Auto-tunable GPU BLAS

Geir Josten Lien

Master of Science in Informatics
Submission date: June 2012
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

NTNU – Trondheim
Norwegian University of
Science and Technology

Auto-tunable GPU BLAS

Geir Josten Lien

Master of Science in Informatics

Submission date:	June 2012
Supervisor:	Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem description

This project focuses on developing techniques for auto-tunable BLAS. In particular, it looks at such BLAS not only for CUDA, but also OpenCL. One or more such BLAS routine(s) will be developed and tested in the framework. This work will be built on/extend master's thesis work by Jarle E. Steinsland[1].

Assignment given: 22. August 2011
Supervisor: Dr. Anne Cathrine Elster, IDI

Abstract

In this paper, we present our implementation of an Auto tuning system, written in C++, which incorporate the use of OpenCL kernels. We deploy this approach on different GPU architectures, evaluating the performance of the approach. Our main focus is to easily generate tuned code, that would otherwise require a large amount of empirical testing, and then run it on any kind of device. This is achieved through the auto tuning framework, which will create different kernels, compile and run them on the device and output the best performing kernel on the given platform.

BLAS is much used in performance critical applications, and is a good candidate for execution on GPUs due to its potential performance increase. Our implementation was benchmarked on various of test environments, with different GPUs, where we achieved comparable results to the ViennaCL library. We also tested against the native vendor specific BLAS libraries from AMD and NVIDIA.

Acknowledgements

This thesis was completed at the High Performance Computing Group at the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU).

First i would like to thank my supervisor Dr. Anne C. Elster, for her support, guidance and the opportunity to write my thesis at the HPC-lab. A big thanks to my co-students at the HPC-lab, for helpful insight and good work-environment: Johannes Kvam, Kjetil Babington, Frederik Vetre, Jan Rovde, Thomas Falch and Rune Erlend Jensen. Finally i want to thank my friends and family for their support through my two last semesters, writing this thesis.

Trondheim, June 2012

Geir Josten Lien

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

"The way the processor industry is going is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it." (Steve Jobs, NY Times interview, June 10 2008)

High performance computing differs from general computing by the large need for computational resources. In many scientific applications, or problems, the need for computational power is the main constraint. By adding more computational power, one can model more complex problems, which would then resemble the reality more closely. Adding computational power does not come for free: It is important to have optimized kernels, so that the higher-level applications can make efficient use of the available resources.

The implementation of Basic Linear Algebra Subprogram (BLAS) interface is a major component of dense linear algebra libraries, and therefor has to be highly optimized. After the introduction of shared memory in GPUs, this also applies for GPU computing. In the beginning of General-purpose computing on graphics processing units (GPGPU) there existed no programming framework, so programmers had to write a non-graphics application code with graphics API, such as OpenCL with shader programs. However, the industry soon released higher level APIs to make the programming of the GPUs easier. Although many different frameworks exist, CUDA and OpenCL are the most popular ones. These frameworks hide the complexity of GPGPU programming, and allows the programmer to develop application code without any deep knowledge about the GPU hardware specifications.

This changes when the programmer wants to optimize the code to utilize the GPU to its full potential. Then the programmer must take into account various factors related to the GPU hardware. When using OpenCL to maximize the performance of the code, memory access optimization and execution parameter tuning is essential.

The job of optimizing code is largely done by programmers, which spend countless hours modifying the code; trying to exploit performance enhancing architectural features. These features will vary from one system to another, depending on what hardware and architecture it has. This customizing results in that the code will face performance problems when ported to another platform, due to the architectural differences between different platforms. Therefor the whole tuning process must be somewhat repeated when porting. This is where software automatic tuning becomes relevant. The term is often abbreviated to *automatic tuning* or *autotuning*.

1.1 Goal

OpenCL is a relatively new framework, and the number of implementations that apply it to solve BLAS operations are steadily growing, however, CUDA is a more widespread API. This thesis addresses the need for applications that applies OpenCL to increase performance on BLAS operations. It tries to eliminate some of the time-consuming work by implementing an auto-tuner, which will do most of the trial-and-error for the programmer.

1.2 Outline

The rest of this thesis will have the following organization. In section 2 we review related work, and point out some of the problems of auto-tuning and GPGPU programming. Section 3 we propose an auto-tuner which uses OpenCL to speed up BLAS operations on a GPU; describing the different parts of the implementation. In section 4, the proposed implementation is evaluated on different hardware setups. Finally, section 5 gives concluding remarks and future work on the implementation.

Chapter 2

Background and Previous Work

The number of articles concerning matrix and vector multiplication are substantial. However, the number of these that discuss the use of auto-tuning towards GPUs, are less frequent. However, General-Purpose computation on Graphics Processing Units (GPGPU) is becoming more and more popular. OpenCL is used, but the number of implementations that achieve good performance compared to the different existing implementations provided by either manufacturers or research groups is small. This thesis continues the work started by Jarle Erdal Steinsland[1]. Regrettably none of the code created by Steinsland was available.

2.1 BLAS

When solving basic linear algebra on computers, the standard framework applied is *Basic Linear Algebra Subprograms* (BLAS[1]). The BLAS routines consists of three levels: The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations[5], and the Level 3 BLAS perform matrix-matrix operations [6]. Due to the way it is implemented, it is not very fast nor optimal. However, because the BLAS are portable and widely available, they are often used in development of high quality linear algebra software. Some of these implementations are provided by the different computer vendors, and are optimized towards specific hardware.

[1]http://www.netlib.org/blas/

2.1.1 Gemm

The gemm routines perform a matrix-matrix operation with general matrices. The operation is defined as:

$$C = \alpha * op(A) * op(B) + \beta * C \tag{2.1}$$

Where:

- op(x) is one of; op(x) = x, or op(x) = x′, or op(x) = conjg(x′)
- A, B and C are matrices
- A is a m-by-k matrix
- B is a k-by-n matrix
- C is a m-by-n matrix
- α (alpha) and β (beta) are scalars

2.1.2 Symm

The Symm routine performs a scalar matrix-matrix product, with one matrix operand being symmetric($A = A^T$), and adds the result to a scalar-matrix product. The operation is defined as:

$$C = \alpha BA + \beta C \tag{2.2}$$

Where:

- Matrix A is symmetric
- B and C are m-by-n matrices
- α (alpha) and β (beta) are scalars

2.2 Auto-tuning

Auto-tuning is know under many different names, but most of them operate under the same basic principals, as described in the ATLAS paper[7]. These three parts are:

- A method adapting software to different environments
- Robust, context-sensitive timers
- Appropriate search heuristics

A key part of the optimization of the program is estimating the optimal values for important parameters, such as block sizes and loop unroll factors. There are mainly two approaches for solving this problem; the first, and more traditional approach, uses analytical models to compute the optimal values. These are often built up by fixed code or algorithms with changeable parameters, which gives different versions of the same code. The other approach utilizes global search over the space of parameter values by generating solutions with many different combinations of parameter values. This will produce a vast number of different solutions. These are then run on the actual hardware to find which performs best.

It is widely believed that the model-driven optimization cannot compete with search-based empirical optimization, due to the lack of ability to catch all the complexities in a modern high-performance architectures. This statement was partially proven wrong by Yotov et al.[8]. In their study they replaced the general search engine in ATLAS with a model-driven engine, and got performance comparable to code generated by the standard ATLAS.

A way of timing the run-time of code is needed. This is to time the code as it is executed on the target-hardware, to find the best performing solution. To prevent the timings from being affected by different load, or other factors, on the target machine, it needs to be robust enough to produce correct timings on the different solutions. It is also important to remember that some OpenCL function-calls can be non-blocking; that means that they return control to the CPU thread before completing their work. This could potentially be a problem if using CPU timers.

The job of searching through all the different code variations, is performed by a search heuristic, and can be time-consuming with large data sets. This calls for a good search heuristic, which quickly can process the search tree and deliver the best performing solution.

2.3 ATLAS

Automatically Tuned Linear Algebra Software[9] offers an alternative solution for solving linear algebra, unlike the hardware-specific implementations. It can achieve near-optimal results compared to the BLAS functions, on different systems. This is done by benchmarking at install and auto-tuning. ATLAS first testes a large amount of implementations, then invoke a search on the different parameters of the given problem. The implementations also uses hand-written assembly code mainly to further improve performance, but also to achieve persistent performance despite compiler change.

An in depth analysis on how ATLAS is made is done by Yotov et al. in *Is search really necessary to generate high-performance BLAS?*[8]. A more detailed description of the whole implementation is given by the original author R. Clint Whaley in *ATLAS version 3.9: Overview and status* [10].

The project has grown from a small spare-time project, into a funded large project. As Whaley states in his invited paper [10]; *ATLAS is overdue for a new stable release. However, users can access some of the new features in the developer series*[2]. ATLAS has been the project from which much of the current knowledge about auto-tuning linear algebra software grew. However, it was not the first. The PHiPAC[11] project was the first to apply the auto-tune theories to linear algebra. The application and success of these different projects vary widely, but are built up on the same basic philosophy of applying empirical results and some degree of automatically tuning to customize the libraries for greater performance.

2.4 ViennaCL

The Vienna Computing Library (ViennaCL) is a scientific computing library written in C++ and based on OpenCL. It allows simple, high-level access to the vast computing resources available on parallel architectures such as GPUs and is primarily focused on common linear algebra operations (BLAS levels 1, 2 and 3) and the solution of large systems of equations by means of iterative methods with optional preconditioner[12].

ViennaCL themselves claim that their compute kernels are not fully optimized yet, and that further speedup will be possible in future releases.

