2010

Graphics hardware acceleration for rotorcraft simulations

Mark William Lohry
Iowa State University

Graphics hardware acceleration for rotorcraft simulations

by

Mark William Lohry

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Aerospace Engineering

Program of Study Committee:
R. Ganesh Rajagopalan, Major Professor
Thomas Rudolphi
Z.J. Wang

Iowa State University

Ames, Iowa

2010

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I offer my sincerest gratitude to Ganesh Rajagopalan for his patience and selflessness that made this and future work possible, for which I am greatly indebted. I would also like to thank my loving family, and to my friends, schoolmates, and coworkers that made it enjoyable. I'd like to thank my committee members and all the faculty of the Aerospace Engineering department who have been inspirational in their humbling mix of intelligence and approachability. Lastly, I'd like to thank the whole of Iowa State University, which has educated dozens of my family members across several generations.

This work is partially funded by Penn State University VLRCOE vertical lift research center of excellence.

ABSTRACT

Interest in scientific programming using graphics processing units (GPUs) has exploded in recent years. The advent of NVIDIA's CUDA programming language in early 2007 enabled GPU acceleration of numerical software to become mainstream. Relative to central processing units (CPUs), these devices have extremely high floating point operation capability and memory bandwidth. Combined with relatively low cost, they are attractive alternatives to more expensive traditional supercomputers.

Porting existing computational fluid dynamics methods to the new hardware is not always straightforward. Modern GPUs are massively parallel, some consisting of over 400 processors, utilizing a unique hierarchy of computational units and memory management. Fully exploiting this architecture for CFD solvers requires the development of new algorithms tailored to the devices. To that end, this work presents a solution method for the Navier-Stokes equations using the SIMPLER algorithm on structured Cartesian grids. A block-iterative scheme with a parallel recursive tridiagonal solver is used for the discretized equations, giving considerable performance advantages over prior point-iterative implementations. Using a $200 GPU in a standard workstation, accelerations of over 20× are observed compared to a serial CPU implementation for rotorcraft simulations.

CHAPTER 1 OVERVIEW

The field of computational fluid dynamics has been closely linked to the advancements of computer technology since the beginning of the digital age. Problems related to CFD were a primary driving force in the early development of computers in the 1930s and 1940s, when military concerns demanded accurate computation of artillery ballistic trajectories. Iowa State's own Atanasoff-Berry Computer, the world's first digital computer, was designed in 1937 specifically for the rapid solution of linear systems of equations, the same core problem that motivates this work over 70 years later.

While numerical solutions to the full Navier-Stokes equations were always of great interest, the necessary computational power for any practical application was far beyond the reach of the early pioneers of CFD. One of the first three-dimensional CFD calculations ever performed was published by Hess and Smith [1] of Douglas Aircraft Co. in 1962. Restricted to non-lifting potential flow about a body, these calculations were performed on some of the most powerful computers of the time, an IBM 704 and later IBM 7090. A converged solution for 300 body panels took approximately 90 minutes on the IBM 7090, which sold for $2,900,000 in 1960 (over $20 million in 2010 dollars), a computation which can be completed in a fraction of a second on a modern desktop computer.

As computational power and availability increased and corresponding costs decreased, solutions to wider subsets of the Navier-Stokes equations became feasible for engineering design. The CFD advances in the past decades cannot be done justice in this paper, but some important milestones bear mentioning. Jameson and Caughey's FLO22 code was capable of solving the three-dimensional transonic full potential equations in 1975, the emergence of several Euler equation solvers in the early 1980s, and eventually full Navier-Stokes solvers such as the

successful NASA-funded CFL3D in the mid-1980s. Despite the exponential increase in both computational power and relevant knowledge, CFD applications remain at the forefront of computational demands, with many of the fastest supercomputers in the world dedicated to the field.

1.1 Modern computing hardware

Until very recently, increases in computational performance had generally been provided by a continuing trend of faster clock speeds. For the last 10 years however, Moore's law has been maintained not by increasing the speed of a single processor, but by increasing the number of processors on a given CPU, with clock speeds generally falling between 2 and 3GHz on typical consumer CPUs during this time. Physical limitations such as heat dissipation of a large amount of power in a small area have been the primary reasons for this clock speed stagnation, and increasingly parallel CPUs are the primary means of computational speed advancement for the forseeable future. Multi-core CPUs for personal desktop use were first introduced in 2005 by manufacturer AMD, and 5 years later, dual-core, quad-core, and even oct-core CPUs are commonplace.

This trend away from serial computational performance and onto parallel performance represents a considerable paradigm shift in all areas of software development, and CFD is no exception. Parallel computing has been commonplace during the history of CFD, with work being performed on such famous vector supercomputers as the 256-processor 16MHz ILLIAC IV, first used in 1976, and the 80MHz, 12-unit Cray-1 soon after. These early vector computers relied on a "Single-Instruction Multiple-Data," or SIMD, method of computation, where all processors executed the same instruction on different data sets. These vector computers were gradually made obsolete by the increasingly powerful and affordable consumer-grade CPUs in development , such as Intel's 8086 in 1978, which remains the basic architecture for the vast majority of desktop CPUs in use today. Parallel computing did not disappear with the vector computers, but instead transitioned into supercomputers constructed of large numbers commodity CPUs connected by various implementation-specific means. Although these com-

puters are still often used in an SIMD method, they can also function as "Multiple-Instruction Multiple-Data," or MIMD, as each CPU has it's own instruction units. MIMD still describes the present multi-core processor architectures in today's desktop computers, in which each processor is capable of operating independently as well as operating cooperatively. As computing hardware evolves, so too must the approach to software design across all fields, including the algorithms used in CFD.

1.2 Graphics processing units

During the 1990s, hardware specially built for graphics rendering known as graphics processing units (GPUs) began to emerge as dedicated devices separated from the CPU. The massive growth of the video game industry, which now rivals the Hollywood movie industry in size, spurred on advances in GPU technology that has outpaced CPU development in many ways. These chips have been point-designed for the calculations involved in rendering 3-dimensional graphics such as mapping 2-dimensional textures onto 3-dimensional geometries, operations which are inherently SIMD in scope. Due to the nature of the computational demands of graphics rendering, GPUs evolved into highly parallel devices with simple instruction units, similar to the vector computers of the 1970s in many respects.

In the early 2000s, researchers began to exploit GPUs for general-purpose computations unrelated to computer graphics, initiating the now rapidly expanding field. In part due to the simplicity of the instruction units, present-day GPUs are capable of vastly outperforming CPUs in terms of pure floating point performance (FLOPS), with recently released graphics cards such as NVIDIA's GTX480 being comprised of 480 processors operating at 1.4GHz. By comparison, CPUs of similar cost at the time of writing consist of 4 processors operating at 3.0GHz. Harnessing this newfound computational horsepower for CFD, however, requires a considerable departure from traditional software design. To that end, this work describes a new approach to a classical CFD solution algorithm which can leverage GPU computing in order to reduce run times by more than an order of magnitude.

1.3 Previous work

As GPU computing is still in it's infancy, there are few published examples of CFD work in the field, most of which have appeared in the past two years. Some notable work includes an early example of a structured 3D Euler solvers by Brandvik et al and Elsen et al [2, 3], unstructured solvers by Corrigan et al [4], and other related work found in [5–15]. Parallel CFD computing on CPUs is well developed by comparison, and important ideas from [16–22] provide some insight into possible approaches for GPUs. It is important to note that traditional algorithms for parallel CPU computing are not in general the most efficient approaches for the considerably different architecture of GPUs. The present work improves on previous work by implementing block-iterative solution methods for structured grids, providing faster solution convergence than the point-iterative methods previously cited.

CHAPTER 2 SIMPLER METHOD

The SIMPLER method of Patankar [23] is used as the underlying method of solving the discretized Navier-Stokes equations in this work. While the GPU-specific methods described later remain applicable to most any CFD algorithm, they will generally be described in the context of a 3-dimensional flow solver using this well-known pressure-based scheme.

2.1 General differential equation

Patankar presents the method in terms of a general differential equation

$$\frac{\partial}{\partial t}(\rho\phi) + \nabla \cdot (\rho\overrightarrow{V}\phi) = \nabla \cdot (\Gamma\nabla\phi) + S \tag{2.1}$$

where ϕ is the dependent variable, Γ is a diffusion coefficient, and S is a source term. In this form, the terms of 2.1 represent unsteady, convection, diffusion, and source terms, respectively. The elegance of this method lies in the ability to choose appropriate values of ϕ, Γ, and S to recover relevant equations; replacing ϕ with the scalar 1 or velocity vector $\overrightarrow{u}$ and corresponding terms for Γ and S leads to the continuity and momentum conservation equations, respectively. By choosing a general differential equation of this form, a method for discretizing and solving it would lead directly to a method for the original physical equations of interest.

