IMPLEMENTATION OF A CARTESIAN GRID INCOMPRESSIBLE NAVIER-STOKES SOLVER ON MULTI-GPU DESKTOP PLATFORMS USING CUDA

by

Julien C. Thibault

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2009

APPROVAL TO SUBMIT THESIS

This thesis presented by Julien C. Thibault entitled *Implementation of a Cartesian Grid Incompressible Navier-Stokes Solver on Multi-GPU Desktop Platforms Using CUDA* is hereby approved:

Inanc Senocak — Date
Advisor

Amit Jain — Date
Committee Member

Timothy Barth — Date
Committee Member

John R. Pelton — Date
Dean of the Graduate College

Dedicated to my parents

ACKNOWLEDGMENTS

Many thanks to Dr. Massimiliano Fatica, Dr. Patrick Legresley, Dr. David Luebke from NVIDIA and Dr. Timothy J. Barth from NASA Ames Research Center for helpful discussions on CUDA and GPU computing. Thanks are extended to Marty Lukes and Luke Hindman of Boise State University for their help with building our desktop supercomputers.

I would like to thank Dr. Amit Jain for allowing me to join the Master program in Computer Science at Boise State University. He has always been there to advise me, through my first semester when I was only an exchange student, and later on when I decided to join the program.

I thank Dr. Inanc Senocak for his support all along this thesis. He made this experience really valuable for me. The subject of this thesis was challenging not only because of the parallel architecture of the GPUs but also because of the numerical methods involved. He helped me understand the challenges of computational fluid dynamics, in which I had only basic knowledge, to allow me to implement the software solution.

Many thanks to Boise State University and to the College of Engineering for the financial aid they provided and their ongoing support during these two last years. Finally I thank NVIDIA Corporation and Micron Technology, Inc. for their hardware donations. This work was partially funded by NASA Idaho EPSCoR Research Initiation grant.

ABSTRACT

Today's Graphics Processor Units (GPU) are powerful computation platforms used not only for graphic rendering but also for multi-purpose computation. Now reaching a teraflops of peak performance and over a 100 GB/sec of bandwidth, GPUs outperform the latest CPUs and provide a new high-performance computing platform. New languages such as CUDA and Brook+ allow developers to target the programmable unit of the GPUs without a graphics programming background. Scientists and engineers in various fields have started benefiting from the last generations of GPUs. In this thesis, the implementation of a Navier-Stokes solver for incompressible flow around urban-like domains is presented. Transport and dispersion of contaminants in urban environments is an area of intense research. The computational fluid dynamic (CFD) models necessary to provide realistic simulations require heavy computation, usually only possible on CPU clusters. This thesis presents the base for an urban dispersion model implementation on desktop platforms, using one or multiple GPUs as coprocessors. The governing equations implemented for this thesis are common to many problems in CFD where flow motion is involved. Using a single Tesla C870 GPU card, the CUDA implementation of the lid-driven cavity problem runs 33 times faster than a serial C code running on a single core of an AMD Opteron 2.4GHz processor. A speedup of 100 was reached by associating the Tesla S870 quad-GPU system to a quad-core CPU machine. Computations for both GPU and CPU are single precision. A more complex application including obstacle capability was developed to model building effects in the domain. Using

the quad-GPU system, the flow-field in a domain of 1.28 km $\times$ 1.28 km $\times$ 320 m was computed. A low Reynolds number flow-field projection of 22 minutes (1000 time steps) could be simulated in 3 minutes. Results show that an urban dispersion is feasible on this type of platform and that models can be run within minutes to provide emergency responses. More generally, it shows that complex CFD problems can benefit from multi-GPU desktop architectures.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API – Application Programming Interface

CB – Chemical-Biological

CFD – Computational Fluid Dynamics

CPU – Central Processing Units

CUDA – Compute Unified Device Architecture

FLOPS – Floating-point Operations Per Seconds

GPGPU – General Purpose GPU

GPU – Graphics Processor Units

HPC – High Performance Computing

LBM – Lattice-Boltzman Method

LES – Large Eddy Simulation

MPI – Message Passing Interface

PDE – Partial Differential Equations

SDK – Software Development Kit

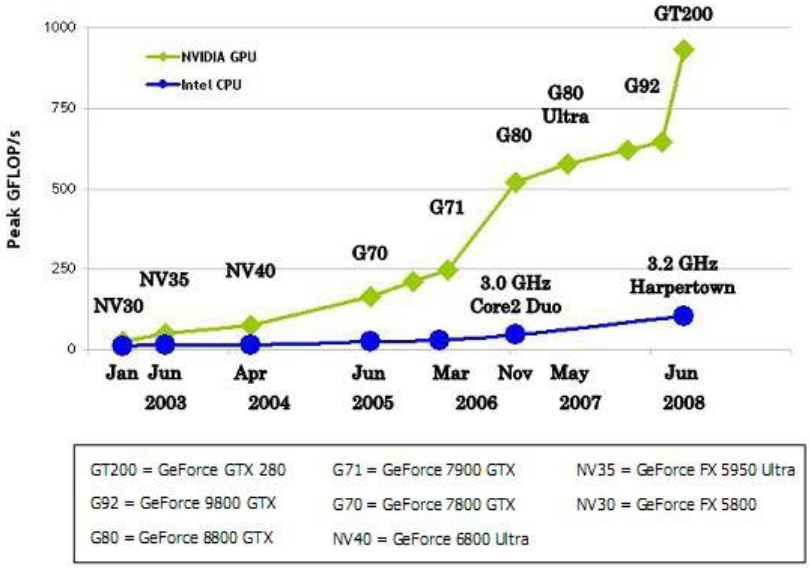
SIMD – Single Instruction Multiple Data

CHAPTER 1

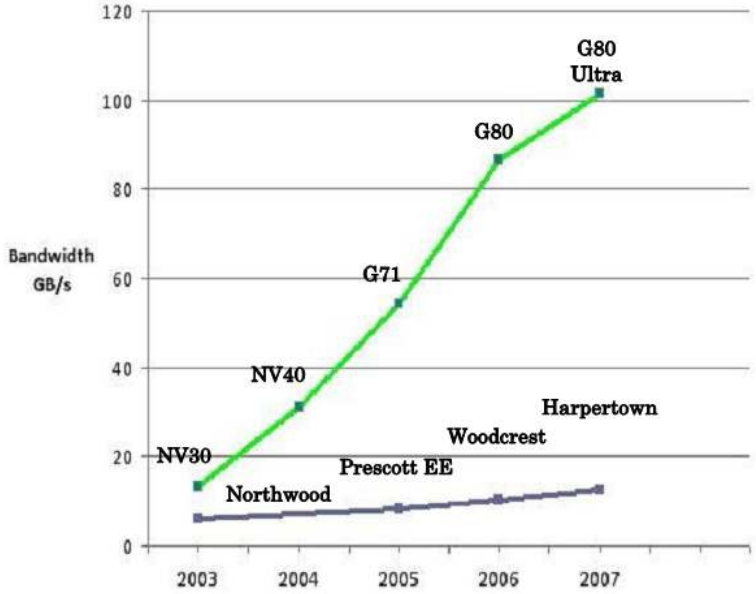
INTRODUCTION

1.1 Problem Context

In the last decade, CPU designers have focused on developing multi-core architectures instead of increasing the clock frequency by putting more transistors on the die because of power constraints [29]. GPU designers have adopted the many-core strategy early on, because graphics rendering is a parallel task. GPUs are based on the stream processing architecture that is suitable for compute-intensive parallel tasks [38, 40]. Modern GPUs can provide memory bandwidth and floating-point performances that are orders of magnitude faster than a standard CPU. Figure 1.1 depicts the growing gap in peak performance between GPU and CPU over the last five years. Currently, NVIDIA GPUs outperform Intel CPUs on floating point performance (Figure 1.1(a)) and memory bandwidth (Figure 1.1(b)), both by a factor of roughly ten [38]. Until recently, using the GPUs for general purpose computation was a complicated exercise. A good knowledge of graphics programming was required, because GPU's old fixed-function pipeline did not allow complex operations [39]. GPUs have evolved into a programmable engine, supported by new programming models trying to find the right balance between low access to the hardware and high-level programmability [39]. The *BrookGPU* programming model, released in 2004 by Stanford University, offered one of the first development platforms for gen-



(a) Floating-point Performance



(b) Memory Bandwidth

Figure 1.1: Performance comparison between Intel CPUs and NVIDIA GPUs (courtesy of NVIDIA).

eral purpose GPU (GPGPU) programming [10, 40]. *BrookGPU* provides a GPU abstraction layer that enables data parallelism. It keeps the programmer away from having an extensive knowledge of graphics programming - like OpenGL - while being platform independent. In 2007, NVIDIA released a new programming model for its own line of GPUs: Compute Unified Device Architecture (CUDA) [38]. With CUDA, NVIDIA offers a common architecture and programming model for its own line of GPUs. The C-based application programming interface (API) of CUDA enables data parallelism through the use of shared memory, but also computation parallelism thanks to the introduction of the thread and grid concepts. The CUDA programming model has found success in the GPGPU community. There is also a recent effort called MCUDA [46] to program multi-core CPU architectures with the same paradigms exposed in CUDA. On the other hand, AMD offers a compute abstraction layer (CAL) for GPU programming. A modified version of the Brook open source compiler (Brook+) was developed to support this cross-platform interface to the GPU. Both AMD CAL and Brook+ are available in AMD's software development kit (SDK) [30].

Advances in many-core architectures have been tremendous, but using the full potential of many-core architectures is not an easy task. Engineers and scientists may need to rewrite and optimize their legacy sequential codes to harness the compute-power of modern day multi-core CPUs, and many-core GPUs. Message Passing Interface (MPI) programming [21] has been widely adopted in parallel scientific computations. The framework provides a high level API that allows programmers to transparently make use of multiple processors on both shared and distributed systems. The programmer does not have to deal with the details of the communication protocol between the nodes. On shared memory systems, POSIX multithreading offers low level functions to implement multi-threaded systems, while OpenMP provides a

certain abstraction layer [12], which makes it more accessible to software developers. In contrast, CUDA offers a different approach that specifically targets the many-cores on a single GPU. It is the programmer's responsibility to optimize the usage of the memory and the threads available on the streaming cores [42]. Multi-GPU parallelism is not currently addressed by CUDA, which means that implementation for multiple GPUs is explicitly performed by programmers. External tools such as OpenMP, MPI or POSIX can be associated to CUDA in order to benefit from a GPU cluster or a multi-GPU desktop platform. In the future, parallel projects like CUDASA [47] might provide new frameworks that will ease development on multi-GPU systems.

1.2 Thesis Statement

1.2.1 Objectives

The current set of computational models that is adopted by the emergency responders to simulate chemical-biological (CB) contaminant dispersion is based on simplified empirical models [27]. The predictive capability of these models is often unsatisfactory, but they are still being employed because of their relatively fast run-times. The *Joint Effect Model* (JEM), funded by the Department of Defense, establishes a key performance parameter (KPP). According to the KPP, urban dispersion models with advanced features turned off shall provide hazard prediction and graphical display within 10 minutes. Many current CFD applications are not able to deliver a solution within that time frame, even when the physics modeling features are turned off [44]. During this research the main objective was to develop an incompressible 3-D Navier-Stokes solver to compute wind fields in urban-like domains. This work serves as a baseline implementation towards a CFD-based urban dispersion model.

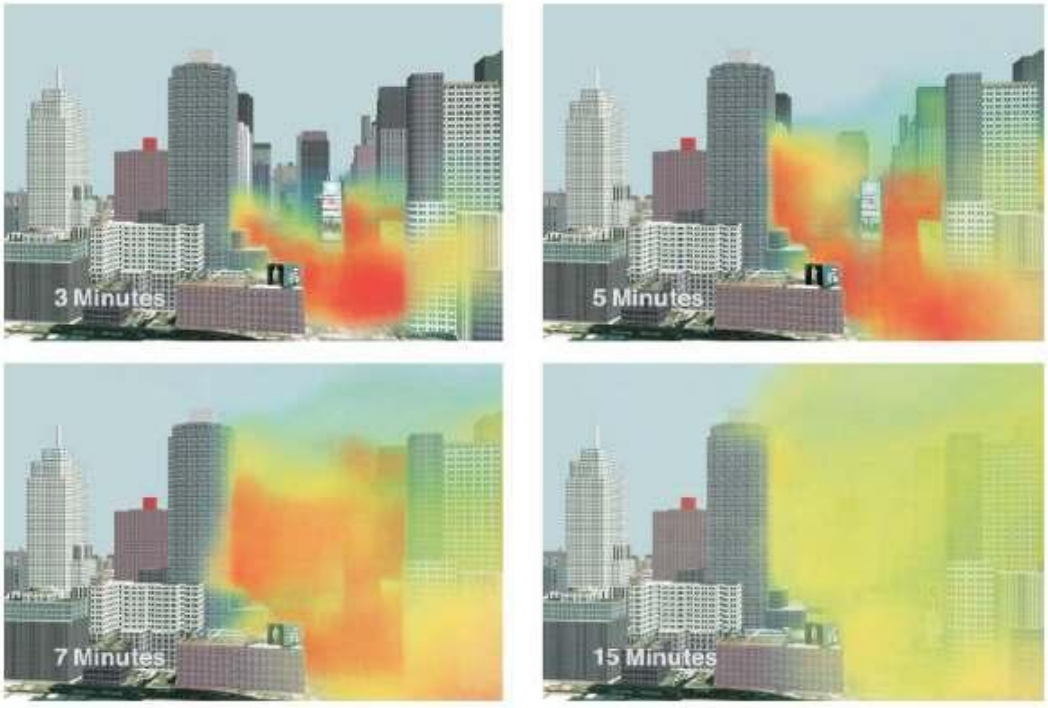


Figure 1.2: CFD Simulation of plume dispersion in Time Square, New York (courtesy of Patnaik et al. [41])

It also provides a novel obstacle capability to simulate the effects of the buildings on the flow field and a subgridscale turbulence model for large-eddy simulations (LES) of turbulent flows. The NVIDIA CUDA technology was chosen to implement the discretized form of the governing equations on CUDA-capable GPU architecture. The goal was to substantially shorten turn-around time for simulations on a multi-CPU / multi-GPU desktop computer. The CFD code that was developed as part of this thesis solves a generic set of partial differential equations for incompressible fluid dynamics (Navier-Stokes equations). Broadly speaking, the computational techniques and algorithms that were developed for multi-CPU/multi-GPU architectures as part of this research can be extended to different thermo-fluid applications such as aerodynamic

flows or weather forecasting.

1.2.2 Procedures

In order to develop the proposed CFD code, a step-by-step development approach was followed. This allowed rigorous accuracy tests of the numerical implementations. The following tasks were accomplished as part of the research:

- Implementation of the 2D wave equation to get familiar with the CUDA paradigms and apply them to a CFD problem on multi-GPU platforms.
- Implementation of a 3D Navier-Stokes solver which is the core of the final CFD code. The code was validated with the laminar channel flow exact solution and lid-driven cavity benchmark simulation [49]
- Implementation of a 3D Navier-Stokes solver with novel obstacle capability (to mimic building effects) [50].
- Implementation of a Smagorinsky large-eddy simulation turbulence model for wind-field modeling in realistic urban environments

In a parallel effort, a serial C code was developed to validate the logic of the new features added at each step of the development. It was also used as a benchmark for speedup analysis. The following high performance computing (HPC) platforms with different GPU-CPU configurations were available to perform speedup and multi-GPU scaling analysis:

- **"hellboy.boisestate.edu"- dual-GPU / dual-core CPU machine**

  2 × NVIDIA Tesla C870 boards (2 × 128 processors)

3.0 GHz Intel Core 2 Duo E8400 CPU (1333 MHz front side bus)

4GB DDR2 800 MHz memory

- **"barth.boisestate.edu"- quad-GPU / 16-core CPU machine**

  1 × NVIDIA S870 Server (4 × 128 processors)

  8 × Dual-core 2.4 GHz AMD Opteron 8216 CPU (1000 MHz front side bus)

  16GB DDR2 667 MHz memory

- **"sawtooth.boisestate.edu"- quad-GPU / dual-core CPU machine**

  2 × NVIDIA GeForce 9800 GX2 boards (2 × 256 processors)

  3.0 GHz Intel Core 2 Duo E8400 CPU (1333 MHz front side bus)

  4GB DDR3 1333 MHz memory

These three computing platforms offered different CPU and GPU configurations. First, the speedup provided by a single GPU over a serial code was determined for different types of CPU. But mainly, it helped analyzing the effects of the CPU and the GPU hardware on multiple-GPU (2, 3 or 4 GPUs) platform performance. The Appendix A.1 gives more details about the GPU computing hardware that was utilized in this research. The implementation of these different tasks and the analysis of the results using the GPU computing infrastructure is presented in the following chapters.

1.3 Prior Work

1.3.1 GPGPU Computing

Prior to the introduction of the CUDA and Brook programming models, several Navier-Stokes solvers have been implemented for the GPU. Harris [26] implemented a 3D solver to create a physically-based cloud simulation using the *Cg* programming language from NVIDIA. It is a high-level programming language for graphics on GPUs, which operates as a layer above OpenGL. His implementation was based on the "stable fluids"method proposed by Stam [45]. This method is adapted to graphics application because of the real-time visualization constraint. In Reference [33] the Navier-Stokes equations are solved for flow around complex geometries following the work of Harris [26]. Due to its relative potential for easy parallelization, the Lattice-Boltzman method (LBM) has also been implemented in different studies addressing complex geometries. In Reference [31], GPU implementation of LBM resulted in speedup of 15× relative to the CPU implementation. In Reference [17], an LBM was implemented on a GPU cluster to calculate winds and contaminant dispersion in urban areas. A speedup of 4.6× relative to a CPU cluster was achieved in their study [17], which demonstrates that GPU clusters can serve as an efficient platform for scientific computing.

High performance parallel computing with CUDA has already attracted various scientists in several disciplines, such as molecular dynamics [3, 32, 52], computational biology [43], linear algebra [6,11], weather forecasting [34] and artificial intelligence [7]. In the computational fluid dynamics (CFD) field, Tolke and Krafczyk [51] implemented a 3D Lattice-Boltzman method for flow through a generic porous medium. They obtained a gain of up to two orders of magnitude with respect to the com-

putation of an Intel Xeon 3.4GHz. Brandvik and Pullan [9] mapped 2D and 3D Euler solvers to the GPU using BrookGPU and CUDA programming models. For the CUDA version of the 3D Euler solver, their computations on NVIDIA 8800GTX showed a speedup of 16 relative to a single core of an Intel Core 2 Duo 2.33GHz, whereas the BrookGPU implementation of the 3D Euler solver showed a modest speedup of only 3 on the ATI 1950XT. Molemaker et al. [35] developed a multi-grid method to solve the pressure Poisson equation. The CUDA implementation of the multi-grid pressure Poisson solver produced a speedup of 55 relative to a 2.2MHz AMD Opteron processor [35]. Recently, Elsen et al. [16] showed that complex scientific simulations are feasible on GPUs. They implemented a BrookGPU version of the compressible Euler equations in order to simulate hypersonic vehicles. Compared to a single core of an Intel Core 2 Duo 2.4GHz the GPU implementation achieved speedups ranging from 15 to 40.

The recent literature attests to the compute-potential of GPU computing with new programming models. Numerous studies have adopted the CUDA programming model to numerical problems that have practical applications in engineering and science at large [36]. Currently, most of the current GPU applications utilize single-GPU platforms. The potential of GPU clusters has already been demonstrated [17]. But the current motherboards can now host multiple GPUs and become the core of a *superdesktop* computer for a relatively cheap cost. This thesis introduces the implementation of a CFD code on multi-GPU/multi-CPU desktop platforms and demonstrates the potential of such platforms.

1.3.2 Contaminant Transport and Dispersion in Urban Environments

Urban dispersion is a scientific field that projects how contaminant plume spreads through urban environments. It is usually associated with chemical-biological (CB) contaminant release (accidental release or terrorist attack) although it was traditionally used for preventive purposes like air pollution warning. Until recently the computational hardware did not allow scientists to run fast-response urban dispersion simulations based on CFD models. Instead, fast-running empirical or semi-empirical models were used to obtain running times acceptable for emergency responses. With the latest advances in high-performance computing hardware, CFD models can now be run on machines delivering teraflops of computing performance. They might provide a new alternative to semi-empirical building resolved (SEB) models [44]. A typical urban CFD domain is usually only a couple square kilometers large, between 1 km^2 and 5 km^2, with a vertical dimension less than 1 km. The grid system resolution is usually between 1 m and 5 m. The urban environment is modeled using 3-D building data, usually extracted from an existing database. Several studies [1,2,15,19] showed that models need to be evaluated against field data. A single-building simulation [15] is a first step to evaluate the accuracy of a model but larger-scale domains are necessary to validate the model with realistic environments. The Oklahoma [2] and Manhattan field experiments [1] provide the necessary data to evaluate models in existing urban areas. In 2003 the Department of Homeland Security sponsored a field experiment in Oklahoma City, the *Joint Urban 2003 Atmospheric Dispersion Study* [2]. The study covered the central business district of the city, a domain of 900 m by 1200 m. It included around 40 city blocks and 150 buildings. Although a cluster of 50 m tall buildings was part of the domain, most of these buildings were under

10 m. Tracer releases were performed on a month period, including day-time and nighttime releases. Several types of sensors (crane, sonic anemometers) were used to measure 5 or 10 minute averaged tracer concentrations. The study showed that the few tall buildings actually had a big impact on the overall plume dispersion.

The *MID05* experiment included six days of tracer and meteorological experiments. The studied area was located in midtown Manhattan (south of Central Park). It covered a domain of 2 km by 2 km including deep street canyons with buildings reaching 150 meters and only a couple meters apart from each other. Thirty-minute averaged tracer concentrations were collected by 64 different outdoor samplers. Since the results of these field experiments were published, several studies have been conducted to evaluate the accuracy of urban dispersion models. The *Joint Urban 2003* data were used in [20] and [28] to evaluate a Reynolds-averaged Navier-Stokes solver CFD model and a semi-empirical building resolved model, respectively. Results showed that after customizing the models, the simulated dispersions were comparable to the observed tracer releases. In [19], six different models were evaluated against the data from the *MID05* experiment. This study showed that in a real emergency context, where no update on the models is possible, the different models do not perform well and results may vary a lot from a model to another. Flaherty et al. [19] pointed that in order to reach high-accuracy simulations, the input data for the simulations must include accurate building data (databases) and accurate atmospheric data. This means that the studied area should be equipped with enough sensors to build an accurate state of the dispersion. Previous experiments on the area should also prescribe boundary conditions if the sensors cannot provide enough data.

There is still a great need for improvement in urban dispersion modeling. Patnaik et al. [41] showed that realistic urban contaminant dispersion is feasible. Figure 1.2

describes how plume dispersion can be visualized with their FAST-3D-CT model [41]. On the other hand those simulations and the postprocessing steps are still very expensive. Currently emergency responders cannot afford this type of simulation. Urban CFD simulations can be accelerated significantly if the numerical methods are carefully implemented [24, 25]. Advances in computing hardware can further accelerate computations, and allow advanced modeling features that were too time-consuming for fast-running simulations. An urban dispersion model based on a second-order accuracy LBM was developed for a 32-node GPU cluster by Fan et al. [17]. They simulated the dispersion of airborne contaminants in the Times Square area of New York City. The simulation covered an area of around 1.16 km $\times$ 1.13 km, including 91 blocks and around 850 buildings. This domain was represented by a grid of 480 $\times$ 400 $\times$ 80 computational nodes, resulting in a resolution of 3.8 m. Results showed a speedup of 4.6$\times$ over the CPU cluster implementation. This is very promising as GPU hardware greatly improved since this study was conducted in 2004. Their work represents the first implementation of the LBM on a GPU cluster. The LBM is very popular today as it is fairly easy to parallelize. This thesis present the first CFD-based implementation of the incompressible Navier-Stokes equations on multi-GPU desktop platforms using CUDA.

