View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Online Research Database In Technology

Technical University of Denmark

Parallel Programming using OpenCL on Modern Architectures

Nielsen, Allan Svejstrup ; Engsig-Karup, Allan Peter; Dammann, Bernd

Publication date:
2012

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Nielsen, A. S., Engsig-Karup, A. P., & Dammann, B. (2012). Parallel Programming using OpenCL on Modern Architectures. Technical University of Denmark (DTU). (D T U Compute. Technical Report; No. 2012-05).

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

IMM Technical Report 2012-05

Parallel Programming using OpenCL on Modern Architectures

Allan Svejstrup Nielsen

Allan P. Engsig-Karup & Bernd Dammann

Abstract

This report is intended as a quick introduction to the OpenCL framework and the aim is to facilitate a smooth transfer into the use OpenCL C for developers with previous GPGPU experience. The purpose of OpenCL is to allow for developers to use all compute resources available on a heterogeneous hardware platform.

As well as being an introduction to OpenCL, the report also presents an overview of AMD GPU hardware, covering both the VLIW5/4 architectures and the upcoming Graphics-Core-Next architecture which is to form the basis of AMDs future generation GPUs that are to be as capable at compute as they are at graphics.

To conclude the presentation of OpenCL as a language for compute, a matrix-matrix multiplication example is devised and optimized for the VLIW4, Tesla and Fermi architectures. The performance is measured as a function of both matrix and work-group size and results are discussed. Where applicable, the equivalent CUDA implementation is tested for comparison.

GPUlab
DTU Informatics
16th March 2012

Technical University of Denmark DTU

Preface

The report at hand was written as part of a 5 ECTS special topic course at DTU Department of Informatics and Mathematical Modeling (DTU Informatics) with Allan P. Engsig-Karup and Bernd Dammann as advisers.

Part of the CUDA code tested and presented within the report was developed during another 5 ECTS course at DTU, 02614 High Performance Computing.

The report represents a brief condensation of much of the current movements within OpenCL and heterogeneous computing.

Contents

I Heterogeneous Computing and OpenCL **1**

1 Scientific Computing Demands Throughput **1**

1.1 A New Market for GPUs . . . 2

1.2 The Open Compute Language . . . 2

2 Trends and Industry Movement **4**

2.1 Programming Paradigms . . . 4

2.2 Current Development in GPU Hardware . . . 5

2.3 OpenCL Gaining Ground . . . 7

2.4 Compute Accelerators . . . 8

3 Outlook **8**

II The OpenCL Framework **9**

4 An Introduction **9**

4.1 Host and Devices . . . 9

4.2 Conceptual Foundations . . . 10

4.2.1 The Platform Model . . . 10

4.2.2 The Execution Model . . . 11

4.2.3 The Memory Model . . . 11

4.3 Contexts . . . 13

4.4 Command-Queues . . . 13

4.5 Programs and Kernels . . . 14

5 Computing with OpenCL **15**

5.1 A General Approach . . . 15

5.2 Code Samples . . . 16

6 The Kernel Programming Language **17**

6.1 OpenCL C . . . 17

6.2 Types and Qualifiers . . . 18

6.3 Built-In Functions . . . 18

6.4 Kernel Compilation . . . 20

7 Summary **20**

III OpenCL Devices and Architecture 21

8 Tuning GPU Kernel Code 22

9 VLIW5/VLIW4 23

9.1 Resource Allocation 25

9.2 Compute Issues with VLIW 25

10 Graphics Core Next 26

10.1 A Non-VLIW SIMD Design 26

10.2 Dissecting a Compute Unit 26

10.3 New Features with GCN 27

10.4 The AMD HD 7970 28

IV Performance Tuning with OpenCL 29

11 Matrix Matrix Multiplication 30

11.1 A Naive Kernel 30

11.2 Exploiting Local Memory 30

11.2.1 Coalesced Global Memory Access 31

11.2.2 Avoiding Bank Conflicts in Local Memory 32

11.2.3 Performance Impact of Work-Group size 32

11.3 Increasing Performance 34

11.3.1 Minimizing Communication 35

11.3.2 Increasing Occupancy 35

12 Device Performance Measurements 36

12.1 AMD VLIW4 - HD 6990 36

12.2 Nvidia Fermi - GTX590 37

12.3 Nvidia Tesla - GTX280 38

12.4 Device Comparison 39

13 Code and Performance Portability 40

V Summary and Outlook 41

14 Conclusion 41

A References 43

Nomenclature

ACE	In the GCN architecture, the two Asynchronous Compute Engines handles resource allocation, context switching, and task priority.
ALU	An arithmetic logic unit is a digital circuit that performs arithmetic and logical operations.
APU	An Accelerated Processing Unit is a processing system that includes additional processing capability designed to accelerate one or more types of computations outside that of a CPU. Examples include AMD Fusion, IBM CELL, Intel HD Graphics, and NVIDIA's Project Denver.
CGMA	Compute to Global Memory Access.
CUDA	Compute Unified Device Architecture is a parallel computing architecture developed by Nvidia.
DMA	Direct Memory Access is a feature of modern computers that allows certain hardware subsystems within the computer to access system memory independently of the central processing unit (CPU).
DSP	A Digital Signal Processor is a specialized microprocessor with an architecture optimized for the fast operational needs of digital signal processing.
ECC	Error-Correcting Code memory is a type of computer data storage that can detect and correct internal data corruption. ECC memory is used in most computers where data corruption cannot be tolerated under any circumstances, such as for scientific or financial computing and as servers.
Fermi	The Nvidia Fermi GPU Architecture is the GPU architecture on which Nvidia GPU products are for the most part based on as of February 2012.
FPGA	A field-programmable gate array is an integrated circuit designed to be configured by the customer or designer after manufacturing?hence field-programmable.
FPU	A Floating-Point Unit is a part of a computer system specially designed to carry out operations on floating point numbers.
GCN	Graphics Core Next is a brand new architecture from AMD, the first product based on this architecture was released January 2012. GCN is AMDs move towards a GPU architecture that is as capable at compute as it is at graphics.
GDDR	Graphic Double Data Rate memory refers to memory specifically designed for use on graphics cards.
GPGPU	General-Purpose computing on Graphics Processing Units (GPGPU or GPGP) is the means of using a graphics processing unit, which typically handles computation only for computer graphics, to perform computations typically otherwise handled by CPUs.
IL	The AMD Intermediate Language is a pseudo-assembly language. The IL code is an intermediate representation compiled from OpenCL code for graphics and compute on AMD cards. Upon execution, the GPU driver on a platform translate the IL into binary code for the GPU to process.

ISA	An Instruction Set Architecture is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the machine language, and the native commands implemented by a particular processor.
Keppler	Keppler is the codename for a new Nvidia GPU architecture that is to be the successor of Nvidias current Fermi GPU architecture.
LDS	Local Data Share is the name of the scratchpad memory within each compute unit in the GCN architecture. In the LDS, data shared between work-items within a work-group is residing.
MIC	Intel Many Integrated Core Architecture or Intel MIC (pronounced Mike) is a multiprocessor computer architecture developed by Intel.
OpenCL	Open Compute Language. OpenCL is a framework for writing programs that execute across heterogeneous hardware platforms.
PTX	Parallel Thread Execution is a pseudo-assembly language. The PTX code is an intermediate representation compiled from CUDA or OpenCL code for graphics and compute on nvidia cards. Upon execution, the GPU driver on a platform translate the PTX into binary code for the GPU to process.
SDK	A Software Development Kit is typically a set of software development tools that allows for the creation of applications for a certain software package, software framework, hardware platform, computer system, video game console, operating system, or similar platform.
SIMD	Single Instruction Multiple Data, is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data simultaneously. Thus, such machines exploit data level parallelism.
SPU	Streaming Processing Unit or just Streaming Processor is the name previously adopted by both AMD and Nvidia for the fundamental calculation units that make up a SIMD.
Tesla	The Nvidia Tesla GPU Architecture is a now discontinued architecture that was used in the Geforce 2xx and Geforce 3xx series. The Tesla Architecture is not to be confused with the Nvidia Tesla Brand which is a line of GPU products designed and marketed specifically towards High Performance Computing.
VLIW	Very Long Instruction Word refers to a processor architecture designed to take advantage of instruction level parallelism. VLIW4 and VLIW5 are GPU architecture namebrands adopted by AMD for the GPUs in the Evergreen and Northern island GPU series.
WebCL	The Khronos Group is working to define a JavaScript binding to the Khronos OpenCL standard for heterogeneous parallel computing. WebCL will enable web applications to harness GPU and multi-core CPU parallel processing from within a Web browser.

Part I

Heterogeneous Computing and OpenCL

In this part of the report at hand, the reasoning behind the industry move towards heterogeneous hardware systems are explored, and the role of OpenCL within this development explained. GPUs are the first compute accelerators, in many years, to find success within general purpose scientific computing. The current hardware trends within this area are discussed with reference to the new programming frameworks, being introduced to program these heterogeneous hardware system.

1 Scientific Computing Demands Throughput

As the usage of computer simulations within engineering and natural sciences increase, so does the demand for computational throughput and speed, from engineers and scientists alike. For the past decades, improvements within manufacturing and processor technologies has been the main driver of performance, allowing new generations of microprocessors to shrink dimensions and thereby reduce power consumption, add transistors and increase operating frequency. All this to add to the throughput generated by general purpose CPUs.

This ride, however wonderful, is over. Since mid 2000s, chip manufactures had to seek out new ways of improving the performance of their hardware. While the number of transistors that could be fitted on a chip continued to increase, the ability to increase clock speed hit a practical barrier often referred to as "the power wall". The exponential increase in power required to increase the micro processors clock frequency hit practical cost and design limits.

The immediate response of the two biggest players in the market, was to add more cores to a die, this however, was not an easy fix, and the added parallelism forces much software to be changed and many algorithms to be revised. It is hard to understate the change that has happened within high performance computing for last couple of years - in addition to the challenges presented by not being able to scale the clock frequency, HPC vendors and users have experienced other problems. The scale out strategy of increasing performance by racking and stacking an ever increasing amount of compute nodes has hit physical limitations of space("The wall wall"), structural support, cooling and electricity supply.

The industryz answer to these issues seem to converge towards what has become known as heterogeneous computing. The idea is to add accelerators to the commodity based racks used. These accelerators are, in contrast to a general purpose CPU, highly specialised units. A general purpose processor will by default include a wide range of functional units, to be able to respond to any computational demand. This is what makes it a general purpose processor. Accelerators are specialised to handle only specific functions or task, i.e. fewer transistors are wasted during compute because only those functional units required by the compute purpose are included.

There is a lot to gain in terms of performance and power from specialised silicon. A recent report released November 2011 from analyst firm IDC argues that heterogeneous computing is going mainstream and will be "indispensable for achieving exascale computing", and already several of the fastest clusters on the TOP500 list use accelerators of some form.[5]

1.1 A New Market for GPUs

High performance computing has for a long time been a scavengers field, and clustered x86 based servers have dominated the market for the last decade and a half. These more general-purpose systems are typically not the best architectures for most problems within scientific computing. However, commodity component based computers were inexpensive, could be racked and stacked, and were continually getting faster, while product development costs for specialized systems could not compete against products sold to retail volume markets.

With the high cost of developing new processor architectures, specialised or not, where does this leave a point of entry for specialised compute accelerators within scientific computing? Well, here come the GPUs. GPUs are the first compute accelerators to be successfully incorporated within scientific computing in a long time. Discrete GPUs are designed for the graphics commodity market, but graphics and compute are becoming increasingly indistinguishable as modern PC games now also utilise the GPUs not only for graphics, but also for in-game physics and other compute intensive tasks. This need applies pressure on the GPU manufactures to deliver products that can deliver on these trends. And once again, commodity hardware fulfill compute needs.

Traditional general purpose CPUs have a substantial part of the chip area dedicated to large caches as well as branch prediction and control logics, to keep a few threads running very fast and keep them from stalling. CPUs are build and designed for control intensive tasks, and they are very good at this. Unfortunately most problems within scientific computing are data or compute intensive rather than control intensive. Graphics is data intensive, and on the GPU chip, the large caches and control logic are removed in place of more ALUs/FPUs. In addition, the GPU memory have a high bandwidth and is close-to-chip. This memory trades bandwidth for high latencies, and the high latency is hidden trough massive multi threading and what is known as zero-overhead scheduling. GPUs are thus designed to handle data intensive tasks, which are very similar to the compute needs within scientific computing, and as they are available as low priced commodity hardware, it is a perfect match in terms of the needs within scientific computing.

1.2 The Open Compute Language

In general, a heterogeneous computing platform consists of microprocessors with different instruction set architectures(ISAs). A modern workstation today can be equipped with multiple CPUs, multiple GPUs from different vendors, and other more exotic accelerators such as DSPs or FPGAs. This multitude of different devices increase the complexity of code and decrease portability by requiring hardware specific code to be interleaved throughout application code. Another challenge lies in balancing the application workload across different devices. This is especially problematic, since different devices typically have different performance characteristics.

This is were the Open Compute Language(OpenCL) enters the scene: OpenCL is developed specifically to ease the burden when writing applications for heterogeneous systems. OpenCL is a new open framework for programming heterogeneous systems, and it is pushed by industry and managed by the non profit technology consortium Khronos Group. As argued above, heterogeneous systems are becoming an important class of platforms, and OpenCL is the first industry standard to address the programming needs that come about with this paradigm change within computing. The OpenCL Working Group within Khronos consists of diverse industry participation, including processor vendors, system OEMs, middleware vendors and application developers.

OpenCL lets programmers write a single portable program that makes use of **all** resources in a heterogeneous platform, and with OpenCL it is possible to write programs that can run on a range of systems, from cell phones, using the OpenCL embedded profile, to nodes in massive super computers. No other parallel programming standard has such a wide reach.

OpenCL 1.0 was first released as royalty-free specifications in December 2008, after Apple proposed an OpenCL working group and contributed with draft specification to the Khronos Group in June 2008. During 2009 multiple conformant implementations were shiped across diverse operating systems and platforms, and in June 2010 OpenCL 1.1 specifications were released and first implementations ship. With OpenCL 1.1 many new features were added, with the most prominent ones being a general uplift in functionality for enhanced programmability as well as new data types. The ability to handle commands from multiple hosts, perform operations on regions of a buffers, enhanced use of events to drive and control command execution, additional OpenCL C built-in functions and improved OpenGL interoperability.

November 15th 2011 at SC11, Seattle, the specifications of openCL 1.2 where released[24], and new features include, but are not limited to

- Device partitioning - enabling applications to partition a device into sub-devices to directly control work assignment to particular compute units, reserve a part of the device for use for high priority/latency-sensitive tasks, or effectively use shared hardware resources such as a cache.
- Separate compilation and linking of objects - providing the capabilities and flexibility of traditional compilers enabling the creation of libraries of OpenCL programs for other programs to link to.
- Built-in kernels represent the capabilities of specialized or non-programmable hardware and associated firmware, such as video encoder/decoders and digital signal processors, enabling these custom devices to be driven from and integrated closely with the OpenCL framework.

With these new additions, OpenCL has become a powerful platform for heterogeneous computing. At SC11, Timothy Mattson from Intel corporation, one of the authors of the OpenCL Programming Guide, disclosed information regarding OpenCL 2.0. Apparently, the OpenCL Working Group within Khronos is working aggressively towards OpenCL 2.0 which will contain many new advancements. Currently, information regarding the specifications of OpenCL 2.0 are very limited, it is however clear that in OpenCL 2.0 we will see a more sophisticated memory model that allows virtual shared memory spanning across platforms! A more flexible execution model is under development with regards to task level parallelism within OpenCL and a work group has been designated to define a standard high level interface to sit on top of OpenCL, this under the name HLL.

In addition to the push for OpenCL 2.0, a new initiative, webCL, was announced in March 2011. WebCL is supposed to provide JavaScript binding to the OpenCL standard for heterogeneous parallel computing. WebCL will enable web applications to harness resources on a heterogeneous platform from within a Web browser, enabling significant acceleration of applications such as image and video processing and advanced physics. More information regarding OpenCL 2.0 should be out late 2012, expect something early 2013.