Temporarily dropping the unsteady term and expanding to a 3-dimensional Cartesian coordinate system gives:

$$\frac{\partial}{\partial x}(\rho u\phi) + \frac{\partial}{\partial y}(\rho v\phi) + \frac{\partial}{\partial z}(\rho w\phi) = \frac{\partial}{\partial x}(\Gamma\frac{\partial\phi}{\partial x}) + \frac{\partial}{\partial y}(\Gamma\frac{\partial\phi}{\partial y}) + \frac{\partial}{\partial z}(\Gamma\frac{\partial\phi}{\partial z}) + S \tag{2.2}$$

and rearranging by derivative direction:

$$\frac{\partial}{\partial x}(\rho u\phi - \Gamma\frac{\partial\phi}{\partial x}) + \frac{\partial}{\partial y}(\rho v\phi - \Gamma\frac{\partial\phi}{\partial xy}) + \frac{\partial}{\partial z}(\rho w\phi - \Gamma\frac{\partial\phi}{\partial z}) = S \tag{2.3}$$

the equation can be more clearly represented by defining the total convective and diffusive fluxes:

$$J_x \equiv \rho u\phi - \Gamma\frac{\partial\phi}{\partial x} \tag{2.4}$$

$$J_y \equiv \rho v\phi - \Gamma\frac{\partial\phi}{\partial y} \tag{2.5}$$

$$J_z \equiv \rho v\phi - \Gamma\frac{\partial\phi}{\partial z} \tag{2.6}$$

simplifying the general differential equation into

$$\frac{\partial J_x}{\partial x} + \frac{\partial J_y}{\partial y} + \frac{\partial J_z}{\partial z} = S \tag{2.7}$$

Integrating over a 3-dimensional control volume

$$\int_V \left(\frac{\partial J_x}{\partial x} + \frac{\partial J_y}{\partial y} + \frac{\partial J_z}{\partial z}\right)\, dV = \int_V S\, dV \tag{2.8}$$

gives

$$J_e - J_w + J_n - J_s + J_t - J_b = S \cdot V \tag{2.9}$$

$$J_e \equiv (J_x)(\Delta y)_e(\Delta z)_e \tag{2.10}$$

$$J_w \equiv (J_x)(\Delta y)_w(\Delta z)_w \tag{2.11}$$

$$J_n \equiv (J_y)(\Delta x)_n(\Delta z)_n \tag{2.12}$$

$$J_s \equiv (J_y)(\Delta x)_s(\Delta z)_s \tag{2.13}$$

$$J_t \equiv (J_z)(\Delta x)_t(\Delta y)_t \tag{2.14}$$

$$J_b \equiv (J_z)(\Delta x)_b(\Delta y)_b \tag{2.15}$$

where subscripts e, w, n, s, t, b represent quantities computed at the six control volume faces, termed *east, west, north, south, top,* and *bottom* for convenience, corresponding to $\pm i, j, k$ directions.

Invoking the continuity equation

$$\frac{\partial}{\partial x}(\rho u) + \frac{\partial}{\partial y}(\rho v) + \frac{\partial}{\partial z}(\rho w) = 0 \tag{2.16}$$

and integrating in the same manner gives:

$$F_e - F_w + F_n - F_s + F_t - F_b = 0 \tag{2.17}$$

where F is the mass flux across the control volume face. Multiplying 2.17 by ϕ_P (subscript P indicating value at volume center) and subtracting from 2.9 gives:

$$(J_e - F_e\phi_P) + (J_w - F_w\phi_P) \tag{2.18}$$

$$+(J_n - F_n\phi_P) + (J_s - F_s\phi_P) \tag{2.19}$$

$$+(J_t - F_t\phi_P) + (J_b - F_b\phi_P) = (S_C + S_P\phi_P)\Delta x\Delta y\Delta z \tag{2.20}$$

where the source term S has been linearized into $S = S_C + S_P\phi_P$ to account for the source term's dependence on the dependent variable ϕ. This is rearranged into the final discretized equation

$$\begin{aligned} a_P\phi_P &= a_E\phi_E + a_W\phi_W + a_N\phi_N + a_S\phi_s + a_T\phi_T + a_B\phi_B + b \\ &= \sum a_{nb}\phi_{nb} + b \end{aligned} \tag{2.21}$$

where

$$a_E \equiv D_e A(|P_e|) + [\![-F_e, 0]\!] \tag{2.22}$$

$$a_W \equiv D_w A(|P_w|) + [\![F_w, 0]\!] \tag{2.23}$$

$$a_N \equiv D_n A(|P_n|) + [\![-F_n, 0]\!] \tag{2.24}$$

$$a_S \equiv D_s A(|P_s|) + [\![F_s, 0]\!] \tag{2.25}$$

$$a_T \equiv D_t A(|P_t|) + [\![-F_t, 0]\!] \tag{2.26}$$

$$a_B \equiv D_b A(|P_b|) + [\![F_b, 0]\!] \tag{2.27}$$

$$b \equiv S_C \Delta x \Delta y \Delta z + a_P^0 \phi_P^0 \tag{2.28}$$

$$a_P \equiv a_E + a_W + a_N + a_S + a_T + a_B + a_P^0 - S_P \Delta x \Delta y \Delta z \tag{2.29}$$

and the diffusion terms are written as

$$D_e \equiv \frac{\Gamma_e \Delta y \Delta z}{(\delta x)_e} \tag{2.30}$$

and P represents the Peclet number as

$$P_e \equiv \frac{F_e}{D_e} \tag{2.31}$$

and

$$A(|P|) \equiv [\![0, (1 - 0.1|P|)^5]\!] \tag{2.32}$$

is the power-law scheme, and the notation $[\![x, y]\!]$ denotes the maximum value of two terms x and y.

2.2 The SIMPLER algorithm

In the present work, three-dimensional incompressible flow is considered, requiring the solution of the mass and momentum equations in order to acquire the scalar pressure field and three components of momentum. Due to the coupled nature of these quantities, an iterative procedure is needed. The Navier-Stokes equations are formulated in terms of the general

differential equation described above. By setting $\phi = u, v, w$, $\Gamma = \mu$, and an appropriate source term, the momentum equations are recovered.

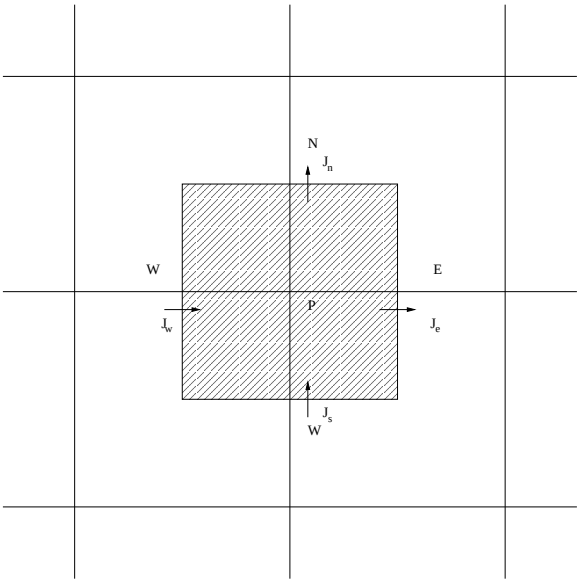

Figure 2.1 Control volume in two dimensions.

In the staggered grid formation, pressure is calculated at cell centers, while velocities are treated at the cell faces. The momentum equation for the u component of the velocity is written as

$$u_e = \frac{\sum a_{nb} u_{nb} + b}{a_e} + d_e(p_P - p_E) \tag{2.33}$$

and a pseudovelocity $\hat{u}_e$ defined such that

$$u_e = \hat{u}_e + d_e(p_P - p_E) \tag{2.34}$$

and similarly for the other velocity components. Combining these equations yields an equation for pressure

$$a_P p_P = a_E p_E + a_W p_W + a_N p_N + a_S p_s + a_B p_B + a_T p_T + b \tag{2.35}$$

where the source term b is

$$b = \frac{(\rho_P^0 - \rho_P)\Delta x \Delta y \Delta z}{\Delta t} + [(\rho\hat{u})_w - (\rho\hat{u})_e]\Delta y \Delta z + [(\rho\hat{v})_s - (\rho\hat{v})_n]\Delta x \Delta z + [(\rho\hat{w})_b - (\rho\hat{w})_t]\Delta x \Delta y \tag{2.36}$$

The SIMPLER algorithm consists of the following steps:

1. Calculate $\hat{u}$, $\hat{v}$, $\hat{w}$ in equation 2.34 by computing the coefficients from the previous iteration.
2. Calculate the coefficients for the pressure equation 2.35.
3. Solve this system of equations for the pressure field p.
4. Using this updated pressure field, solve the momentum equations.
5. Calculate the source term b and solve the pressure equation using the new velocity values.
6. Utilizing this pressure correction p' in place of p, correct the velocity field with equation 2.33.
7. Repeat the process until convergence is reached.