CHAPTER 2

TECHNICAL BACKGROUND

2.1 GPGPU as a Solution

2.1.1 Evolution of the Graphics Pipeline

Originally built for graphics rendering, GPUs are now powerful programmable engines, suitable for general purpose computation [39]. In this section, the main stages of the graphics pipeline will be introduced and the evolution of the GPU architecture will be briefly described. The graphics pipeline can be divided into 5 main stages [40]:

- Vertex processing: The different input objects are formed from individual vertices and made part of the scene system through scaling and shading (interaction with the lights). This stage is adapted to parallelization as each vertex can be computed independently.

- Primitive assembly: The vertices are collected and assembled into triangles.

- Rasterization: This stages determines which pixels are going to be hidden from the camera point-of-view. Each triangle is filled with a "fragment" which represents the actual pixels that are displayed at the screen-space location.

- Fragment operations: The final color of each fragment is determined by combining the pixel's attribute (color, depth, position) with textures fetched from

the memory. This stage is very demanding but each fragment can be computed in parallel.

- Composition: Fragments are assembled into the final image, keeping only one color per pixel.

Figure 2.1 describes the different processing steps that are needed to translate the input vertices into the final pixel data. The first generations of GPUs were imple-

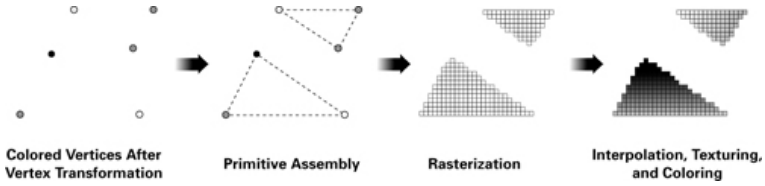


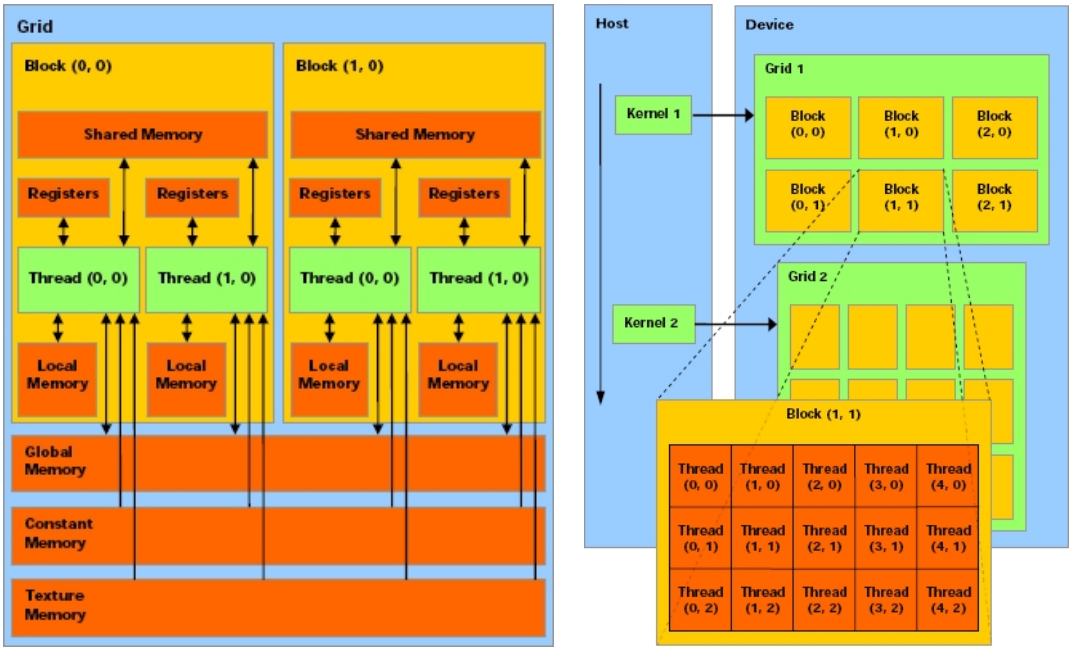
Figure 2.1: Processing steps for graphics rendering (courtesy of NVIDIA)

mented with a fixed-function pipeline. But the demand for more complex lighting and shading effects required some modifications on the pipeline to allow user-defined functions for vertex and fragment operations. Consequently, vertex and fragment programs, in addition to a larger limit on their size, gained more features to render more complex effects. The unified Shader Model 4.0, now supported by NVIDIA's and AMD's GPUs, defines the different features the vertex and fragments should support. This includes dynamic flow control (branches, loops) and 32-bit integers and 32-bit floating-point numbers. On one hand GPU designers built increasingly parallel sets of pipelines to compute vertices and fragments faster. On the other hand the programmable hardware unit of the pipeline became more and more complex and eventually became common to the fragment and vertex programs. The GPU architecture is now adapted to high-performance general computation as data-parallelism is

conserved and programmers only have to target a single hardware unit which supports the functions they can expect from a CPU.

2.1.2 CUDA Hardware Architecture

In GPU designs, transistors are devoted to data processing rather than data caching and flow control [38]. A GPU is an example of a Single Instruction, Multiple Data (SIMD) multiprocessor. In the CUDA programming model, compute-intensive tasks of an application are grouped into an instruction set and passed on to the GPU such that each thread core works on different data but executes the same instruction. The CUDA memory hierarchy does not really differ from the one for a conventional multiprocessor. Closer to the core, the local registers allow fast ALU operations. The shared memory, seen by all the cores of a single multiprocessor, can be compared to a first-level cache (L1), as it provides a memory closer to the processors that will be used to store data that tend to be used over time by any core [38]. The difference in CUDA is that the programmer is responsible for the management of this "GPU cache". The last level in this hierarchy is the global memory, the RAM of the device. It can be accessed by any processor of the GPU, but for a higher latency cost. Threads can actually perform simultaneous scatter or simultaneous gather operations if those addresses are aligned in memory [38]. Coalesced memory access is crucial for superior kernel performance as it hides the latency of the global memory. The challenge for a CUDA software developer is then, not only the parallelization of the code, but also the optimization of the memory accesses by making the best use of the shared memory and the coalesced access to the global (device) memory. Each multiprocessor also has read-only constant cache and texture cache. The constant cache can be used by the threads of a multiprocessor when trying to read the same constant value at



(a) Hardware architecture (b) Programming Model

Figure 2.2: The CUDA Model (courtesy of NVIDIA). In this example, the CUDA grid is composed of 3×2 blocks, each containing 5×3 threads.

the same time. Texture cache on the other hand is optimized for 2D spatial locality and should be preferred over global device memory when coalesced read cannot be achieved [38].

2.1.3 CUDA Programming Model

The computation core of the CUDA programming model is the kernel, which is passed on to the GPU and executed by all the processor units, using different data streams. Figure 2.2(b) presents the layout of the threads in the CUDA programming model. Each kernel is launched from the host side (CPU), and it is mapped to a thread grid on the GPU. Each grid is composed of thread blocks. All the threads from a particular block have access to the same shared memory and can synchronize together. On the

other hand, threads from different blocks cannot synchronize and can exchange data only through the global (device) memory [38]. A single block can only contain a limited number of threads, depending on the device model. But different blocks can be executed in parallel. Blocks executing the same kernel are batched together into a grid. Blocks are managed by CUDA and executed in parallel in a batch mode. The programmer needs to define the number of threads per block and the grid size (number of blocks) before launching the kernel. As mentioned earlier, CUDA API is an extension to the C programming language. It provides functions to manage the computations on the GPU. The full list of functions is discussed in detail in the CUDA programming guide [38]. The major functions used in this study are `cudaMalloc()` and `cudaMemcpy()` functions. These functions allocate memory on the GPU and copy data from the CPU memory into the device memory of the GPU, respectively. The `cudaFree()` function is used to free memory on the device. The kernel is launched by specifying the size of the grid (number of blocks) and the size of the blocks (number of threads) using the following prototype: `kernelName<<gridSize, blockSize>>()`. `__synchthreads()` can be used inside a kernel to synchronize all the threads of a same block. Global synchronization is not addressed by the CUDA model. A practical way to force a global synchronization is to exit the kernel before launching a new one. In addition, the CUDA API introduces the qualifiers `_shared_`, `_device_` and `_constant_` to define the type of memory a variable should use. The function qualifiers `_device_`, `_global_`, and `_host_` specify whether the GPU or the CPU should execute and call the qualified function [38].

2.1.4 Compilation and Development Tools

CUDA Toolkit

The CUDA Toolkit [36] enables CUDA applications to run on a general purpose computer with Windows, Linux or Mac operating systems, using the GPU as a coprocessor to the CPU. Jobs are launched on the GPU from the host process using remote procedure calls, supported by the CUDA toolkit.

NVCC Compiler

Nvcc [37] is used to compile CUDA code, which is a combination of C/C++ code for the host side and GPU code for the device. The host code can actually be compiled with any general purpose C/C++ compiler that is available on the host platform such as the GNU C compiler *gcc* [22], while the device code uses proprietary NVIDIA compilers and assemblers. Compilation options for the host code are then similar to a general purpose compiler, but NVCC offers more options to specify the mode of CUDA compilation, such as the *release*, *emulation* or *fat device code binaries* modes.

CUDA Software Development Kit (SDK)

The SDK [36] offers the necessary libraries and configuration files to build a CUDA application. Several applications are available for demonstration purpose and code source. The SDK 1.1 was used to build the applications developed during this research.

CUDA Profiler

The SDK [37] also offers a profiler. It can output different statistics about the CUDA program that is executed against it. The profiler provides information about GPU time, CPU time, the number of coalesced access memory and branch conditions, which can be displayed in summary tables but also in various kinds of plots. The CUDA profiler is useful to identify kernel implementations that have a high number of non-coalesced stores/loads and warp serializations (branch conditions).

2.2 Governing Equations

Before describing the CUDA implementation of the CFD code, it is necessary to introduce the governing equations. They are defined in the following sections along with the numerical methods that allow their implementation.

2.2.1 Wave Equation

The wave equation is a computationally tractable partial differential equation (PDE) that can serve as a model to learn and experiment with the CUDA programming model before targeting more complex PDEs such as the Navier-Stokes equations. The two dimensional wave equation is defined as:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \tag{2.1}$$

where u is the amplitude of the wave and c the propagation speed of the wave.

2.2.2 Governing Equations of Incompressible Fluid Flows

Continuity Equation

The continuity equation Eq. 2.2 describes the conservation of mass:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{2.2}$$

where u, v and w are the components of the velocity vector in the x, y and z directions, respectively.

Navier-Stokes Equations

The Navier-Stokes equations, along with the continuity equation (Eq. 2.2), describe the motion of incompressible viscous fluids:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\nabla.\mathbf{u} = -\frac{1}{\rho}\nabla P + \nu\nabla^2\mathbf{u} + \nabla.\bar{\bar{\boldsymbol{\tau}}}^R \tag{2.3}$$

where $\mathbf{u}$ is the velocity vector, P is the pressure, ρ is the density, $\bar{\bar{\boldsymbol{\tau}}}^R$ is the subgrid scale Reynolds stress term, and ν is the kinematic viscosity. Under laminar flow conditions the Reynolds stresses are absent from Eq. 2.3. For turbulent flow conditions, a turbulent eddy viscosity needs to be defined. In the Reynolds-averaged Navier-Stokes approach u, v, w, and P represent ensemble-averaged quantities and in the large-eddy-simulation approach u, v, w, and P represent filtered quantities [48].

2.2.3 Turbulence Modeling

The present research effort mainly focused on laminar flow simulations. Additionally, an LES subgrid scale model for turbulent flows was implemented, which will be

validated in future. The Large Eddy Simulation (LES) approach is used for the turbulence closure problem. In LES modeling, large scale energetic flow structures are resolved by the computations, while the effect of unresolved small-scale flow structures are represented by a sub-grid scale (SGS) model. The presented implementation uses the Smagorinsky eddy viscosity model [48]. The Smagorinsky model is characterized by a mixing-length l_{mix} which is proportional to the filter width is defined as:

$$l_{mix} = C_s \Delta, \tag{2.4}$$

where C_s is the Smagorinsky constant ($C_s \approx 0.1$) and Δ is the characteristic width of the filter. In the present study, a box filter is implemented, in which case:

$$\Delta = \sqrt[3]{\Delta x \Delta y \Delta z}, \tag{2.5}$$

where Δx, Δy and Δz are the grid resolutions in the x, y, z directions, respectively. The mixing-length appears in the eddy viscosity as follows

$$\nu_t = l_{mix}^2 \sqrt{S_{ij} S_{ji}}, \tag{2.6}$$

where the strain rate tensor S_{ij} is given by:

$$S_{ij} = \frac{1}{2} \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \tag{2.7}$$

2.3 Numerical Methods

2.3.1 Wave Equation

Second order accurate numerical schemes in both time and space are used to discretize the wave equation. The discretized form of the equation can be written as follows

$$u_{i,j}^{t+1} = c\left(1 - 2\rho^2\right) u_{i,j}^{t} + \rho^2 \left(u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t}\right) - u_{i,j}^{t-1} \qquad (2.8)$$

where u is the amplitude of the wave, and i, j, and t are the indices for the x-direction, y-direction and the time level, respectively. ρ defines the ratio of time step to the spatial resolution ($\Delta t/\Delta h$). The wave propagation speed c has a value of 1.0. The wave equation is discretized on a spatial domain size of $[0, \pi; 0, \pi]$.

2.3.2 Incompressible Navier-Stokes Equations

Staggered Grid

The Navier-Stokes equations can be discretized on either a staggered or a non-staggered (collocated) grid. The staggered grid eliminates the odd-even pressure oscillations that may occur on non-staggered grids [48]. A staggered configuration is used for this implementation. Figure 2.3 shows the location of the computational nodes for each component (u, v, w) of the velocity field and the pressure P on a staggered grid.

Projection Algorithm

Second-order accurate central difference scheme is used to discretize the advection and diffusion terms of the Navier-Stokes equations on a uniform staggered grid [18].

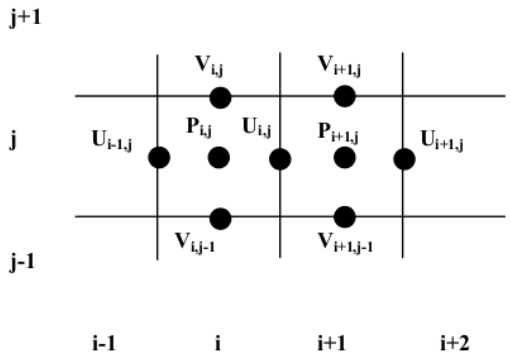


Figure 2.3: Staggered Grid. Pressure P is located in the cell centers. u and v components of the velocity are located respectively in the midpoints of the vertical and horizontal edges.

Both first-order accurate, explicit Euler scheme and second-order accurate Adams-Bashfort were used for the time derivative term. The projection algorithm [14] is then adopted to find a numerical solution to the Navier-Stokes equation for incompressible fluid flows. In the projection algorithm, the velocity field $\mathbf{u}^*$ is predicted using the momentum equations without the pressure gradient term [14, 18]. The predicted velocity, using a first order Euler scheme, can be written as follows:

$$\mathbf{u}^* = \mathbf{u}^t + \Delta t \left(-\mathbf{u}^t \nabla . \mathbf{u}^t + \nu \nabla^2 \mathbf{u}^t\right), \tag{2.9}$$

where the index t and Δt represents the time level and time step size, respectively. $\mathbf{u}^t \nabla . \mathbf{u}^t$ represents the advective term and $\nu \nabla^2 \mathbf{u}^t$ represents the diffusion term. The discretization of these terms is given in Appendix B.2. A second-order accurate method in space and time is preferable to reduce numerical errors. The second order Adams-Bashfort scheme gives

$$\mathbf{u}^* = \mathbf{u}^t + \Delta t \left[1.5 \left(-\mathbf{u}^t \nabla . \mathbf{u}^t + \nu \nabla^2 \mathbf{u}^t\right) - 0.5 \left(-\mathbf{u}^t \nabla . \mathbf{u}^t + \nu \nabla^2 \mathbf{u}^t\right)\right] \tag{2.10}$$

The discretization for the diffusive and advective terms remains the same (see Appendix B.2) but now the predicted velocity at $(t+1)$ depends on the diffusive and advective terms at both t and $(t-1)$.

The predicted velocity field $\mathbf{u}^*$ does not satisfy the divergence free condition because the pressure gradient term is not included in Eq. 2.9. By enforcing the divergence free condition on the velocity field at time $(t+1)$, the following pressure Poisson equation can be derived from the momentum equations given in Eq. 2.3:

$$\nabla^2 P^{t+1} = \frac{\rho}{\Delta t} \nabla . \mathbf{u}^* \tag{2.11}$$

In the present study, the above equation is solved using a Jacobi iterative solver to time march the equations to a steady-state solution. A more efficient solver (e.g., geometric multi-grid method) should be adopted for time-accurate unsteady simulations. The pressure field at time $(t+1)$ is then used to correct the predicted velocity field $\mathbf{u}^*$ as follows:

$$\mathbf{u}^{t+1} = \mathbf{u}^* - \frac{\Delta t}{\rho} \nabla P^{t+1} \tag{2.12}$$

2.3.3 Turbulence Modeling

Smagorinsky Subgrid Model

The eddy viscosity ν_t defined by Eq. 2.6 is calculated at the pressure point. For the magnitude of the strain rate tensor defined in Eq. 2.6 the discretization given in Appendix B.3 is used. The viscosity is calculated at the pressure point while the viscous terms are evaluated at the cell sides when computing the predicted velocity. A spatial interpolation is then necessary.

Pressure Poisson Solver

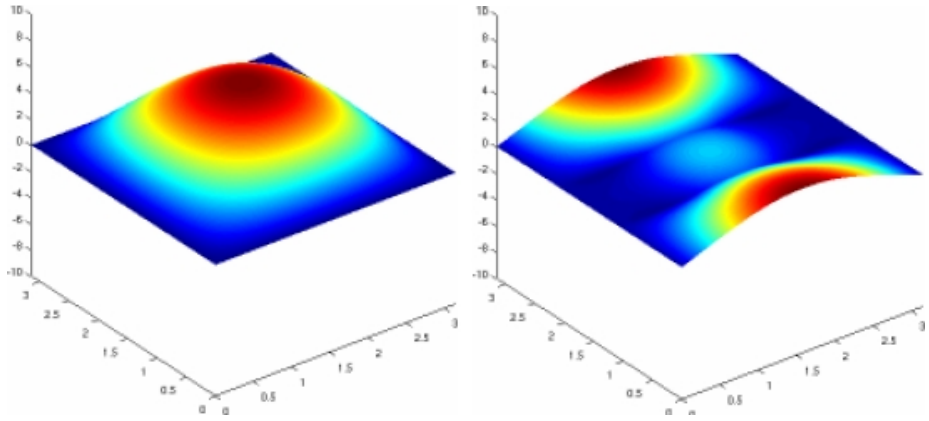
The Jacobi solver implemented in this study is not sufficient for turbulence modeling. Conservation of mass and momentum needs to be be strictly enforced as turbulent models are very sensitive to numerical errors. This can be achieved by using a multigrid method on top of the Jacobi solver.

CHAPTER 3

GPU IMPLEMENTATION & VALIDATION

3.1 Implementation of the Wave Equation

3.1.1 Wave Propagation Problem



(a) Dirichlet boundary conditions

(b) Mixed Neuman Dirichlet boundary conditions

Figure 3.1: Wave simulation on a 1024×1024 domain with different initial conditions and boundary conditions.

The 2D wave equation is a fairly simple partial differential equation (PDE) to implement. But at the implementation level, the wave equation shares certain common features with other PDEs (time-dependent solution, necessity to apply boundary conditions, etc.). Figure 3.1 shows two different waves obtained from simulations with 1024×1024 computational nodes. The first plot is from a simulation with Dirichlet

boundary conditions on all sides ($u = 0$), while the second plot uses mixed Dirichlet and Neumann boundary conditions.

3.1.2 Main Code (Host-Side)

The discretized form of the wave equation (Eq. 2.8) shows that the computation of a node value at the time level $(t + 1)$ requires information about the same node for the two previous time levels (t and $t - 1$) and information about the four neighboring nodes from the current time level (t). Therefore, it is necessary to have two matrices to keep track of the last two time levels and another matrix to store the computation results. The next time step starts after each node is computed. At this stage, the matrices are swapped so that the oldest one can be reused to store the result of the next iteration. The matrix representing the time level t now represents the time level $(t-1)$, and the one that stores the results of the previous iteration represents the time level t. This process is repeated at each time steps. Note that costly data transfer between the device and the host is avoided by employing pointers. Figure 3.2 shows how the matrices are swapped at the end of each time step (`uold`, `u`, `unew` representing u^{t-1}, u^t and u^{t+1}, respectively).

3.1.3 Single-GPU Implementation

The computational domain for the wave equation is first decomposed with a checkerboard approach. Each sub-domain is assigned to a particular CUDA block by copying the corresponding data to the shared memory. It includes the subdomain inner cells but also the cells representing the neighboring cells. Figure 3.3 shows an example for a 4×4 thread block for simplicity purposes. Only 4×4 cells are computed by the threads

```
//for each time step
for (t=0; t < ntstep; t++)
{
   //call kernel to compute ut
   wave << grid, block >> (unew, u, uold)
   //rotate matrices
   utemp = uold; uold=u; u=unew; unew=utemp
}
```

Figure 3.2: Host-side code for the CUDA implementation of the wave equation. The wave kernel is launched at each time step for synchronization across CUDA blocks.

but 6×6 cells are actually needed to be able to compute the inner cells on the north, south, east and west edges. Note that in the actual implementation, each block holds 16×16 threads, for a total of 256 threads per block for faster computations. CUDA actually allows 512 threads per block but the CUDA user guide advises to use 256 threads, which was confirmed in the results of this study. The grid size depends on the size of the physical problem. This first implementation assumes a square domain that can be divided into several 16×16 subdomains. A 1024×1024 domain for example would contain 64×64 blocks. A CUDA block is not a physical multi-processor, the

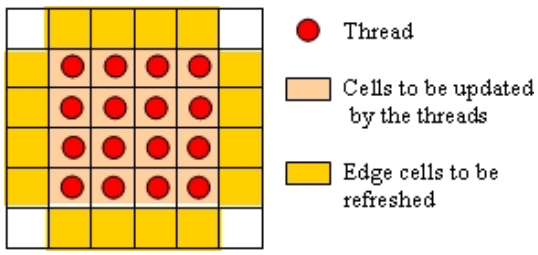


Figure 3.3: Assignment of a subdomain of 4×4 to a CUDA block. Threads work on the inner cells but need extra data to represent the borders of the subdomain (ghost cells).

user can have multiple sub-domains per multi-processor, CUDA takes care of this

step automatically. However, one of the consequences of this feature is that there is no global synchronization of threads from different blocks. This is a performance issue for numerical methods that requires time marching. One needs to synchronize threads after each time step to make sure that the ghost cells are updated. One way to address the issue is to use a multi-pass approach, where a kernel is launched at every time step. This introduces an overhead due to kernel load and, for a shared memory implementation, an overhead due to data transfer between the shared and the global memory. Each block needs to copy its subdomain from the global memory to the shared memory before computation and from the shared memory to the global memory after computation. This last step is required as shared memory data is lost when a new kernel is launched. An alternative approach is to update the ghost cells only every N iteration using a pyramidal approach as explained in [13]. This method reduces the multi-pass approach overhead by having the blocks working on overlapping data. Tests showed that his approach could offer a maximum speedup of 6.5× when compared with a Pentium 4 CPU for a 2D problem [13]. This might be acceptable but the quantity of overlapping data for a 3D application would make it unpractical for large 3D problems [13]. Additionally, it implies that the updated status of the domain is needed only every N iterations. Despite the overhead due to kernel load and memory transfer at every time step, the multi-pass recipe seems more appropriate than a pyramidal approach as the final application presented in this thesis is a large 3D simulation problem.

3.1.4 Dual-GPU Implementation

As mentioned earlier, CUDA does not provide any API to handle multiple GPUs. The multi- GPU implementation presented here uses POSIX multi-threading. Advanced

frameworks like MPI could have been used but POSIX threading assures that the communication overhead of the dual GPU implementation is minimized. OpenMP [12] is also an option with extra features for general parallel computing on shared memory platforms. In the present implementation, the parallelization strategy needs to handle thread operations such as *synchronize*, *create* and *kill*, for which it suffices to use POSIX multi-threading. In dual-GPU implementation, the kernel is almost the same as the one used for the single-GPU implementation. On the other hand, the host code needs to be modified to handle the domain decomposition and multi-threading. Each GPU is responsible for half of the domain (horizontal stripe). One CPU thread is assigned to each GPU. First, it initializes the memory on its device, and then iterate through the time steps by launching the kernel as many times as necessary. After each iteration, the two GPUs need to exchange ghost cells at the domain decomposition boundary. The CPU threads synchronize using a barrier once the edge cells are copied from the device to the host memory. The device memory is then updated and a new iteration starts.

