EFFICIENT COMPUTATIONAL FLUID DYNAMICS METHODS FOR GPGPUs

JASON NORMORE

Efficient Computational Fluid Dynamics Methods for GPGPUs

by

Jason Normore

A thesis submitted to the

School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Master of Science

Computational Science

Memorial University of Newfoundland

November, 2010

St. John's, Newfoundland

Abstract

Computational fluid dynamics (CFD) is an area of fluid mechanics that involves using numerical methods to solve fluid systems. Most practical CFD problems involve solving a minimum of two nonlinear coupled partial differential equations, which is computationally expensive for practical fluid systems. The methods proposed in this thesis take advantage of the parallelism of the graphics processing unit (GPU) to increase the efficiency of two CFD techniques for general purpose fluid flow simulations. The improved techniques produce very good results for increased efficiency, while keeping the overall methods practical and able to run on readily available and inexpensive GPU hardware. We discuss the advantages and disadvantages of the techniques developed, along with how different techniques affect the results, applications of the developed methods, and possible extensions to the methods.

Acknowledgements

We acknowledge NSERC, ACENET, and NVIDIA for their support of this work. Acknowledgments are also made to my supervisor, Wolfgang Banzhaf, for his support and guidance throughout the course of the development of this work. Finally, acknowledgments are made to Simon Harding for his support and guidance in both the design and development of this work, through collaboration on a related project, and of this thesis.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Computational fluid dynamics (CFD) is an area of fluid mechanics that involves using numerical methods to solve fluid systems. Most practical CFD problems involve solving a minimum of two nonlinear coupled partial differential equations, which is computationally expensive for practical fluid systems.

The objective of this work is

- to develop efficient and accurate methods for simulating and analyzing general purpose fluid flow problems for the parallel architecture of graphics processing units (GPUs).
- to compare several performance measures of CFD on GPUs against those of CFD on a traditional CPU.
- to analyze different CFD solution methods for the purpose of determining which are best suited for general purpose CFD on GPUs.
- to describe applications that are good candidates for CFD on GPUs.

If, for example, an evolutionary algorithm was used in a shape optimization method where each evaluation was a solution of a fluid system for the purpose of minimizing the drag and/or maximizing the lift (or any other physical variables), current CFD solvers would be infeasible for use in a reasonable time period since an evolutionary algorithm could require millions of evaluations (CFD solutions). A more efficient solution method is thus required to perform so many evaluations, to successfully converge toward an optimal shape. This is the context in which the system described in this thesis was designed, as a fitness function in a genetic programming technique used for optimized shape design.

This work builds on previously developed CFD methods for general purpose fluid flow in its development of highly parallel CFD methods on GPUs. Chapter 2 is a description of

some general CFD theory and of these already-defined CFD methods.

The method proposed takes advantage of the parallelism of graphics processing units (GPUs). GPUs are in almost all modern computers used mainly for video and graphics display. They are many-core (like) processing units, originally designed for operating in the graphics pipeline on individual pixels, but as their computational power increased they became popular for scientific and general purpose computation. Chapter 3 is a review of the GPU architecture and general programming techniques for GPUs.

The proposed methods involve the design of an algorithm for CFD solution methods that take advantage of the GPU parallelism. Since the initial conception and development of this work there have been some algorithms designed and implemented that accomplish the task of fluid simulations on GPUs. Chapter 4 is a literature review of some of the current methods for CFD on GPUs.

The proposed methods are still unique in that they take advantage of different optimization techniques, such as smart register usage and shared memory usage. Chapter 5 describes the methods used in the design and implementation of the proposed CFD technique on GPUs, along with results and the analysis of the results in chapter 6.

Computational fluid dynamics is a large field, with many applications and solution methods. This thesis defines algorithms not only for a single CFD method, but for multiple methods, and describes in detail their application (with results) to an optimized shape design technique. Chapter 8 describes some of the more important applications of the methods developed in this thesis.

Contributions made through this work are

- the development of efficient and accurate methods for simulating and analyzing general purpose fluid flow problems for GPUs.
- the analysis of several different CFD solution methods and determination of which

are best suited for general purpose CFD on GPUs.

- an analysis of several applications that are candidates for CFD on GPUs.

2 Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) is an area of fluid mechanics that uses numerical methods and algorithms to solve fluid flow problems. Within CFD itself there are many different areas, involving different solution methods. The choice of a solution method is largely dependent on the problem, on the context of the problem, and on what is required of a solution for later analysis (post-processing).

This chapter will give a brief overview of current CFD solution methods and what their advantages and disadvantages are for different types of problems. It will start with the governing equations for fluid flow, and to solve these equations we must discretize the governing equations, and in order to do that we must choose a grid type for this discretization. The following section will cover some of these discretization techniques, followed by a section on the different grid types. We will then discuss solution techniques for solving the discretized governing equations, and the advantages/disadvantages of these techniques for different types of problems. Finally, we will discuss and compare several different methods for solving systems of linear equations.

2.1 Governing Equations

The governing equations of a fluid system are at a minimum the continuity equation for mass and the Navier-Stokes equation, although others may be applied as required by the system in question, such as the equation of state, conservation of mass, conservation of energy, and/or boundary condition equations. Since our application of fluid dynamics required only the continuity equation for mass and the Navier-Stokes equation, that will be the limit of what we discuss in this section.

The continuity equation is a description of the transport of mass with a conservation of that mass,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{u}) = 0 \tag{1}$$

where ρ is the density of the fluid, t is the time, and $\vec{u}$ is the velocity vector [16].

Since this work is concerned with incompressible fluid flow, we need the incompressible Navier-Stokes equation,

$$\rho \left(\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} \right) = -\nabla p + \mu \nabla^2 \vec{u} + \vec{f} \tag{2}$$

where p is the pressure, and $\vec{f}$ is any external forces [16]. Equations (1) and (2) comprise the required equations for solving incompressible transient (time dependent) fluid flow. It is important to understand that not all solutions to fluid flow are required to be transient, some simple flows have time independent solutions, or steady-state solutions. The steady-state governing equations for incompressible fluid flow are [12]

$$\nabla \cdot (\rho \vec{u}) = 0 \tag{3}$$

and

$$\rho (\vec{u} \cdot \nabla \vec{u}) = -\nabla p + \mu \nabla^2 \vec{u} + \vec{f} \tag{4}$$

2.2 Discretization Techniques

Once a mathematical model is defined (i.e. Equations (3) and (4) define the mathematical model for incompressible fluid flow) we need a method for approximating the differential equations by obtaining a system of equations at a set of discrete points in space and time. Many different discretization methods can be used, which should all give the same result in the limit of a very fine mesh, but the three main methods used in most commercial and

academic applications are the finite difference method (FDM), the finite element method (FEM), and the finite volume method (FVM) [12]. A description of each of the three main discretization methods follows.

2.2.1 Finite Difference Method

The finite difference method (FDM) is the oldest method for numerically solving PDEs, it is believed to have been introduced by Euler in the 18th century [1]. The finite difference method covers the problem domain by a grid. Any PDE's in the mathematical model are used in differential form, and at each grid point the differential equations are approximated by using partial derivative approximations (such as a central difference approximation, or a higher order approximation, depending on the accuracy requirements). Most of the time the FDM is applied to a structured grid, but it can be applied to any grid type (See Section 2.3 for description of different grid types).

2.2.2 Finite Element Method

The finite element method (FEM) is similar to the finite volume method (FVM), in that the problem domain is divided into a set of discrete volumes (see the next section), or finite elements as the name of the method suggests. These finite elements are generally unstructured, normally triangles or quadrilaterals in 2D, and tetrahedrals or hexahedrals in 3D. The FEM approximates the solution by a shape function within each element in a way that guarantees continuity across the element boundaries [12].

The main advantage of the FEM is that it can handle complex or arbitrary geometries quite easily, and its grid is easily refined. The main disadvantage is that, as with any method using unstructured grids, the matrices of the linearized equations are not structured as well as for structured grids, which would make an efficient solution more difficult to produce [12].

2.2.3 Finite Volume Method

The finite volume method (FVM) is the chosen method for the applications described in this work. The finite volume method uses an integral form of the conservation equations (1) to (4). This method requires the problem domain be divided into control volumes (CVs) so that the integral form of the equation can be applied over each control volume. The reason for the popularity of the FVM is that it conserves the equations in question over each CV, leading to a more stable solution than the FDM for many problems that involve conservation equations (such as fluid simulations), and that it is easily formulated over unstructured grids. For example, if we take the steady-state incompressible Navier-Stokes equation (4), neglecting the external force term for simplicity, the first step in discretization using the FVM is to integrate over each CV,

$$\rho \int_{CV} (\vec{u} \cdot \nabla \vec{u}) \, dV = - \int_{CV} \nabla p \, dV + \int_{CV} \mu \nabla^2 \vec{u} \, dV \tag{5}$$

By using Gauss' divergence theorem,

$$\int_{CV} \nabla \cdot \vec{a} \, dV = \int_{A} \vec{n} \cdot \vec{a} \, dA \tag{6}$$

parts of the conservation equations can be simplified to integrate over the entire surface of the CV [32].

The advantage, as mentioned previously, that the solution is conserved over each CV, leading to more stable solutions. The FVM is also very well suited for complex geometries due to its capability to handle unstructured grids and that it conserves the solution over each CV. The main advantage is the simplicity of the method, it is simple to understand and program, and all terms that are approximated have some physical meaning, which explains its popularity amongst engineers [12]. One of the main disadvantages of the FVM method

is that it is difficult to develop solutions of higher than second order because it requires interpolation, differentiation, and integration [12].

2.3 Grids and Mesh Types

A mesh, or grid (these terms will be used interchangeably throughout this thesis), is a discretization of a geometric domain into small simple shapes, such as triangles or quadrilaterals in 2D and tetrahedrals and hexahedrals in 3D [4]. With respect to solving numerical problems, such as the simulation of a fluid system, there are three main types of meshes: structured meshes, unstructured meshes, and hybrid meshes.

First we discuss the structured mesh, also called a regular mesh, which is named so because the grid is laid out in a regular repeating pattern. The simplest form of this mesh is a single repeating topographical pattern such as a grid of quadrahedrals as shown in Fig. 2.1, but a structured mesh can also be a set of different repeating patterns, sometimes called a block-structured mesh, as shown in Fig. 2.2. The main benefit of the use of a structured mesh is its simplicity, it is easy to understand and construct, and also simple to implement in most programming language (a simple array). Drawbacks of the use of this type of mesh is in its difficulty in accurately handling complex geometries in an efficient manner, and that it may be difficult and time consuming to construct a structured mesh around an arbitrary shape [8].

The second type of mesh is the unstructured mesh, which uses an arbitrary set of shapes or elements to cover the problem domain, where the shapes are not required to have a pattern. An example is given in Figure 2.3. A benefit of the use of this type of grid is that it can easily handle complex geometries and conditioning the mesh is much simpler. Drawbacks of this type of mesh is that it is more computationally expensive to generate, it uses much more memory because all control volumes must be stored along with their locations and any neighbor information required, and it is much more difficult to implement

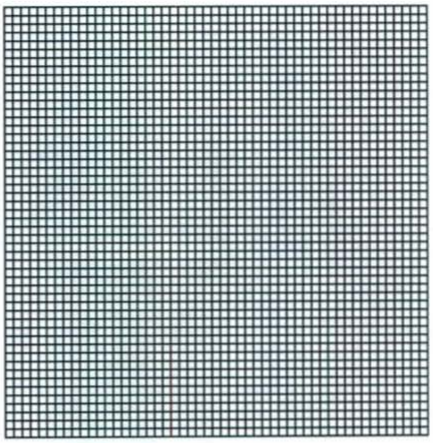

Figure 2.1: Simple structured mesh with one repeating pattern

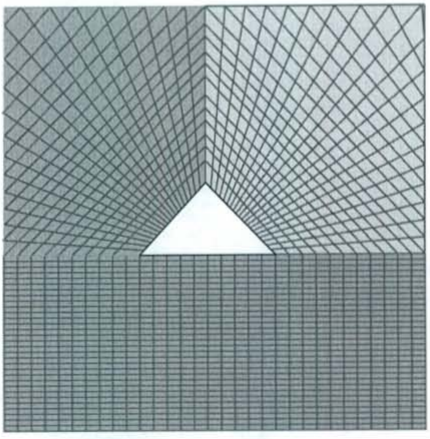

Figure 2.2: Structured mesh with multiple repeating patterns

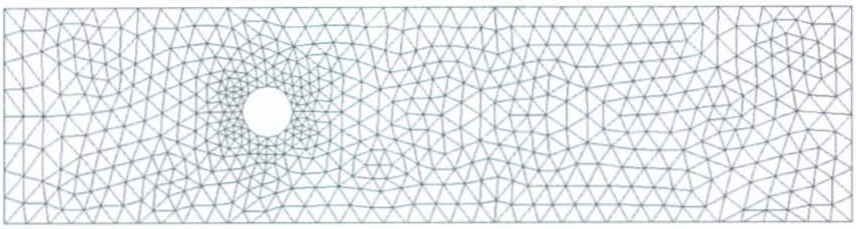

Figure 2.3: Unstructured mesh around circle

because it requires lookup tables for neighboring control volumes.

Third, and finally, is the hybrid mesh, which as its name suggests is a mix between a structured and unstructured mesh. A hybrid mesh is a combination of small structured meshes in an overall unstructured pattern [4]. This type of mesh has many of the same advantages of the previous two meshes, such as easy grid refinement near boundaries and require less memory. The disadvantages are that this type of mesh is difficult to implement and to generate for complex systems, and since it requires a lot of user interaction it is slow to generate compared to, for example, an automatically generated unstructured mesh.

2.4 Methods for Solving Fluid System

This section will describe two of the methods used in this thesis for solving the governing equations of fluid flow. Since the governing equations that are used in this thesis are nonlinear and coupled by pressure, both solvers are in the category of 'pressure correction' methods, which is a guess-and-correct iterative approach for solving these types of equations [30]. The first method that will be discussed is for solving the steady-state equations, equations (3) and (4), and is called the **S**emi-**I**mplicit **M**ethod for **P**ressure **L**inked **E**quations (SIMPLE) method. The second method is for solving the transient governing flow equations, equations (1) and (2), and is called the **P**ressure **I**mplicit with **S**plitting of **O**perators (PISO) method.

2.5 The SIMPLE Method

The SIMPLE (Semi-Implicit Method for Pressure Linked Equations) method was first developed in 1972 by Patankar and Spalding, [24], as a means to solve nonlinear coupled fluid flow equations. This method is an iterative solver that uses a guess-and-correct procedure at each iteration to converge towards an approximation to the exact solution.

The reason this method is required to solve the governing equations given in Section 2.1 is that these equations are coupled by the pressure field and that the Navier-Stokes equations are nonlinear (because of the velocity variable). Therefore we must decouple these equations and linearize them. The decoupling of the equations (through the pressure field) is accomplished by beginning each iteration with a guessed pressure field, initially this would be user-defined. During subsequent iterations a corrected pressure field is used as the guessed pressure field [21].

This section will first present a general description of the SIMPLE algorithm, for which any discretization technique can be applied. Then we will present an outline of a general derivation of the SIMPLE method, again for which any of the discretization methods described in Section 2.2 can be applied. It may be useful to follow along with the general algorithm in Section 2.5.1 while going through the derivation in Section 2.5.2.

2.5.1 The Algorithm

A general algorithm for the SIMPLE method is described in Algorithm 2.1 [32]. For more information on the exact nature of each step, see Section 2.5.2 for a complete derivation and definition of this method.

2.5.2 Derivation

Here we will outline the general derivation [32] of the 2 dimensional SIMPLE method for the steady-state fluid problem (since we use it only for the steady-state problem in this

Algorithm 2.1 SIMPLE algorithm

1: Initialize guesses for p^*, u^*, v^*.
2: **while** !convergence **do**
3: // STEP 1: Solve discretized momentum equations to get u^*, and v^*
4: $a_{i,j}u^* = \sum a_{nb}u^*_{nb} + \frac{1}{2}(p^*_{i-1,j} - p^*_{i+1,j})A_{i,j} + b_{i,j}$
5: $a_{i,j}v^* = \sum a_{nb}v^*_{nb} + \frac{1}{2}(p^*_{i,j-1} - p^*_{i,j+1})A_{i,j} + b_{i,j}$
6:
7: // STEP 2: Solve pressure correction equation to get p'
8: $a_{i,j}p'_{i,j} = a_{i-1,j}p'_{i-1,j} + a_{i+1,j}p'_{i+,j} + a_{i,j-1}p'_{i,j-1} + a_{i,j+1}p'_{i,j+1} + b_{i,j}$
9:
10: // STEP 3: Correct pressure and velocities
11: $p_{i,j} = p^*_{i,j} + p'_{i,j}$
12: $u_{i,j} = u^*_{i,j} + \frac{1}{2}d\_u_{i,j}(p'_{i-1,j} - p'_{i+1,j})$
13: $v_{i,j} = v^*_{i,j} + \frac{1}{2}d\_v_{i,j}(p'_{i,j-1} - p'_{i,j-1})$
14:
15: $p^* = p$; $u^* = u$; $v^* = v$
16: **end while**

work) on a structured grid, which can be easily extended to higher dimensions and/or to a time-dependent problem. Similar to other guess-and-correct procedures, the SIMPLE method must begin with a "guessed" velocity field. Therefore the first step in the SIMPLE iteration is to solve the 2 dimensional finite-volume discretized momentum equations for a structured grid (see appendix A for a derivation of these equations) for the guessed velocity values u^* and v^*

$$a_P u^*_P = \sum a_{nb}u^*_{nb} + \frac{1}{2}\left(p^*_w - p^*_e\right)dy + b_P \tag{7}$$

$$a_P v^*_P = \sum a_{nb}v^*_{nb} + \frac{1}{2}\left(p^*_s - p^*_n\right)dx + b_P \tag{8}$$

where we use the east-west-north-south notation for neighboring nodes (capital E,W,S,N,P, where P is the local point) and neighboring faces (lowercase e,w,s,n) between nodes as defined in appendix A, except for coefficient indices which are unique to every node (each point P has 5 coefficients a_W, a_E, a_S, a_N, a_P). Since these equations are considered guesses

at this node, the SIMPLE method defines the two velocity and one pressure correction values (can also be viewed as the error in the guess), respectively, as

$$p = p^* + p' \tag{9}$$

$$u = u^* + u' \tag{10}$$

$$v = v^* + v' \tag{11}$$

The subtraction of the actual valued discretized equations (with u and v) and the guessed valued discretized equations (7) and (8) produces

$$a_P \left(u_P - u_P^*\right) = \sum a_{nb} \left(u_{nb} - u_{nb}^*\right) + \frac{1}{2} \left[\left(p_w - p_w^*\right) - \left(p_e - p_e^*\right)\right] dy \tag{12}$$

$$a_P \left(v_P - v_P^*\right) = \sum a_{nb} \left(v_{nb} - v_{nb}^*\right) + \frac{1}{2} \left[\left(p_s - p_s^*\right) - \left(p_n - p_n^*\right)\right] dx \tag{13}$$

Rearrangement and substitution of the correction values from equations (9), (10), (11) produces

$$a_P u'_P = \sum a_{nb} u'_{nb} + \frac{1}{2} \left(p'_w - p'_e\right) dy \tag{14}$$

$$a_P v'_P = \sum a_{nb} v'_{nb} + \frac{1}{2} \left(p'_s - p'_n\right) dx \tag{15}$$

Here the main approximation of the SIMPLE method occurs, the $\sum a_{nb} u'_{nb}$ and $\sum a_{nb} v'_{nb}$ terms are dropped to simplify the equations to

$$u'_P = \frac{1}{2} d\_u_P \left(p'_w - p'_e\right) \tag{16}$$

$$v'_P = \frac{1}{2} d\_v_P \left(p'_s - p'_n\right) \tag{17}$$

where $d\_u = dy/a_P$ and $d\_v = dx/a_P$. Now that we can correct the velocity fields using equations (10), (11), (16), and (17) all we need is the pressure correction in order to be able to apply these equations at every iteration. Since we also need to apply the continuity equation (3), we can use this to derive an equation for the pressure correction p'. The 2 dimensional finite-volume discretized continuity equation is simply

$$\left[(\rho u A)_e - (\rho u A)_w\right] + \left[(\rho v A)_n - (\rho v A)_s\right] = 0 \tag{18}$$

If we insert the corrected velocities (equations (10) and (11)) into this equation (along with equations (16) and (17) into these) we get

$$\left(\rho\left(u^*_e + \frac{1}{2} d\_u_e \left(p'_P - p'_E\right)\right) dx\right) - \tag{19}$$

$$\left(\rho\left(u^*_w + \frac{1}{2} d\_u_w \left(p'_W - p'_P\right)\right) dx\right) + \tag{20}$$

$$\left(\rho\left(v^*_n + \frac{1}{2} d\_v_n \left(p'_P - p'_N\right)\right) dy\right) - \tag{21}$$

$$\left(\rho\left(v^*_s + \frac{1}{2} d\_v_s \left(p'_S - p'_P\right)\right) dy\right) = 0 \tag{22}$$

which can be rearranged to produce the pressure correction equation

$$a_P p'_P = a_W p'_W + a_E p'_E + a_S p'_S + a_N p'_N + b'_P \tag{23}$$

where

$$a_W = \rho d\_u_w dy$$

$$a_E = \rho d\_u_e dy$$

$$a_S = \rho d\_v_s dx$$

$$a_N = \rho d\_v_n dx$$

$$a_P = a_W + a_E + a_S + a_N$$

$$b'_P = \rho u^*_w dy - \rho u^*_e dy + \rho v^*_s dx - \rho v^*_n dx$$

The pressure correction equation can then be solved to produce the pressure correction field and applied to the velocity and pressure fields using equations (16), (17), and (9).

2.6 The PISO Method

The PISO (Pressure Implicit with Splitting of Operators) method is derived directly from the SIMPLE method. Like the SIMPLE method it is an iterative method. In fact, the first three steps per iteration are exactly the same as those for the SIMPLE method. It involves two more steps to solve a second pressure correction equation and then apply this second pressure correction to the flow fields.