2.5 CUDA

Compute Unified Device Architecture (CUDA) is NVIDIAs hardware and software architecture that allows for execution of non-graphical programs on NVIDIA GPUs. CUDA provides both a lowlevel and a high-level API to interface with the GPU, as well as the C for CUDA programming language to write kernels to be

[2]http://sourceforge.net/projects/math-atlas/files/

executed on a GPU[13]. A kernel written in C for CUDA is executed on a NVIDIA GPU by a set of threads. The threads are divided into groups, called thread blocks, and the thread blocks are organized in a grid. A thread block executes on a single streaming multiprocessor and a thread runs on a core. All threads within a grid execute the same kernel. When a thread block is scheduled for execution, its threads are divided into groups of 32 threads, called a warp, that execute the kernel concurrently.

The differences between the competing OpenCL and CUDA are rather small, the biggest being that CUDA is hardware-specific to NVIDIA GPUs. OpenCL offers greater possibilities for task-parallelism. Also, pointers in OpenCL kernels must be annotated with their memory space.

2.6 CUBLAS

CUBLAS[13] is an implementation of BLAS on top of the NVIDIA CUDA driver. The way this implementation works, is by creating matrix and vector objects in the GPU memory space, filling them with data, calling a sequence of CUBLAS functions and, finally, uploading the results from GPU memory space back to the host. CUBLAS has support for all of the 152 standard BLAS routines, with single, double or complex data type. The implementation delievers from 6x to 17x speedup[13] over MKL BLAS[3].

2.7 OpenCL

The OpenCL(Open Computing Language) is the first open, royalty-free standard for cross-platform programming. It is general purpose, parallel and available on all processors and accelerators on heterogeneous platforms. It was originally developed by Apple, but is today maintained by the Khronos Group[4]. OpenCL lets the programmer write a single portable program that uses all resources on the heterogeneous platform. The first proposal of OpenCL specifications came in June 2008, and already in December 2008 OpenCL 1.0 was released, then OpenCL 1.1 was publicly released in June 2010. The newest version (OpenCL 1.2) was announced on November 15th, during Supercomputing 2011. New features in OpenCL 1.2 included seamless sharing of media and surfaces with DirectX 9 and 11, enhanced image support, custom devices and kernels, device partitioning and separate compilation and linking of objects.

[3]http://software.intel.com/en-us/articles/intel-mkl/

[4]http://www.khronos.org/opencl/

OpenCL is built up of an API, a programming language and an architecture. NVIDIA and ATI/AMD delivers implementations of OpenCL, as-well as Apple and IBM. These runs on devices that either supports the CUDA architecture[14], or on a device that supports AMD Accelerated Parallel Processing (APP) Technology [15] (previously know as the ATI Stream Technology).

2.7.1 OpenCL Architecture

OpenCL operates with three different models for their architecture; the platform model, the execution model and the memory model.

Platform Model

OpenCL operates on the given problem through an abstract, hierarchical platform model. The host coordinates execution, sends and receives data to and from an array of Compute Devices. Each of these devices are made up of several Compute Units, which again consists of an array of Processing Elements, as shown in Figure 2.1.

This model does not specify exactly what kind of hardware constitutes a Compute Device, and thereby ensures that OpenCL may be run on a variety of different devices. These are typically GPUs, multicore CPUs, Digital Signal Processors or the Cell Broadband Engine.

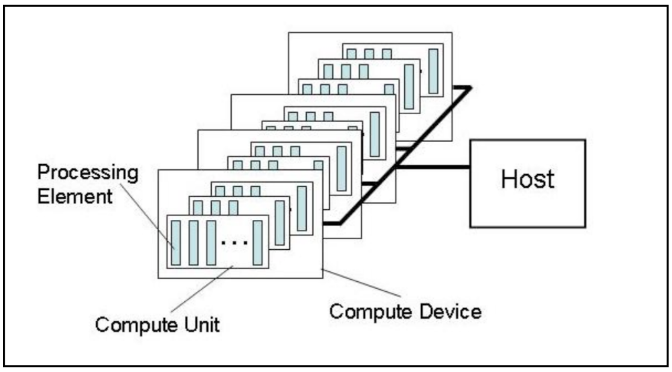


Figure 2.1: Diagram describing the OpenCL Platform Model[2].

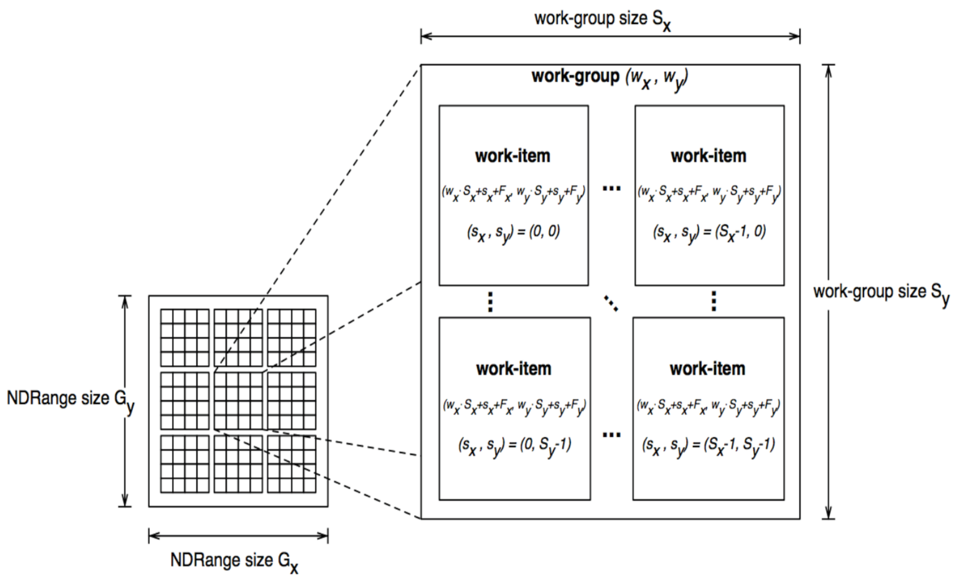
Execution Model

The OpenCL execution model is divided into two different part; one program that is executed on the host, and a number of *kernels* that is executed on one or more devices. These two parts are different, both in language and composition. More about the OpenCL programming model in section 2.7.3. However, the host part will not differ much from any other standard application.

In order to execute a kernel on the device, the host must establish a OpenCL *context*. The context is an object containing information relevant to the execution of the kernel, like *devices*, *memory* and *command queues*. A context may have multiple command queues and devices.

Data movement and OpenCL tasks between the host and device, is handled by *Command Queues*. The commands in the queue can either be executed out-of-order or in-order, which is up to the programmer to decide. The available commands are kernel execution, data transfer and synchronization. After the host submits a kernel for execution on the device, an index space called NDRange is defined.

For each point in the index space, one kernel is executed. These are called work-items, and each one is uniquely specified by an index. As can be seen by Figure 2.2, the work-items are grouped together with multiple other work-items to form a work-group. The two different groups have unique indexation. Each work-item within a work-group have a unique index, but also each work-group within the index space.



work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

Figure 2.2: NDRange from the OpenCL execution model[2].

Memory Model

The model consists of four different parts: Global memory, constant memory, local memory and private memory. Memory management in OpenCL is explicit, meaning that data has to be moved from host to global memory, then to local memory and back. OpenCL has a relaxed consistency memory model built up as shown in Figure 2.3. This means that different work-items may see a different view of the global memory as the program progresses. Within a work-item, reads and writes are consistently ordered. However, synchronization is required between work-items to ensure consistency. This relaxed memory model is an important part of OpenCL's effort to provide parallel scalability. Programs that relies on strong memory consistency for synchronization and communication usually fail to execute in parallel, because memory ordering requirements force a serialization and thereby hinders scalability.

The host can interact directly with global and constant memory, but not with the other two. The global memory of the device is typically large, but comes with a low bandwidth, and is often located on the off-chip DRAM. Global memory permits read/write access to all work-items on the device, within the same context. This allows work-items to read from or write to any element of a memory object.

Constant memory is located in the same region as global memory, and thus shares the same properties. This region contains objects allocated and initialized by the host. These do not change during kernel execution, and are thereby read-only.

Private memory is only accessible by each associated work-item, but all the work items in a work-group shares the local memory. Local and constant memory are typically located on-chip, and are therefor small, with large bandwidth and low latency. However, local memory can also be mapped to the global memory sections.

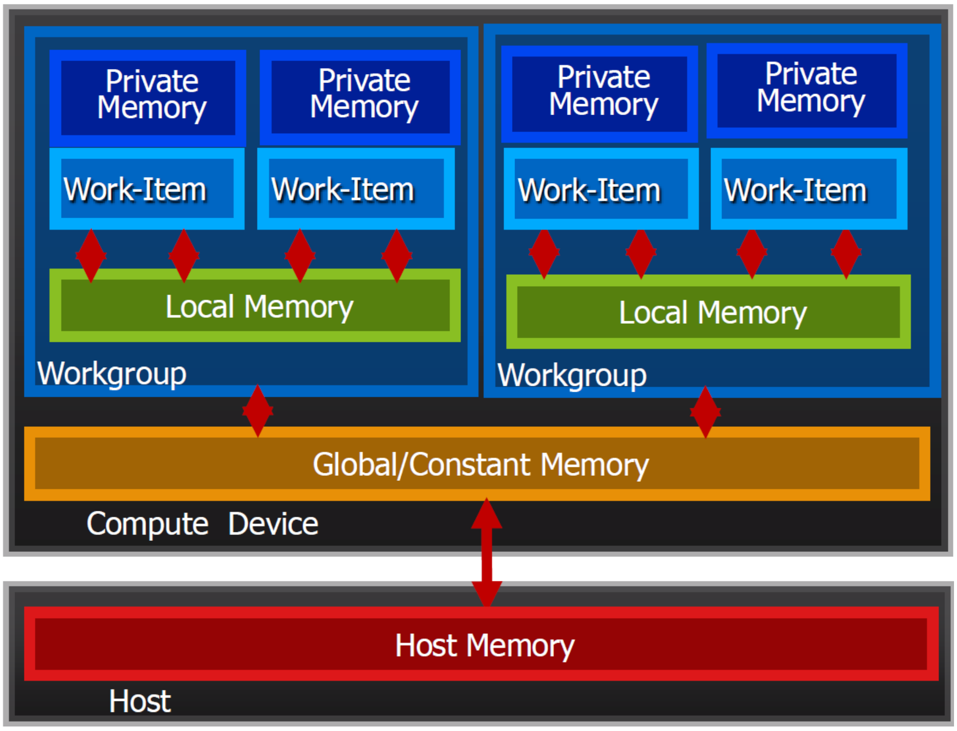


Figure 2.3: OpenCL memory model[2].