3.2 Implementation of a 3D Incompressible Navier-Stokes Solver

3.2.1 Lid-Driven Cavity Problem

The lid-driven cavity problem described in Figure 3.4 is a well-established benchmark case in the CFD field [23] and can be used as a validation case to check the correct implementation of the Navier-Stokes equations, because there is no net mass and momentum transport across the domain. The lid-driven cavity is a cubic container filled with a fluid. Its lid moves at constant velocity U_{lid} (Figure 3.4) and drives the

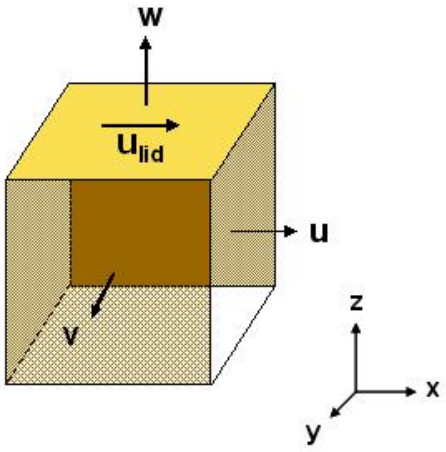


Figure 3.4: A schematic of the physical domain for the lid-driven cavity problem. No-slip conditions are applied on the YZ planes in the east and west directions and on the bottom XY plane in the south direction. Free-slip (symmetry) condition is applied to the front and back XZ planes. A constant velocity is applied on the XY plane in north direction. The velocity component in the x-direction is set to a constant U_{lid} value.

fluid inside the container. The flow is assumed to be laminar in this problem. The viscosity is then considered constant and the first-order Euler scheme (Eq. 2.9) is used for time integration. The Reynolds number for this problem is $Re = U_{lid} \times L/\nu$, where ν is the viscosity of the fluid and L is the height of the cube.

3.2.2 Single-GPU Implementation

Domain Decomposition and Thread Assignment

Let NX, NY and NZ be the number of computational nodes in the x, y and z directions for a flow domain, respectively. The 3D domain of size $NX \times NY \times NZ$ is represented by a 2D matrix of width NX and height $NY \times NZ$ on the host side, as shown in Figure 3.5. On the GPU side, the same representation is used to store data in global memory. This 2D mapping translates to efficient data transfer between the

host (CPU) and the device (GPU). Note that several matrices are needed to represent the pressure and velocity components at different time levels. Memory allocation on the device is done only once before starting the time stepping. Figure 3.6 gives the

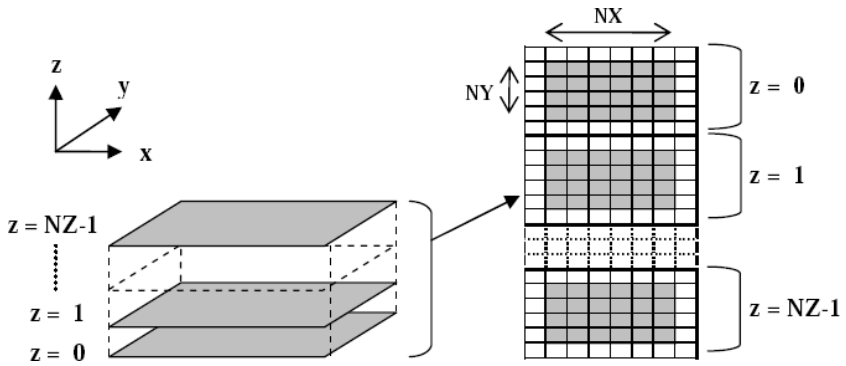
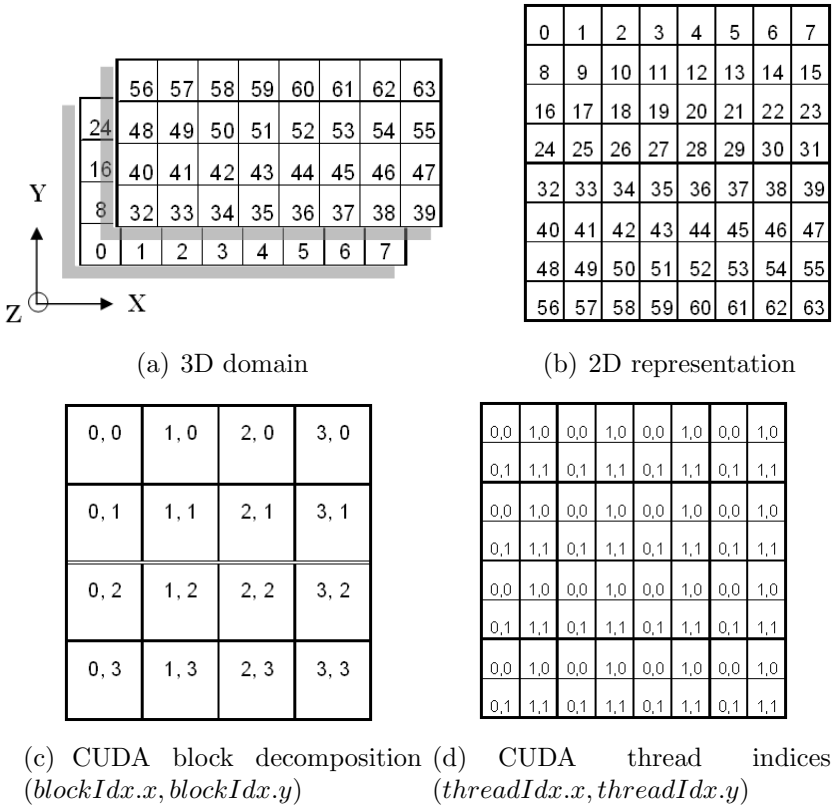


Figure 3.5: Mapping of a 3D computational domain to a 2D matrix. The mapping is used on both the CPU and the GPU sides. Cells in white on the 2D matrix represent the ghost (halo) cells to apply the boundary conditions.

logic used to access data from the GPU. In this example a 2D CUDA grid of 4×4 blocks is mapped onto a 3D domain of $8 \times 4 \times 2$. Each block is a set of 2×2 threads, each thread being mapped to a cell of the domain. The 3D domain is actually represented by a 1D array in the device memory. The corresponding indices of each domain cell are given in Figure 3.6(a) and 3.6(b). The translation from 3D to 1D is defined by:

$$A\left[i + j \times NX + k \times NX \times NY\right] = B\left[k\right]\left[j\right]\left[i\right],$$

where A is a 1D array with $(NX \times NY \times NZ)$ elements and B a 3D matrix of dimension $(NX \times NY \times NZ)$. If the size of the domain is $(NX \times NY \times NZ)$, then the index translation from the thread ID (in a 2D CUDA grid) to a 1D array element index in global memory is given by:



(a) 3D domain (b) 2D representation

(c) CUDA block decomposition $(blockIdx.x, blockIdx.y)$ (d) CUDA thread indices $(threadIdx.x, threadIdx.y)$

Figure 3.6: Example of index logic to map a 2D CUDA block decomposition onto a 3D domain. The 3D domain (a) is represented in a 2D-way (b). A 2D CUDA grid is then mapped onto the 2D domain (c). (d) represents the thread indices associated to the CUDA block decomposition.

$$\begin{aligned} I &= gridDim.x \times blockDim.x \times blockDim.y \times blockIdx.y \\ &+ threadIdx.y \times gridDim.x \times blockDim.x \\ &+ blockIdx.x \times blockDim.x + threadIdx.x \end{aligned}$$

where I is the index used to access the global memory (1D array).

$(blockDim.x, blockDim.y)$ represents the dimensions of a single CUDA block and $(blockIdx.x, blockIdx.y)$ the ID (2D coordinates) of the block to which the current thread belongs to. The ID of the thread inside the block is given by $(threadIdx.x, threadIdx.y)$.

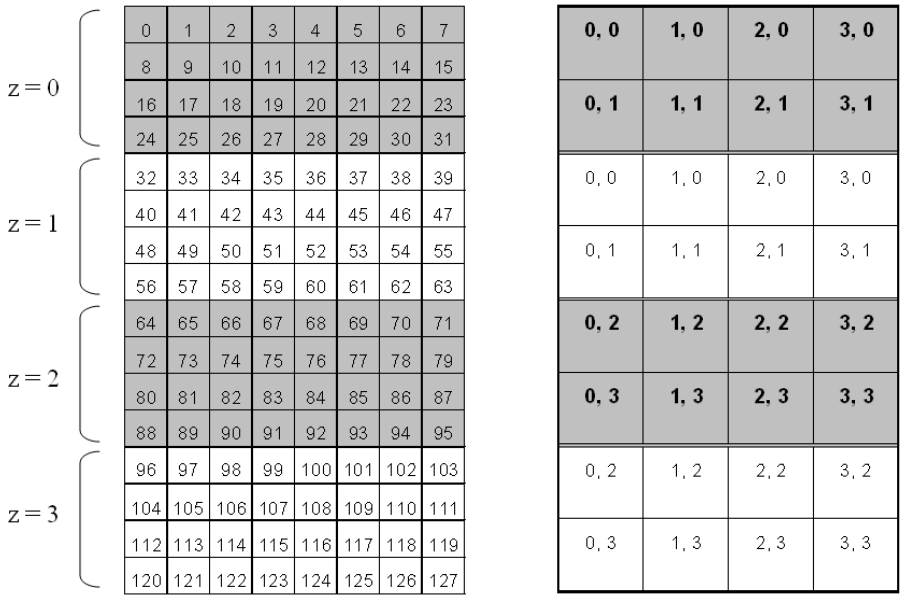
Using the example given in Figure 3.6 and Eq. 3.1, thread (0,1) from block (0,3) updates the 1D array element at the index I defined by:

$$\begin{aligned} I &= 4 \times 2 \times 2 \times 3 \\ &+ 1 \times 4 \times 2 \\ &+ 0 \times 2 + 0 \\ I &= 56 \end{aligned}$$

Using this index logic, each thread becomes responsible for one computational node of the domain. As discussed in Section 3.2.4, in some cases it might be interesting to have one thread responsible for multiple grid cells. In Figure 3.7, a 2D CUDA grid of 4×4 blocks and 2×2 threads per block is mapped onto the 3D domain. Each thread block is responsible for 2 subdomains aligned in the z-direction. For this approach, the size of the CUDA grid is defined by:

$$(gridDim.x, gridDim.y) = (GRID\_SIZE\_X, GRID\_SIZE\_Y \times SIZE\_Z), \quad (3.1)$$

where GRID_SIZE_X is the number of blocks in the x-direction, GRID_SIZE_Y is the number of blocks in the y-direction necessary to represent a slice of the domain in the XY plane, and SIZE_Z is the number of levels in the z-direction on which a single block work. The index of the first element on which a given thread should work is defined by:



(a) 2D representation of the memory (b) Mapping of the CUDA grid

Figure 3.7: Final index logic to map the CUDA block decomposition onto a 3D domain. (a) An $8 \times 4 \times 4$ domain is stored as a 1D array in memory. (b) A 2D CUDA grid is mapped onto the 1D array in memory, each 2×2 thread block working on two levels in the z-direction.

$$\begin{aligned} I_{temp} &= gridDim.x \times blockDim.x \times blockDim.y \\ &\times (blockIdx.y \% GRID\_SIZE\_Y) \\ &+ threadIdx.y \times gridDim.x \times blockDim.x \\ &+ blockIdx.x \times blockDim.x + threadIdx.x \\ I &= I_{temp} + (NX \times NY \times SIZE\_Z) \times \lfloor blockIdx.y / GRID\_SIZE\_Y \rfloor \end{aligned} \quad (3.2)$$

In the example given in Figure 3.7, the size of the CUDA grid defined by Eq. 3.1 becomes:

$$(gridDim.x, gridDim.y) = (4, 2 \times 2) = (4, 4)$$

In this configuration the thread (1,0) from the block (1,2) would start working with the element located at the index I calculated as follows:

$$\begin{aligned} I_{temp} &= (4 \times 2 \times 2 \times (2\%2)) + (0 \times 4 \times 2) + (1 \times 2) + 1 \\ I &= I_{temp} + (8 \times 4 \times 2) \times \lfloor 2/2 \rfloor \\ I &= 67 \end{aligned}$$

Thread (1,0) from block (1,2) is then repsonsible for the array elements at the indices 67 and $(67 + NX \times NY) = (67 + 8 \times 4) = 99$. The code for the index logic is given in Appendix C.2.

In each kernel, only one thread updates the value of a given compuational node in global memory. Having a CUDA block configuration where the number of threads is a multiple of 16 allows fast coalesced scatter operations. Coalesced access allow multiple threads to read or write to the global memory in a single memory transaction. Good usage of coalescing can hide the memory latency. Figure 3.8 depicts data access in a coalesced fashion, where threads access words in sequence. In this implementation, coalesced access is performed whenever the thread defined by $(threadIdx.x, threadIdx.y)$ and $(blockIdx.x, blockIdx.y)$ accesses global memory at the index I defined earlier. As the update of a computational node usually requires knowledge on the neighboring values, gather operations cannot always be coalesced. The right pattern presented in Figure 3.9 shows how the thread would access the global memory if the east neighboring value had to be read. With the architecture of the GPUs utilized in this research (compute capability 1.0), this would result in a non-coalesced gather. With the latest NVIDIA hardware (compute capability 1.2 and higher), the rules for coalesced access have been relaxed. Coalescing can now be

performed even if multiple threads access the same address or if words in memory are not accessed in sequence (with some limitations [38]). Figure 3.10 shows a couple examples of coalesced access patterns for devices of compute capability 1.2 or higher. The kernels using only global memory would definitely benefit from a new generation GPU as threads use the same shifting to read a global memory value at a given time. For a shared memory implementation, the gain in performance would probably not be so visible as most of the non-coalesced memory accesses are avoided by loading the necessary data into the shared memory. The inner cells represented in shared memory are loaded in a coalesced way as each thread copies one value from global memory. On the other hand part of the ghost cells need to be loaded in a non-coalesced way, wether a new generation GPU is considered or not.

Main Code (Host-side)

Figure 3.11 shows the host side code for the time stepping. The code snippet is composed of two nested loops. The outer loop is used to advance the solution in time, and the inner loop is used for the iterations of the Jacobi solver to numerically solve the pressure Poisson equation (Eq. 2.11). With the first order Euler scheme, the velocity field at time t depends only on the velocity field at $t-1$. Six different matrices are used to represent the velocity fields at the time t (`u`, `v`, `w`) and $t-1$ (`uold`, `vold`, `wold`). The matrices are swapped at the end of each time step for reuse as shown in Figure 3.11. In a similar way, the Jacobi solver requires two matrices p and pold, which are swapped after each iteration of the Jacobi solver. As shown in Figure 3.11 the GPU code is composed of six different kernels to implement the major steps of the projection algorithm [14]. Separate kernels are needed to achieve global synchronization across the CUDA blocks before proceeding to the next time

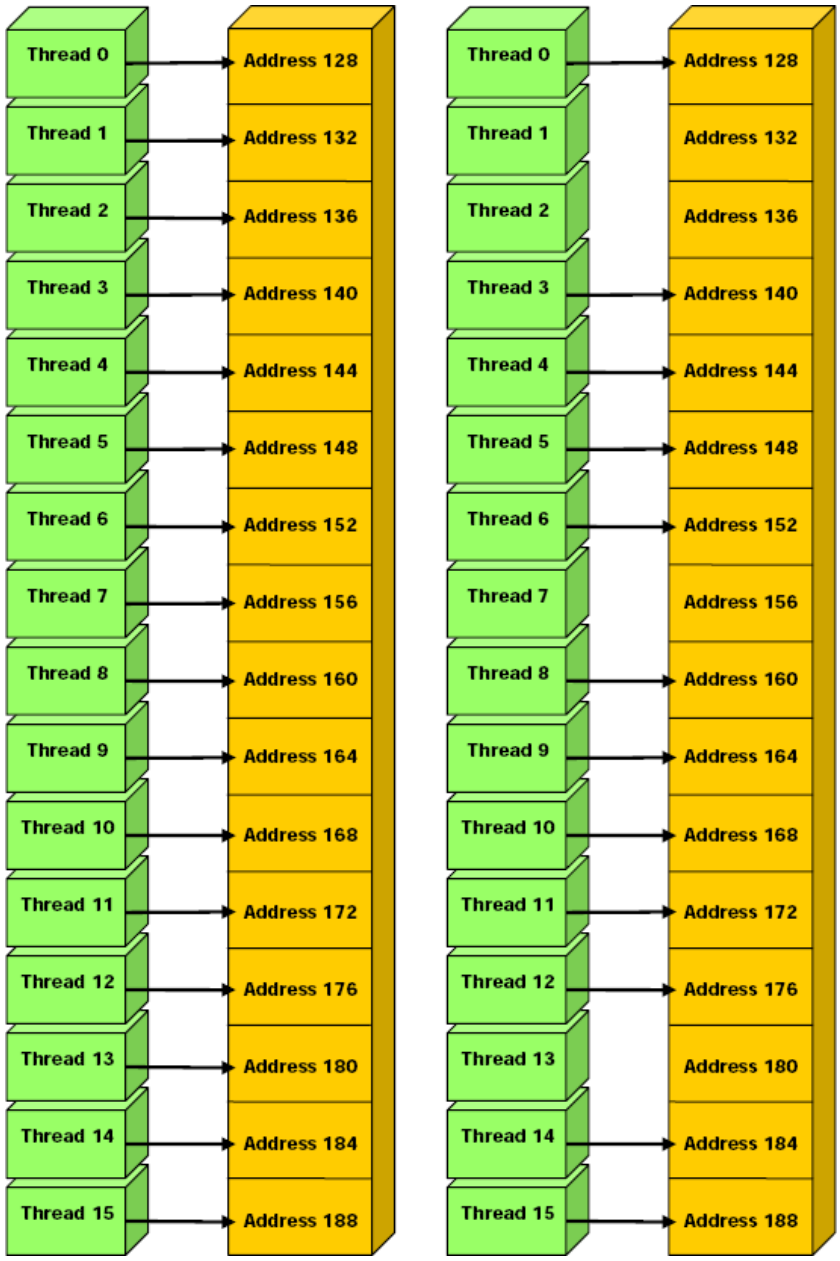


Figure 3.8: Examples of coalesced global memory access patterns (courtesy of NVIDIA). Left: coalesced `float` memory access, resulting in a single memory transaction. Right: coalesced `float` memory access (divergent warp), resulting in a single memory transaction.

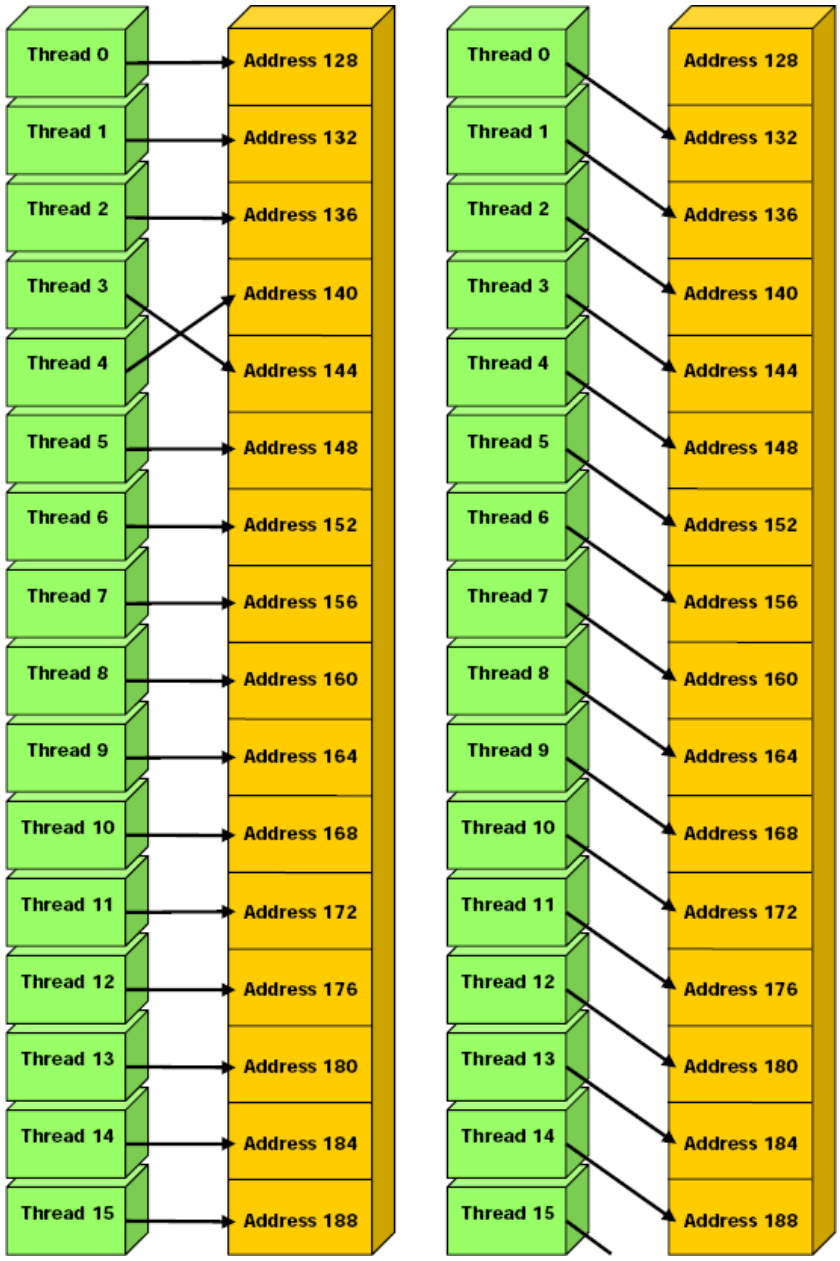


Figure 3.9: Examples of global memory access patterns that are non-coalesced for devices of compute capability 1.0 or 1.1 (courtesy of NVIDIA). Left: non-sequential `float` memory access, resulting in 16 memory transactions. Right: access with a misaligned starting address, resulting in 16 memory transactions.

