Technical University of Denmark

Wojciech Pawlak (s091820)

Final Report

A GPU-accelerated Navier-Stokes Solver using OpenCL

Special Course at GPUlab, Scientific Computing Section, DTU
Supervisor: Allan P. Engsig-Karup Ph.D.

August 2, 2012

DTU Informatics
Department of Informatics and Mathematical Modelling

Contents

List of Figures

List of Tables

1 Introduction

The goal of this report is to document the work done for individual special course "A GPU-accelerated Navier-Stokes Solver using OpenCL". The project took place in GPUlab of Scientific Computing Section of Department of Informatics and Mathematical Modeling (DTU Informatics) at Technical University of Denmark. The project work for this special course took place between late March until early August 2012 and was supervised by Allan P. Engsig-Karup Ph.D. The course was worth 5 ECTS points.

The practical goal of this project was to design and implement a scientific computing application for execution on GPUs. The application was a Partial Differential Equations (PDE) solver for Navier-Stokes equations, which form a basis for most of Computational Fluid Dynamics (CFD) problems. The application was supposed to make us The solution was based on sequential implementation by Griebel et al.[2]. The implemented solution was verified and validated through simulation of standard benchmarking problems in the field like lid-driven cavity problem, flow-over a backward-facing step, flow past an obstacle and others. The performance of implemented PDE solver was analysed for different sizes of two-dimensional grids. As different optimization techniques were applied for improving performance of the PDE solver, solution consists of several versions, varying in the level of optimization. The target platform was a Graphical Processing Unit (GPU). The technology used in implementation was OpenCL, an open standard framework for implementation of programmes that execute across various heterogeneous multicore architectures like CPUs or GPUs. Thus understanding the intrinsics of platform and the approach to write parallel programs using chosen technology was a prerequisite to meet the goals of the project. Moreover, for the project to be feasible familiarization with the application of basic principles of numerical approximation and discretization was necessary.

The report is structured as follows. Section 2 is a survey of current (as of July 2012) state of technologies that can be used for development targeted to GPU architecture. A look into how different vendors implement the OpenCL standard is presented. In addition, a general overview of current achievements in GPU architectures are shown. Section 3 consists of description of necessary theory behind the problem in question. Section shows the Navier-Stokes equations and how are they discretized. The algorithm is described here. In Section 4 design of the application and its implementation is described. It shows how the algorithm is implemented and presents the details about structure of the code. Furthermore, challenges behind implementation are presented. Section 5 consists of the results and performance analysis. The report is concluded with Section 6, where general findings are summarized and possible improvements and future work is presented.

2 Survey of GPGPU programming

2.1 General trends in GPGPU programming

General-purpose computing on graphics processing units (abbreviated GPGPU) is a technique that utilizes graphics processing units (GPUs). Initially they were used to handle advanced computations in computer graphics in games and visualisations. However, there is a growing trend to extend their usage to perform data-intensive computations that were until not long ago still bound to CPUs only. This means an overall acceleration in variuos general-purpose scientific and engineering applications. In this project, a GPU-based computation-intensive program was implemented, so the platform at hand had to be thoroughly investigated to make most use of it. Thus, this section will focus on this platform mainly and will try to define current efforts in GPGPU programming as of July 2012. The descriptions will be supported with personal experiences gathered during the development process.

Massively parallel, programmable GPU architecture enables superior performance and power efficiency thorugh thousands of smaller, more efficient cores that are mainly intended to be used for parallel computation. Current CPUs consist of a few cores optimized for serial processing. While CPU runs the remainder of the code that mainly controls the flow of the program, GPU allow for order of magnitude faster processing of data through parallelizable data-intensive algorithms. To exploit the high performance of GPUs though, the implementation must express sufficient fine-grained parallelism. Moreover, GPU–CPU data transfer should be minimized and coalesced memory access achieved. A programming model to achieve that has to be provided yielding best performance results possible with a burden of low-level programming. Although the field is relatively young, it progresses extremely fast as already hit the mainstream.

First immediate observation is that there is a still diffusion of non standardized methods in the field, although standardization efforts can be observed. OpenCL is the currently dominant open general-purpose GPU computing language. Dominant proprietary framework is NVIDIA's CUDA framework. One option for writing code to execute on GPU is thus to use directly one of these frameworks. However, for a large scientific Fortran or C code, that was written long before the GPUs even existed, this requires a large effort to rewrite, debug, and maintain a separate version of the code.