The solution to the pressure and velocity equation systems that arise in steps 3, 4, and 5 represents the vast majority of computational effort required in the algorithm. Methods for the solution of these linear systems is described in the following chapters. Complete details of the SIMPLER method can be found in Patankar [23].

CHAPTER 3 PARALLEL SOLUTION METHODS FOR LINEAR SYSTEMS

3.1 Schwarz domain decomposition

The alternating Schwarz method [24,24–29] is a robust, straightforward method for solving boundary value problems on multiple domains. It is an overlapping method, where the subdomains overlap each other in order to pass information, as opposed to a substructuring method (i.e., Schur complement) that directly solves the sparse matrix. The algorithm for solving an unknown u as in Figure 3.1 is as follows:

1. Pass u_2^{n-1} at Γ_1 to Ω_1
2. Solve Ω_1 with the u_2^{n-1} guess as the Dirichlet boundary condition on Γ_1, and regular boundaries on $\partial\Omega_1$
3. Pass u_1^n at Γ_2 to Ω_2
4. Solve Ω_2 with u_1^n as Dirichlet boundary condition on Γ_2, and regular boundaries on $\partial\Omega_2$
5. Repeat to convergence

Developed by Hermann Schwarz in the 19th century, this method represents an analytical solution to the original elliptic boundary value problem. This provides some benefits that cannot be found in the parallelization methods which are drawn strictly from consideration of a numerical solution. In the Schwarz method, it is unnecessary for the overlapping grids to match, or even to use the same solution algorithms, provided that the boundary condition exchange is performed with a proper interpolation.

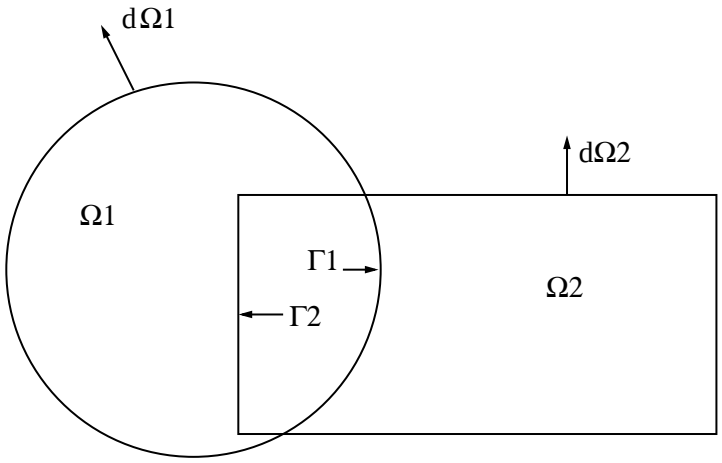

Figure 3.1 Alternating Schwarz method

3.1.1 Application to SIMPLER

In the SIMPLER method, this domain decomposition approach is easily implemented for overlapping matched grids, where data is transferred as in Figure 3.2, where one domain's interior cell data becomes the other domain's new boundary data as in the algorithm above. Incompressible flow solvers have been created using this method in Brakkee et al in [18,19,30].

Extension to two-dimensional domains, figure 3.3, and three-dimensions follows the same pattern in the case of two subdomains.

3.1.2 Extension to many subdomains

In the case of more than two overlapping subdomains, special care must be taken for exchange of data. In Figure 3.4, a 4-color scheme (red-black-green-orange) is used. After the iteration, data from the internal points of the red cells update the boundary conditions of non-red cells, and red cell boundary conditions are left unchanged. After the next iteration, data from black cells update non-black cell boundary conditions, etc., repeating the process. Extrapolation to three dimensions would use 8 colors for a structured cartesian grid.

This coloring scheme can be extended indefinitly to any number of subdomains as in Figure 3.5. It is important to note here that in the limit of many subdomains, the iterative nature of passing the artificial boundary values leads to a global convergence rate that approaches that

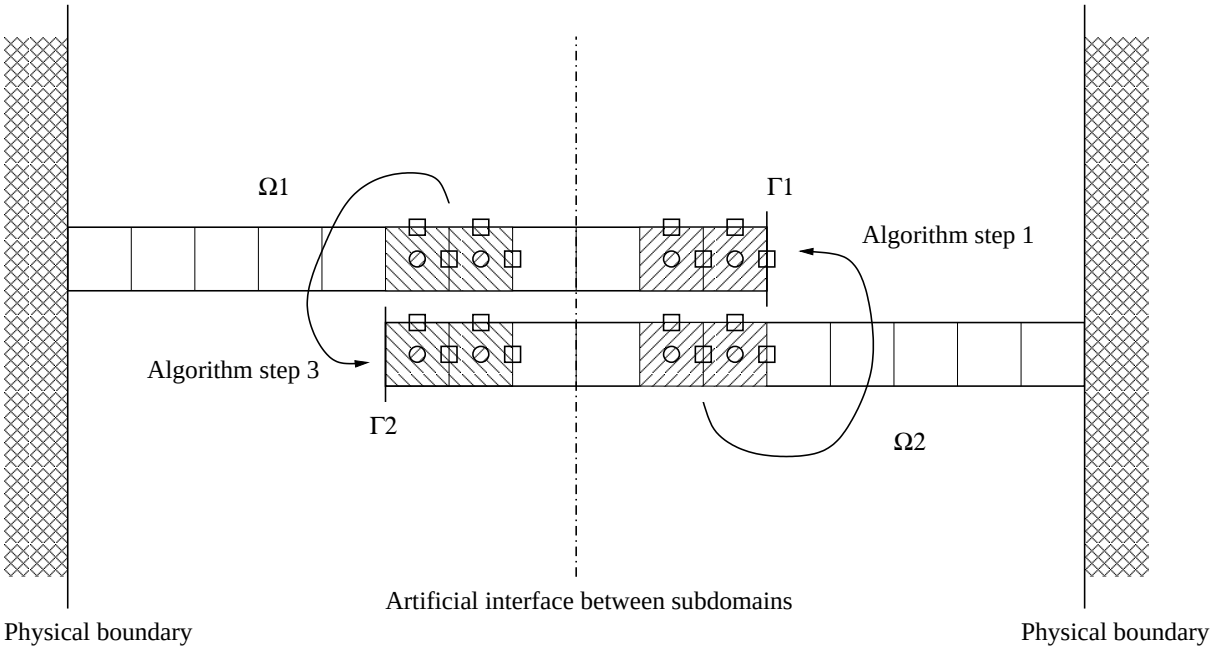

Figure 3.2 Schwarz method applied to 1D slice of staggered grid, 3 overlapping cells

of the point-iterative Gauss-Seidel method [25]. This attribute leads to a case of diminished returns in parallel efficiency when the subdomains are being solved using a more efficient line-implicit solution algorithm, as is the case here. Extension beyond approximately eight subdomains has been found to lead to poor global convergence behavior. This is not a strict limit however, as implementation of multigrid strategies can almost completely mitigate this behavior.