2 Trends and Industry Movement

2.1 Programming Paradigms

So far, GPUs have received the most attention, and constitute the group of accelerators that have first been widely adopted for compute purposes. OpenCL is not the only programming framework that offers to harness the power of these units. The Microsoft DirectCompute API introduced with DirectX 11 allows developers to use GPUs to accelerate windows application, and Nvidia are aggressively promoting their own framework CUDA by building a vast infrastructure to support developers. CUDA is not restricted to any particular operating system, but only works with Nvidias own products, though efforts are being made now by Nvidia to add support for x86 instruction set architectures.

Microsoft DirectCompute and Nvidia CUDA are limited either to a given operating system or certain GPUs, but OpenCL constitute an entire open source framework to harness parallel compute power from any device spanning from regular CPUs to exotic compute devices such as DSPs or FPGAs, not restricted to any particular operating system or vendor.

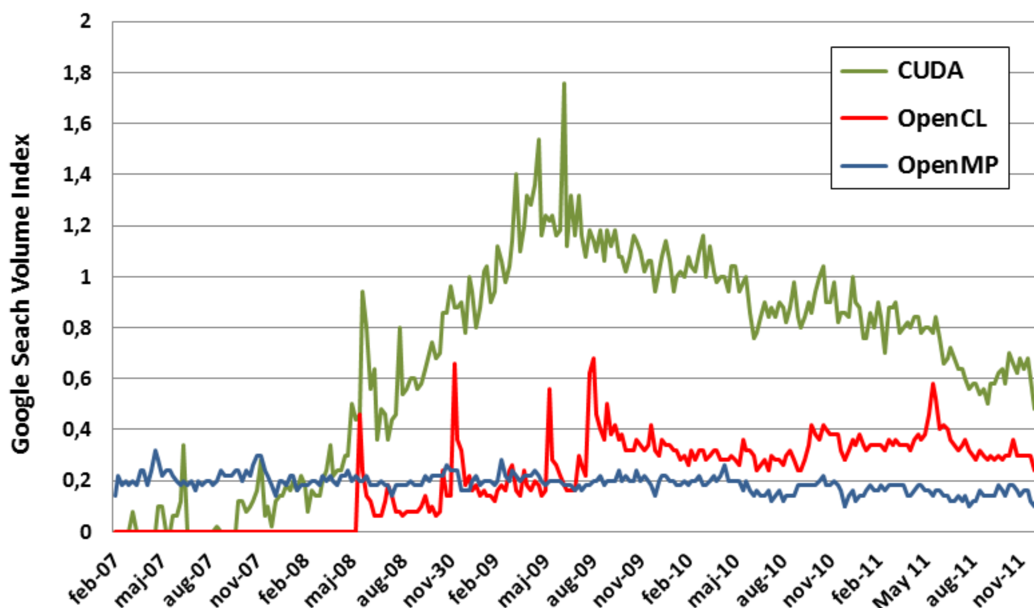


Figure 2.1: Google trends search volume index comparing searches on CUDA, OpenCL and OpenMP. The search volume for DirectCompute is to low to depict.

Since Microsoft's DirectCompute was introduced in 2009, it has received only limited attention from the community of developers, whereas CUDA and OpenCL are supposedly on the rise. OpenCL is by many considered the outsider when it comes to GPU computing, given Nvidias large commitment to promoting CUDA. A search on Google trends reveals that OpenCL is less of an outsider than one might think. In figure 2.1, the amount of searches on CUDA, OpenCL and OpenMP as a reference is depicted, with the search volume index zero set at February 2007, when the first CUDA SDK was made public.

It is clear that CUDA has received the most attention, which in part must be due to the massive marketing efforts by Nvidia. The amount of searches for CUDA seem to be on the decline compared to OpenCL though. One thing is search volume, another is the actual demand by industry for people with experience within these new languages. On SimplyHired.com, a vertical search engine company based in Silicon Valley, supposedly in the

process of building the largest online database of jobs on the planet, it is possible to compare trends of key words within job descriptions harvested throughout the web. In figure 2.2, Cuda, OpenCL and OpenMP are compared in terms of how often these words appeared in job descriptions within an 18 month period. From the figure it is clear how OpenCL is receiving almost as much attention as CUDA does by industry. Also, competences within these languages are already more sought after than OpenMP, which is the defacto standard for writing portable SMP application software.

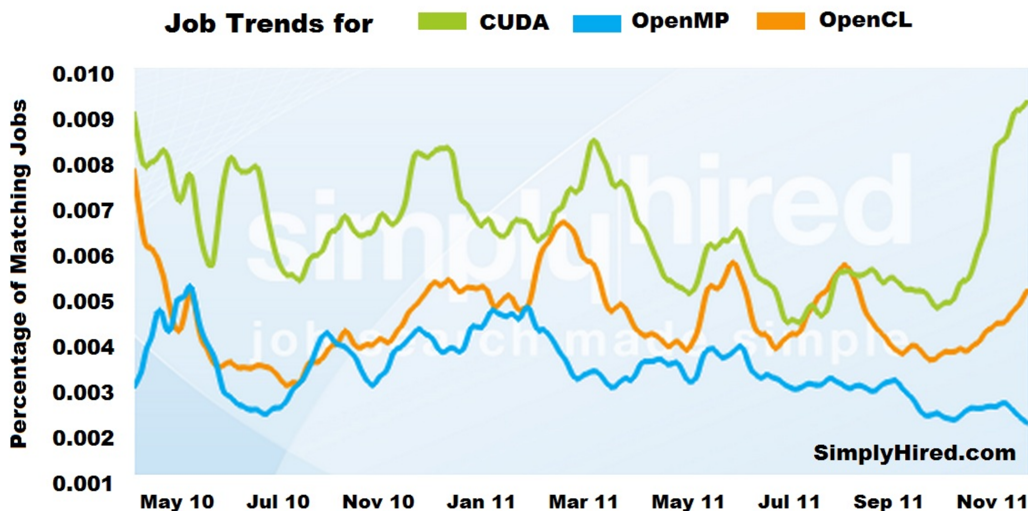


Figure 2.2: SimplyHired.com trend lines comparing how often CUDA, OpenCL and OpenMP appear as key words in job descriptions throughout the web.

There is more than one fundamental difference between CUDA and OpenCL, CUDA is for GPU-CPU devices only, while OpenCL is to be a general language for compute on heterogeneous systems composed of other types of accelerators and even embedded devices. That said, currently the only accelerators for compute having gained wide adoption is the GPUs. The advantage here with OpenCL is that one is not limited to GPUs from a single vendor only.

2.2 Current Development in GPU Hardware

Currently, there are two major producers of GPUs, these being Nvidia and AMD. In table 2.1, the specifications of the current high end products from both companies are listed.

The AMD HD 7970 was released January 2012 and is the worlds first GPU to be fabricated using 28nm technology, and more importantly, it is also the first GPU based on AMDs new "Graphics Core Next", GCN, architecture. AMDs most recent shift was moving to the VLIW4 architecture, essentially a revised VLIW5, in Q4 2010 and now they are making their biggest leap yet with GCN.

GCN is to form the basis of a GPU that performs well at both graphical and computing tasks. AMDs VLIW architecture was stretched as far as reasonable for computing purposes, and as more developers are getting on board for GPU computing, a clean break was needed in order to build a better performing GPU to meet their needs. Some observers regard GCN as AMDs Fermi, as it is supposed to be an entirely new architecture making the GPUs equally capable at computing as they are at graphics.

Currently, Nvidia offers a range of devices all based on their Fermi architecture, products based on the successor Keppler are not expected to be out until sometime mid/end 2012. Nvidia constricts the double precision performance of their Geforce commodity line aimed at personal computers to 1/4 the performance of the equivalent Tesla cards. For many problems within scientific computing, double precision is essential, and as such, with Nvidia products, one is often forced to buy the expensive Tesla cards marketed for compute purposes and workstation use.

The Tesla line of compute and workstation products are essentially carrying the same chip as the high-end consumer products under the Geforce brand. The Tesla line-up cards, however, contains two DMA engines for bi-directional PCIe communication, which can boost performance in compute operations with a tight feedback loop to the main memory. The Tesla line-up cards are also equipped with ECC protected memory and ship with cluster management and GPU monitoring software, and a 18-24 month availability is guaranteed. Furthermore, all Tesla brand cards are stress tested for several days before being shipped.

AMD has yet to release any information regarding Firestream products for compute purposes and workstation use, based on their new GCN architecture. However the HD 7970 card itself carries some interesting features worth mentioning. First, the memory bandwidth of 264GB/s feeding the HD 7970 chip is among the highest yet to be seen. Many applications within scientific computing are memory bound, so this should raise some attention within the community of developers.

Another interesting difference to note is how Nvidia's commodity lines are configured, to deliver only one quarter of the double precision performance of the equivalent Tesla series cards, whereas AMD's new products are configured to deliver half the double precisions performance of the equivalent Firestream cards, giving the AMD commodity cards a substantial advantage in double precision pure compute throughput

However, the HD7970 is still a commodity product, so no ECC memory, and no added engines for bi-directional PCI communication. But as the first GPU card, it does support PCIe 3.0 which has twice the bandwidth of the PCIe 2.1 standard which is used on the Geforce GTX580 and the Tesla C2050. The new AMD launch offers a very cost effective alternative to Nvidias workstation cards with more than 80% higher bandwidth and 80% higher peak theoretical performance at almost one quarter of the price. With the very aggressive performance/price strategy of AMD, it makes it very attractive for developers to build compute applications in OpenCL, to be able to harness the power from these devices.

GPU	AMD HD 6970	AMD HD 7970	Nvidia GTX580	Nvidia Tesla C2050
Architecture	VLIW4	GCN	Fermi	Fermi
Peak Single	2700 GFLOPs	3788 GFLOPs	1584 GFLOPs	1030 GFLOPs
Peak Double	675 GFLOPs	947 GFLOPs	198 GFLOPs	515 GFLOPs
Memory Size	2 GB	3 GB	3 GB	3 GB
Bus width	256 bit	384 bit	384 bit	384 bit
Bandwidth	176 GB/s	264 GB/s	192.4 GB/s	144 GB/s
Price Tag	$324.99	$549.99	$549.99	$1,999.99

Table 2.1

2.3 OpenCL Gaining Ground

OpenCL and CUDA are the two major competing programming frameworks for GPU computing and so far CUDA has received the most attention due to Nvidias concerted efforts to establish CUDA as the dominant programming framework for GPU applications. However, according to some prominent sources [21], OpenCL is now maturing to the extend that focus of efforts by the GPU computing community may be shifting.

The lack of competitive products for high-end GPU computing has been a limitation for the adoption of OpenCL within HPC and so fare OpenCL has received more attention with regards to client-side computing, especially for mobile platforms which increasingly incorporate GPU silicon into chip design. OpenCL for High-Performance computing has also been inhibited by a lack of maturity, which has resulted in lower performance on OpenCL compared to that of CUDA.

Despite the above, AMD has continued to promote OpenCL and recent compiler and library releases have improved performance considerably. Kyle Spafford from the Future Technology Group at Oak Ridge National Laboratory (ORNL) has been benchmarking OpenCL and CUDA for some time and is now convinced that OpenCL performance is on par with that of CUDA. He recently presented his findings at the Georgia Tech's Keeneland Workshop.[22]

Spafford ran the ORNL Scalable Heterogeneous Computing Benchmark Suite (SHOC) that has been optimized for both CUDA and OpenCL and found that OpenCL can match the performance of CUDA on most of the basic math kernels. Only the Fast Fourier Transform code achieves a significant performance advantage with CUDA, and this is attributed to the use of fast intrinsics in the CUDA implementation whereas the OpenCL implementation is employing a slower but more accurate version, see figure 2.3.

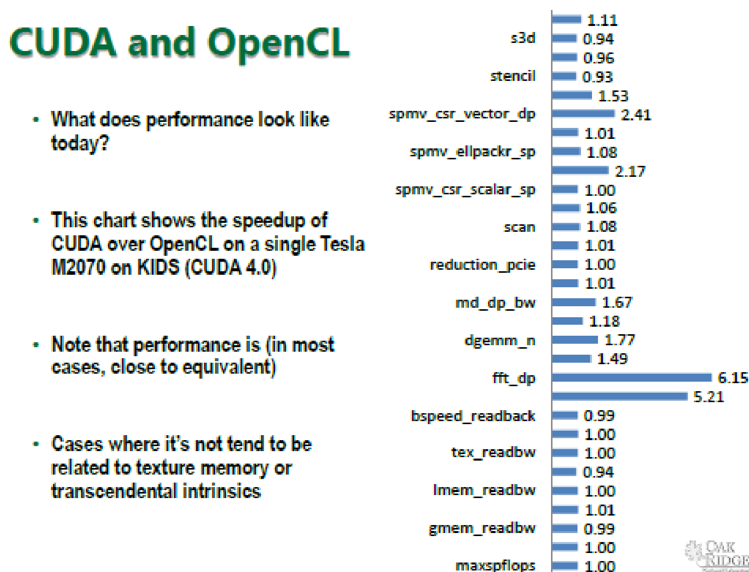


Figure 2.3: CUDA relative to OpenCL performance of basic math kernels on Nvidia hardware. [21]

2.4 Compute Accelerators

GPUs have gained an enormous amount of success within high performance computing in only a few years, but GPUs are not the only accelerators out there hoping to get a piece of the rapidly growing HPC hardware market.

Currently, a multitude of special purpose accelerators from various vendors are being released to the market. Mitrionics, Nallatech and Pico Computing are building FPGA logic boards and Texas Instruments and Altera are building DSPs style devices which can function as accelerators to commodity hardware trough PCIx interfaces [7].

On the Fusion developer summit June 2011, AMD disclosed more details around their APU(accelerated processing unit) work, announcing their first product commodity APU products for consumers. At Fusion11, roadmaps were presented stressing how APUs are not only to be a consumer product, but will also eventually form the basis of AMD's efforts within high performance computing[9]. APUs, as such, are not a new class of accelerators, but rather a CPU device and a GPU device merged onto the same chip, sharing the same memory space.

Intel is also in the process of building its own compute accelerator, this under the architecture codename MIC (Many Integrated Core). Prototype devices of this compute accelerator, called Knights Corner, are already being implemented in large HPC clusters. This new architecture is based upon Intels abandoned Larrabee project, and something in between GPU and CPU. The purpose of the device is compute acceleration, but it can be programmed with standard approaches such as Fortran, C++ and so on. When promoting the MIC, Intel is stressing the ease at which costumers can port their application to use accelerators[8].

3 Outlook

Scientist and engineers alike, who use computer simulations as part of their work have come to rely on an ever increasing supply of computational throughput and speed. Within recent years, this supply has come under pressure since the speed of single core general purpose CPUs can no longer be scaled due to the constraints imposed by the very laws of physics. The demand for more computational throughput is still there though, and so a new approach of increasing throughput is needed. The industry has decided that this approach is to be heterogeneous computing [6].

The move towards heterogeneous computing is supported by the entire industry, but the language and manor in which these systems are programmed has yet to be defined. OpenCL pose one way of going about it, and promises fully portable solutions. Within the next couple of years, AMD will be pushing its efforts on APUs, Intel is launching the MIC build for High Performance Computing and Nvidia will undoubtedly continue pushing CUDA to the masses. As such, it is difficult to make predictions as to where things are heading within high performance computing, the only thing that is fairly certain is that it is going heterogeneous.

Part II
The OpenCL Framework

4 An Introduction

OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. In OpenCL, parallel compute work is performed in kernels, and these kernels are executed on OpenCL devices. The kernels are written in a language called OpenCL C which is an extension of C99 with certain limitations and additions. OpenCL provides parallel computing using task-based and data-based parallelism and it has been adopted by Intel, IBM, AMD, Nvidia and ARM among others.

Learning OpenCL can seem overwhelming. The API is designed so to be able to handle parallel compute operations in diverse domains, from super computing to embedded devices. Thus, OpenCL is a lot more than just the kernel programming language OpenCL C. However as we shall see, there is a classic approach that any OpenCL program follows.

1. Discover the components that make up the heterogeneous system
2. Probe the characteristics of these components so that the software can adapt to the specific features of different hardware elements
3. Create the blocks of instructions (kernel) that will run on the platform
4. Set up and manipulate memory objects involved in the computation
5. Execute the kernels in the right order and on the right components of the system
6. Collect the final results

Setting up the code to go trough the above listed steps can be tedious, but once written and once learned, the full span of generality of OpenCL opens up and this code can often be reused for other projects.

