28   clReleaseMemObject(buffer);
29 } // bitonicSortGPU
```

Listing 24: Host code for a bitonic sort implementation. (Bainville 2011).

At first the input's size is rounded up to a power of two (`adjustedSize`). A read- and writable buffer is created with this size and the input data is written to it. If the input was smaller than the created buffer, the remaining part is filled with the maximum representable value of the input data type. This values should remain at the back side of the buffer during the sorting procedure and not disturb the actual input. As sorting networks are compare-and-swap-based no additional buffer is required. The input can be sorted in place. After the buffer is set up we can begin sorting the elements.

As we can see in figure 19, each column of red boxes contains half as many comparisons as inputs, which can be executed in parallel. Therefore, a kernel will be enqueued for every column of red boxes consisting of $\frac{n}{2}$ work items. The distance between two wires of a comparator is equal across all red boxes of a column and will be called `inc`. The red boxes itself are combined in colums of alternating blue and green boxes. The width of these boxes (the number of wires they span) will be called `boxWidth` and determines the initial `inc` (which is a half of the width) of their contained columns of red boxes. Furthermore, the `boxWidth` allows each comparator to derive the sorting direction (ascending or descending) from the wires'/work items' indexes.

The approach is implemented using two nested loops. The outer loop iterates the `boxWidths` which are powers of two (starting with two) until the last blue box, which is equally long to the number of inputs. The inner loop iterates over the distances between two wires of the comparators inside the red box columns (`inc`). This value starts with a half of the `boxWidth` and decreases to the next lower power of two each loop until one. In each iteration of the inner loop a kernel is enqueued. Arguments are the inner loops `inc`, the other loops `boxWidth` and the buffer with the values. The global work size is the number of comparisons of the current red box column which is a half of the input size. The local work size can be chosen freely except it must not be larger than the global one. After all passes have been executed, the sorted values can be read back from device memory. Listing 25 shows the corresponding kernel code for the described bitonic sorter implementation.

```
1  __kernel void BitonicSort(__global uint* data, uint inc, uint boxwidth) {
2    uint id  = get_global_id(0);
3
4    uint low = id & (inc - 1);      // bits below inc
5    uint i   = (id << 1) - low;     // insert 0 at position inc
6    bool asc = (i & boxwidth) == 0; // test bit at boxwidth
7
8    data += i;
9
10   uint x0 = data[0];
11   uint x1 = data[inc];
12
13   if (asc ^ (x0 < x1)) {
14     data[0  ] = x1;
15     data[inc] = x0;
16   } // if
17 } // BitonicSort
```

Listing 25: OpenCL Kernel code for one iteration of a bitonic sort. The implementation is based on an article about experiences with OpenCL and sorting by Eric Bainville (Bainville 2011).

The kernel begins by calculating the index of the first wire for the comparator corresponding to the current work item. This is achieved by modifying the global id. The bits below the `inc` bit (the lower part) stay the same. This value determines the wire index in the upper half of a red box. The bit at position `inc` of the global id corresponds to the upper (0) or lower (1) half of the red box. This bit shall be zero to get the upper wire of the comparator. The remaining (upper) part of the global id above the `inc` bit is a multiple of the width of a red box and therefore determines the index of the red box inside the column. As only $\frac{n}{2}$ work items have been enqueued, the upper part has to be doubled to have box indexes for the full range of wires. The combined value is stored in `i`. The next step is to decide upon the sorting order of the current comparator. This can be done by looking at the bit at position `boxWidth` of the upper wire's index. If this bit is zero, the comparator belongs to a blue box and the sorting order is ascending. Otherwise we are in a green box and have to sort descending. Afterwards the two values at the input wires of the comparator are loaded from global memory. The positions are `i` and `i + inc`. If the values have to be swapped they are written back to global memory in opposite order.

Looking at the performance data in figure 18 we can see that the GPU bitonic sorter performs equally well as C++'s `std::sort`. If we look very closely we can even see that the bitonic sorter's runtime raises a bit faster than the one of the CPU sorting algorithm. This can be explained simply by their runtime complexities which are $\mathcal{O}(n\,log^2\,n)$ for the bitonic network and $\mathcal{O}(n\,log\,n)$ for `std::sort`. When profiling the code the first problem that can be noticed are the number of enqueued kernels, which is 1053 for an input of $2^{26}$ elements. Furthermore, the large number of work items are quite lightweight. The latter is not a problem as such but it is important to notice that here is still space for optimization. Fortunately, both issues can be solved by combining several separate kernel invocations into a larger, heavier kernel. This is also known as kernel fusion and topic of the next section.

### 5.2.2 Bitonic Sort using kernel fusion

The basic concept of kernel fusion is to reduce the amount of redundant global memory loads and stores to the same locations between two or more separate kernel invocations. Using the example of the bitonic sorting network for 16 inputs in figure 19, kernel fusion could be used to combine the kernel invocations for the second and third column of red boxes. Considering only the upper blue box, work item zero and one would load the values of wire zero to three. After the first red box all four values would be stored back to global memory. On the next kernel invocation, work item zero and one would again load the same values for the following two red boxes of the same column. By combining the two separate kernel invocations into one, the redundant global memory store and load between the two kernels could be avoided. This idea can be taken further to combine even more invocations into a single kernel. However, fusing kernels tends to making the work items heavier in resource consumption. Considering the bitonic sorting network where each work item initially had to store two values from global memory, the fused kernel (combining two invocations)

would require storing four values per work item (and reducing the global work size to $\frac{n}{4}$). This value increases by powers of two and therefore sets a limit to the amount of kernels that can be fused together. Listing 26 shows the changes to the previous bitonic sorter host implementation.

```
1  void bitonicSortFusionGPU(uint* data, cl_uint n,
2      cl_context context, cl_command_queue queue, cl_kernel kernel2, cl_kernel kernel4,
3      cl_kernel kernel8, cl_kernel kernel16, size_t workGroupSize) {
4    size_t adjustedSize = roundToPowerOfTwo(n);
5    cl_mem buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, adjustedSize * sizeof(uint),
6        nullptr, nullptr);
7
8    clEnqueueWriteBuffer(queue, buffer, false, 0, n * sizeof(uint), data, 0, nullptr, nullptr);
9    if (adjustedSize != n) {
10     cl_uint max = numeric_limits<uint>::max();
11     clEnqueueFillBuffer(queue, buffer, &max, sizeof(uint), n * sizeof(uint),
12         (adjustedSize - n) * sizeof(uint), 0, nullptr, nullptr);
13   } // if
14
15   for (cl_uint boxwidth = 2; boxwidth <= adjustedSize; boxwidth <<= 1) {
16     for (cl_uint inc = boxwidth >> 1; inc > 0; ) {
17       int ninc = 0;
18       cl_kernel kernel;
19
20       if (inc >= 8) {
21         kernel = kernel16;
22         ninc = 4;
23       } else if (inc >= 4) {
24         kernel = kernel8;
25         ninc = 3;
26       } else if (inc >= 2) {
27         kernel = kernel4;
28         ninc = 2;
29       } else {
30         kernel = kernel2;
31         ninc = 1;
32       } // if
33
34       clSetKernelArg(kernel, 0, sizeof(cl_mem), &buffer);
35       clSetKernelArg(kernel, 1, sizeof(cl_uint), &inc);
36       clSetKernelArg(kernel, 2, sizeof(cl_uint), &boxwidth);
37       size_t threads = adjustedSize >> ninc;
38       size_t globalWS[] = { threads };
39       size_t localWS[] = { min(workGroupSize, threads) };
40       clEnqueueNDRangeKernel(queue, kernel, 1, nullptr, globalWS, localWS, 0, nullptr, nullptr);
41
42       inc >>= ninc;
43     } // for
44   } // for
45
46   clEnqueueReadBuffer(queue, buffer, true, 0, n * sizeof(uint), data, 0, nullptr, nullptr);
47
48   clReleaseMemObject(buffer);
49 } // bitonicSortFusionGPU
```

Listing 26: Changes to the host code from listing 24 for a bitonic sort implementation using kernel fusion. (Bainville 2011).

The outer and inner loop enqueuing the kernels stay the same. However, if the increment `inc` is large enough a fused kernel can be used which processes more than one column of red boxes with multiple increments (decreasing powers of two). The largest fused kernel being used is `kernel16` which processes 16 input values and fuses four invocations with four different increments. Larger kernels are still possible, but showed to consume to much registers to increase performance (register spilling). Nevertheless, larger kernels may be beneficial on newer hardware with more available registers. By using `kernel16` the inner loop's `inc` variable can be incremented to the fourth lower power of two instead of the next one, which is controlled by the `ninc` variable. Kernels with eight, four and two inputs follow analogously. The kernel with two inputs is actually the original one used before kernel fusion. After the appropriate kernel has been chosen it is enqueued with the same parameters as an unfused kernel. Only the number of enqueued work items decreases as each work item now processes more values. The corresponding kernel code is based on the previous section in listing 25. As several very similar kernels have to be created, the preprocessor is used to reduce redundant code.