3.1.3 Domain decomposition results

Despite the limitations towards a large degree of parallelization in certain implementations, the analytical basis of the method leads to two important advantages in application. First, an existing computer program will generally require very little modification to include this parallelization approach. Since data exchange is performed only through boundary values, each subdomain can effectively be treated by an independent program, needing only a functionality to occasionally exchange this data. Secondly, the quantity of data that must be passed between

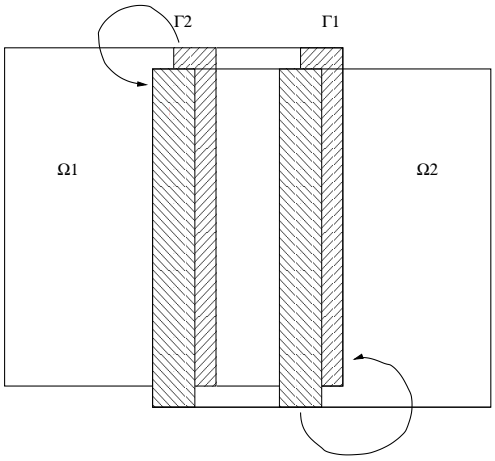

Figure 3.3 Data exchange between two 2D subdomains

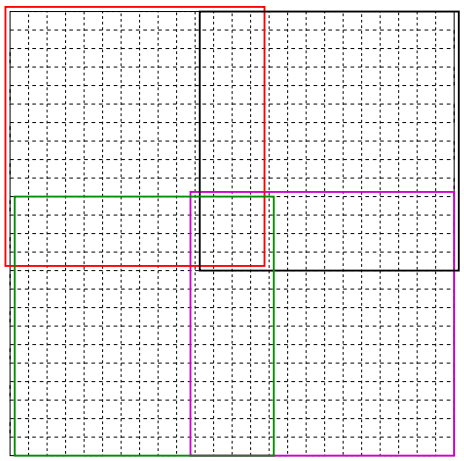

Figure 3.4 Coloring scheme with four overlapping subdomains.

processes is minimized, decreasing the "granularity" of parallel computation.

A 2-dimensional driven cavity is considered here, with results given for a 128×128 grid at $Re = 100$. Subdomain solutions are performed by an alternating line-iterative Gauss-Seidel method, utilizing the Thomas algorithm for solution of the tridiagonal systems that arise. The flowfield, seen at 3.6, consists of a "lid" at the top of a cavity moving to the right with unit velocity, and all other walls non-slip, creating the areas of recirculation seen in the lower left and right corners of the cavity. Velocity centerlines, Figures 3.8 and 3.7, validate that the original single-domain computation is successfully recovered in the decomposed domains.

There are several factors of note in Figure 3.9 showing residual history between the various

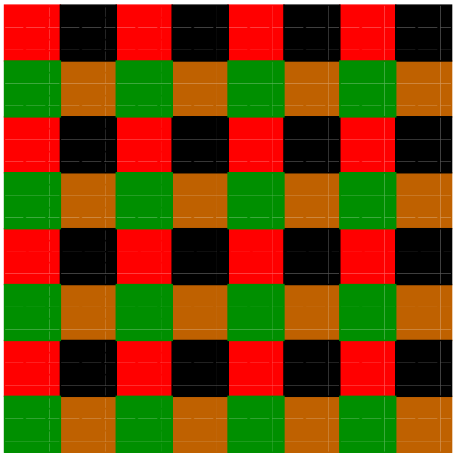

Figure 3.5 Coloring scheme for case of many subdomains.

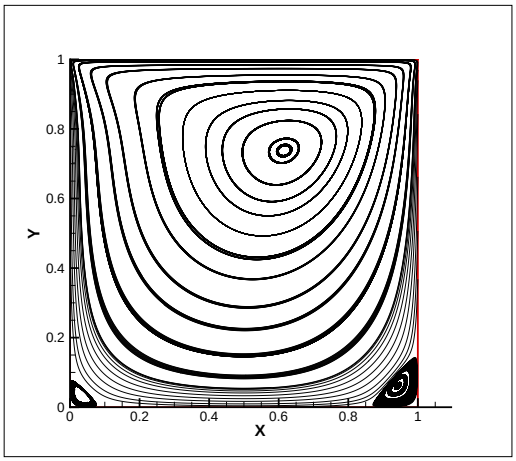

Figure 3.6 Streamlines at convergence, 128x128 grid, $Re = 100$

decompositions. First, the convergence rate on a per-iteration basis (not wall-clock time) is generally worse for the decomposed domains, which is to be expected in light of previous discussion. Additionally, increasing the number of overlapping cells between domains provides a better convergence rate at the expense of larger computational cost in the subdomains. This can be explained by the effect of one subdomain propagating deeper into another during boundary condition exchange. Also of note is the parallel efficiency observed in table 3.1, where increasing the number of subdomains is shown to require a larger number of iterations for a given convergence criteria. Parallel efficiency η is defined as the function of iteration time of the serial algorithm T_{ser}, parallel algorithm T_{par}, and number of processors N.

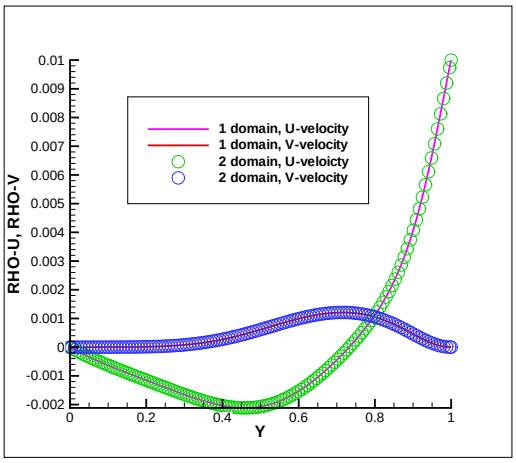

Figure 3.7 Velocity at y-centerline, 128x128 grid

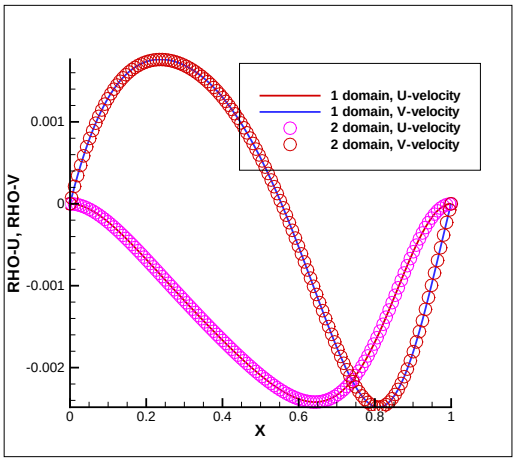

Figure 3.8 Velocity at x-centerline, 128x128 grid

$$\eta = \frac{T_{ser}}{NT_{par}} \tag{3.1}$$

3.2 Tridiagonal systems

In the discretized equation systems 2.21 for a structured grid, banded matrices naturally arise. Direct solution of these matrices is computationally prohibitive, so many iterative methods may be used for numerical solution. In this work, a line iterative method is used known as successive line over-relation, or SLOR, whereby unknowns along a solution line are treated

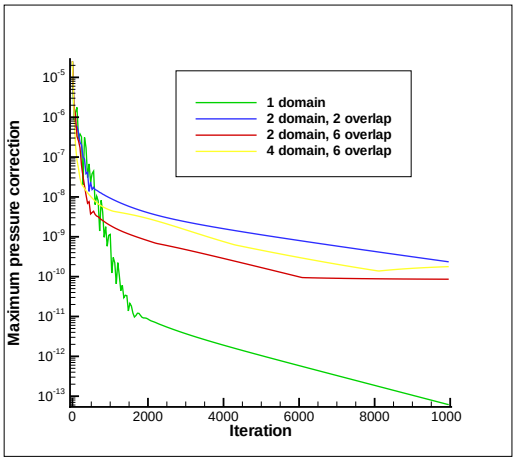

Figure 3.9 Residual history, 128x128 grid.

Table 3.1 Convergence time to 10^{-9} pressure correction L2 norm for single- vs multi-domain, 128x128 grid.

	Iterations	Real time (s)	Speed-up	Parallel efficiency
1 domain	232	11.95		
2 domain	269	6.56	1.82	91%
4 domain	276	3.36	3.55	89%

implicitly, while their neighbors are treated explicitly, resulting in a tridiagonal system of the form 3.2. Here, the unknowns along a line are given by x, the influence of the values of neighbors along the line are incorporated in a, b, and c, and source terms and off-line neighbors are incorporated in d. By solving along a line as in Figure 3.10 and sweeping in alternating directions (described in more detail in section 4.2.2), a solution can be determined with considerably more efficiency than the case of a point-iterative method. It is this method which was used for the solution of subdomains in the previous section.