The PISO method uses the concept of operator splitting to derive a second pressure correction equation. This technique is used to "split" the spacial and temporal components so that we can solve for a time step at each iteration.

This method is used as the transient (time-dependent) solver in this work. The reason this method is ideal as a transient solver for the governing equations is that each iteration can be considered a time step because of the application of the second pressure correction equation [2].

Similar to Section 2.5, this section will provide a description of the PISO algorithm and a general derivation of the PISO method.

2.6.1 The Algorithm

A general algorithm for the PISO method is described in Algorithm 2.2 [32]. For more information on the exact nature of each step, see Section 2.6.2 for a complete derivation and definition of this method.

Algorithm 2.2 PISO algorithm

1: Initialize guesses for p^*, u^*, v^*.
2: **repeat**
3: // STEP 1-3
4: Exactly those of the SIMPLE method. See Section 2.5
5:
6: // STEP 4: Solve second pressure correction equation to get p'
7: $a_{i,j}p''_{i,j} = a_{i-1,j}p''_{i-1,j} + a_{i+1,j}p''_{i+,j} + a_{i,j-1}p''_{i,j-1} + a_{i,j+1}p''_{i,j+1} + b_{i,j}$
8:
9: // STEP 5: Correct pressure and velocities using second pressure correction
10: $p_{i,j} = p^*_{i,j} + p''_{i,j}$
11: $u_{i,j} = u^*_{i,j} + \frac{1}{2}d\_u_{i,j}(p''_{i-1,j} - p''_{i+1,j})$
12: $v_{i,j} = v^*_{i,j} + \frac{1}{2}d\_v_{i,j}(p''_{i,j-1} - p''_{i,j-1})$
13:
14: $p^* = p$; $u^* = u$; $v^* = v$
15: **until** convergence

2.6.2 Derivation

The PISO method is derived directly from the SIMPLE method, for this reason a repetition of the first three steps of this method will not be described here, see Section 2.5.2 for this information. The PISO method involves solving an extra pressure correction equation in addition to the steps from the SIMPLE method, and so can be loosely called a guess-and-correct-and-correct type solver. This extra pressure correction equation, that we will call the second pressure correction equation, is derived [32] by taking the discretized momentum

equations at the end of the SIMPLE method to be

$$a_P u_P^{**} = \sum a_{nb} u_{nb}^{**} + \frac{1}{2}\left(p_w^{**} - p_e^{**}\right) dy + b_P \tag{24}$$

$$a_P v_P^{**} = \sum a_{nb} v_{nb}^{**} + \frac{1}{2}\left(p_s^{**} - p_n^{**}\right) dx + b_P \tag{25}$$

where u^{**}, v^{**}, and p^{**} are the values at the end of the SIMPLE method steps. If we solve these momentum equations again, defining them as u^{***} and v^{***}, we get

$$a_P u_P^{***} = \sum a_{nb} u_{nb}^{**} + \frac{1}{2}\left(p_w^{***} - p_e^{***}\right) dy + b_P \tag{26}$$

$$a_P v_P^{***} = \sum a_{nb} v_{nb}^{**} + \frac{1}{2}\left(p_s^{***} - p_n^{***}\right) dx + b_P \tag{27}$$

If we subtract these respective equations from each other we produce

$$a_P u_P^{***} = \frac{\sum a_{nb} u_{nb}^{**}}{a_P} + \frac{1}{2} d\_u_P \left(p_w^{***} - p_e^{***}\right) \tag{28}$$

$$a_P v_P^{***} = \frac{\sum a_{nb} v_{nb}^{**}}{a_P} + \frac{1}{2} d\_v_P \left(p_s^{***} - p_n^{***}\right) \tag{29}$$

where p'' is the second pressure correction, defined in

$$p^{***} = p^{**} + p'' \tag{30}$$

If we substitute u^{***} and v^{***} into the discretized continuity equation and perform the same steps as we did to produce the first pressure correction equation (see Section 2.5.2) we arrive at the second pressure correction equation

$$a_P p''_P = a_W p''_W + a_E p''_E + a_S p''_S + a_N p''_N + b''_P \tag{31}$$

where

$$a_W = \rho d\_u_w dy$$

$$a_E = \rho d\_u_e dy$$

$$a_S = \rho d\_v_s dx$$

$$a_N = \rho d\_v_n dx$$

$$a_P = a_W + a_E + a_S + a_N$$

$$b''_P = \left(\frac{\rho A}{a}\right)_w \sum a_{nb} \left(u^{**}_{nb} - u^{*}_{nb}\right) - \left(\frac{\rho A}{a}\right)_e \sum a_{nb} \left(u^{**}_{nb} - u^{*}_{nb}\right)$$
$$\left(\frac{\rho A}{a}\right)_s \sum a_{nb} \left(v^{**}_{nb} - v^{*}_{nb}\right) - \left(\frac{\rho A}{a}\right)_n \sum a_{nb} \left(v^{**}_{nb} - v^{*}_{nb}\right)$$

2.7 Methods for Solving Systems of Linear Equations

2.7.1 Jacobi Method

The Jacobi method is a method for solving systems of linear equations, or $Ax = b$ matrix equations for x. It is named after the German mathematician Carl Gustav Jakob Jacobi and is also known as the *method of simultaneous displacements* [18].

If we define $A \in \mathbb{R}^{N \times M}$, $b \in \mathbb{R}^{1 \times N}$, and $x \in \mathbb{R}^{M \times 1}$ as

$$A = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \cdots & \alpha_{1,M} \\ \alpha_{2,1} & \ddots & & \\ \vdots & & \ddots & \\ \alpha_{N,1} & & & \alpha_{N,M} \end{pmatrix} \tag{32}$$

$$b^T = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_N \end{pmatrix} \tag{33}$$

$$x = \begin{pmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_N \end{pmatrix} \tag{34}$$

The mathematical formulation of the Jacobi method is as follows,

$$\xi_i^{k+1} = \frac{\sum_{j}^{i \neq j} \xi_j^k}{\xi_i^k} \tag{35}$$

where k is defined as the iteration number.

One main requirement of this method is that the matrix form of the system of linear equations must have non-zero diagonal elements. The method simply solves each diagonal element of the matrix and then updates the value of the diagonal elements with the newly solved approximate values after they have all been solved (approximated), as seen in equation (35). This process is iterated until convergence is reached.

Given a sparse matrix array, u, and coefficient arrays a_W, a_E, a_S, a_N, a_P, b (to represent direct neighbor nodes in a structured mesh) that we would encounter in the CFD techniques discussed in this work for a 2 dimensional system of size $m \times n$, the equation to solve at iteration k would look like

$$u_{i,j}^{[k+1]} = \frac{a\_W_{i,j} u_{i-1,j}^{[k]} + a\_E_{i,j} u_{i+1,j}^{[k]} + a\_S_{i,j} u_{i,j-1}^{[k]} + a\_N_{i,j} u_{i,j+1}^{[k]} + b_{i,j}}{a\_P_{i,j}} \tag{36}$$

for $i = 2 : m + 1$ and $j = 2 : n + 1$ represent the nodes in the CFD system (mesh). The pseudo-code (MATLAB like) might look like Algorithm 2.3.

Algorithm 2.3 Jacobi algorithm

```
1:  for iter=0:maxiter do
2:    for i = 2:(m+1) do
3:      for j = 2:(n+1) do
4:        unew(i,j) = (a_W(i,j) u(i-1,j) + a_E(i,j) u(i+1,j) + a_S(i,j) u(i,j-1) + a_N(i,j)
          u(i,j+1) + b(i,j)) / a_P(i,j)
5:      end for
6:    end for
7:    if convergence then
8:      break
9:    end if
10:   u = unew
11: end for
```

The important feature to note about this method is that a temporary array must be used to store the updated diagonal elements of the matrix (unew array in the pseodo code) so that they can all be updated at the same time at the end of each iteration (line 10), which is why this method is also known as the *method of simultaneous displacements*. This extra temporary array to store updated values requires an extra memory overhead for the application of this method of the size $m \times n$, while the worst case time complexity of this method is still $O(m \times n \times maxiter)$.

2.7.2 Gauss-Seidel Method

The Gauss-Seidel (GS) method is an iterative method for solving the same types of problems as the Jacobi method (discussed in Section 2.7.1). It is named after the German mathematicians Carl Friedrich Gauss and Philipp Ludwig von Seidel and is also known as the *method of successive displacements* [18].

The GS method is very similar to the Jacobi method in that it does an update of each diagonal element per iteration, but instead of using a temporary array to simultaneous updates

at the end of the iteration, it updates the diagonal values "on the fly".

If we describe A, b, and x as we did in the previous section, the GS formulation is as follows

$$\xi_i^{k+1} = \frac{\sum_{j=1}^{i-1} \xi_j^{k+1} + \sum_{j=i+1}^{N} \xi_j^k}{\xi_i^k} \tag{37}$$

where k, again, represents the iteration number. As it can be seen the difference is in what iteration number is used for the x elements.

Given a array for a sparse matrix, u, and coefficient arrays a_W, a_E, a_S, a_N, a_P, b (to represent direct neighbor nodes in a structured mesh) that we would encounter in the CFD techniques discussed in this work for 2 dimensional system of size $m \times n$, the equation to solve at iteration k for the GS method would look like

$$u_{i,j}^{[k+1]} = \frac{a\_W_{i,j} u_{i-1,j}^{[k+1]} + a\_E_{i,j} u_{i+1,j}^{[k]} + a\_S_{i,j} u_{i,j-1}^{[k+1]} + a\_N_{i,j} u_{i,j+1}^{[k]} + b_{i,j}}{a\_P_{i,j}} \tag{38}$$

for $i = 2 : m + 1$ and $j = 2 : n + 1$. The important difference between this equation and its corresponding Jacobi method equation (Eq. (36)) is that values from iteration $k + 1$ are used to update the current value at $k + 1$ for previously encountered nodes. The pseudo-code (MATLAB like) looks like Algorithm 2.4, where we can see that at line 4 we update the array u and not the temporary array *unew* as in the Jacobi pseudo-code. At first glance it might seem like a common mistake to forget the temporary array, but this method actually converges about twice as fast as the Jacobi method for diagonally dominant or s.p.d[1] matrices [18]. Not only does the GS method converge faster, it also has less memory overhead since it does not require the extra temporary array to store updated diagonal values.

[1] symmetric and positive definite

Algorithm 2.4 Gauss-Seidel algorithm

```
for iter=0:maxiter do
    for i = 2:(m+1) do
        for j = 2:(n+1) do
            u(i,j) = (a_W(i,j) u(i-1,j) + a_E(i,j) u(i+1,j) + a_S(i,j) u(i,j-1) + a_N(i,j) u(i,j+1)
            + b(i,j)) / a_P(i,j)
        end for
    end for
    if convergence then
        break
    end if
end for
```

2.7.3 Successive Over-relaxation Method

The successive over-relaxation (SOR) method is another variant of the Gauss-Seidel method. The SOR method is a two stage iterative update method, where the first stage is exactly the same as the Gauss-Seidel method, and the second stage is to apply a relaxation to the updated values at each iteration, through the application of a relaxation parameter. This relaxation parameter is denoted by the symbol ω here and is less than 1 for under-relaxation and greater than 1 for over-relaxation. If $\omega = 1$ then the SOR method is identical to the Gauss-Seidel method.

If we describe A, b, and x as we did in the previous two sections, the SOR formulation is as follows

$$\xi_i^{GS} = \frac{\sum_{j=1}^{i-1} \xi_j^{k+1} + \sum_{j=i+1}^{N} \xi_j^k}{\xi_i^k} \tag{39}$$

$$\xi_i^{k+1} = \xi_i^k + \omega(\xi_i^{GS} - \xi_i^k) \tag{40}$$

which can be combined to give

$$\xi_i^{k+1} = \omega \left(\frac{\sum_{j=1}^{i-1} \xi_j^{k+1} + \sum_{j=i+1}^{N} \xi_j^k}{\xi_i^k} \right) + (1-\omega)\xi_i^k \tag{41}$$

Again, given the field array in question, u, and coefficient arrays a_W, a_E, a_S, a_N, a_P, b (to represent direct neighbor nodes in a structured mesh) that we would encounter in the CFD techniques discussed in this work for 2 dimensional system of size $m \times n$, the two-stage equations to solve for u at iteration k for the SOR method would look like

$$u_{i,j}^{GS} = \frac{a\_W_{i,j} u_{i-1,j}^{[k+1]} + a\_E_{i,j} u_{i+1,j}^{[k]} + a\_S_{i,j} u_{i,j-1}^{[k+1]} + a\_N_{i,j} u_{i,j+1}^{[k]} + b_{i,j}}{a\_P_{i,j}}$$

$$u_{i,j}^{[k+1]} = u_{i,j}^{[k]} + \omega(u_{i,j}^{GS} - u_{i,j}^{[k]})$$

These equations can be combined to give

$$u_{i,j}^{[k+1]} = \omega \left(\frac{a\_W_{i,j} u_{i-1,j}^{[k+1]} + a\_E_{i,j} u_{i+1,j}^{[k]} + a\_S_{i,j} u_{i,j-1}^{[k+1]} + a\_N_{i,j} u_{i,j+1}^{[k]} + b_{i,j}}{a\_P_{i,j}} \right) + (1-\omega) u_{i,j}^{[k]} \tag{42}$$

for $i = 2 : m+1$ and $j = 2 : n+1$. The pseudo-code (MATLAB like) looks like Algorithm 2.5, where we can see that if we let $\omega = 1$, this code reduces to that of the Gauss-Seidel code in Section 2.7.2. This method has the advantage that it converges very much faster than even the GS method [18].

The choice of the relaxation parameter is a condition of optimization in this method. A theorem of Ostrowski states that if A is s.p.d[2] and $D - \omega L$ (where D and L are results of a

[2] symmetric and positive definite

Algorithm 2.5 Successive over-relaxation algorithm

```
for iter=0:maxiter do
  for i = 2:(m+1) do
    for j = 2:(n+1) do
      u(i,j) = ω((a_W(i,j)u(i − 1, j) + a_E(i,j)u(i + 1, j) + a_S(i,j)u(i, j − 1) +
      a_N(i,j)u(i, j + 1) + b(i,j))/a_P(i,j)) + (1 − ω)u(i,j)
    end for
  end for
  if convergence then
    break
  end if
end for
```

matrix splitting method) is nonsingular[3], then the SOR method converges for all $0 < \omega < 2$ [18]. For the Poisson problem and other similar problems (such as the problems in the CFD techniques in this work), it can be shown that the SOR method converges most rapidly if ω is chosen as [18]

$$\omega_{opt} = \frac{2}{1 + \sin(\pi h)} \approx 2 - 2\pi h. \tag{43}$$

where π is the constant 3.14159... and h is the mesh element width. Figure 2.4 shows computational results for the SOR method (with optimal ω) vs the Jacobi and Gauss-Seidel methods on a very simple 2 dimensional problem. It is clear that the SOR method converges at a faster rate than the Jacobi and GS methods.

2.7.4 Conjugate Gradient Method

The conjugate gradient (CG) method is a pseudo-iterative method for solving systems of linear equations. It was first proposed in 1952 by Hestenes and Stiefel [18]. The term pseudo-iterative is used here because the method has the feature that it always converges to the exact solution of $Au = f$ in a finite number of iterations, and so in this sense it is

[3] a matrix A is nonsingular if there exists a matrix B such that $AB = BA = I$

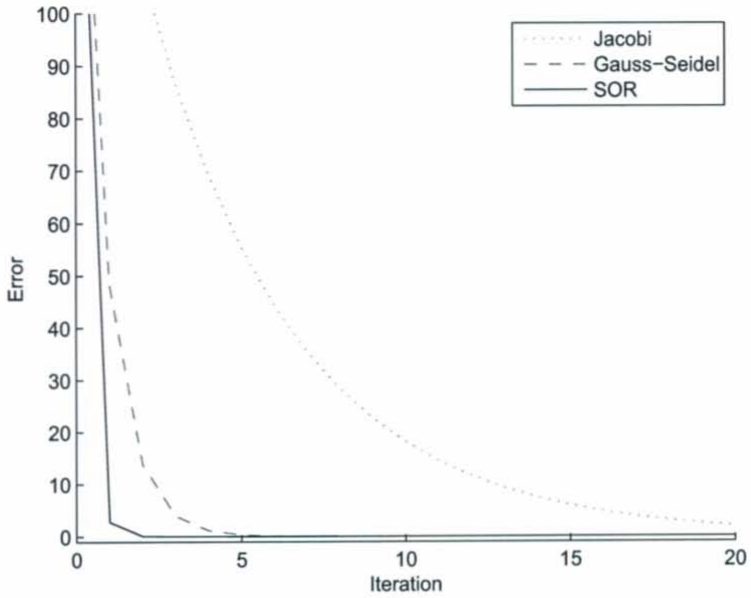

Figure 2.4: Error vs iteration for three linear solver methods

mathematically a direct method like Gaussian elimination in which a finite set of operations produce an exact result.

The CG method is a generalization of the method of steepest decent, which is an iterative method for minimization of a function. This minimization is produced by the extension of the estimate at each iteration in the direction of the local downhill gradient, which is calculated using the residual $r = Au - f$ at the current iteration, until a tolerance is satisfied. The general CG algorithm is documented in Algorithm 2.6 [18].

The convergence rate of this algorithm depends on the condition number of the matrix A. It is common practice to reduce the condition number for this matrix A at each iteration, which will speed up convergence of the algorithm.

Algorithm 2.6 Conjugate Gradient algorithm

Choose initial guess u_0
$r_0 = f - Au_0$
$p_0 = r_0$
for $k = 1, 2, \ldots$ **do**
  $w_{k-1} = Ap_{k-1}$
  $\alpha_{k-1} = (r_{k-1}^T r_{k-1})/(p_{k-1}^T w_{k-1})$
  $u_k = u_{k-1} + \alpha_{k-1} p_{k-1}$
  $r_k = r_{k-1} - \alpha_{k-1} w_{k-1}$
  if $\|r_k\|$ is less than some tolerance **then**
    break
  end if
  $\beta_{k-1} = (r_k^T r_k)/(r_{k-1}^T r_{k-1})$
  $p_k = r_k + \beta_{k-1} p_{k-1}$
end for

The preconditioned conjugate gradient (PCG) has the same basic form as the CG method, except for a step to solve the system $Mz = r$, which is the application of the preconditioner. The basic idea is to choose M for which $M^{-1}A$ is better conditioned than A and that systems that involve M are easier to solve than those that involve A. The PCG algorithm is described by Algorithm 2.7 [18].

Algorithm 2.7 Preconditioned Conjugate Gradient algorithm

$r_0 = f - Au_0$
Solve $Mz_0 = r_0$ for z_0
$p_0 = z_0$
for $k = 1, 2, \ldots$ **do**
  $w_{k-1} = Ap_{k-1}$
  $\alpha_{k-1} = (r_{k-1}^T r_{k-1})/(p_{k-1}^T w_{k-1})$
  $u_k = u_{k-1} + \alpha_{k-1} p_{k-1}$
  $r_k = r_{k-1} - \alpha_{k-1} w_{k-1}$
  if $\|r_k\|$ is less than some tolerance **then**
    break
  end if
  Solve $Mz_k = r_k$ for z_k
  $\beta_{k-1} = (z_k^T r_k)/(r_{k-1}^T r_{k-1})$
  $p_k = z_k + \beta_{k-1} p_{k-1}$
end for

This algorithm converges in much less iterations than the previously mentioned iterative methods (Jacobi, Gauss-Seidel, and successive over-relaxation). The fact that it converges in less iterations does not mean it is faster. Because this method requires many operations per iteration, its run time may vary depending on how efficient these operations are on the hardware used. This will be discussed in more detail in chapter 5.

3 GPU Architecture

Graphics Processing Units (GPUs) have a many-core parallel architecture. They consist of a set of stream processors that execute programs (also called kernels) in parallel. GPUs were originally designed for graphics processing, so the stream processors are designed for small and fast operations (per stream processor) such as filtering a texture.

This chapter will discuss how GPUs compare to CPUs. We will also discuss how one can program GPUs (the programming model), and then the hardware model and architecture details of GPUs. Finally, we will discuss several performance measures for evaluating applications on GPUs.

3.1 GPU vs CPU

GPUs have evolved over the years to become, in some cases, a more efficient (both computationally and in terms of cost) than the traditional CPU. Figure 3.1 illustrates the evolution of the GPU vs the CPU over past years with respect to the number of floating point operations per second (flops/s). From this figure we can see that GPUs greatly surpass CPUs in this respect, even as CPUs evolve into having multiple cores. Today, average GPU (even the GPUs in most workstations) are more efficient in this respect than high-end CPUs.

As previously mentioned, GPUs are a set of stream processors. This concept is the main advantage with respect to GPUs (for certain applications). The main difference in CPUs and GPUs is how they weigh their priorities in design with respect to control units and cache vs number of cores. Figure 3.2 illustrates this concept. We can see in this figure that the CPU (on the left) has a lot more of its internal structure dedicated to control and cache than to the number of cores available, while the GPU (on the right) puts much more emphasis on the number of cores available and less on control and cache.