```
1  #define CONCAT(a, b) a ## b           // concat token a and b
2  #define CONCAT_EXP(a, b) CONCAT(a, b) // concat token a and b AFTER expansion
3
4  inline void order(uint* x, uint a, uint b, bool asc) {
5    if(asc ^ (x[a] < x[b])) {
6      uint auxa = x[a];
7      uint auxb = x[b];
8      x[a] = auxb;
9      x[b] = auxa;
10   } // if
11 } // order
12
13 #define merge1(x, asc)
14
15 #define BITONIC_SORT_FUSION(lvl, logLvl, lvlHalf)                                    \
16   inline void CONCAT_EXP(merge, lvl)(uint* x, bool asc) {                            \
17     for (int j = 0; j < lvlHalf; j++)                                                \
18       order(x, j, j + lvlHalf, asc);                                                 \
19     CONCAT_EXP(merge, lvlHalf)(x, asc);                                              \
20     CONCAT_EXP(merge, lvlHalf)(x + lvlHalf, asc);                                    \
21   } /* merge */                                                                      \
22                                                                                      \
23   __kernel void CONCAT_EXP(BitonicSortFusion, lvl)(__global uint * data, int inc, int boxwidth) { \
24     int id = get_global_id(0);                                                       \
25                                                                                      \
26     inc >>= (logLvl - 1);                                                            \
27     int low = id & (inc - 1);                                                        \
28     int i = ((id - low) << logLvl) + low;                                            \
29     bool asc = ((boxwidth & i) == 0);                                                \
30                                                                                      \
31     data += i;                                                                       \
32                                                                                      \
33     uint x[lvl];                                                                     \
34     for (int k = 0; k < lvl; k++)                                                    \
35       x[k] = data[k * inc];                                                          \
36                                                                                      \
37     CONCAT_EXP(merge, lvl)(x, asc);                                                  \
38                                                                                      \
39     for (int k = 0; k < lvl; k++)                                                    \
40       data[k * inc] = x[k];                                                          \
41   } /* BitonicSortFusion */
42
43 BITONIC_SORT_FUSION(2, 1, 1);
44 BITONIC_SORT_FUSION(4, 2, 2);
45 BITONIC_SORT_FUSION(8, 3, 4);
46 BITONIC_SORT_FUSION(16, 4, 8);
```

Listing 27: OpenCL code of several (fused) kernels performing 1, 2, 3 or 4 iterations (2, 4, 8 or 16 inputs per work item) of a bitonic sort. (Bainville 2011).

The kernel code starts with the `order` procedure which orders two values (at index `a` and `b`) of a given array `x` either ascending or descending depending on the value of `asc`. The following `BITONIC_SORT_FUSION` macro declares the actual kernel plus a merge procedure. Arguments to the macro are the number of input wires the expanded kernel should process (`lvl`) as well as the logarithm of base two for this value (`logLvl`). Finally also the half of the `lvl` argument has to be specified. This value cannot be calculated as it is used in token pasting and macro expansion.

The macro begins by declaring a merge procedure for the current number of input wires. Arguments are a pointer to the array holding the values of the input wires (`x`) and a boolean determining the sort order (`asc`). For the half number of input wires, comparison have to be made (cf. a larger red box in figure 19). Then both halves of the input array are merged using two invocations of the merge procedure of the next lower level (half the current input wires). This is the actual fused part of the kernel which now processes the next two smaller red boxes sharing the same inputs as the initial red box. Although the merge procedure is recursive, it cannot be implemented as such due to OpenCL not allowing recursive function calls. The reason for this is that OpenCL programs do not have a runtime stack for storing local (not `__local`!) variables. They have to be stored in registers and therefore the maximum amount of necessary register space needed by the program has to be determinable at compile time (and will be fully allocated when the kernel is executed). The final `merge2` procedure will not have to call any recursive merges any more. Therefore the macro `merge1` is declared empty (recursion anchor).

After the merge procedur the actual kernel for `lvl` number of inputs follows. The code is basically taken from the previous, non-fused version from the last section. The first adjustment made

is to decrease the kernel's increment to the one of the last fused original kernel's invocation. Furthermore, when constructing the wire index of the first value to load, not only the bit at `inc`'s position is set to zero but `logLvl` bits, as `lvl` values will be loaded instead of two. These values are loaded from global memory starting at the computed index `i` with a stride of `inc`. The core forms the invocation of the bitonic merge on the loaded array[1]. The sorted values are then written back to global memory. Finally the macro is expanded to produce a merge procedure and a kernel for the levels 2, 4, 8 and 16.

The difference in performance can be seen clearly when comparing the fused bitonic sorter against the previous implementation in figure 18. For sorting $2^{26}$ elements the fused variant only requires 3.808 seconds which is roughly two times faster than the original version (7.678 seconds). This is mostly due to reduced global memory traffic by cutting down the number of enqueued kernels which has shrunken to 294 (compared with the initial 1053). In comparison to `std::sort`, which requires 9.039 seconds, this is a speedup of 2.37.

The implementation can still be improved by taking advantage of local memory. Eric Bainville also shows a version of the fused level four kernel using local memory in his excellent article from which most of this bitonic sort approaches are taken from (Bainville 2011). The benchmarks however only show an insignificant boost (several milliseconds) in performance on the cost of a relatively complex kernel. Therefore this idea will not be covered.

Another idea would be to place the loaded values from the wires into local memory as local memory is accessible on a 4-byte boundary (1-byte with OpenCL extension cl_khr_byte_addressable_store). Arrays in registers can only by accessed by index at full register boundaries (16 byte) forcing the compiler to allocate more register space than the initial array requires. By placing arrays in local memory the register usage of work items could be cut down allowing higher occupancy on the cost of slower memory access. With a work group size of 256 (used in all benchmarks and the maximum of the used GPU), a level 32 kernel would already consume the full 32 KiB of local memory on the used GPU. However, when placing the array in registers, a level 32 kernel would already spill registers into global memory which is incredibly slow. Therefore a level 32 kernel might be beneficial when using local memory. Nevertheless, this approach has not been tested with the bitonic sorter. This idea is reconsidered when discussing radix sort in section 5.2.4.

### 5.2.3 Radix Sort

We have already discussed radix sort in the CPU implementation section 5.1.2. Implementing radix sort for the GPU comes down again to three steps which are executed in passes. The first step is to create a histogram for the radix of the current pass. Secondly, the histogram has to be scanned and finally, the input values are permuted according to the scanned histogram.

Creating the histogram would be easy. For each input element a work item is created which determines the histogram element (bucket) which should be incremented. By using atomic operations which are available since OpenCL 1.1 (extension in OpenCL 1.0) the histogram can simply be stored in global memory and concurrently updated from all threads without errors. As global memory atomics are relatively slow, each work group should first compute a histogram in local memory which is then finally reduced to one in global memory.

However, this kind of histogram is of little to no use during the fully parallel permute stage. The reason for this is that every work item must be able to determine the destination address for its value independently of other threads. Although it would be possible for all work items of the same histogram bucket to obtain different destination addresses again by using atomic increments on the histogram (similar to the CPU implementation where each bucket's histogram value is incremented after writing a value to this bucket), it does not work on a GPU. Because OpenCL does not guarantee the order in which work items are executed, work items with higher global id

[1] As the loaded values of the input wires have to be simply sorted, it is possible to replace the bitonic merge by any kind of sorting algorithm.

might write values to a bucket before work items with lower id do. As a consequence the permute pass would not be stable anymore which is crucial to not ruin the permutation of a previous pass.

A solution to this problem can be found by thinking about the information a work item needs to find its correct destination address. In addition to the the start address of the region where all values of the same bucket should be written to (as in a single histogram), each work item also needs to know how many work items having a value belonging to the same bucket have a smaller global id than the current work item. This can be solved by spreading the histogram down to all individual work items. As each work item now owns a separate histogram memory consumption raises enormously. Therefore, multiple input values are processed in each work item. The histograms, after they have been created by the work items, are moved to global memory as such that the histogram entries of the same bucket across all work items lie consecutively and ascendingly (concerning the global id) in memory followed by the next group of buckets and so forth. As a result, the scan step can simply scan the whole block of histogram memory without any special cases. Therefore, the vector scan kernel developed in section 4.5 can be fully reused as building block for this radix sort implementation. Listing 28 shows the host code for an OpenCL radix sort implementation.