4.1 Host and Devices

Within the OpenCL paradigm, a "host program" is the outer control logic that handles the execution of an application. This host program would normally run on a general purpose CPU such as the x86 processor in most computers. Compute work is offloaded to OpenCL devices trough the use of kernels. The hardware on which the host program runs can still be used for compute and execute these kernels.

OpenCL devices are grouped into three categories, these being CPUs, GPUs and Accelerators, and the actual compute work takes place on these. As of OpenCL 1.2, device partitioning is possible. In OpenCL 1.2, a device can be partitioned into sub-devices so to directly control work assignments to particular compute units and thereby reserve parts of a given device for high priority or latency sensitive tasks if needed.

As mentioned previously, in OpenCL, one distinguish between three types of devices; CPUs, GPUs and accelerators. The CPU device is a single homogeneous device that maps across a

set of available cores. The GPU devices correspond to the throughput optimized devices marketed toward graphics and general-purpose computing. The final class of OpenCL devices is the accelerator device, this class is intended to cover a broad range of devices ranging from IBMs Celll Broadband architecture to DSP style devices.

4.2 Conceptual Foundations

To ease the transition to, and the understanding of the OpenCL framework, the conceptual foundations and models are discussed on a high level in this section. For a detailed list and walk trough of API calls, the OpenCL Programming Guide [1] provides a good reference. Also, numerous sources online exist[25]. Particular important API calls will appear within this section, along with the equivalent CUDA terminology when applicable.

4.2.1 The Platform Model

The OpenCL platform model defines, that within an OpenCL application there is always a single host and it is this host which handles input and output as well, as the control flow within the application. The host is connected to one or more devices, and it is on the devices where work is done. The devices are also at times referred to as compute devices. An OpenCL device contains one or more compute units, and each compute unit contains one or more processing elements (PEs); this is schematically shown in figure 4.1. The OpenCL mapping to the actual hardware is implementation specific, and platform SDKs from various hardware vendors are available. A complete list of conformant products can be found on the Khronos Group webpage[15].

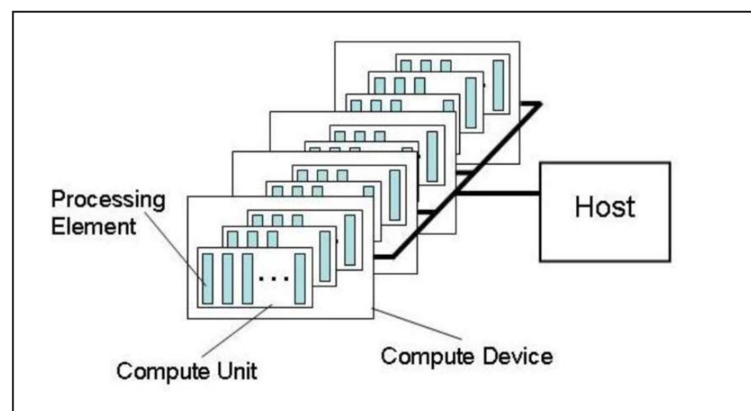


Figure 4.1: [24] A host will include one or more OpenCL devices defined in the SDK. Each OpenCL device contain one or more Compute units which again contain processing elements.

4.2.2 The Execution Model

The compute work within an OpenCL application is carried out in a collection of one or more kernels. These kernels execute on OpenCL devices. Within the OpenCL execution model, it is defined how these kernels execute. Kernels are defined on a host and the host program issues a command that submits the kernel for execution on a specified OpenCL device. The kernels are executed on an integer index space created within the OpenCL runtime. An instance of the kernel code is executed for each point within the index space. Each instance of an executing kernel is called a work item, and is identified by its coordinates, the global ID, in the index space.

The collection of work items all issue the same sequence of instructions defined within the kernel code, programmed in OpenCL C, but the behaviour of each work item can differ due to branch statements within the code. Work items are organized into work-groups, and the work-groups provide a more coarse grained decomposition of the index space and always exactly span the global index space.

The way kernels are executed in OpenCL is very similar to how they are executed within the CUDA framework. In table 4.1, a translation of terminology used within the device memory model and the execution model in CUDA and in OpenCL is given. In OpenCL, threads are called work items and a thread-block is a work group. In section 6, the work item build in functions to be used within kernel code are covered.

CUDA Terminology	OpenCL Terminology
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Local Memory	Private Memory
Thread	Work-item
Thread-block	Work-group

Table 4.1: Table illustrating terminology difference within the CUDA and the OpenCL compute execution model.

4.2.3 The Memory Model

One of the challenges and issues when programming a heterogeneous compute system, is managing multiple address spaces. The host has the familiar address space expected on a CPU platform, but the devices can have a range of different memory architectures. In OpenCL, this is handled with the introduction of memory objects. The objects are explicitly defined on the host and explicitly moved between the host memory space and that of a given OpenCL device. This adds a layer of complexity when programming, but lets OpenCL support much wider range of platforms. On the SC11 conference, November 2011, Seattle, directions for OpenCL 2.0 was briefly introduced during a presentation by Intel employee Tim Mattson. Tim Mattson is working within the Khronos OpenCL group, and disclosed that one of the things they are working on with OpenCL 2.0 is to have a unified memory space that simplifies handling of memory objects. Data is still to be moved so to ensure that is is always close to compute, but the goal is to abstract this away from developers as much as possible.

There are two different types of memory objects defined within OpenCL. Buffers and image objects. A buffer object is a continuous block of memory onto which a programmer can map

data structures and access them trough pointers. Buffer objects provide flexibility to define just about any data structure that the programmer wishes, subject to the limitations of OpenCL C naturally. Image objects are restricted to holding image data structures. Image storage formats may be optimized to the needs of a specific OpenCL device. OpenCL give a vendor implementation the freedom to customize the image format, and the image memory objects are therefore opaque in the sense that the content can only be manipulated trough functions provided in the OpenCL framework. Image objects are not covered in detail in this report, refer to chapter 8 in the OpenCL programming Guide [1] for more information.

The OpenCL memory model defines five distinct memory regions.

Host Memory: This memory region is visible only to the host. As with most details concerning the host, OpenCL defines only how the host memory interacts with OpenCL objects and constructs.

Global memory: This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or written to any element a memory object in global memory. Reads and writes to global memory may be cached depending on the capabilities of the device.

Constant memory: This memory region of global memory remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory. Work items have read only access to these objects.

Local memory: This memory region is local to a work-group, and the memory can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. For OpenCL mapping to AMD hardware, see part III.

Private memory: This region of memory is private to a work-item. Variables defined in one work items private memory are not visible to other work items.

In figure 4.2, the relation between memory regions and the execution model is shown. When the host needs to interact with the OpenCL memories, it does so in one of two ways. By explicitly copying data or by mapping and unmapping regions of a memory object. Manipulating these memory objects is done trough their associated contexts.

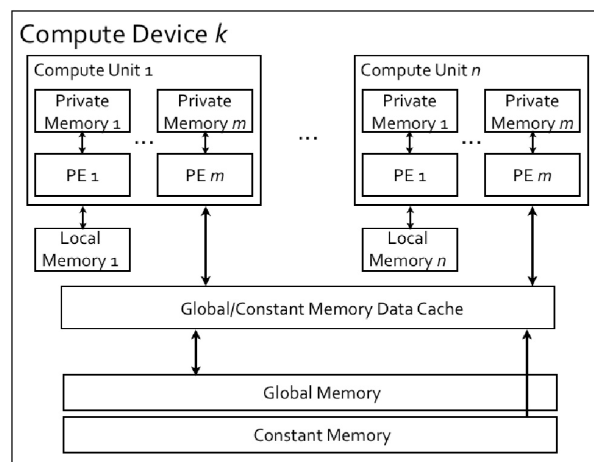


Figure 4.2: The schematics show how the OpenCL execution model map to the OpenCL memory model.

4.3 Contexts

Contexts within OpenCL are used as a sort of container. As the name implies, the context defines the environment, in which the kernels are defined and execute. A context coordinates the mechanisms for host-device interaction, manages the memory objects, and keeps track of the programs and kernels created for a each device within the context. Contexts are thus essential to the use of OpenCL. Contexts are created and can be manipulated by the host using functions from the OpenCL API. More specifically, a context is defined in terms of the following resources

Devices: A Collection of OpenCL devices.

Kernels: The OpenCL functions that execute on OpenCL devices.

Program objects: The program source code and executables that implement the kernels

Memory objects: A set of objects in memory that are visible to OpenCL devices and contain values that can be operated on by instances of a kernel

Here the program objects contain the code for the kernels, and the program object is built at runtime within the host program, kernel and program objects are covered in section 4.5.

4.4 Command-Queues

Interaction with devices is done only by submitting commands to a command queue. Once the host decides which devices to work with and a context is created containing these device, one command queue needs to be created per device. That is, each command queue is associated with one device only. Whenever the host needs an action to be performed by a device, it has to submit a command to the command queue associated with the device. OpenCL supports three types of commands that can be submitted to a command queue.

- **Kernel execution commands** execute a kernel on the processing elements of an OpenCL device.
- **Memory commands** transfer data between the host and different memory objects, move data between memory objects.
- **Synchronization commands** put constraints on the order in which commands execute.

In a typical application, after the tedious work of defining contexts, associating devices as well as initialising memory objects, kernels and programs, attention shifts to the command queue. First a command for moving memory objects to a device is submitted to the command queue of the device to be used. Secondly, a kernel is submitted to the same queue for execution and finally a command for moving data from device to host is submitted. In this example, one needs to add synchronization commands to ensure that previous commands have finished before moving on to the next.

Multiple kernels can be submitted to the same queue, and these kernels may even need to interact in some form. If so, they have to use synchronization commands. To support more complicated interactions, commands submitted to the command queue generate event

objects. A command queue can be told to wait until certain conditions on the event object exist, and these events can also be used to coordinate execution between the host and the OpenCL device.

Commands submitted to a commands queue always execute asynchronously to the host program, that is, the host program submit commands to the command queue and then continues without waiting for the commands to finish. If this is necessary, synchronization commands that act as a barrier until a given command has finished exists, more on this in section 6.3. Commands within a single queue execute relative to each other in on of two modes

- In-order execution: Commands are launched in the order in which they appear in the command-queue and complete in order. In other words, a prior command o the queue completes before he following command begins. This serializes the execution order of commands in a queue.
- Out-of-order execution: Commands are issued in order but do not wait to complete before the following commands execute. Any order constraints are enforced by the programmer through explicit synchronization mechanisms.

In-order execution is supported by all platforms, but out-of-order execution is optional. Out-of-order execution is convenient for load balancing many work loads so to keep all compute units on a device fully engaged.

4.5 Programs and Kernels

Program and kernel objects are two of the most important objects within OpenCL. In OpenCL, the functions that execute on a device are known as kernels and these functions are written in the OpenCL C kernel programming language covered in section 6.1. In order to pass arguments and enqueue the kernel function for execution on a device, an application must create a kernel object from the kernel function. Kernel objects are created from program objects, and the kernel objects may be seen as a handle to kernel functions.

A program object contains a collection of kernel functions. The program object is created from kernel function source code or compiled program binary as well as build information for any of the devices to which the program object is attached. The kernel object, created from a program object, is then used to access properties of the compiled kernel function, enqueues it for execution on a device, and sets argument calls.

There is a number of API calls available to create and build program objects and to query the program objects for information. The API call to create program object takes a context as well as the OpenCL C kernel function source code of one or multiple kernel functions. When the program is created, it is thus associated with a context and contain a number of kernel functions. The program needs to be built for a set of devices. When building the program, one needs to pass the program object to be built, a list of devices to build the program to, and various compile options, a comprehensive list of these can be found in [1] table 6.1-6.4.

With the program object, containing kernel function source code and build information, one can create the kernel objects that acts as handles to the kernel functions. To generate a kernel object, one has to pass a program object along with the name of the kernel function for which to create the kernel object as arguments to the relevant API call.

The kernel object created is thus associated to a context and a set of devices trough a program object, and one now sets arguments to the kernel object as well as enqueue it for execution on any device included in the associated program object.

5 Computing with OpenCL

5.1 A General Approach

As mentioned, programming with OpenCL can seem overwhelming initially since the programmer is not only left with the burden of writing and optimizing low level code within kernels in OpenCL C, but also has to apply many API calls to even initiate an application. However, all application, no matter how small or big, follow approximately the same pattern stated below

1. Query system for available platforms, choose a platform.
2. Query system for available devices on the chosen platform, choose one or more devices.
3. Create a context, and associate one or multiple devices to the context.
4. Create command queue within a context for each device that is to be used.
5. Create a program object on the context with OpenCL C kernel function source code.
6. Build program object for a range of devices.
7. Generate kernel object from program object for a kernel function.
8. Set arguments to kernel object.
9. Enqueue transfer of data from host to device.
10. Enqueue kernel object for execution on device.
11. Enqueue transfer of data from device to host.

5.2 Code Samples

In this section, four functions that may be useful in writing a simple OpenCL program and in getting to know OpenCL are presented. These functions are used in the OpenCL application in IV, were the performance of various OpenCL matrix matrix multiplication kernels are investigated. There are numerous OpenCL examples available online, particularly on the webpages of Nvidia and AMD[12].

The first function presented prints an enumerated list of the platforms available on the system to the command line and asks the user to choose which platform to use. The syntax to call the function `getUserDefPlatform` is stated below and the function code can be found in appendix 14. The integer `p` is the number in the list of platforms returned that the command line user had chosen.

```
cl_platform_id* platformIds;
char* platformName;
int p;
getUserDefPlatform(&platformIds, &platformName, &p);
```

After retrieving a platform, one can query the system for devices available on the platform. A function `getUserDefDevice` for doing so is given in appendix 14. The function again prints an enumerated list of devices available on a given platform to the command line and then asks the user to choose which device to use. Here the integer `d` is the number in the list of devices returned that the command line user had chosen.

```
cl_device_id* deviceIds;
char* deviceName;
int d;
cl_uint deviceCount;
getUserDefDevice(platformIds, p, &deviceIds, &deviceName, &d, &deviceCount);
```

With a device, one might be interested in retrieving device information such as global memory, number of compute units and so on. A function `getDeviceInfo` for doing so is given in appendix 14, and the function call is stated below

```
getDeviceInfo( cl_device_id* ID, cl_uint deviceCount );
```

The function takes a pointer to the first element in an array of `cl_device_ids`, and `cl_uint` number of elements in the array. With a platform and devices retrieved, one also need to create and build a program. The function that was used to do so in part IV is given in appendix 14 and the function call is stated below. The function takes a context, an array of devices as well as the number of devices and the filename to the kernel source code.

```
cl_program program;
buildProgramFromSources(context, deviceCount, deviceIds, "kernelFileName.cl"
    , &program);
```

6 The Kernel Programming Language

6.1 OpenCL C

The kernels in OpenCL are programmed in a language called OpenCL C which is an extension to the C99 standard. The kernel programming language shares many similarities with "C for CUDA" which is used to program kernels in the CUDA framework, and so developers who are experiences within CUDA will find the transaction to OpenCL very smooth. Due to the many similarities, where different naming conventions are the only thing separating C for CUDA with OpenCL C, references to equivalent CUDA terminology is given throughout this section.