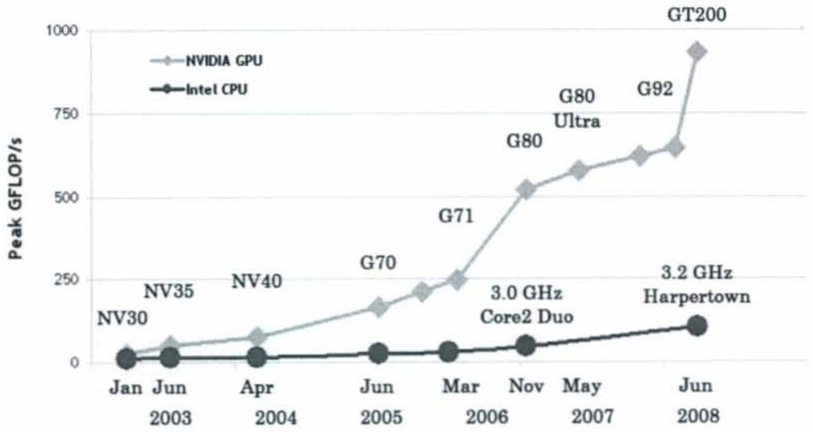

Figure 3.1: GPU vs CPU Performance Evolution [22]

3.2 Programming Model

The programming model used for the nVidia GPUs (called CUDA) is built around a SIMT (single-instruction multiple-thread) architecture concept. SIMT is not the same as the traditional SIMD (single instruction multiple data) concept in that SIMD applies the same instruction to multiple pieces of data simultaneously, while SIMT executes the same thread (code block) simultaneously with a single instruction. A typical CUDA execution on a GPU consists of a mapping of the threads (or kernels) to a two-level grid of a user-specified size. The threads are mapped as a set of threads, grouped into blocks. The number of blocks in the grid is called the grid size and the number of threads per block is called the block size. Figure 3.3 illustrates this block/thread mapping. Once the threads are mapped they are then enumerated and distributed to the available cores on the device. Scheduling of these kernels, as threads of the grid are terminated and new ones are executed, is performed automatically on the GPU itself, Section 3.3 describes how they are distributes and scheduled amongst the cores on the device.

The language that nVidia provides to develop programs for their GPUs is called CUDA.

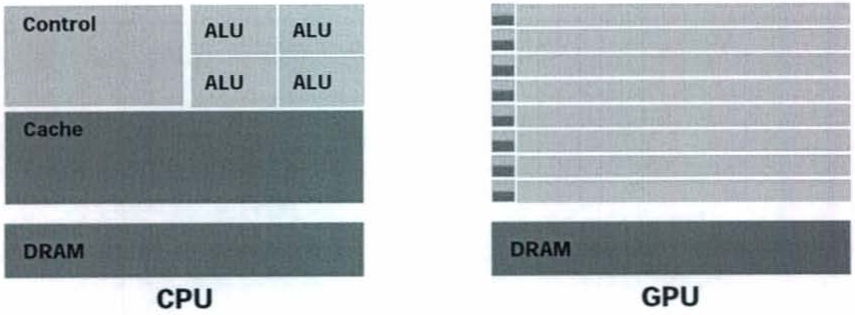

Figure 3.2: GPU vs CPU Architecture [22]

It is a C-like language, where the developer is expected to create individual functions that, as the kernels discussed previously, are executed by N times N number of threads. A sample kernel that does simple vector addition and the associated call in C is (from [22])

```
\
// kernel function
__global__ void VecAdd(float* A, float* B, float* C)
{
int i = threadIdx.x;
C[i] = A[i] + B[i];
}
// main program, run on host
int main()
{
// Kernel invocation
VecAdd<<<1, N>>>(A, B, C);
}
```

The *__global__* specifier identifies a kernel function, there are other specifiers that can

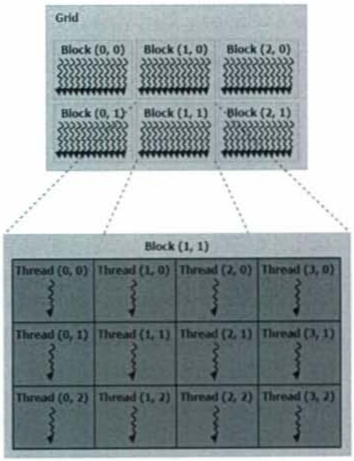

Figure 3.3: GPU Programming Model [22]

be used but are out of the scope of this thesis, see [22]. Since this function is executed in parallel on N different threads, we have each one doing a simple addition on its respective location in the vectors. The kernel function is given the thread number that it is executed on by the *threadIdx* variable.

The kernel is executed on the host with the call $VecAdd <<< 1, N >>> (A, B, C)$ tells the GPU to execute the *VecAdd* kernel with 1 block in the grid and N threads per block. This block/thread number can change depending on the application, for example we could use the code

```
\
// kernel function
__global__ void VecAdd(float* A, float* B, float* C)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
```

```
}
// main program, run on host
int main()
{
// Kernel invocation
VecAdd<<<16, N/16>>>(A, B, C);
}
```

We will see in the next section that this code will be more efficient due to the scheduling method in use on the GPU.

3.3 Hardware Model

The GPUs in this work, nVidia GPUs, are designed with an array of multi-threaded Streaming Multiprocessors (SMs). Each of these processors contain a set of Scalar Processor (SP) cores (currently all nVidia devices contain eight cores per SM), a multithreaded instruction unit, and a shared memory unit for that multiprocessor. Outside of the array of SMs there is a memory space, called device memory, that is in use by all components of the GPU. Device memory is the slowest on-card memory, while shared memory (per SM) is the fastest on-card, next to the registers of course but not far behind [22].

Figure 3.4 illustrates the components of the SM. As we can see it contains the SPs, the instruction unit, and shared memory per SM, but it also contains a "constant memory" and a "texture memory". Constant memory space is a read-only region of device memory and texture memory space is again a read-only region of device memory, each used for their specific purposes (and optimizations) in many different graphics processing applications. Both memory spaces are preferred over the use of device memory directly, in that they have faster read times. Shared memory is still preferred over both constant and texture memory

because they are spaces within device memory (which is the slowest on-card memory) and therefore shared memory has much faster access time [22].

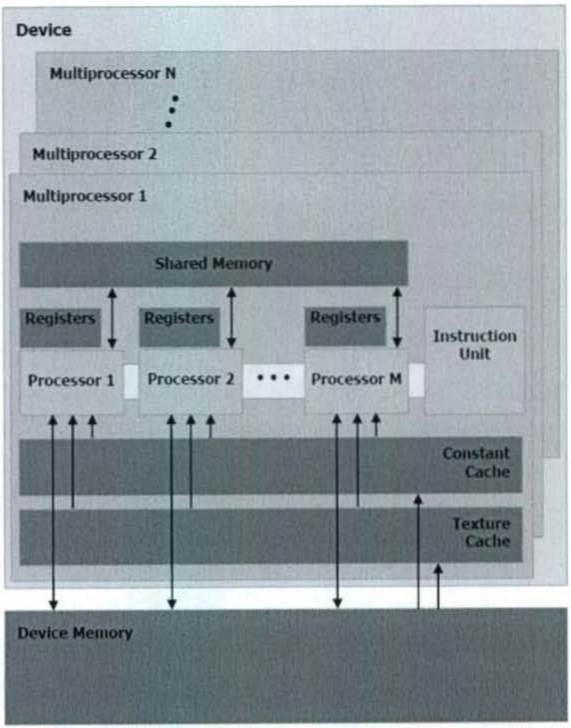

Figure 3.4: GPU Streaming Multiprocessor Components [22]

An important factor in the design of these streaming multiprocessors on the GPUs is the scalability without having to alter the programming model. Figure 3.5 illustrates this factor. We can see that each block is passed off to a single SM, and so as the number of SMs increase, the scheduler on the GPU can just divide any extra blocks that are waiting in the execution queue amongst the additional multiprocessors with very little scheduling overhead.

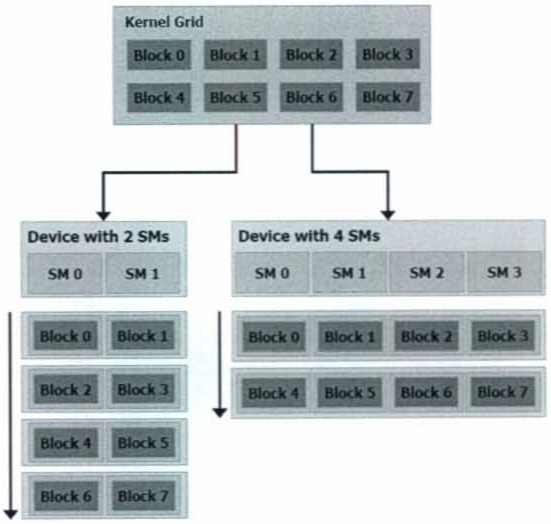

Figure 3.5: Scalability of GPU Streaming Multiprocessors [22]

For more information about the nVidia GPU architecture(s) and programming model (CUDA) refer to [22].

3.4 Performance Measures

This section will describe several measurements of performance for programs designed for the GPU. These measures will be used to evaluate the results in this work.

3.4.1 Run Time and Speedup

Run time is defined as the amount of time required for a program to execute to successful completion. For example, a solution method for solving systems of linear equations would execute to completion when it has converged to a final result. A CFD solution method (SIMPLE or PISO) would be complete when it executed a single iteration without failure or

converged to a final steady-state result or performed a set amount of time steps, depending on the post execution analysis to be performed.

The speedup of an algorithm (for the purpose defined in this work) is the comparison of run time of execution of that algorithm on the GPU against run time of the same algorithm on the CPU. This measurement is important because it displays the increase in efficiency (with respect to run time performance) of one technique over another. In the case of this work, run time of CFD simulations on GPUs over traditionally CFD simulations on CPUs is observed.

3.4.2 Memory Requirements

Memory requirements of a system is a simple measure of the amount of memory required for a program to execute successfully on a system. The memory usage may fluctuate throughout execution, so we can measure minimum, maximum, and/or average memory usage on a system. This measure is important since we are developing a technique using a different implementation (such as in this work). If the new technique requires too much memory it may be impractical, even it has other significant performance gains.

3.4.3 CUDA Occupancy

The CUDA Occupancy is a measure of a kernel invocation that describes how well the kernels make use of the multiprocessor resources located on GPUs, such as allocated registers and shared memory. This concept is best described by nVidia [22]:

> The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. Each multiprocessor on the device has a set of N registers available for use by CUDA thread programs. These registers are a shared resource that are allocated among the thread blocks executing on a multiprocessor. The CUDA compiler attempts to minimize register usage to maximize the number of thread blocks that can be

> active in the machine simultaneously. If a program tries to launch a kernel for which the registers used per thread times the thread block size is greater than N, the launch will fail.

nVidia provides a very simple tool (in the form of an Excel spreadsheet) for calculating the occupancy measure for various kernels. This tool is called the CUDA GPU Occupancy Calculator and to use it one has to enter the number of threads per block that are being used in the kernel invocation (determined in the code prior to kernel invocation), the number of registers per thread in use (retrieved by looking at the generated .cubin file after compilation), and the amount of shared memory per block (also retrieved from the compiler-generated .cubin file). This spreadsheet will not only calculate the total occupancy (as a percentage), but individual occupancies (per resource) and describe any limiting factors in the kernel considered. This is a very useful tool for optimizing any kernel for invocation on a nVidia GPU.

4 Related Work on GPUs

This section is a discussion of related work to what is documented in this thesis. We will first discuss general CFD on GPUs and other similar work that has been done in this field, such as other solution methods to fluid systems. We will then discuss work relating to solving systems of linear equations on GPUs.

4.1 Computational Fluid Dynamics on GPUs

In scientific literature fluid flow problems on GPUs follow two separate paths, due to the nature of GPUs, which were originally designed for graphics processing. First, as in this work, they are developed to solve scientific real-world fluid flow problems. Second, they are developed to simply look realistic, with applications in video games, visual effects, or for non-scientific simulation systems, such as in [23].

This work focuses on the scientific application of fluid flow simulations. In particular it focuses on the development of a general purpose fluid flow solver and develops the methods to be applicable to any general purpose fluid flow solver, including scaling of dimension, size, and accuracy.

In November 2006 nVidia released a new programming model for their GPUs called CUDA (Compute Unified Device Architecture) that allowed general purpose computations (without knowledge of the graphics pipeline) on their GPUs [34], and since then many of the major GPU manufacturers have followed, with the release of their own general purpose programming models for their cards. All GPU for CFD work before that time took advantage of the shader programming model, that was developed for GPUs to perform graphics specific problems such as depth buffer and color blending operations, such as [19], [20], and [31]. This shader programming model is graphics API dependent (therefore was different for different graphics APIs), with languages such as Cg, OpenGL Shader Language

(GLSL), and High Level Shading Language (HLSL). The shader language programming model is now considered to be out-dated for general purpose programs and less efficient for general purpose applications on GPUs, since the problem would have to be mapped to a graphics problem in order to be solved efficiently [34].

Tong et al. in a 2007 paper [31] outline their study and design of a fluid solver on GPUs using this graphics pipeline method, making use of OpenGLs shader language (GLSL). In their study, they use a fragment program that fetches packed texture data[4] directly and processes the data in parallel by distributing among them several pipelines. They use a Jacobi method for solving systems of linear equations, applying similar graphics pipeline techniques.

It should be noted that at the time of conception of the work presented in this thesis, the majority of work on this subject used the out-dated shader language methods since the new CUDA programming model was a very new development at the time, and that the main goal of this work is not to replicate other modern CFD on GPU work (that use the CUDA programming model) such as that of [34] and [27], but to develop a general purpose solver that can be used for a variety of fluid problems and will run efficiently on modern GPUs.

The technique developed in [34] is developed for non-physical visualizations only using the Stable Fluids method, which is a method that discards components of the flow that have little visual effects. The technique developed in [23] makes use of the Smoothed Particle Hydrodynamics (SPH) method. The SPH method is similar to the approach of this work in that it is a general purpose fluid solver method, but it uses a very different technique called a Lagrangian approach to simulating the fluid flow. The Lagrangian approach simulates fluid by simulating particles or particle packets, as opposed to the method in this work which simulates the fluid continuum through a discretization of that continuum. This type of method was not discussed in previous sections of this work since it is a different method

[4]texture packing is the method where multiple values are stored into a single RGBA texel vector for memory usage and read/write efficiency

entirely compared to the methods used in this work.

4.2 Solving Systems of Linear Equations on GPUs

Recently there has been a lot of scientific literature on the subject of solving systems on linear equations using GPUs, such as in [33], [9], and [11]. All of this work has been focused on solving systems of linear equations using the Gauss-Seidel method (see Section 2.7.2 for a discussion of this method). The work outlined in this thesis also depends on other methods on GPUs such as the conjugate gradient method. The reason for the lack of literature in this subject area is not that this is the first time it has been conceived but that an implementation of the conjugate gradient method on GPUs involved simple matrix algebra operations that are well developed for GPUs (see Section 5.4 for references and a description of its implementation).

The work developed in [9] uses the Gauss-Seidel technique for solving systems of linear equations on GPUs. The authors of this work develop three different strategies for solving systems of linear equations on GPUs: row-based, column-based, and block-based. Each strategy is named for its parallelization scheme. The row-based strategy involves solving each row separately (in parallel), the column-based strategy involves solving each column in parallel and the block-based strategy involves solving custom blocks of the system in parallel. This work found up to a 10x speedup was achievable. Because the work described in this reference was developed for dense systems, as opposed to very sparse systems that are in use in this thesis, it is less relevant than others methods (with respect to practicability for the work developed in this thesis) but important parallel concepts for solving systems of linear equations can be derived from this work.

Work developed in [33] is very similar to the Gauss-Seidel techniques developed in this thesis. This work involves the development of a system of linear equations solver using a Red-Black (or checkerboard) parallelization technique where all nodes in a system are

colored so that any given node has no neighbors with the same color as that of itself. The work developed in this reference found that a 57x times speedup was achieved with it's developed method and that this technique leads to several implicit optimizations that the GPU architecture, most notably is that memory read patterns are optimized for the GPU architecture (memory reads are coalesced[5]).

Finally, work such as that of [27] and [34] involve solving fluid systems on GPUs (as discussed in the previous section). The fact that the linear solvers in both of these works use point-iterative methods (such as Jacobi and Gauss-Seidel) is a indicator that these methods are possibly superior for solving systems of linear equations on GPUs. This fact is discussed and tested in Section 5.4.

[5]Coalesced memory simply means that the pattern of memory access is uniform across the threads

5 Methods

To describe the methods used for the design and implementation of the methods discussed in chapter 2 the chosen mesh type and discretization technique used for this work must first be defined. This chapter will first discuss the discretization technique and its advantage, then discuss the mesh type used in the implementation of the methods and give reasons for this choice. Finally Section 5.3 and following will describe the design of the SIMPLE and PISO methods, and all operations required in these methods for GPUs.

5.1 Discretization Technique

The discretization technique used for this work is the finite-volume method (FVM). See Section 2.2.3 for a description of this technique. The justification for using this discretization technique is that it is easy to implement and, since this work is meant for general purpose CFD simulations, this technique provides a robust and stable backbone to this work.

Details of the 2 dimensional finite-volume discretization on a uniform (structured) mesh can be found in Appendix A.

5.2 Mesh Type

The mesh type used here is the structured mesh. See Section 2.3 for a detailed description of this type of mesh. We use this mesh type because it is appropriate for the architecture of the GPU and it's the programming model. GPUs were originally designed for graphics processing, such as image filtering or any operations that involve processing of a large number of pixels. For this reason, advantages can be gained by the use of an algorithm designed with this "pixel processing" idea in mind. Since the structured mesh type is a uniform set of nodes on the CFD system, this maps very well to the "pixel processing"

approach so that we can map a single GPU thread to a CFD mesh node, just as GPU threads were originally designed to map to a single graphics pixel.

The advantages of this compared to the unstructured mesh type are, first that we save memory since we do not have to store all the unstructured mesh nodes and edges in memory (it is common to have much less memory on a GPU than on a host), second that memory access to CFD data such as flow field values (velocity and pressure) or coefficient arrays is inherently coalesced. Coalesced memory simply means that the pattern of memory access is uniform across the threads.

5.3 SIMPLE/PISO -GPU Methods

Generally, both the SIMPLE and the PISO methods involve the same operations (as described in Section 2):

- Construct coefficient matrices for systems of linear equations
- Solve systems of linear equations
- Apply corrections to flow fields
- Check convergence (residual sum)

The SIMPLE method requires 3 operations of both construction of the coefficient matrices and solving of system of linear equations per iteration. It then only requires 1 application of corrections to the flow fields and 1 convergence step per iteration. The PISO method requires, in addition to all of the operations above, another one operation for both coefficient construction and solving a system of linear equations, and again another for application of the new corrections to the flow fields.

Mapping of the numerical CFD nodes to GPU threads, or stream processors, is a one-to-one mapping (See Fig. 5.1), that is one thread for each node in the discretized CFD

system. The justification for this one-to-one mapping lies in the nature of the original design of GPUs, that is, they were originally designed for graphics processing i.e. pixel processing. This means that the GPU architecture is best suited for "small"[6] operations on many threads/nodes/pixels (see Section 5.2).

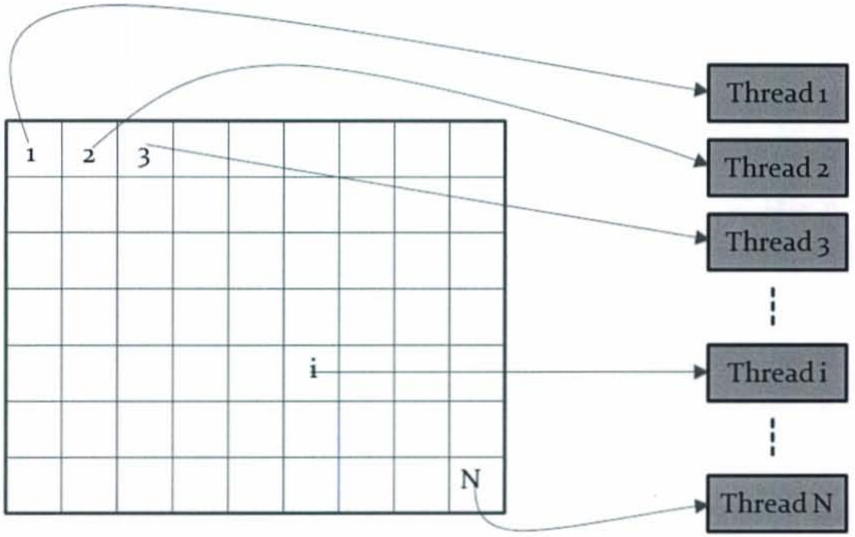

Figure 5.1: CFD Node to GPU Thread Mapping

The following sections will describe the methods used for solving systems of linear equations on the GPU in the SIMPLE and the PISO methods and discuss why they were the best choice since this is largely the most important operation in the context of efficiency. We will then discuss the remainder of the operations in the order presented above.

5.4 Solving Systems of linear equations

The most important part of our implementation is the solution method used for solving systems of linear equations. Normally (on a CPU), a preconditioned conjugate gradient

[6]simple updates, etc.. a kernel invocation has a 5 second execution limit on some systems

(CG) method would the best choice for these linear solvers, but the CG method involves a matrix-vector multiplication and a vector-vector summation, which, compared to a linear solution method such as the Gauss-Seidel (GS) or successive over-relaxation (SOR) methods (with a single update per iteration), is much more expensive computationally on a GPU. The reason this is so much more expensive on a GPU is that GPUs do not handle random memory access very well, that is they are built and optimized for a uniform memory access strategy (coalesced memory access). For this reason, the SOR method is used for all linear solvers (this will be justified in this section). Although the CG method converges in less iterations, the cost of run time out-weighs the cost of convergence time. The SOR method is implemented on the GPU through a two step iterative method, also known as a red-black method, at each iteration all odd nodes are first updated in parallel, and then all even nodes are updated in parallel.

5.4.1 GPU Gauss-Seidel/SOR Method

The GPU implementation of the Gauss-Seidel or successive over-relaxation method (both methods are very similar and the terms will be used interchangeably in the rest of this thesis) is a type of domain decomposition of the numerical fluid system. The implementation is not a straightforward domain decomposition, however, it involves making two passes over the system per iteration, although each node is only updated once per iteration. See Sections 2.7.2 and 2.7.3 for a description of the sequential Gauss-Seidel and SOR method from which this method is derived.

Initially the method "colors" each node in the system two alternating colors so that no node has neighboring nodes of the same color, such as in Figure 5.2. This coloring of nodes (two colors for a uniform two dimensional mesh) is why this parallel technique for the GS is also known as the red-black or the checkerboard method. Once each node is assigned a virtual color, we continue as we would in the sequential version of the method,

Figure 5.2: Red Black Nodes

except for one change: at each iteration there are two passes over the nodes, the first pass updates one color nodes (the red nodes) and the second pass updates the second color nodes (black nodes). Then we iterate as normal until convergence is reached. So far the parallel algorithm may look something like (in sequential form for now) Algorithm 5.1.