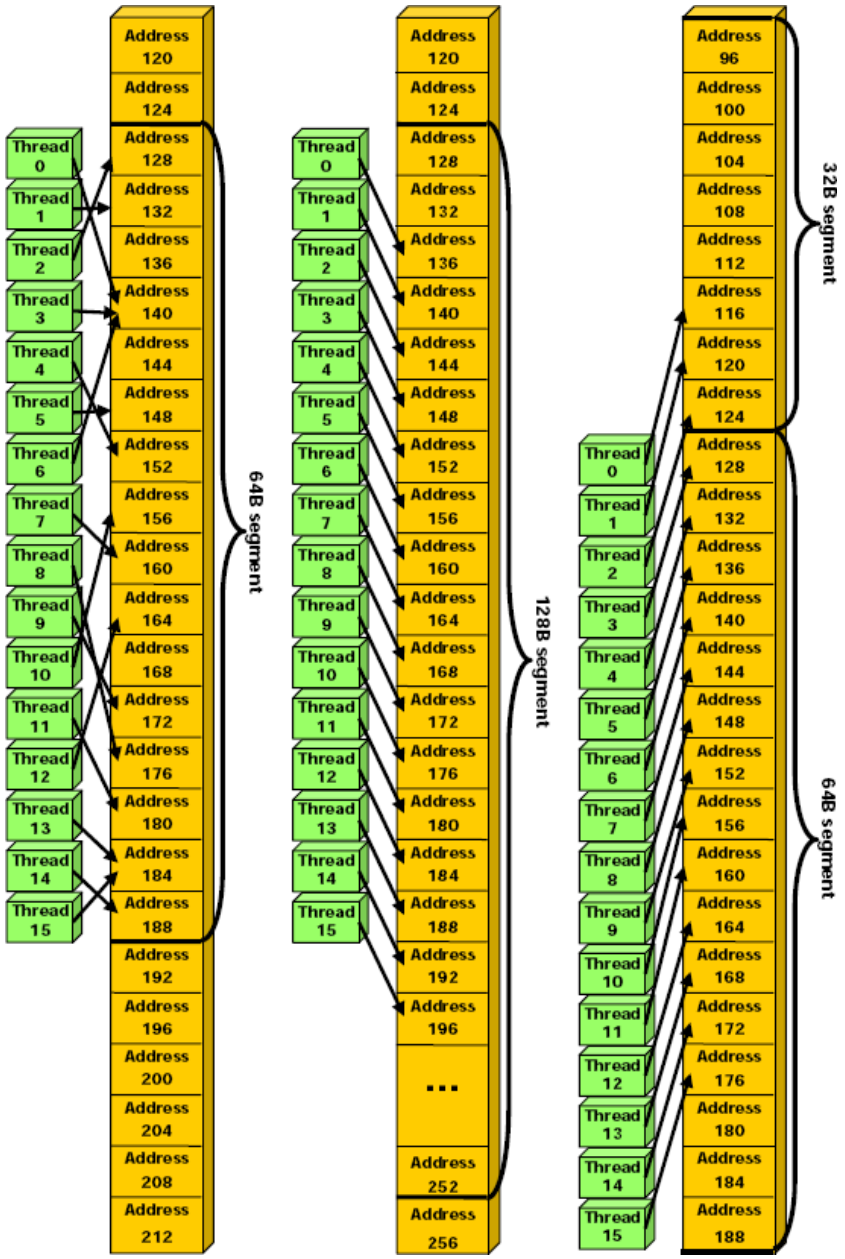


Figure 3.10: Examples of global memory access by devices with compute capability 1.2 and higher (courtesy of NVIDIA). Left: random `float` memory access within a 64B segment, resulting in one memory transaction. Center: misaligned `float` memory access, resulting in one transaction. Right: misaligned `float` memory access, resulting in two transactions.

step for the computations. The code is composed of two nested loops. The outer loop is used for time marching to advance the solution in time, and the inner loop is used for the iterations of the Jacobi solver to numerically solve the pressure Poisson equation. In the current implementation, the velocity field at time t depends only on the velocity field at $t-1$ (explicit time marching scheme). Six different matrices are used to represent the velocity fields at the time t (u, v, w) and $t-1$ (u_{old}, v_{old}, w_{old}). Using the same procedure as for the wave equation implementation, the matrices are swapped at the end of each time step as shown in the code snippet given in Figure 3.11. In a similar way, the Jacobi solver requires two matrices p and p_{old}, which are swapped after each iteration of the Jacobi solver. The GPU code is composed of six different kernels as shown in Figure 3.11), the existence of each being justified by the necessity of a global synchronization before proceeding to the next step of the computation. The implementation of the different kernels is given in the Appendix C.

3.2.3 Multi-GPU Implementation

In the multi-GPU implementation, each GPU is responsible for a subdomain of size $NX \times NY \times (NZ/number\_of\_GPUs)$, as shown in Figure 3.12(a) The whole domain is represented on the host side while the GPUs only store their respective subdomains in global memory, and the ghost cells used to update the cells at the bottom and the top of the subdomain. As shown in Figure 3.12(b), $2 \times NX \times NY$ ghost cells need to be filled with data from the GPUs responsible for the top and bottom neighboring subdomains. At the GPU level, the subdomain is mapped to a 2D CUDA grid the same way it was for the single-GPU implementation. With the domain decomposition shown in Figure 3.12(a), each GPU needs neighboring data computed by other GPUs which means all GPUs need to synchronize to exchange velocity and pressure fields

```
//for each time step
for (t=0; t < ntstep; t++)
{
    //call kernel to compute momentum (ut, vt, wt)
    momentum << grid, block >> (u, v, w, uold, vold, wold)
    //call kernel to compute boundary conditions
    momentum_bc << grid, block >> (u, v, w)
    //call kernel to compute the divergence (div)
    divergence << grid, block >> (u, v, w, div)
    //for each Jacobi solver iteration
    for (j=0; j < njacobi; j++)
    {
        //call kernel to compute pressure
        pressure << grid, block >> (u, v, w, p, pold, div)
        //rotate matrices
        ptemp = pold; pold=p; p=ptemp;
        //call kernel to compute boundary conditions
        pressure_bc << grid, block >> (p)
    }
    //call kernel to correct velocity (ut, vt, wt)
    correction << grid, block >> (u, v, w, p)
    //call kernel to compute boundary conditions
    momentum_bc << grid, block >> (u, v, w)
    //swap pointers
    utemp = uold; uold=u; u=utemp;
    vtemp = vold; vold=v; v=vtemp;
    wtemp = wold; wold=w; w=wtemp;
}
```

Figure 3.11: Partial host-side code that implements the projection algorithm [14] to solve the Navier-Stokes equations for incompressible fluid flow. The outer loop is used for time stepping while the inner loop is in the iterative solution of the pressure Poisson equation.

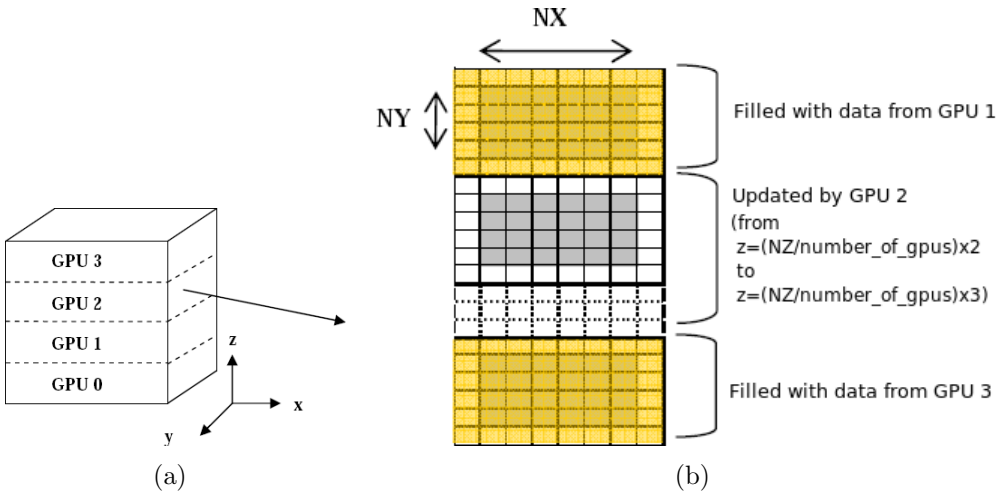


Figure 3.12: a) Subdomain assignment for multi-GPU solution. b) Representation of the GPU global memory. Each GPU needs ghost cells to represent the top and bottom neighboring cells which are updated by other GPUs (represented here in red).

at each time step. But a GPU cannot directly exchange data with another GPU. Hence, ghost cells at the multi-GPU domain decomposition boundaries needs to be copied back to the host, which adds an extra communication overhead to the overall computation in addition to the CUDA kernel launches at every time step. As mentioned earlier, multi-GPU parallelism is not currently addressed by CUDA. One CPU thread is assigned to each GPU so that each device has its own context on the host. Figure 3.13 shows the host side code snippet for the multi-GPU implementation of the projection algorithm. Each CPU thread executes the code given in Figure 3.13. First the GPUs copy the top and bottom cells of their subdomains from their global memory to a matrix on the host side. After the GPUs are synchronized using a POSIX barrier (`pthread_barrier_wait`), the GPUs read from the host-side matrix data that represent their ghost cells and update their global memory. For the velocity field, this process happens twice per time step, once after the solution of the momentum

```
for (t=0; t < steps; t++)
{
    //copy velocity ghost cells from host to GPU
    ...
    momentum<<<grid,block>>>(u,v,w,uold,vold,wold,ngpus,*device);
    momentum_bc<<<grid,block>>>(u,v,w,ngpus,*device);
    //copy velocity border cells from GPU to host memory
    ...
    //synchronize with other GPUs before reading
    pthread_barrier_wait(&barrier);
    //copy velocity ghost cells from host to GPU
    ...
    divergence <<< grid, block >>>(u,v,w, div,ngpus,*device);
    //for each Jacobi solver iteration
    for(m = 0; m< njacobi; m++)
    {
        pressure <<<grid,block>>>(div,pold,p,ngpus,*device);
        ptemp = pold; pold=p; p=ptemp;
        pressure_bc <<<grid,block >>>(d_p,ngpus,*device);
        //copy pressure border cells from GPU to host memory
        ...
        //synchronize with other GPUs before reading
        pthread_barrier_wait(&barrier);
        //copy pressure ghost cells from host to GPU
        ...
    }
    correction <<<grid,block>>>(u,v,w,p,ngpus,*device);
    momentum_bc<<<grid,block>>>(u,v,w,ngpus,*device);
    //copy velocity border cells from GPU to host memory
    ...
    //synchronize with other GPUs before reading
    pthread_barrier_wait(&barrier);
    //rotate matrices
    utemp = uold; uold=u; u=utemp;
    vtemp = vold; vold=v; v=vtemp;
    wtemp = wold; wold=w; w=wtemp;
}
```

Figure 3.13: Partial host-side code that implements the projection algorithm [14] to solve the Navier-Stokes equations for incompressible fluid flow. The outer loop is used for time stepping while the inner loop is in the iterative solution of the pressure Poisson equation. A CPU thread is created for each available GPU and executes the code above. Synchronization between the CPU threads is done through a Posix *barrier*.

equations and once after the correction step. For the pressure field, data exchange occurs after each Jacobi solver iteration.

3.2.4 GPU Shared Memory Implementation

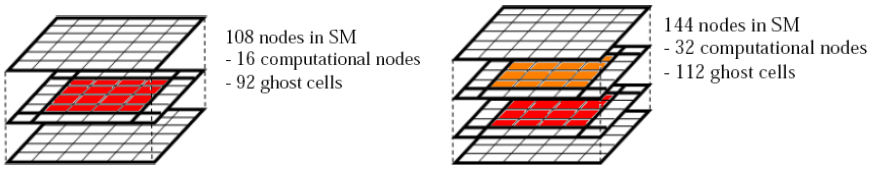


Figure 3.14: Two different approaches for shared memory usage in a 4×4 block configuration. Colored cells are updated by the threads while the white cells are only used as data source (ghost cells). Each cell center represents a computational node. a) Each thread updates one cell only (red cells). b) Each thread works on 2 cells in the same vertical column. Cells in red are updated during the first iteration and orange ones in the second iteration.

Usage of the shared memory (SM) in a kernel is a three-step process. First, the block threads copy the subdomain they are responsible for from the global memory to the shared memory. Then computation is done by the threads, using data from the shared memory. Finally the result of the computation is written back to the global memory before exiting the kernel. This back and forth data transfer between the global memory to the shared memory creates an overhead that is not present in a global memory implementation. Hence, the arithmetic intensity of the kernel should be sufficiently large to compensate for the overhead of data copying in order to benefit from the shared memory implementation. One way to achieve this is to increase the size of the subdomain that is mapped to a thread block. Figure 3.14 compares two different domain decompositions where each block contains 4×4 threads. In the first one (Figure 3.14(a)), the block is directly mapped to a subdomain of 4×4

computational nodes. In order to update those computational nodes, the subdomain and all its surrounding nodes (ghost cells) need to be copied to the shared memory. To update 4×4 cells, $6 \times 6 \times 3$ cells actually need to be copied to the shared memory. In which case, less than 15% of the shared memory will be updated by the thread block. The second approach shown in Figure 3.14(b) allows threads to update multiple cells in a distinct vertical column. In this example each thread works on two cells (one in the red plane and in the orange plane). The threads are now working on $4 \times 4 \times 2$ cells and $6 \times 6 \times 4$ cells are required in total. The cells to be updated now represent 22% of the data brought to the shared memory. This can be easily implemented by having a for loop iterating in the z-direction for each thread. Notice that the number of iterations in the z-direction is known in advance as it is defined by the programmer. The directive `#pragma unroll` can be used to unroll the *for* loop. Then the cost of the for loop is not critical while the mapping shown in Figure 3.14(b) reduces the amount of time spent in transferring data from the global to the shared memory. As the size of the block and the number of cells to update per thread increase, the overhead due to data copying to the shared memory is compensated by the time spent on actual computations. For the current shared memory implementation, each block works on two different levels in the XY-plane. The size of the shared memory being limited to 16 KB, no more than four levels (two inner levels and two ghost levels) can be copied into the shared memory if the block size is 16×16. Note that if the shared memory gets too large, few threads can be created at the same time as less registers are available [42]. An alternative implementation would be to have fewer threads per block but more levels for each thread to work. Further tests will give us more insight on the optimal configuration. The way the indices are computed to access global memory and shared memory with this technique is given in Appendix C.2.

3.2.5 Validation

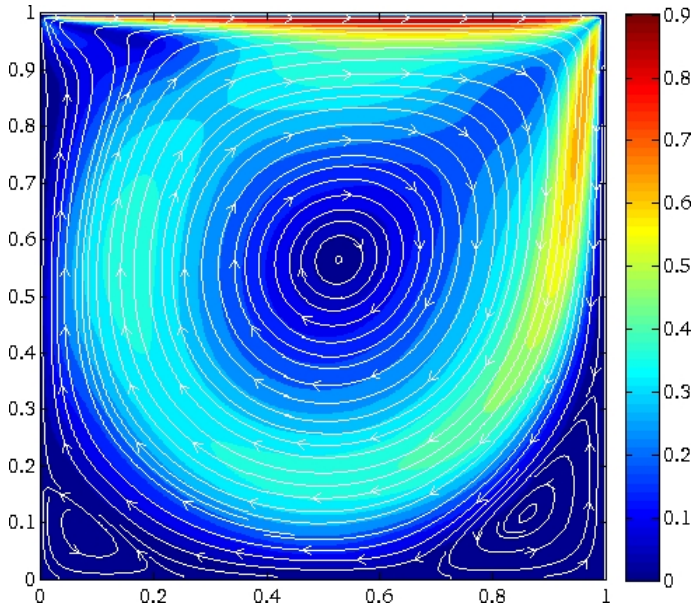


Figure 3.15: Distribution of velocity magnitude and streamlines at steady-state for Re=1000. Low velocity regions are represented in dark blue while high velocity regions are represented in red.

To validate the implementation, the GPU solution output was compared to numerical data from Ghia et al. [23] Figure 3.16 shows that the present results obtained from the GPU code are in excellent agreement with the results of [23]. Figure 3.15 shows the velocity streamlines at steady-state. The flow structure inside a cavity for various Reynolds numbers is well established. Any mistake in the implementation can be quickly detected by inspecting the streamlines and the distribution of the velocity field. For $Re = 1000$, one should observe a main circulation at the core of the cavity, and smaller recirculation zones at the bottom corners (Figure 3.15). The size of these corner vortices increases with the Reynolds number.

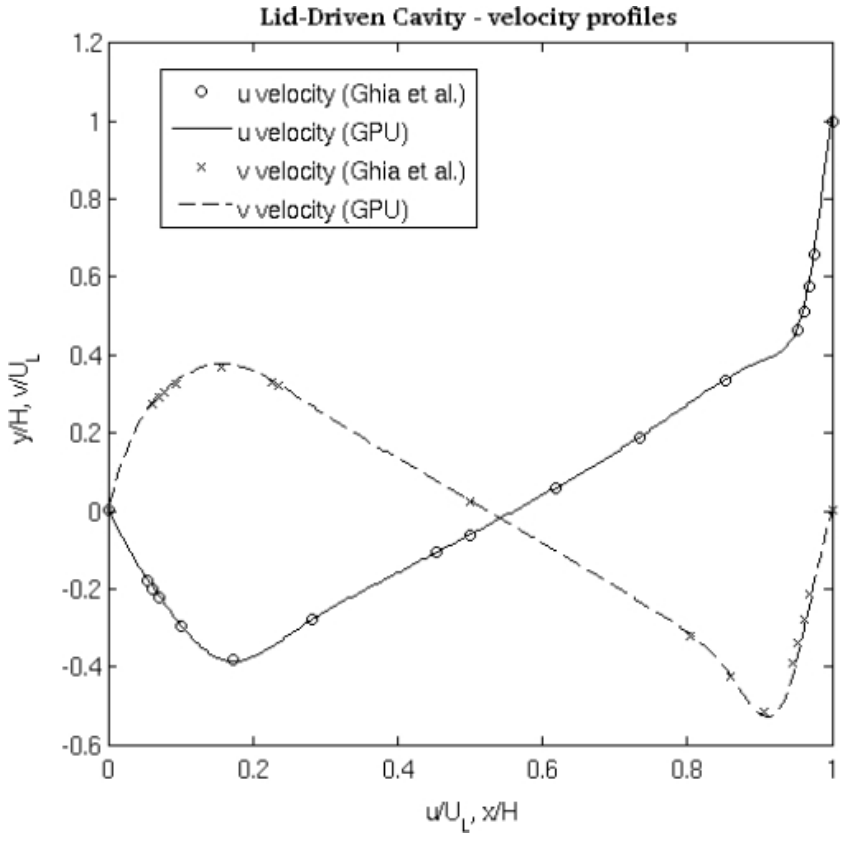


Figure 3.16: Validation of the GPU code results with benchmark data given in Reference [23]. Both u and v components of the velocity field are shown.

3.3 Complex Geometry Capability

3.3.1 Additional Features

In order to compute flows in urban environments, new features had to be added to the Navier-Stokes solver. A geometry capability was added to handle building effects in the domain. To provide an accurate simulation with turbulent flow, first the momentum kernel was modified to use the second-order Adams-Bashfort scheme (Eq 2.10) instead of the first-order Euler scheme (Eq. 2.9). Second, an obstacle logic was implemented to impose boundray conditions on the buildings. Finally, a new

kernel was added to compute the turbulent viscosity defined in Eq. 2.6.

3.3.2 Obstacle Logic

To represent obstacles in the domain, a new logic must be applied to impose boundary conditions on the pressure and the velocity fields. The obstacles are represented by a flag matrix of ones and zeros, *ones* representing obstacles and *zeros* representing open spaces. Figure 3.17 is a 2D example of this representation. The pressure gradient

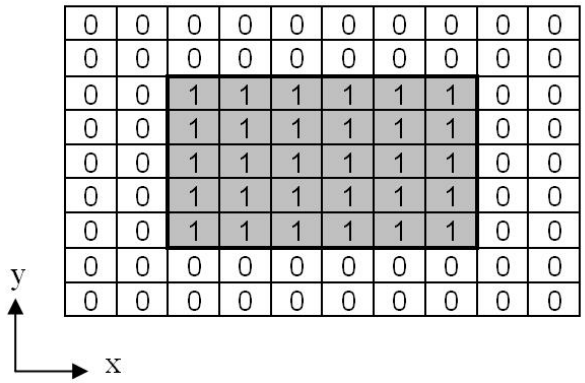


Figure 3.17: Flag matrix used to represent the obstacles at the pressure points. The gray cells (1's) represent a building.

at the obstacle boundary should be set to zero. This can be easily achieved using a mask. This logic is applied directly in the pressure kernel on top of the Poisson equation:

$$
\begin{aligned}
B &= (F_{i+1,j,k} + F_{i-1,j,k} - 2) \times \frac{1}{dx^2} \\
&+ (F_{i,j+1,k} + F_{i,j-1,k} - 2) \times \frac{1}{dy^2} \\
&+ (F_{i,j,k+1} + F_{i,j,k-1} - 2) \times \frac{1}{dz^2} \\
A &= \frac{(F_{i+1,j,k} - 1)P^n_{i+1,j,k} + (F_{i-1,j,k} - 1)P^n_{i-1,j,k}}{dx^2} \\
&+ \frac{(F_{i,j+1,k} - 1)P^n_{i,j+1,k} + (F_{i,j-1,k} - 1)P^n_{i,j-1,k}}{dy^2} \\
&+ \frac{(F_{,j,ki+1} - 1)P^n_{i,j,k+1} + (F_{i,j,k-1} - 1)P^n_{i,j,k-1}}{dz^2} \\
P^{n+1}_{i,j,k} &= \frac{1}{B} \times (\frac{1}{dt} div(i,j,k) + A),
\end{aligned}
\tag{3.3}
$$

where F represents the flag matrix (obstacles). A new logic should also be applied to the velocity field to assure a null velocity inside the buildings and at the walls. Figure 3.18 describes the logic used to assure a correct U-velocity in the XZ plane. First the velocity inside the building is set to zero using the following mask:

$$
\begin{aligned}
U_{i,j,k} &= (F_{i,j,k} - 1) \times (F_{i+1,j,k} - 1) \times U_{i,j,k} \\
V_{i,j,k} &= (F_{i,j,k} - 1) \times (F_{i,j+1,k} - 1) \times V_{i,j,k} \\
W_{i,j,k} &= (F_{i,j,k} - 1) \times (F_{i,j,k+1} - 1) \times W_{i,j,k}
\end{aligned}
\tag{3.4}
$$

Note that the walls of the buildings follow the cell edges. Because of the staggered grid configuration, each velocity component is not coincident with the walls. For example, the U-velocity in the XY plane is represented at the vertical walls but not at the horizontal ones (Figure 3.18(a)). To force a null velocity at the horizontal walls it is necessary to sweep the velocity vectors inside the building (Figure 3.18(b)). The velocity at the wall is then

$$\begin{aligned} U_{wall} &= U_{ext} - U_{int} \\ &= U_{ext} - (-U_{ext}) \\ &= 0 \end{aligned} \tag{3.5}$$

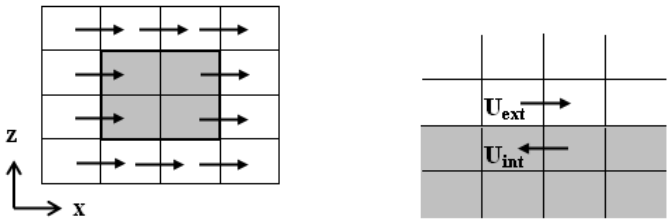
The logic used to sweep the vectors representing the U component of the velocity relative to the z-direction (horizontal walls) is presented below:

$$\begin{aligned} U_{i,j,k} &= F_{i,j,k} \times \\ &\quad [(F_{i,j,k+1} - 1) \times U_{i,j,k+1} + (F_{i,j,k-1} - 1) \times U_{i,j,k-1} \\ &\quad + (F_{i,j,k+1} + F_{i,j,k} + F_{i,j,k-1} - 2) \times U_{i,j,k}] \\ &\quad + (1 - F_{i,j,k}) \times U_{i,j,k} \end{aligned} \tag{3.6}$$

As the studied domain is 3-D, each velocity component has to be updated twice using this logic. The same logic is applied to U in the y-direction, to V in the x and z directions and to W in the x and y directions. The logic used to sweep the vectors representing the V and W components of the velocity relative to the i-direction (vertical walls) is presented below:

$$\begin{aligned} V_{i,j,k} &= F_{i,j,k} \times \\ &\quad [(F_{i+1,j,k} - 1) \times V_{i+1,j,k} + (F_{i-1,j,k} - 1) \times V_{i-1,j,k} \\ &\quad + (F_{i+1,j,k} + F_{i,j,k} + F_{i-1,j,k} - 2) \times V_{i,j,k}] \\ &\quad + (1 - F_{i,j,k}) \times V_{i,j,k} \end{aligned} \tag{3.7}$$

$$
\begin{aligned}
W_{i,j,k} &= F_{i,j,k} \times \\
&\quad [(F_{i+1,j,k} - 1) \times W_{i+1,j,k} + (F_{i-1,j,k} - 1) \times W_{i-1,j,k} \\
&\quad + (F_{i+1,j,k} + F_{i,j,k} + F_{i-1,j,k} - 2) \times W_{i,j,k}] \\
&\quad + (1 - F_{i,j,k}) \times W_{i,j,k}
\end{aligned} \tag{3.8}
$$



(a) Staggered configuration (b) Vector sweeping

Figure 3.18: Obstacle logic applied to the U-component of the velocity in the XZ plane

The corners of the building need to be treated separately, after the vector sweeping process. For each velocity component the corners are set to zero. The corners to treat are located using the following logic:

U-velocity (logic applied in the z-direction and j-direction)

$$
\begin{aligned}
G^U_{i,j,k} &= F_{i,j,k} + F_{i,j,k+1} + F_{i,j,k-1} + F_{i+1,j,k} + F_{i+1,j,k+1} + F_{i+1,j,k-1} + F_{i-1,j,k} \\
&+ F_{i,j,k} + F_{i,j+1,k} + F_{i,j-1,k} + F_{i+1,j,k} + F_{i+1,j+1,k} + F_{i+1,j-1,k} + F_{i-1,j,k}
\end{aligned} \tag{3.9}
$$

V-velocity (logic applied in the z-direction and i-direction)

$$\begin{aligned} G^V_{i,j,k} &= F_{i,j,k} + F_{i,j,k+1} + F_{i,j,k-1} + F_{i,j+1,k} + F_{i,j+1,k+1} + F_{i,j+1,k-1} + F_{i,j-1,k} \\ &+ F_{i,j,k} + F_{i+1,j,k} + F_{i-1,j,k} + F_{i,j+1,k} + F_{i+1,j+1,k} + F_{i-1,j+1,k} + F_{i,j-1,k} \end{aligned} \tag{3.10}$$

W-velocity (logic applied in the i-direction and j-direction)

$$\begin{aligned} G^W_{i,j,k} &= F_{i,j,k} + F_{i+1,j,k} + F_{i-1,j,k} + F_{i,j,k+1} + F_{i+1,j,k+1} + F_{i-1,j,k+1} + F_{i,j,k-1} \\ &+ F_{i,j,k} + F_{i,j+1,k} + F_{i,j-1,k} + F_{i,j,k+1} + F_{i,j+1,k+1} + F_{i,j-1,k+1} + F_{i,j,k-1} \end{aligned} \tag{3.11}$$

If ($G^Q_{i,j,k} = 7$ or $G^Q_{i,j,k} = 6$) then $Q_{i,j,k}$ is represented at a corner. $Q_{i,j,k}$ is then set to zero.