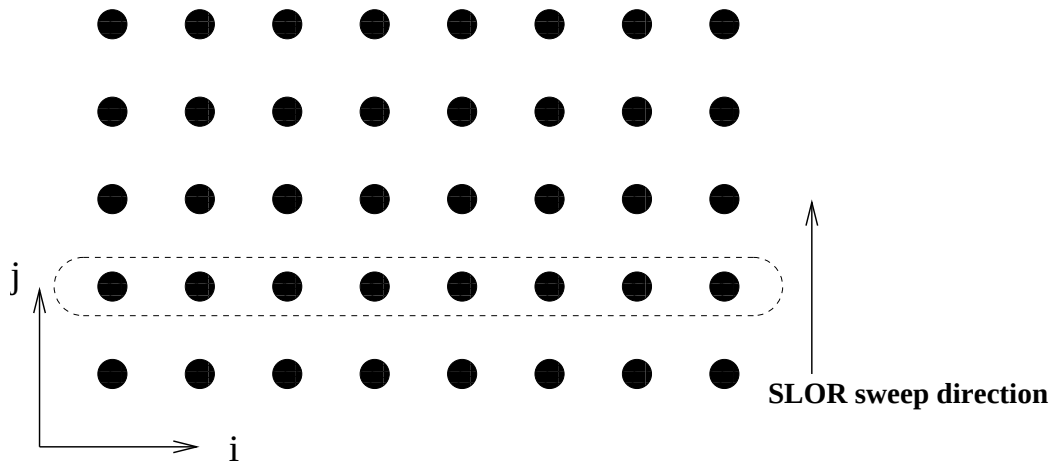

Figure 3.10 Successive line over-relaxation (SLOR.)

$$\begin{bmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & a_{n-1} & b_{n-1} & c_{n-1} & \\ & & & a_n & b_n & \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_n \end{pmatrix} \tag{3.2}$$

3.2.1 Parallel implemention of serial Thomas algorithm

The Thomas algorithm, also known as the tridiagonal matrix algorithm (TDMA), is a reduced form of Gaussian elimination for tridiagonal systems. Considering the form of equation 3.2, the method is comprised of two steps. First, a forward sweep:

$$c_i = \begin{cases} c_1/b_1 & \text{for } i = 1 \\ c_i/(b_i - c_{i-1}a_i) & \text{for } i = 2 \rightarrow n \end{cases} \tag{3.3}$$

$$d_i = \begin{cases} d_1/b_1 & \text{for } i = 1 \\ (d_i - d_{i-1}a_i)/(b_i - c_{i-1}a_i) & \text{for } i = 2 \rightarrow n \end{cases} \tag{3.4}$$

followed by back substitution

$$x_i = \begin{cases} d_i & \text{for } i = n \\ d_i - c_{i-1}a_i & \text{for } i = n-1 \rightarrow 2 \end{cases} \tag{3.5}$$

This is an inherently serial procedure, but it can be used in a parallel environment considering that a given SLOR sweep can typically contain over 100 independent tridiagonal systems. In application, however, this creates many fine-grained parallel tasks which may not be suitable for the hardware in use. The primary advantage of this approach is the simplicity of implementation on a shared-memory computer; through the addition of a handful of OpenMP directives for the parallel loop structure, an appreciable acceleration can be achieved with little effort.

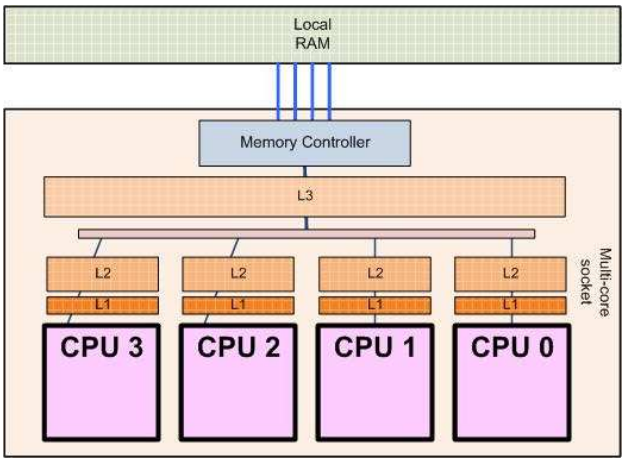

Figure 3.11 Quad-core processor diagram, *Intel*.

Parallel efficiency is inhibited with this naive approach due to the data dependency issues that arise in the collection of coefficients for the tridiagonal system. For a computational thread solving for unknowns along a gridline, it must access the neighboring values as well, which are to be incorporated in the right hand side of the system. If those neighboring values are in the process of being updated by a different thread, it will be subject to a "memory lock" which prevents what would be an indeterminate behavior where one process attempts to write to a data location while another attempts to read from it. Increasing the grid size (and thus the amount of computation necessary for solution of a tridiagonal system) has been found to alleviate some of this efficiency inhibition on the modern multi-core CPU architectures as in Figure 3.11. This is due to a decrease in the granulairty of the problem, reducing the memory conflicts.

Table 3.2 Parallel efficiency for SIMPLER iterations of varying grid sizes.

Problem size	CPU number	Time per iteration (s)	Speed-up	Parallel efficiency
$30 \times 30 \times 30$	1	0.0915	–	–
	2	0.0619	1.48	74%
	4	0.0754	1.22	30%
$64 \times 64 \times 64$	1	3.34	–	–
	2	2.49	1.34	67%
	4	2.31	1.45	36%
$128 \times 128 \times 128$	1	36.0	–	–
	2	26.6	1.35	68%
	4	24.0	1.50	38%

Seen in table 3.2, the parallel performance of this approach is quite poor in comparison to the domain decomposition method. Despite this, it should not be totally disregarded, as its implementation is a near-trivial way to gain an appreciable amount of performance from a modern multi-core processor.

3.2.2 Parallel Cyclic Reduction

The parallel cyclic reduction (PCR) algorithm of Hockney [31] is a modification of the original cyclic reduction method for tridiagonal systems. PCR is highly amenable to GPU computing both for its regular memory access and its use of all available vector processors. Consider the three equations adjacent to the i^{th} row of the tridiagonal system:

$$\begin{aligned} a_{i-1}x_{i-2} + b_{i-1}x_{i-1} + c_{i-1}x_i &= d_{i-1} \\ a_i x_{i-1} + b_i x_i + c_i x_{i+1} &= d_i \\ a_{i+1}x_i + b_{i+1}x_{i+1} + c_{i+1}x_{i+2} &= d_{i+1} \end{aligned} \tag{3.6}$$

Multiplying the d_{i-1} equation by $\alpha_i = -a_i/b_{i-1}$ and the d_{i+1} equation by $\gamma_i = -c_i/b_{i+1}$ yields

$$\begin{array}{ccccccc}
\left(\frac{-a_i}{b_{i-1}}\right) a_{i-1}x_{i-2} & + & \left(\frac{-a_i}{b_{i-1}}\right) b_{i-1}x_{i-1} & + & \left(\frac{-a_i}{b_{i-1}}\right) c_{i-1}x_i & = & \left(\frac{-a_i}{b_{i-1}}\right) d_{i-1} \\
a_i x_{i-1} & + & b_i x_i & + & c_i x_{i+1} & = & d_i \\
\left(\frac{-c_i}{b_{i+1}}\right) a_{i+1}x_i & + & \left(\frac{-c_i}{b_{i+1}}\right) b_{i+1}x_{i+1} & + & \left(\frac{-c_i}{b_{i+1}}\right) c_{i+1}x_{i+2} & = & \left(\frac{-c_i}{b_{i+1}}\right) d_{i+1}
\end{array} \tag{3.7}$$

and summing eliminates the x_{i-1} and x_{i+1} terms, yields a modified set of equations

$$a'_i x_{i-2} + b'_i x_i + c'_i x_{i+2} = d'_i \tag{3.8}$$

where

$$\begin{aligned}
a'_i &= \alpha_i a_{i-1} \\
b'_i &= b_i + \alpha_i c_{i-1} + \gamma_i a_{i+1} \\
c'_i &= \gamma_i c_{i+1} \\
d'_i &= d_i + \alpha_i d_{i-1} + \gamma_i d_{i+1}
\end{aligned} \tag{3.9}$$

Transforming each equation results in a banded system of the form:

$$\begin{bmatrix}
b'_1 & 0 & c'_1 & & & & \\
0 & b'_2 & 0 & c'_2 & & & \\
a'_3 & 0 & b'_3 & 0 & c'_3 & & \\
 & \ddots & \ddots & \ddots & \ddots & \ddots & \\
 & & a'_{n-2} & 0 & b'_{n-2} & 0 & c'_{n-2} \\
 & & & a'_{n-1} & 0 & b'_{n-1} & 0 \\
 & & & & a'_n & 0 & b'_n
\end{bmatrix}
\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}
=
\begin{pmatrix} d'_1 \\ d'_2 \\ \vdots \\ \vdots \\ \vdots \\ d'_{n-1} \\ d'_n \end{pmatrix} \tag{3.10}$$