Algorithm 5.1 Parallel (Red-Black) Gauss-Seidel algorithm

```
1: for iter=0:maxiter do
2:    for i = all RED nodes do
3:       update u(i)
4:    end for
5:    for i = all BLACK nodes do
6:       update u(i)
7:    end for
8:    if convergence then
9:       break
10:   end if
11: end for
```

The advantage of this algorithm, in a parallel sense, is that all RED nodes can be updated simultaneously and all BLACK nodes can be updated simultaneously since from the update equation (42) we know that only neighboring nodes are read each node update. Since

neighboring nodes will definitely not be updated at the same time (because of the different coloring), this allows us to perform updates on all nodes of the same color simultaneously.

Now that we have the general idea of the algorithm, we can move on to a more custom implementation for GPUs. First of all, we can map each node to a single thread, as described previously in Figure 5.1. With this implementation we now have, for a 2 dimensional system, 5 reads and 1 write per node update. The write is to the node that is mapped to the thread (the local node) and the reads are from the local node and its direct neighbors, as indicated by the white dots in Figure 5.3.

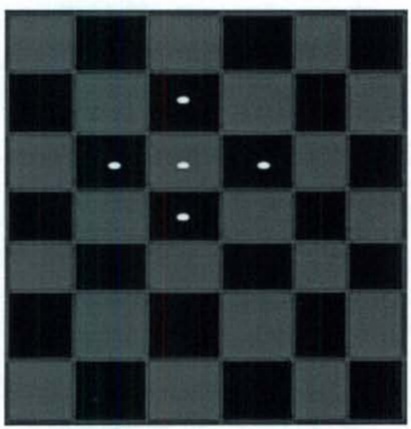

Figure 5.3: Red Black nodes with local and neighboring nodes

As of the algorithm developed so far, we use global GPU memory for the 5 reads per node update, which requires many duplicate reads per update since all neighboring nodes of a single local node are being read at least one more time and up to 4 more times per half iteration (per single color update pass). If we recall Section 3, the GPU programming model uses a set of blocks, where each block contains a set of threads, and each block has access to more efficient memory (called shared memory in the section above). If we make use of this shared memory per block we can remove nearly all of these duplicate reads by

loading all nodes in a block into shared memory before we do the update. The set of nodes required for a block to update all of its associate threads (from the running example) are indicated by white lines in Figure 5.4. If we load all of these nodes into shared memory, including the ghost layer which is the layer of non local nodes (nodes that do not need to be updated by this current block) that surround the edges of the block, we can reduce the number of duplicate reads by a factor of almost 4 along with the memory access time for these reads since shared memory is much more efficient.

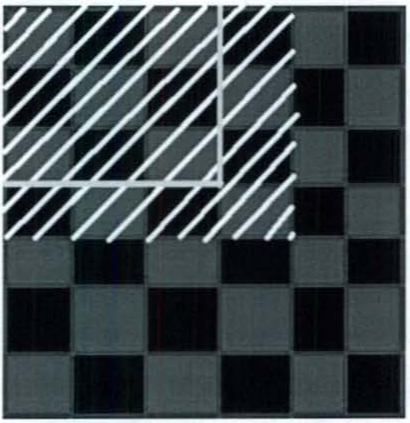

Figure 5.4: Red Black nodes that must be read per block (block is highlighted in yellow)

The pseudocode (with some real CUDA commands) CUDA kernel (see Section 3) that is executed for each thread would then look something like Algorithm 5.2. Lines 2 to 15 load all local nodes and the ghost layer into shared memory. Line 17 uses the *__syncthreads()* function, which causes all threads in the block to wait at this location in the code until all of them have reached that point, this way all required data is loaded into shared memory before we start to do any updates (reads and writes) using this data.

Algorithm 5.2 GPU Guass-Seidel algorithm

```
1: // Load local node into shared memory
2: u_shared[s_i][s_j] = u[ij];
3: // check if on edge node, if yes then load ghost layer
4: if threadIdx.x == 0 then
5:    u_shared[s_i-1][s_j] = u[i-1][j];
6: end if
7: if threadIdx.x == BLOCK_SIZE_X-1 then
8:    u_shared[s_i+1][s_j] = u[i+1][j];
9: end if
10: if threadIdx.y == 0 then
11:    u_shared[s_i][s_j-1] = u[i][j-1];
12: end if
13: if threadIdx.y == BLOCK_SIZE_Y-1 then
14:    u_shared[s_i][s_j+1] = u[i][j+1];
15: end if
16: // wait for all threads in block to finish loading shared memory
17: __syncthreads();
18: for i,j = all RED or BLACK nodes only do
19:    update u[i][j]
20: end for
21: if convergence then
22:    break
23: end if
```

5.4.2 GPU Conjugate Gradient Method

As discussed in Section 2.7.4, the conjugate gradient (CG) method is a pseudo-iterative method for solving systems of linear equations. The conjugate gradient method is actually the best choice as a linear solver for the SIMPLE and especially the PISO method since the linear solver is required to act as a real solver and not simply a smoother (see Section 2). The reason it is usually the best choice is simply that it converges to the actual solution in less iterations than other linear solver options discussed. As it will be discussed in Section 7.1, where comparison results are showing in Section 6.1, the conjugate gradient may be the most efficient in terms of number of iterations required to converge to a solution, but this does not necessarily mean it is the most efficient performance wise when compared to other linear solvers on the GPU.

Much of the literature on the subject of parallelizing the CG method states that it is a very good candidate for parallelization because it requires only two types of operations. They are matrix-vector multiplication (e.g. line 5 of Algorithm 2.6) and vector-vector summation (e.g. line 8 of Algorithm 2.6) per iteration, which can both be parallelized very efficiently on traditional parallel systems[7] for large sparse matrices. Although both operations can be parallelized for the GPU, they are not as efficient (again, due to their optimization for uniform memory access), and therefore make the CG method less efficient when compared to other methods for solving systems of linear equations such as the GS/SOR method per iteration.

The loss in efficiency per iteration comes from these two operations, matrix-vector multiplication and vector-vector addition. The GPU parallelization of the matrix-vector multiplication is a simple and well established method [3] to map each thread in the GPU programming model to a single matrix row. Each thread then simply performs the dot product of its associated row to the vector in question. Since the matrix is sparse, in a two

[7] distributed and shared memory systems

dimensional case in this problem where we have the matrix as the coefficient matrix which contains only five elements and therefore each thread requires only 5 multiplications and 5 additions. The CUDA kernel for matrix vector multiplication is

```
extern "C" __global__ void mat_vec_mult(
float* x, float* a_W, float* a_E,
float* a_S, float* a_N, float* a_P,
float* b, int num_rows, float* y_out, unsigned int Nx)
{
int i = blockIdx.x*blockDim.x + threadIdx.x;
int j = blockIdx.y*blockDim.y + threadIdx.y;
int row = i + j * Nx;

if(row < num_rows)
{
float dot = 0.0f;

dot += a_P[row] * x[row];
if(row >= 1) dot += -a_W[row] * x[row-1];
if(row < num_rows-1) dot += -a_E[row] * x[row+1];
if(row >= Nx) dot += -a_S[row] * x[row-Nx];
if(row < num_rows-Nx) dot += -a_N[row] * x[row+Nx];

y_out[row] = dot;
}
}
```

The GPU parallelization of the vector-vector summation is even simpler than the matrix-

vector multiplication. It is a simple thread to vector element mapping where each thread would perform the addition of its corresponding elements. The vector-vector summation kernel looks like

```
extern "C" __global__ void vec_vec_add(
float* x1, float* x2, float* x_out, unsigned int Nx)
{
int i = blockIdx.x*blockDim.x + threadIdx.x;
x_out[i] = x1[i] + x2[i];
}
```

In order to be most efficient, there are different methods for both vector-vector addition and subtraction. Besides these two matrix and vector operations there are even simpler operations that on a traditional distributed system one would not ordinarily parallelize (except for extremely large vectors). One such operation, on line 8 of Algorithm 2.6, is $r_k = r_{k-1} - \alpha_{k-1} w_{k-1}$, which is a vector-vector addition as already covered but in addition there is also a scalar-vector addition. Scalar-vector additions can be performed more efficiently on the GPU especially when the vector is already loaded into GPU memory (as it is for the GPU CG method) by simply mapping each vector element to a thread and having each thread perform a simple multiplication of the scalar and its associated vector element.

The GPU operations defined above can then be substituted for their respective sequential operations in Algorithm 2.6 or 2.7 to produce the GPU CG method or the GPU preconditioned CG method.

5.4.3 Summary

A performance comparison of these two GPU methods for solving systems of linear equations can be found in the results Section 6.1 and a discussion of these results in the discussion Section 7.1.

5.5 Coefficient Calculation

The first operation to be parallelized on the GPU architecture is the construction of the coefficient matrix for the system of linear equations. This operation can be performed with a single GPU program, or kernel, for each type of equation, e.g velocity, pressure correction, second pressure correction. Since the matrix is sparse, involving only coefficients for direct neighbor nodes, memory on the device need only be allocated for these neighboring coefficients, not for the full matrix. Access to field values at the local node and at direct neighbors is also required, and since global memory access on GPUs is their largest bottleneck, shared memory is used to store nodes per block in order to reduce the number of duplicate memory accesses.

5.6 Corrections

The application of the corrections to the flow fields is simply a kernel that applies the correction to each node. Since these corrections require only access to local field values at each node (as opposed to field values at neighbor nodes), a simple update per kernel is most efficient.

5.7 Convergence

Convergence of both the SIMPLE and PISO methods can be determined in many different ways, depending on the application of the method. The most popular methods are a check of the velocity residual sum against a tolerance, or a check of the norm of pressure correction against a tolerance.

For both residual sum and norm calculations on the GPU we must perform a sum. This may seem simple but to efficiently do this on a GPU a little work is required. To do an efficient sum of a large vector on the GPU we do a parallel sum reduction. The method

used in this work is defined in [25], and uses a tree based approach within each thread block, as illustrated in Figure 5.5. This algorithm works by assigning a uniform and contiguous subset of the vector to each thread block, each thread block then performs the sum of its associated subset and stores the result in the first memory location of its subset (denoted by the child node in the figure). It recursively does this until only one value is left (moved down the tree in the figure), which is the sum of the original vector. The time complexity of this technique is $O(N/\#Blocks + logN)$, vs $O(N)$ if we were to use a simple loop for summation.

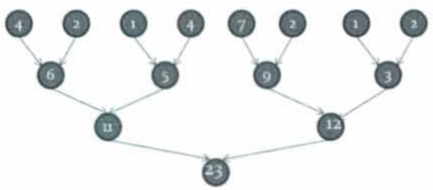

Figure 5.5: Parallel sum reduction using tree based approach within each thread block

Advantages of this technique is not only that the majority of the computations are performed on the GPU but that the vector itself never needs to leave the GPU (which is preferred since all other calculations for the SIMPLE and PISO methods are on the GPU). Further, retrieve the sum only one value needs to be copied from GPU memory to host memory. As we have noted in Section 3, copying from device to host memory is one of the most serious bottlenecks in any GPU implementation.

6 Results

The results obtained in this work can be qualified by performance tests and compared to the sequential version of the methods. Memory requirements of the developed GPU code, and finally a measure known as CUDA occupancy of the GPU kernels can also be measured. CUDA occupancy is a measure of the multiprocessor occupancy of a GPU by a given CUDA kernel (see Section 3.4.3 for a detailed description).

Besides these direct results, there are also indirect results that affected decisions on how the work was developed and what methods were used in the development. Some of these results related to methods for solving systems of linear equations on the GPU.

This section will first describe results obtained in comparing different methods for solving systems of linear equations on the GPU. We will then describe the results of performance measures of both CFD methods that have been developed in this work. Next we describe the memory requirements of these CFD methods, and finally we report on the CUDA occupancy of the GPU methods developed here.

A discussion of each topic in this section can be found in each respective section of Chapter 7.

6.1 Solving systems of linear equations

Figures 6.1 and 6.2 show a comparison of the run time per iteration of the GPU implementation of the SOR and conjugate gradient methods as the number of nodes in the system increases. The purpose of these results is to justify the choice of either method for use in the GPU CFD methods. Both figures represent the same comparison tests, the two methods are displayed together in 6.1 to highlight the very large difference in run times, and separately in Figure 6.2 to highlight the time scales.

These results were obtained on a system with a nVidia GeForce 9800 GT GPU, AMD

Athlon 3200 CPU at 2.0GHz, and 2 GB of memory. This GPU is an average GPU for a home desktop computer. The SOR method required about 55 iterations in order to reach convergence, while the CG method required about 25 iterations for convergence[8].

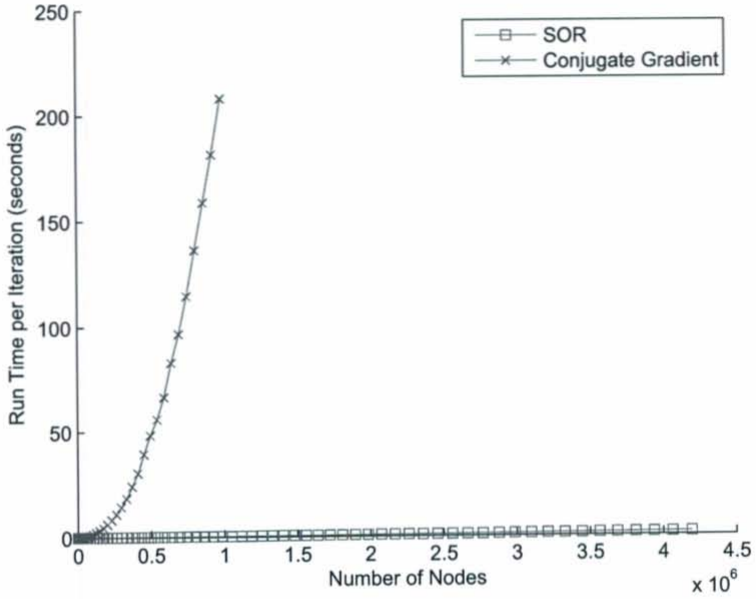

Figure 6.1: A comparison of run times per iteration for GPU implementations of SOR and Conjugate Gradient methods

[8]The number of iterations to reach convergence varied by a small amount as the size of the system changed

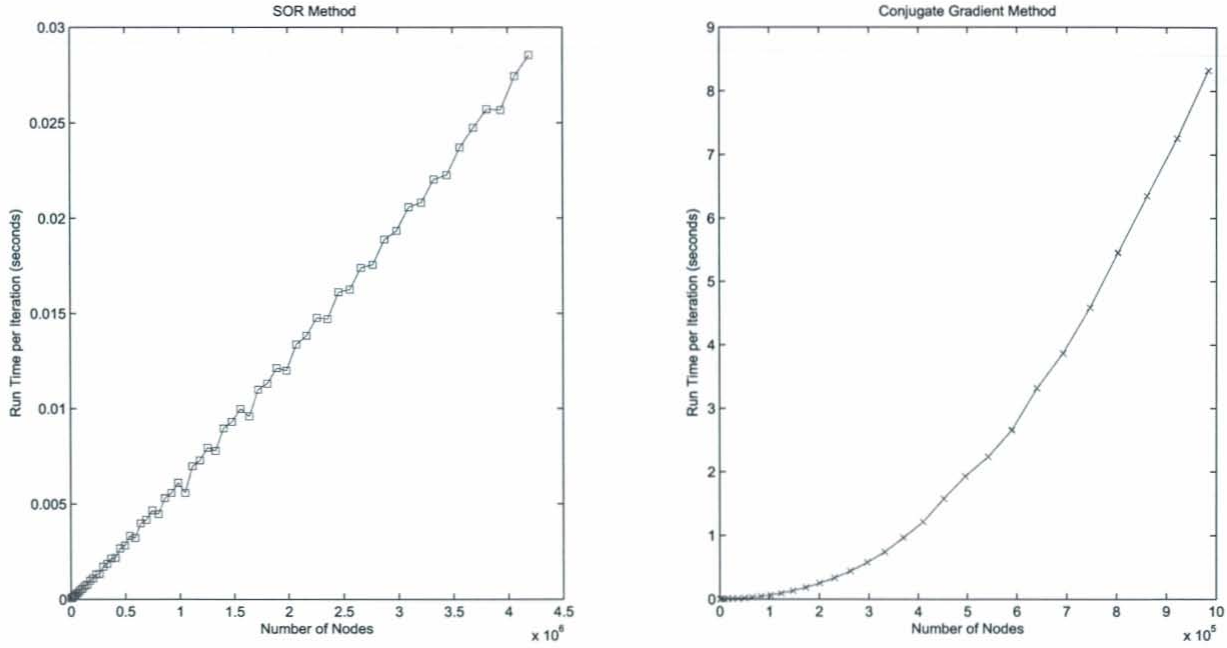

Figure 6.2: Run times per iteration for GPU implementations of SOR and Conjugate Gradient methods (separate plots)

6.2 Performance

The performance measures used here are run times and speedups, as compared to the sequential version of the algorithms running on CPUs only. These measures are evaluated for an increasing number of nodes in the system, illustrating the scalability of our approach.

For the purpose of comparative results on GPUs vs CPUs four machines were used, giving 3 CPUs and 3 GPUs to compare algorithm performances. This section describes first the machines used to test the methods, then the performances of both the SIMPLE and PISO methods.

6.2.1 Test Machines

For test purposes, four machines were used, each one containing a different GPU. Table 6.1 shows CPU and GPU information (CPU, GPU, and Memory[9]) for each test machine used. It must be noted that in any CPU test only one core per processor was used to provide a true comparison with the sequential algorithm.

Table 6.1: Test Machines

GPU	CPU	Memory
nVidia GeForce 8200	AMD Athlon 4850e at 2.5GHz	2 GB
nVidia GeForce 9800 GT	AMD Athlon 3200 at 2.0GHz	2 GB
nVidia GeForce 9800 GTX+	AMD Athlon 4850e at 2.5GHz	2 GB
nVidia Tesla C1060	Intel Xeon X5550 at 2.67GHz	3 GB

Table 6.2 shows features of the GPUs that each machine contained, along with some technical specifications of the GPUs. The GPUs range from a very low end nVidia GeForce 8200, which is an onboard[10] card that shares its global memory with the host machine and therefore has slow memory access, to a moderate nVidia GeForce 9800GT, and finally to a high-end nVidia Tesla C1060. Tesla cards are designed for use in scientific computing

[9]Host memory, not including GPU memory

[10]The video card is part of the motherboard and shares its memory with the host memory

applications, while the GeForce models are designed for graphics processing, e.g. video games.

Table 6.2: Test GPUs

GPU	Cores	Memory
nVidia GeForce 8200	8	256 MB (Shared)
nVidia GeForce 9800 GT	112	1024 MB
nVidia GeForce 9800 GTX+	128	512 MB
nVidia Tesla C1060	240	4096 MB

6.2.2 The SIMPLE Method

Figure 6.3 shows the run times as we increase the number of nodes in the numerical fluid system using the SIMPLE method, while Figure 6.4 represents the speedups of the GPU vs the CPU in each respective test machine.

Table 6.3 shows a summary of Figure 6.4 using peak speedup between the GPU and its respective CPU in the test machine. From these results we can see that the nVidia GeForce 9800GT/AMD Athlon 3200 combination provides the best speedup and scalability with a peak speedup of about 360×, which decreases the least as the number of nodes in the system is increased (see Figure 6.4). This behavior is not expected for this GPU since it is not the higher-end card in the test set, it is actually in the middle of the test set with respect to processing power.

Table 6.3: Speedup Results Per Machine for SIMPLE Method

GPU	CPU	Peak Speedup
nVidia GeForce 8200	AMD Athlon 4850e at 2.5GHz	20×
nVidia GeForce 9800 GT	AMD Athlon 3200 at 2.0GHz	360×
nVidia GeForce 9800 GTX+	AMD Athlon 4850e at 2.5GHz	360×
nVidia Tesla C1060	Intel Xeon X5550 at 2.67GHz	230×

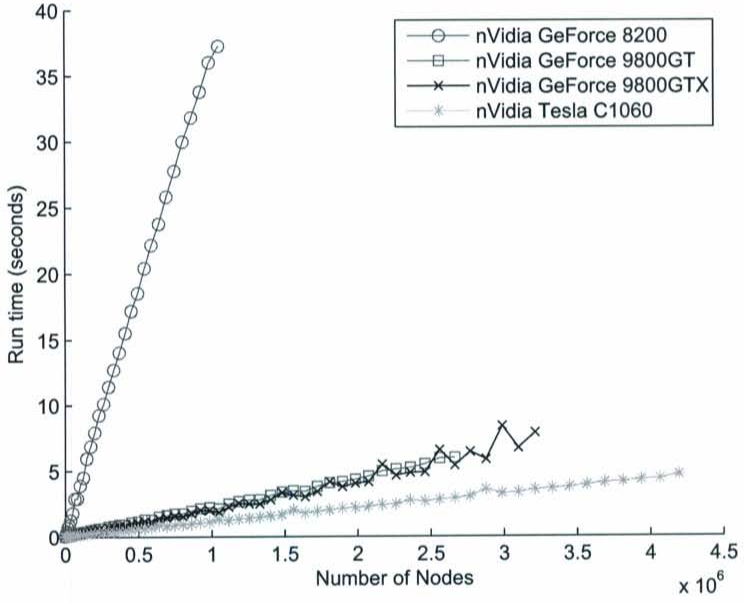

Figure 6.3: Comparison of run times for SIMPLE method across different GPUs

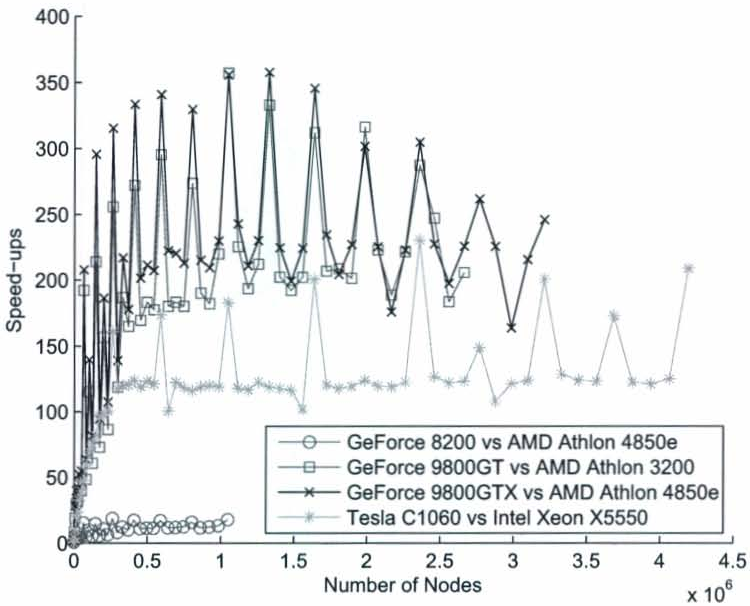

Figure 6.4: Speedups of GPU vs the CPU in each respective test machine for SIMPLE method

The speedup of GPUs with respect to CPUs in the same machines are difficult to measure since it is a comparison of the GPU against a different CPU than the one in the test machine. For example, Table 6.3 shows that the PISO test with a higher end GPU (nVidia Tesla C1060) has a smaller peak speedup (230×) than the PISO test with a much lower end GPU (nVidia GeForce 9800GT) (360×). For this case, we would expect a higher end GPU to have a larger peak speedup than the lower end GPU, but these results show the opposite. This discrepancy is due to the different CPUs (in each machine) that the GPUs are compared against for the speedup calculations. A better illustration of speedup would be produced by comparing all GPUs against the same CPU. Table 6.4 shows the speedup comparison of all GPUs against all CPUs of Table 6.2. Perhaps a better way to analyze this information is to view plots for which the data in this table was derived; Figures 6.5, 6.6, and 6.7 illustrate the speedup comparisons for all CPUs against all GPUs of Table 6.2 for the SIMPLE method.