In order to manage the memory, the host uses *Memory objects*. These are categorized into two types: *buffer* objects and *image* objects. The elements in a buffer are stored in a sequential fashion, and stores a one-dimensional collection of objects. The image object is used to store a two- or three-dimensional texture, frame buffer or image. The data in the buffer object is stored with the same format as it accessed by the kernel. This does not ally to the image object, as the data format used to store the data within the object may vary from that used by the kernel.

2.7.2 OpenCL Programming Language

The OpenCL C programming language is based on ISO/IEC 9899:1999 C language specifications. It uses a subset of the language and adds extensions for parallelism. These extensions are: Vector types, work-items/work-groups, synchronization, address space qualifiers and a number of built in functions.

2.7.3 The Programming Model

Data parallel and task parallel are the two supported programming models in OpenCL. Hybrids of these two are also supported.

A data parallel model defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object. The index space generated when the kernel is scheduled for execution, defines the work-items and how the data maps onto the work-items.

In the task parallel programming model only one instance of a kernel is executed on a device, and parallelism is expressed using vector data types and queuing several tasks.

2.8 Compilation Model

OpenCL uses dynamic (run time) compilation model, like OpenGL and DirectX. This type of model consists of two steps. Step one: Compile the code to an Intermediate Representation (IR), which is usually an assembler or a virtual machine. This is also called offline compilation, and is performed by the front-end compiler. Step two: The IR is compiled to machine code for execution. This step is less time consuming that the first. It is known as online compilation, and is performed by the back-end compiler.

In dynamic compilation, step one is usually done once, and the IR is stored. The IR is then loaded by the program, and step two is performed during run time.

2.9 Performance

As previously stated, the ultimate goal of optimization it to receive better performance. This can not only be achieved by putting more funds into better hardware, but also by better understanding the problems you are trying to solve. When working with symmetric matrices, it is essential to exploit this for good performance. The symmetry implies that only one triangles of the matrix needs to be calculated. This is itself offers a speedup factor of 2.

2.9.1 Theoretical Issues

Branching

Branches are problematic in the sense that they may be unpredictable. So one should strive to remove the branches altogether. This springs from the basic principles of branch elimination; it is often more efficient to express a value as a simple function, opposed to selecting a result through a change in control flow. Less branches also makes the performance of the program more predictable. The compiler will avoid scheduling problems, the traces will be less complex and most variables will be write-once.

Memory transfer

Host-device data transfer should be kept to a minimum. This is due to the relative slow bandwidth of the PCIe bus, compared to global memory access. PCIe v2.0 bus has 8 GB/s of bandwidth, per lane. For comparison, the GTX480 has a bandwidth of 177,4 GB[14]. PCIe v3.0 will have a bandwidth of 16 GB/s, and the first GPU to make use of this standard will be the Radeon HD 7970. This GPU will operate with a bandwidth of 264 GB/s. This bandwidth gap will only continue to increase in the future.

When a transfer is necessary, one should try to overlap this with computation. Intermediate data can be allocated, operated and de-allocated on the GPU, further reducing transfer.

Reaching maximum bandwidth depends on three factors; concurrency of memory requests, data size in one stride and address alignment.

Non-Coalescing Memory Access

Accessing memory can, as previously stated, be an costly operation. Certain memory access patterns allow the data to be read and written in one operation. This requires the access to be coalesced: The programmer has to prevent strided and misaligned accesses. A way of achieving this, is to use local memory. Device memory allocated through OpenCL is guaranteed to be aligned to at least 256 bytes[16]. As a consequence of this, choosing thread block sizes as multiples of 16, allows memory accesses by half warps that are aligned to segments.

2.9.2 Optimizing Techniques

To achieve good performance it is vital to apply some sort of optimization techniques. Some are mentioned in the sections above. However, more techniques can be applied to achieve better performance.

Blocking

Blocking or loop tiling is applied to improve cache reuse.[17] This is done by sizing the data to fit inside the faster parts of memory. One can then perform more operations on the given data, before having to transfer more from slower parts of memory.

An example of how to perform this blocking is demonstrated in Algorithm 2.1 and Algorithm 2.2 [5]. Algorithm 2.2 demonstrates how the loop looks after applying blocking.

Algorithm 2.1 Original matrix multiplication

```
DO I = 1, M
DO K = 1, M
DO J = 1, M
Z(J, I) = Z(J, I) + X(K, I) * Y(J, K)
```

Algorithm 2.2 After blocking B*B

```
DO K2 = 1, M, B
DO J2 = 1, M, B
DO I = 1, M
DO K1 = K2, MIN(K2 + B - 1, M)
DO J1 = J2, MIN(J2 + B - 1, M)
Z(J1, I) = Z(J1, I) + X(K1, I) * Y(J1, K1)
```

[5] http://en.wikipedia.org/wiki/Loop_ tiling

Loop Unrolling

Loop unrolling can be applied to avoid unnecessary extra calculations and branches.

2.9.3 Describing Execution of Parallel Programs

When executing a single program across multiple processing elements, we can divide the run time into two components: The time used for communication and the time spent on computation. When operating on a shared memory system, the communication part mainly consists of synchronization overhead. How the total run time is distributed among the two will vary with the application, but generally it is related to the problem size and the number of processes used. The basic model for describing time usage of a parallel program is gives as:

$$T_{para} = T_{comp}(n, p) + T_{comm}(n, p). \tag{2.3}$$

where n is the problem size parameter, p is the number of processes used, T_{comp} is the time used in computation and T_{comm} is the time spent communicating.

The communication of an application usually consists of communication flow between different parts of the application, typically host-device or internode in device. The total communication time is therefore calculated as a sum of time spent in each flow,known as the Hockney model[18], as shown in equation 2.4:

$$T_{comm} = t_{startup} + \omega x t_{word}. \tag{2.4}$$

where $t_{startup}$ is the time spent sending a message of zero bytes; this includes any time spent manipulating the message. This time is usually called the latency of the system, and is assumed to be constant. t_{word} is the time required to send one data word. This value is also assumed to be constant, and is given as the inverse bandwidth of the communication channel. ω is the total number of data words.

2.9.4 Measure Performance

The quantification of results, is an important part of the process when working with any performance-critical application. Two common metrics used when comparing performance of different BLAS implementations, is FLOPS and speedup. Floating-point operations per second (FLOPS) indicate the performance of the application, by showing how many floating-point operations it can perform per second.

Amdahl's and Gustafson's law

There is one specific concept that can really limit the possible speedup possible, and that is called Amdahl's law. Most parallel programs also include a serial part, that has to be executed serially on one core. Speedup is defined as:

$$S(p) = \frac{t_s}{t_p}. \tag{2.5}$$

Where t_s and t_p is the serial and parallel computation time, respectively. If we generalize equation 2.5, and denote p as the number of processors, the maximum speedup of a parallelization be $t_p = \frac{t_s}{p}$ and thus $S(p) = \frac{t_s}{\frac{t_s}{p}} = p$. t_p will in most cases involve serial parts, as mentioned above. We let f denote the serial fraction of the computation. This gives:

$$S(p) = \frac{t_s}{ft_s + (1-f)t_p} = \frac{t_s}{ft_s + \frac{(1-f)t_s}{p}}. \tag{2.6}$$

We are interested in finding the maximum speedup when the computation includes a serial part, so we let p grow and find that:

$$S_{max} = \lim_{p\to\infty} \frac{t_s}{ft_s + \frac{(1-f)t_s}{p}} = \frac{t_s}{ft_s} = \frac{1}{f}. \tag{2.7}$$

What this tells us is that the speedup is strictly limited by the serial fraction. So for example, an application with a serial fraction of 10%, the maximum speedup would be $\frac{1}{0.1} = 10$, no matter how many processors you put at the job. This is a quite grim prospect, when working with parallelism. However, one prerequisite for Amdahl's law to apply is a fixed problem size N; also known as *strong scaling*.

So if one changes the problem size as one changes the number of processing elements, and assumes that the serial fraction is not dependent on N, the spent by the serial part will have less impact on overall speedup. This is based on that as more compute elements are used, one is able to compute larger problem sets in a given time period compared to the serial one. This concept is called Gustafson's law[19], or *weak scaling*, and is more optimistic than Amdahl's law.

2.10 AMD Architecture

The first cards of the Radeon HD 5800 series were launched September 23, 2009, and the series was concluded with the release of HD5500 and 5400 in February 2010 [20]. Performance is differentiated between the GPUs by the number of SIMD arrays each GPU has, the core clockspeed, the memory bus width and the number of texture units and Render Output Units(ROP).

A GPU consists of several compute units (also called SIMD Engines) and each compute unit comprise 16 stream cores, which consists of five processing elements, depending of the GPU model; as some of the low end GPUs only have four. See Figure 2.4 for a diagram of the GPU architecture and Figure 2.5 for a diagram of a stream core. All the stream cores within a compute unit will perform the same instruction in a lock-step fashion, at each cycle. A VLIW[6] is utilized to issue the instructions to the processing elements.