Reordering this system into even and odd coefficients yields

$$\begin{bmatrix} b'_1 & c'_1 & & & & & & & & \\ a'_3 & b'_3 & c'_3 & & & & & & & \\ & \ddots & \ddots & \ddots & & & & & & \\ & & a'_{n-3} & b'_{n-3} & c'_{n-3} & & & & & \\ & & & a'_{n-1} & b'_{n-1} & 0 & & & & \\ & & & & 0 & b'_2 & c'_2 & & & \\ & & & & & a'_4 & b'_4 & \ddots & & \\ & & & & & & \ddots & \ddots & \ddots & \\ & & & & & & & a'_{n-2} & b'_{n-2} & c'_{n-2} \\ & & & & & & & & a'_n & b'_n \end{bmatrix} \begin{pmatrix} x_1 \\ x_3 \\ \vdots \\ x_{n-3} \\ x_{n-1} \\ x_2 \\ x_4 \\ \vdots \\ x_{n-2} \\ x_n \end{pmatrix} = \begin{pmatrix} d'_1 \\ d'_3 \\ \vdots \\ d'_{n-3} \\ d'_{n-1} \\ d'_2 \\ d'_4 \\ \vdots \\ d'_{n-2} \\ d'_n \end{pmatrix} \tag{3.11}$$

Examining equation 3.11 reveals two important characteristics: the equations have been decoupled into two halves, and it still has a tridiagonal form which can be recursively treated with the original reduction formula. 3.10 shows that each reduction eliminates some off-diagonal elements while pushing others further from the main diagonal. By then reducing the system for levels $l = 1, 2, ..., \log_2 n - 1$, equation 3.9 can be expressed in terms of the reduction level by

$$\begin{aligned} \alpha_i^{[l]} &= -a_i^{[l-1]} / b_{i-2^{(l-1)}}^{[l-1]} \\ \gamma_i^{[l]} &= -c_i^{[l-1]} / b_{i+2^{(l-1)}}^{[l-1]} \\ a_i^{[l]} &= \alpha_i a_{i-2^{(l-1)}}^{[l-1]} \\ b_i^{[l]} &= b_i^{[l-1]} + \alpha_i c_{i-2^{(l-1)}}^{[l-1]} + \gamma_i a_{i+2^{(l-1)}}^{[l-1]} \\ c_i^{[l]} &= \gamma_i c_{i+2^{(l-1)}}^{[l-1]} \\ d_i^{[l]} &= d_i^{[l-1]} + \alpha_i d_{i-2^{(l-1)}}^{[l-1]} + \gamma_i d_{i+2^{(l-1)}}^{[l-1]} \end{aligned} \tag{3.12}$$

where the superscripts in brackets indicate reduction level, and superscripts in parentheses indicate integer exponents for determining the appropriate indices. After reducing to the final level $q = \log_2 n - 1$, the unknown values are obtained from

$$x_i = d_i^{[q]} / b_i^{[q]} \tag{3.13}$$

Following 3.12 verbatim results in accessing indices outside the range of the system. For example, an update to the location $i = 1$ at recursion level $l = 2$ requires information from element $i - 2^{(l-1)} = -1$, which does not fall in the original range of $i = 1 \leq n$. This is accounted for by noting that any system of equations can be extended by

$$\left.\begin{array}{rcl} a_i & = & 0 \\ b_i & = & 1 \\ c_i & = & 0 \\ d_i & = & 0 \end{array}\right\} \text{for } i < 1 \text{ and } i > n \tag{3.14}$$

which results in $x_i = 0$ for out-of-range elements, and doesn't affect the original system. 3.12 clearly requires $O(n)$ operations at each of $\log_2 n$ reductions, while 3.13 takes n operations, yielding a total algorithm cost of $O(n \log_2 n)$. While this behavior is worse than the $O(n)$ complexity of Gauss elimination, all steps exhibit n-parallelism, admitting solutions in times proportional to $\log_2 n$. The ability to scale to very large number of computational elements combined with the regular vector nature of the memory accesses at all steps in the algorithm make PCR the most attractive parallel tridiagonal solver for this work.

3.2.2.1 PCR Implementation in CUDA

The memory access patterns and scalability of parallel cyclic reduction provide a convenient mapping to the CUDA model of GPU programming [32]. While each GPU multiprocessor consists of 8 thread processors running concurrently, peak efficiency is achieved when each block uses in excess of 64 threads [33]. Equation 3.12 shows that for a given reduction level $[l]$, each coefficient for an equation i can be reduced independently based only on the previous reduction values at $[l - 1]$. The implication of this behavior is that every i^{th} equation of a system can be mapped to its own processing thread.

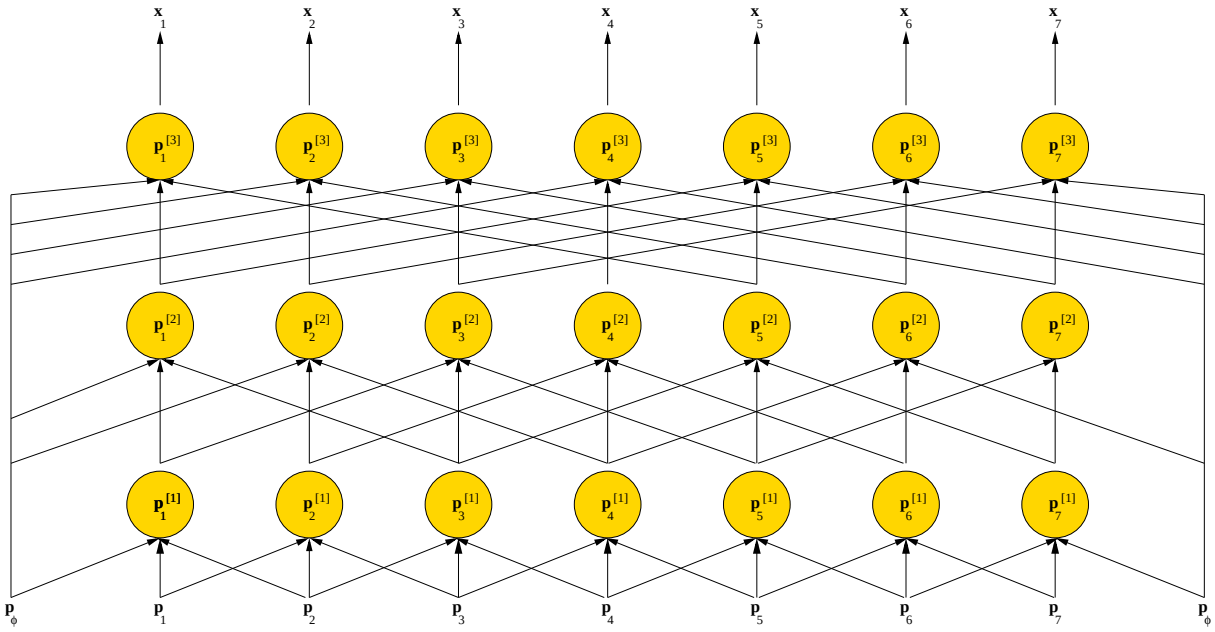

Figure 3.12 Communication pattern for parallel cyclic reduction, with each vertical path representing a single thread. p_i represents the coefficient vector $\{a_i, b_i, c_i, d_i\}$ and p_ϕ is the coefficient vector for ranges outside the equation system, given by 3.14.

By implementing a PCR kernel utilizing thread block dimensions equal to the number of unknowns in the system, code complexity is reduced; more importantly, a very high degree of parallelism can be expressed that provides exploitation of the GPU architecture. Since a typical CFD simulation requires structured grid dimensions resulting in tridiagonal systems of 100 or more, this is of great benefit to overall solution efficiency. Figure 3.12 shows the data dependencies for the solution algorithm.

CHAPTER 4 GPU PROGRAMMING FOR CFD

The computational power of modern graphics cards can be directly attributed to consumer demand for increasingly impressive visual effects in video games. In recent years, total revenue in the video game industry has rivaled that of the music and film industries. As a result of this market demand, intense competition between the two primary GPU manufacturers NVIDIA and ATI, and the fundamentally parallel data-intensive nature of graphics rendering, the increase in raw computational power of graphics cards has far outpaced the advances in the CPU market (Figures 4.1,4.2). The demands of projecting 3-dimensional digital environments onto 2-dimensional displays has led to GPUs becoming massively parallel computational devices with very high memory bandwidth [33].