Furthermore, in the advent of heterogeneous computing systems of which CPU-GPU combination is an example grow in their complexity. They usually consist of a variety of device architectures like: multi-core CPUs, GPUs, Accelerated Processing Units (APUs) or specialized FPGAs to name a a few. Programs need to be designed to work on multiple platforms and be implemented with parallelism in mind that span over number of devices. Such scale is not addressed in this report though.

Currently, there are two significant industry players in the GPU market: NVIDIA and AMD. This is due to the fact that they are two remaining vendors of GPU architectures nowadays.

As a GPU programming pioneer and 5-years-long-runner NVIDIA has succeeded in providing means to program their proprietary hardware. Most of their efforts go into marketing their CUDA technology and the platforms that support it. From NVIDIA's perspective the main goal now is to bring GPU computing into the post-CUDA age. CUDA C and Fortran are the most widely used programming languages for GPU programming nowadays. However, as the underlying technology is proprietary to NVIDIA and a software model of GPU computing offered is relatively low-level, the use of CUDA today tends to be restricted to computer scientists. The average programmer can be discouraged by the number of new constructs to learn and the lack of abstraction. Researchers are mostly interested in ad-hoc boost in performance of their algorithms usually written in high-level languages. They seek automation in process of porting their codes to GPU platforms.

Meanwhile, AMD chose another approach and is busy implementing and supporting the latest specification of OpenCL standard.

Other companies like Intel try to catch up on them introducing their own specialized hardware - CPUs with embedded Hardware Graphics and technologies and providing implementation and support for the current standards like OpenCL.

2.2 OpenCL

OpenCL (Open Computing Language) is an open standard that defined and maintained by Khronos Group and was initially developed by Apple Inc. It is a vendor-independent framework that was from the beginning meant to enable the programmer to develop portable code that can be executed on heterogeneous platforms consisting of CPUs, GPUs and potentially other processing units like APUs or FPGAs. Thanks to that it form a real base in the age of heterogeneous computing. Currently, it has several adopters, among others Intel, AMD, NVIDIA and ARM Holdings.

OpenCL consists of a programming language based on C99 used to write kernels. These are basic units of executable code, C-like functions that are executed on devices that support OpenCL. It also comprise of a set of APIs. Runtime API is used to control kernel execution and manage the scheduling, computation and the memory resources. The

Platform Layer API defines the environment called a context on the platform at hand and creates a hardware abstraction layer over diverse computational resources. Parallel programming in kernels is supported on the level of task-based and data-based parallelism.

A compatible compiler like Visual C++ compiler, gcc, Nvidia's nvcc or Intel's compiler is used to compile a collection of kernels and other supporting functions into an OpenCL Program that is executed on the GPU. This is analogous to a dynamic library in C. An application has a queue that stores kernel execution instances in-order. The execution though might be in-order or out-of-order. The most basic unit of work on an OpenCL device is called a work item.

As OpenCL matures, lots of learning materials can be found in the Internet. In addition, there are numerous courses held at universities all over the world that choose OpenCL as a technology for teaching High Performance Scientific Computing. However, a significant sign of the maturity of some technology can be usually measured in the number of the published books available in the market. OpenCL has a few from which a book by Gaster et al. [1] seemed to be most helpful.

When it comes to the usage of OpenCL in industry, academic institutions, research labs and FPGA vendors are leading implementers. The OpenCL standard homepage[**?**] specifies a number of specialized products that use OpenCL. To name some more popular customer products, latest version of Adobe Photoshop CS6 uses OpenCL to accelerate the image drawing. A different usage is a Winzip 16.5 that uses OpenCL to accelerate extraction and encryption of the archives. An interesting use of OpenCL is an OpenCL Studio.

