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.

July 27, 2012

DTU Informatics
Department of Informatics and Mathematical Modelling

Contents

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

General-purpose computing on graphics processing units 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

2.0.1 Automation of porting to GPU process

http://stackoverflow.com/questions/1126989/what-future-does-the-gpu-have-in-computing

2.1 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

2.1.1 Differences to CUDA

2.2 Previous research

2.3 Recent developments in CUDA and OpenCL

2.3.1 OpenCL

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

OpenCL Studio

http://www.youtube.com/user/OpenCLStudio

2.3.2 CUDA

2.3.3 Other Technologies

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

http://msdn.microsoft.com/en-us/library/hh265137

OpenACC

Directives like OpenMP for multicore CPU programming

http://openacc.org/

AMD Accelerated Parallel Processing (APP) SDK

formerly ATI Stream

http://developer.amd.com/sdks/AMDAPPSDK/Pages/default.aspx

http://www.amd.com/us/products/technologies/amd-app/Pages/eyespeed.aspx

2.4 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

2.4.1 NVIDIA

Tesla

Fermi

Kepler

2.4.2 AMD

APU

2.4.3 Intel

Intel SDK for OpenCL Applications 2012

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 Benchmarking code

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

4.2.3 Visualisation code

Matlab scripts to visualise code. Tests.

4.2.4 Naive kernels

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

4.2.5 Shared memory kernels

5 Performance Analysis

5.1 Computed results

6 Conclusions

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. 4.2