OpenCL C imposes a number of restrictions compared to regular C99, key restrictions are

- **No** function pointers
- **No** bit-fields
- **No** variable length arrays
- **No** recursion
- **No** standard headers

The full list of restrictions can be found in the OpenCL specification, or at the end of chapter 4 in the OpenCL programming guide. OpenCL C adds the following features to C99:

- **Vector data types.** Many OpenCL devices, both CPUs, GPUs and accelerators support vector instructions. The vector instruction set is accessed in C/C++ code through build-in functions or device specific assembly instructions. In OpenCL C, vector data types can be used in the same way scalar types are used in C. This makes it much easier fore developers to write vector code because similar operators can be used for both vector and scalar data types. It also makes it easy to write portable vector code because the OpenCL compiler is now responsible for mapping the vector operations in OpenCL C to the appropriate vector ISA for a device.
- **Address space qualifiers.** OpenCL devices, such as GPUs, implement a memory hierarchy. The address spaces qualifiers are used to identify a specific memory region in the hierarchy
- **Additions to the language for parallelism.** These include support for work items, work groups, and synchronization between work items in a work group
- **Images.** OpenCL C adds images and sampler data types and built-in functions to read and write images.
- **Built-in functions.** OpenCL C adds an extensive set of built-in functions such as math, integer, geometric and relation functions

6.2 Types and Qualifiers

In OpenCL C, most scalar data types of C99 are supported. The 64 bit floating point data type called `double` are optional, and only available if `cl_khr_fp64` is supported by the device. OpenCL C also comes with optional support of the `half` data type, which is a 16 bit floating-point type conformal to the IEE 754 half precision storage format. The data type half can be used only to declare a pointer to a buffer that contains half values.

In addition to supporting most scalar data types, OpenCL C adds support for vector data types and the extensive set of build in functions are for the most part also supported on these. The vector data types are essential to use within some device architectures, how so is covered in detail in part III where OpenCL mapping to various devices are presented.

The vector data type is defined with the type name, that is `short`, `int`, `float`, `long`, `double`, `half` followed by a literal value `n` that defines the number of elements in the vector. Supported values of `n` are 2, 3, 4, 8 and 16. The specific API calls to use vector data types in OpenCL can be found in the OpenCL specification or the OpenCL programming guide.[1]

OpenCL C supports a range of qualifiers, the most important ones to know is the function qualifiers and the address space qualifiers. In table 6.1, these qualifiers are listed along with the equivalent C for CUDA terminology.

CUDA Terminology	OpenCL Terminology
`__global__` function	`__kernel` function
`__device__` function	no annotation needed
`__constant__` variable declaration	`__constant` variable declaration
`__device__` variable deceleration	`__global` variable deceleration
`__shared__` variable declaration	`__local` variable deceleration

Table 6.1: The table show qualifiers that are added to functions and data when writing kernels in both CUDA and OpenCL. The biggest difference between the two is that in CUDA, `__global__` functions are GPU entry points, and `__device__` functions are to be executed on the GPU, but are not callable from the host. In OpenCL, entry point functions are annotated with the `__kernel` qualifier, but non-entry point functions do not need to be annotated.

6.3 Built-In Functions

Within the OpenCL C programming language, a rich set of build-in, math, integer, geometrical and relational functions for scalar and vector data types are included. Many of these build-in functions are similar to the functions available in common C libraries such as the functions defined in math.h. A comprehensive list of supported functions can be found in the OpenCL specification or the OpenCL Programming Guide [1] which also includes descriptions of how to use Atomic functions and functions that operate on the special Image memories.

Work-item Functions

When programming a kernel for compute purposes, one will almost surely encounter the use of work item functions. The work-item functions are functions called in a kernel that identifies the position, of a given work-item executing, within the integer index space on which the kernel is executed.

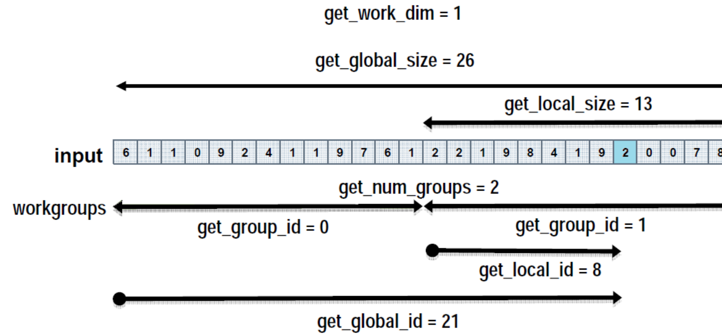


Figure 6.1: Example of work-item functions

Figure 6.1 shows an example on how the global and local work sizes can be accessed by a kernel executing. In this example, a kernel is executed over a global work size of 26 items and a work-group size of 13 items per group. C for CUDA and OpenCL share a similar kernel execution model, so developers familiar with C for CUDA will find the terminology comparison in table 6.2 useful, as well as the associated descriptions.

An important thing to take notice of is how the OpenCL specification do not describe how the global and local IDs map to work-items and work-groups. Within an application, one cannot assume that a work-group whose group ID is **0** will contain work-items with global IDs `0 ... get_local_size(0) - 1`. The actual mapping is determined by the OpenCL implementation and the device on which the kernel is executing.

CUDA Terminology	OpenCL Terminology	Description
gridDim	get_num_groups()	Returns the number of work-groups that will execute a kernel.
blockDim	get_local_size()	Returns the number of local work-items.
blockIdx	get_group_id()	Returns the work-group ID.
threadIdx	get_local_id()	Returns the unique local work-item ID, i.e. work-item within a work-group.
No direct equivalent	get_global_id()	Returns the unique global work-item ID.
No direct equivalent	get_global_size()	Returns the number of global work-items.

Table 6.2: Indexing functions for use in Kernels. The table shows the various indexing mechanisms provided by CUDA and OpenCL. CUDA provides kernel indexing via special pre-defined variables, while OpenCL provides the equivalent information through function calls. OpenCL also provides global indexing information, while CUDA requires manual computation of global indices.

Synchronization Functions

As in C for CUDA, OpenCL C implements a synchronization function. In OpenCL this function is called `barrier`. The barrier synchronization function `void barrier(cl_mem_fence_flags flags)` is used to enforce memory consistency between work-items within a work-group, there is no way to synchronize different work-groups in a kernel. Within a work-group, all work items must execute this function before any are allowed to continue execution beyond the barrier. The flags argument specifies the memory address space and can be set to a combination of the following lateral values

- `CLK_LOCAL_MEM_FENCE`: The barrier function will either flush any variables stored in local memory or queue a memory fence to ensure correct ordering of memory operations to local memory.
- `CLK_GLOBAL_MEM_FENCE`: The barrier function will queue a memory fence to ensure correct ordering of memory operations to global memory. This can be useful when work-items, for example, write to buffer or image memory objects and then want to read the updated data.

6.4 Kernel Compilation

When compiling OpenCL C kernels using the nvcc compiler, the code is translated into PTX(Parallel Thread Execution) code. PTX is a pseudo-assembly like language described in [13]. The nvidia graphics driver contains a compiler which translates the PTX code into a binary code which can be run on the Nvidia GPUs. Currently, the PTX intermediate representation can be obtained by calling `clGetProgramInfo()` with `CL_PROGRAM_BINARIES`. Whit the AMD stream SDK platform, the kernel code is compiled into the assembly like Intermediate Language(IL), described in [14] which again is compiled by the AMD graphics driver into a binary that can run on the AMD GPUs.

7 Summary

OpenCL provides many benefits, the most important one being portability. OpenCL kernels can execute on GPUs and CPUs from both Intel, AMD, Nvidia and IBM and new OpenCL capable devices appear regularly. OpenCL constitutes a full framework for heterogeneous computing. Despite the advantages, OpenCL has some significant drawbacks. As many sources on-line will tell you, it is not easy to learn, and even introductory applications are difficult for newcomers to grasp. Another drawback for more experienced developers is that the infrastructure in terms of libraries to support OpenCL is still quite limited. Also, whereas say CUDA is an all in one packet for developers, OpenCL is mostly only a language description i.e. with OpenCL, the SDK, IDE, debugger etc, come from different vendors, and one has to go to different places to obtain them.

The OpenCL group within Khronos is working on a higher level interface, to be specified with OpenCL 2.0, so to make it simpler to program applications with OpenCL. With time the infrastructure will get better, the amounts of libraries available is increasing already, see reference [18]. OpenCL, despite being hard, is the only language dedicated to compute purposes on heterogeneous platforms, and provides an extensive set of features for doing so. In the next part of this report, the compute architecture of AMD OpenCL devices is discussed, including how the OpenCL memory and execution model map to the devices.

Part III

OpenCL Devices and Architecture

OpenCL is designed so that the execution model and memory model can be mapped to a wide range of architectures, allowing for tuning and acceleration of kernel code. How the OpenCL execution and memory model map to a given device is implementation specific.

As previously mentioned, devices are split into three classes within OpenCL. CPUs, GPUs and accelerators. In this report the emphasis is on GPU devices for compute purposes. There are two main vendors of GPUs, these being Nvidia and AMD. Nvidia is aggressively promoting its own framework for programming GPUs, CUDA, and in the process the architecture of both Tesla and Fermi have become well documented in numerous sources.

AMD hardware is of interest within the context of this report since OpenCL is the only viable framework to program these in. Also, an attractive property of AMD hardware is that it, specification wise, promises more performance at a smaller cost than Nvidia hardware. Currently the AMD GPU products within the Nothern Island series are all based on the VLIW4 architecture, a revision of the VLIW5 architecture used in the Evergreen series, but the next generations of AMD GPUs will be based on the Graphics Core Next architecture. The first card in the new Southern Island series, which is to be based on GCN architecture, was announced December 2011 and launched January 2012. The AMD HD 7970 is a single chip high-end card with a 264GB/s memory bandwidth, 947GFLOPS peak double precision floating point performance and PCIe 3.0 interface. The HD 7970 is available in stores at a USD 549 price tag. With these numbers AMD are sure to raise some attention within the GPGPU community.

In the following sections, the VLIW4/VLIW5 architectures are presented, followed by a discussion about the pros and cons of the VLIW4/5 and what lead AMD to move towards an entirely new architecture, the GCN. Information about the compute capabilities of the GCN architecture is still somewhat scars, the compute capabilities are presented based on the information that could be found from mainly slides on the GCN architecture from on AMD Fusion Developer Summit June 2011 [9] as well as GCN reviews on Anandtech and Tomshardware [27].

8 Tuning GPU Kernel Code

With different GPU vendors and different architectures, one might fear that optimizations to kernel code on one architecture would yield no benefits on other architectures. This is, fortunately, not entirely true. Whereas the difference between CPU and GPUs is vast in many ways, GPU architectures are sufficiently similar that kernel code which performs well on one architecture is *likely* to also perform reasonable well on another GPU architecture. This claim is justified with actual performance measurements in part IV.

GPU cores, which are mapped to OpenCL as compute units, are typically wide vector based cores and have high-bandwidth, high-latency global memories in which the latency is hidden through massively multi-threading and zero overhead scheduling. They incorporate local scratch-pad memories, that can be exploited to increase memory throughput to ALUs, and are all in all throughput optimized.

As such, the different GPU architectures currently on the market will benefit from many of the same performance/tuning considerations. This is good news for people familiar with kernel optimization in C for CUDA on the Tesla and Fermi architectures, since their knowledge of optimizing applications from these architectures is equally valuable when optimizing applications to AMD GPU architectures. Below is a list of considerations one should keep in mind when optimizing kernels for any GPU architecture.

- Facilitate high occupancy of work-groups on compute units
- Always read/write to global memory in coalesced fashion
- Exploit local memory to reduce load and stores in global memory
- Avoid bank conflicts when utilizing local memories
- Minimize redundant communication, global-local-private memory

That said, OpenCL code is portable, but there are no guarantees that it is also performance portable. It is possible that code which performed very well on one architecture is slow on another, and for this reason efforts within auto-tuning of applications are now being made. Performance portability issues can have many reasons, bust it is likely to occur due to different register and scratch pad memory space per compute unit on the new architecture. Work items(in work-groups) are assigned to compute units under the limitations of available registers and local memory, as required by the kernel code. The number of work items assigned to a compute unit is reduced on a per work-group basis, and if moving to a new architecture results in a mismatch yielding low occupancy on the compute units, performance drops drastically.

GPU architectures as a whole, from a birds eye view, seem very similar, but there are some notable differences in between VLIW4/5, GCN, Tesla and Fermi. These difference should be considered when optimizing kernel code for a given architecture. In the following sections, these architecture specifics are discussed with reference to the VLIW4 architecture of the AMD HD 6970 along with the presentation of VLIW4 itself.

9 VLIW5/VLIW4

The Radeon HD6970, which is of interest here, is based on the VLIW4 architecture. The schematics of the chip is shown in figure 9.1 The device is divided into two halves where scheduling and dispatch is performed by the level wave scheduler for each half. There are 24, 16 lane wide SIMD cores, each lane (SPU) executes four way VLIW instructions and contains private level 1 cache as well as local data share. The OpenCL mapping is such, that a SIMD correspond to a compute unit and each SPU corresponds to a processing element.

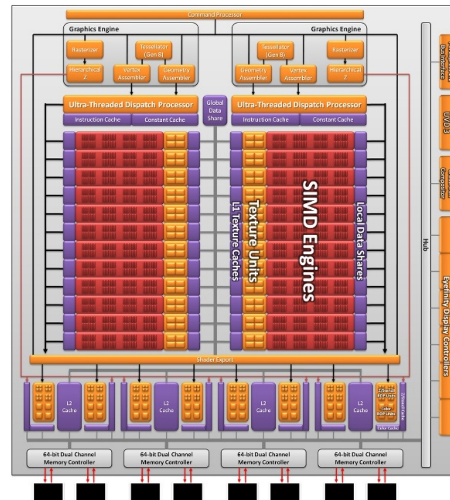


Figure 9.1: [26] The AMD Radeon HD6970 GPU chip schematics.

The fundamental execution unit of AMDs previous designs, used in the Evergreen and Northern Island series of GPU products has been the Streaming Processing Unit, previously known as the SPU. In every modern AMD design other than the Cayman (69xx) this is a Very Long Instruction Word 5 (VLIW5) design. With Cayman this was reduced to VLIW4. Each of these fundamental execution units contains 5 or 4 fundamental math units, which can execute individual instructions in parallel. In terms of OpenCL lingo, each of these SPs constitute a processing element.

The compute unit, that is, the SIMD core contains 16 VLIW4 SPs, and each SP has access to a 32Kb local data share. The local data share is in OpenCL terms the local memory which can be accessed by work items within the same work-group, each work-group being dispatched to a compute unit. During execution, the work-group is divided into "wavefronts". A wavefront consists of 64 work items, and is thus processed on the 16-wide SIMD over four cycles.

Returning to the concept of VLIW4, each SP (OpenCL processing element) has the ability to conduct four independent instructions in parallel. For high performing code, it is thus vital to ensure some form of explicit vectorization within the kernel code. This helps the compiler extract independent instructions to the pipeline. In figure 9.2, an example of an instruction stream executing on a VLIW4 processing element and the same instruction stream executing on a Tesla architecture processing element is given. Under ideal circumstances, where all ALUs are being fed with independent instructions, the VLIW is very efficient, but if there is little or no instructions independence for the compiler to extract, then a lot of ALUs will lie idle and one cannot expect to get close to peak performance.

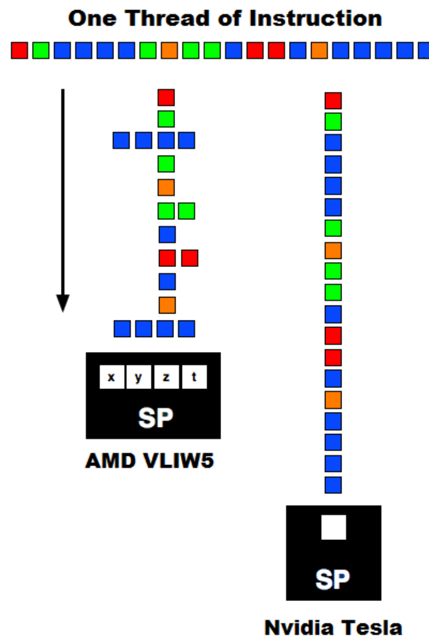


Figure 9.2: [26] Instruction stream executing on VLIW4 and Tesla architecture processing elements

As figure 9.2 illustrates, the HD6970 can benefit hugely by having multiple independent instructions being performed within each work item. With high occupancy of the ALUs within each VLIW, the architecture excels, since control work is moved to the compiler which saves chip space that can be used for even more ALUs. On the other hand, when little instruction independency is present, chip area is wasted on ALUs lying idle. The VLIW approach of the processing elements is in contrast to Nvidias Tesla and Fermi architectures which rely entirely on scalar streaming processors.