Table 6.4: Test Results of all GPUs vs all CPUs for SIMPLE Method

GPU	CPU	Peak Speedup
nVidia GeForce 8200	AMD Athlon 4850e at 2.5GHz	20×
nVidia GeForce 9800 GT	AMD Athlon 4850e at 2.5GHz	300×
nVidia GeForce 9800 GTX+	AMD Athlon 4850e at 2.5GHz	350×
nVidia Tesla C1060	AMD Athlon 4850e at 2.5GHz	550×
nVidia GeForce 8200	AMD Athlon 3200 at 2.0GHz	20×
nVidia GeForce 9800 GT	AMD Athlon 3200 at 2.0GHz	350×
nVidia GeForce 9800 GTX+	AMD Athlon 3200 at 2.0GHz	410×
nVidia Tesla C1060	AMD Athlon 3200 at 2.0GHz	650×
nVidia GeForce 8200	Intel Xeon X5550 at 2.67GHz	10×
nVidia GeForce 9800 GT	Intel Xeon X5550 at 2.67GHz	120×
nVidia GeForce 9800 GTX+	Intel Xeon X5550 at 2.67GHz	135×
nVidia Tesla C1060	Intel Xeon X5550 at 2.67GHz	220×

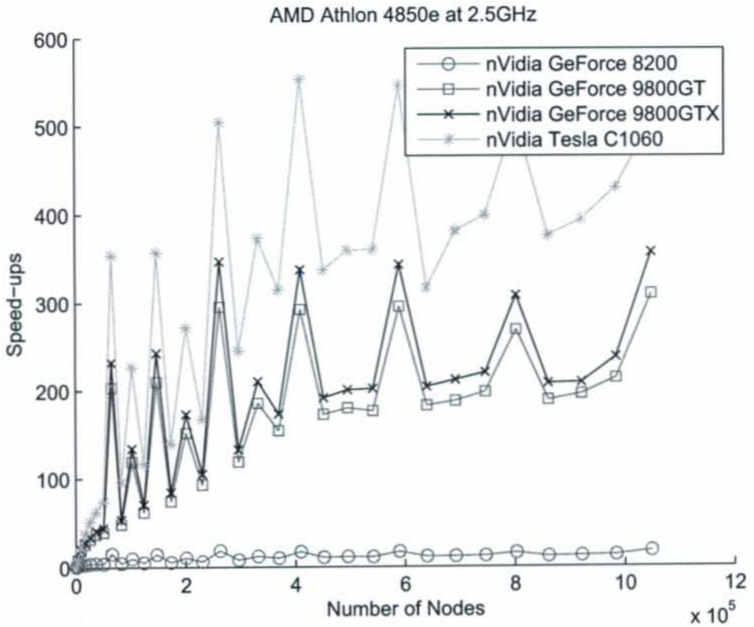

Figure 6.5: Full Speedup comparison for SIMPLE method using CPU AMD Athlon 4850e at 2.5GHz

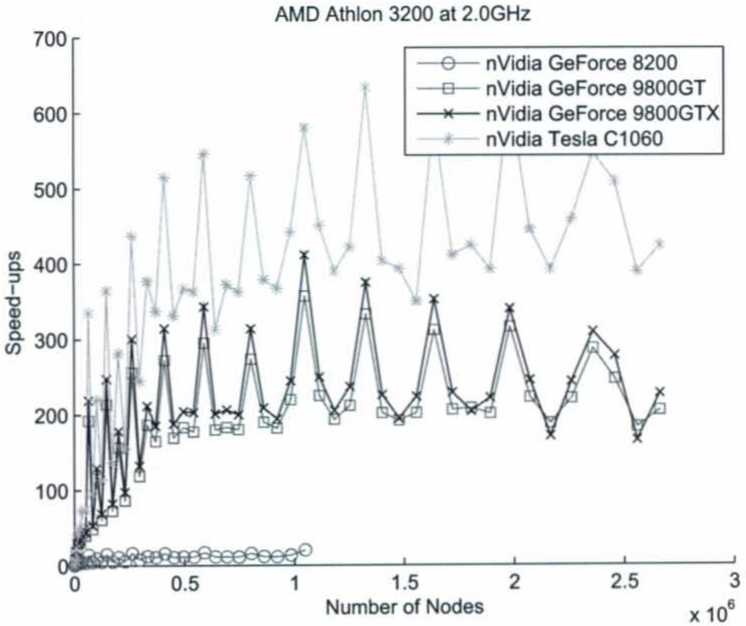

Figure 6.6: Full Speedup comparison for SIMPLE method using CPU AMD Athlon 3200 at 2.0GHz

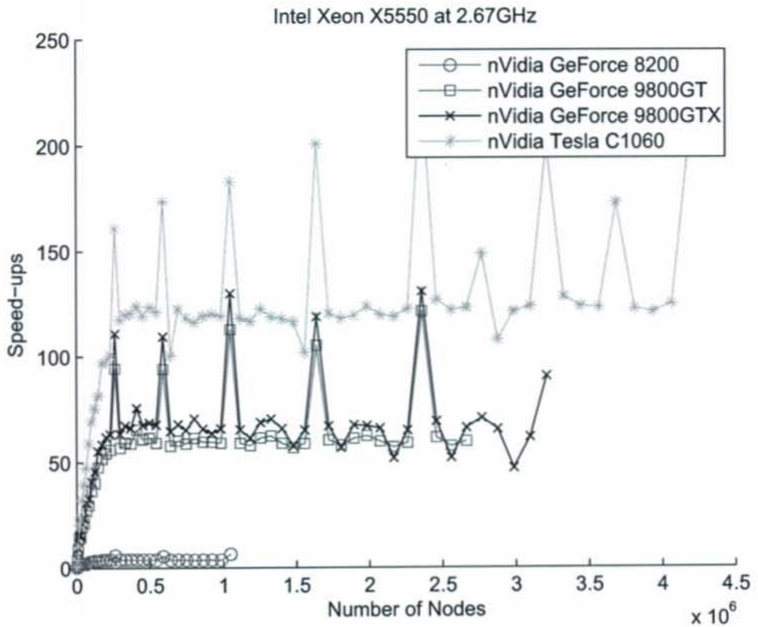

Figure 6.7: Full Speedup comparison for SIMPLE method using CPU Intel Xeon X5550 at 2.67GHz

6.2.3 The PISO Method

Figure 6.8 depicts run times as we increase the number of nodes in the numerical fluid system using the PISO method, while Figure 6.9 shows the speedups of these tests.

Table 6.5 shows the set of tests performed for the transient (PISO) developed in this work along with the hardware tested on (both GPU and CPU) and the peak speedup between the GPU and its respective CPU in the test machine.

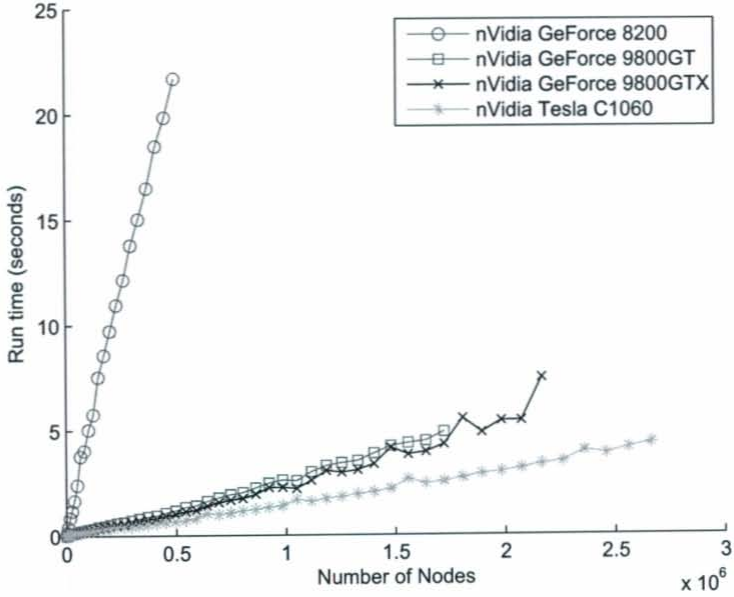

Figure 6.8: Comparison of run times for PISO method across different GPUs

Table 6.5: Speedup Results Per Machine for PISO Method

GPU	CPU	Peak Speedup
nVidia GeForce 8200	AMD Athlon 4850e at 2.5GHz	20×
nVidia GeForce 9800 GT	AMD Athlon 3200 at 2.0GHz	325×
nVidia GeForce 9800 GTX+	AMD Athlon 4850e at 2.5GHz	365×
nVidia Tesla C1060	Intel Xeon X5550 at 2.67GHz	165×

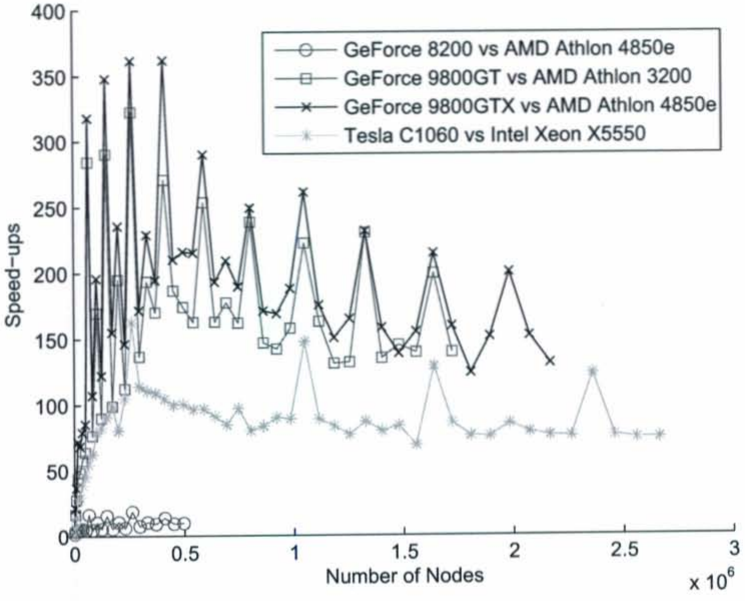

Figure 6.9: Speedup for PISO method

As mentioned previously, speedups of GPUs with respect to CPUs in the same machines are difficult to measure. Using the same argument as in the previous section we come to the conclusion that we require a speedup measure of all GPUs against all CPUs for the test machines. Table 6.6 shows the speedup comparison of all GPUs against all CPUs in Table 6.2. Perhaps a better way to analyze this information is to view plots for which the data in this table was derived; Figures 6.10, 6.11, and 6.12 illustrate the speedup comparisons for all CPUs against all GPUs in Table 6.2 for the PISO method.

Table 6.6: Test Results of all GPUs vs all CPUs For PISO Method

GPU	CPU	Peak Speedup
nVidia GeForce 8200	AMD Athlon 4850e at 2.5GHz	20×
nVidia GeForce 9800 GT	AMD Athlon 4850e at 2.5GHz	350×
nVidia GeForce 9800 GTX+	AMD Athlon 4850e at 2.5GHz	410×
nVidia Tesla C1060	AMD Athlon 4850e at 2.5GHz	550×
nVidia GeForce 8200	AMD Athlon 3200 at 2.0GHz	20×
nVidia GeForce 9800 GT	AMD Athlon 3200 at 2.0GHz	325×
nVidia GeForce 9800 GTX+	AMD Athlon 3200 at 2.0GHz	380×
nVidia Tesla C1060	AMD Athlon 3200 at 2.0GHz	490×
nVidia GeForce 8200	Intel Xeon X5550 at 2.67GHz	5×
nVidia GeForce 9800 GT	Intel Xeon X5550 at 2.67GHz	110×
nVidia GeForce 9800 GTX+	Intel Xeon X5550 at 2.67GHz	130×
nVidia Tesla C1060	Intel Xeon X5550 at 2.67GHz	165×

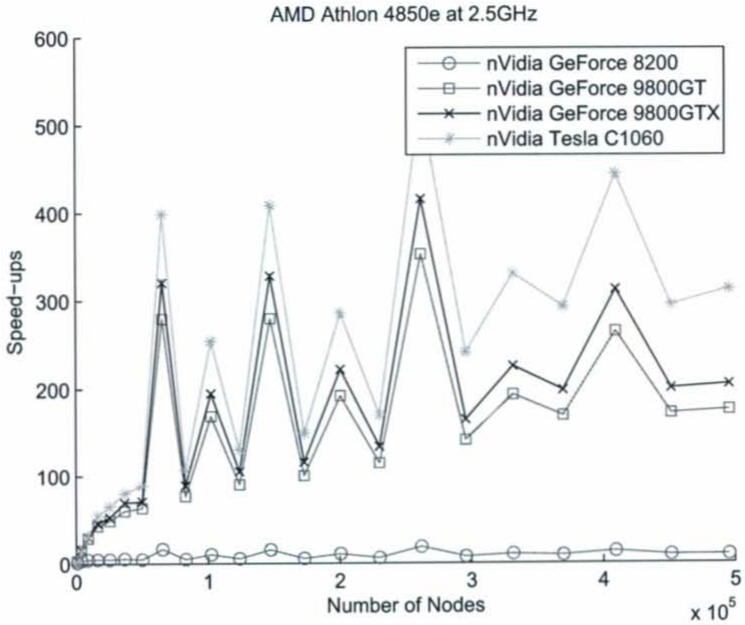

Figure 6.10: Full Speedup comparison for PISO method using CPU AMD Athlon 4850e at 2.5GHz

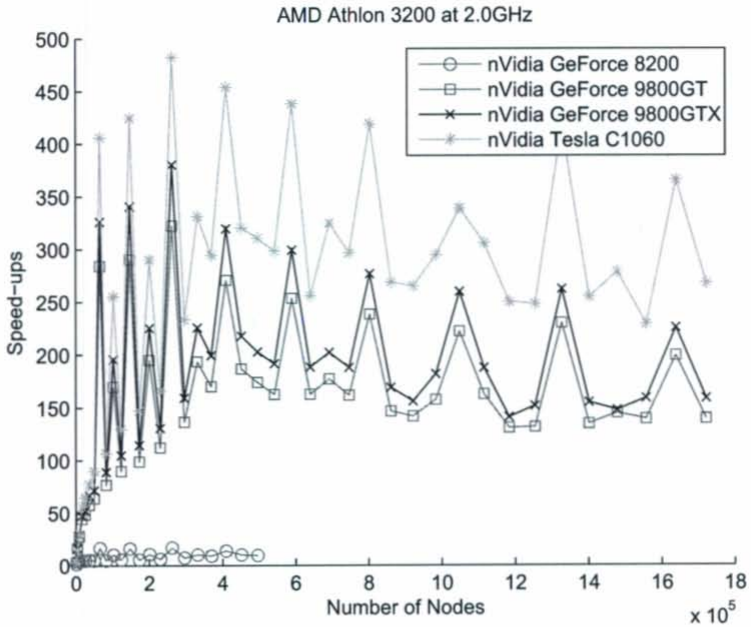

Figure 6.11: Full Speedup comparison for PISO method using CPU AMD Athlon 3200 at 2.0GHz

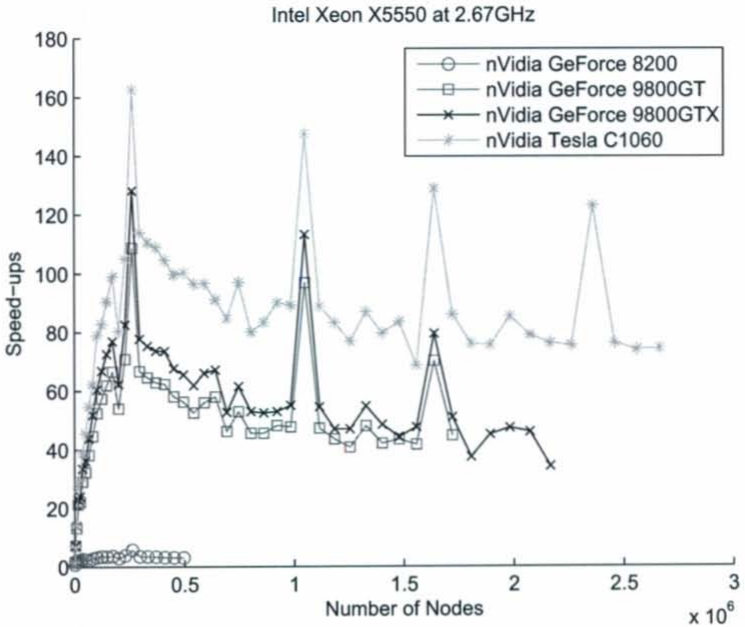

Figure 6.12: Full Speedup comparison for PISO method using CPU Intel Xeon X5550 at 2.67GHz

6.3 Memory Requirements

Figure 6.13 represents total memory usage (on the GPU at the time of query) of the SIMPLE method as the number of nodes in the system increases on all test GPUs described in Table 6.2. Memory usage represents the total memory usage on the GPU, which includes the usage of anything else involving the GPU at that time (such GPU initialization memory usage). Therefore a better measure would be relative memory usage, which is memory usage for a zero node system subtracted from the total memory usages just described. Figure 6.14 represents this relative memory usage as the number of nodes increases. Figures 6.15 and 6.16 represent these measures for the PISO method, respectively.