All of the processing elements can perform single-precision floating point operations and the fifth processing element in a stream core can also execute transcendental operations. To perform double-precision operations, two or four of the non- transcendental operations capable processing elements are combined.

Every compute unit have 32 kB of local, on-chip memory called local data share (LDS) and a 8 kB L1 cache. L2 cache is shared by several compute units. The local data share is divided into 32 memory banks, that are four bytes wide and 256 bytes deep[3]. One memory operation can be performed for each bank each cycle, but if more than one operation maps to the same memory bank, a bank conflict occurs and the operations are serialized.

A compute unit also have 256 kB of available registers. The register space comprise 16384 general purpose registers, where one register contains four 32-bit values.

[6]Very Long Instruction Word

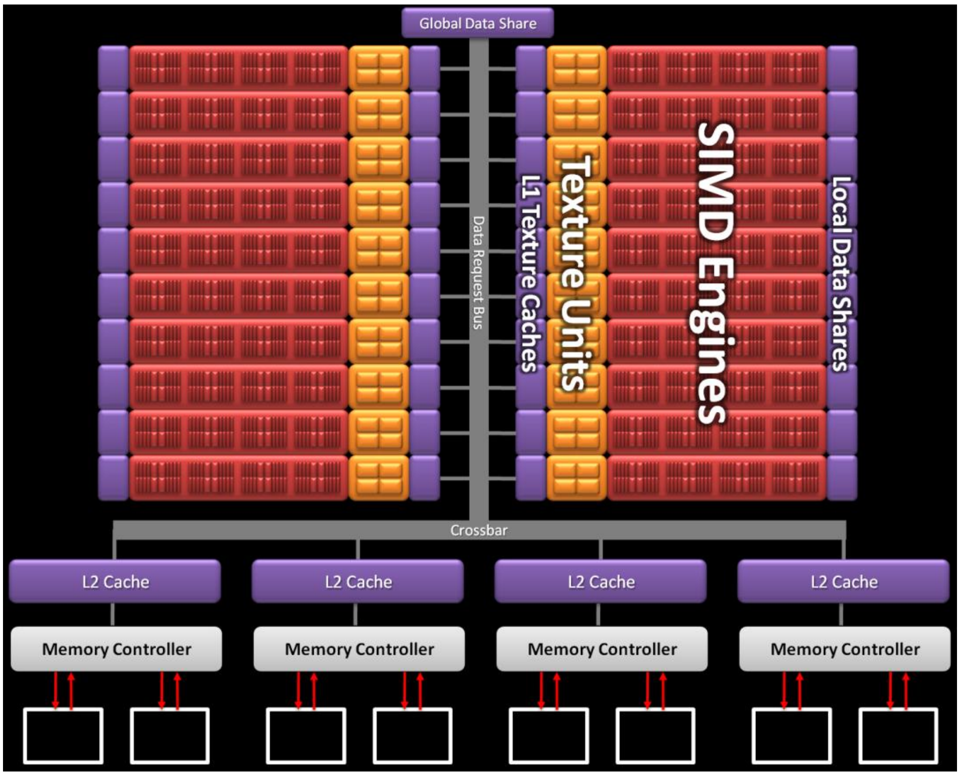


Figure 2.4: Radeon HD5870 architecture. [3]

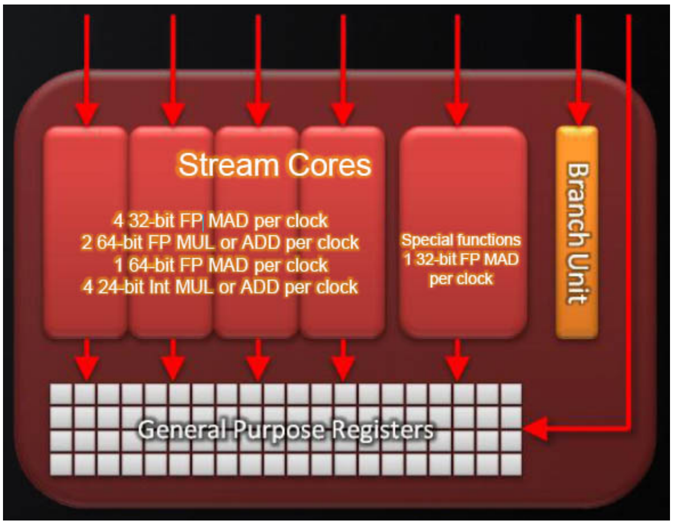

Stream Cores
4 32-bit FP MAD per clock
2 64-bit FP MUL or ADD per clock
1 64-bit FP MAD per clock
4 24-bit Int MUL or ADD per clock
Special functions
1 32-bit FP MAD
per clock
Branch Unit
General Purpose Registers

Figure 2.5: Radeon HD5870 thread processor.[3]

2.10.1 OpenCL running on AMD GPU

When work-items are executed on a GPU, they are grouped together in wavefronts. A wavefront consists of 64 work-items, that are executed in lockstep on a compute unit. Every work-group is divided into an integer number of wavefronts and to achieve optimal performance, the number of work-items within a work-group should be divisible with the wavefront size[15].

As a kernel is being executed, a work-group is assigned to a single compute unit and a work-item runs on a stream-core. Four work-items from the wavefront being executed are pipelined on one stream core to hide memory latencies. At each cycle, 16 of the work-items in a wavefront execute one instruction. When a wavefront is looked at as a whole, this give the appearance that one instruction is executed every four cycles. If the execution paths of work-items within a wavefront diverges, their execution are serialized.

The use of private memory in kernels will map the general purpose registers, see Figure 2.5, as long as the capacity allows. If more memory is required, the compiler will solve this by generating spill code, and move remaining blocks over to general memory.

2.11 NVIDIA Architecture

The first GPUs of NVIDIAs Fermi architecture were released in April of 2010, with the rest following throughout the spring and fall [4]. The GTX570 was released in December the same year.

A GPU consists of several graphics processing clusters (GPC). Each graphics processing cluster is made up of four streaming multiprocessors, which comprise 32 cores[4]. See Figure 2.6. In addition to the 32 cores, each streaming multiprocessor also contains four special function units that can perform transcendental functions and 16 load and store units enabling a streaming multiprocessor to calculate 16 source and destination memory addresses per clock cycle. Each core contains a fully pipelined integer arithmetic logic unit and a floating point unit, as seen on figure 2.6.

All streaming multiprocessors have 32768 32-bit registers and 64kB of onchip memory[4]. The on-chip memory can be configured as 16kB of shared memory and 48kB of L1 cache or as 48kB of shared memory and 16kB of L1 cache. Additionally, all the streaming multiprocessors share 768kB of L2 cache, as seen on Figure 2.7.

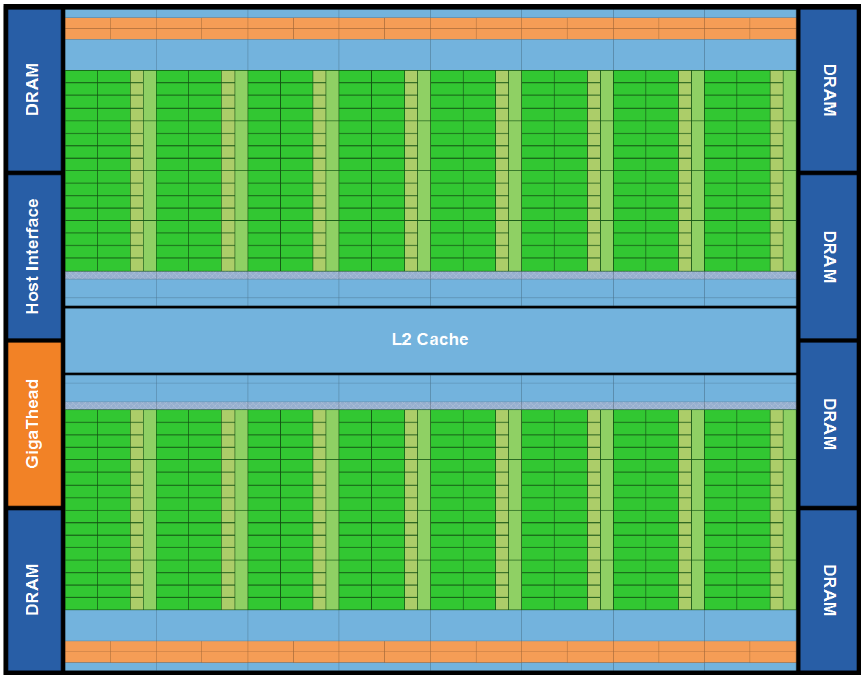


Figure 2.6: Fermi architecture.[4]

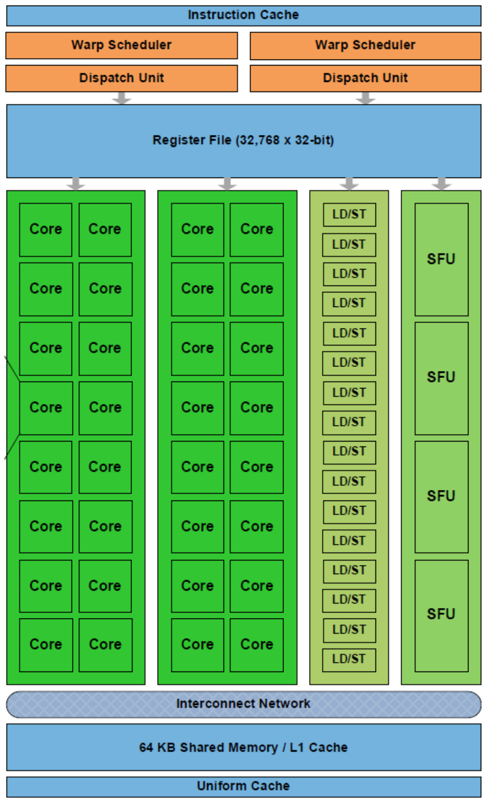


Figure 2.7: Fermi streaming multiprocessor.[4]

In both the AMD and the NVIDIA architecture, the intermediate language that programs the device is a lanewide SIMD model such that an instruction stream represents a single lane of the SIMD unit. This means that each SIMD unit is visible, rather than vectorwide instructions as with x86 SSE[7] and AVX[8]. Program counters are managed per SIMD vector such that a hardware thread is really a wide vector. The single-lane programs are grouped together using some degree of hardware vectorization, which allows efficient management of divergent branches, and stacks of masks to support predicated execution. To enable instruction-level parallelism, the AMD architecture use VLIW, as mentioned above. This is done by applying these instruction streams in each lane extracted by the low-level shader compiler from the intermediate language[21].