```
1  #define RADIX 4
2  #define BUCKETS (1 << RADIX)
3  #define RADIX_MASK (BUCKETS - 1)
4  #define BLOCK_SIZE 32
5  #define VECTOR_WIDTH 8
6
7  void radixSortGPU(uint* data, cl_uint n,
8      cl_context context, cl_command_queue queue, cl_kernel histogramKernel, cl_kernel permuteKernel,
9      cl_kernel scanBlocksKernel, cl_kernel addSumsKernel, size_t workGroupSize) {
10   size_t adjustedSize = roundToMultiple(n, workGroupSize * BLOCK_SIZE);
11   cl_mem srcBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE, adjustedSize * sizeof(uint),
12       nullptr, nullptr);
13   cl_mem dstBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE, adjustedSize * sizeof(uint),
14       nullptr, nullptr);
15
16   clEnqueueWriteBuffer(queue, srcBuffer, false, 0, n * sizeof(uint), data, 0, nullptr, nullptr);
17   if (adjustedSize != n) {
18     cl_uint max = numeric_limits<uint>::max();
19     clEnqueueFillBuffer(queue, srcBuffer, &max, sizeof(uint), n * sizeof(uint),
20         (adjustedSize - n) * sizeof(uint), 0, nullptr, nullptr);
21   } // if
22
23   size_t histogramSize = (adjustedSize / BLOCK_SIZE) * BUCKETS;
24   histogramSize = roundToMultiple(histogramSize, workGroupSize * 2 * VECTOR_WIDTH); // for scan
25   cl_mem histogramBuffer = clCreateBuffer(context, CL_MEM_READ_WRITE,
26       histogramSize * sizeof(cl_uint), nullptr, nullptr);
27
28   size_t globalWS[] = { adjustedSize / BLOCK_SIZE };
29   size_t localWS[] = { workGroupSize };
30
31   for (cl_uint bits = 0; bits < sizeof(uint) * 8; bits += RADIX) {
32     clSetKernelArg(histogramKernel, 0, sizeof(cl_mem), &srcBuffer);
33     clSetKernelArg(histogramKernel, 1, sizeof(cl_mem), &histogramBuffer);
34     clSetKernelArg(histogramKernel, 2, sizeof(cl_uint), &bits);
35     clEnqueueNDRangeKernel(queue, histogramKernel, 1, nullptr, globalWS, localWS,
36         0, nullptr, nullptr);
37
38     scanRecursiveVecGPU_r(histogramBuffer, histogramSize, context, queue, scanBlocksKernel,
39         addSumsKernel, workGroupSize);
40
41     clSetKernelArg(permuteKernel, 0, sizeof(cl_mem), &srcBuffer);
42     clSetKernelArg(permuteKernel, 1, sizeof(cl_mem), &dstBuffer);
43     clSetKernelArg(permuteKernel, 2, sizeof(cl_mem), &histogramBuffer);
44     clSetKernelArg(permuteKernel, 3, sizeof(cl_uint), &bits);
45     clEnqueueNDRangeKernel(queue, permuteKernel, 1, nullptr, globalWS, localWS, 0, nullptr, nullptr);
46
47     std::swap(srcBuffer, dstBuffer);
48   } // for
49
50   clEnqueueReadBuffer(queue, srcBuffer, true, 0, n * sizeof(uint), data, 0, nullptr, nullptr);
51
52   clReleaseMemObject(srcBuffer);
53   clReleaseMemObject(dstBuffer);
54   clReleaseMemObject(histogramBuffer);
55 } // radixSortGPU
```

Listing 28: Host code for a radix sort implementation. (Advanced Micro Devices, Inc. 2013a).

At the top of the source code several macros define constants. The first three should already be known from the previous CPU implementation in section 5.1.2. A major difference is the relatively small radix used with the GPU implementation. This is justified by the huge amount of memory required to store a histogram of $2^{RADIX}$ buckets for each work item. New is the `BLOCK_SIZE` constant which defines how many elements each work item processes. Its value of 32 has been determined by benchmarks and may be specific to the used GPU. The `VECTOR_WIDTH` macro defines the width of the vector types used in the scan kernel (cf. vector scan implementation in chapter 4.5). The host code stars by creating two buffers with the size of the input rounded up to be a multiple of the work group size times the `BLOCK_SIZE`. One will be used to hold the input values at the beginning of each pass (source) and the other one will be used by the permute step to write the output values to (destination). The source buffer is initialized with the unsorted input sequence from host memory. Any free space after the written sequence is filled up with the largest representable value of the input elements' data type. Beside the source and destination buffer a further histogram buffer is needed to store a histogram for each work item. The size of this buffer has to be rounded up to fit the input size requirements for the vector scan. The global work size is the number of input elements divided by `BLOCK_SIZE` as each work item processes `BLOCK_SIZE` values. The local work size can be chosen at will as the algorithm does not depend on local memory or work group synchronization. After the setup is complete we can start executing several sort passes identically to the CPU variant. In each pass a histogram kernel is enqueued which calculates the per work item histograms and stores them into the histogram buffer. Arguments of the kernel are the source buffer containing the input values, the histogram buffer for writing the histograms to and the bit offset of the current pass to select the corresponding bits of the input keys. After the histograms are created and stored in global memory, the histogram buffer can be scanned. The vector scan algorithm (host and kernel) from section 4.5 will be used directly for this job. The scanned histograms are then input to the permute kernel which will reorder the values from the source buffer into the destination buffer. Finally the references to the source and destination buffer are swapped for the next pass. After all passes have been executed the resulting sorted sequence can be read from the source buffer (destination buffer of the last pass). Listing 29 shows the corresponding kernel code.

```
1  #define RADIX 4
2  #define BUCKETS (1 << RADIX)
3  #define RADIX_MASK (BUCKETS - 1)
4  #define BLOCK_SIZE 32
5
6  __kernel void Histogram(__global uint* data, __global uint* histograms, uint bits) {
7    size_t globalId = get_global_id(0);
8
9    uint hist[BUCKETS] = {0};
10   for (int i = 0; i < BLOCK_SIZE; ++i) {
11     uint value = data[globalId * BLOCK_SIZE + i];
12     uint pos = (value >> bits) & RADIX_MASK;
13     hist[pos]++;
14   } // for
15
16   for (int i = 0; i < BUCKETS; ++i)
17     histograms[get_global_size(0) * i + globalId] = hist[i];
18 } // Histogram
19
20 __kernel void Permute(__global uint* src, __global uint* dst, __global uint* scannedHistograms,
21     uint bits) {
22   size_t globalId = get_global_id(0);
23
24   uint hist[BUCKETS];
25   for (int i = 0; i < BUCKETS; ++i)
26     hist[i] = scannedHistograms[get_global_size(0) * i + globalId];
27
28   for (int i = 0; i < BLOCK_SIZE; ++i) {
29     uint value = src[globalId * BLOCK_SIZE + i];
30     uint pos = (value >> bits) & RADIX_MASK;
31     uint index = hist[pos]++;
32     dst[index] = value;
33   } // for
34 } // Permute
```

Listing 29: OpenCL code for the histogram and permute kernels of a radix sort, where each work item stores it's histogram in registers. (Advanced Micro Devices, Inc. 2013a)

The `Histogram` kernel begins by querying the global id of the work item and setting up a zero initialized array for holding the histogram values. Each work item then processes `BLOCK_SIZE`

elements of the input buffer. For each element the bits corresponding to the current pass' bit offset are extracted to increment the appropriate bucket of the work item's histogram. After processing the input elements has finished, the histogram is split into its buckets and moved to global memory.
The `Permute` kernel begins with copying the scanned histogram back from global memory into registers. Each histogram bucket now contains the number of elements belonging into this bucket from work items with lower global id plus the sum of all elements of preceding buckets. Then the work item again reads in the `BLOCK_SIZE` input elements for which the scanned histogram is now available. Each value is written to the destination buffer at the start address of the corresponding bucket for this work item. The address is incremented afterwards.

Considering performance (cf. figure 18), the radix sort GPU implementation is roughly equally fast as the fused bitonic sort. When looked closely, one can even see that radix sort's runtime increases a little bit slower than the one of the bitonic sorter. However, the GPU radix sort cuts off at the last few input sizes as the GPUs global memory is exhausted and the driver starts to swap GPU memory into the systems main memory. Nevertheless, the GPU implementation is far behind the CPU variant with 1.884 seconds at $2^{25}$ (!) elements compared to 0.449 seconds on the CPU. When profiling the kernel one can see that most of the time is spent executing and waiting for global memory requests. If we have a look at the kernel code in listing 29 again, we can see that the `Histogram` kernel performs `BLOCK_SIZE` read operations of four byte (size of uint). Furthermore adjacent work items access global memory with a stride of `BLOCK_SIZE` times four byte. As a result each four byte value from each work item is loaded using a full memory transaction. However, on NVIDIA GPUs memory transactions always load a full memory segment of 64 or 128 words (32-bit values) (NVIDIA Corporation 2009, p.13). Global memory access should therefore be coalesced, meaning that adjacent work items should access adjacent elements in global memory. In this case, all work items can be serviced in as few memory transactions as possible. However, this is not possible in our radix sort implementation unless the `BLOCK_SIZE` is reduced to one. Nevertheless, another way of increasing memory bandwidth utilization is to request fewer but larger pieces of memory. This approach also improves performance on AMD GPUs and will be covered in section 5.2.5. A further but more subtle issue can be seen in the disassembly of both kernels. The local (not `__local`) array `uint hist[BUCKETS]` is placed in a work item's registers. However, a register's size is four words (equal to an `uint4`). To allow indexing the array (runtime indexing into registers is not possible), the array has to be either placed strided into registers (each array element occupies only the first component of each vector register) or the compiler has to generate appropriate code to transform an index into the corresponding register address and vector component. An unorthodox but working solution is to put the array into shared memory which can be addressed at a 4-byte boundary (1-byte with OpenCL extension cl_khr_byte_addressable_store). This idea will be covered in the following section.

### 5.2.4 Radix Sort using local memory

Instead of placing each work item's histogram in it's own registers, local memory is used to store the array. Thus, wasted register space and instruction overhead necessary to access an array placed in registers is avoided. However, accessing local memory is usually slower and accesses might suffer from bank conflicts. This implementation will take the radix sort of the previous section 5.2.3 and move the histogram from registers to local memory. Listing 30 shows the required changes to the host code from listing 28.