2.2.1 Current specification

The current specification of the OpenCL API is 1.2 released November 15, 2011. However, as each of the adopters implement the standard specification on their own, the scope and support for new version varies. What can be said is that all of them guarantee a support for the version 1.1 of standard. However, some of them like AMD competes to adopt new standard as it introduces some valaubale improvements to the OpenCL programming model. In particular, subdevice abstraction provides ways to partition a device into parts that can be treated as seperate device could in 1.1. Partitioning of devices gives more control over assignment of computation to compute units. Furthermore, custom devices and built-in kernels address the problem with embedded platforms like specialized FPGAs or non-programmable hardware with associated firmware (e.g. video encoder/decoders or DSPs). These cannot support OpenCL C directly, so built-in kernels can represent these hardware and firmware capabilities. Development closer to GPU touched the Image format of memory resources.

2.2.2 OpenCL SDKs

There are different OpenCL Desktop Implementations by NVIDIA, AMD and Intel. Intel supports OpenCL thorugh its own Intel SDK for OpenCL Applications 2012 released recently. So does AMD through its APP SDK. NVIDIA supports OpenCL through CUDA SDK.

2.2.3 Comparison with CUDA

Multiple comparisons have been drawn between CUDA and OpenCL since its inception.[59][60] They both draw the same conclusions: if the OpenCL implementation is correctly tweaked to suit the target architecture, it performs no worse than CUDA. Because the key feature of OpenCL is portability (via its abstracted memory and execution model), the programmer is not able to directly use GPU-specific technologies, unlike CUDA. CUDA is more acutely aware of the platform upon which it will be executing because it is limited to Nvidia hardware, and therefore, it provides more mature compiler optimisations and execution techniques. Furthermore, the CUDA compiler displayed more mature compilation techniques, such as more aggressive pragma unroll addition to loops. Therefore, the developer is required to add in the optimisations manually to the kernel code. This is indicative of the maturity of the CUDA toolkit versus the newer OpenCL toolkits. It is likely in the future that this gap will be closed as the toolchains mature.

2.2.4 WebCL

Bringing parallel computation to the Web through JavaScript binding to OpenCL On 17 April 2012 Khronos released a WebCL working draft[

2.2.5 Future developments

OpenCL-HLM

OpenCL-SPIR

Long-Term Core Roadmap

2.3 Other GPGPU Technologies

2.3.1 CUDA

4.2

In Decemeber, NVIDIA made its nvcc compiler open.

2.3.2 DirectCompute

enable cross-platform development

2.3.3 C++ AMP (C++ Accelerated Massive Parallelism)

2.3.4 OpenACC

Directives like OpenMP for multicore CPU programming

announcement of a new directives-based parallel programming standard for accelerators. Called OpenACC, the open standard is intended to bring GPU computing into the realm of the average programmer, while making the resulting code portable across other accelerators and even multicore CPUs

But the real end game for OpenACC supporters is for the directives to be incorporated into the OpenMP standard. Since OpenACC was derived from work done within the OpenMP Working Group on Accelerators,

2.3.5 OpemHMPP

2.3.6 AMD Accelerated Parallel Processing (APP) SDK

formerly ATI Stream

2.4 Current advances in GPU Architectures

today we have two viable and competitive product lines, Nvidia and Advanced Micro Devices (AMD) GPUs, with support for a wide range of programming languages.

GPUs on super-computers

Clusters of GPUs - HPC 500 - most fast are using the GPUs.

2.4.1 NVIDIA

Tesla

Fermi

Kepler

Customer Cards

Geforce, Quadro Tesla for workloads where data reliability and overall performance are critical

2.4.2 AMD

APUs

GPUs

multicore x86

Radeon

AMD APP Acceleration

AMD is still pushing its OpenCL strategy for GPU computing.

2.4.3 Intel

Intel's upcoming Many Integrated Core (MIC) coprocessor, Intel MIC

2.5 Examples of research projects on GPU

2.6 Experiences with OpenCL

2.6.1 Developing OpenCL Code

Most of the development was done on Intel processor and NVIDIA Geforce card on Windows platform. Microsoft Visual Studio 2010 IDE was used for development.

During the development for GPU platform it is invaluable to have an access to a decent debugger that allows the programmer to debug kernels. During this project there was a the lack of debugging options. Such situation was due to the fact that the choice of debugger is dependant on the platform used. Some combinations will then be not supported as of now, when OpenCL is still in its infancy as a standard. So it was the case during the development process for this project as NVIDIA card was combined with OpenCL technology. Two working options for debugging tried were NVIDIA Nsight (currently in version 4.2) and AMD gDEBugger (currently in version 6.2).