AMD and NVIDIA solves the instruction-level parallelism in different manners. AMDs approach is described over, while NVIDIA design achieves instruction-

[7]Streaming SIMD Extensions
[8]Advanced Vector Extensions

level parallelism by co-issuing two threads at once over two execution pipelines. Both designs are superscalar in that execution resources can issue memory access, arithmetic, and other operations from threads running on the same core, if not necessarily from the same thread.

2.11.1 OpenCL running on NVIDIA GPU

Given that the CUDA and OpenCL architecture are so similar, the way OpenCL executes on NVIDIA GPU is rather similar as on a AMD GPU.

When work-items are executed on a GPU, they are, as threads are in CUDA, divided into warps. Each work-group contains an integer number of warps. A work-group runs on a streaming multiprocessor and a workitem runs on a single core. All work-items within a warp execute the same instruction in lockstep. If the code path of work-items within a warp diverges, they are serialized. The number of work-items within a work-group should be divisible by the warp size for optimal performance.

OpenCL private memory maps to registers on the GPU. If a work-item needs more registers than is available, spill code is generated by the compiler and the extra memory needed is placed in a region of global memory. Local memory in OpenCL maps to shared memory on the GPU. Avoiding bank conflicts is the key to achieve optimal performance when using local memory.

When work-items within a warp access global memory, the memory access is coalesced into as few memory transactions as possible. How many transactions that is issued is dependent on the size of the elements accessed and the memory access pattern of the work-items.

Chapter 3

Implementation

This section will describe our implementation of the program. We start by looking at how the framework is set up, since this is the baseline of the implementation. Then, we present our OpenCL Generator, which is responsible for generating the kernels that are executed on the device. Finally, we present how we find the best performing kernel using search.

3.1 Framework

Our framework is written in C++, and is made up of five classes, plus the main class and some help functions to assist with debugging. The use of C++, instead of C, enabled us to structure the framework like an abstraction of a OpenCL program. The classes are used to set up all the steps of initiation, configuration and execution. See Figure 3.1 for a class diagram.

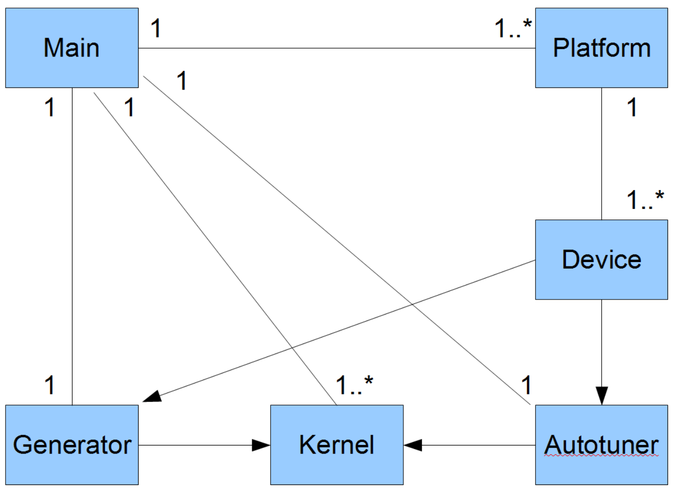


Figure 3.1: Class diagram of the program.

The Platform class is used to query the system for information required by the OpenCL Platform layer, and store this for the later stages of the process. The main purpose for this class, is to allow the user to decide on what platform to use, and in the next step; what devices is associated with the given platform.

The Device class is, as mentioned over, an abstraction for the device layer in OpenCL. This class contains functions to retrieve and initialize devices, and contains a list of all the available devices on the given platform. This class is used during the generation and auto-tuning stage, to establish what device is to be used. The class was also utilized during debugging, to output different information about the device in question.

The Kernel class represented the different kernels that was executed on the devices. It contains the different variables that are needed when en-queuing a kernel for execution. The block sizes and different values for global- and local work sizes are stored in each object. The class itself allows for kernels to be created, for arguments to be set and for the actual en-queuing. The different kernel objects will also contain timing information about itself, for use in the later stages of the program.

The actual generation of the kernels, and all the functions related to it, is done in the Generator class. This class contains all the different kernel versions, described in Section 3.2. The main method took in the chosen device and auto-tuner, and outputs a list of different kernels generated by different functions within the Generator class. The kernels are generated as a string, then converted to a *cl_ program* which is used to create the different kernel objects.

The Autotuner class has a number of tasks. First, it is responsible for creating the context and the command queue, which are needed to run OpenCL. It will also allocate memory for the different matrices and fill them with random, or seeded[1] content. These matrices are first allocated on host memory, then written to device as buffers. The Autotuner also contains the search function for finding the fastest kernel version.

All the classes and functions utilized the Error class, which helped debugging or catching errors which occurred during compilation or runtime.

The framework applied the following sequence when executed:

1. Initiate needed objects from the different classes.
2. Initiate host memory, and run random number generator.
3. Query system for platforms.
4. Query system for devices.
5. Create context and command queue.
6. Create device buffers, and write host data to device buffers.
7. Use chosen device as input to the generator. This creates and compiles the program; generates kernels, sets kernel arguments.
8. En-queues kernels for execution, records computation time and returns C matrix to host.
9. List of kernels is run through the search function, and outputs the best performing kernel.
10. Clean up, and exit application.

3.2 OpenCL Generator

The Generator is the core class of the framework. This is where the source code for the kernels get generated. The different kernels are based on different models to conduct the matrix multiplication.

[1] A seed is utilized to initialize a pseudo-random number generator

3.2.1 No Memory Blocking

The simplest of the five models. Parameters that are required to generate the different versions are the number of work-items per work-group and the block size of C. The number of work-items in each work-group had to be divisible with the size of the C matrix block.

Depending on the work-item distribution, each work-item calculated one or more elements of the C matrix. Given this, each work-item will take at least one row from matrix A and column from matrix B. Pseudo code for this version is given in Algorithm 3.1, and a illustration can be found in Figure 3.2.

Algorithm 3.1 Pseudo code for No Memory Blocking version.

```
Get index of elements to process
for Elements of C do
    $C = \alpha * A * B + \beta * C$
end for
```

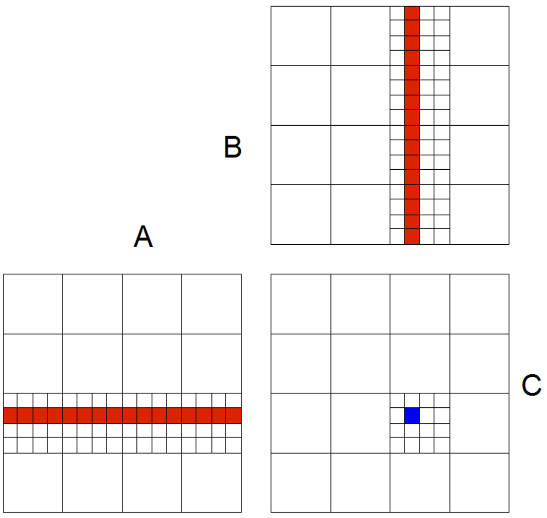


Figure 3.2: Illustration of the No Memory Block version. This is a shrunk version, however the principle is the same on a large scale. As seen from the figure, a sub-row from A and a sub-column from B are read by the work-item computing the blue element of C.

3.2.2 Both Matrices in Local Memory

Parameters that are required to generate the different versions are the number of work-items per work-group and the size of the blocking.

Depending on the work-item distribution, each work-item calculated one or more elements of the C matrix. Given this, each work-item will take at least one row from matrix A and column from matrix B, which are moved from global memory to local memory. Each work-item will then use local memory to calculate the C element. Pseudo code for this version is given in Algorithm 3.2, and a illustration can be found in Figure 3.3.

Algorithm 3.2 Pseudo code for Both Matrices in Local Memory version.

```
Get index of elements to process
Allocate local sub-matrices
Declare start, end and step values
for Elements in block do
    Copy sub-matrices to local memory
    for Elements of C do
        $subC+ = localA * localB$
    end for
    $C = \alpha * subC + \beta * C$
end for
```

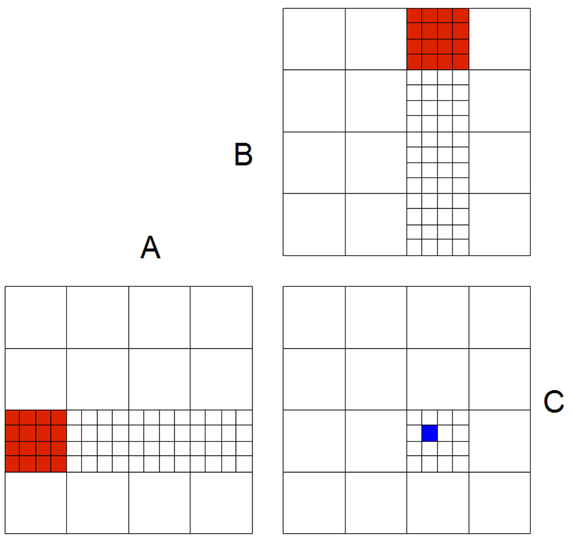


Figure 3.3: Illustration of the Both Matrices in Local Memory version. In this version a block of A and B are read into local memory. Each work-item then reads one row from A and a column of B into private memory, and computes the given element of C. This is a shrunk version, however the principle is the same on a large scale.