As a result, four different kernels are necessary for obstacle capability:

- `obstacles_zero_mask`: forces the velocity to be 0 in closed spaces using a mask
- `obstacles_sweep_uvw_kii`: sweeps U, V and W in the z, x and x directions, respectively
- `obstacles_sweep_uvw_jkj`: sweeps U, V and W in the y, z and y directions, respectively
- `obstacles_corners`: treats the corners of the buildings separately.

The code presented in Figure 3.19 represents the host-side code used for the obstacle logic. The obstacle logic is applied twice in the main code (Figure 3.20), once after the predicted velocity is computed and once after the velocity is corrected.

```
obstacles_zero_mask<<< grid, threads >>>
                        (d_flag, d_unew, d_vnew, d_wnew, d_ut, d_vt, d_wt);
obstacles_sweep_uvw_kii<<< grid, threads >>>
                        (d_flag, d_ut, d_vt, d_wt, d_unew, d_vnew, d_wnew);

//swap matrices
d_utemp = d_unew; d_unew = d_ut; d_ut = d_utemp;
d_vtemp = d_vnew; d_vnew = d_vt; d_vt = d_vtemp;
d_wtemp = d_wnew; d_wnew = d_wt; d_wt = d_wtemp;

obstacles_sweep_uvw_jkj<<< grid, threads >>>
                        (d_flag, d_ut, d_vt, d_wt, d_unew, d_vnew, d_wnew);
obstacles_corners<<< grid, threads >>>
                        (d_flag, d_unew, d_vnew, d_wnew);
```

Figure 3.19: host-side code used for the obstacle logic

3.3.3 Main Code

Figure 3.20 presents the host-side code for the Navier-Stokes solver with geometry capability. The obstacle logic code section is summarized in Figure 3.19.

3.3.4 Multi-GPU Implementation

The multi-GPU implementation uses the same domain decomposition among the GPUs as the one used for the lid-driven cavity problem. As each GPU is responsible for an horizontal slice of the domain, this decomposition does not minimize data exchange in an urban-like domain, where the height is the smallest dimension. To keep the same logic, the problem is flipped so that the height becomes the width. In this problem the inlet velocity was applied to the W-component and boundary conditions were updated.

```
//Time marching
for (t=1; t < ntstep; t++)
{
    //call kernel to compute momentum (ut, vt, wt)
    momentum << grid, block >> (u, v, w, uold, vold, wold)
    //apply obstacle logic
    ...
    //call kernel to compute boundary conditions
    momentum_bc << grid, block >> (u, v, w)
    //call kernel to compute the divergence (div)
    divergence << grid, block >> (u, v, w, div)
    //for each Jacobi solver iteration
    for (j=0; j < njacobi; j++)
    {
        //call kernel to compute pressure
        pressure << grid, block >> (u, v, w, p, pold, div)
        //rotate matrices
        ptemp = pold; pold=p; p=ptemp;
        //call kernel to compute boundary conditions
        pressure_bc << grid, block >> (p)
    }
    //call kernel to correct velocity (ut, vt, wt)
    correction << grid, block >> (u, v, w, p)
    //apply obstacle logic
    ...
    //call kernel to compute boundary conditions
    momentum_bc << grid, block >> (u, v, w)
    //rotate matrices
    utemp = uold; uold=u; u=utemp;
    vtemp = vold; vold=v; v=vtemp;
    wtemp = wold; wold=w; w=wtemp;
}
```

Figure 3.20: Partial host-side code to calculate flow field with 3D obstacles.

3.3.5 Final Output

Surface-Mounted Cube

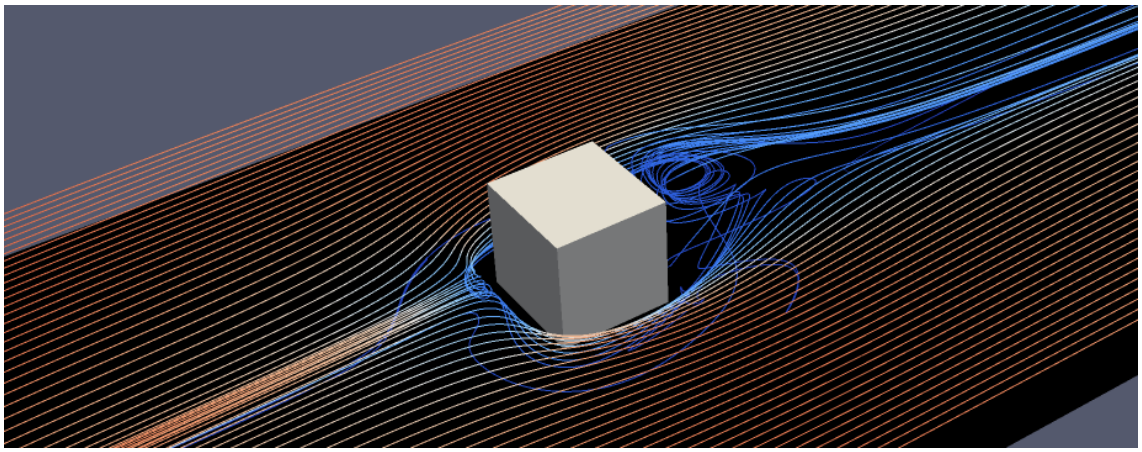
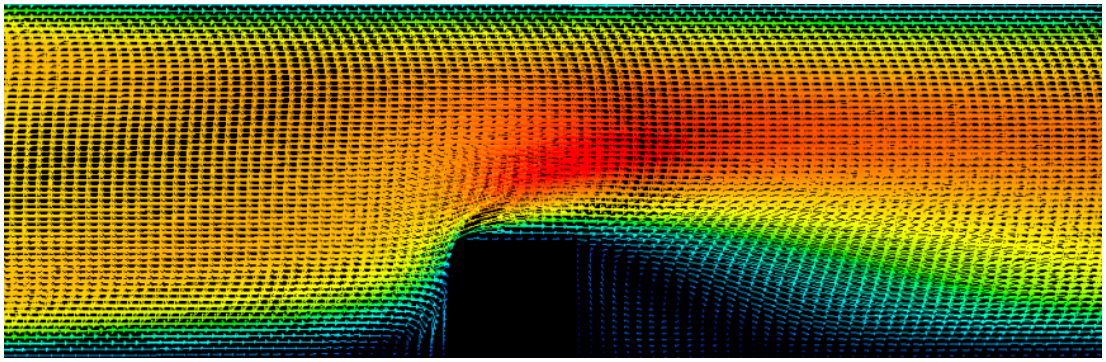


Figure 3.21: Flow around a surface-mounted cube for a laminar regime ($Re = 42$). Grid size is $256 \times 128 \times 64$. Red streamlines represent high velocity magnitudes while blue streamlines represent lower velocity magnitudes

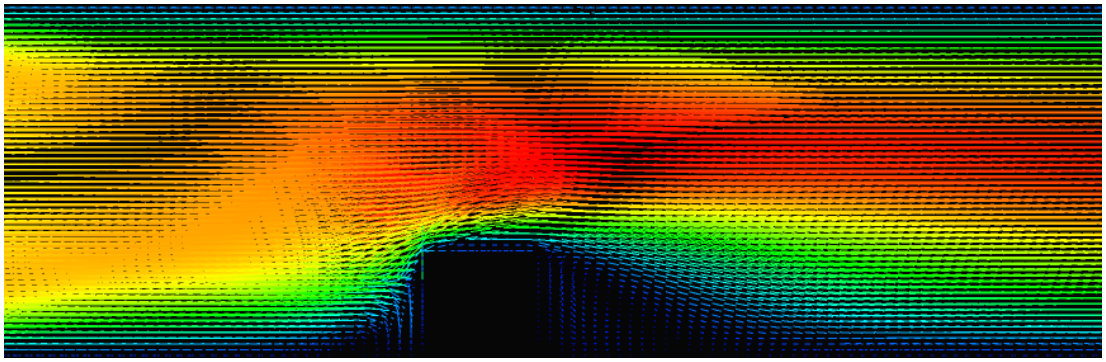
Figures 3.21 shows the vortices that occur in a laminar regime ($Re = 42$) around and behind the cube. Figure 3.22 compares the results to a *FLUENT* [4] simulation. Both pictures show the same recirculation patterns in the front and behind the cube. Colors differ because of different mapping between *FLUENT* and *ParaView*, which is used to display the results from the GPU simulation. Further validation should be done by comparing the velocity profiles, in front of the cube and behind it.

Urban-like Domains

Figure 3.23 gives the output of low Reynolds number flow around urban-like domain for different time steps. The grid is composed of $256 \times 256 \times 64$ computational nodes, representing a domain of 1.28 km $\times$ 1.28 km $\times$ 320 m. The inlet velocity is $U_{inlet} = 1$



(a) FLUENT (CPU)



(b) CUDA (GPU)

Figure 3.22: *FLUENT (CPU)* and *CUDA (GPU)* simulations for a laminar flow around a surface-mounted cube ($Re = 42$). Grid size is $256 \times 128 \times 64$.

m/s. The viscosity is set to $\nu = 0.5$ m^2/s. The Reynolds number, defined here as $Re = U_{inlet} \times (height/4) / \nu$, is then 155. A first-order upwind scheme is used for the discretization in space and a first-order Euler scheme is used for time integration. Figure 3.24 shows simulation outputs for the same settings but using a second order Adams-Bashfort scheme.

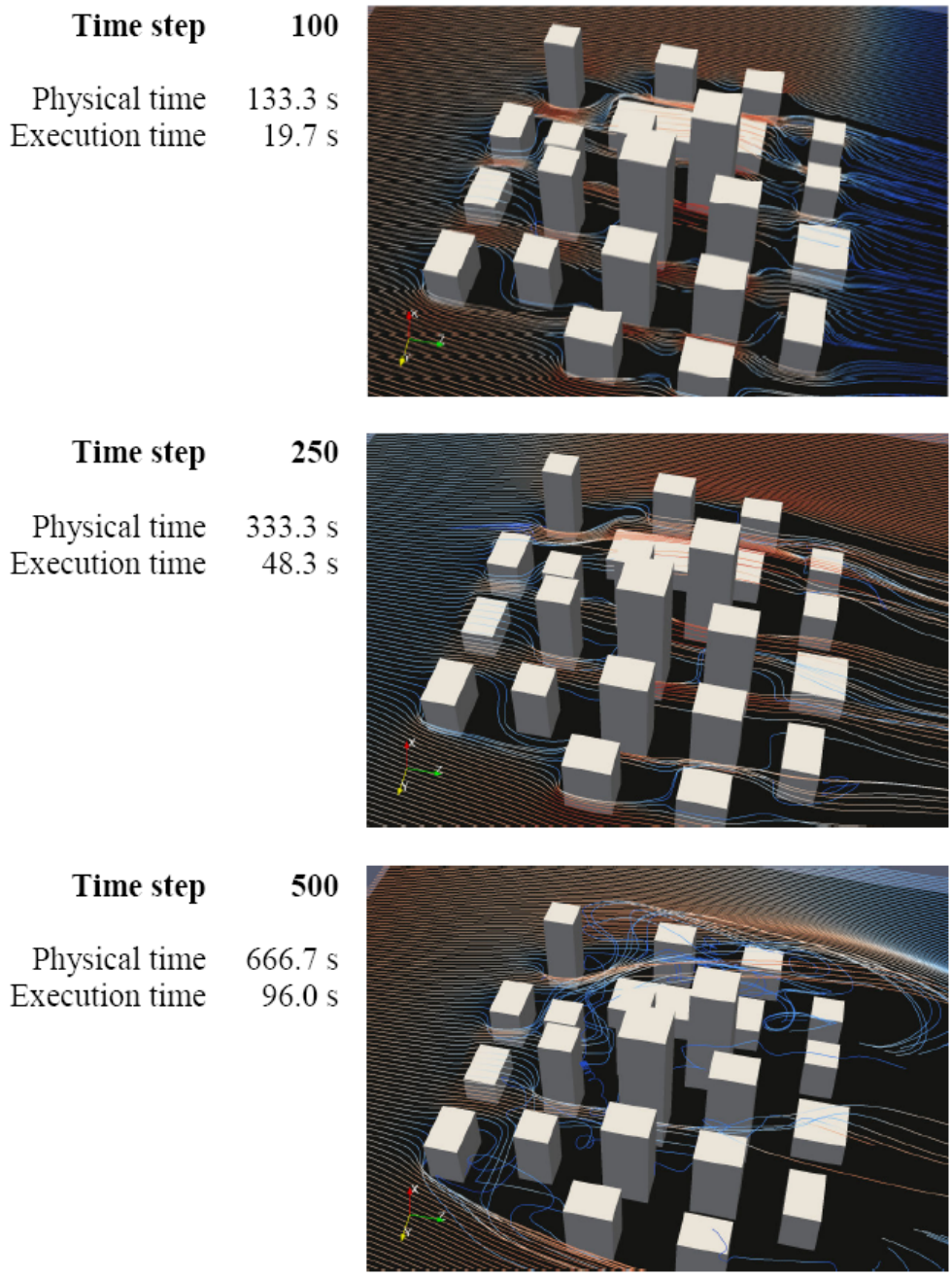
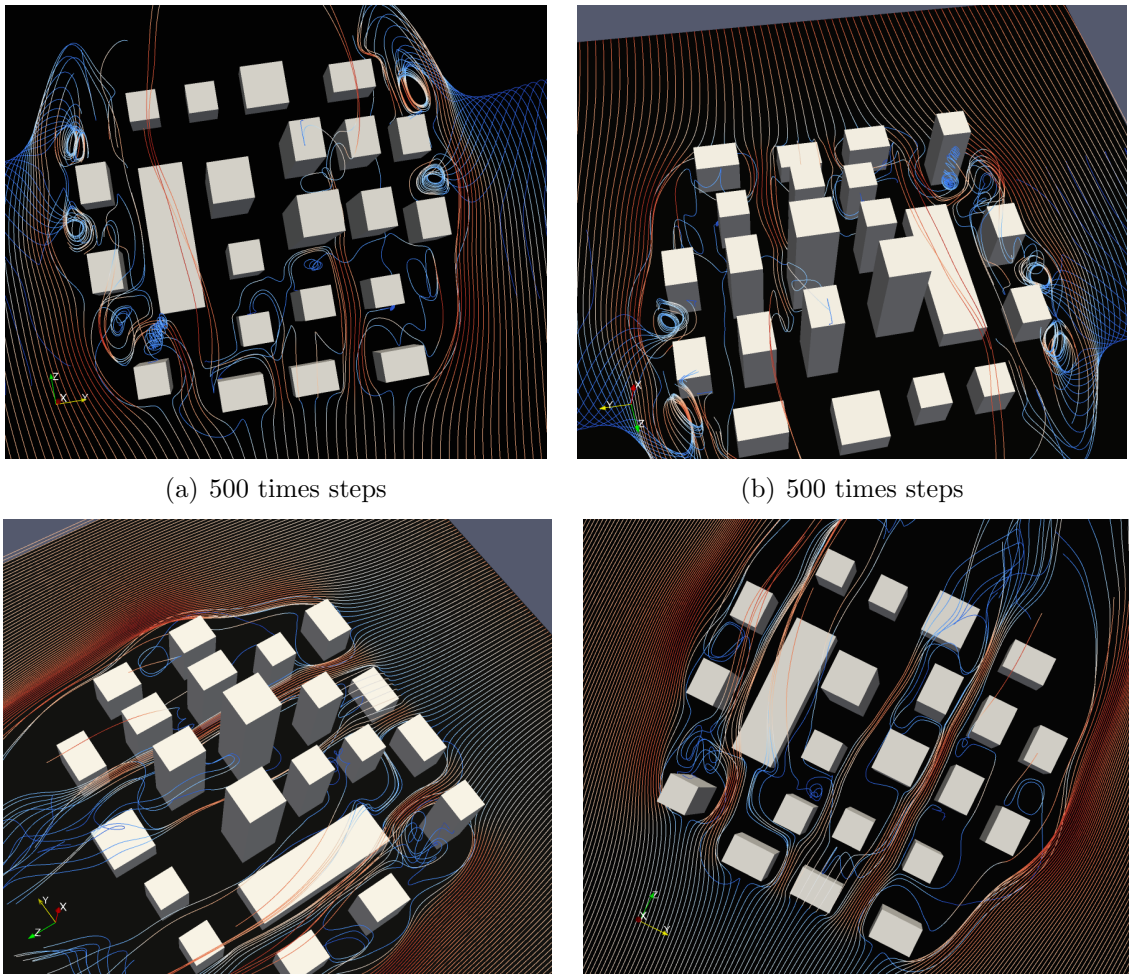


Figure 3.23: Low Reynolds number flow in an urban-like domain ($Re = 155$). Execution times are relative to the quad-GPU platform running a simulation using $256 \times 256 \times 64$ computational nodes and representing a domain of 1.28 km $\times$ 1.28 km $\times$ 320 m. Red streamlines represent high velocity magnitudes while blue streamlines represent lower velocity magnitudes



(a) 500 times steps

(b) 500 times steps

(c) 1000 times steps

(d) 1000 times steps

Figure 3.24: Simulation of a domain of 1.28 km $\times$ 1.28 km $\times$ 320 m on a $256\times256\times64$ grid using a second-order Adams-Bashfort scheme. The Reynolds number for these simulations is $Re = 155$ and 1000 times steps represent over 10 minutes of physical time.

CHAPTER 4

COMPUTATIONAL PERFORMANCE ANALYSIS

4.1 2D Wave Equation

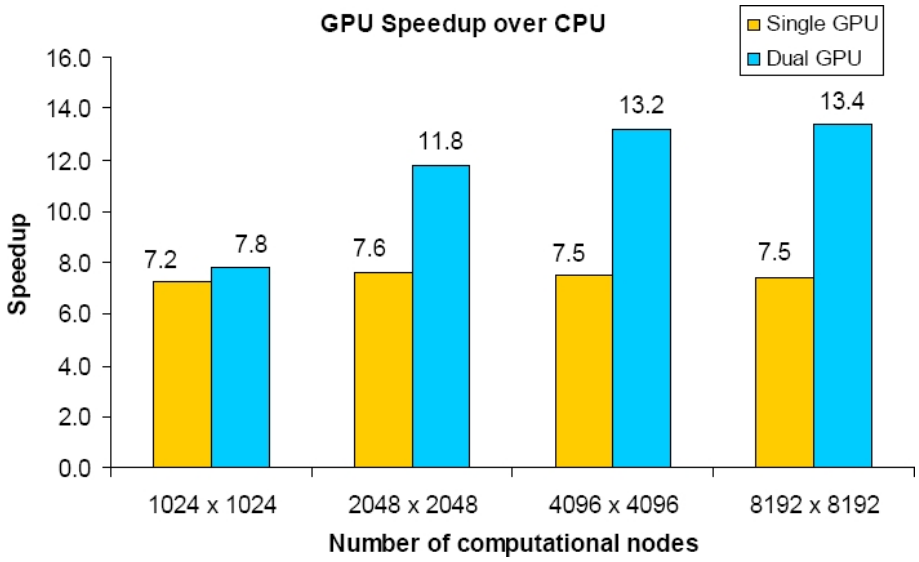


Figure 4.1: Acceleration of the wave simulation. GPU speedup relative to a serial CPU implementation is plotted for different physical domain sizes. The single GPU solution gives constant speedup over the serial code while the dual-GPU results indicate speedup for sufficiently large problems.

The overhead caused by the ghost cell transfer from the device to the host memory and vice versa is an important issue when small problems are considered for the dual-GPU solution. For a 1024×1024 domain the speedup with the dual-GPU is barely greater than the speedup obtained with a single GPU. As the problem size gets

larger, the performance of the dual-GPU solution gives a scaling factor of 1.79, which is closer to the ideal scaling factor of 2× relative to a single GPU implementation. The overhead of the dual GPU implementation over the single GPU implementation is defined as

$$\tau_{dual} = \frac{2 \times speedup_{single} - speedup_{dual}}{2 \times speedup_{single}} \tag{4.1}$$

where $speedup_{single}$ is the single GPU speedup relative to the serial CPU code, and $speedup_{dual}$ is the dual-GPU speedup relative to the serial CPU code. Figure 4.2 shows that the overhead drops to about 10% when working on an 8192×8192 domain. The

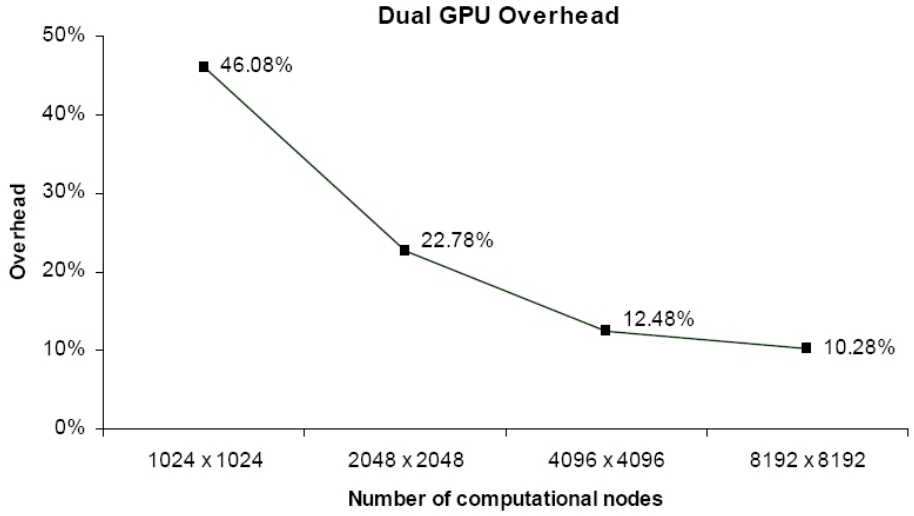


Figure 4.2: Overhead of the dual GPU implementation over the single GPU implementation.

ratio of number of ghost cells to the total number of computational nodes decreases with increasing problem size. For example when the domain width is doubled (overall domain size quadruples), the GPUs have to compute 4 times more cells. On the other hand the number of ghost cells to be exchanged across GPUs only doubles in size. As the communication overhead is directly proportional to the number of ghost cells,

multi-GPU solutions will be more adapted to large scale problems where the overhead of ghost cell communication across the GPUs is less significant than the computation of the inner cells on the GPUs.