Another concern, which the developer should be aware of when programming kernel code for the HD6970, is how control flow executions is handled. All branching and mask management is handled on the scheduler, see figure 9.1, and the latency for doing so is on the order of 40 or more cycles between executing one clause from a given wavefront until executing the next. Different wavefronts execute at different point in their instructions streams. All branching is performed at wavefront granularity and any sub-wavefront branching will require replaying of the clauses within the wave scheduler until all combinations have executed and hence this divergence creates inefficiency. Combining this with the massive 40 cycle latency to the scheduler for each clause, the HD6970 is not just bad at branch divergence, it is *very* bad at it, and can only be covered if the code contains a lot of ALU work and that enough wave fronts can reside within each SIMD.

In the next section we will take a look at how register and local memory resources are allocated. If the reader is familiar with C for CUDA kernel programming and optimization on Tesla and Fermi GPUs, then he will recognise the importance of controlling resources to obtain sufficiently high occupancy on the compute units, in order to hide latency on global memory interaction. This is also of importance when optimizing kernel code for the HD 6970 VLIW4 architecture.

9.1 Resource Allocation

The GDDR5 memory connected to the GPU chip has a very high bandwidth, but this high bandwidth is traded in for long latency access time. The GPU works around this trough massively multi threading. With many wavefronts residing on a SIMD, chances are that there is always at least one wavefront which is ready for execution. For this to work properly, a lot of wavefronts need to be residing within the SIMD simultaneously.

On the HD6970, a maximum of 32 wavefronts can reside simultaneously on the SIMD, with 64 work items per wavefront this equals 2048 work items. Also, a work-group can contain no more than 256 work-items. One can compare this number to that of the Fermi architecture, on the Fermi, a maximum of 1024 work-items per work-group is allowed. Say that a developer had hard coded a kernel to be executed with a 32x32 work-group, then the OpenCL application tuned for Fermi would not even run on the HD6970!

In actual applications, obtaining 32 simultaneously residing wavefronts is not likely, the reason is that each work item needs a certain amount of registers and possible local memory, and this is a limited resource. On the HD6970 there is 256KB of registers on each SIMD unit, these are available as 256 128bit vector registers for each work item across a 64 element wavefront. In addition to the registers, 32KB of local memory is available on each SIMD unit accessible as a random access 32 bank SRAM, with two ports per work item for read and write. Both the registers and the local memory is divided among the work groups executing on the SIMD unit, and occupancy of the compute unit will be reduced on per work-group basis. Whether it is the registers or local memory that show to be a resource bottleneck is naturally algorithm specific.

9.2 Compute Issues with VLIW

The use of VLIW architecture has been with AMD since the HD2000 launched in 2007. The reason for using these designs is, that the by far most common graphics operation is a 4 component dot product + a scalar operation, which fits perfectly intro a VLIW5 SPU, so when performing these common graphics operations, the VLIW5 based GPU is extremely fast. However, with the introduction of DirectX 10, the arithmetic workload handled by GPUs began to change. GPUs where now also utilised for other tasks such as in-game physics. The distinction between what is graphics and what is compute, was and still is, slowly disappearing and the number of compute tasks for the GPU to perform is on the rise. This development gradually reduced the efficiency of the VLIW5 architecture and the first step from AMD to handle this challenge was with Cayman (HD6970) moving to VLIW4 from VLIW5.

In terms of doing compute work, the VLIW4 architecture has its fair share of issues. With the VLIW there is no dynamic scheduling during execution. With graphics, it is rather easy to efficiently compile and schedule shaders under those circumstances, but with compute, this is not always the case. There is a wider range of things going on, and it is difficult to figure out what instructions will play out nicely with each other. Another issue is the complexity of a VLIW instruction set, which is a real pain when optimizing and hand tuning a program. Again, this is not normally a problem for graphics, but is often is for compute. The very complex nature of VLIW makes it hard to disassemble and to debug, and this makes it difficult to predict performance and fix performance critical sections of the code. Ideally a coder should never have to work in assembly, but for HPC and other uses there is a good deal of performance to be gained by doing so and optimizing down to the single instruction.

The fundamental issue with the VLIW design is that it is great for graphics, but not so great for computing. As compute work is becoming an essential task for GPUs even within computer games, the VLIW4 architecture simply will not do for future generations of GPUs. AMD was in need of a new architecture to base it future products on, and this new architecture is what the next section is about.

10 Graphics Core Next

Graphics Core Next is the name of the brand new AMD GPU architecture, that is to form the basis of a GPU which is as capable at compute as it is at graphics. AMD is moving away from the VLIW architecture, and towards something that is more familiar to what Nvidia is offering with its line of GPUs.

Whereas on the AMD developers webpage on can find a full reference guide to the HD 6900 series Instruction Set Architecture, information regarding the compute capabilities of the GCN architecture is still somewhat limited, as the first product based on GCN architecture, the HD7970, was only just released 2 weeks prior to finishing this report.

The architecture and compute capabilities presented within this section are based mostly on information condensed from AMD slide shows, as well as information presented in articles on anandtech.com and tomshardware.com[27]. A nice feature about this new architecture is how AMD has adopted some name conventions of the OpenCL execution model so that mapping from OpenCL terminology to hardware comes naturally.

10.1 A Non-VLIW SIMD Design

AMD is replacing the VLIW architecture with a traditional SIMD vector processor. Whereas the SIMD with VLIW lanes before constituted a core in itself, this is no longer the case, and so elements of Cayman does not directly map to elements of GCN. The SIMD on the GCN is a true 16-wide vector SIMD, not to be confused with the SIMD on Cayman which is a collection of VLIW SPs. On the GCN SIMD, a single instruction and up to 16 data elements are fed to a vector SIMD to be processed over a single clock cycle. Wavefronts still consists of 64 work items, meaning that it takes 4 cycle to complete a single instruction for an entire wavefront. A SIMD has access to a 64KB register file.

As with the VLIW SPs in Cayman, the SIMD is capable of performing a number of different integer and floating point operations, details about this is still to be released. One thing that is certain though, is that double precision floating point performance of the ALUs improved radically and the GCN architecture is to support double precissions performance at up to 1/2 the performance of single precission. The VLIW4 suported only 1/4.

10.2 Dissecting a Compute Unit

A compute unit in the GCN architecture consists of four of the SIMDs as described in the previous section. With four SIMDs in a Compute Unit, the GCN compute unit can work on four instructions at once just as the Cayman SIMD could. In the GCN, control hardware responsible for fetching, decoding and scheduling wavefronts is now moved down to the Compute Unit itself. In addition to the four SIMDs, a scalar unit is also added, the purpose of which is to be explained later. The schematics of the compute unit are shown in figure 10.1. In addition to the 64KB registers per SIMD, a 64KB local data scratch pad memory

for the SIMDs to share is added. Within each GCN SIMD a maximum of 10 wavefronts can reside, with 4 SIMDs per compute unit this adds up to a maximum of 40 wavefronts residing within one compute unit at any given time.

As apparent from the schematics, the compute unit of the GCN architecture is entirely different than the compute unit of the VLIW4 architecture. Let us stop for a moment and consider the differences.

One of the weakness of the old VLIW architecture, is that instructions are statically scheduled ahead of time by the compiler and as a result any non-interdependencies that appear as code is being executed, cannot be exploited since there is no deviation from scheduled instructions so VLIW slots may can go unused. In the CGN architecture, scheduling is moved from compiler to hardware, and the compute unit is now in charge of scheduling execution within its domain. Moving the scheduler to the chip is a trade-of, since this chip space can no longer be filled with additional functional units, but moving scheduling into hardware allows for a more dynamic scheduling which is an advantage for compute work.

Another difference is the addition of a scalar unit per CU to support the four SIMDs, it serves to further keep inefficient operations out of the SIMDs, leaving the vector ALUs on the SIMDs to execute instructions. The scalar unit is composed of a single scalar ALU along with 8KB registers. The scalar unit handles everything from single integer operations to control flow operations, like conditional branches and jumps. Besides avoiding feeding SIMDs non-vectorized datasets, this will also, along with the closer scheduler, improve the latency for control flow operations, where the VLIW4 architecture on Cayman had a nasty 40+ cycle latency that requires many wavefronts residing in the SIMDs to efficiently hide.

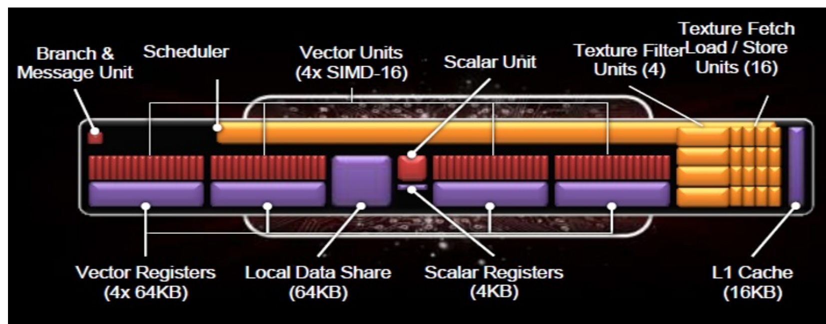


Figure 10.1: [27] The Compute Unit of the AMD GCN achitecture. The compute unit is the fundamental unit of computation in the GCN architecture, and it is the basic building block for making GPUs in the sense that any GPU in the series will implement a number of these compute units in multiples of 4.

10.3 New Features with GCN

On GCN based devices, the global memory will be cached, and the architecture will support 64KB or 128KB of L2 cache per memory controller. The L2 cache is write-back and will be fully coherent, so that all compute units see the same data. The CGN architecture also adds proper ECC suport, though this will not be implemented on the HD7970.

To add to the compute capabilities of the architecture, AMD employ a new Asynchronous Compute Engines to serve as the command processor for compute operations on GCN. The purpose of the ACE will be to accept work and to dispatch it off to the compute units for processing. The GCN can this way work on several different task, with the ACE handling resource allocation, context switching and task priority. In addition, GCN allows for dual

DMA engine so to fully take advantage of the large amount of bandwidth PCIe 3.0 offers between the GPU and the CPU.

A major move towards GPGPU is how GCN implements the underlying features necessary to support C++ and other advanced languages. GCN adds support for pointers, virtual functions, exception support and even recursion all so to eventually move towards being able to write GPU code from a higher level then C, and thus to make it easier to program code that employ GPUs as compute accelerators. These additions to GCN fit very well with the announcement about the directions and content of OpenCL 2.0 described in [11]. Also, the hardware is being adapted towards support of an ISA that uses unified memory so that programs can reference memory anywhere, without the need to explicitly copy memory from one device to another before working on it. This is to be abstracted away from developers and handled by the compiler. The GCN architecture also comes with hardware changes that are made so to improve debug support by allowing debugging tools to tap the GPU for information at more stages than was possible with the old VLIW5/4 architectures.

10.4 The AMD HD 7970

The AMD HD7970 GPU contains 32 of the compute units described in the previous section. The schematics of the GPU is shown in figure 10.2. The HD7970 features 6, 64-bit GDDR5 memory controllers each coupled with 128KB L2 cache for a combined bus-width of 384 bit and 768KB global memory cache. Work-groups are still executed in batches of wavefronts of 64 work-items. 10 wavefronts can reside within each of the four SIMD units in a compute unit. With 32 compute units and up to 40 residing wavefronts per compute unit, 2560 work-items can simultaneously reside on the HD7970.

Figure 10.2: [27] Schematics of the GCN based AMD HD7970 GPU.

Part IV

Performance Tuning with OpenCL

Within this part of the report, the performance of various matrix matrix multiplication kernels implemented in OpenCL on Tesla, Fermi, VLIW4 architectures are investigated. No GCN architecture based cards were available in the GPUlab at the time of writing this report. The specifications and compute capabilities of the GPUs mentioned are listed in table 10.1. GTX280 is a single chip GPU while both GTX590 and HD6990 are dual chip. All performance testing was done on one chip only, so specifications are listed on a per chip basis.

Initially, a range of different matrix matrix multiplication kernels are presented and explained, from naive implementations to more complex implementations utilizing scratch pad memory, register space, granularity considerations, minimized communication patterns and optimized reads and writes to local and global memory spaces. All performance optimization considerations during the development of the kernels are made with respect to the VLIW4 architecture. The performance of the kernel code is measured as a function of work-group size on both Tesla, Fermi and VLIW4 architectures so to identify optimal work-group size conditions. Following these initial tests, kernel performance is measured as a functions of matrix size. When applicable, the equivalent CUDA code is also tested for comparison between compiler behaviour. Following these measurements on hardware in the GPUlab the performance portability across architectures of the code written is discussed.

Name	GTX280	GTX590	HD6990	HD7970
Chip	GT200	2 x GF110	2 x Cayman	Tahiti
Architecture	Tesla	Fermi	VLIW4	GCN
TDP	236Watts	365Watts	375Watts	230Watts
Processor clock	602Mhz	607Mhz	830Mhz	925Mz
Compute Units	30	16	24	32
FP Performance, SP	938GF	1244GF	2548GF	3789GF
FP Performance, DP	78GF	311GF	637GF	947GF
Memory Clock Frequency	1107Mhz	1707Mhz	2500Mhz	2750Mhz
Std. Memory Configuration	1024MB	1536MB	1536MB	3GB
Memory Interface Width	512-bit	384-bit	256-bit	384-bit
Global Memory Bandwidth	141.7GB/s	163.9GB/s	160GB/s	264GB/s
Global Memory Cache	NA	768KB L2	NA	768KB L2
Local Memory Banks	16	32	32	32
Local Memory pr CU	16KB	48KB	32KB	64KB
Register Space (Private) pr CU	64KB	128KB	256KB	256KB
Max Work-Groups pr CU	8	8	8	?*
Max Work-Items pr Work-Group	512	1024	256	?*
Max Work-Items pr CU	1024	1536	2048	2560
Max Warps/Wavefronts pr CU	32	48	32	40
Work-items in Warp/Wavefront	32	32	64	64

Table 10.1: GPU specifications and compute capabilities with regards to the OpenCL memory and execution model. All numbers are specified on a per chip basis, except the TDP. (*) numbers could not be confirmed at time of writing.

11 Matrix Matrix Multiplication

11.1 A Naive Kernel

Matrix matrix multiplication is commonly used to demonstrate optimization techniques due to the fact that, given enough tuning effort, matrix-matrix multiplication can run near the peak performance of most modern processors. Below is the OpenCL C kernel code that was used when measuring performance stated

```
__kernel void MatMulNaive(__global float *A, __global float *B, __global float
    *C, int dim) {
  // Retrieve work item global index
    int i = get_global_id(0);int j = get_global_id(1);
    // If work item within matrix dimension, do do product
    if((i<dim) && (j<dim)){

    float tmp = 0.0;int k;
    for (k=0;k<dim;k++){
        tmp = tmp + A[j*dim+k]*B[k*dim+i];
  }
    C[j*dim+i] = tmp;
    }
}
```

Comparing to a normal CPU sequential implementation, the two outer loops that iterate over all elements in the output matrix are replaced with work-item function calls that identify the global work item ID in each of the two dimensions. Each work item calculate the dot product that constitutes a single element in the output matrix.

11.2 Exploiting Local Memory

Getting good performance from a GPU involves optimizing memory access and transfer. With the current construction of the OpenCL-C matrix matrix multiplication kernel, each work-item need to do roughly one load from global memory per floating point operation, thus, we have a compute to global memory access (CGMA) ratio of 1. The CGMA ratio is highly relevant. Consider for a moment the HD6990. It boasts a massive 2548Gflops single precision performance, which in itself is very impressive. But how many floats can actually be transferred to the chip from global memory? Under ideal circumstances(160GB/s), the chip can be fed 40 Giga floats per second. From this information it can be deduced that when running the OpenCL C kernel presented in the previous section, the hundreds of ALUs present on the chip will be sitting idle most of the time throughout the computation. It is thus clear that the code is heavily constricted by the speed at which memory can be transferred to and from chip to global memory, so by the afore mentioned reasons, increasing the CGMA ratio should be a priority, and this leads us to the use of local memory.