The former supports CUDA code only. The tool is an application based on a client-server architecture. The Visual Studio Plugin serves as a client to the Monitor client process run on the machine. This allows for remote access to GPUs. Moreover, the breakpoints can be set inside the CUDA kernel code so as the current state of local variables, warps and working items will be presented. This valuable data is available through a set of info windows. Tool does not work with OpenCL code though, as it is not possible to set breakpoints in OpenCL code. Tool is distributed as a plugin to Visual Studio 2010.

The later works only with OpenCL code, but supports only the AMD devices. However, it stops on breakpoints on host. Thus it can be partially helpful during the development as it can be configured to stop on specific OpenCL functions and OpenCL errors. It also offers function call history and function properties that gives an overview into the state of paramters that OpenCL API is called with. Finally, there is also an explorer that enables programmer to browse a tree structure of OpenCL constructs used.

On the side, there is an Intel SDK that allows for checking if the code is ready to be run on Intel platform like CPU or new 3rd generation processors with hardware graphics embedded. An offline OpenCL compiler can be used to check the validity of kernel code.

Development on higher level of abstraction is possible through C++ wrapper API. However, this was not attempted in this project. Moreover, OpenCL API is ported in form of wrappers to popular programming languages like Java (JavaCL) or Python (PyOpenCL). Such projects surely will advance a wide adoption of OpenCL standard.

2.6.2 Profiling OpenCL Code

Because most of the development process for this project was done on the NVIDIA platform, some of its tools like NVIDIA Visual Profiler 4.2 or NVIDIA Nsight Profiler Visual Studio Edition 2.2 were researched. The latter can only be used with CUDA code, so it was useless for this project. The Visual Profiler was the only program that proved to be helpful as it provided detailed statistics about metrics and events of the application execution. Furthermore, it provides a timeline graph where memory transfers and kernel computations are marked in time. Finally, analysis results provide guidelines that programmer can use to optimize the code. This tool is distributed as part of CUDA Toolkit as of version 4.2. Another helpful resource provided by NVIDIA is its CUDA GPU Occupancy Calculator spreadsheet that allow easy calculation of multiprocessor occupancy of a GPU by a given CUDA kernel. This can be used to optimize OpenCL kernels on NVIDIA platform.

Another profiling tools from the competitor AMD are AMD APP Profiler and AMD KernelAnalyzer. Gaster et al. in their book[1] provide an overview and step-by-step description of how to use this tool. These tools were not used during this project, because the code was not developed on the AMD platform. This is an obvious requirement while using these tools. Tools are available for free in the AMD Developer Zone.

3 Navier-Stokes Solver

3.1 Computational fluid dynamics

3.2 Description of Navier-Stokes Solver

Reynolds number < 1000

3.3 Other attempts of implementing the solver

grid The most common form for a stream to take in GPGPU is a 2D grid because this fits naturally with the rendering model built into GPUs. Many computations naturally map into grids: matrix algebra, image processing, physically based simulation, and so on.

4 Design and Implementation

4.1 Design

A set of kernels

Modularity

All code should be executed on GPU for valid comparison.

4.2 Implementation

4.2.1 Developing with OpenCL

A simplest standard OpenCL pipeline would consist of following steps:

1.

2.

find platform find device create context and queue build program write device buffers execute kernel in range read device buffers

Code in project is based on structure from Griebels code[2]. Least as possible was changed in structure as to keep it comparable with the original code.

Code is written in C and OpenCL C.

Two-dimensional arrays in original One-dimensional arrays for OpenCL code. 1D arrays cannot be passed to OpenCL kernels.

Worksize

Implementation consists of sets of kernels as

Griebels cod intrinsics: Memory allocation

Code is portable and works under both on Windows and Unix platforms. Tested on Windows 7 and Linux clusters. Compilers used are nvcc and Visual Studio on Windows and gcc on Linux.

Table Computer specs.

Table

4.2.2 Functions

Map

The map operation simply applies the given function (the kernel) to every element in the stream. A simple example is multiplying each value in the stream by a constant (increasing the brightness of an image). The map operation is simple to implement on the GPU. The programmer generates a fragment for each pixel on screen and applies a fragment program to each one. The result stream of the same size is stored in the output buffer.