3.2.3 A in Local Memory B in Private Memory

Parameters that are required to generate the different versions are the number of work-items per work-group and the size of the blocking.

Depending on the work-item distribution, each work-item calculated one or more rows of the C matrix. Given this, each work-item will take one block from matrix A and move this from global memory to local memory. A column from matrix B will be moved to private memory. Each work-item will then calculate the C element, with the private A and the local B. Pseudo code for this version is given in Algorithm 3.3, and a illustration can be found in Figure 3.4.

Algorithm 3.3 Pseudo code for A in Local Memory B in Private Memory version.

```
Get index of elements to process
Allocate local sub-matrix for A
Allocate private memory for B
for Elements in block do
    Copy sub-matrices to local memory
    for Elements of C do
        $subC+ = localA * privateB$
    end for
    $C = \alpha * subC + \beta * C$
end for
```

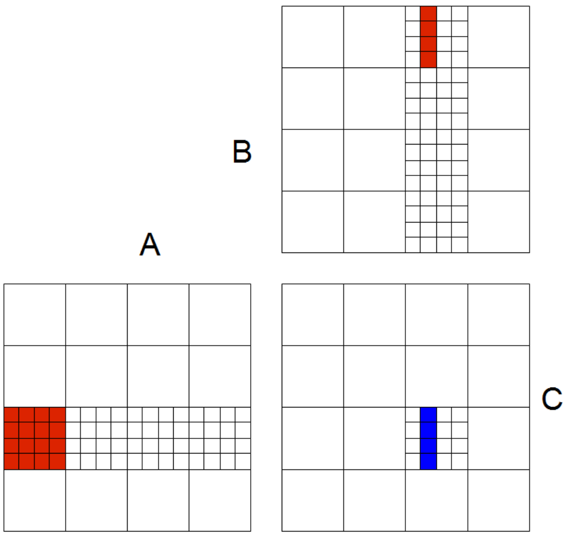


Figure 3.4: Illustration of the A in Local Memory B in Private Memory version. This is a shrunk version, however the principle is the same on a large scale.

3.2.4 Both Matrices in Private Memory

Parameters that are required to generate the different versions are the number of work-items per work-group and the size of the blocking.

Depending on the work-item distribution, each work-item calculated one or more elements of the C matrix. Given this, each work-item will take one row from matrix A and B and move this from global memory to local memory. Each work-item will then calculate the C element, with the two private matrices. Pseudo code for this version is given in Algorithm 3.4, and a illustration can be found in Figure 3.5.

Algorithm 3.4 Pseudo code for Both Matrices in Private Memory version.

```
Get index of elements to process
Allocate private memory for matrices A and B
for Elements in block do
    Copy A and B sub-matrices to private memory
    for Block size do
        $subC+ = privateA * privateB$
    end for
    $C = \alpha * subC + \beta * C$
end for
```

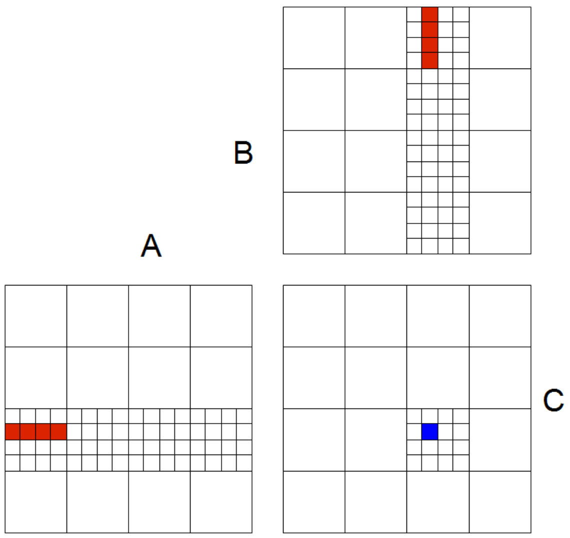


Figure 3.5: Illustration of the Both Matrices in Private Memory version. This is a shrunk version, however the principle is the same on a large scale. In this version, a row from matrix A and a column from matrix B is read into private memory. The work-items then compute the given element of C.

3.2.5 Both Matrices in Local and Private Memory

Parameters that are required to generate the different versions are the number of work-items per work-group and the size of the blocking for local and private memory.

Depending on the work-item distribution, each work-item calculated one or more elements of the C matrix. Given this, each work-item will take one block from matrix A and B and move this from global memory to local memory. Then a row from matrix A and column from matrix B will be read into private memory. Each work-item will then calculate the C element, with the two private matrices. Pseudo code for this version is given in Algorithm 3.5, and a illustration can be found in Figure 3.6.

Algorithm 3.5 Pseudo code for Both Matrices in Local and Private Memory version.

```
Get index of elements to process
Allocate local memory for matrices A and B
Allocate private memory for matrices A and B
for Number of sub-matrices do
    Copy A and B sub-matrices to local memory
    for Block size do
        Read block of matrices A and B to private memory
        $subC += privateA * privateB$
    end for
    $C = \alpha * subC + \beta * C$
end for
```

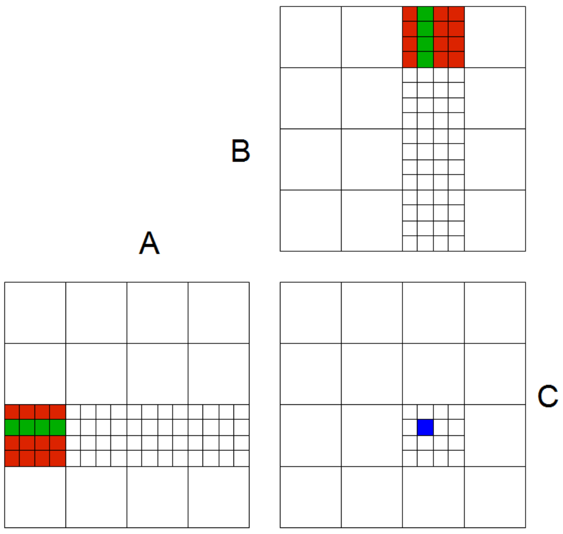


Figure 3.6: Illustration of the Both Matrices in Local and Private Memory version. This is a shrunk version, however the principle is the same on a large scale. First a block of matrices A and B is read into local memory, as shown by the red. Then a row from A and a column from B is read into private memory, as indicated by green. This then used by a work-item to compute given element of C.

3.3 Search

In order to reduce the amount of generated kernels, the first step was to go for a model based auto-tuner. This drastically reduces the search space by only tuning a set of key parameters, as described in Section 2.2, and also ensures that only valid kernels are generated. This allows for a less complex search algorithm to be applied. With a basic linear search, we found that the best performing kernel could be found in reasonable time.

After all the kernel source code had been generated, the different kernel functions created OpenCL programs which were passed to the Kernel class. Different arguments are set on the returning kernel objects, depending on different parameters.

All the kernels are then executed on the device a given number of times, to time the compute time. The use of multiple runs is to ensure that the different kernels got a fair timing. The median of the different runs is the deciding factor when the search compares the different kernels.

OpenCL comes with built-in functions to measure execution time, and these were used to retrieve profiling information from the different kernels. The use of this information ensures that the timings are context safe and reliable.

Pseudo code for the search function can be found in Algorithm 3.6.

Algorithm 3.6 Search algorithm pseudo code.

```
Create buffers for matrices A, B and C
Copy generated data to buffers
for all Kernel objects do
    Create and compile OpenCL program
    Set kernel arguments
    for $i = 1 \to 5$ do
        Execute kernels on device
        Store kernel timing
    end for
    Calculate median time
    if currentBestTime > kernelTime then
        Set new best kernel time
    end if
end for
Return best kernel
```

Chapter 4

Results

This section will describe the results from our tests on different GPUs. First we will describe our test environment, and how we measure performance. Finally we will present the results from running the kernels with various matrix sizes.

4.1 Test Environment Methodology

Our program has been run on three different GPUs: AMD HD5870, NVIDIA GTX570 and NVIDIA C2050. The systems these GPUs operate in are described in Table 4.1, Table 4.2 and Table 4.3. A more detailed desciption of the three GPUs used for benchmarking our implementation can be found in Table 4.4. Most of the development was done on System 1, Table 4.1.

The first step of the benchmarking, is to set desired values for N and K (matrix dimensions), and decide what blocking size to apply. Then run the application, which generates the different kernels with the different parameters. The generator calculates the global work size to apply, according to the given parameters. The kernels are run five, or more, times on the device to acquire fair timings. After all kernels are processed, the search algorithms quickly determines what kernel has performed best. The metric applied to determine the best is the average of all the runs. Information about the kernel, total and average run-time is reported back to the user. The use of mean arithmetic can be discussed to be a bad choice due to the effect single outliers can have on the mean value. If this a recurring problem, a median arithmetic might give better results. A function to write the kernel to file is also present.

In order to retrieve reliable benchmarks to test our application against, two different applications was run on the same systems we ran benchmarks on. On both NVIDIA and AMD GPU architectures, the ViennaCL[12] library was applied. On test environments running with AMD GPUs, the AMD Accelerated Parallel Processing Math Libraries[22] (APPML) was used. While NVIDIA test environments ran the NVIDIA GPU Computing SDK[13], which includes the cuBLAS libraries. Excluding Test environment 3, which did not support *compute_30*. The different versions can be seen in Table 4.5. The benchmarking was done on six different matrix sizes, to show how the performance span from the smallest, N=512, to the largest, N=3072. The data generated for the matrices are floats.