For the purpose of decreasing the amount of loads and stores to global memory throughout a program, GPUs incorporate a highly efficient, high speed, low latency, close to compute, scratch pad memory that is fully under the programmers control. This scratch pad memory is accessed in OpenCL C trough the `__local` qualifier. This form of programmable local data share is more efficient area/power wise than the classic hardware controlled cache employed in CPUs and it is also less wasteful since hardware controlled cache often load data that is never used so less dedicated memory is needed. However, the use of such programmable

close to compute memory space add to complexity of programming, since memory transfer from one level to another now needs to be handled explicitly.

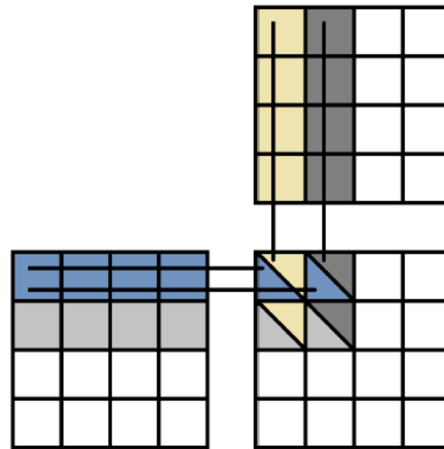


Figure 11.1: Matrix matrix multiplication, highlighting the calculation of a 2x2 tile.

The above considerations leads us to the conclusion that we need a way of reducing the number of loads to the global memory, and to do so we will exploit these local memory structures. Consider a matrix matrix multiplication as the one shown in figure 11.1, one readily notice how the work-items within the highlighted 2x2 tile share common loads. As explained in section 4.2.2, OpenCL-C, work-items are executed concurrently in work-groups, and a work-group is executed on a compute unit with which the local memory is associated. If one apply the pattern as shown in figure 11.1 to to the work-group so that the work items corporately load a 2x2 tiles, one could potentially reduce the traffic to global memory from 16 reads and 4 writes to 8 reads and 4 writes. Using a 16x16 work-group to calculate a 16x16 tile, one can reduce the traffic from $2 \cdot 16 \cdot 16 \cdot 16$ reads and $16 \cdot 16$ writes to $2 \cdot 16 \cdot 16$ reads and $16 \cdot 16$ writes. This would reduce global memory loads and stores needed to 1/11. In the kernel code exploiting local memory we take care to read memory in a coalesced fashion from global memory as well as avoid bank conflicts when reading and writing to local memory. These issues are discussed further in the sections below.

11.2.1 Coalesced Global Memory Access

As previously mentioned, the matrix matrix operation being performed on the kernel is heavily memory bound, and it is thus of priority to optimize how fast memory is transferred to the GPU. It is therefore of importance to consider memory coalescing techniques that can more effectively move data from the global memory into local memory and registers. Many sensors are provided in each DRAM chip, and they work in parallel, each sensing the content of a byte within these consecutive locations. Whenever data is accessed in global memory, many consecutive locations that include the requested location are accessed and transferred, so if one can make use of data from multiple consecutive locations, less data needs to be transferred to deliver the loads requested. On the contrary, if multiple data elements from non-consecutive locations are requested, more data needs to be transferred, much of which is not needed, which results in waste of the precious bandwidth.

On AMD GPUs, work groups are executed on compute units concurrently in chunks of 64 work items, such a collection of work items is referred to as a wavefront. The instructions in

the wavefront are executed on 16-wide SIMD over four cycles. When the work items within a wavefront executes a load instruction, the hardware detects whether the work-items access consecutive global memory locations. If so, less memory needs to be transferred, and the available bandwidth is thus used more effective, these considerations are the same as when programming kernels to the Fermi or Tesla architectures whether in CUDA or OpenCL.

11.2.2 Avoiding Bank Conflicts in Local Memory

So far we have assumed that there is no need to worry about how to read and write to local memory. Even though local memory is very fast, this is not entirely true. The data stored within local memory is located in what is known as banks. When writing data to a one dimensional array in the local memory, the first data element will receive the first address in the first bank, the second data element receives the first address in the second bank and so on. This continues until sixteen data elements are stored in the sixteen available banks. Now, when data element seventeen is stored, it is stored in the second address in the first bank and data element eighteen is stored in the second address in the second bank and so the pattern continues.

Why does one need to be aware of this? Well, each bank can read and write to its own data in parallel with all other banks, so technically it is possible to read sixteen data elements at once given that these sixteen elements are distributed among different banks. To understand how this can have an impact on performance, we again need the concept of wavefronts. In a SIMD core on the VLIW4 architecture, all 16 work-items in a sub-wavefront are executed simultaneously. If all these work items simultaneously try to read from the local memory in different addresses all associated with a single bank, this bank can only serve one work-item at a time, and the work items will have to take turn receiving data from the local memory, which serializes the execution and this is bad for performance. Such a situation is referred to as a bank conflict. However, if all work-items are to access the same address in a given bank, the bank will "broadcast" and serve all at once. Ideally, one would like that each work-item in the sub-wavefront access data elements from different banks so that all work-items can be served in parallel. In the local memory matrix matrix multiplication code previously stated, care was taken so that bank conflicts would not arise.

11.2.3 Performance Impact of Work-Group size

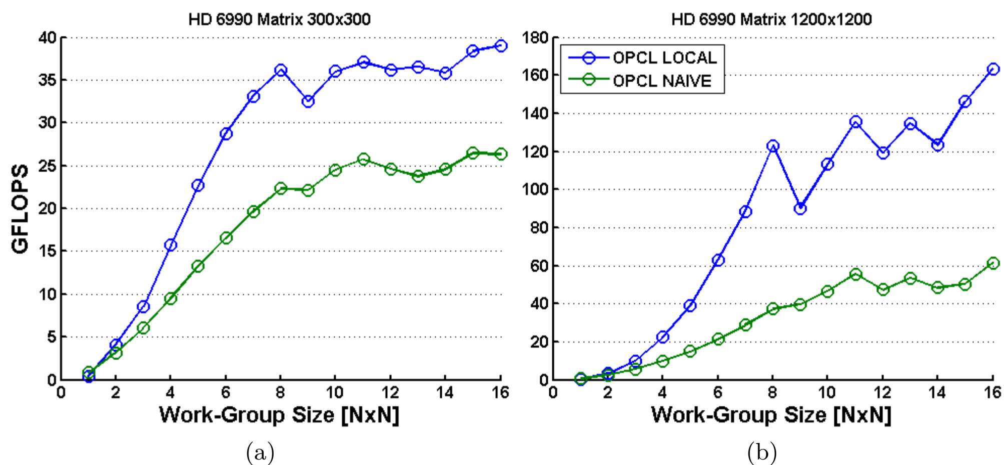
The performance of the two OpenCL-C kernels presented in the previous section where measured as a function of work-group size, this for multiplication of two different sizes of square matrices 300x300 and 1200x1200. Results are presented and discussed within this section. All timings for performance measurements include time of transfer to and from host and device. Throughout the section, the notation LOCAL and NAIVE is used for the two kernels respectively.

In figure 11.2, measurements of the kernels running on one of the cores of the HD6990 GPU are presented. As expected, the performance of the Naive kernel is a long way from what one might hope specification wise. Despite not tilling the matrix, the performance increase with work-group size, but how can this be? As previously mentioned, the application is heavily memory bound. Work-items are executed in subwavefronts within a SIMD core, and if work-items within a sub-wavefront issue reads to consecutive memory locations, the hardware detects this, and memory is transferred coalesced. The HD6990 has a 256bit wide global memory bus, so it can transfer 8 floats at a time, the performance peaks at 8x8 and 16x16 work-group sizes as apparent.

Another issue with small work-groups less that 8x8 items is how work is divided. A maximum of 8 work-groups can be residing at any one time in the SIMD core and each work-group is divided into wavefronts, a collections of 64 work-items which is executed concurrently over 4 cycles on the SIMD core. When the work-groups contain less than 8x8 (64) work-items, the wavefront scheduled for the work-group is only partially filled with actual compute work, and so one wavefront will be issued per work-group regardless that it is not full. With a work-group containing, say, 4x4 work item, only 16 work items would be computed per wavefront, substantially less than the 64 possible and performance surfers as an effect. For these reasons, the AMD HD69xx series OpenCL reference guide [20] recommends to always use work-groups containing a multiple of 64 work-items. On Fermi and Tesla architectures, work-groups are scheduled in chunks of 32 items, and as such, it is recommended to only use work-groups containing a multiple of 32 work-items on these architectures.

The performance of the kernel utilizing local memory is as expected substantially better. As work-group and tile size are kept equal, the amount of global memory reads needed decrease as work-group size increase. As memory becomes less of a wall, other effects such as the scheduling mentioned previously become more apparent, such as the peaks at 8x8 and 16x16 where wavefronts are fully utilised and full bus width memory lines are read coalesced.

Matrix multiplication between matrices of size 1200x1200 elements also achieve substantially higher performance than do the matrix multiplication between 300x300 element matrices. This is due to the fact that the overheads of setting up the computation remain fairly constant, and the data which needs to be transferred increase with size^2, while the computations needed increase with size^3, so for when multiplying larger matrices, more time is spend on computing and less on transferring data, which lead to larger performance.



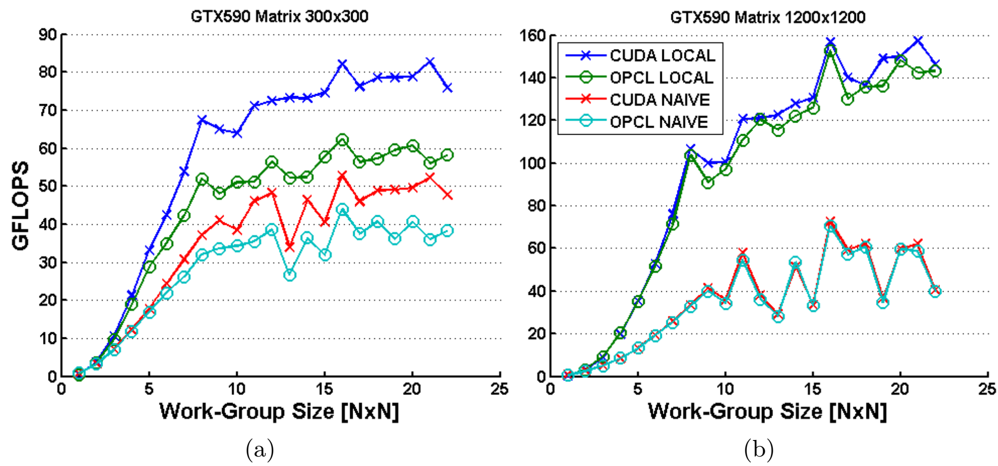
(a) (b)

Figure 11.2: Performance as a function of work-group size on the AMD HD 6990.

Both OpenCL C kernels were also tested on the available Tesla and Fermi architectures, the results where compared to the performance binaries compiled from the equivalent C for CUDA code and the figures 11.3 and 11.4 present the measured performance. Similar effects are observed, and performance peaks at 8x8 and 16x16 work-group sizes. These contain 64 and 256 work item elements, which are both multiples of 32 and 64, so using such work-group sizes fits both architectures well.

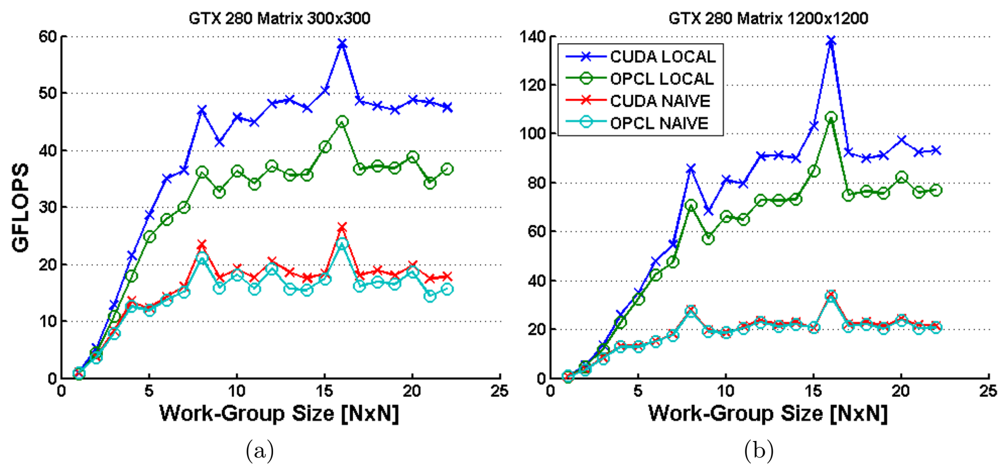
It seems that there are differences between the performance of the OpenCL C code and the equivalent CUDA code, but it looks as if the effects disappear for larger matrices. Multiplying

two 1200x1200 large matrices on the GTX590, the performance obtained for a given work-group size is the same whether using OpenCL or CUDA. It would be interesting to see what the difference is, but this would require looking into the intermediate PTX code, and this is beyond the scope of this report.



(a) (b)

Figure 11.3: Performance as a function of work-group size on the Nvidia GTX590



(a) (b)

Figure 11.4: Performance as a function of work-group size on the Nvidia GTX280

11.3 Increasing Performance

At this point, impressive matrix matrix multiplication performance was achieved on the HD6990 without too much effort. However, there is still room for further improvements. In the following sections, we seek to build an even faster kernel code, getting closer to the theoretical peak performance of the card at hand.

11.3.1 Minimizing Communication

In the local memory matrix matrix multiplication kernel described in section 3.3, data from the matrices A and B is first moved into local memory, and then later moved from local memory into registers so to be used in computations. There is a potential performance increase waiting if one could somehow avoid some of this, while still keeping the reduction of global memory access broad about by tilling.

The solution is to increase the work in each work-item. Now using (16,1) work-groups to compute a 16x16 tile of the C matrix. A work-group copies a 16x16 tile of A into local memory, and read a 1x16 tile of B into registers as needed when computing the dot product. Doing so correctly, we still obtain coalesced reads when reading from global memory to the registers, as the sixteen work-items in a sub-wavefront simultaneously read elements from consecutive locations.

11.3.2 Increasing Occupancy

The trick implemented in the previous section is clever and reduce the amount of communication between global memory, local memory and registers. But at what cost did it do so?

One of the powerful features of GPU cards is the super high bandwidth global memory. This high bandwidth memory unfortunately does not come without a downside to it. The high bandwidth memory is cursed with a very long latency. GPUs cope with this burden trough massive multi threading. When an instruction executed by the work-item in a wavefront is waiting for the results of a previously initiated long latency operation, the wavefront is not selected for execution, another wavefront residing in the SIMD core that is no longer waiting for data is selected for execution instead. Without latency hiding, it is not possible to get anywhere near the peak specified bandwidth in practical applications, so filling each SIMD core with many wavefronts is crucial for good performance. In the case of the kernel code in the previous section, each work-group holds 16 work-items. The maximum number of work-groups in the Cayman per SIMD core is 8, so only $8 \cdot 16 = 128$ work items are residing in the SIMD core. Under ideal circumstances 2048 work items can simultaneously reside in the SIMD core.

So how can we increase the number of work-items simultaneously residing in the SIMD core, while still reduce redundant commutation between memory spaces? The solution is to modify the kernel code from the previous section so to use a (64,1) work-group that calculates a 16x64 tile of the C matrix. This is easier said than done since this kernel use 64 work-items in each work-group to load the 16x16 A tile. The problem is solved with a little bit of index arithmetics though. Besides the change of how to load and store the tile of the A matrix, compared to the previous kernel, only minor changes are needed. With the kernel code as explained, eight work-groups, each containing 64 work items are residing within the SIMD core for a total of 512 work-items. For all the work-groups to reside in the SIMD core, enough scratch pad memory and register space must be available. $16 \cdot 16 = 256$ floats are loaded into local memory in each work-group, that is 1KB of local memory is needed per work-group, with 8 work-groups, 8KB is needed. On the Cayman chip, each SIMD core has 32KB of LDS memory available, so local memory space will not restrict occupancy. 256KB register space is available on each SIMD core, this is more than enough for the kernels written. In addition to the kernel code as presented, another version where the inner loop has been unrolled is also tested in the sections to come.