```
1  size_t localSize = (workGroupSize * BUCKETS * sizeof(cl_uint));
2  ...
3  for (cl_uint bits = ...) {
4    ...
5    clSetKernelArg(histogramKernel, 3, localSize, nullptr);
6    ...
7    clSetKernelArg(permuteKernel, 4, localSize, nullptr);
```

Listing 30: Changes to the host code of listing 28 for a radix sort implementation using local memory to store each thread's histogram. (Advanced Micro Devices, Inc. 2013a).

As local memory is shared inside a work group, we have to allocate a block of local memory large enough to hold all histograms of all work items of a work group. The local memory needs to be allocated in both kernels. Listing 31 shows the required changes to the kernel code from listing 29.

```
    clSetKernelArg(permuteKernel, 4, localSize, nullptr);
#endif

#if 0
__kernel void HistogramLocal(__global uint* data, __global uint* histograms,
    uint bits, __local uint* hist) {
  size_t globalId = get_global_id(0);
  size_t localId = get_local_id(0);

  hist += localId * BUCKETS;
  for (int i = 0; i < BUCKETS; ++i)
    hist[i] = 0;
  ...
} // HistogramLocal

__kernel void PermuteLocal(__global uint* src, __global uint* dst, __global uint* scannedHistograms,
    uint bits, __local uint* hist) {
  size_t globalId = get_global_id(0);
  size_t localId = get_local_id(0);

  hist += localId * BUCKETS;
  ...
} // PermuteLocal
```

Listing 31: Changes to the OpenCL kernel code of listing 29 for a radix sort implementation using local memory to store each thread's histogram. (Advanced Micro Devices, Inc. 2013a)

Instead of declaring `hist` as local variable inside the kernel, it is now passed as pointer to `__local` memory as an argument to both kernels. The pointer has then to be offsetted to the current work item's histogram place. Except initialization to zero in the `Histogram` kernel, the remaining code stays the same.

Benchmarking the changed code shows a subtle but noticeable difference in performance. The initial 1.884 seconds of the original implementation from section 5.2.3 could be reduced to 1.565 seconds, simply adjusting the code to fit better to the GPUs hardware features. Although this code has executed faster in the benchmarks, newer GPUs might already provide scalar registers having no troubles storing an array of scalar data types.

### 5.2.5 Radix Sort using local memory and vector loads

A further optimization of the presented radix sort implementation is to replace the many small global memory reads in both radix sort kernels by fewer but larger fetches. This can be achieved by declaring the pointers to the input buffers in both kernels to be of a vector type. Therefore the number of iterations of the fetching loops (and therefore the number of fetches) is reduced by a factor of the width of the used vector type. Disadvantages of using vector types may be increased register consumption and increased source code length (vector components cannot be iterated over). As this only concerns the kernel itself, no changes to the host code are required. Listing 32 shows the changes to the previous radix sort implementation using local memory from listing 31.

```
#define BLOCK_SIZE 128
#define BLOCK_SIZE_16 (BLOCK_SIZE / 16)

__kernel void HistogramBlock(__global uint16* data, __global uint* histograms, uint bits,
    __local uint* hist) {
  ...
  for (int i = 0; i < BLOCK_SIZE_16; ++i) {
    uint16 value = data[globalId * BLOCK_SIZE_16 + i];
    uint16 pos = (value >> bits) & RADIX_MASK;
    hist[pos.s0]++;
    hist[pos.s1]++;
    ...
    hist[pos.sF]++;
  } // for
```