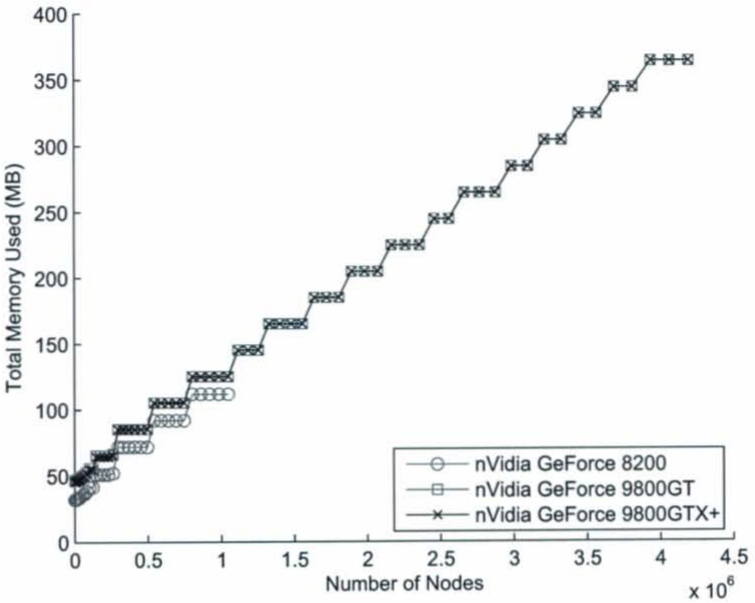

Figure 6.13: Total memory usage for SIMPLE methods per GPU

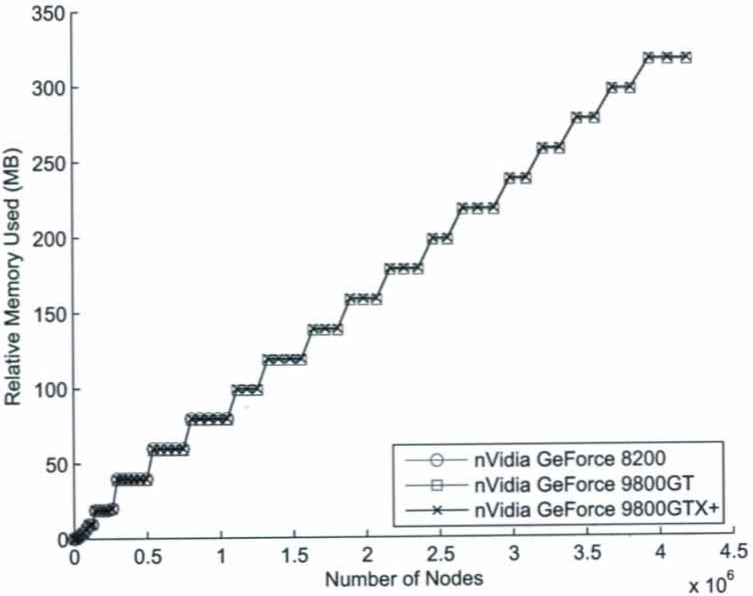

Figure 6.14: Relative memory usage for SIMPLE methods per GPU

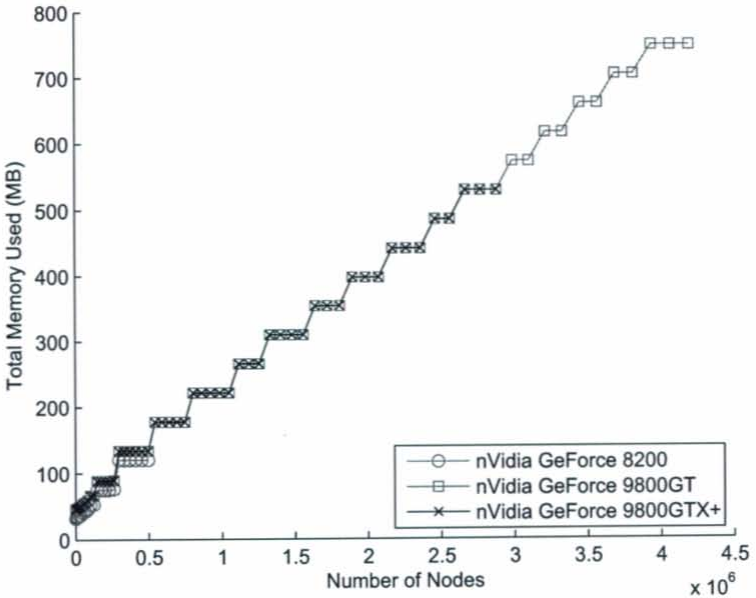

Figure 6.15: Total memory usage for PISO methods per GPU

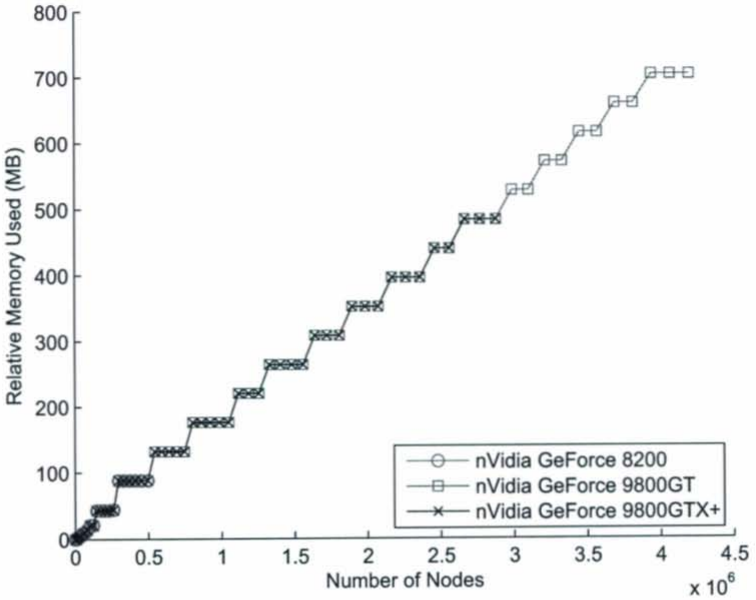

Figure 6.16: Relative memory usage for PISO methods per GPU

6.4 CUDA Occupancy

Please refer to Section 3.4.3 for a description of this result measurement on GPUs.

Table 6.7 represents a summary of the occupancies and min/max number of kernel invocations for all kernels implemented for both methods (SIMPLE and PISO), including the CG method, which is not used in either method, but for which performance measures were required and will be discussed in Section 7.1. For a detailed listing of occupancy data for each kernel, see Appendix B. From Table 6.7 we can calculate the average occupancy of the SIMPLE method kernels (including shared kernels) to be 74%, with a minimum number of kernel invocations per iteration of 28 and a maximum of 2018. We can also calculate the

average occupancy of the PISO method kernels (including shared kernels) to be 74%, with a minimum number of kernel invocations per iteration of 32 and a maximum of 2022.

Table 6.7: CUDA kernel occupancy and call data summary for PISO method

CUDA Kernel	Occupancy	Min Calls Per Iteration	Max Calls Per Iteration
SIMPLE Method Kernels			
constructCoefficients_pc	33%	1	1
constructCoefficients_uv	33%	1	1
applyCorrections_kernel	83%	1	1
PISO Method Kernels			
constructCoefficients_pc	33%	1	1
constructCoefficients_pcc	33%	1	1
constructCoefficients_uv	33%	2	2
apply_piso_corrections	100%	1	1
applyCorrections_kernel	83%	1	1
calculate_uc_vc	83%	1	1
Shared Kernels			
get_drag_lift	100%	1	1
redblack_shared_maxres_iteration_kernel	100%	20	2010
setBoundaryValues_kernel	100%	1	1
reduce	100%	3	3
Conjugate Gradient Kernels			
vec_element_mult	100%	0	0
scalar_plus_scalar_vec_mult	100%	0	0
vec_vec_subtract	100%	0	0
mat_vec_mult	67%	0	0
vec_scalar_product	50%	0	0
scalar_subtract_scalar_vec_mult	100%	0	0
vec_vec_mult_element	100%	0	0

7 Discussion

7.1 Solving systems of linear equations

The conjugate gradient method may be the most efficient approach in terms of number of iterations required to converge (see Section 2.7.4) to a solution, but this does not necessarily mean it is the most efficient performance wise when compared to other linear solvers on the GPU. Section 6.1 contains Figures (6.1 and 6.2) that show tests of the two GPU linear solvers that we have implemented for this work, SOR and conjugate gradient methods. As we can see from these figures, the run time for the CG method is almost a factor of 10^3 greater than that for the SOR method at the largest possible number of nodes that we could get a test to complete (due to a 5 second maximum kernel runtime timeout on nVidia GPUs). If the trend were to continue for the CG method all the way to the 4×10^6 nodes that the SOR method was terminated at (purposely stopped, no restrictions at that point for SOR method), the run time would be ever more infeasible. Indeed, if the run time were to increase linearly it would be about 70 seconds per iteration at 4×10^6 nodes.

The largest bottleneck in the CG method we suspect to be the time required to load and execute the CUDA kernel on a GPU each time a matrix-vector multiplication or a vector-vector addition is required. Since, according to the CG Algorithm (Algorithm 2.6), each CG iteration requires 8 kernel calls (8 of these operations) as opposed to the SOR method which requires only 2 kernel calls (with the red-black parallel technique) per iteration. This enforces the suspicion that kernel load time is very probably the largest bottleneck.

Although the CG method converges much faster than the SOR method, these run time results caused our choice of the solver for systems of linear equations (within the CFD solution methods) to be the SOR method.

7.2 Performance

The discussion in this section corresponds to the results presented in Section 6.2. As we can see from Figure 6.3 (representing the run time per iteration of the GPU SIMPLE method), the nVidia GeForce 8200 GPU is the slowest by a large margin of all test GPUs, while the nVidia GeForce 9800 GT and GTX+ GPUs are very similar and in the middle of the pack, and finally the nVidia Tesla C1060 is the fastest per iteration when running the SIMPLE method algorithm developed for the GPU.

Figure 6.4, representing the speedup of the GPU vs the respective CPU on each test machine, and Table 6.3, representing the peak speedup of the GPU vs the respective CPU on each test machine shows, that the GeForce 9800 GT(X+) cards perform the best. This, however, is a misleading test, since the CPUs on these machines vary drastically. The CPU on the Tesla machine is much more powerful than the rest of the CPUs. The purpose of displaying this type of speedup (per machine) is to illustrate the different speedups that can be achieved for different machines, since usually a user running the software would not have multiple machines to mix and match the GPUs and CPUs in order to find the best combination. Even on the very powerful CPU, the GPU (although it is itself the most powerful of the test set) shows a speedup of $230\times$. This is substantial, especially considering that it happens at every iteration, and there could be thousands of iterations per execution.

Figures 6.5, 6.6, 6.7, and Table 6.4 summarize these figures for peak speedups and represent a mix and match of the speedup tests for all combinations of GPU vs CPU in the test set. The purpose of this information is to illustrate the performance increase in different machine setup situations. For example on the least powerful CPU (the AMD Athlon 3200, which is still very similar to the AMD Athlon 4850e), we obtain a speedup of up to a $650\times$ with the Tesla GPU, all the way down to a speedup of $20\times$ with the GeForce 8200 GPU. In fact, the most illustrative results are showing the mid-level GPUs (the 9800 GT(X+)'s), which obtain a speedup of $350\times$ to $410\times$ on this CPU. Since this type of card is very

common amongst normal users and machines (and is a very affordable card), we see the amount of performance gain one can achieve with moderate means.

Perhaps the most interesting result is that of the GPUs on the most powerful CPU (the Intel Xeon X5550). Traditionally, the way to decrease run time of CFD code was to throw more processing power at it, which is similar to these tests (since this is a powerful processor), but these tests show that even with such a CPU one can achieve a further speedup of up to $220\times$. Even at the lower end a 10 fold speedup is achievable, or about a 120 to 135 fold speedup for the average GPU. These results demonstrate that if we are running our simuation on the least powerful card we can find (the onboard GeForce 8200) and our simulation takes 100 iteations to converge (which is a very low estimate) we have achieved an overall (CFD simulation run time) speedup of $10 * 100 = 1000$ times over the fastest CPU we could find.

A note must be made on the oscillatory behavior of all speedup results in Figures 6.3, 6.5, 6.6, 6.7. This behavior is suspected to be caused by a combination of the system size affecting the GPU multiprocessor occupancy (since it performs best when the system is a power of 2) and of host memory paging. This suspicion is supported by the fact that the peaks of the oscillations are at systems sizes that are a power of 2.

All of these results have the same trend for the PISO method as they did for the SIMPLE method, as can be seem from the figures in Section 6.2.3. There is a slight slow-down when testing the PISO method (compared to the SIMPLE method). This is expected and is due to the extra steps required per iteration (see Section 2.6), which involve construction of another set of coefficients, solving another system of linear equations and applying corrections a second time. Although there is a slow-down, we still achieve a very good speedup, even the least powerful GPU on the most powerful CPU produces a speedup of $5\times$ per PISO iteration, while the most powerful GPU on the least powerful CPU produces a peak speedup of $490\times$.

7.3 Memory Usage

Memory requirements for this work are important since the amount of memory on GPUs is many factors smaller than that of the host system on which the GPUs run, see Table 6.2 for the amount of memory per test machine used to measure performance of this work, whereas it would be difficult to find a personal computer today with less than 1 to 2 GB of memory. Many have even more memory than that. Another issue with memory on GPUs vs memory on their hosts is that it is much more difficult to upgrade the amount of memory on GPUs than on host machines (PCs).

Memory usage for this work is actually at the low end of the spectrum of possible memory usage, depending of the mesh type and discretization technique. This means that if we had chosen another mesh type, such as an unstructured mesh, which would require lookup tables for all flow values and matrix coefficient arrays along with memory for size, position, and direction arrays for vertices and edges of the mesh elements, then the memory requirement would be much higher. Since this work uses a structured mesh type, the mesh is uniform and therefore the code requires no lookup tables, positions, directions, or sizes of the mesh elements (since they can be produced from mesh dimensions and size, which are single integer values as opposed to arrays). Therefore, structured meshes have a lower memory usage.

The peak memory usage of the SIMPLE method is about 40 MB for a mesh of size 1024x256 nodes (which is the mesh size used in the shape design optimization application described in Section 8.1), compared to the PISO method, which has a peak memory usage of about 63 MB for the same mesh size. The reason for the increase in memory usage between the SIMPLE method and the PISO method is that the PISO method requires extra flow information and concurrent coefficient calculations for use in the extra steps to solve the transient system (see Section 2 for a description of the this method compared to the SIMPLE method).

The fact that peak memory usage is so low (less than 100 MB) for practical applications is a very good indication of the practicality of this approach on GPUs. The size of the systems can be greatly increased, for example in direct numerical turbulence modeling (see Section 8.2), without worrying about the use of excessive memory for the GPU to handle.

7.4 CUDA Occupancy

The most important kernel is the one that is invoked the most frequent, which is the *red-black_shared_maxres_iteration_kernel*. This kernel is invoked anywhere between 20 and 2000 times per iteration, compared to other kernels which are only invoked a few times per iteration. This kernel uses 9 registers per thread and 800 bytes of shared memory per block, 128 threads per block this is just below the 100% threshold. If just another 2 registers were in use this would drop to 83% occupancy which would cause a very large decrease in performance due to the frequency of invocation of this kernel. Again, the full details of the occupancy information for this kernel can be found in appendix B.

The worst occupancy is achieved by the three matrix coefficient construction methods (*constructCoefficients_**), at 33% each. Although these are only invoked a total of between 2 and 4 times per iteration, this is still a very low CUDA occupation results. The reasons for the low occupation is the use of so many registers, about 31 registers per thread, while shared memory per block used is negligable. The multiprocessors on the GPU have a limited amount of registers to use for each thread, 31 at 128 threads per block, which gives about 3968 registers required for each block. Since most GPUs only allow 4096 registers per thread block this only allows 2 threads per mutliprocessor, therefore this kernel is limited by the number of registers. So many registers are in use because the coefficient construction is one of the most complicated kernels discussed here especially with respect to the memory locations required, since it requires access to all current and past (previous of iteration) local flow variables (velocity and pressure) and their neighboring values, the

physical and numerical flow parameters such as viscosity, density, system size, mesh parameters, along with write access to the memory locations for the actual coefficients (which for 2 dimensions consists of six arrays). Each one of these parameters requires a register to pass the data to the kernel.

So where does this leave us? First, we have optimized the most important kernel, *redblack_shared_maxres_iteration_kernel*, for use of the nVidia GPUs. Second, there are some low occupancies, which means that there is room for improvement in the design of the kernels with low occupancies.

The discussion of conjugate gradient occupancies were intentionally left out until now since we have determined by the performance measurements (Section 7.1) that the SOR method was the superior choice for all CFD methods on GPUs. Still we can note that all CG kernels are 100% occupancy except for the matrix-vector multiplication and the scalar product (which is very similar to the reduce method discussed above). The fact that there are so many kernels for the CG method itself is a negative result, since we have already determined that the load/execution time of the kernels is what causes the reduction in efficiency for this method.

8 Applications

8.1 Evolutionary Shape Design Optimization

The purpose of the design of these CFD algorithms for GPUs was to increase performance enough to allow for their use as fitness functions in a genetic programming (GP) technique used for optimized shape design. Since traditional CPU algorithms require too much time to allow for a realistic convergence of GP, which may require millions of evaluations, these GPU methods were designed to perform GP on a cluster of machines (each equipped with a GPU).

The optimized shape design problem is a problem where we are given a fluid system (e.g. wind tunnel) and some constraints for an obstacle in the system (such as minimum shape), and expected to provide results in the form of a shape around the initial constraints that would optimize specific parameters (such as drag and/or lift). For example, we could be given the frame of a car (with car seats, engine, etc) and would require the method to generate a shape that could be used as the optimal aerodynamic body for that specific car design.

Figure 8.1 illustrates a sample evolution using this technique. This image depicts a timeline (top to bottom) of evolving shapes, beginning with the shape with the most drag profile and progressively decreasing the drag as we move towards the bottom. This work uses the self-modifying cartesian genetic programming (SMCGP) technique developed in [15] and is in collaboration with Simon Harding and Wolfgang Banzhaf. At each evolution step in the GP an evaluation (CFD solution) occurs on each individual in the GP population. For the purpose of this work an individual is considered to be the fluid system (obstacle shape), and is generated through the SMCGP method.

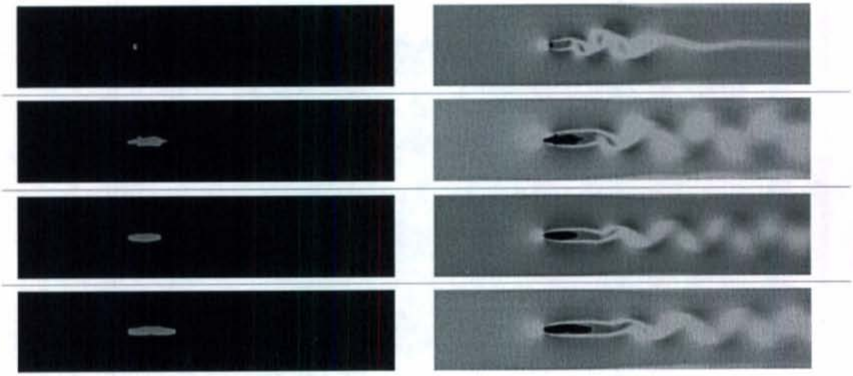

Figure 8.1: Sample Evolution

To tackle the optimized shape design problem we expect that a million evaluations are required. Table 8.1 illustrates run time estimates for the GP to converge with a 1024x512 discretized fluid system. As shown in this table, it is evident that to perform this shape optimization on a single CPU is very impractical (as it would require 10 years). But it requires only 11 hours on a cluster of 50 (average) GPUs.

Table 8.1: Optimized Shape Design Problem: Estimated Times

Hardware	GP Convergence Time
1 CPU	10 years
50 CPU Cluster	70 days
1 GPU	1.4 years
50 GPU Cluster	10 days

8.2 Direct Turbulence Modeling

Direct turbulence modeling, also called direct numerical simulation (DNS), is the practice of solving time-dependent governing equations (as defined in Section 2.1) on a sufficiently fine spatial mesh and with a sufficiently small time resolution in order to resolve the smallest turbulent eddies and the fastest fluctuations.

DNS is useful for the development and validation of new turbulence models, for measurement of flow details that cannot be measured with traditional turbulence models or that are too detailed to be measured, or for advanced experimental techniques such as calibrating hot-wire anemometry probes in near-wall turbulence [32], or even extending turbulence measurements to compressible flows which could be useful in the development and testing of advanced high-speed mixing techniques.

DNS has many disadvantages, compared to other turbulence simulation techniques, among them is that it is very computationally expensive. To resolve the varying degrees of length and time scales required to perform DNS would require a very fine spatial mesh and very small time scales. For example, to resolve the smallest and largest turbulence length scales a DNS of a turbulent flow with a Reynolds number of 10^4 (just above the turbulent flow threshold) one would require on the order of 10^3 mesh nodes in each coordinate direction [32].

Although this disadvantage of DNS is reasonable, it can be overcome by more powerful hardware and more efficient algorithms. The technique developed in this work is exactly that, a more efficient algorithm running on more powerful hardware. As we have seen from the results of this work, we can achieve very large increases in efficiency by taking advantage of the GPU architecture, which can be applied to DNS to overcome this disadvantage.

9 Future Work

The methods developed here are a proof of concept in that we have developed basic technique and algorithms for efficient general purpose CFD simulations on GPUs. These methods can be extended in many ways to further increase not only its efficiency but its practical applications.

Improvements in efficiency can be achieved through an extension of the methods on multiple GPUs at once, whether the GPUs be in the same machine or across a network of machines (in distributed network), which would give multiple levels of parallelism. Such parallelism would be a relatively simple extension in that it would require only to decompose the problem into another set of domains to execute across each GPU. Another improvement in efficiency can be achieved through the extension of the linear solvers used to a multigrid method. Multigrid methods use multiple resolutions of the mesh in order to solve a systems of linear equations with increased efficiency. There has already been work done in the area of multigrid techniques on GPUs since we started this work (see [11]).

Besides improvements in efficiency, the practicality of the code could be improved. One could extend the approach to allow for simulations on unstructured meshes. Unstructured meshes would permit more complex flow with less mesh nodes (meaning more efficiency) because with unstructured meshes one can generate the mesh around obstacles and put more mesh nodes near computationally complex areas that require more accuracy (such as boundaries or wakes) and less in areas where the flow is simple and requires less accuracy.

10 Conclusion

The purpose of this thesis was the development of efficient and practical general purpose methods for simulating fluid flow on graphics processing units. CFD is computationally expensive and requires a lot of processing power to perform even moderate simulations in a reasonable period of time. With the techniques described in this work we can perform fast and accurate fluid simulations with no major loss of accuracy to improve performance.

In Chapter 5 we developed the techniques and algorithms used on the GPUs simply by extending traditional techniques onto this modern hardware. Chapters 6 and 7 illustrated the performance gains that this work resulted in for general purpose fluid simulations. These results show that we achieve very high (up to $650\times$) speedups per iteration (time step) with the simple and inexpensive hardware presented in this work. We have also seen these speedups are per time step, so that overall the speedup increases dramatically as most fluid simulations require hundreds, sometimes thousands, of time steps to complete.

References

[1] J. D. Anderson Jr. *Computational Fluid Dynamics: The Basics with Applications*. McGraw-Hill Inc, 1995.

[2] I. E. Barton. Comparison of SIMPLE- and PISO-type Algorithms for Transient Flows. *International Journal for Numerical Methods in Fluids*, 26:459–483, 1998.

[3] N. Bell and M. Garland. Efficient sparse matrix-vector multiplications on cuda. nVidia Technical Report NVR-2008-004, nVidia Corporation, 2008.

[4] M. Bern and P. Plassmann. Mesh generation. *Handbook of Computational Geometries, Elsevier Science*, 2000.

[5] J. Blazek. *Computational Fluid Dynamics: Principles and Applications*. Elsevier Science Ltd, Oxford, UK, 2001.

[6] T. Cebeci. *Turbulence Models and Their Applications*. Horizons Publishing Inc, Long Beach, California, 2005.

[7] T. Cebeci, J. P. Shao, F. Kafyeke, and E. Laurendeau. *Computational Fluid Dynamics for Engineers*. Horizons Publishing Inc, Long Beach, California, 2005.

[8] ChmlTech Ltd. CFD grid generation methods. http://www.chmltech.com/cfd/grid_generation.pdf.

[9] H. Courtecuisse and J. Allard. Parallel dense gauss-seidel algorithm on many-core processors. *High Performance Computation Conference (HPCC), IEEE CS Press*, 2009.