Preliminary results have shown that a dual-GPU configuration is feasible, and it is a promising platform to accelerate the numerical solution of PDEs. The single-GPU implementation offers a quite constant speedup (around 7.5) with different problem sizes, whereas the dual-GPU performance is more dependent on the problem size. The speedup ranged from 7.8 to 13.8 due to the trade off between communication and computation. Better speedup and scaling performance were observed as the physical problem size increased. This is an encouraging result as most practical CFD applications are computationally intensive efforts.

4.2 Incompressible Navier-Stokes Solver

4.2.1 Serial CPU Code Benchmarking

Table 4.1 shows that the performance of the in-house serial CFD code is comparable to the NPB benchmark codes in terms of single precision Giga Floating Point Operations per Second (GFLOPS). Using only a single core, the performance of our CFD code is approximately 1.6 GFLOPS on the Intel Core 2 Duo 3.0 GHz, 1.0 GFLOPS on the AMD Opteron 2.4 GHz processors. Interestingly, the GFLOPS performance drops to approximately 0.50 when the computational problem size is substantially increased. Figure 4.3 shows more details about the performance of our serial CPU version CFD code with increasing problem size. Before proceeding to the GPU speedup assessment, the FLOPS performance of the serial CPU implementation of the CFD code was tested against comparable applications. Both the serial CPU version and

Table 4.1: GFLOPS performance of the serial CPU version of our CFD code and NPB benchmark codes on two different computers (Intel Core 2 Duo (E8400) 3.0 GHz and AMD Opteron (8216) 2.4 GHz). *LU* factorizes an equation into lower and upper triangular systems. The iteration loop of *MG* consists of the multigrid V-cycle operation and the residual calculation. *SP* is a simulated CFD application. Our CFD code simulates a lid-driven cavity problem.

Benchmark	Size	GFLOPS		Ratio
		Intel Core 2 Duo 3.0 GHz	*AMD Opteron 2.4 GHz*	Intel / AMD
LU.S	$12 \times 12 \times 12$	2.52	1.55	1.62
LU.W	$33 \times 33 \times 33$	2.54	1.02	2.48
LU.A	$64 \times 64 \times 64$	2.13	0.68	3.13
LU.B	$102 \times 102 \times 102$	1.20	0.68	1.78
MG.S	$32 \times 32 \times 32$	2.35	1.26	1.86
MG.W	$128 \times 128 \times 128$	1.64	0.87	1.88
MG.A	$256 \times 256 \times 256$	1.67	0.73	2.29
MG.B	$256 \times 256 \times 256$	1.78	0.79	2.27
SP.S	$12 \times 12 \times 12$	3.00	1.55	1.94
SP.W	$36 \times 36 \times 36$	2.36	0.76	3.09
SP.A	$64 \times 64 \times 64$	1.46	0.70	2.08
SP.B	$102 \times 102 \times 102$	1.38	0.49	2.81
in-house CFD code	$32 \times 32 \times 32$	1.58	1.03	1.54
	$1024 \times 32 \times 1024$	1.42	0.54	2.64

GPU version of our CFD code use the same numerical methods. The *NAS Parallel Benchmarks* [5] (NPB's) were derived from CFD codes. NPB was designed to compare the performance of parallel computers and it is widely recognized as a standard indicator of computer performance. In Table 4.1 the *LU*, *MG* and *SP* benchmarks from NPB are compared to the in-house developed serial CFD code written in *C*. The code was compiled with GNU C Compiler [22] (gcc) using optimization level *O3* with CPU architecture specifications (i.e., `-march=core2` for Intel Core 2 Duo;

`-march=opteron` for the AMD Opteron). The NPB benchmarks were compiled with the Intel Fortran compiler. On the Intel Core 2 Duo 3 GHz processor, the serial

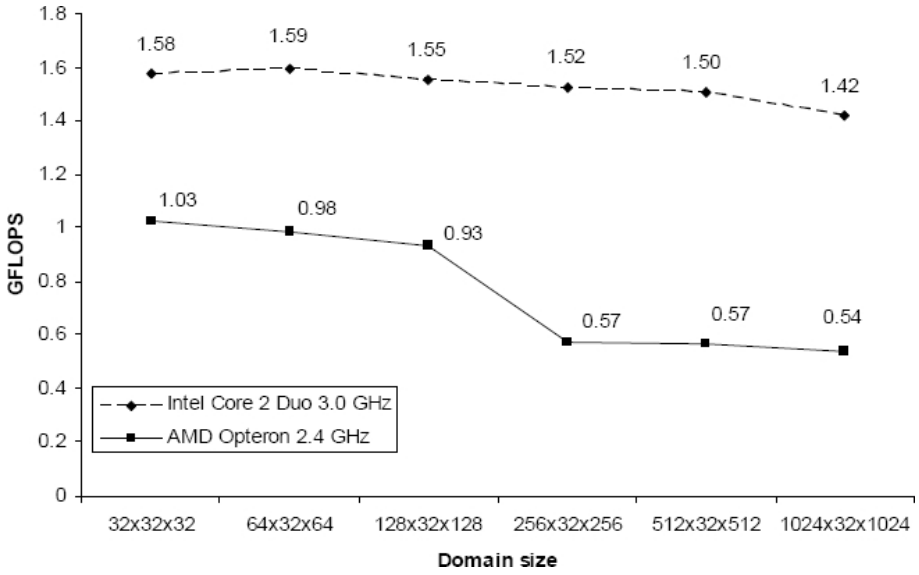


Figure 4.3: GFLOPS performance of the serial (CPU) in-house developed CFD code with increasing domain sizes.

CPU version of the CFD code performs pretty well as the GFLOPS number drops only by 10% when the domain size increases by a factor of 1024 (i.e., domain size increases from 32^3 to $1024^2 \times 32$). To put this into context, the SP benchmark performance drops by 54% when the domain size increases by a factor of 614 (i.e., domain size increases from 12^3 to 102^3). Figure 4.3 shows single precision GFLOPS performance drop on AMD Opteron 2.4 GHz when the domain size gets larger than $128 \times 32 \times 128$ (20 MB in memory). This was also observed with the NPB benchmark codes for problem sizes requiring over 20 MB of memory. The results shown in Figure 4.3 and Table 4.1 indicate that the serial CPU version of our CFD code is fairly optimized, giving performance comparable to NPB benchmark codes. Advance

code optimizations techniques may improve the GFLOPS performance of the serial CPU version of our CFD code, but it is not pursued in the present study.

4.2.2 Kernel Acceleration Using the Shared Memory

Projection algorithm involves distinct steps in a predictor-corrector fashion in the solution of the fluid flow equations. In the current study, each step is implemented as a kernel to be computed on the GPU. Both global and shared memory versions of the kernels were implemented. Figure 4.4 shows the benefit of adopting the shared memory depending on the kernel. Usage of the shared memory in the momentum and

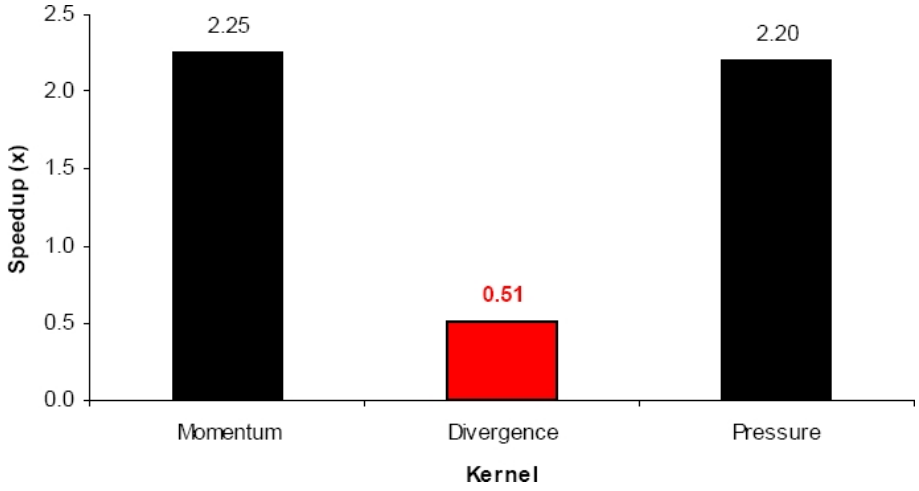


Figure 4.4: Kernel speedup of shared memory implementation relative to a full global memory implementation (domain size is $256 \times 32 \times 256$). Tests showed that the momentum and pressure kernels benefit from a shared memory implementation, giving a speedup of more than $2\times$ relative a kernel implementation that uses only the global memory.

pressure kernels make them perform over $2\times$ faster relative to a kernel implementation that uses only the global memory. These two kernels benefit from the shared memory because the overhead due to copying data to the shared memory is largely

compensated by their high arithmetic intensity and the need for each thread to access memory in a non-coalesced way. For the momentum and the pressure kernels, each thread needs to read the current value of the cell but also the ones from its direct neighbors in the x, y and z directions. The value to be updated can be accessed in a coalesced way but not its neighbors. The other kernels (*velocity correction* and *boundary conditions* for momentum and pressure) are not presented as they are similar to the *divergence* kernel in terms of the arithmetic operations. Either their arithmetic intensity is very low (*boundary conditions*) or the threads needs to access only a few data in a non-coalesced way (*velocity correction*). Figure 4.5 shows that using the

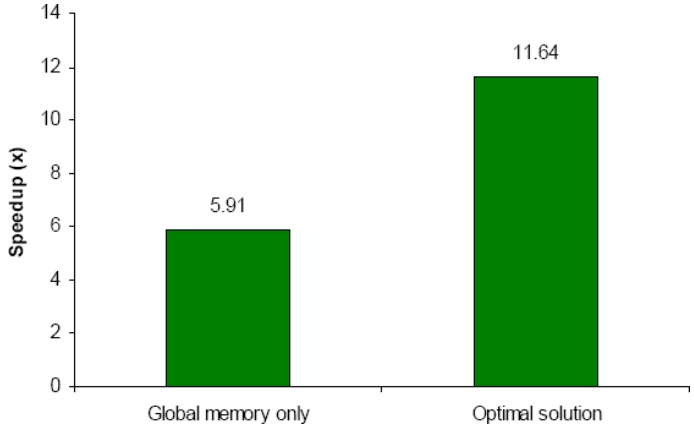


Figure 4.5: GPU speedup relative to the serial (CPU) code for global memory-only and optimized versions (domain size is $256 \times 32 \times 256$). The optimal solution uses shared memory for the momentum and pressure kernels while the other kernels use global memory only.

shared memory only for pressure and momentum kernels (optimal solution) makes the application run twice faster. Note that the GPU code would perform slower if all the kernels were implemented using the shared memory on the device. Hence, this implementation of the incompressible Navier-Stokes equations on a GPU is uniquely

adapted to the device memory architecture for superior performance in computational speed.

4.2.3 GPU Speedup Relative to CPU

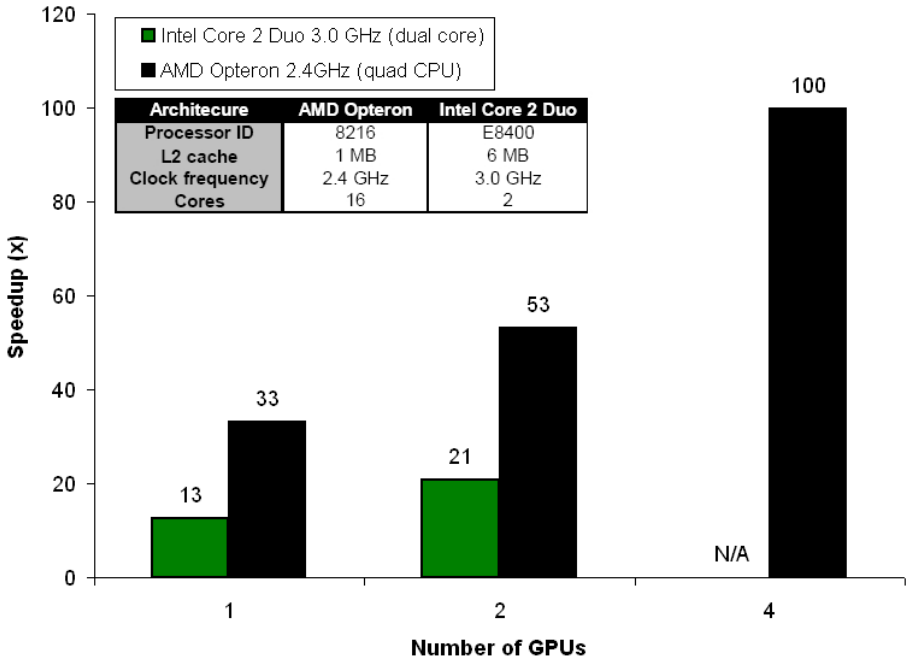


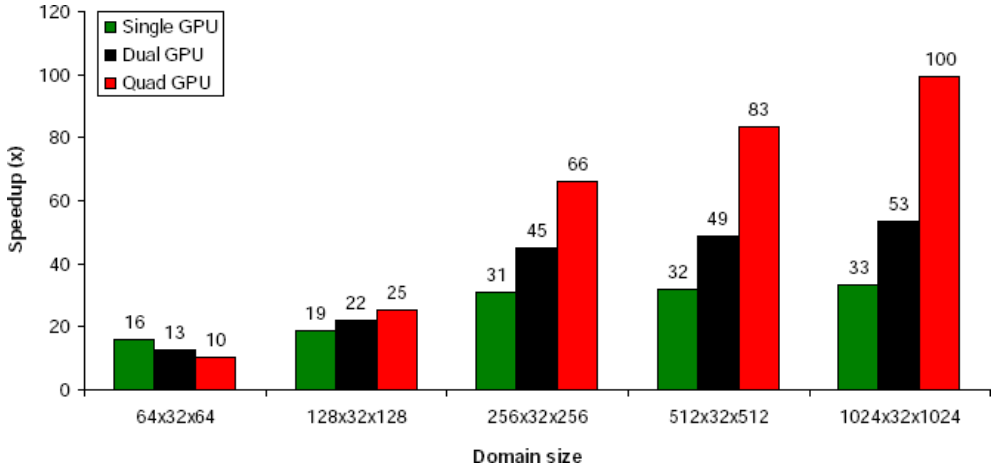
Figure 4.6: GPU speedup over serial CPU code for a domain of $1024 \times 32 \times 1024$ computational nodes. Quad-GPU results are not available for the Intel Core 2 Duo configuration because no quad-GPU/dual Intel Core 2 Duo platform was available for this study.

Using only a single CPU core, the serial CPU version of our CFD code takes 82,930 seconds on the Intel Core 2 Duo 3.0 GHz CPU and 218,580 seconds on AMD Opteron 2.4 GHz CPU to simulate the lid-driven cavity problem with a computational grid of $1024 \times 32 \times 102$ for 10,000 time steps. The serial CPU version of the CFD code runs faster on Intel Core 2 Duo CPU than on AMD Opteron CPU because of its larger L2 cache and its better clock frequency. On the other hand the execution time for

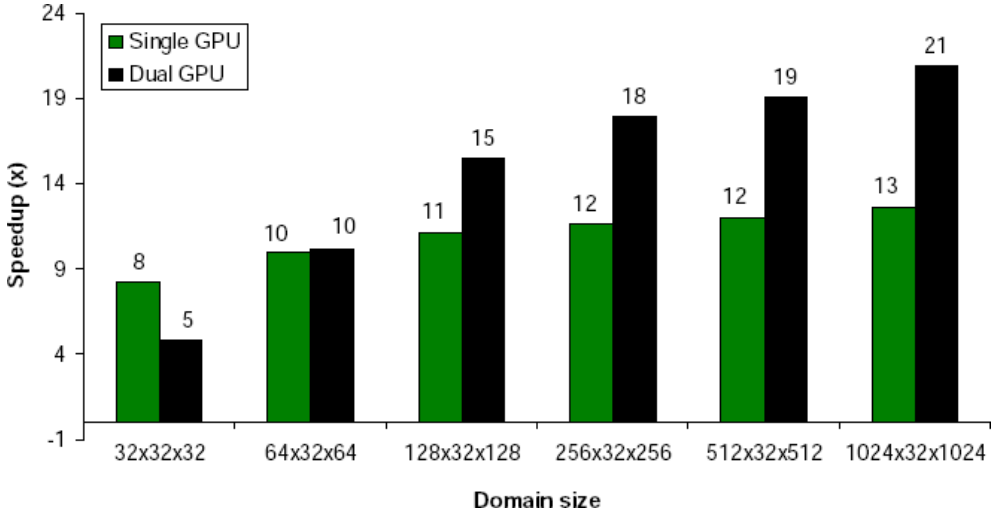
the GPU code is barely dependent on the CPU clock speed. GPU performance was nearly the same on both the Intel and AMD platforms. As a result GPU performance relative to the CPU performance is better for the AMD Opteron 2.4 GHz platform as shown in Figure 4.6. On the Intel Core 2 Duo platform the GPU code performs 13 and 21 times faster than the CPU code with one and two GPUs, respectively. On the AMD Opteron 2.4 GHz platform the GPU code performs 33, 53 and 100 times faster using one, two and four GPUs respectively. In the wave equation only one kernel was launched per time step. For the lid-driven cavity, multiple kernels are launched but the overhead is compensated by the higher arithmetic intensity of each kernel, especially for the momentum and pressure kernels. For example, it takes about 240 floating point operations to compute the velocity at a certain computational node (*momentum kernel*), while only 9 operations are necessary for the wave amplitude.

4.2.4 Multi-GPU Scaling Analysis

Figure 4.7 shows computational speedup with respect to different problem sizes. On the AMD Opteron platform (Figure 4.7(a)), depending on the problem size, the quad-GPU performance varies from 10× to 100× relative to the serial CPU version of the CFD code. On the Intel Core 2 Duo platform (Figure 4.7(b)), the dual-GPU performance varies from 5× to 21×. The speedup numbers are impressive for large problem size, because the arithmetic intensity on each GPU increases with problem size, and the time spent on data communication among GPUs compared to the time spent on computation becomes relatively shorter. For small problems, a multi-GPU computation performs slower than the single-GPU computation. On the AMD Opteron platform, for a problem of size $64 \times 32 \times 64$, the dual-GPU solution performs slower than the single-GPU, and the quad-GPU solution performs slower



(a) AMD Opteron 2.4 GHz with NVIDIA S870 Quad Tesla server



(b) Intel Core 2 Duo 3.0 GHz with dual NVIDIA C870 Tesla boards

Figure 4.7: Single and multi-GPU speedup relative to a single CPU core

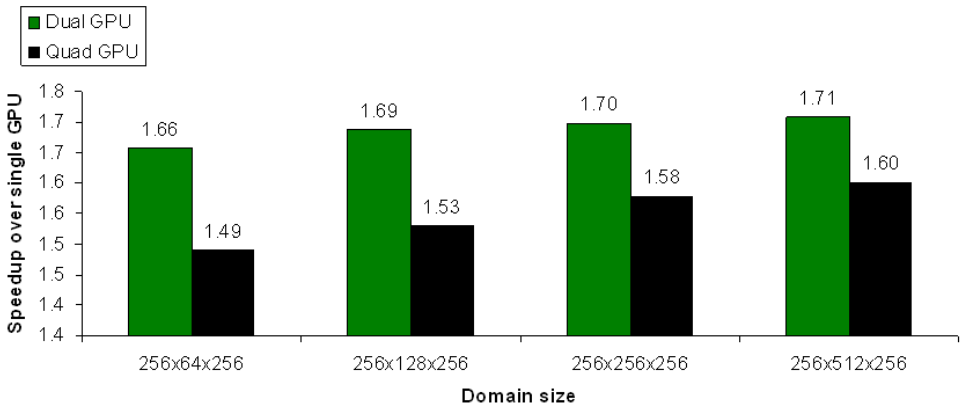


Figure 4.8: Multi-GPU scaling on the S870 server with dual-CPU platform. The multi-GPU platform does not scale well when then there is not a one CPU core per GPU ratio.

than the dual-GPU (Figure 4.7(a)). As more GPUs are available, the domain treated in the simulation should be larger to have each GPU working on large subdomain, and hide the latency due to GPU data exchange. Based on the current implementation, tests have shown that performance is better when there is a one-to-one matching between the number of GPUs and number of CPUs on desktop platforms. For example, Figure 4.8 shows that a dual-CPU platform coupled to the quad-GPU S870 server does provide any gain in performance over a dual-CPU dual-GPU Tesla C870 platform. Note that this statement is dependent on the current implementation, and performance may be improved by overlapping communication with computation.

Figure 4.9 shows the multi-GPU performance scaling on the NVIDIA Tesla S870 server. The speedup results shown in Figure 4.7(a) are converted to scaling numbers. By increasing the problem size and adjusting the size of the data to exchange between the GPUs, the performance on the quad-GPU platform is 3× the performance of a single GPU, and the dual-GPU solution performs 1.6× faster than the single

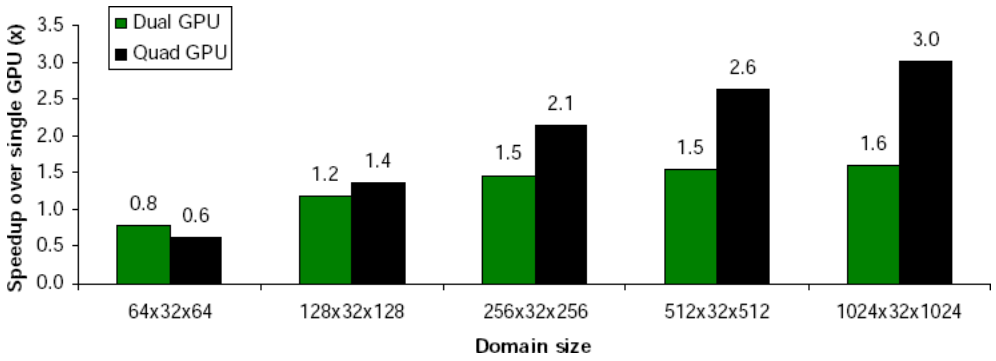


Figure 4.9: Multi-GPU scaling on the S870 server with quad-CPU platform. As the problem size increases the multi-GPU solutions scale better.

GPU. These performance numbers are less than the ideal performance numbers of 4× and 2×, respectively. The bottleneck of the multi-GPU solution is the data exchange between the GPUs, which requires synchronization and data transfer form the different GPUs to the host and vice versa.

4.3 Complex Geometry Capability Implementation

4.3.1 Impact of Thread Block Configuration

In this section, the execution time of the code for 3 different thread block configuration is analyzed. The simulation was run for 200 time steps and the Jacobi solver was set to use 20 iterations per time step. With an 8×8 thread block configuration the overall compute time is 198 seconds. This is twice slower than the 16×8 or 16×16 configuration (around 98 seconds). Figure 4.10 and Table 4.2 present the time spent in each kernel for the three block configurations. First, the pressure and the obstacle logic kernels are the most time-consuming kernels. The pressure kernel is actually called 20 times per timestep to solve the poisson equation, which explains why it is

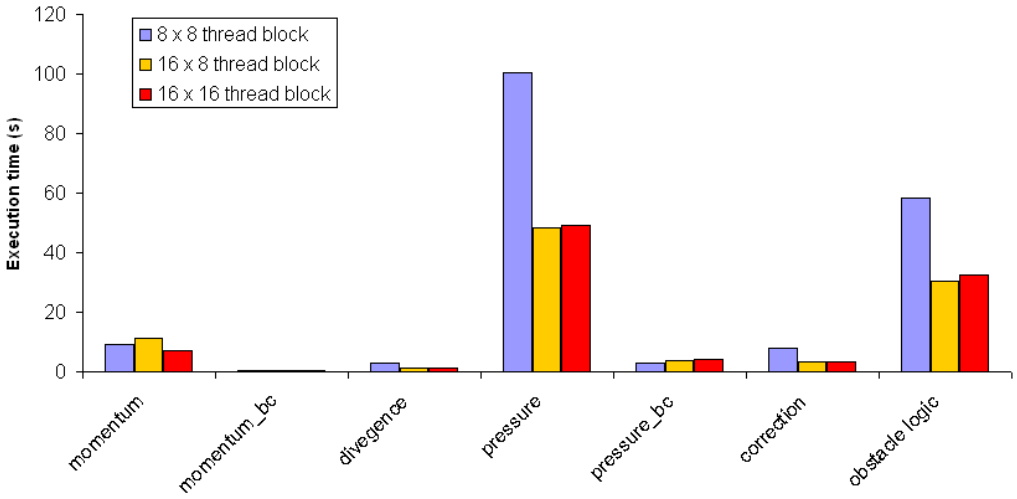


Figure 4.10: Comparison of kernel execution times for different block configurations. The urban-like domain was represented by a $256 \times 256 \times 64$ grid and each simulation ran for 200 time steps.