```
15    ...
16  } // HistogramBlock
17
18  __kernel void PermuteBlock(__global uint16* src, __global uint* dst, __global uint* scannedHistograms
        ,
19      uint bits, __local uint* hist) {
20    ...
21    for (int i = 0; i < BLOCK_SIZE_16; ++i) {
22      uint16 value = src[globalId * BLOCK_SIZE_16 + i];
23      uint16 pos = (value >> bits) & RADIX_MASK;
24      uint16 index;
25      index.s0 = hist[pos.s0]++;
26      index.s1 = hist[pos.s1]++;
27      ...
28      index.sF = hist[pos.sF]++;
29      dst[index.s0] = value.s0;
30      dst[index.s1] = value.s1;
31      ...
32      dst[index.sF] = value.sF;
33    } // for
34  } // PermuteBlock
```

Listing 32: Changes to the OpenCL kernel code of listing 31 for a radix sort implementation using local memory to store each thread's histogram and vector types for loading values from global memory. (Advanced Micro Devices, Inc. 2013a)

The first difference is the increase of the `BLOCK_SIZE` constant which was 32 and has been raised to 128. It turned out that the larger memory requests now impose a smaller problem concerning the total runtime. Thus, more of them can be made inside each work item (may be specific to a GPU and should therefore be tested individually). To maximize memory throughput the largest supported vector type is chosen for the input buffer which is `uint16`. As a result the fetching loops only need to iterate over a 16th of `BLOCK_SIZE` input elements. The number of required iterations is placed in the macro `BLOCK_SIZE_16`. In each loop iteration a full `uint16` (64 byte) value is fetched from global memory and the target bucket in the histogram is calculated. Now comes the drawback of using vector types. As we cannot loop over the components of the fetched value, we have to write code for each histogram access using a different component separately.

When we look at the benchmark chart in figure 18 we can see that our effort paid off. Compared with the version without vector fetches we could achieve a speedup of 1.5 (1.019 vs. 1.565 seconds). Beside a far better memory bus utilization also the amount of independent instruction increased leading to better ALU utilization. However, despite the optimizations, the GPU radix sort implementation runs still slower than the original CPU code. Reasons are the slow global memory write accesses in both radix sort kernels and the relatively small `RADIX` of four compared to 16 in the CPU variant which leads to eight passes on the GPU compared to only two on the CPU.

## 5.3 Further implementations

Beside the presented kernels a further approach has been implemented and benchmarked which has not been covered. However some experiences may help future developers.

#### Odd-Even Transition (Kipfer and Westermann 2005)

The odd-even transition sorting network from gpu gems 2 chapter 46 (Kipfer and Westermann 2005) has been implemented. However, performance was significantly worse than the other covered algorithms.

Furthermore, several libraries exist supporting GPGPU accelerated sorting routines.

#### clpp (krys@polarlights.net 2011)

The OpenCL Data Parallel Primitives Library is an open source and freely available library offering a few GPU primitives such as a scan and sort implementation. However, the project seems to be inactive (last release in July 2011).

**libCL (Stier 2012)**
libCL is an open-source library with the intention to provide a variety of all kinds of parallel algorithms. Although the list of functions is still small, it provides an implementation of radix sort.

**ArrayFire (AccelerEyes 2013)**
ArrayFire is a commercial GPU software acceleration library provided by AccelerEyes. It provides a lot of GPGPU accelerated algorithms using CUDA and OpenCL including a sorting routine.

## 5.4 Summary and conclusion

Sorting an array of elements is one of the most used routines of a programmer. Numerous algorithms and libraries (also standard libraries) exist targeting the CPU and providing highly optimized and ready-to-use sorting routines. However, the availability of GPU accelerated routines is still limited and mainly consists of some experimental libraries. In this chapter we have seen how to implement both, a comparison based and an integer based sorting algorithm using OpenCL. Contrary to traditional concepts, sorting networks where introduced offering a higher amount of parallelism on the cost of runtime complexity. The bitonic sorter has been discussed as example of how a sorting network can be constructed and operated for a given input size. Kernel fusion has been presented as a further way of optimizing algorithms consisting of multiple OpenCL kernels or kernel invocations. Radix sort has been introduced as example of a fast integer based sorting routine. Beside noticing the bad handling of arrays in registers, the final radix sort implementation once again showed the importance of global memory utilization. In conclusion, a fast comparison based sorting routine could be presented beating C++'s `std::sort` with a speedup of 2.37 (9.039 vs. 3.808 seconds on $2^{26}$ elements). Unfortunately, radix sorting is still faster on the CPU with the OpenCL implementation being more than twice as slow as the simple C++ variant.

# 6 Conclusion

## 6.1 Summary

OpenCL is a free and open standard for general purpose parallel programming across various hardware devices. It is maintained by the Khronos Group and implemented by hardware vendors like NVIDIA, AMD or Intel. OpenCL is natively available for C and C++ although many wrappers for other languages exist. To use OpenCL in a C/C++ application an SDK is needed which contains the required header files (also available from Khronos) and static libraries. For running an application using OpenCL a corresponding driver has to be installed offering the API functions as well as a compiler to create kernels. Furthermore, one or more appropriate hardware devices supporting OpenCL are necessary and may even be used together.

The hardware architectures for which OpenCL can be used are quite different, especially concerning the differences between CPUs and GPUs. Modern consumer CPUs have a small number of independent, high power cores (mostly four or eight) which can be used independently by an application using threads. Synchronization between the threads is easy and threads may be spawned at any time. High throughput can be achieved by distributing the work on all available cores and make use of vector instruction sets like SSE or AVX. Memory is provided as a large block behind a hierarchical cache system.

GPUs however employ a massive amount of less powerful cores (up to several thousands) packed together in multiprocessors. Work is processed in smaller work groups and larger n-dimensional ranges which often occupy hundreds or thousands of threads (which often have to be a multiple of hardware specific resources) executing the same code step by step. Synchronization is limited and the amount of threads used has to be predetermined every time work is offloaded to the GPU. High throughput is achieved by utilizing all components of the GPU (ALU, read and write units, etc.) as much as possible. Branching and synchronization should be avoided and vector types/operations should be used wherever possible to maximize the ALUs throughput. The large global memory is cached but very slow and sensitive concerning access patterns. Local memory is available as programmers-controlled cache but again suffers from misaligned accesses (bank conflicts). However, GPUs can achieve a much higher computational throughput than CPUs when used correctly.

Using OpenCL in an application starts by choosing an available platform/implementation and device. Additionally, a context and a command queue have to be created. Kernels are mostly created directly from their source code which has to be provided at run time. The source code is then compiled and linked into a program for the selected device and the kernels can be queried by name. To provide larger input and to retrieve output from a kernel, buffer objects have to be created. These may be read from and written to, either asynchronously or synchronously to the main application, by enqueuing operations on the command queue. Kernels are also enqueued and executed asynchronously after their arguments have been set.

To show the strengths and weaknesses of OpenCL and GPGPU computing on a few real world examples, three kinds of problems have been chosen for which various approaches have been implemented and benchmarked.

1. At the beginning, multiplying two square matrices was ported to OpenCL. The CPU implementation is trivial but performs badly. Fortunately, highly tuned BLAS libraries exist which can do the multiplication a lot faster on the CPU. However, the first and again quite

simple OpenCL implementation already beats the efficient BLAS sgemm routine. With some optimizations, making the code more complicated, we could push performance even further, achieving a speedup of 36. By the example of matrix multiplication we also took a look into performance analysis using profiler information from AMD's CodeXL.

2. The second problem tackled was the all-prefix sum, a linear algorithm. The CPU implementation can be coded in a minute using a simple loop in a few lines of code. A naive, also quite easy to understand GPU approach performed fairly poor. By trying a more sophisticated tree-based reduce and scatter algorithm, small improvements could be made. Adding local memory and finally also using vector types and applying the algorithm on several layers in the end showed significant cut-downs on the run time (at the cost of hundreds of lines of code). However, the runtime of the CPU could not be beaten.

3. The last implementation chapter focused one of the most famous topics in computer science, sorting. The two standard library routines `qsort` and `std::sort` have been benchmarked together with a radix sort implementation, the latter delivering amazing results. For sorting on the GPU, sorting networks have been introduced, because they are easy to parallelize. In particular, the bitonic sorter has been implemented and optimized. Despite the nasty input size restriction, the GPU implementation runs more than twice as fast as the comparison based variants on the CPU, which can be clearly seen as success. Radix sort, however, suffers from the GPUs highly parallel nature. Despite several optimizations it could not catch up and remains far behind the CPU implementation.