[10] K. Crane. GPU Fluid Solver. Web, 2006. http://www.cs.caltech.edu/~keenan/project_fluid.html.

[11] Z. Feng and P. Li. Multigrid on gpu: tackling power grid analysis on parallel simt platforms. *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, 2008.

[12] J. H. Ferziger and M. Peric. *Computational Methods for Fluid Dynamics*. Springer-Verlag Berlin Heidelberg, third edition, 2002.

[13] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, Maryland, third edition, 1996.

[14] S. Haque. Convergence of the successive overrelaxation method. *IMA Journal of Numerical Analysis*, 7:307–311, 1987.

[15] S. L. Harding, J. F. Miller, and W. Banzhaf. Self-modifying cartesian genetic programming. *GECCO '07 Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007.

[16] P. K. Kundu and I. M. Cohen. *Fluid Mechanics*. Elsevier Inc, fourth edition, 2008.

[17] N. Lambropoulos, E. S. Politis, K. C. Giannakoglou, and K. D. Papailiou. Co-located pressure-correction formulations on unstructured 2-D grids. *Computational Mechanics*, 27:258–264, 2001.

[18] R. J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industry and Applied Mathematics Proceedings, 2007.

[19] W. Li, Z. Fan, X. Wei, and A. Kaufman. *GPU Gems 2: Chaptor 47, Flow Simulations with Complex Boundaries*. Pearson Education Inc, Upper Saddle River, NJ, 2005.

[20] Y. Liu, X. Lui, and E. Wu. Real-time 3D fluid simulations on GPU with complex obstacles. *Proceedings of the 12th Pacific Conference on Computer Graphics and Applications (PG04)*, 2004.

[21] Lomax, Pulliam, and Zingg. *Fundamentals of Computational Fluid Dynamics*. Springer-Verlag Berlin Heidelberg, 2001.

[22] nVidia. *CUDA Programming Guide*. nVidia Corporation, 2009. Version 2.3.

[23] N. Osborne. GPU Fluid Simulation. Intern report, School of Computer and Information Science, Edith Cowan University, 2009.

[24] S. V. Patankar and D. B. Spalding. A calculation procedure for heat, mass, and momentum transfer in three-dimensional parabolic flows. *International Journal of Heat Mass Transfer*, 15:1787–1806, 1972.

[25] Sengupta, Harris, and Garland. Efficient parallel scan algorithms for gpus. nVidia Technical Report NVR-2008-003, nVidia Corporation, 2008.

[26] T. W. H. Sheu and R. K. Lin. An incompressible navier-stokes model implemented on nonstaggered grids. *Numerical Heat Transfer*, 44(Part B):277–294, 2003.

[27] A. F. Shinn. Computational fluid dynamics (cfd) using graphics processing units. Mechanical Science and Engineering Dept., UIUC, 2009.

[28] A. F. Shinn. Implementation issues for cfd algorithms on graphics processing units. Mechanical Science and Engineering Dept., UIUC, 2009.

[29] Shyy, Udaykumar, Roa, and Smith. *Computational Fluid Dynamics with Moving Boundaries*. Dover Publications Inc, Mineola, NY, 1996.

[30] T. Stoesser. Solution Methods for the Navier Stokes Equations. Course Notes, Spring 2007. Georgia Tech, Computational Fluid Dynamics, CE7751–ME7751.

[31] Z. Tong, Q. Huang, J. He, and J. Han. An improved study of physically based fluid simulations on gpu. *Computer-Aided Design and Computer Graphics, 2007 10th IEEE International Conference*, 2007.

[32] Versteeg and Malalasekera. *Computational Fluid Dynamics: The Finite Volume Method.* Pearson Education Limited, Harlow, England, second edition, 2007.

[33] N. Wolovick. Peak performance for an application in cuda. Universidad Nacional de Cordoba, Argentina, May 2010.

[34] E. Wu and Y. Liu. Emerging technology about GPGPU. *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference*, 2008.

A 2D Finite-Volume Discretization of Governing Equations on Structured Grid

In order to discretize the momentum equations, which are just the Navier-Stokes equation for each dimension, using the finite-volume discretization technique we make use of equation 5. Since we are applying a uniform grid our control volume (CV) is constant, and since we are in 2 dimensions it will be an area, A, where $A = dx * dy$, with dx and dy the infinitesimal horizontal and vertical node sizes. This leads to the 2 dimensioned finite-volume discretization of the Navier-Stokes equation for a uniform grid

$$\rho \int_A (\vec{u} \cdot \nabla \vec{u})\, dA = -\int_A \nabla p\, dA + \int_A \mu \nabla^2 \vec{u}\, dA \tag{44}$$

In 2 dimensions, if we expand this out using the components $\vec{u} = u\hat{i} + v\hat{j}$, this will simplify to two momentum equations

$$\textbf{u: } \rho \int_A \left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} \right) dA = -\int_A \frac{\partial p}{\partial x} dA + \int_A \mu \left(\frac{\partial}{\partial x}\frac{\partial u}{\partial x} + \frac{\partial}{\partial y}\frac{\partial u}{\partial y} \right) dA \tag{45}$$

$$\textbf{v: } \rho \int_A \left(v\frac{\partial v}{\partial y} + u\frac{\partial v}{\partial x} \right) dA = -\int_A \frac{\partial p}{\partial y} dA + \int_A \mu \left(\frac{\partial}{\partial x}\frac{\partial u}{\partial x} + \frac{\partial}{\partial y}\frac{\partial u}{\partial y} \right) dA \tag{46}$$

Taking just one of these equations, say Eq. 45, the u momentum equation, and using the fact that in 2 dimensions the control volume will be an area, A, where $A = dx * dy$ and dx and dy are the horizontal and vertical node sizes in the mesh, we can rewrite Eq. 45 as

$$\rho \int_{dx}\int_{dy} u\frac{\partial u}{\partial x}\,dxdy + \rho \int_{dx}\int_{dy} v\frac{\partial u}{\partial y}\,dxdy$$
$$= -\int_{dx}\int_{dy} \frac{\partial p}{\partial x}\,dxdy + \int_{dx}\int_{dy} \mu\frac{\partial}{\partial x}\frac{\partial u}{\partial x}\,dxdy + \int_{dx}\int_{dy} \mu\frac{\partial}{\partial y}\frac{\partial u}{\partial y}\,dxdy \tag{47}$$

where the integrals can be solved in any order. If we linearize this equation, by letting the extra u and v on the left-hand side become constant, and with the knowledge that

$$\int_{dx} dx = \Delta x\,,\ \int_{dy} dy = \Delta y \tag{48}$$

where Δx and Δy are the actual horizontal and vertical node sizes in the mesh, solving all integrals of Eq. 47 becomes

$$\left[\left(\rho u_{const}\Delta y\right)_e u_e - \left(\rho u_{const}\Delta y\right)_w u_w\right] + \left[\left(\rho v_{const}\Delta x\right)_n u_n - \left(\rho v_{const}\Delta x\right)_s u_s\right] =$$
$$-\left(p_e - p_w\right)\Delta y + \left[\mu_e \Delta y\left(\frac{\partial u}{\partial x}\right)_e - \mu_w \Delta y\left(\frac{\partial u}{\partial x}\right)_w\right]$$
$$+\left[\mu_n \Delta x\left(\frac{\partial u}{\partial y}\right)_n - \mu_s \Delta x\left(\frac{\partial u}{\partial y}\right)_s\right] \tag{49}$$

where we employ the east-west-north-south notation to define neighboring nodes (capital E,W,S,N,P, where P is the central point) and neighboring faces (lowercase e,w,s,n) between nodes as illustrated in Figure A.1.

Since the non-constant values of u and p are all on the control volume faces we don't have values at these locations because the finite-volume method stores values at node points at the center of control volumes. We therefore need to approximate these unknown face values by known node values. Using central difference approximation we get the equations

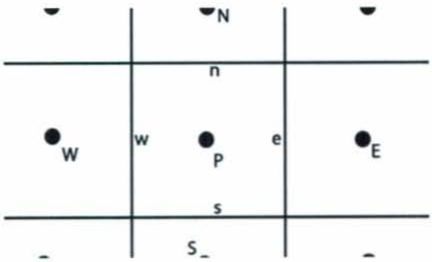

Figure A.1: east-west-north-south notation to define neighboring nodes

$$
\begin{aligned}
u_e &= \frac{u_E + u_P}{2} \\
u_w &= \frac{u_P + u_W}{2} \\
u_n &= \frac{u_N + u_P}{2} \\
u_s &= \frac{u_P + u_S}{2} \\
p_e &= \frac{p_E + p_P}{2} \\
p_w &= \frac{p_P + p_W}{2}.
\end{aligned}
\tag{50}
$$

With a simple difference approximation to approximate $\left(\frac{\partial u}{\partial x}\right)$ and $\left(\frac{\partial u}{\partial y}\right)$ we define

$$
\begin{aligned}
\left(\frac{\partial u}{\partial x}\right)_e &= \frac{u_E - u_P}{\Delta x} \\
\left(\frac{\partial u}{\partial x}\right)_w &= \frac{u_P - u_W}{\Delta x} \\
\left(\frac{\partial u}{\partial y}\right)_n &= \frac{u_N - u_P}{\Delta y} \\
\left(\frac{\partial u}{\partial y}\right)_s &= \frac{u_P - u_S}{\Delta y}
\end{aligned}
$$

Rewriting Eq. 49 with these approximations we get

$$\left(\frac{1}{2}\rho u_{const}\Delta y\right)_e (u_E + u_P) - \left(\frac{1}{2}\rho u_{const}\Delta y\right)_w (u_P + u_W)$$
$$+ \left(\frac{1}{2}\rho v_{const}\Delta x\right)_n (u_N + u_P) - \left(\frac{1}{2}\rho v_{const}\Delta x\right)_s (u_P + u_S) =$$
$$-\frac{1}{2}(p_E - p_W)\Delta y + \mu_e \Delta y \left(\frac{u_E - u_P}{\Delta x}\right) - \mu_w \Delta y \left(\frac{u_P - u_W}{\Delta x}\right)$$
$$+ \mu_n \Delta x \left(\frac{u_N - u_P}{\Delta y}\right) - \mu_s \Delta x \left(\frac{u_P - u_S}{\Delta y}\right) \tag{51}$$

If we let

$$\begin{aligned} F_e &= \frac{1}{2}\rho u_e \Delta y \\ F_w &= \frac{1}{2}\rho u_w \Delta y \\ F_n &= \frac{1}{2}\rho v_n \Delta x \\ F_s &= \frac{1}{2}\rho v_s \Delta x \end{aligned} \tag{52}$$

$$\begin{aligned} D_e &= \mu_e \frac{\Delta y}{\Delta x} \\ D_w &= \mu_w \frac{\Delta y}{\Delta x} \\ D_n &= \mu_n \frac{\Delta x}{\Delta y} \\ D_s &= \mu_s \frac{\Delta x}{\Delta y} \end{aligned} \tag{53}$$

Eq. 51 further simplifies to

$$F_e(u_E + u_P) - F_w(u_P + u_W) + F_n(u_N + u_P) - F_s(u_P + u_S) =$$
$$-\frac{1}{2}(p_E - p_W)\Delta y + D_e(u_E - u_P) - D_w(u_P - u_W)$$
$$+ D_n(u_N - u_P) - D_s(u_P - u_S) \tag{54}$$

which can again be rearranged to produce

$$a_E u_E + a_W u_W + a_S u_S + a_N u_N + a_P u_P = b \tag{55}$$

Here

$$a_E = F_e - D_e$$
$$a_W = -F_w - D_w$$
$$a_N = F_n - D_n \tag{56}$$
$$a_S = -F_s - D_s$$
$$a_P = F_e - F_w + F_n - F_s + D_e + D_w + D_n + D_s$$
$$b = \frac{1}{2}(p_W - p_E)\Delta y \tag{57}$$

which gives us the final 2 dimensions finite-volume discretization for a uniform grid for the u momentum equation. The same technique can be used to discretize the v momentum equation to produce

$$a_E v_E + a_W v_W + a_S v_S + a_N v_N + a_P v_P = b \tag{58}$$

where Eq. 56 and Eq. 52 still apply.

B CUDA Occupancy Data

The following figures represent the CUDA occupancy for all kernels in this work.