Table 4.2: Kernel execution times for different block configurations. The urban-like domain was represented by a $256 \times 256 \times 64$ grid and each simulation ran for 200 time steps.

Kernel	8×8 thread block		16×8 thread block		16×16 thread block	
	Time	*Weight*	*Time*	*Weight*	*Time*	*Weight*
momentum	9.26 s	5.07 %	11.31 s	11.44 %	7.14 s	7.27 %
momentum_bc	0.47 s	0.26 %	0.46 s	0.47 %	0.48 s	0.49 %
divergence	2.89 s	1.59 %	1.25 s	1.27 %	1.29 s	1.31 %
pressure	100.49 s	55.08 %	48.18 s	48.75 %	49.35 s	50.27 %
pressure_bc	3.06 s	1.68 %	3.81 s	3.86 %	4.10 s	4.17 %
correction	8.08 s	4.43 %	3.18 s	3.22 %	3.31 s	3.37 %
obstacle logic	58.19 s	31.89 %	30.61 s	30.97 %	32.49 s	33.09 %

the most time-consuming kernel for any configuration. The obstacle logic kernels is actually a set of 3 kernels, each being called twice per time step (after the predicted velocity is computed and after the velocity correction). Surprisingly, the *momentum* kernel is the only one benefiting from a 16×16 thread configuration. The *pressure_bc* kernel is actually faster with an 8×8 thread block configuration. All the other kernels are faster with 16×8 threads per block. Those tests confirmed that each kernel should be tested separately as the optimal thread block configuration is dependent on the kernel implementation.

4.3.2 Weight of Data Transfer in Multi-GPU Implementation

Table 4.3 shows that about 93% of the execution time is spent in kernel execution and about 7% is spent in data exchange between the GPUs through the host. This does not include the time spent waiting during thread synchronization which actually forces each GPU to wait for the slowest GPU to finish updating the ghost cells on the host side. The weight of most data transfers is below 1% (Table 4.3). Transfers 3 and 4 are executed at each iteration of the Jacobi solver. It represents about 10% of the time spent in the pressure kernel. Like the pressure kernel, these data transfers would take less time overall if the Jacobi solver used less iterations. Data transfers in general are faster for GPU 0 and 3 as GPU 0 requires ghost cells only for the top layer of its subdomain and GPU 3 requires ghost cells only for the bottom layer (Figure 3.12(a)).

Bandwidth tests for host-to-device memory transfers and device-to-device memory transfers are given in Table A.1. Results show that pinned memory should be preferred over pageable memory. Data transfers using pinned memory are twice faster for the S870 quad-GPU system and about 1.6 times faster for the C870 boards.

Table 4.3: Execution time of the different kernels and data exchange in a quad-GPU simulation. An urban-like domain was represented by a $256 \times 256 \times 64$ grid and each simulation ran for 1000 time steps.

	Execution time (s)				Average time (s)	Weight
	GPU 0	*GPU 1*	*GPU 2*	*GPU 3*		
Kernel						
momentum	8.83	8.94	8.93	9.50	9.05	4.86%
momentum_bc	0.80	0.78	0.76	0.78	0.78	0.42%
divergence	1.65	1.67	1.66	1.75	1.68	0.90%
pressure	99.85	101.12	101.01	99.79	100.44	53.92%
pressure_bc	7.28	7.13	7.03	7.15	7.14	3.84%
correction	4.55	4.61	4.60	4.66	4.60	2.47%
obstacle logic	48.09	48.73	48.68	49.37	48.72	26.15%
Ghost cell exchange						
transfer 0	0.65	0.95	1.04	0.71	0.84	0.45%
transfer 1	0.45	0.91	0.91	0.47	0.68	0.37%
transfer 2	0.85	1.21	1.31	0.88	1.06	0.57%
transfer 3	4.05	5.61	5.48	3.81	4.73	2.54%
transfer 4	4.37	6.59	7.52	4.82	5.83	3.13%
transfer 5	0.45	0.96	0.96	0.58	0.73	0.39%

Ghost cell exchange sections:

transfer 0:	copy `u,v,w`	from host to device	(before *momentum* kernel)
transfer 1:	copy `u,v,w`	from device to host	(before *divergence* kernel)
transfer 2:	copy `u,v,w`	from host to device	(before *divergence* kernel)
transfer 3:	copy `p`	from device to host	(in the Jacobi solver)
transfer 4:	copy `p`	from host to device	(in the Jacobi solver)
transfer 5:	copy `u,v,w`	from device to host	(before starting a new time step)

Table 4.4: Register usage (single and multi-GPU implementations)

Kernel	Register usage (per thread) *Single-GPU*	*Multi-GPU*
momentum	29	32
momentum_bc	11	11
divergence	9	9
pressure	14	18
pressure_bc	13	12
correction	12	12
obstacles_zero_mask	9	14
obstacles_sweep_uvw_kii	14	17
obstacles_sweep_uvw_jkj	14	19
obstacles_corners	17	17
nu_turbulent	9	9

Depending on the hardware and the type of memory, data transfers from host to device are between 20 and 90 times slower than local device memory transfers. Bandwidth tests on the new Tesla C1060 using a PCI Express Gen. 2 bus show that data transfers using pinned memory are almost twice faster than for the C870 utilized in this research. The multi-GPU implementation would benefit from this platform, along with the faster on-device memory provided by the C1060.

4.3.3 Register Usage

The GPUs utilized for this research allow 8192 registers per block. Using a 16×16 thread configuration, each kernel can use a maximum of $8192/256 = 32$ registers per threads. Table 4.4 presents the number of registers used in the different kernels.

Modifications on the momentum kernel had to be done to reduce the number of

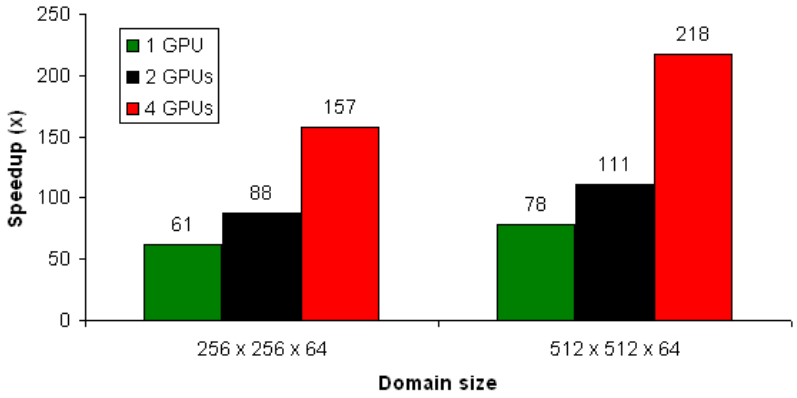


Figure 4.11: Single and multi-GPU speedup relative to a single CPU core of an AMD Opteron 2.4 GHz for urban simulations

registers to 32 per thread. Experience showed that reducing the register pressure is a complicated exercise when dealing with complex kernel implementations. If the number of registers cannot be reduced to 32, then the number of threads per blocks has to be reduced, which might cause a loss of performance depending on the kernel implementation. The new NVIDIA GPUs, such as the Tesla C1060, have twice more registers available, for a total of 16384 registers per block. This allows the implementation of more complex kernels and usage of more local variables.

4.3.4 Speedup Analysis

Figure 4.3.4 shows the speedup of the CUDA implementation for urban-like domains relative to a single CPU core of an AMD Opteron 2.4 GHz. The serial code followed a straightforward implementation. The CPU algorithms need to be optimized for a fair comparison relative to a GPU implementation. For a domain of $256 \times 256 \times 64$ the serial C code uses about 210 MB of memory. As the size of the domain increases by a factor of 4 ($512 \times 512 \times 64$) the serial code needs actually over 5 times longer

to run the same number of time steps. The differences of speedup for the single and dual-GPU for the two different problem sizes is due to this slow down in the serial code for increasing domains. On the other hand, the quad-GPU scales well and performs better for the larger domain, even by taking the serial C code slowdown in consideration. The quad-GPU implementation actually performs almost twice faster than the dual-GPU. For a domain size of $256 \times 256 \times 64$ the quad-GPU performs 2.6 times and 2.8 times faster than the single GPU implementation. These results confirm that the multi-GPU implementation is suitable for big problems.

CHAPTER 5

CONCLUSIONS

5.1 Results

This thesis presents the implementation of the Navier-Stokes equations for incompressible fluid flow on desktop platforms with multi-GPUs. NVIDIA's CUDA programming model was used to implement the discretized form of the governing equations. The major steps of the projection algorithm [14] are implemented with separate CUDA kernels, and a unique implementation that exploits the memory hierarchy of the CUDA programming model is suggested. Kernels for the velocity predictor step and the solution of the pressure Poisson equation were implemented using the shared memory of the device, whereas a global memory implementation was pursued for the kernels that are responsible to calculate the divergence field and velocity corrections and to apply the boundary conditions. This unique combination resulted in factor of two speedup relative to a full global memory implementation on the device. Overall, the numerical solution of incompressible fluid flow equations was accelerated by a factor of 100 using the NVIDIA S870 Tesla server with quad GPUs. The speedup number is measured relative to the serial CPU version of the CFD code that was executed using a single core of an AMD Opteron 2.4 GHz processor. With respect to a single core of an Intel Core 2 Duo 3.0 GHz processor, a speedup of 13 and 21 was achieved with single and dual GPUs (NVIDIA Tesla C870), respectively. Single

precision computations and same numerical methods were adopted in both the CPU and GPU versions of the CFD code. Tests showed that multi-GPU scaling and speedup results improve with increasing computational problem size, suggesting that computationally big problems can be tackled with GPU clusters with multiple GPUs in each node. Scaling results also showed that in a multi-GPU desktop platform, one CPU core should be dedicated to each active GPU in order to obtain good scaling performance across multi-GPUs.

An urban geometry capability was also added to the Navier-Stokes solver to model building effects on the flow-field. A novel obstacle logic was applied to the pressure and velocity fields. This was achieved by creating four extra kernels and updating the existing ones. Using the quad-GPU system (NVIDIA Tesla S870), a low Reynolds number ($Re = 155$) flow-field computation of 22 minutes (1000 time steps) in an urban-like domain of 1.28 km $\times$ 1.28 km $\times$ 320 m was simulated in 3 minutes. This work represents the first incompressible Navier-Stokes solver based on a CUDA implementation for multi-GPU desktop platforms. Results showed that multi-GPU desktop platforms have substantial potential for large CFD problems.

Preliminary tests with the Tesla C1060 showed that multi-GPU simulations could be accelerated by a factor of 5 by using the latest NVIDIA GPU hardware on PCIe Gen.2 bus. While the PCIe Gen.2 bus offers a better bandwidth for data transfers from host to device, the new Tesla C1060 allows more coalesced accesses than its predecessors and offers a better memory bandwidth at the GPU level. Multi-GPU platforms will definitely benefit from the GPU hardware evolutions and should be seriously considered as a new solution for high performance computing in CFD.

5.2 Further Work

This work lays the foundation for an urban dispersion model on multi-GPU desktop platforms. A turbulent model was prototyped by adding a kernel to compute the turbulent viscosity. The momentum kernel was modified to use a second-order Adams-Bashfort scheme. Validation of a turbulent model was beyond the scope of this thesis and would require numerous tests against benchmark data. But preliminary results show that a Navier-Stokes-based urban dispersion model is feasible on a multi-GPU platform. New kernels for the transport equation for contaminant dispersion would have to be added to the existing set. Advection schemes for contaminant transport that minimizes numerical diffusion while ensuring physically correct values need to be implemented. As discussed by Blocken et al. [8], a realistic urban simulation needs to address the atmospheric boundary layer. Wall functions would have to be defined and applied to the buildings for more accurate wind-field computation. The Jacobi solver used here for the pressure Poisson equation should be used in conjunction with a multigrid method. It would enforce conservation of mass strictly at each timestep. This is necessary as turbulent flows are very sensitive to numerical errors.

In the current implementation of the Navier-Stokes solver, a uniform grid is used to represent the domain. As more features are added to the basic Navier-Stokes solver, more kernels and more computation is required. Extending this work to use an adapted mesh would avoid unnecessary computation while focusing on the domain areas of interest, such as city rifts.

The output of the simulation is currently a *Plot3D* file that represents the different quantities of interest in the studied domain. This type of file can be used as an input by almost every visualization tool, such as *ParaView*. Then, after a simulation the

output needs to be post-processed. A visualization software can present the quantities of interest as streamlines, contours, velocity vectors, etc. More detailed analysis can also be done through profile plotting. The Tesla cards that were used during this research do not offer any video output. But switching to a different hardware, such as a GPU card from the GeForce series, would enable simple display during the simulation in addition to the *Plot3D* output. The implementation on the GPU side would require an interface with OpenGL, which is offered in CUDA.

REFERENCES

[1] K.J. Allwine and J.E. Flaherty. Urban Dispersion Program Overview and MID05 Field Study Summary. Technical report, PNNL-16696, Pacific Northwest National Laboratory (PNNL), Richland, WA (US), 2007.

[2] K.J. Allwine, M.J. Leach, L.W. Stockham, J.S. Shinn, R.P. Hosker, J.F. Bowers, and J.C. Pace. Overview of Joint Urban 2003–An atmospheric dispersion study in Oklahoma City. *8th Symposium on Integrated Observing and Assimilation Systems for Atmospheres, Oceans, and Land Surface*, 2004.

[3] J.A. Anderson, C.D. Lorenz, and A. Travesset. General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units. *Journal of Computaional Physics*, 227(10):5342–5359, May 2008.

[4] ANSYS, Inc. Fluent, CFD Flow Modeling Software & Solutions, Version 6.3.26. `http://www.fluent.com/`.

[5] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications and High Performance Computing*, 5(3):63–73, 1991.

[6] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, and E.S. Quintana-Ortı. Solving Dense Linear Systems on Graphics Processors. Technical report, Technical Report ICC 02-02-2008, Universidad Jaume I, Depto. de Ingenieria y Ciencia de Computadores, February 2008.

[7] A. Bleiweiss. GPU Accelerated Pathfinding. In *Proceedings of the 23rd ACM Siggraph/Eurographics symposium on Graphics hardware*, pages 65–74. Eurographics Association Aire-la-Ville, Switzerland, 2008.

[8] B. Blocken, T. Stathopoulos, and J. Carmeliet. CFD Simulation of the Atmospheric Boundary Layer: Wall Function Problems. *Atmospheric Environment*, 41(2):238–252, 2007.

[9] T. Brandvik and G. Pullan. Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware. *46th AIAA Aerospace Sciences Meeting and Exhibit*, 2008.

[10] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004.

[11] M. Castillo, E. Chan, F.D. Igual, R. Mayo, E.S. Quintana-Ortı, G. Quintana-Ortı, R. van de Geijn, and F.G. Van Zee. Making Programming Synonymous with Programming for Linear Algebra Libraries. *FLAME Working Note*, 31:08–20, April 2008.

[12] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[13] S. Che, J. Meng, J.W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors. In *The First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[14] A.J. Chorin. Numerical Solution of Navier-Stokes Equations. *Mathematics of Computation*, 22(104):745–762, 1968.

[15] S.R. Diehl, D.A. Burrows, E.A. Hendricks, and R. Keith. Urban Dispersion Modeling: Comparison with Single-Building Measurements. *Journal of Applied Meteorology and Climatology*, 46(12):2180–2191, 2007.

[16] E. Elsen, P. LeGresley, and E. Darve. Large calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics*, 2008.

[17] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU Cluster for High Performance Computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society Washington, DC, USA, 2004.

[18] J.H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer New York, 2002.

[19] J.E. Flaherty, K.J. Allwine, M.J. Brown, W.J. Coirier, S.C. Ericson, O.R. Hansen, A.H. Huber, S. Kim, M.J. Leach, J.D. Mirocha, et al. Evaluation study of building-resolved urban dispersion models. In *7th Symposium on the Urban Environment*, 2007.

[20] J.E. Flaherty, D. Stock, and B. Lamb. Computational Fluid Dynamic Simulations of Plume Dispersion in Urban Oklahoma City. *Journal of Applied Meteorology and Climatology*, 46(12):2110–2126, 2007.

[21] MPI Forum. The Message Passing Interface (MPI) Standard. `http://www-unix.mcs.anl.gov/mpi/`.

[22] GCC, GNU Compiler Collection, Ver. 4.1.2, Sept. 2007. `http://gcc.gnu.org`.

[23] U. Ghia, K.N. Ghia, and C.T. Shin. High-RE Solutions For Incompressible-Flow Using the Navier-Stokes Equations and a Multigrid Method. *Journal of Computational Physics*, 48(3):387–411, 1982.

[24] A. Gowardhan. *Towards Understanding Flow and Dispersion in Urban Areas Using Numerical Tools*. PhD thesis, University of Utah, UT, 2008.

[25] A. Gowardhan, E.R. Pardyjak, I. Senocak, and M.J. Brown. A CFD Based Wind Solver For a Fast Response Dispersion Model. In *Seventh Biennial Tri-Laboratory Engineering Conference, Albuquerque, New Mexico*, May 2007.

[26] M.J. Harris. *Real-Time Cloud Simulation and Rendering*. PhD thesis, University of North Carolina, 2003.

[27] J. Heagy, N. Platt, S. Warner, and J. Urban. Joint Effects Model Urban IPT. *CBIS Conference, Austin TX*, 2007. `http://www.dtic.mil/ndia/2007cbis/thursday/heagyThurs945.pdf`.

[28] E.A. Hendricks, S.R. Diehl, D.A. Burrows, and R. Keith. Evaluation of a Fast-Running Urban Dispersion Modeling System Using Joint Urban 2003 Field Data. *Journal of Applied Meteorology and Climatology*, 46(12):2165–2179, 2007.

[29] J.L. Hennessy, D.A. Patterson, D. Goldberg, and K. Asanovic. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[30] M. Houston. Stream Computing. In *International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH 2008 classes*, number 15. ACM Press/Addison-Wesley Publishing Co. New York, NY, 2008.

[31] W. Li, Z. Fan, X. Wei, and A. Kaufman. GPU-Based Flow Simulation with Complex Boundaries. *GPU Gems*, 2:747–764, 2005.

[32] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig. Molecular Dynamics Simulations on Commodity GPUs with CUDA. *Lecture Notes in Computer Science*, 4873:185, 2007.

[33] Y. Liu, X. Liu, and E. Wu. Real-time 3D fluid simulation on GPU with complex obstacles. In *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on*, pages 247–256, 2004.

[34] J. Michalakes and M. Vachharajani. GPU Acceleration of Numerical Weather Prediction. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7. IEEE Computer Society, Washington, DC, 2008.

[35] J. Molemaker, J.M. Cohen, S. Patel, and J. Noh. Low Viscosity Flow Simulations for Animation. *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, 2008.

[36] Nvidia. CUDA Zone, the resource for CUDA developers. `http://www.nvidia.com/cuda`.

[37] Nvidia. CUDA Programming Tools, 2007. `http://www.nvidia.com/object/cuda_programming_tools.html`.

[38] Nvidia. CUDA Compute Unified Device Architecture Programming Guide, Version 2.0, 2008. `http://www.nvidia.com/object/cuda_documentation.html`.

[39] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[40] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krueger, A.E. Lefohn, and T.J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[41] G. Patnaik, J.P. Boris, T.R. Young, and F.F. Grinstein. Large scale urban contaminant transport simulations with MILES. *Journal of Fluids Engineering*, 129:1524, 2007.

[42] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.H. Wen-mei. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82. ACM New York, NY, 2008.

[43] M.C. Schatz, C. Trapnell, A.L. Delcher, and A. Varshney. High-Throughput Sequence Alignment Using Graphics Processing Units. *BMC Bioinformatics*, 8:474, 2007.

[44] I. Senocak, J. Thibault, and M. Caylor. Rapid-response Urban CFD Simulations Using a GPU Computing Paradigm on Desktop Supercomputers. In *8th Symposium on the Urban Environment*, 2009.

[45] J. Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co. New York, NY, 1999.

[46] J.A. Stratton, S.S. Stone, and W.H. Wen-mei. MCUDA: An Efficient Implementation of CUDA Kernels on Multi-cores. *Center for Reliable and High-Performance Computing*, 2008.

[47] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl. CUDASA: Compute Unified Device and Systems Architecture. In *EG Symp. Parallel Graph. Vis., S*, pages 49–56, 2008.

[48] J.C. Tannehill, D.A. Anderson, and R.H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. Taylor & Francis Group, 1997.

[49] J. Thibault and I. Senocak. CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows. *47th AIAA Aerospace Sciences Meeting and Exhibit*, 2008.

[50] J. Thibault and I. Senocak. Accelerating the Incompressible Flow Computations with a Multi-GPU Computing Paradigm. *10th U.S. National Congress for Computational Mechanics*, July 2009.

[51] J. Toelke and M. Krafczyk. TeraFLOP Computing on a Desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008.

[52] I.S. Ufimtsev and T.J. Martínez. Quantum Chemistry on Graphical Processing Units. 1. Strategies for Two-Electron Integral Evaluation. *Journal of Chemical Theory and Computation*, 4(2):222–231, 2008.

APPENDIX A

HIGH PERFORMANCE COMPUTING INFRASTRUCTURE

A.1 GPU Hardware Specifications

- **Tesla C870 GPU Computing Processor** (Figure A.1(a))

  128 streaming processor cores at 1.35 GHz

  1.5 GB of dedicated memory at 800 MHz

  76.8 GB/sec of memory bandwidth

  Fits in one full-length, dual slot with one open PCI Express x16 slot

- **GeForce 9800 GX2** (Figure A.1(b))

  Dual-GPU system

  256 thread processors (128 per GPU) at 1.5 GHz

  1 GB of dedicated memory at 1000 MHz

  64 GB/sec of memory bandwidth per GPU

- **Tesla S870 GPU Computing System** (Figure A.1(c))

  4 × Tesla GPUs (128 thread processors per GPU)

  6 GB of dedicated memory (1.5 GB per GPU) at 800 MHz

76.8 GB/sec of memory bandwidth per GPU

Standard 19", 1U rack-mount chassis

Connects to host via low power PCI Express x16 adapter card

2 PCI Express connectors driving 2 GPUs each (4 GPUs total)

- **Tesla C1060 GPU Computing Processor**

  240 streaming processor cores at 1.3 GHz

  4 GB of dedicated memory at 800 MHz

  102 GB/sec of memory bandwidth

(a) Tesla C870 Board (b) GeForce 9800 GX2 (c) Tesla S870 server

Figure A.1: GPU computing hardware utilized in the research. One of the computers is equipped of 2 Tesla C870 (a), a second one is equipped with 2 GeForce 9800 GX2 (b) and another one is connected to a Tesla S870 server (c).

A.2 Hardware Bandwidth Tests

Table A.1 gives the results of memory bandwidth tests on different hardware. The values presented here were obtained with the *bandwidthTest* application provided in the CUDA SDK.