The first applications of general-purpose computation on GPUs (GPGPU) have their roots in the OpenGL and Direct3D APIs, originally intended solely for graphics applications. Writing mathematical software for these APIs required a mapping of the mathematics of interest onto the language of computer graphics, which is considerably more time-consuming than the historical CPU approach. With the advent of Stanford's *BrookGPU* in the early 2000s, applications could be written in a C-like syntax that abstracted the graphics hardware, greatly expanding the accessibility of GPGPU programming to those without graphics-specific knowledge. In the past five years, NVIDIA's *CUDA*, Microsoft's *DirectCompute*, ATI/AMD's *Stream*, and the collaborative *OpenCL* frameworks have replaced BrookGPU as the tools of choice for exploiting graphics hardware. It is the author's opinion that NVIDIA's CUDA (*Compute Unified Device Architecture*) implementation is the most mature technology of these at the time of writing, and is used throughout the work presented.

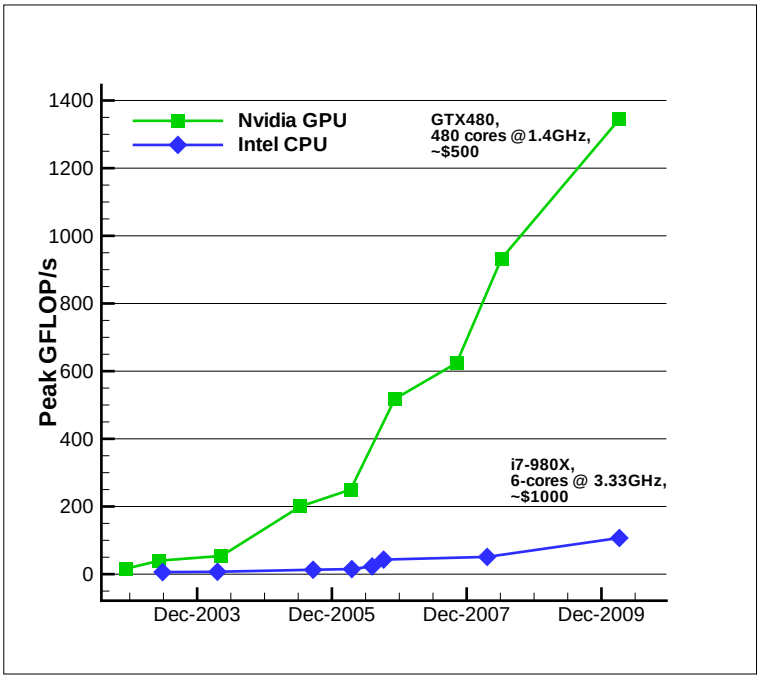

Figure 4.1 Peak floating point performance history of NVIDIA GPUs and Intel CPUs.

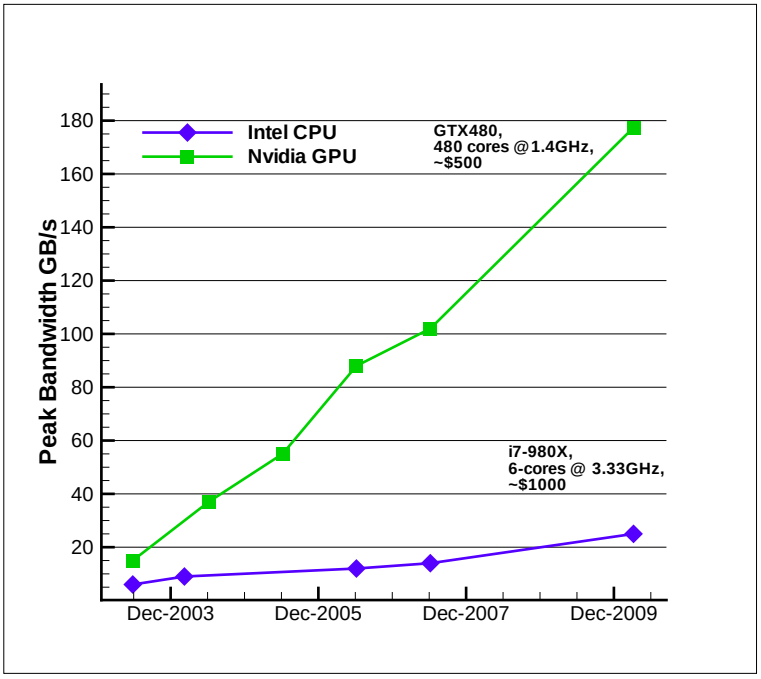

Figure 4.2 Peak memory bandwidth history of NVIDIA GPUs and Intel CPUs.

4.1 Programming model

All available GPGPU frameworks share some necessary similarities in order to treat GPU architectures as generic vector processors. In this section, the focus is on utilizing CUDA, but the higher level concepts carry over to other implementations.

4.1.1 Hardware overview

Writing effective software in CUDA C requires some knowledge of the underlying hardware's strengths, weaknesses, and strict limitations. While some functional knowledge of CPU architecture aids in the writing of efficient mathematical software in CPU-based languages such as C, C++, or Fortran, the massively parallel and close-to-hardware nature of GPU programming makes hardware familiarity a necessity. The focus here is on NVIDIA's GT200-series GPUs, originally released mid-2008. Older and newer architectures differ in technical details, but the general programming approach will remain valid for the forseeable future.

The GT200-series GPU consists of up to 30 multithreaded Streaming Multiprocessors (SMPs), each of which consist of 8 thread processors, for a total of 240 processors (also referred to as "CUDA cores".) Each multiprocessor has a low-latency local "shared" memory accessible by all of its thread processors, and all multi- and thread processors have access to a higher latency global "device" memory. The shared memory functions as a user-controlled cache, and its efficient utilization is essential. In order to address this architecture, CUDA programs are written using a Single-Instruction-Multiple-Thread (SIMT) programming model, functionally similar to the Single-Instruction-Multiple-Data (SIMD) approach utilized on the vector computers of past decades.

4.1.2 Software overview

CUDA programming introduces the *kernel* concept, which takes the form of a C function. Kernels are executed concurrently by the GPU's multiprocessors, which in turn utilize their parallel thread processors for computation. An abstraction of this parallelization is expressed

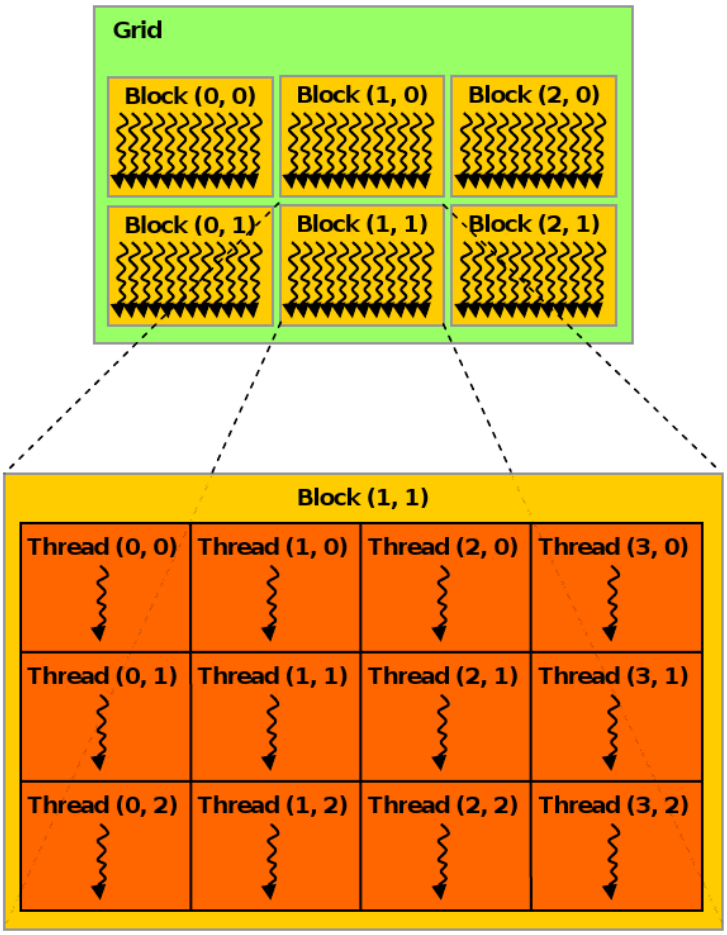

Figure 4.3 Grid of thread blocks, CUDA Programming Guide.

by the programmer when a kernel is invoked by specifying an hierarchy of *thread blocks* utilizing the `<<<...>>>` syntax.