12 Device Performance Measurements

The OpenCL C kernels presented in the previous sections are tested in terms of performance as a function of matrix size, using a work-group size of 16 by 16 work items for the **NAIVE** and **LOCAL** kernels and as otherwise specified. The following conventions used

- **NAIVE** The naive kernel implementation, section 11.1
- **LOCAL** Using local memory facilities, section 11.2
- **FAST1** Reducing communication, section 11.3.1
- **FAST2** Increasing occupancy, section 11.3.2
- **FAST3** Unrolling loops, section 11.3.2

12.1 AMD VLIW4 - HD 6990

Performance measurements are depicted in figure 12.1. The results are mostly as expected. **FAST1** takes a massive performance hit, and this will be further commented upon in the portability section. Unrolling loops makes the code slightly slower, maybe it prevents the compiler of doing something better. All in all the HD 6990 delivers an impressive throughput, at a matrix multiplication between matrices of size 3200x3200, it reaches 450Gflops.

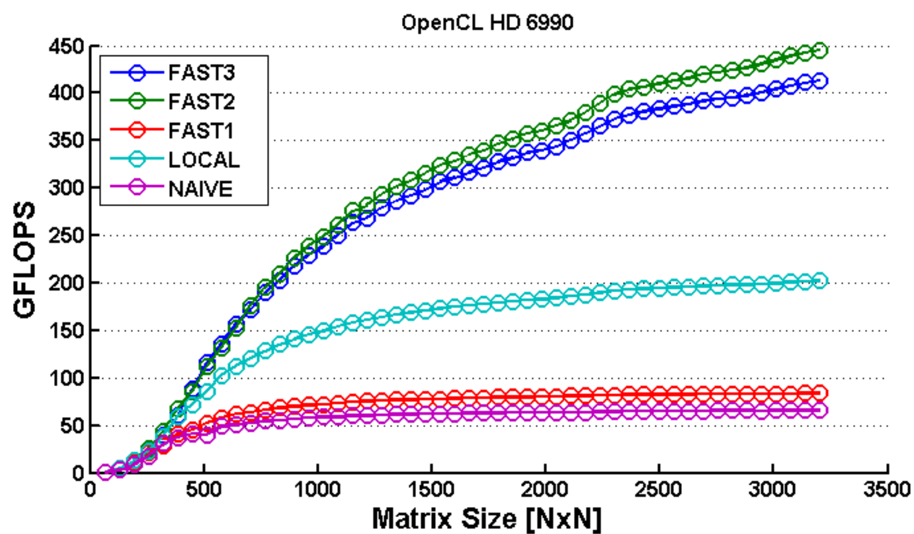


Figure 12.1: Performance as a function of matrix size measured on the AMD HD 6990.

12.2 Nvidia Fermi - GTX590

Performance measurements of the kernel code written in OpenCL is depicted in figure 12.2 and the CUDA equivalent code in 12.3. There are many interesting observations to be made. First of, the OpenCL version delivers better performance! Another interesting difference is how unrolling loops in CUDA has a large negative effect while it does not alter the performance whatsoever in OpenCL. Again we notice that **FAST1** is slower, but it seems to take less of a performance hit in OpenCL than in CUDA.

All in all, the more complicated kernels, **FAST1**,**FAST2**,**FAST3** perform somewhat different in OpenCL than in the equivalent CUDA versions, but the simpler kernels **NAIVE** and **LOCAL** yield almost identical performance numbers. It is hard to say why this is so. One way of investigating the reason for the observed effects would be to look at the intermediate PTX code of the kernels.

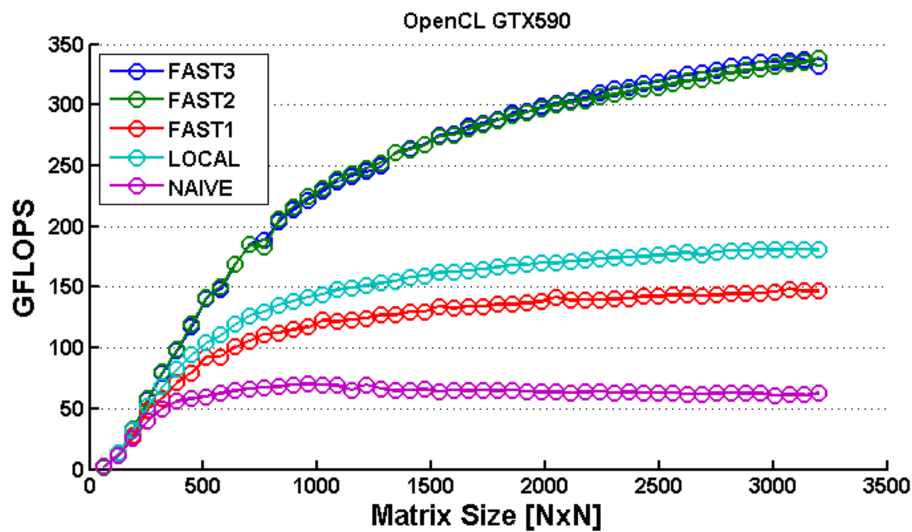


Figure 12.2: OpenCL implementation measured on GTX 590

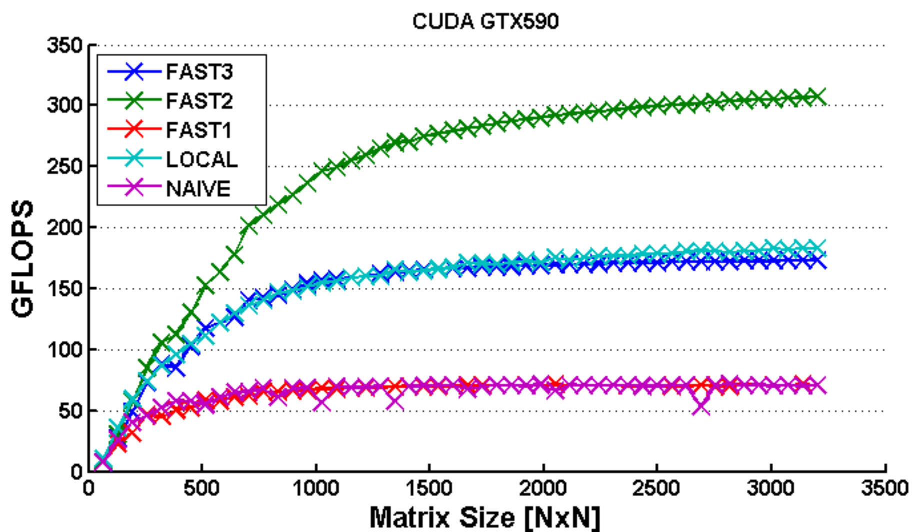


Figure 12.3: CUDA implementation measured on GTX 590

12.3 Nvidia Tesla - GTX280

Figures 12.4 and 12.5 contain the performance measurements on the Tesla architecture based GTX280. In contrast to the HD6990 and GTX590, GTX280 show both in CUDA and OpenCL clearly better performance from the kernel in which the inner loop was unrolled. CUDA is slightly faster on all kernels except again **FAST1** which as expected take a big performance hit, but it is substantially slower on CUDA.

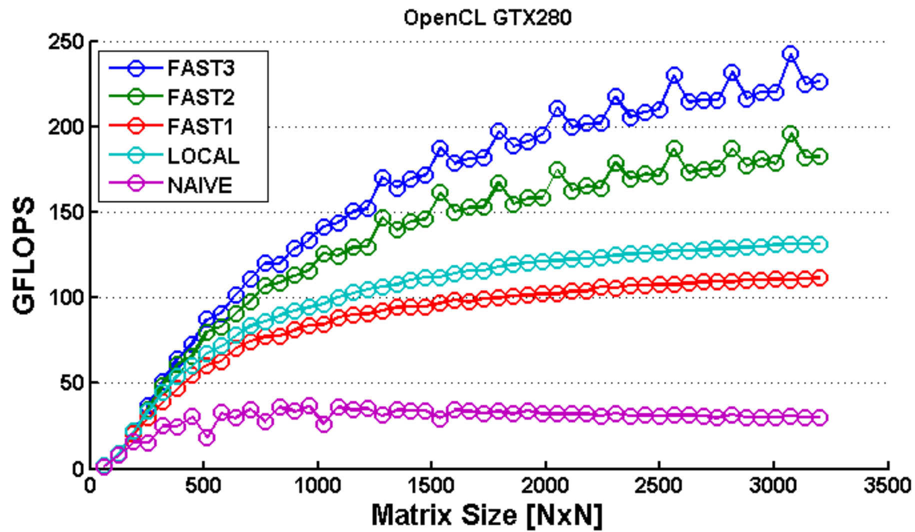


Figure 12.4: OpenCL implementation measured on GTX 280

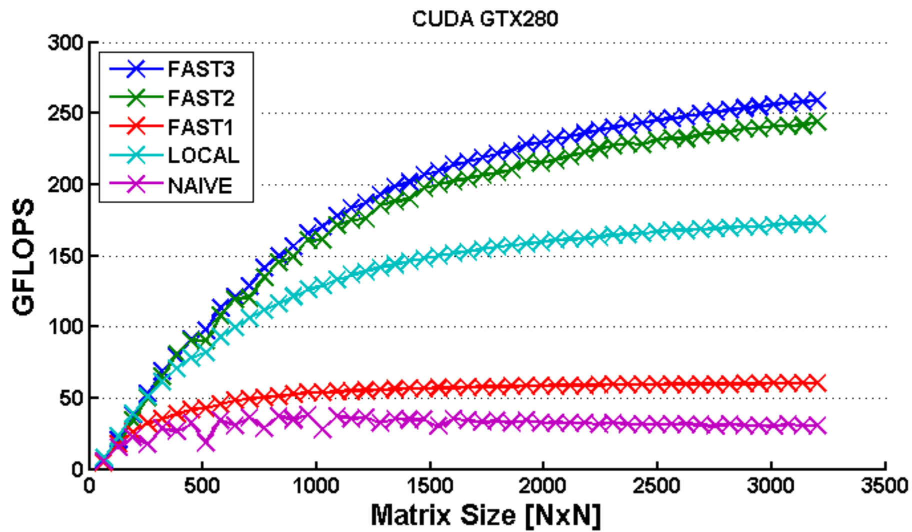


Figure 12.5: CUDA implementation measured on GTX 280

12.4 Device Comparison

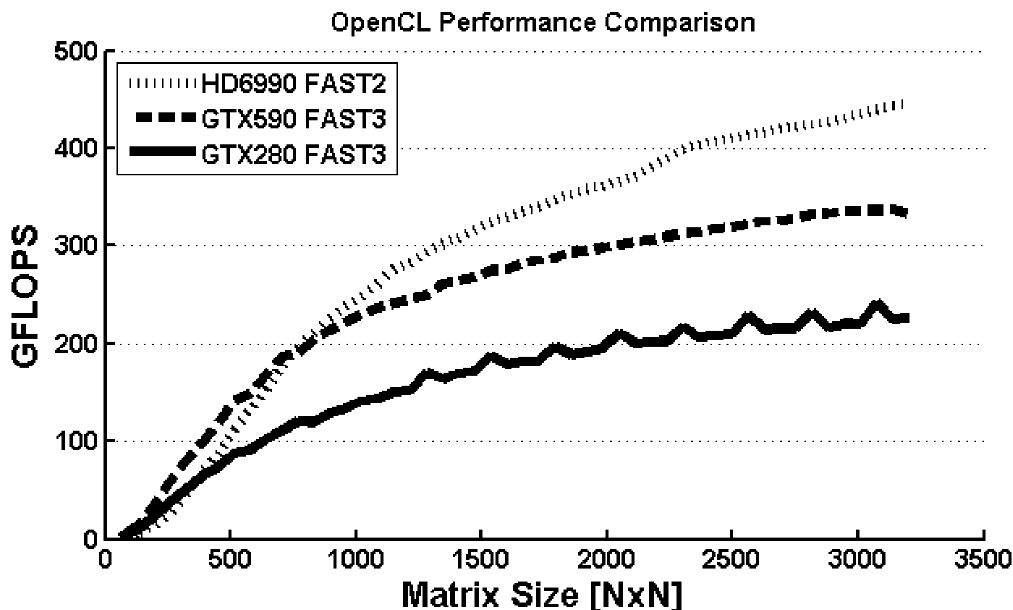


Figure 12.6: Performance of OpenCL implementation as a function of matrix size for the fastest kernel on each GPU card tested.

The performance of the fastest OpenCL C kernel for each architecture is plotted in figure 12.6. The HD 6990 delivers impressive throughput, close to half a teraflop single precision performance using only general optimization techniques and it clearly outperforms the GTX590 GPU. It should be noted however that further optimization within IL or PTX code might change the ratio of performance in between devices.

The kernel code performs very well on the VLIW4 architecture of the HD6990. This is not surprising. It is easy for the compiler to extract explicit vectorization of the OpenCL C kernel **FAST3** so to keep all ALUs filled. In the **LOCAL** kernel, it is not possible to extract independent instructions within each work-item, and so 3 out of 4 ALUs will lie idle most of the time when the code is running on the VLIW4 architecture. The performance of the **LOCAL** kernel on the HD6990 and the GTX590 is roughly the same.

If one where to add branch statements, that would result in branch divergence, to the kernel code, one might see the GTX590 pass the HD6990 in terms of performance. The reason for being that the VLIW4 architecture has a very high 40+ cycle latency in between executing divergent branch clauses in subwavefronts.

It would be interesting to run the kernel performance measurements on the new GCN based HD7970 to see if the performance increase is as massive as promised by the specifications. This possibly with additions of code including branch divergence which should be handled better in the HD7970 due to the closer scheduler.

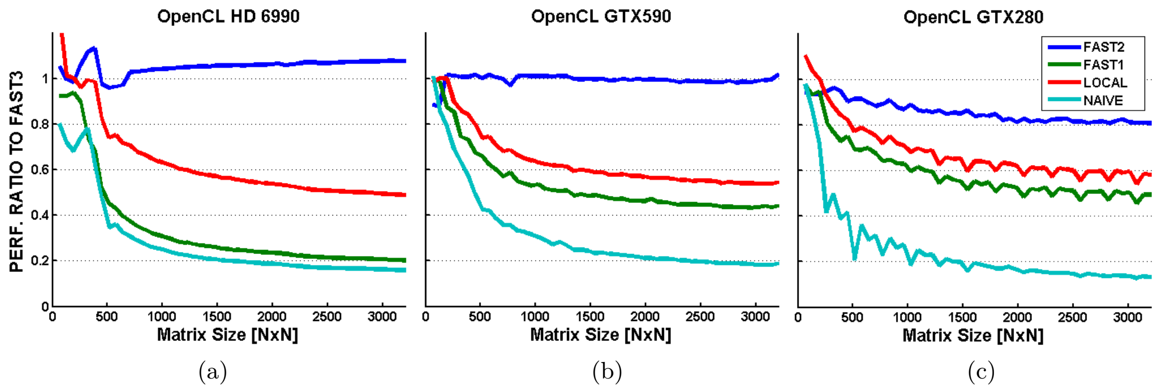
13 Code and Performance Portability

An important thing to consider when writing OpenCL C kernels that has to work across different architectures is how optimizations on one architecture will effect performance on other architectures.

In figure 13.1, speed relative to **FAST3** on each architecture is depicted. For this problem and the kernel implementations, the performance behaviour is very similar on the different architectures. One of the implementations however, **FAST1**, takes a much larger performance hit on the HD6990 than it does on the the GTX280 and the GTX590.

This effect can, in part, be attributed to the variation in vector width. **FAST1** uses work-groups of only 16 work-items each, that is, only 16 work-items are scheduled in a full 64-wide vector wave front, wasting a lot of ALUs. It should also be clear from a pure occupancy point of view, with only 128 work-items residing out of 2048 possible slots, one is asking for trouble. The GTX590 and GTX280 uses a 32-wide vector width, which is half as long as that of the HD6990, so also only half as many cycles are wasted. In addition, the SMP compute units on the Tesla and Fermi architectures can hold 1024 and 1536 work-items respectively, so the relative work-item occupancy is higher on these.