## 6.2 Conclusion

In conclusion it can be said that GPUs are different from CPUs in the way they process work. This originates from their quite dissimilar hardware architectures as we discussed in the beginning. But not only the hardware itself, also the way algorithms have to be designed and programs have to be written differs largely from traditional single-thread-orientated CPU algorithms. Although it is already quite common to assign several tasks to several CPU cores (e.g., GUI and work thread), bringing parallelism down to small routines is still far from being common. When a program is designed to be run on massively parallel hardware we tend to spend more time on fighting with the peculiarities of the hardware than thinking about the actual problem we try to solve. This was one of the hardest challenges when implementing scan. The scan operation imposes high data dependency between the processed elements and therefore makes it perfectly suitable for as few processing units as possible. The GPU however benefits from a maximum of parallelism, on data and instruction level. Most of the time required to develop an efficient GPU scan was spent on trying to get enough parallelism on the cost of minimal redundancy in additions. Although the tree based approach worked well in the end, a lot of computing power is wasted during the up-sweep and down-sweep phase inside each work group. The same problem affected the radix sort GPU implementation. The per work item histogram was only required because the GPU does not ensure the order in which work items are processed. And to improve memory reading performance for the scan step by aligning the histogram buffer for coalesced access, the scattered, unblocked writes of the histograms at the end of the permute step also teared down performance, because GPUs profit enormously from reading and writing larger, consecutive blocks of memory. On the CPU, apart from worse caching, scattered accesses is less of a problem. The final argument, that should arise from this observations, is that programmers have to worry about a lot more different stuff than they have to when coding for a processor. Imagine Java or C# programmers who are used to object-oriented design, dynamic memory allocation, huge standard libraries, containers and other complex data structures and all that candy of modern languages, when they find themselves porting one of their well-written, readable algorithms to the GPU where they have to break everything down into well-layouted arrays, think about the vectorizability of the produced instructions, repeatedly check the API documentation for a small number of built-in functions that might be of use and hunt down performance issues to stalled memory reads and bank conflicts. GPGPU computing is far different from modern software design and programming. Although the API is easily understood and the kernel language quickly learned, it takes months

and probably years of practice to get a good feeling for designing GPU accelerated algorithms. GPUs are technological marvels offering immense computational power if used correctly. But it takes time to learn and study them in order to unfold their full potential.

Furthermore, it has to be said that not all algorithms fit the requirements of a GPU. In fact, only a small subset really benefits from the massively parallel hardware. In most cases, it is easier to focus on spreading a workload on several cores of the CPU instead of dealing with thousands of threads. Also the required memory transfer to and from the GPU is a crucial factor when deciding to offload a piece of work or not. If an algorithm does not fit the GPU one should not execute it there. However, the small subset of algorithms which is often parallel in its nature may benefit tremendously from graphics hardware. We have discussed such a kind of algorithm on the example of multiplying matrices. The final speedup of 36 is amazing and shows the potential of GPU hardware when used at the right place.

So, when should OpenCL be used? Apart from having the right kind of algorithm to take advantage of GPGPU acceleration, also time and budget play an important role. Developing an algorithm for the GPU can be tedious and cumbersome and therefore time consuming. Furthermore, GPUs also impose a higher risk of failing to achieve the desired performance boost. The scan algorithms with optimizations may have taken two weeks to develop with the final outcome that the CPU variant is still faster. Experimenting with GPUs in a real world project should therefore be considered well, also concerning the aspect of system requirements. From a developers perspective, appropriate GPUs have to be available for implementing and testing and additional tools have to be installed for debugging and profiling. Concerning customers, appropriate hardware has to be available, drivers have to be installed and maybe the case of missing compatible hardware or drivers has to be handled (fallback CPU implementation). Talking about tools, the currently available suits from AMD (CodeXL), NVIDIA (Nsight, Visual Profiler) and Intel (SDK for OpenCL Applications) already provide decent help when developing software for their respective GPUs. However, they are still quite away from being as suitable and feature-rich as corresponding CPU debugger integration and profilers. Last but not least also maintainability and reusability play a vital role in modern software companies. Both aspects suffer in most cases as OpenCL algorithms tend to be optimized strongly to a specific kind of hardware and problem.

Finally, we have to see how long GPGPU computing will be done in this fashion. With the increasing power of on-chip GPUs on modern Intel processors, using GPUs for smaller tasks might become more attractive as no memory transfers will be required. AMD is also working intensively on a new generation of processing devices called Accelerated Processing Units (APU). By using a so-called heterogeneous system architecture (HSA), GPU and CPU are combined tightly on a single chip. The GPU will have direct access to the system's main memory with cache coherence to the CPU using the same address space as the CPU (theoretically allowing pointers to be passed). Furthermore, the GPU will allow multitasking via context switches between running tasks. As a result, long-running calculations will no longer freeze the display. However, in another corner of the high performance hardware sector, Intel is heavily working on their new Many Integrated Core (MIC) architecture. The Intel Xeon Phi (previously named Knights Corner) is a co-processor card with up to 61 individual Intel processors and up to 16 GiB dedicated on-card memory. From an OpenCL programmers perspective, the Xeon Phi is equally programmed as a GPU. But the card is composed of modified x86 processors and can therefore also execute conventional software written in traditional languages. As a result, language integration is also easy. In fact, Intel's compiler already offers appropriate `#pragma`s to offload parts of a C++ program to the co-processor card in a similar fashion as OpenMP. In addition, almost all drawbacks of GPUs vanish as all threads (four cores on each processor) can be fully synchronized, memory can be allocated dynamically on demand and all existing language features including libraries can be used. However, a Xeon Phi is still financially out of reach for consumers and still more expensive than decent GPUs with prices among several thousand dollars. But as time passes by, they may become affordable. Intel already announced that the next generation of the Xeon Phi (codename Knights Landing) will be capable of being used as the systems main processor. So maybe in ten years we find ourselves with a hundred main processors in our notebooks capable of handling even computationally expensive software with ease. Who needs GPU computing then?

## 6.3 Personal experiences

### 6.3.1 Progress

I began working on this thesis on 2012-07-23 by creating the code base for the sort project. Since then, 415 days have passed (today is the 2013-09-11). 436 commits have been made during this long period of time. The individual commit dates clearly mirror certain situations in my life at the time they were made. Figure 20 presents how the lines of code have evolved over time, figure 21 presents the commits made in this time.

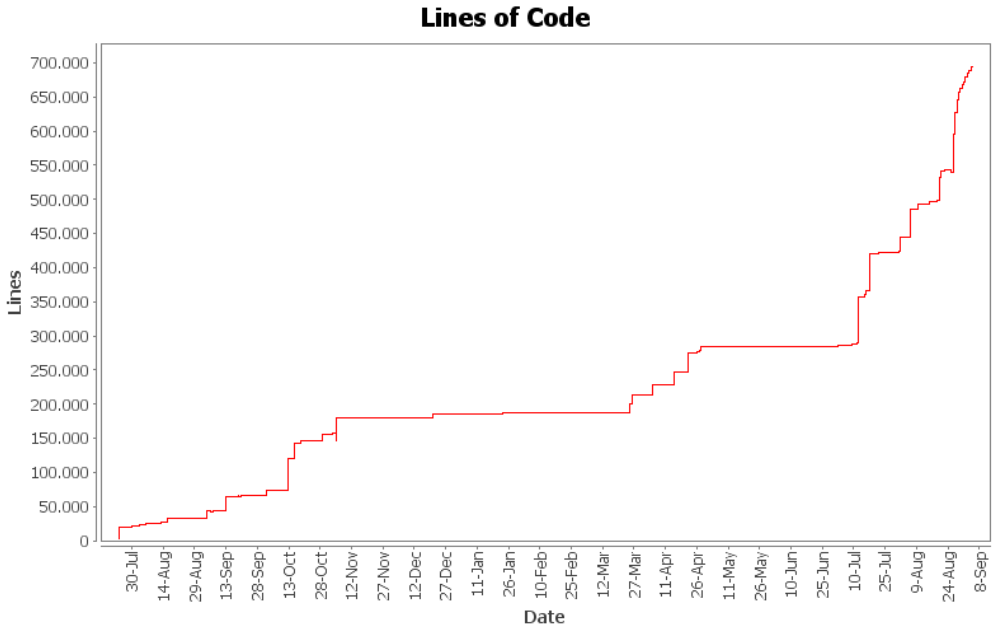


Figure 20: Lines of code of all files in the repository over time. Chart generated using StatSVN.

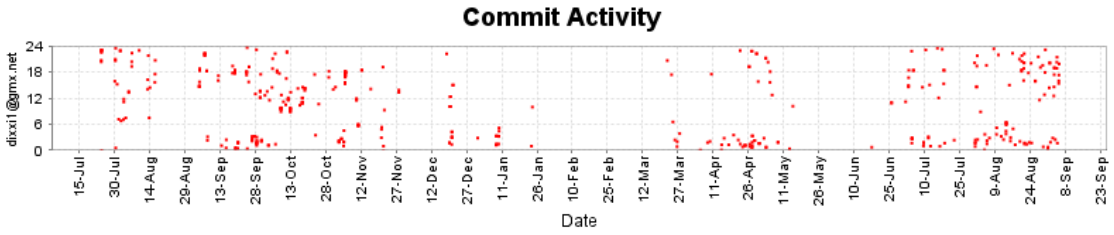


Figure 21: My commits over time. Chart generated using StatSVN.

During the first days I worked intensively on setting up a comprehensive testing environment which should allow easy integration of new algorithms. At the beginning of my internship during the summer holidays in 2012, the commit dates started to occur in the morning (I tried to work two hours on this thesis before going to work), some at lunch time when I had come up with something during my work, but most of them in the late night. I soon realized an impact on my well-being from all the day and night programming and stopped working on my thesis several weeks later (cf. mid to late August in figure 21). However, my motivation for the internship also dropped and I had a bad conscience from showing no progress on my thesis. I quit my internship three weeks earlier to enjoy the remaining summer holidays and concentrate on the thesis. Fortunately, I could collect and integrate a lot of algorithms in this time from the beginning of September into the first weeks of university until the mid of October (cf. figure 21). Although I found less time from October to November due to studies, I could finish most of the code base and collected several benchmark data. My supervisor, FH-Prof. DI Dr. Heinz Dobler, and I were confident that

I should be able to easily finish the thesis until February 2013, the first official data for handing in.

However, I made the mistake of accepting two tutorials (Programming and Project Engineering). Together with the increasing work load of my studies (mostly the large software projects by FH-Prof. DI Johann Heinzelreiter), I found almost no time to start writing. The two groups of commits around the 2012-11-20 and 2013-01-09 in figure 21 were for the two presentations about the thesis in the bachelor seminar course. Apart from them, development was halted until the end of march, which is can be clearly seen in the advancement of the lines of codes in figure 20. After the end of the sixth semester at the end of March, I found again time to continue with my writing. I started to work on the introduction and OpenCL chapter.

I began my mandatory internship, which is part of the Software Engineering course, at the 2013-04-04. From the time of my commits during April in figure 21 one can see that I was again writing on my thesis after work, sometimes until the morning hours. I found myself in the same situation as during my first internship during the summer, except there was no option to prematurely quit the internship. I was even working with the same technologies during work and at home. I lost most of my motivation and interest after a month and stopped working on my thesis after finishing the OpenCL chapter, which I sent to my supervisor. I also fought with bad health conditions, both physical and psychical.