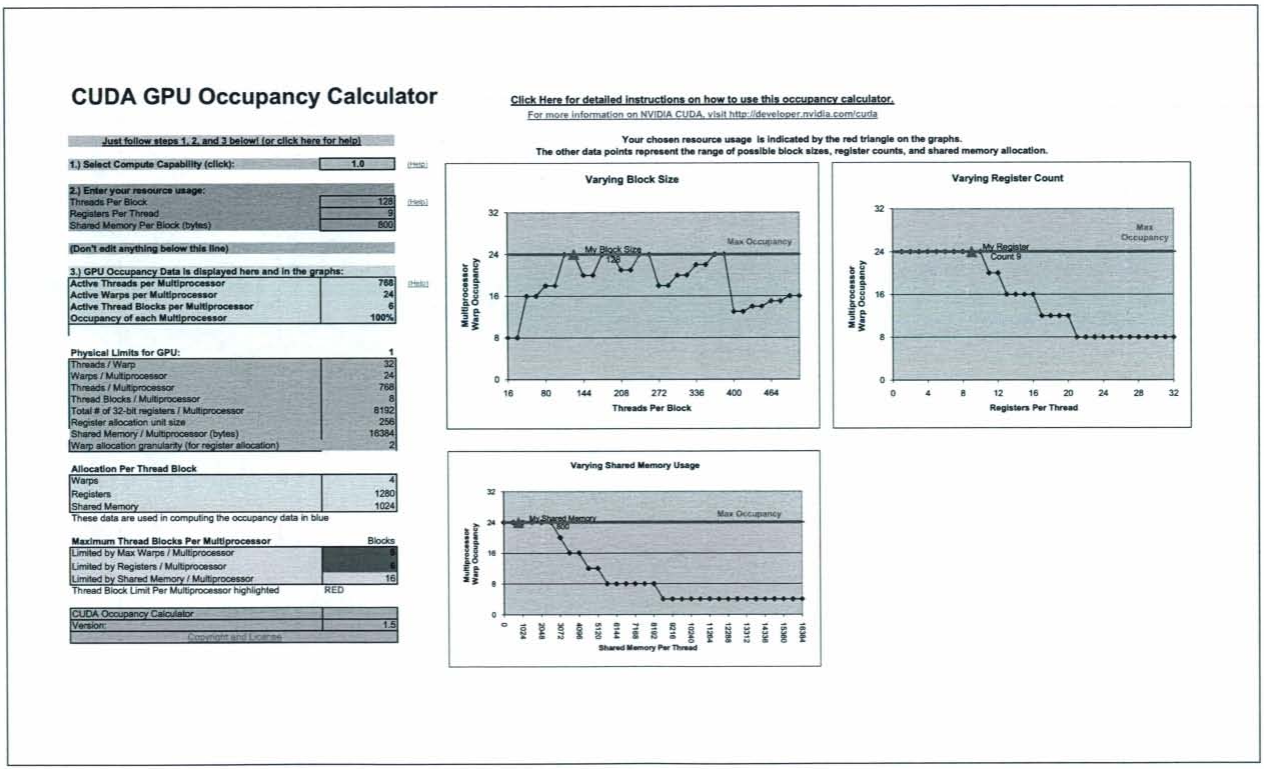

Figure B.1: redblack_shared_maxres_iteration_kernel kernel occupancy data

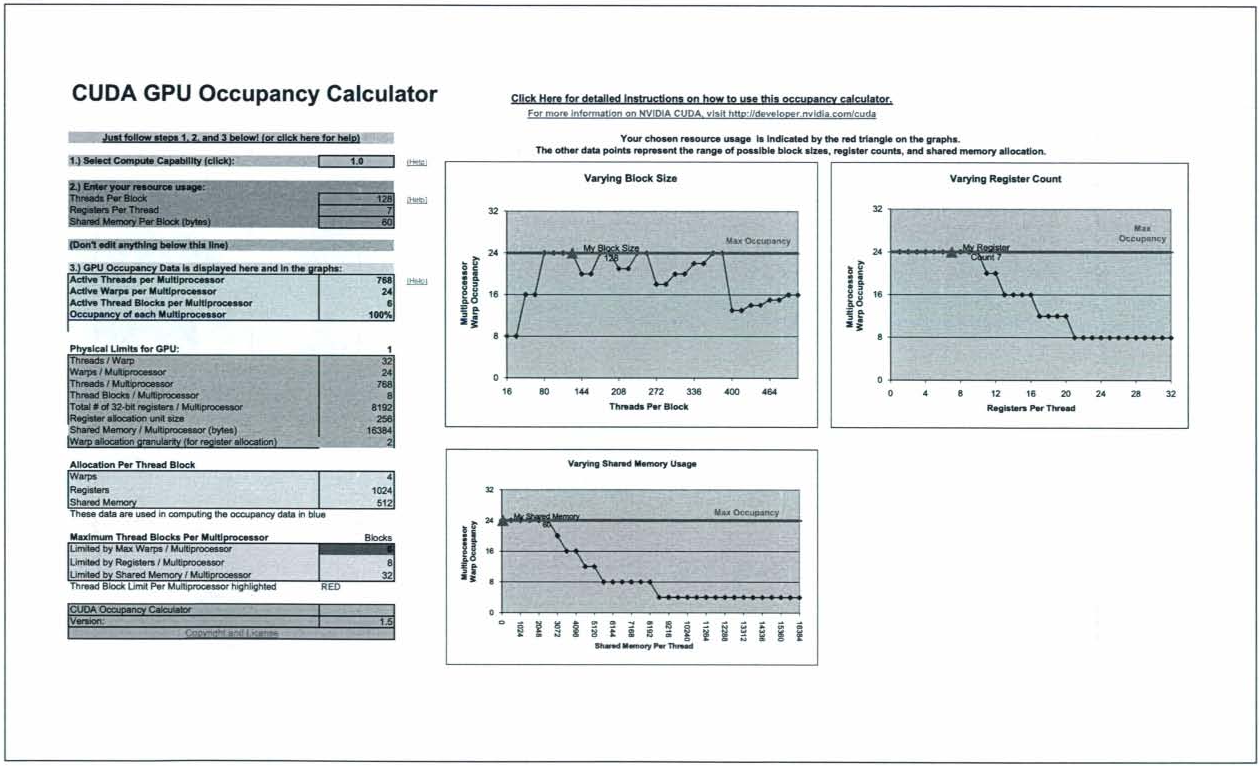

Figure B.2: get_drag_lift kernel occupancy data

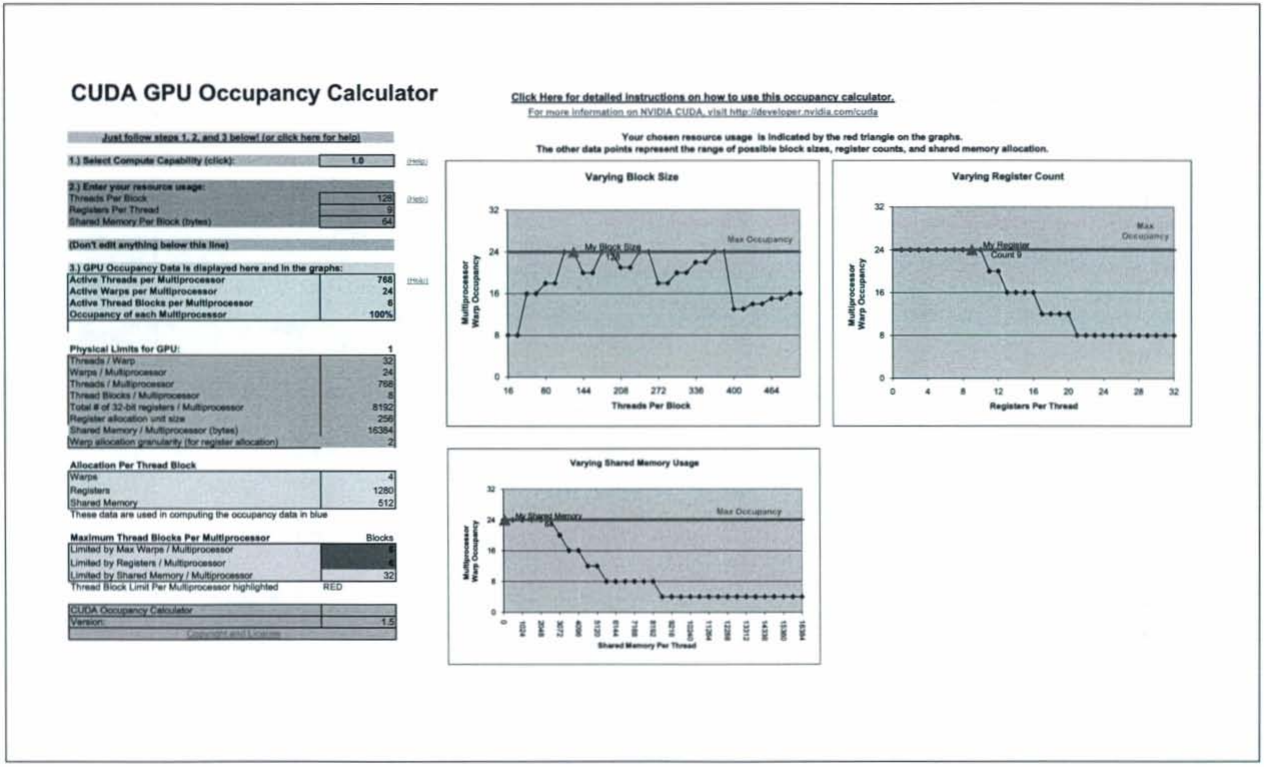

Figure B.3: setBoundaryValues_kernel kernel occupancy data

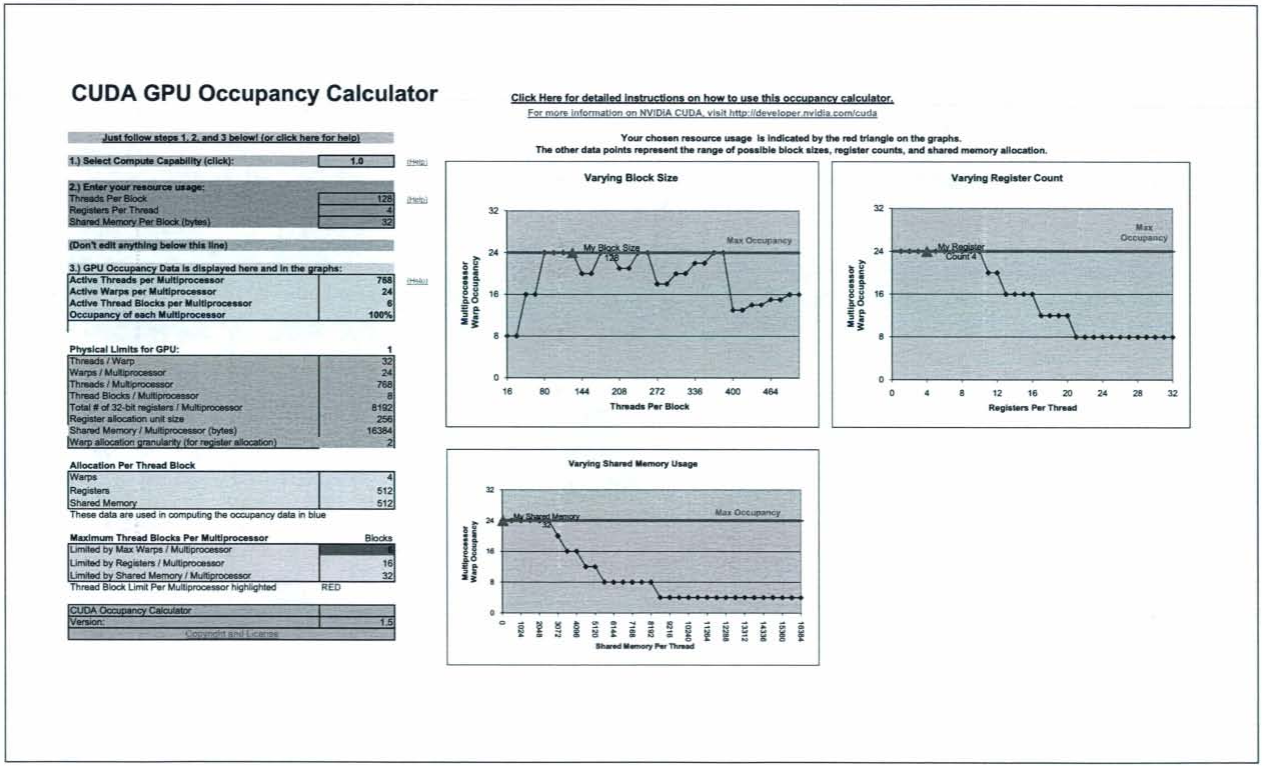

Figure B.4: reduce kernel occupancy data

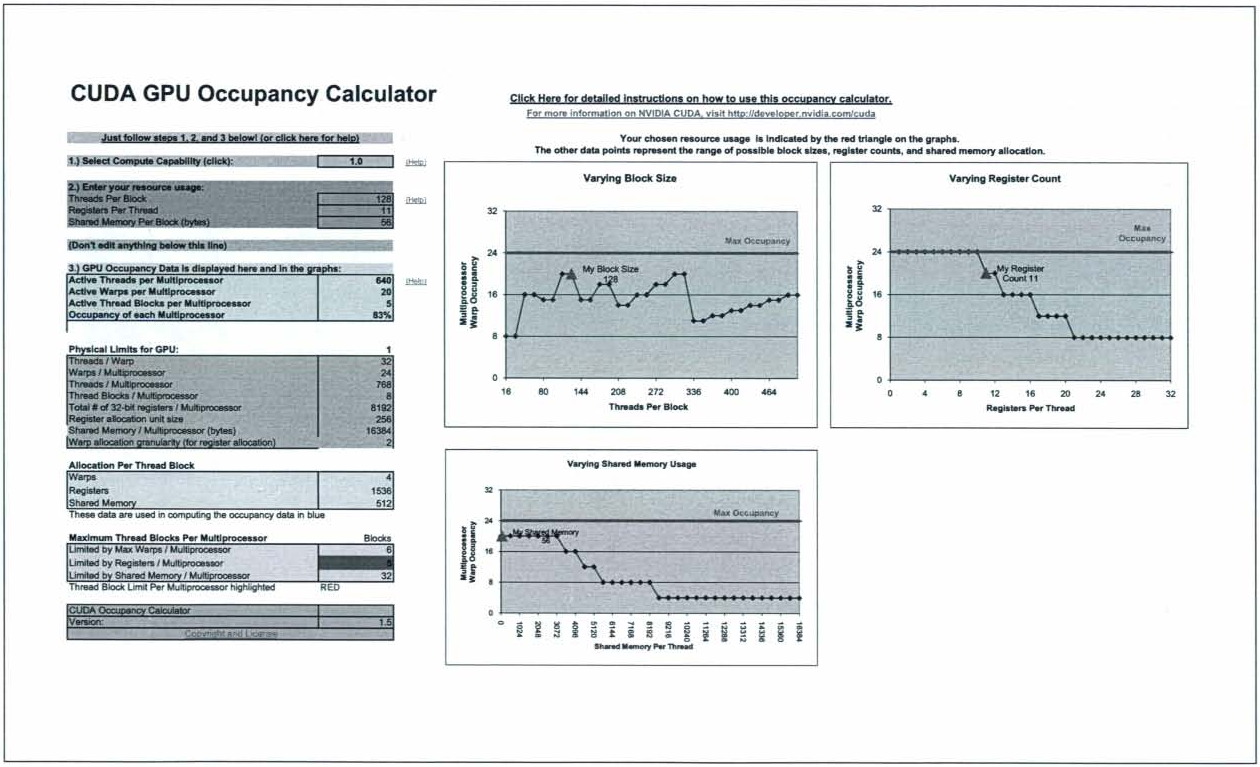

Figure B.5: applyCorrections_kernel kernel occupancy data

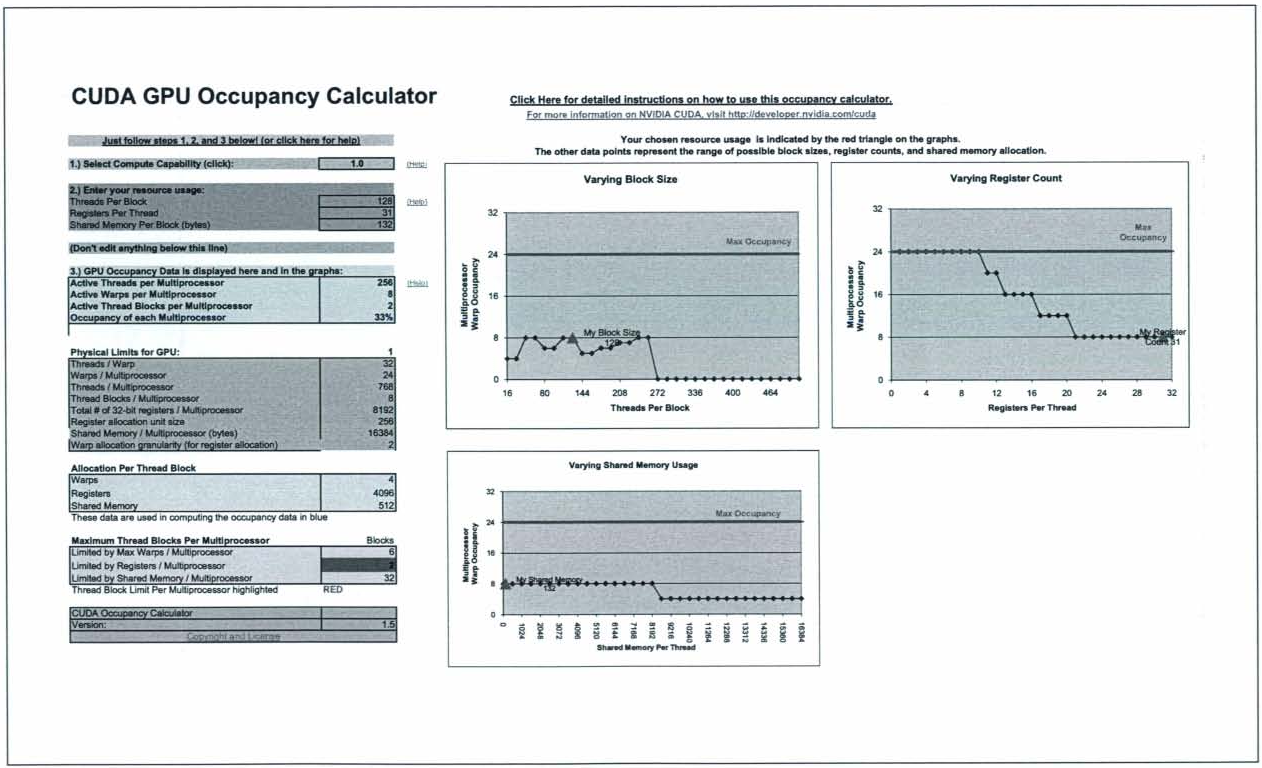

Figure B.6: constructCoefficients_uv kernel occupancy data

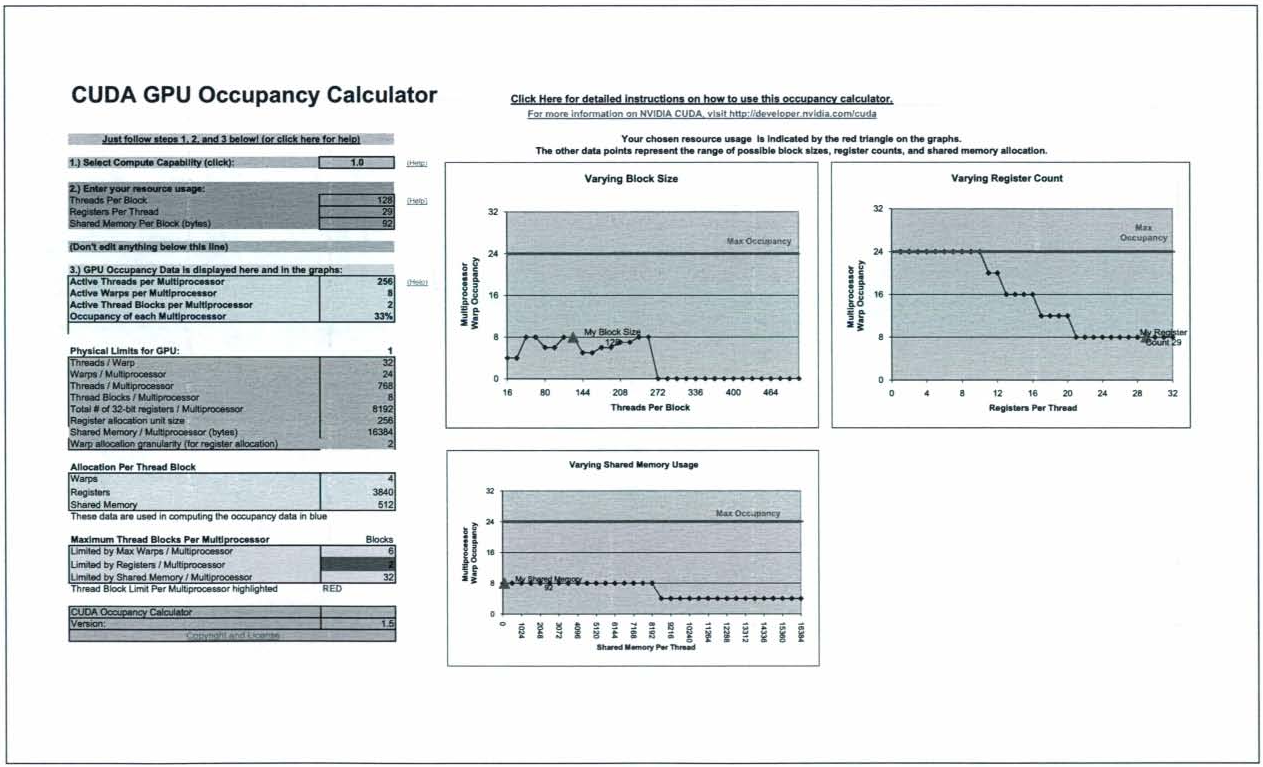

Figure B.7: constructCoefficients_pc kernel occupancy data

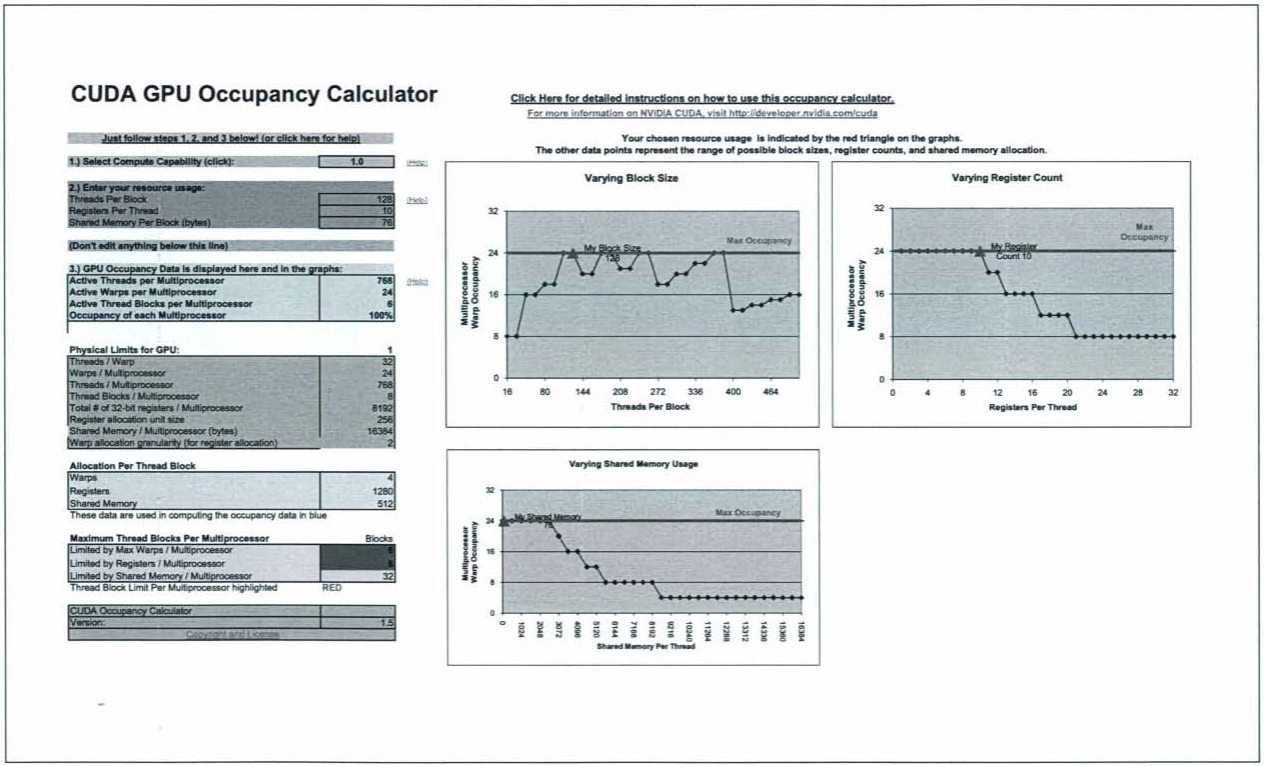

Figure B.8: apply_piso_corrections kernel occupancy data

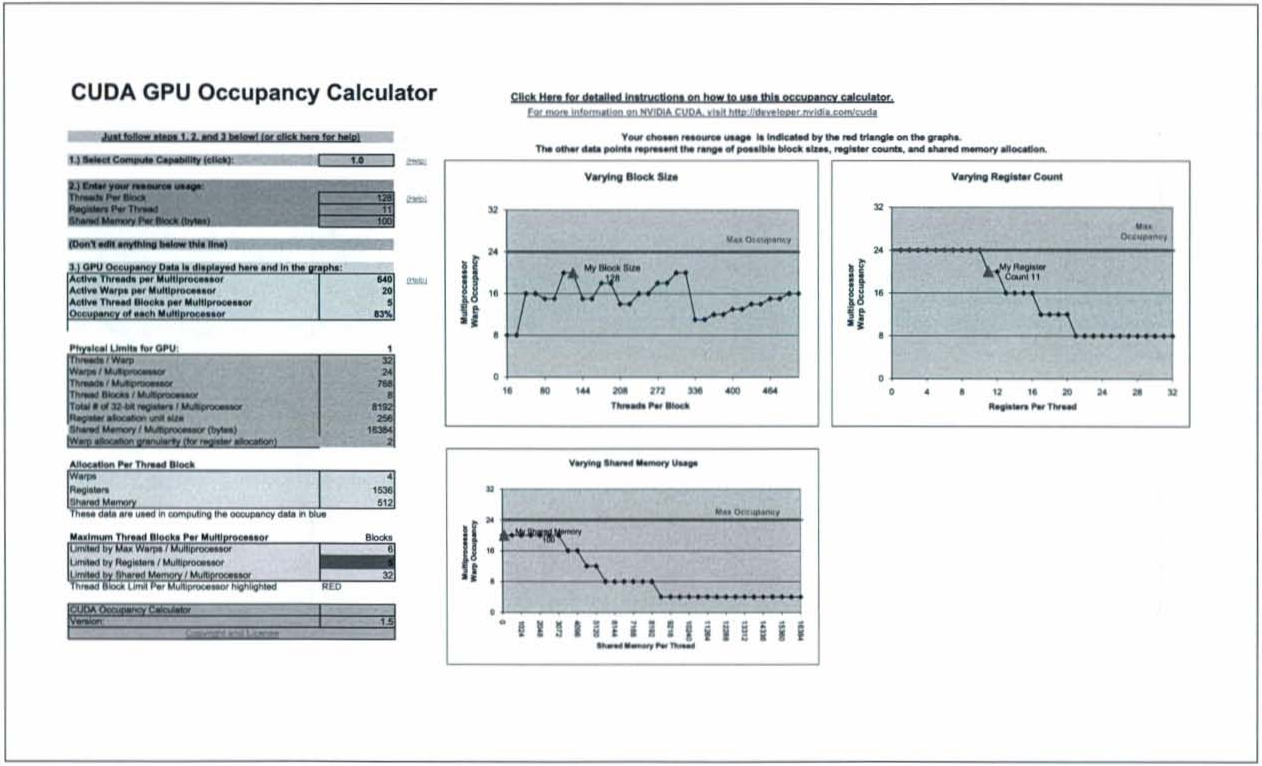

Figure B.9: calculate_uc_vc kernel occupancy data

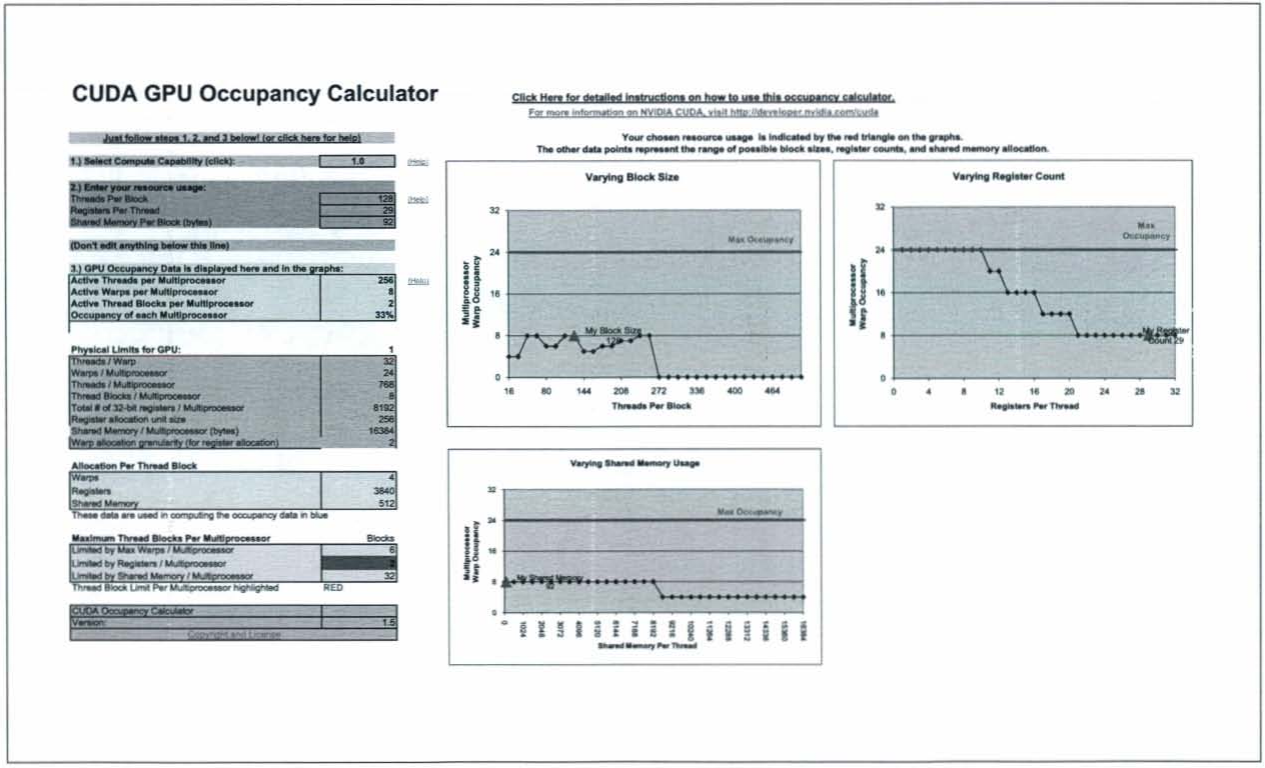

Figure B.10: constructCoefficients_pcc kernel occupancy data