Reduce

Some computations require calculating a smaller stream (possibly a stream of only 1 element) from a larger stream. This is called a reduction of the stream. Generally a reduction can be accomplished in multiple steps. The results from the prior step are used as the input for the current step and the range over which the operation is applied is reduced until only one stream element remains.

Gather

The fragment processor is able to read textures in a random access fashion, so it can gather information from any grid cell, or multiple grid cells, as desired.

4.2.3 Stencil computation

4.2.4 Benchmarking code

Code is timed with standard library time.h clock function. No OpenCL timer code because on CPU. Code timed with buffer allocation and memory access and without.

4.2.5 Visualisation code

Matlab scripts to visualise code. Tests.

4.2.6 Naive kernels

Straightforward port of functions to kernels. Getting rid of for loops. Ensuring that boundaries are not crossed.

4.2.7 Shared memory kernels

4.2.8 Development

Production code should, however, systematically check the error code returned by each API call and check for failures in kernel launches (or groups of kernel launches in the case of concurrent kernels)

5 Performance Analysis

Synchronization between work-items is possible only within workgroups using barriers and memory fences. It is not possible to synchronize workitems that are in different workgroups. This would hinder data parallelism.

Memory management is explicit, so a programmer moves data from host to global memory of device first using OpenCL constructs. Then it can be moved to local memory. To read the results on the host the process has to be reverted.

Workgroup size for a given algorithm should be selecten to be an even multiple of the width of the hardware scheduling units.

5.1 Computed results

Theoretic bandwidth vs. Effective bandwidth

Trap of premature optimization

Visual Profiler Recommendations

Bandwidth

Data Transfer Between Host and Device

PCIe

coalescing global memory accesses

compute capability 2.x

Branching and Divergence Avoid different execution paths within the same warp.

vector operations, vector variables

built-in math functions, fast optimized library

Syncrhonization on Global and Local barriers

Synchronization between kernels in the queue. All the kernels are waiting for another to finish to work on nits results. One device - one queue.

CPU code in VIsual studio 2010 basic release build optimizations on both CPU execution and CPU control parts of GPU execution

Asynchronous reading of memory CL_FALSE instead of CL_TRUE

allocating memory withou copyting from host pointer

No print options in the loop

Workgroup size

Size of the grid

Kernels are executed asynchronously enquing memory resources asynchronously using events to track the execution status

6 Conclusions

6.1 General

The project's goal was to get acquainted with doing computations on GPU.

No prior experience with CFD codes and optimizations of numerical methods. If had any could come up with implementation of other numerical method that could be easier to parallelize.

No prior experience with OpenCL (in most parts the same as CUDA, but lack tools)

Lot of problems with constructing even the naive kernel as to get the maximum benefit from parallelizing code on GPU, focus should be first drawn to find ways to parallelize sequential code. Code was thus looked as subblocks of same computations. Most data parallelism should be found.

Problems with bandwidth and minizming transfer between the host an the device, because it was not directly possible to create a kernel from many parts of the algorithm. Examples were researched. Stencil code implementations were found.

To watch the results of minimization in the use of global memory, some version of kernels were implemented. Prefer shared memory access where possible.

Float and double precison.

Other platforms

Different sizes of arrays with regular step

NVIDIA Profiler used

6.2 Improvements

Another type of memory object supported by OpenCL could be used and tested, namely images. They are optimized for two-dimensional access patterns. Such access was used in this project as grids of cells are two-dimensional. The challenge would be to learn how to work with this type of memory object as the data stored in it is accessible only through specialized access functions. In comparison, the buffer objects that were used map directly to the standard array representation that programmers are used to when using C programming language.

6.3 Future Work

Solution could be simulated on Tegra chip on Android Platform to check the performance of computation on yet another type of platform. This time a mobile GPU would be used. Such

References

[1] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Elsevier /Morgan Kaufmann, San Francisco, CA, USA, 2011. 2.2, 2.6.2

[2] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. *Numerical Simulation in Fluid Dynamics: A Practical Introduction*. Siam Monographs on Mathematical Modeling and Computation. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998. 1, 4.2.1

List of Appendices

A Appendix: Source Code

B Appendix: Platforms and devices

C Appendix: Visualisations