I had the chance to learn a lot during my internship. However, as I was working with the same technologies as I use in my thesis, I came up with more and more design mistakes and ideas for improving my algorithms. During the last month of my internship, I felt the need to refactor most of the code produced during September and October 2012. This was the time where most of the final algorithms and sophisticated charts were created. By further tweaking the algorithms, I gathered a lot of in-depth knowledge about my implementations (which have mostly found their way into the matrix multiplication chapters). A finished my internship by the end of June, fortunately with amazing results and satisfied colleagues.

During the remaining time until the final deadline for the second bachelor exam (which is at the mid of September), I was working hard to write the implementation chapters. One of the big problems was the huge amount of information I had gathered around my implementations (mostly performance aspects). I was proud of all the peculiarities I found out about my GPU and how it behaves when processing my algorithms. However, time was of essence and I had to carefully decide on the algorithms and analyses to include. E.g., only the matrix multiplication contains a detailed view on performance counters. There was simply no time to include this kind of information in the other implementation chapters (although I would have loved to). Beside a week in France (the small gap around the end of July in figure 21) I used every day of my summer holidays for writing on my thesis. I finished the "Release Candidate" on 2013-08-25.

The remaining two weeks until the deadline, I was fully occupied with the creating of the second part of my thesis, which I surprisingly managed to finish during this time, producing an acceptable result. (Funny observation: During this time I also managed to change my sleep-wake cycle to more normal conditions, which can be seen at the commit times in figure 21). The last few days I was working on applying the feedback to my "Release Candidate".

### 6.3.2 Positive and negative aspects

Below is a list stating several positive (✓) and negative (✗) aspects encountered during the creation of this thesis:

- ✓ I started early with the implementation of my thesis at the beginning of my summer holidays in 2012.
- ✓ My supervisor gave me feedback to my first part of the thesis (Introduction and OpenCL chapters) which several hours. Amazing!
- ✓ Although I handed in my "Release Candidate" version of the thesis two weeks before the deadline, my supervisor still took for giving me excellent, detailed feedback and moved the deadline a few days to provide me enough time to include the feedback into the thesis.
- ✓ I was allowed to write the thesis and hold my presentations in English which helped me to improve my command of this language.
- ✓ Writing the thesis in LaTeXgave me the chance to learn a new way of creating professional looking documents.
- ✓ I deepened my knowledge about GPUs and OpenCL tremendously, which will hopefully be beneficial for my future career.
- ✓ Although I was running late with handing in my thesis, my supervisor did not put additional pressure on me. He acted understanding, helpful and diplomatic.
- ✓ The initial knowledge about OpenCL which I gathered during the summer holidays in 2012 qualified me for an interesting internship.
- ✓ A learned how to setup a cross platform and cross language (C++ and Fortran) build system with Code::Blocks and make.
- ✓ A learned about several differences between the used compilers (GNU g++, Microsoft Visual Studio, Intel C++).
- ✗ I have selected to many subtopics/problems I wanted to cover in my thesis. It would probably have been a better idea to concentrate only on one problem (like matrix multiplication) and rename the thesis accordingly.
- ✗ Although LaTeXdoes an amazing job when used correctly, trying to achieve extraordinary formatting (like having dots after a chapter number but not after a section number) is often tedious or impossible without having to hack into internal mechanisms.
- ✗ The curriculum of the fifth semester does not include any time required to write the bachelor thesis. Other courses provide time and even award ETCS for writing the thesis. Furthermore, the additional month of studies before the internship in the sixth semester also consumes a lot of time which would be much needed for writing the bachelor thesis. I cannot provide a solution (there is probably not an easy one without having to reduce the amount of covered material), but I want to point this out.
- ✗ I should have focused less on improving the performance of my algorithms as it took me a lot of time and was not required for the thesis (many improved versions were included though).
- ✗ I should have started earlier to write the actual document. This would have helped me to select the relevant algorithms sooner, focusing my work on code which will eventually become part of the final product.

# List of Figures

# List of Listings

# References


AccelerEyes (2013). *ArrayFire*. Version 1.9. URL: http://www.accelereyes.com/arrayfire_tour (visited on 2013-08-06).

Advanced Micro Devices, Inc. (2013a). *Accelerated Parallel Processing (APP) SDK*. Version 2.8. URL: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/ (visited on 2013-07-09).

— (Apr. 2, 2013b). *Accelerated Parallel Processing Math Libraries (APPML)*. Version 1.10. URL: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-math-libraries/ (visited on 2013-07-17).

— (2013c). *CodeXL*. Version 1.2.3897.0. URL: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/codexl/ (visited on 2013-07-09).

*BLAS (Basic Linear Algebra Subprograms)* (2011). URL: http://www.netlib.org/blas/ (visited on 2013-07-10).

Bainville, Eric (June 25, 2011). *OpenCL Sorting*. URL: http://www.bealto.com/gpu-sorting_intro.html (visited on 2013-08-14).

Batcher, Ken E. (1968). "Sorting networks and their applications". In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. AFIPS '68 (Spring), pp. 307–314.

Bitonic (Oct. 8, 2012). *File:BitonicSort1.svg*. Wikipedia. URL: http://en.wikipedia.org/wiki/File:BitonicSort1.svg (visited on 2013-08-07).

Blelloch, Guy E. (1989). "Scans as primitive parallel operations". In: *Computers, IEEE Transactions on* 38.11, pp. 1526–1538. ISSN: 0018-9340.

— (1990). "Prefix Sums and Their Applications". In: *Sythesis of parallel algorithms*. Morgan Kaufmann Publishers Inc., pp. 35–60. URL: http://www.cs.cmu.edu/~guyb/papers/Ble93.pdf (visited on 2013-07-31).

Fernando, Randima and Mark J. Kilgard (2003). *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321194969.

Gaster, Benedict et al. (2011). *Heterogeneous Computing With OpenCL*. Morgan Kaufmann Publishers Inc. ISBN: 9780123877666.

Harris, Mark, Shubhabrata Sengupta, and John D. Owens (2008). *GPU Gems 3*. GPU Gems. Addison Wesley Professional. Chap. 39. Parallel Prefix Sum (Scan) with CUDA. URL: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html.

Hillis, W. Daniel and Jr. Guy L. Steele (Dec. 1986). "Data parallel algorithms". In: *Commun. ACM* 29.12, pp. 1170–1183. ISSN: 0001-0782. URL: http://cva.stanford.edu/classes/cs99s/papers/hillis-steele-data-parallel-algorithms.pdf (visited on 2013-07-31).

Kanellos, Michael (Feb. 10, 2003). *Moore's Law to roll on for another decade*. CNET. URL: http://news.cnet.com/2100-1001-984051.html (visited on 2013-03-25).

Kanter, David (Dec. 14, 2010). *AMD's Cayman GPU Architecture*. URL: http://www.realworldtech.com/cayman/ (visited on 2013-08-20).

Kipfer, Peter and Rüdiger Westermann (2005). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. GPU Gems. Addison-Wesley Professional. Chap. Chapter 46. Improved GPU Sorting. ISBN: 0321335597. URL: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html.

Lang, Hans W. (Dec. 7, 1998). *Bitonic sorting network for n not a power of 2*. URL: http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm (visited on 2013-08-16).

McClanahan, Chris (2010). *History and Evolution of GPU Architecture*. Georgia Tech College of Computing. URL: http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf (visited on 2013-03-23).

Merritt, Rick (June 20, 2011). *OEMs show systems with Intel MIC chips*. URL: http://www.eetimes.com/electronics-news/4217092/OEMs-show-systems-with-Intel-MIC-chips (visited on 2013-04-09).

Micikevicius, Paulius (June 6, 2012). *GPU Performance Analysis and Optimization*. NVIDIA Corporation. URL: http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0514-GTC2012-GPU-Performance-Analysis.pdf (visited on 2013-04-15).

Moore, Gordon E. (Apr. 19, 1965). "Cramming more components onto integrated circuits". In: *Electronics*. URL: http://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf (visited on 2013-03-25).

Munshi, Aaftab, ed. (Nov. 14, 2012). *The OpenCL Specification*. Version 1.2. Khronos Group. URL: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf (visited on 2013-04-08).

NVIDIA Corporation (July 2009). *NVIDIA OpenCL Best Practices Guide*. URL: http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf (visited on 2013-08-19).

— (2013a). *CUBLAS (CUDA Basic Linear Algebra Subroutines)*. URL: https://developer.nvidia.com/cublas (visited on 2013-07-17).

— (2013b). *GeForce GTX 580. Specifications*. URL: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications (visited on 2013-04-04).

— (Jan. 28, 2013c). *NVIDIA's Next Generation CUDA$^{TM}$ Compute Architecture: Kepler TM GK110. The Fastest, Most Efficient HPC Architecture Ever Built*. Version 1.0. URL: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (visited on 2013-04-13).

Shimpi, Anand Lal and Ryan Smith (Apr. 23, 2012). *The Intel Ivy Bridge (Core i7 3770K) Review*. URL: http://www.anandtech.com/show/5771/the-intel-ivy-bridge-core-i7-3770k-review (visited on 2013-04-21).

StefanVanDerWalt (Oct. 4, 2010). *File:Matrix multiplication diagram 2.svg*. Wikipedia. URL: http://en.wikipedia.org/wiki/File:Matrix_multiplication_diagram_2.svg (visited on 2013-04-28).

Stier, Jochen (July 2012). *libCL*. URL: http://www.libcl.org/ (visited on 2013-08-19).

Strassen, Volker (1969). "Gaussian Elimination is not Optimal." ger. In: *Numerische Mathematik* 13, pp. 354–356. URL: http://eudml.org/doc/131927.

Tompson, Jonathan and Kristofer Schlachter (Apr. 24, 2012). *An Introduction to the OpenCL Programming Model*. New York University. URL: http://www.cs.nyu.edu/~lerner/spring12/Preso07-OpenCL.pdf (visited on 2013-07-08).

Volkov, Vasily (Sept. 22, 2010). *Better Performance at Lower Occupancy*. UC Berkeley. URL: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf (visited on 2013-07-15).

Warfield, Bob (Sept. 6, 2007). *A Picture of the Multicore Crisis*. URL: http://smoothspan.wordpress.com/2007/09/06/a-picture-of-the-multicore-crisis/ (visited on 2013-03-25).

krys@polarlights.net (July 2011). *OpenCL Data Parallel Primitives Library*. Version V1.0 beta 3. URL: http://code.google.com/p/clpp/ (visited on 2013-08-06).

# A Utility functions

Listing 33 shows the definition of often used utility functions.