Test Environment 1	
Processor	AMD Phenom II X4 965 3,4 GHz
Memory	4GB
GPU	AMD Radeon HD 5870
Driver version	8.01.01.1215
Operating System	Windows 7 (64bit)

Table 4.1: Table describing the test environment with

Test Environment 2	
Processor	Intel Core i7
Memory	4GB
GPU	NVIDIA GTX570
Driver version	301.32
Operating System	Windows 7 (64bit)

Table 4.2: Table describing the test environment with

Test Environment 3	
Processor	Intel Core i7
Memory	6GB
GPU	NVIDIA C2050
Driver version	295.41
Operating System	Ubuntu 11.10 (64bit)

Table 4.3: Table describing the test environment with

GPU	AMD HD5870	NVIDIA GTX570	Tesla C2050
Shared Memory	32 KB	48 KB	64 KB
Memory	1024 MB	1280 MB	3072 MB
Memory bandwidth	152 GB/s	133.9 GB/s	144 GB/s
Number Cores	320	512	448
Clock frequency	850 MHz	732 MHz	1.1 GHz

Table 4.4: Specifications of the GPU types used

Implementation	Version
Auto tunable GPU BLAS	1.0
NVIDIA cuBLAS	4.2.9
AMD APPML clBLAS	1.6.180
ViennaCL	1.2.1

Table 4.5: Benchmarked implementations

4.2 Performance Measurements

Our results with the AMD Radeon HD5870 is shown in Figure 4.1, the NVIDIA GTX570 in Figure 4.2 and the results from the NVIDIA Tesla C2050 is shown in Figure 4.3.

When looking at the results from the AMD Radeon HD5870, one can see that our implementation is evenly matched on the 512 sized matrix, but performs below on the other matrix sizes. The performance does not vary much, unlike AMDs BLAS implementation. The AMD BLAS implementation performs around 250 GFLOPS at 512 matrix size, but increases to between 500 and 600 GFLOPS on the other matrix sizes. It greatly outperforms both ViennaCL and our implementation.

The tests on the NVIDIA GTX570 revealed the same tendencies as on the AMD GPU. The factory implementation greatly outperforms ViennaCL and our implementation. Our implementation performs under the ViennaCL on the smaller matrices, but is equal at 2560 and 3072.

The tests on the NVIDIA Tesla C2050 does not feature the vendor-tuned BLAS library, due to some missing features on the GPU in question. Our implementation once again performs under the ViennaCL, but does not alternate much in performance.

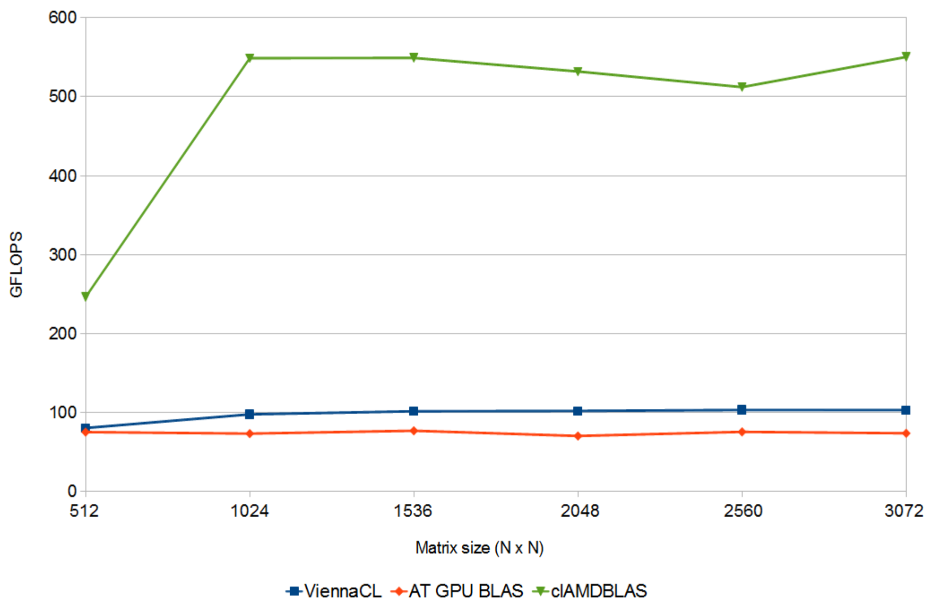


Figure 4.1: Graph describing the performance of the different implementations running on the AMD Radeon HD5870

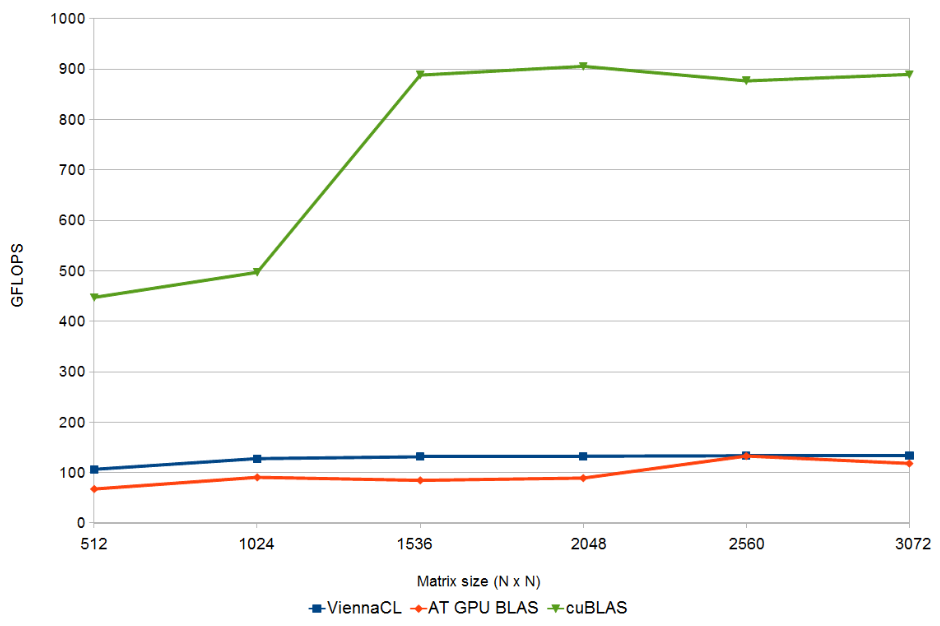


Figure 4.2: Graph describing the performance of the different implementations running on the NVIDIA GTX570

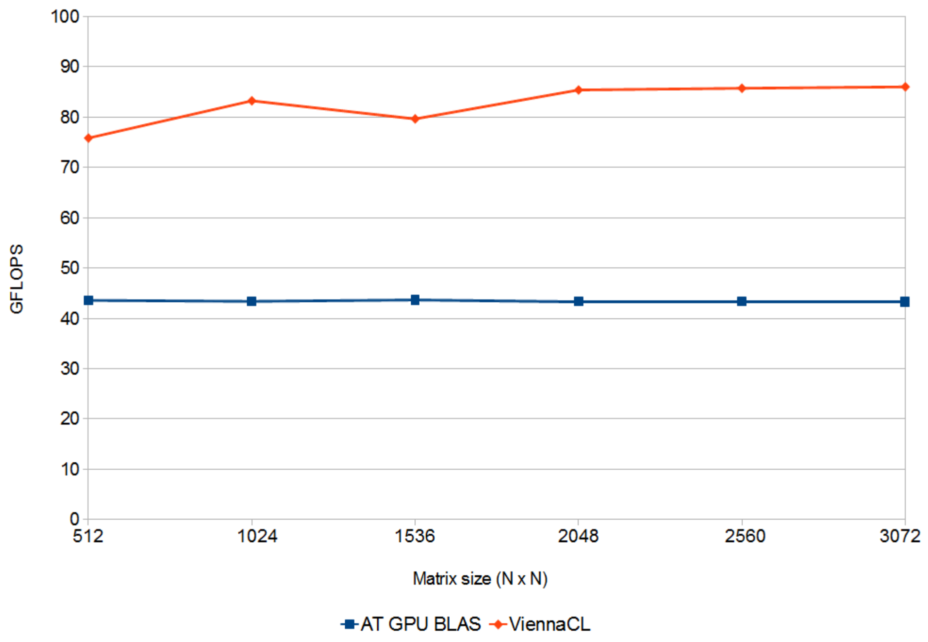


Figure 4.3: Graph describing the performance of the different implementations running on the NVIDIA Tesla C2050

Chapter 5

Conclusion and Future Work

This thesis has come up with a auto-tuning framework that generates OpenCL code, that runs on both AMD and NVIDIA GPUs. The OpenCL code can also be run on Intel and AMD CPUs, but this is counterproductive due to the loss in performance.

We chose three different GPUs for our benchmarking, two from NVIDIA and one from AMD. Only two of the kernel models turned out to perform best on the three different platforms. Interestingly, the two NVIDIA platforms came out with two different models. The AMD Radeon HD5870 and NVIDIA GTX570 both came up with the version with both blocks in local memory, described in Algorithm 3.2. The NVIDIA Tesla C2050 used the both blocks in local and private memory as its best performing kernel, described in Algorithm 3.5. This might be due to the fact that the Tesla cards are designed to be used for parallel computing, and makes better use of the private memory due to larger memory banks and faster clock speed, seen in Table 4.4.

We had two main issues during our development and testing. The first was the management of the memory when partitioning the blocks on the GPU. However, as mentioned in Section 2.2, this is one of the reasons that a auto-tunable framework is desired. It will help the programmer save time by eliminating the vast search-space for different options of blocking, global and local work sizes. The second was 32/64 bit problems. When testing our and the other benchmarking implementations on various platforms, we encountered a certain amount of problems when trying to run or compile the project. On one of the original test platforms, this got so bad and time-consuming that the platform had to be dropped altogether. This is partly because some of the older cards, and drivers, does not support some of the newer features in the different libraries. This is show by our inability to run the cuBLAS library on Test Environment 3, as described in Section 4.2. Here the GPU (Tesla C2050) lacked the compute_30 capabilities, and to rewrite the NVIDIA implementation of the BLAS library was deemed too complex.