Table A.1: Bandwidth tests for memory transfers between host and device

		Bandwidth (MB/s)	
	Memory transfer	*Pageable memory*	*Pinned memory*
S870	Host to device	840	1,895
	Device to host	691	1,246
	Device to device	63,795	63,795
C870	Host to device	2,022	3,179
	Device to host	1,932	3,064
	Device to device	64,012	64,012
C1060 *(with PCIe gen. 2.0)*	Host to device	2,302	5,793
	Device to host	2,053	5,605
	Device to device	73,715	73,715

APPENDIX B

DISCRETIZATION OF THE GOVERNING EQUATIONS

B.1 Continuity Equation

$$div(i,j,k) = \frac{U_{i,j,k} - U_{i-1,j,k}}{dx} + \frac{V_{i,j,k} - V_{i,j-1,k}}{dy} + \frac{W_{i,j,k} - W_{i,j,k-1}}{dz} \tag{B.1}$$

B.2 Navier-Stokes Equations

B.2.1 General Form of the Navier-Stokes Equations

The 3D Navier-Stokes equation (Eq. 2.3) reads as follows:

$$\begin{aligned}
\frac{\partial u}{\partial t} &= -\frac{\partial uu}{\partial x} - \frac{\partial uv}{\partial y} - \frac{\partial uw}{\partial z} - \frac{1}{\rho}\frac{\partial P}{\partial x} \\
&\quad + \frac{\partial}{\partial x}\left[2\nu\frac{\partial u}{\partial x}\right] + \frac{\partial}{\partial y}\left[\nu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right] + \frac{\partial}{\partial z}\left[\nu\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right)\right] - g_x \\
\frac{\partial v}{\partial t} &= -\frac{\partial uv}{\partial x} - \frac{\partial vv}{\partial y} - \frac{\partial vw}{\partial z} - \frac{1}{\rho}\frac{\partial P}{\partial x} \\
&\quad + \frac{\partial}{\partial x}\left[\nu\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right)\right] + \frac{\partial}{\partial y}\left[2\nu\frac{\partial v}{\partial y}\right] + \frac{\partial}{\partial z}\left[\nu\left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right)\right] - g_y \\
\frac{\partial w}{\partial t} &= -\frac{\partial uw}{\partial x} - \frac{\partial vw}{\partial y} - \frac{\partial ww}{\partial z} - \frac{1}{\rho}\frac{\partial P}{\partial x} \\
&\quad + \frac{\partial}{\partial x}\left[\nu\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right)\right] + \frac{\partial}{\partial y}\left[\nu\left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}\right)\right] + \frac{\partial}{\partial z}\left[2\nu\frac{\partial w}{\partial z}\right] - g_z,
\end{aligned} \tag{B.2}$$

where P is the pressure, g_x, g_y, g_z are the volume forces in the x, y and z directions, respectively.

B.2.2 Discretization of the Advection and Diffusion Terms

U-velocity

$$\begin{aligned}
\frac{\partial uv}{\partial x} &= \frac{1}{dx}\left(\left[\frac{U_{i+1,j,k}+U_{i,j,k}}{2}\right]^2-\left[\frac{U_{i,j,k}+U_{i-1,j,k}}{2}\right]^2\right)\\
\frac{\partial uv}{\partial y} &= \frac{0.25}{dy}\left(\left[\left(U_{i,j,k}+U_{i,j+1,k}\right)\left(V_{i,j,k}+V_{i+1,j,k}\right)\right]\right.\\
&\quad \left.-\left[\left(U_{i,j,k}+U_{i,j-1,k}\right)\left(V_{i,j-1,k}+V_{i+1,j-1,k}\right)\right]\right)\\
\frac{\partial uw}{\partial z} &= \frac{0.25}{dz}\left(\left[\left(U_{i,j,k}+U_{i,j,k+1}\right)\left(W_{i,j,k}+W_{i+1,j,k}\right)\right]\right.\\
&\quad \left.-\left[\left(U_{i,j,k}+U_{i,j,k-1}\right)\left(W_{i,j,k-1}+W_{i+1,j,k-1}\right)\right]\right) \qquad \text{(B.3)}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial}{\partial x}\left[2\nu\frac{\partial u}{\partial x}\right] &= \frac{1}{dx}\left[2\nu\left(\frac{U_{i+1,j,k}-U_{i,j,k}}{dx}\right)-2\nu\left(\frac{U_{i,j,k}-U_{i-1,j,k}}{dx}\right)\right]\\
\frac{\partial}{\partial y}\left[\nu\left(\frac{\partial u}{\partial y}+\frac{\partial v}{\partial x}\right)\right] &= \frac{1}{dy}\left[\nu\left(\frac{U_{i,j+1,k}-U_{i,j,k}}{dy}+\frac{V_{i+1,j,k}-V_{i,j,k}}{dx}\right)\right.\\
&\quad \left.-\nu\left(\frac{U_{i,j,k}-U_{i,j-1,k}}{dy}+\frac{V_{i+1,j-1,k}-V_{i,j-1,k}}{dx}\right)\right]\\
\frac{\partial}{\partial z}\left[\nu\left(\frac{\partial u}{\partial z}+\frac{\partial w}{\partial x}\right)\right] &= \frac{1}{dz}\left[\nu\left(\frac{U_{i,j,k+1}-U_{i,j,k}}{dz}+\frac{W_{i+1,j,k}-W_{i,j,k}}{dx}\right)\right.\\
&\quad \left.-\nu\left(\frac{U_{i,j,k}-U_{i,j,k-1}}{dz}+\frac{W_{i+1,j,k-1}-W_{i,j,k-1}}{dx}\right)\right] \qquad \text{(B.4)}
\end{aligned}$$

V-velocity

$$\begin{aligned}
\frac{\partial uv}{\partial x} &= \frac{0.25}{dx}\left(\left[\left(U_{i,j,k}+U_{i,j+1,k}\right)\left(V_{i+1,j,k}+V_{i,j,k}\right)\right]\right. \\
&\quad \left.-\left[\left(U_{i-1,j,k}+U_{i-1,j+1,k}\right)\left(V_{i,j,k}+V_{i-1,j,k}\right)\right]\right) \\
\frac{\partial vv}{\partial y} &= \frac{1}{dx}\left(\left[\frac{V_{i,j+1,k}+V_{i,j,k}}{2}\right]^2-\left[\frac{V_{i,j,k}+V_{i,j-1,k}}{2}\right]^2\right) \\
\frac{\partial vw}{\partial z} &= \frac{0.25}{dz}\left(\left[\left(V_{i,j,k}+V_{i,j,k+1}\right)\left(W_{i,j,k}+W_{i,j+1,k}\right)\right]\right. \\
&\quad \left.-\left[\left(V_{i,j,k}+U_{i,j,k-1}\right)\left(W_{i,j,k-1}+W_{i,j+1,k-1}\right)\right]\right) \qquad \text{(B.5)}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial}{\partial x}\left[\nu\left(\frac{\partial v}{\partial x}+\frac{\partial u}{\partial y}\right)\right] &= \frac{1}{dx}\left[\nu\left(\frac{V_{i+1,j,k}-V_{i,j,k}}{dx}+\frac{U_{i,j+1,k}-U_{i,j,k}}{dy}\right)\right. \\
&\quad \left.-\nu\left(\frac{V_{i,j,k}-V_{i-1,j,k}}{dy}+\frac{U_{i-1,j+1,k}-U_{i-1,j,k}}{dy}\right)\right] \\
\frac{\partial}{\partial z}\left[2\nu\frac{\partial u}{\partial x}\right] &= \frac{1}{dy}\left[2\nu\left(\frac{V_{i,j+1,k}-V_{i,j,k}}{dy}\right)-2\nu\left(\frac{V_{i,j,k}-V_{i,j-1,k}}{dy}\right)\right] \\
\frac{\partial}{\partial z}\left[\nu\left(\frac{\partial u}{\partial z}+\frac{\partial w}{\partial x}\right)\right] &= \frac{1}{dz}\left[\nu\left(\frac{V_{i,j,k+1}-V_{i,j,k}}{dz}+\frac{W_{i,j+1,k}-W_{i,j,k}}{dy}\right)\right. \\
&\quad \left.-\nu\left(\frac{V_{i,j,k}-V_{i,j,k-1}}{dz}+\frac{W_{i,j+1,k-1}-W_{i,j,k-1}}{dy}\right)\right] \qquad \text{(B.6)}
\end{aligned}$$

W-velocity

$$\begin{aligned}
\frac{\partial uw}{\partial y} &= \frac{0.25}{dx}\left(\left[\left(U_{i,j,k}+U_{i,j,k+1}\right)\left(W_{i+1,j,k}+W_{i,j,k}\right)\right]\right. \\
&\quad \left.-\left[\left(U_{i-1,j,k}+U_{i-1,j,k+1}\right)\left(W_{i,j,k}+W_{i-1,j,k}\right)\right]\right) \\
\frac{\partial vw}{\partial y} &= \frac{0.25}{dy}\left(\left[\left(V_{i,j,k}+V_{i,j,k+1}\right)\left(W_{i,j+1,k}+W_{i,j,k}\right)\right]\right. \\
&\quad \left.-\left[\left(V_{i,j-1,k}+V_{i,j-1,k+1}\right)\left(W_{i,j,k}+W_{i,j-1,k}\right)\right]\right) \\
\frac{\partial ww}{\partial z} &= \frac{1}{dz}\left(\left[\frac{W_{i,j,k+1}+W_{i,j,k}}{2}\right]^2-\left[\frac{W_{i,j,k}+W_{i,j,k-1}}{2}\right]^2\right) \qquad \text{(B.7)}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial}{\partial x}\left[\nu\left(\frac{\partial w}{\partial x}+\frac{\partial u}{\partial z}\right)\right] &= \frac{1}{dx}\left[\nu\left(\frac{W_{i+1,j,k}-W_{i,j,k}}{dx}+\frac{U_{i,j,k+1}-U_{i,j,k}}{dz}\right)\right. \\
&\quad \left.-\nu\left(\frac{W_{i,j,k}-W_{i-1,j,k}}{dx}+\frac{U_{i-1,j,k+1}-U_{i-1,j,k}}{dz}\right)\right] \\
\frac{\partial}{\partial y}\left[\nu\left(\frac{\partial w}{\partial y}+\frac{\partial v}{\partial z}\right)\right] &= \frac{1}{dy}\left[\nu\left(\frac{W_{i,j+1,k}-W_{i,j,k}}{dy}+\frac{V_{i,j,k+1}-V_{i,j,k}}{dz}\right)\right. \\
&\quad \left.-\nu\left(\frac{W_{i,j,k}-W_{i,j-1,k}}{dy}+\frac{V_{i,j-1,k+1}-V_{i,j-1,k}}{dz}\right)\right] \\
\frac{\partial}{\partial z}\left[2\nu\frac{\partial w}{\partial z}\right] &= \frac{1}{dz}\left[2\nu\left(\frac{W_{i,j,k+1}-W_{i,j,k}}{dz}\right)-2\nu\left(\frac{W_{i,j,k}-W_{i,j,k-1}}{dz}\right)\right] \qquad \text{(B.8)}
\end{aligned}$$

B.3 Strain Rate Tensor

$$
\begin{aligned}
S_{ij}S_{ij} = \\
&4\left(\frac{U_{i,j,k}-U_{i-1,j,k}}{dx}\right)^2+4\left(\frac{V_{i,j,k}-V_{i,j-1,k}}{dy}\right)^2+4\left(\frac{W_{i,j,k}-W_{i,j,k-1}}{dz}\right)^2 \\
&+0.5\left[\left(\frac{U_{i,j+1,k}-U_{i,j,k}}{dy}+\frac{V_{i+1,j,k}-V_{i,j,k}}{dx}\right)^2+\left(\frac{U_{i,j,k}-U_{i,j-1,k}}{dy}+\frac{V_{i+1,j,k}-V_{i,j-1,k}}{dx}\right)^2\right. \\
&\left.+\left(\frac{U_{i-1,j+1,k}-U_{i-1,j,k}}{dy}+\frac{V_{i,j,k}-V_{i-1,j,k}}{dx}\right)^2+\left(\frac{U_{i-1,j,k}-U_{i-1,j-1,k}}{dy}+\frac{V_{i,j-1,k}-V_{i-1,j-1,k}}{dx}\right)^2\right] \\
&+0.5\left[\left(\frac{U_{i,j,k+1}-U_{i,j,k}}{dz}+\frac{W_{i+1,j,k}-W_{i,j,k}}{dx}\right)^2+\left(\frac{U_{i,j,k}-U_{i,j,k-1}}{dy}+\frac{W_{i+1,j,k-1}-W_{i,j,k-1}}{dx}\right)^2\right. \\
&\left.+\left(\frac{U_{i-1,j,k+1}-U_{i-1,j,k}}{dz}+\frac{W_{i,j,k}-W_{i-1,j,k}}{dx}\right)^2+\left(\frac{U_{i-1,j,k}-U_{i-1,j,k-1}}{dy}+\frac{V_{i,j,k}-V_{i,j-1,k}}{dy}\right)^2\right] \\
&+0.5\left[\left(\frac{V_{i,j,k+1}-V_{i,j,k}}{dz}+\frac{W_{i,j+1,k}-W_{i,j,k}}{dz}\right)^2+\left(\frac{V_{i,j,k}-V_{i,j,k-1}}{dz}+\frac{W_{i,j+1,k-1}-W_{i,j,k-1}}{dy}\right)^2\right. \\
&\left.+\left(\frac{V_{i,j-1,k+1}-V_{i,j-1,k}}{dz}+\frac{W_{i,j,k}-W_{i,j-1,k}}{dy}\right)^2+\left(\frac{V_{i,j-1,k}-V_{i,j-1,k-1}}{dz}+\frac{W_{i,j,k-1}-W_{i,j-1,k-1}}{dy}\right)^2\right]
\end{aligned}
\tag{B.9}
$$

APPENDIX C

CUDA CODE

C.1 Parameter Definition

```
/** Max number of GPUs **/
#define MAX_CPU_THREAD 4

/** GRID and BLOCK dimensions **/
#define GRID_SIZE_X  8
#define GRID_SIZE_Y  8
#define GRID_SIZE_Z  16

#define BLOCK_SIZE_X  16
#define BLOCK_SIZE_Y  8

#define SIZE_Z  2

/** discretization parameters **/

#define NX (GRID_SIZE_X*BLOCK_SIZE_X)
#define NY (GRID_SIZE_Y*BLOCK_SIZE_Y)
#define NZ (GRID_SIZE_Z*SIZE_Z)

#define L_NX 16.0f
#define L_NY 2.0f
#define L_NZ 8.0f

#define dx   ((float)(L_NX/((float)(NX-2))))
#define dy   ((float)(L_NY/((float)(NY-2))))
#define dz   ((float)(L_NZ/((float)(NZ-2))))
#define dxi  ((float)(1.0f/dx))
#define dyi  ((float)(1.0f/dy))
```

```
#define dzi  ((float)(1.0f/dz))
#define dxi2 ((float)(dxi*dxi))
#define dyi2 ((float)(dyi*dyi))
#define dzi2 ((float)(dzi*dzi))

#define B ((float)(0.5 / ( 1./(dx*dx) + 1./(dy*dy) + 1./(dz*dz) )))

/** Navier-Stokes parmeters **/

#define NU     0.4f
#define GAMMA  0.00f
#define CFL    0.10f
#define dt_max ((float)(0.4 * dz * dz / NU))

#define u_inlet 1.0f
#define v_inlet 0.0f
#define w_inlet 0.0f
#define u_max   ((float)(1.5f*u_inlet))

#define nITERATIONS 20

#define Re (u_inlet * L_NZ / NU)

#define Cs 0.01f
#define delta  ((float)(powf((dx*dy*dz),1.0/3.0)))
#define Cs_delta_2 ((float)((Cs*delta)*(Cs*delta)))
```

C.2 Memory Indexing

```
/* define positions in the global memory */
I = gridDim.x*blockDim.x*blockDim.y*(blockIdx.y % GRID_SIZE_Y)
   + threadIdx.y*gridDim.x*blockDim.x
   + blockIdx.x*blockDim.x
   + threadIdx.x;
I = I + (NX*NY*SIZE_Z)*(blockIdx.y/GRID_SIZE_Y);

/* define positions in the shared memory */
c  = (threadIdx.x + 1) + (threadIdx.y + 1)*(BLOCK_SIZE_X + 2); //center
jp = c + (BLOCK_SIZE_X + 2); //north neighbor
jm = c - (BLOCK_SIZE_X + 2); //south neighbor
```

C.3 Momentum

C.3.1 Global Memory Implementation

```
//---------------------------------------------------------------------
/* U-component */
//---------------------------------------------------------------------
//diffusion
diff = dxi2*2.0*NU*(( d_u[I+1 + k*NX*NY]- d_u[I + k*NX*NY])
                  - ( d_u[I + k*NX*NY]  - d_u[I-1 + k*NX*NY] ))
     + dyi*NU*((d_u[I+NX + k*NX*NY]  - d_u[I + k*NX*NY] )*dyi
            + ( d_v[I+1 + k*NX*NY]   - d_v[I + k*NX*NY] )*dxi
            - ((d_u[I + k*NX*NY]     - d_u[I-NX + k*NX*NY] )*dyi
            + ( d_v[I+1-NX + k*NX*NY]- d_v[I-NX + k*NX*NY] )*dxi))
     + dzi*NU*((d_u[I + (k+1)*NX*NY] - d_u[I + k*NX*NY])*dzi
            + ( d_w[I+1 + k*NX*NY]   - d_w[I + k*NX*NY] )*dxi
            - ((d_u[I + k*NX*NY]     - d_u[I + (k-1)*NX*NY])*dzi
            + ( d_w[I+1+(k-1)*NX*NY] - d_w[I + (k-1)*NX*NY] )*dxi));
//advection
adv = dxi*0.25
      * ((d_u[I + k*NX*NY] + d_u[I+1 + k*NX*NY])
        *(d_u[I + k*NX*NY] + d_u[I+1 + k*NX*NY])
        -(d_u[I-1 + k*NX*NY] + d_u[I + k*NX*NY])
        *(d_u[I-1 + k*NX*NY] + d_u[I + k*NX*NY]))
       + GAMMA*dxi*0.25
      * ((fabsf(d_u[I + k*NX*NY] + d_u[I+1 + k*NX*NY])
             * (d_u[I + k*NX*NY] - d_u[I+1 + k*NX*NY]))
       - (fabsf(d_u[I-1 + k*NX*NY] + d_u[I + k*NX*NY])
             * (d_u[I-1 + k*NX*NY] - d_u[I + k*NX*NY])))
    + dyi*0.25
     * ((d_v[I + k*NX*NY] + d_v[I+1 + k*NX*NY] )
      * (d_u[I + k*NX*NY] + d_u[I+NX + k*NX*NY])
      - (d_v[I - NX + k*NX*NY] + d_v[I+1-NX + k*NX*NY])
      * (d_u[I - NX + k*NX*NY] + d_u[I + k*NX*NY]))
      + GAMMA*dyi*0.25
     * ((fabsf(d_v[I + k*NX*NY]+ d_v[I+1 + k*NX*NY])
            * (d_u[I + k*NX*NY] - d_u[I+NX + k*NX*NY]))
      - (fabsf(d_v[I - NX + k*NX*NY] + d_v[I+1-NX + k*NX*NY])
            * (d_u[I - NX + k*NX*NY] - d_u[I + k*NX*NY])))
    + dzi*0.25
     * ((d_w[I + k*NX*NY] + d_w[I+1 + k*NX*NY])
```

```
      * (d_u[I + k*NX*NY] + d_u[I + (k+1)*NX*NY])
      - (d_w[I + (k-1)*NX*NY] + d_w[I+1 + (k-1)*NX*NY])
      * (d_u[I + (k-1)*NX*NY] + d_u[I + k*NX*NY]))
     + GAMMA*dzi*0.25
     * ((fabsf(d_w[I + k*NX*NY] + d_w[I+1 + k*NX*NY])
           * (d_u[I + k*NX*NY] - d_u[I + (k+1)*NX*NY]))
      - (fabsf(d_w[I + (k-1)*NX*NY] + d_w[I+1 + (k-1)*NX*NY])
           * (d_u[I + (k-1)*NX*NY] - d_u[I + k*NX*NY])));

d_unew[I + k*NX*NY] = d_u[I + k*NX*NY] + dt*(-adv + diff);
```

C.3.2 Shared Memory Implementation

```
//---------------------------------------------------------------------
/* U-component */
//---------------------------------------------------------------------

//diffusion
diff = dxi2*2.0*NU*(( s_u[k][c+1]- s_u[k][c])
                  - ( s_u[k][c]  - s_u[k][c-1] ))
          + dyi*NU*((s_u[k][jp]  - s_u[k][c] )*dyi
                 + ( s_v[k][c+1] - s_v[k][c] )*dxi
                 - ((s_u[k][c]   - s_u[k][jm])*dyi
                 + ( s_v[k][jm+1]- s_v[k][jm])*dxi))
          + dzi*NU*((s_u[k+1][c]  - s_u[k][c]  )*dzi
                 + ( s_w[k][c+1]  - s_w[k][c]  )*dxi
                 - ((s_u[k][c]    - s_u[k-1][c])*dzi
                 + ( s_w[k-1][c+1]- s_w[k-1][c])*dxi));

//advection
adv = dxi*0.25
        * ((s_u[k][c] + s_u[k][c+1])*(s_u[k][c] + s_u[k][c+1])
         - (s_u[k][c-1] + s_u[k][c])*(s_u[k][c-1] + s_u[k][c]))
        + GAMMA*dxi*0.25
        * ((fabsf(s_u[k][c] + s_u[k][c+1]) * (s_u[k][c] - s_u[k][c+1]))
         - (fabsf(s_u[k][c-1] + s_u[k][c]) * (s_u[k][c-1] - s_u[k][c])))
   + dyi*0.25
        * ((s_v[k][c] + s_v[k][c+1] ) * (s_u[k][c] + s_u[k][jp])
         - (s_v[k][jm] + s_v[k][jm+1]) * (s_u[k][jm] + s_u[k][c]))
        + GAMMA*dyi*0.25
        * ((fabsf(s_v[k][c]  + s_v[k][c+1]) * (s_u[k][c] - s_u[k][jp]))
```

```
          - (fabsf(s_v[k][jm] + s_v[k][jm+1]) * (s_u[k][jm] - s_u[k][c])))
   + dzi*0.25
       * ((s_w[k][c] + s_w[k][c+1]) * (s_u[k][c] + s_u[k+1][c])
        - (s_w[k-1][c] + s_w[k-1][c+1]) * (s_u[k-1][c] + s_u[k][c]))
       + GAMMA*dzi*0.25
       * ((fabsf(s_w[k][c]  + s_w[k][c+1])  * (s_u[k][c] - s_u[k+1][c]))
        - (fabsf(s_w[k-1][c]+ s_w[k-1][c+1])* (s_u[k-1][c] - s_u[k][c])));

d_unew[I + (k-1)*NX*NY] = s_u[k][c] + dt*(-adv + diff);
```

C.4 Divergence

```
d_div[I+k*NX*NY] = (d_unew[I+k*NX*NY] - d_unew[I-1 + k*NX*NY])  *dxi
                 + (d_vnew[I+k*NX*NY] - d_vnew[I-NX + k*NX*NY]) *dyi
                 + (d_wnew[I+k*NX*NY] - d_wnew[I + (k-1)*NX*NY])*dzi;
```

C.5 Pressure

C.5.1 Pressure Constant

```
#define B    ((float)(0.5 / ( 1./(dx*dx) + 1./(dy*dy) + 1./(dz*dz) )))
```

C.5.2 Global Memory Implementation

```
A = (d_p[I+1 + k*NX*NY] + d_p[I-1 + k*NX*NY]) *dxi2
  + (d_p[I+NX + k*NX*NY]+ d_p[I-NX + k*NX*NY])*dyi2
  + (d_p[I+(k+1)*NX*NY] + d_p[I+(k-1)*NX*NY]) *dzi2;

d_pnew[I+k*NX*NY] = -B * (dti * d_div[I+k*NX*NY] - A);
```

C.5.3 Shared Memory Implementation

```
A = (s_p[k][c+1] + s_p[k][c-1])*dxi2
```

```
  + (s_p[k][jp] + s_p[k][jm])*dyi2
  + (s_p[k+1][c] + s_p[k-1][c])*dzi2;

d_pnew[I + (k-1)*NX*NY] = -B * (dti * d_div[I + (k-1)*NX*NY] - A);
```

C.6 Velocity Correction

```
d_unew[I+k*NX*NY] = d_unew[I+k*NX*NY]
                  - dt*dxi *(d_pnew[I+1 + k*NX*NY] - d_pnew[I+k*NX*NY]);
d_vnew[I+k*NX*NY] = d_vnew[I+k*NX*NY]
                  - dt*dyi *(d_pnew[I+NX + k*NX*NY]- d_pnew[I+k*NX*NY]);
d_wnew[I+k*NX*NY] = d_wnew[I+k*NX*NY]
                  - dt*dzi *(d_pnew[I+ (k+1)*NX*NY]- d_pnew[I+k*NX*NY]);
```