```
size_t roundToPowerOfTwo(size_t x) {
    --x;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    if (sizeof(size_t) >= 8)
        x |= x >> 32;

    return x + 1;
} // roundToPowerOfTwo

size_t roundToMultiple(size_t x, size_t multiple) {
    if (x % multiple == 0)
        return x;
    else
        return (x / multiple + 1) * multiple;
} // roundToMultiple
```

Listing 33: Utility functions used in host codes throughout the thesis

# B Benchmark environment

All benchmarks have been created using the following software and hardware configuration:

- Windows 7 x64
- Intel Core i5 M 480
  - 2 physical cores (4 virtual with Hyper Threading)
  - 2.67 GHz (2.93 GHz with Turbo Boost)
- 4 GiB DDR3 RAM system memory
- AMD Radeon HD 5650
  - 400 shader cores
  - 1 GiB memory

## OpenCL device info of GPU

The following information has been retrieved by calling `clGetDeviceInfo` with all defined constants in OpenCL 1.2. For a complete list and further explanation visit the corresponding API documentation at Khronos' Website.

| **Flag** | **Value** |
|---|---|
| CL_DEVICE_ADDRESS_BITS | 32 |
| CL_DEVICE_AVAILABLE | 1 |
| CL_DEVICE_BUILT_IN_KERNELS | |
| CL_DEVICE_COMPILER_AVAILABLE | 1 |
| CL_DEVICE_DOUBLE_FP_CONFIG | |
| CL_DEVICE_ENDIAN_LITTLE | 1 |
| CL_DEVICE_ERROR_CORRECTION_SUPPORT | 0 |
| CL_DEVICE_EXECUTION_CAPABILITIES | CL_EXEC_KERNEL |
| CL_DEVICE_EXTENSIONS | cl_khr_global_int32_base_atomics<br>cl_khr_global_int32_extended_atomics<br>cl_khr_local_int32_base_atomics<br>cl_khr_local_int32_extended_atomics<br>cl_khr_3d_image_writes<br>cl_khr_byte_addressable_store<br>cl_khr_gl_sharing<br>cl_ext_atomic_counters_32<br>cl_amd_device_attribute_query<br>cl_amd_vec3<br>cl_amd_printf<br>cl_amd_media_ops<br>cl_amd_media_ops2<br>cl_amd_popcnt<br>cl_khr_d3d10_sharing |
| CL_DEVICE_GLOBAL_MEM_CACHE_SIZE | 0 |

| | |
|---|---|
| CL_DEVICE_GLOBAL_MEM_CACHE_TYPE | CL_NONE |
| CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE | 0 |
| CL_DEVICE_GLOBAL_MEM_SIZE | 1073741824 |
| CL_DEVICE_HALF_FP_CONFIG | |
| CL_DEVICE_HOST_UNIFIED_MEMORY | 0 |
| CL_DEVICE_IMAGE_SUPPORT | 1 |
| CL_DEVICE_IMAGE2D_MAX_HEIGHT | 16384 |
| CL_DEVICE_IMAGE2D_MAX_WIDTH | 16384 |
| CL_DEVICE_IMAGE3D_MAX_DEPTH | 2048 |
| CL_DEVICE_IMAGE3D_MAX_HEIGHT | 2048 |
| CL_DEVICE_IMAGE3D_MAX_WIDTH | 2048 |
| CL_DEVICE_IMAGE_MAX_BUFFER_SIZE | 65536 |
| CL_DEVICE_IMAGE_MAX_ARRAY_SIZE | 2048 |
| CL_DEVICE_LINKER_AVAILABLE | 0 |
| CL_DEVICE_LOCAL_MEM_SIZE | 32768 |
| CL_DEVICE_LOCAL_MEM_TYPE | CL_LOCAL |
| CL_DEVICE_MAX_CLOCK_FREQUENCY | 450 |
| CL_DEVICE_MAX_COMPUTE_UNITS | 5 |
| CL_DEVICE_MAX_CONSTANT_ARGS | 8 |
| CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE | 65536 |
| CL_DEVICE_MAX_MEM_ALLOC_SIZE | 536870912 |
| CL_DEVICE_MAX_PARAMETER_SIZE | 1024 |
| CL_DEVICE_MAX_READ_IMAGE_ARGS | 128 |
| CL_DEVICE_MAX_SAMPLERS | 16 |
| CL_DEVICE_MAX_WORK_GROUP_SIZE | 256 |
| CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS | 3 |
| CL_DEVICE_MAX_WORK_ITEM_SIZES | 256,256,256 |
| CL_DEVICE_MAX_WRITE_IMAGE_ARGS | 8 |
| CL_DEVICE_MEM_BASE_ADDR_ALIGN | 2048 |
| CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE | 128 |
| CL_DEVICE_NAME | Redwood |
| CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR | 16 |
| CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT | 8 |
| CL_DEVICE_NATIVE_VECTOR_WIDTH_INT | 4 |
| CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG | 2 |
| CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT | 4 |
| CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE | 0 |
| CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF | 0 |
| CL_DEVICE_OPENCL_C_VERSION | OpenCL C 1.2 |
| CL_DEVICE_PARENT_DEVICE | 0 |
| CL_DEVICE_PARTITION_MAX_SUB_DEVICES | 5 |
| CL_DEVICE_PARTITION_PROPERTIES | |
| CL_DEVICE_PARTITION_AFFINITY_DOMAIN | |
| CL_DEVICE_PARTITION_TYPE | |
| CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR | 16 |
| CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT | 8 |
| CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT | 4 |
| CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG | 2 |
| CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT | 4 |
| CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE | 0 |
| CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF | 0 |
| CL_DEVICE_PRINTF_BUFFER_SIZE | 1048576 |
| CL_DEVICE_PREFERRED_INTEROP_USER_SYNC | 1 |
| CL_DEVICE_PROFILE | FULL_PROFILE |
| CL_DEVICE_PROFILING_TIMER_RESOLUTION | 1 |
| CL_DEVICE_QUEUE_PROPERTIES | CL_QUEUE_PROFILING_ENABLE |
| CL_DEVICE_REFERENCE_COUNT | 1 |
| CL_DEVICE_SINGLE_FP_CONFIG | CL_FP_INF_NAN<br>CL_FP_ROUND_TO_NEAREST<br>CL_FP_ROUND_TO_ZERO<br>CL_FP_ROUND_TO_INF<br>CP_FP_FMA |
| CL_DEVICE_TYPE | CL_DEVICE_TYPE_GPU |

| | |
|---|---|
| CL_DEVICE_VENDOR | Advanced Micro Devices, Inc. |
| CL_DEVICE_VENDOR_ID | 4098 |
| CL_DEVICE_VERSION | OpenCL 1.2 AMD-APP (1124.2) |
| CL_DRIVER_VERSION | 1124.2 (VM) |