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 1, 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.[1]. 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 GPU programming

2.1 Current state of GPGPU

General-purpose computing on graphics processing units (abbreviated GPGPU) is Diffusion of non standardized methods CPU for a few specialized uses Order of magnitude faster processing of parallelizable algorithms

For the exploitation of the high performance of GPUs, codes must express sufficient fine-grained parallelism, minimize GPU–CPU data transfer, and achieve coalesced memory access. One option for writing the GPU code is to do so directly in CUDA [1] or OpenCL [27]. For a large Fortran code written well before the existence of GPUs, this entails a massive effort to rewrite, debug, and maintain a separate branch of the code. An alternative to performing such a monolithic code rewrite is to follow the accelerator design, as so called by Cohen and Molemaker, of porting bottleneck loops or subroutines in isolation, with memory transfer calls made just before and after these bottleneck loops or subroutines. Although this allows for the possibility of an incremental porting effort, it suffers the drawback of introducing the crippling bottleneck of a large amount of data transfer across the system bus, which has at least an order of magnitude lower bandwidth than that of the internal GPU memory, and thus severely restricts the possible gain in performance when porting

Heterogeneous: Developers leverage AMD GPUs and CPUs for optimal application performance and user experience

Industry Standards:OpenCL and DirectCompute11enable cross-platform development

High performance: Massively parallel, programmable GPU architecture enables superior performance and power efficiency

With OpenCL

Leverage CPUs and GPUs to accelerate parallel computation Get dramatic speedups for computationally intensiveapplications Write accelerated portablecode across different devices and architectures

AMD APP Acceleration

AMD is still pushing its OpenCL strategy for GPU computing.

From NVIDIA's perspective, the overriding goal is to bring GPU computing into the post-CUDA age. CUDA C and Fortran are the most widely used programming languages for GPU programming today, but the underlying technology is proprietary to NVIDIA and offers a relatively low-level software model of GPU computing. As a result, the use of CUDA today tends to be restricted to computer science types, rather than the average programmer or researcher.

Can write in C++ too.

Ports to Java (JavaCL) and Python (PyOpenCL).

Adopters:

2.1.1 Automation of porting to GPU process

2.2 OpenCL

OpenCL (Open Computing Language) is an open standard defined by Khronos Group [16]. In contrast to the CUDA library supported mainly by NVIDIA it is a vendor-independent framework that enables the programmer to develop software that can be executed on heterogeneous platforms consisting of CPUs, GPUs and potentially other processing units like FPGAs.

22Kernel Basic unit of executable code -similar to a C function Data-parallel or task-parallelProgram Collection of kernels and other functions Analogous to a dynamic libraryApplications queue kernel execution instances Queued in-order Executed in-order or out-of-order

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

2.2.1 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.

The largest problem during GPU programming is lack of debugging options. The choice of debugger is dependant on the platform used. Some combinations will then be not supported. It was the case during the development process for this project as NVIDIA card was combined with OpenCL technology. Two working options for debugging found: NVIDIA Nsight (currently in version 4.2) and AMD gDEBugger (currently in version 6.2).

First supports CUDA code only. The tool is an application in 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 and the current state of local variables, warps and working items will be presented. This valuable data is available through a set of info windows. Warps can be observed. Tool does not work with OpenCL code though. It is not possible to set breakpoints in OpenCL code. Tool is distributed as a plugin to Visual Studio 2010.

The second works only with OpenCL code on the AMD devices. Stops on breakpoints on host. Can be asked to stop on specific OpenCL functions. Offers function call history an function properties. There is also an explorer where OpenCL constructs are presented in a tree structure.

In addition, 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 compiler is used tor these purposes.

OpenCL Desktop Implementations

OpenCL Books

2.2.2 Profiling OpenCL Code

Nsight Profiler

AMD gDEBugger

2.2.3 Differences to 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.

2.3 Previous research

2.4 Recent developments in CUDA and OpenCL

2.4.1 OpenCL

What applications use OpenCL GPU-acceleration? OpenCL in Photoshop CS6 WinZip 16.5

OpenCL Studio

2.4.2 CUDA

4.2

In Decemeber, NVIDIA made its nvcc compiler open.

2.4.3 Other Technologies

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

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,

AMD Accelerated Parallel Processing (APP) SDK

formerly ATI Stream

2.4.4 WebCL

Bringin parallel compuataion to the Web through JavaScript binding to OpenCL

2.4.5 Other

OpenCL-HLM

OpenCL-SPIR

Long-Term Core Roadmap

2.5 Current advances in NVIDIA and AMD 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

2.5.1 NVIDIA

Tesla

Fermi

Kepler

2.5.2 AMD

APU

GPUs

multicore x86

2.5.3 Intel

Intel SDK for OpenCL Applications 2012

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

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

Code in project is based on structure from Griebels code[1]. 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

4.2.1 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.

4.2.2 Stencil computation

4.2.3 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.

4.2.4 Visualisation code

Matlab scripts to visualise code. Tests.

4.2.5 Naive kernels

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

4.2.6 Shared memory kernels

4.2.7 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.

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] 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

List of Appendices

A Appendix: Source Code

B Appendix: Platforms and devices

C Appendix: Visualisations