It is important to do these kind of considerations when writing the code. The code written in the previous sections scaled nicely from one architecture to another, but let's say when writing **FAST2** we decided to use a 1x96 work-item work-group. This would likely work out nicely on the Tesla or Fermi architectures, each work group having three full warps, for a total of 24 warps residing on a SMP for a total of 768 work items. What would happen on the VLIW4 or GCN architectures? Well, the 96 work items would be divided into two wavefronts of 64 work items, one wave front only half filled, so 1/4 of all possible ALU instructions are wasted, and the compute unit has to execute 1/3 as many work items equivalent. This would almost surely lead to a big performance penalty, and this is the sort of considerations one need to make when porting code from one architecture to another.



(a) (b) (c)

Figure 13.1: Performance relative to **FAST3** implementation on each architecture.

Part V
Summary and Outlook

14 Conclusion

Within recent years, the otherwise ever increasing supply of computational throughput has come under pressure due to the fact that the speed of single core general purpose CPUs can no longer be scaled due to the constraints imposed by the very laws of physics. The demand for more computational throughput is still there though, and so a new approach of increasing throughput is needed. The industry has decided that this approach is to be heterogeneous computing.

The move towards heterogeneous computing is widely supported, but the language and manor in which these systems are to be programmed has yet to be defined. The move towards heterogeneity on hardware level requires new programming models and paradigms. OpenCL, as presented in this report, is the first industry wide language to cope with heterogeneity and poses a solution to the challenges faced. OpenCL constitutes an entire framework for task and data parallel heterogeneous computing. OpenCL has been criticized for being hard to use with a complicated platform and memory model, but these issues are being handled. The Khronos OpenCL Group is currently working on OpenCL 2.0, which among other things aim to implement support for a unified memory space as well as other initiatives that are to move abstractions to compile level.

AMDs new Graphics-Core-Next architecture has been presented in the report using the information available at the time of writing. The GCN architecture is to form the basis of AMDs new Southern Island series of GPUs that are to be as capable at compute as they are at graphics. AMD offer very powerful and cost efficient hardware, and this along with the portability are strong motivators for using OpenCL when building applications that are to harness the compute power of GPU accelerators.

A References

[1] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fun, Dan Ginsburg. *OpenCL Programming Guide.* ISBN: 978-0-321-74964-2. 2011 Addison-Weslet, Pearson Education.

[2] Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, Dana Schaa. *Heterogeneous Computing with OpenCL.* ISBN: 978-0-12-387766-6. 2011 Morgan Kaufmann.

[3] John Shalf, NERSC Division, Lawrence Berkeley National Laboratory. *The New Landscape of Parallel Computer Architecture.* SciDAC 2007. Journal of Physics: Conference Series 78 (2007) 01206.

[4] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik and Olaf O. Storaasli. *State-of-the-art in heterogeneous computing.* Scientific Programming 18 (2010) 133.

[5] IDC Executive Brief. *Heterogeneous Computing: A New Paradigm for the Exascale Era.* Adapted from IDC HPC End-User Study of Processor and Accelerator Trends in Technical Computing by Earl C. Joseph and Steve Conway, IDC 228098, November 2011 IDC 1208

[6] **Trends within high performance computing**
http://top500.org/blog/2009/05/20/top_trends_high_performance_computing
http://hpcwire.com/hpcwire/2011-11-09/arm_yourselves_for_exascale,_part_1.html
http://hpcwire.com/hpcwire/2011-11-09/arm_yourselves_for_exascale,_part_2.html
http://hpcwire.com/hpcwire/2011-12-08/revisiting_supercomputer_architectures.html

[7] **Accelerators for High Performance Computing**
http://hpcwire.com/hpcwire/2011-11-14/texas_instruments_enters_hpc_market_with_multicore_c
http://hpcwire.com/hpcwire/2011-11-07/pico_computing_unveils_accelerator_board_with_new_x

[8] **Intel MIC Development**
http://intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html

[9] **AMD Fusion Summit June 2011, Graphics Core Next Slides and Key Notes**
http://developer.amd.com/afds/assets/presentations/2620_final.pdf
http://developer.amd.com/afds/pages/keynote.aspx

[10] **AMD Accelerated Processing Units**
http://developer.amd.com/assets/apu101.pdf
http://hpcwire.com/hpcwire/2011-11-02/first_hpc_cluster_with_amd_fusion_chips_debuts_at_sa
http://hpcwire.com/hpcwire/2011-11-02/sandia_installs_hpc_cluster_with_amd_fusion_chips.htm

[11] **OpenCL 2.0 by Tim Mattson, Intel, SC11, Seattle, November 2011**
http://www.youtube.com/watch?v=Pqdq2SvNZP4

[12] **OpenCL code samples from Nvidia and AMD.**
http://developer.amd.com/sdks/AMDAPPSDK/samples/Pages/default.aspx
http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/OpenCL/website/samples.ht

[13] **Reference guide to Nvidia Parallel Thread Execution**
http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/ptx_isa_2.0.pdf

[14] **Reference guide to AMD Intermediate Language**
http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Intermediate_Language_(IL)_Specifi

[15] **Complete list of OpenCL Conformant Products**
http://khronos.org/conformance/adopters/conformant-products/

[16] **OpenCL 1.1 Quick Reference Card**
http://khronos.org/files/opencl-1-1-quick-reference-card.pdf

[17] **AMD Developer Central. OpenCL Technical Overview Video Series**
http://developer.amd.com/documentation/videos/OpenCLTechnicalOverviewVideoSeries/Pages/defa

[18] **AMD Developer Central. SDK, Tools and Libraries**
http://developer.amd.com/zones/openclzone/pages/toolsandlibraries.aspx

[19] **Nvidia SDK OpenCL Programming Guide**
http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_OpenCL_Progran

[20] **AMD SDK OpenCL Programming Guide**
http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf

[21] **Performance Comparison, CUDA and OpenCL**
http://www.cc.gatech.edu/ vetter/keeneland/tutorial-2012-02-20/13-shoc.pdf
http://arxiv.org/PS_cache/arxiv/pdf/1001/1001.3631v1.pdf
http://www.hpcwire.com/hpcwire/2012-02-28/opencl_gains_ground_on_cuda.html

[22] **Georgia Tech's Keeneland Workshop.**
http://keeneland.gatech.edu/2012-02-20-workshop

[23] **Khronos OpenCL API Registry**
http://www.khronos.org/registry/cl/

[24] **OpenCL 1.1 and 1.2 Specification**
http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf
http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[25] **Various Online Sources for an Introduction to OpenCL**
http://drdobbs.com/parallel/231002854
http://codeproject.com/Articles/92788/Introductory-Tutorial-to-OpenCL
http://macresearch.org/opencl_episode1
http://software.intel.com/en-us/articles/OpenCL-Introduction-Tutorial/

[26] **Review of Cayman and VLIW4 architecture**
http://www.anandtech.com/show/4061

[27] **Review of AMD new architecture, Graphics Core Next**
http://www.opengpu.net/EN/attachments/154_HiPEAC2012_OpenGPU_AMD.pdf
http://anandtech.com/show/4455/
http://anandtech.com/show/5261/

Code Appendix

Choosing Platform

```
int getUserDefPlatform(cl_platform_id** ID, char** Name, int* P) {
  // Get number of platforms
  cl_uint platformCount = 0;
  cl_int err = clGetPlatformIDs(0, NULL, &platformCount);
  if (err != CL_SUCCESS) {
    printf("Failed to get number of platforms available\n");
    return 1;
  }

  // Allocate and get platforms
  *ID = (cl_platform_id*) malloc(platformCount * sizeof(cl_platform_id));
  err = clGetPlatformIDs(platformCount, *ID, NULL);
  if (err != CL_SUCCESS) {
    printf("Failed to get platform ids available\n");
    return 1;
  }

  // Iterate through platforms and display the name of each.
  printf("Platforms available: \n");
  for (int p = 0; p < platformCount; p++) {
    // Get info length
    size_t required;
    err = clGetPlatformInfo(*ID[p], CL_PLATFORM_NAME, 0, NULL, &required);
    if (err != CL_SUCCESS) {
      printf("Failed to get length of info string\n");
      return 1;
    }

    // Get info
    *Name = (char*) malloc(required * sizeof(char));
    err = clGetPlatformInfo(*ID[p], CL_PLATFORM_NAME, required, *Name, NULL);
    if (err != CL_SUCCESS) {
      printf("Failed to get info string\n");
      return 1;
    }

    // Print name
    printf("[%d] \t %s\n",p+1,*Name);

    // Free name
    free(*Name);
  }

  // Ask user for which platform to use
  printf("Choose a platform: ");
  scanf("%d",P);
  *P = *P - 1;

  // Get info length
  size_t required;
  err = clGetPlatformInfo(*ID[*P], CL_PLATFORM_NAME, 0, NULL, &required);
  if (err != CL_SUCCESS) {
    printf("Failed to get length of info string\n");
    return 1;
  }
```

```
  // Get info
  *Name = (char*) malloc(required * sizeof(char));
  err = clGetPlatformInfo(*ID[*P], CL_PLATFORM_NAME, required, *Name, NULL);
  if (err != CL_SUCCESS) {
    printf("Failed to get info string\n");
    return 1;
  }

  return 0;

}
```

Choosing Device

```
int getUserDefDevice(cl_platform_id* platformIds, int p, cl_device_id** ID,
    char** Name, int* D, cl_uint* deviceCount){

  // Get number of devices
  cl_int err;
  //cl_uint deviceCount = *Count;
  err = clGetDeviceIDs(platformIds[p], CL_DEVICE_TYPE_ALL, 0, NULL,
      deviceCount);
  if (err != CL_SUCCESS) {
    printf("Failed to get number of devices in platform\n");
    return 1;
  }

  // Get device ids
  *ID = (cl_device_id*) malloc(*deviceCount * sizeof(cl_device_id));
  err = clGetDeviceIDs(platformIds[p], CL_DEVICE_TYPE_ALL, *deviceCount, *ID,
      NULL);
  if (err != CL_SUCCESS) {
    printf("Failed to get devices ids in platform\n");
    return 1;
  }
  cl_device_id* id = *ID;

  // Iterate through devices and display the name of each.
  printf("\nDevices available on platform %d: \n",p+1);
  for (int d = 0; d < *deviceCount; d++) {
    // Get info length
    size_t required;
    err = clGetDeviceInfo(id[d], CL_DEVICE_NAME, 0, NULL, &required);
    if (err != CL_SUCCESS) {
      printf("Failed to get length of info string\n");
      return 1;
    }

    // Get info
    *Name = (char*) malloc(required * sizeof(char));
    err = clGetDeviceInfo(id[d], CL_DEVICE_NAME, required, *Name, NULL);
    if (err != CL_SUCCESS) {
      printf("Failed to get info string\n");
      return 1;
    }

    // Print name
```

```
    printf("[%d] \t %s\n",d+1,*Name);

    // Free name
    free(*Name);
  }

  // Ask the user to choose a device
  printf("Choose a device: ");
  scanf("%d",D);
  *D = *D - 1;

  // Get info length
   size_t required;
  err = clGetDeviceInfo(id[*D], CL_DEVICE_NAME, 0, NULL, &required);
  if (err != CL_SUCCESS) {
    printf("Failed to get length of info string\n");
    return 1;
  }

  // Get info
  *Name = (char*) malloc(required * sizeof(char));
  err = clGetDeviceInfo(id[*D], CL_DEVICE_NAME, required, *Name, NULL);
  if (err != CL_SUCCESS) {
    printf("Failed to get info string\n");
    return 1;
  }

  return 0;

}
```

Device information

```
int getDeviceInfo( cl_device_id* ID, cl_uint deviceCount){

  cl_int err; size_t size;

  for (int d = 0; d<deviceCount; d++){

    // Get device name and print to screen

    err = clGetDeviceInfo (ID[d],CL_DEVICE_VENDOR,0,NULL,&size);
    char* vendor = (char*) malloc(sizeof(char)*size);
    err = clGetDeviceInfo (ID[d],CL_DEVICE_VENDOR,size,vendor,NULL);

    err = clGetDeviceInfo (ID[d],CL_DEVICE_NAME,0,NULL,&size);
    char* name = (char*) malloc(sizeof(char)*size);
    err = clGetDeviceInfo (ID[d],CL_DEVICE_NAME,size,name,NULL);

    printf("Compute capabilities of %s, %s\n\n",name,vendor);

    err = clGetDeviceInfo (ID[d],CL_DEVICE_VERSION,0,NULL,&size);
    char* version = (char*) malloc(sizeof(char)*size);
    err = clGetDeviceInfo (ID[d],CL_DEVICE_VERSION,size,version,NULL);
    printf("\t OpenCL Device Support \t \t %s\n",version);

    err = clGetDeviceInfo (ID[d],CL_DRIVER_VERSION,0,NULL,&size);
```

```
    char* driver = (char*) malloc(sizeof(char)*size);
    err = clGetDeviceInfo (ID[d],CL_DRIVER_VERSION,size,driver,NULL);
    printf("\t OpenCL Software Driver \t %s\n",driver);

    err = clGetDeviceInfo (ID[d],CL_DEVICE_PROFILE,0,NULL,&size);
    char* profile = (char*) malloc(sizeof(char)*size);
    err = clGetDeviceInfo (ID[d],CL_DEVICE_PROFILE,size,profile,NULL);
    printf("\t OpenCL Profile Supported \t %s\n\n",profile);

    cl_uint maxComputeUnits;
    err = clGetDeviceInfo (ID[d],CL_DEVICE_MAX_COMPUTE_UNITS,sizeof(cl_uint),&
        maxComputeUnits,NULL);
    printf("\t Compute Units Available \t %d\n",maxComputeUnits);

    size_t maxWorkGroupItems;
    err = clGetDeviceInfo (ID[d],CL_DEVICE_MAX_WORK_GROUP_SIZE,sizeof(size_t)
        ,&maxWorkGroupItems,NULL);
    printf("\t Max Work-items per Work-group \t %d\n",maxWorkGroupItems);

    cl_ulong localMem;
    err = clGetDeviceInfo (ID[d],CL_DEVICE_LOCAL_MEM_SIZE,sizeof(cl_ulong),&
        localMem,NULL);
    printf("\t Compute Unit Local Memory \t %dKb\n\n",localMem/1024);

    cl_ulong globalMem;
    err = clGetDeviceInfo (ID[d],CL_DEVICE_GLOBAL_MEM_SIZE,sizeof(cl_ulong),&
        globalMem,NULL);
    printf("\t Global Memory Size \t\t %dMB\n",globalMem/1048576);

    cl_bool ECC;
    err = clGetDeviceInfo (ID[d],CL_DEVICE_ERROR_CORRECTION_SUPPORT,sizeof(
        cl_bool),&ECC,NULL);
    switch (ECC)
    {
    case CL_TRUE:
      printf("\t Error Correction Support \t Yes\n");
      break;
    case CL_FALSE:
      printf("\t Error Correction Support \t No\n");
      break;
    }

    cl_ulong globalCache;
    err = clGetDeviceInfo (ID[d],CL_DEVICE_GLOBAL_MEM_CACHE_SIZE,sizeof(
        cl_ulong),&globalCache,NULL);
    printf("\t Global Memory Cache \t\t %dKB\n",globalCache/1025);

    cl_uint globalCacheLine;
    err = clGetDeviceInfo (ID[d],CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE,sizeof(
        cl_uint),&globalCacheLine,NULL);
    printf("\t Global Memory Cacheline \t %dBytes\n\n",globalCacheLine);

  }

  return 0;

}
```

Building program

```c
int buildProgramFromSources(cl_context context, cl_uint deviceCount,
    cl_device_id* deviceIds, const char* filename, cl_program* program) {
  // Open file
  FILE* file = fopen(filename, "r");

  // Find size and go back to start
  fseek(file, 0, SEEK_END);
  size_t length = ftell(file);
  fseek(file, 0, SEEK_SET);

  // Allocate and read content
  char* source = (char*) malloc(length * sizeof(char));
  fread(source, length, 1, file);

  // Close file
  fclose(file);

  // Create program
  cl_int err;
  *program = clCreateProgramWithSource(context, 1, &source, &length, &err);
  if (err != CL_SUCCESS) {
    printf("Failed to create program in context.\n");
    return 1;
  }

  // Build sources
  err = clBuildProgram(*program, deviceCount, deviceIds, NULL, NULL, NULL);
  if (err != CL_SUCCESS) {
    printf("Failed to build program for devices in platform.\n");
    return 1;
  }

  // Free content
  free(source);

  return 0;
}
```