In the end our implementation did not outperform the implementation written by the Vienna University of Technology. However, we did perform equally at some matrix sizes. ViennaCL is also a larger, and internationally recognized project[1]. The results also show that the vendor-specific implementations of the BLAS library vastly outperforms the two other implementations. This is not surprising, with the knowledge the corporations have of their own GPUs, and tune for specific hardware.

The performance in term of NVIDIA versus AMD is inconclusive. The best performing GPU was the NVIDIA GTX570, but this is also a newer card. However, that a card is newer does not always guarantee that it will perform better, as was the case with GTX680. This performed worse than the previous generations, due to architectural changes to get better performance per watt, at the cost of computational capabilities. Our and ViennaCLs performance on the Tesla C2050 card was lower than on the other two GPUs.

Our goal of making a auto-tuning framework that generates kernels that are executed on GPUs, for performing matrix-multiplication, is by us considered to be successful. We achieve comparable performance as the more profiled ViennaCL library. Although the native vendor supplied libraries performed better, this is expected.

5.1 Future Work

To have a framework that can have uses outside being a thesis project, it is important to add for than just a few BLAS routines. All the major BLAS libraries have support for all or at least most of the BLAS routines[23]. It is also important to further tune the kernels, to achieve performance closer to that of the vendor implementations.

At the moment, the generator only accepts matrices that are multiples of the blocking sizes used. To enable use of other sizes, creation of some clean-up code to handle the results is required. Or a less pretty version with zeroes to pad the matrices.

It will also be helpful, and important, to add functionality for transposing matrices. This will again increase the credibility for the overall implementation.

[1]http://gpgpu.org/2012/01/02/viennacl-1-2-0-released

With the new Ivy Bridge[2] released, it is warranted to test out the APU platform with our framework. Benchmarks performed by different externals suggest an increase in integrated GPU performance of 25% to 68%[3] compared to the previous generations.

[2]http://en.wikipedia.org/wiki/Ivy_Bridge_(microarchitecture)
[3]http://www.anandtech.com/show/5626/ivy-bridge-preview-core-i7-3770k/

Bibliography

[1] J. E. Steinsland, *Auto-tunable GPU BLAS*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, June 2011.

[2] Khronos Group, `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`, *The OpenCL Spesification*, 2011. Downloaded 24.10.2011.

[3] ATI(AMD), `http://developer.amd.com/gpu_assets/Heterogeneous_Computing_OpenCL_and_the_ATI_Radeon_HD_5870_Architecture_201003.pdf`, *ATI Radeon HD5870 Architecture*, 2010. Downloaded 14.04.2012.

[4] Nvidia, `http://www.nvidia.co.uk/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, *Fermi GPU family*, 2010. Downloaded 25.04.2012.

[5] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, "An extended set of fortran basic linear algebra subprograms," *ACM Transactions on Mathematical Software*, vol. 14(1), pp. 1–17, March 1988.

[6] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software*, vol. 16(1), pp. 1–17, March 1990.

[7] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the atlas project," *Parallel Computing 27*, vol. 1(2), pp. 3–35, 2000.

[8] K. Yotov, X. Li, G. Ren, M. J. Garzaran, D. Padua, K. Pingali, and P. Stodghill, "Is search really necessary to generate high-performance blas," *Proceedings of the IEEE*, vol. 93(2), pp. 358–386, 2005.

[9] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," *In Proceedings of 1998 ACM/IEEE Conference on Supercomputing*, pp. 1–27, 1998.

[10] R. C. Whaley, "Atlas version 3.9: Overview and status," *Software Automatic Tuning: From Concepts to State-of-the-Art Results, New York*, pp. 19–32, 2010.

[11] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology," *ICS '97 Proceedings of the 11th international conference on Supercomputing*, 1998.

[12] Vienna University of Technology, `http://viennacl.sourceforge.net`, *ViennaCL*, 2012. Downloaded 13.05.2012.

[13] NVIDIA Corporation, `http://developer.nvidia.com/cublas`, *CUBLAS*, 2011. Downloaded 24.10.2011.

[14] NVIDIA Corporation, `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/OpenCL_Programming_Guide.pdf`, *OpenCL Programming Guide for the CUDA Architecture*, 2010. Downloaded 17.10.2011.

[15] AMD Inc., `http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf`, *AMD Accelerated Parallel Processing OpenCL Programming Guide*, 2011. Downloaded 17.10.2011.

[16] NVIDIA Corporation, `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/OpenCL_Best_Practices_Guide.pdf`, *OpenCL Best Practices Guide*, 2010. Downloaded 31.01.2012.

[17] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 63–74, 1991.

[18] R. W. Hockney, "The communication challenge for mpp: Intel paragon and meiko cs-2," *Parallel Computing 20*, vol. 20(3), pp. 389–398, 1994.

[19] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31(5), pp. 532–533, 1988.

[20] Wikipedia, `http://en.wikipedia.org/wiki/Evergreen_(GPU_family)`, *Evergreen GPU family*, 2010. Downloaded 24.04.2012.

[21] B. R. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 2012.

[22] AMD, `http://developer.amd.com/libraries/appmathlibs/pages/default.aspx`, *Accelerated Parallel Processing Math Libraries*, 2012. Downloaded 13.05.2012.

[23] Netlib, `http://www.netlib.org/blas/`, *BLAS (Basic Linear Algebra Subprograms)*, 1990. Downloaded 23.05.2012.

Appendix A

Appendix

This appendix will present the kernels that was chosen as the best performing on the different test environments. For both the NVIDIA GTX570 and AMD Radeon HD 5870 the same model was chosen, as seen in A.1. While the NVIDIA Tesla C2050 chose a different version, seen in A.2.

```
#define BLOCK_SIZE 8
#define AS(i, j) As[i + j * BLOCK_SIZE]
#define BS(i, j) Bs[i + j * BLOCK_SIZE]

__kernel __attribute__((reqd_work_group_size(8, 8, 1)))
__kernel void BothLocalMem(__global float *a, __global
    float *b, __global float *c,
  int m, int n, int k, float alpha, float beta){

    unsigned int width = get_global_size(0);

  //Get index of element to be processed
    int gidx = get_group_id(0);
    int gidy = get_group_id(1);
    int lidx = get_local_id(0);
    int lidy = get_local_id(1);

  //Allocate local submatrices
    __local float As[BLOCK_SIZE * BLOCK_SIZE];
    __local float Bs[BLOCK_SIZE * BLOCK_SIZE];

    int aBegin = width * BLOCK_SIZE * gidy;
    int aEnd   = aBegin + width - 1;
    int aStep  = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * gidx;
    int bStep  = BLOCK_SIZE * width;

    float Csub = 0;

  //Do work
    for (int aa = aBegin, bb = bBegin; aa <= aEnd; aa +=
        aStep, bb += bStep) {

        AS(lidy, lidx) = a[aa + width * lidy + lidx];
        BS(lidy, lidx) = b[bb + width * lidy + lidx];

        barrier(CLK_LOCAL_MEM_FENCE);

        for (int kk = 0; kk < BLOCK_SIZE; ++kk)
            Csub += AS(lidy, kk) * BS(kk, lidx);

        barrier(CLK_LOCAL_MEM_FENCE);
    }

  int cc = width * BLOCK_SIZE * gidy + BLOCK_SIZE * gidx;
    c[cc + width * lidy + lidx] = Csub;
}
```

Listing A.1: Version selected by NVIDIA GTX570 and AMD Radeon HD 5870

```
#define BLOCK_SIZE 8
__kernel __attribute__((reqd_work_group_size(8, 8, 1)))
__kernel void BothLocalPrivate(__global float *a, __global
    float *b, __global float *c,
  int m, int n, int k, float alpha, float beta){

   //Get index of element to process
   int gidx = get_group_id(0);
   int gidy = get_group_id(1);

   //Work-item
   int lidx = get_local_id(0);
   int lidy = get_local_id(1);

   //Matrix dimensions
   int widthx = get_global_size(0);
```

```
    int widthy = get_global_size(1);
    int qq = n;

    //Number of submatrixes to be processed by each worker
    int numSubMat = qq/BLOCK_SIZE;

    float4 resp = (float4)(0,0,0,0);
    __local float A[BLOCK_SIZE][BLOCK_SIZE];
    __local float B[BLOCK_SIZE][BLOCK_SIZE];

    for (int k=0; k<numSubMat; k++)
    {
        //Copy submatrixes to local memory. Each worker
            copies one element
        A[lidx][lidy] = a[BLOCK_SIZE*gidx + lidx + widthx*(
            BLOCK_SIZE*k+lidy)];
        B[lidx][lidy] = b[BLOCK_SIZE*k + lidx + qq*(
            BLOCK_SIZE*gidy+lidy)];
        barrier(CLK_LOCAL_MEM_FENCE);

        for (int k2 = 0; k2 < BLOCK_SIZE; k2+=4)
        {
        //Do work in private memory
            float4 temp1=(float4)(A[lidx][k2],A[lidx][k2
                +1],A[lidx][k2+2],A[lidx][k2+3]);
            float4 temp2=(float4)(B[k2][lidy],B[k2+1][lidy
                ],B[k2+2][lidy],B[k2+3][lidy]);
            resp += temp1 * temp2;
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    c[BLOCK_SIZE*gidx + lidx + widthx*(BLOCK_SIZE*gidy+lidy)
        ] = resp.x+resp.y+resp.z+resp.w;
}
```

Listing A.2: Version selected by NVIDIA Tesla C2050