A group of thread blocks, or just "blocks," is termed a *grid*. A grid is a 1- or 2-dimensional collection of blocks, with each block being executed by a single multiprocessor (Figure 4.3.) Each block within a grid is comprised of the same shape of a 1-, 2-, or 3-dimensional collection of threads, which are executed concurrently by the thread processors on the block's multiprocessor.

In the SIMT model, the same kernel is executed across every block when invoked. In order for this one kernel to operate on a large array of data simultaneously, every thread is aware of its block indices within the grid (`blockIdx.x, blockIdx.y`), and its thread indices

within the block (`threadIdx.x, threadIdx.y, threadIdx.z`) through the intrinsic variables in parantheses.

Listing 4.1 3-dimensional matrix addition in CUDA

```
__global__ void Add3DMatrix(float* A, float* B, float* C)
{
    int i = blockIdx.x;
    int j = blockIdx.y;
    int k = threadIdx.x;
    int idim = gridDim.x;
    int jdim = gridDim.y;
    int index = k * idim * jdim + j * idim + i;
    C[index] = A[index] + B[index];
}

int main()
{
    ...
    dim3 blocks( idim, jdim, 1 );
    dim3 threads( kdim, 1, 1 );
    Add3DMatrix<<< blocks, threads >>>( A, B, C );
}
```

In the above CUDA code snippet, two arrays `A` and `B` of size `float[idim*jdim*kdim]` are added together to form `C`. This one example incorporates a number of important attributes of general CUDA programming, as well as the specific methods used for structured grid CFD algorithms in this work. Through the use of the built-in variables, every thread within the kernel can execute the same instruction on different data.

4.1.3 Memory hierarchy

In addition to the computational hierarchy of grids and blocks, memory use must also be managed manually. The *device memory*, shown in figure 4.4, is the largest portion of memory available on the graphics card, and is globally accessible by all threads. This memory is typically allocated by the host through the use of `cudaMalloc` commands which allocate pieces of memory which can be associated with a pointer, similar to the traditional `malloc` of the C language. It is this memory space which stores all the grid and flowfield information relevant to the computations. The device used here has 896 MB of device memory, which corresponds to a gridsize of approximately 128^3 which can be stored entirely on the device

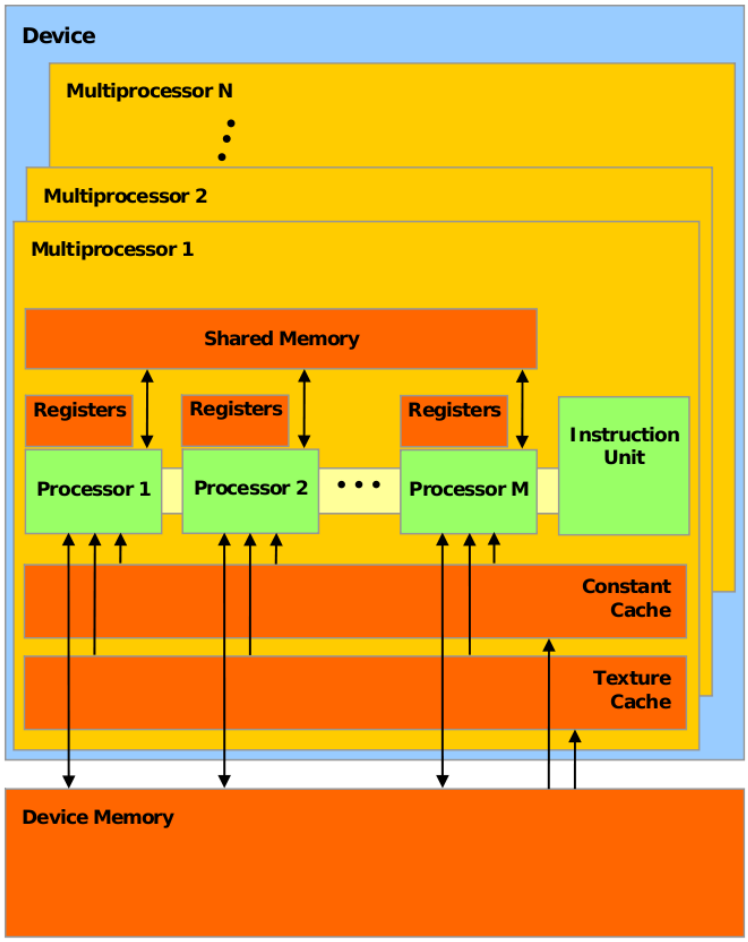

Figure 4.4 Hardware schematic, CUDA Programming Guide.

without resorting to spillover onto main system memory.

The *shared memory* is located on each multiprocessor, and has the lifetime of the block running on it. This is very low-latency memory similar to a user-controlled cache. Global memory suffers from a latency of 400-600 clock cycles for operations, while shared memory accesses have a latency of 4 clock cycles or less. Due to this very large discrepancy and need to manually manage it, shared memory must be effectively used to achieve full performance on the GPU. Every kernel then typically follows this sequence of operations:

1. Determine grid location corresponding to block and thread ID's.

2. Copy relevant data from device to shared memory.

3. Perform computation using entirely shared memory.

4. Copy solution back to device memory.

The latency in memory copy operations between shared and device memory can be largely hidden by the on-chip thread scheduler, provided the memory accesses are done in a sequential manner. This is termed *memory coalescing*, and the requirements for low-latency access is described in detail in the CUDA Programming Guide. The grid representation used in the next section naturally provides coalesced memory access for most operations, aiding performance significantly.

Listing 4.2 3-dimensional matrix addition in CUDA with shared memory

```
__global__ void Add3DMatrix(float* A, float* B, float* C)
{
    int i = blockIdx.x;
    int j = blockIdx.y;
    int k = threadIdx.x;
    int idim = gridDim.x;
    int jdim = gridDim.y;
    int index = k * idim * jdim + j * idim + i;
    int bdim = blockDim.x;
    extern __shared__ char shared[];

    __syncthreads();

    float* A_s = (float*)shared;
    float* B_s = (float*)&A_s[bdim];
    float* C_s = (float*)&B_s[bdim];

    __syncthreads();

    A_s[k] = A[index];
    B_s[k] = B[index];

    __syncthreads();

    C_s[k] = A_s[k] + B_s[k];

    __syncthreads();

    C[index] = C_s[k];
}

int main()
{
    ...
    dim3 blocks( idim, jdim, 1 );
```

```
    dim3 threads( kdim, 1, 1 );
    Add3DMatrix<<< blocks, threads,kdim*3*sizeof(float) >>>( A, B, C );
}
```

The modification to listing 4.1 shown above implements matrix addition using shared memory. Of note is the addition of a third argument to the <<<...>>> syntax, which specifies the amount of shared memory to be used in the kernel. The kernel itself now contains an external reference to the shared memory, which are addressed by local pointers.

4.2 Solver implementation

The goal of this work is to describe a method for a 3-dimensional structured grid CFD solver using the SIMPLER algorithm. To that end, new approaches must be developed to efficiently map the historically serial CPU strategies to the complexities of massively parallel GPUs. Presented here is a solution that strives to maintain a balance between code simplicity, legacy Fortran compatability, hardware scalability, numerical efficiency, and applicability to a realistic range of engineering problem sizes. Details of earlier work in implementing CUDA solvers include [2, 4, 8], all of which used explicit (point-iterative) solution schemes. No prior published work utilizing a block-iterative method exists.

4.2.1 Structured grid representation

The mathematical foundation of CFD is to discretize the Navier-Stokes equations into linear systems of equations $\mathbf{Ax} = \mathbf{b}$ which can be solved by any available means. In a prototypical finite volume fluid dynamics flow solver, here we consider the software subroutines to fall into one of two categories: those responsible for calculating the coefficients of $\mathbf{A}$ and $\mathbf{b}$, and those responsible for the numerical solution of $\mathbf{x}$; we first consider the former.

In the approach here, all the computational work to be done in computing the system's coefficients are of the *trivially parallelizable* variety. That is, every calculation involved can be performed independently of every other at all cells simultaneously. In the SIMPLER algorithm, this includes calculations such as the mass flow rate equation 2.17 and conductance, and subsequently the coefficients of the momentum equations 2.22, as well as various boundary

conditions and source terms. All terms within these equations are considered constant for a given iteration, and thus can be computed in any order without error.

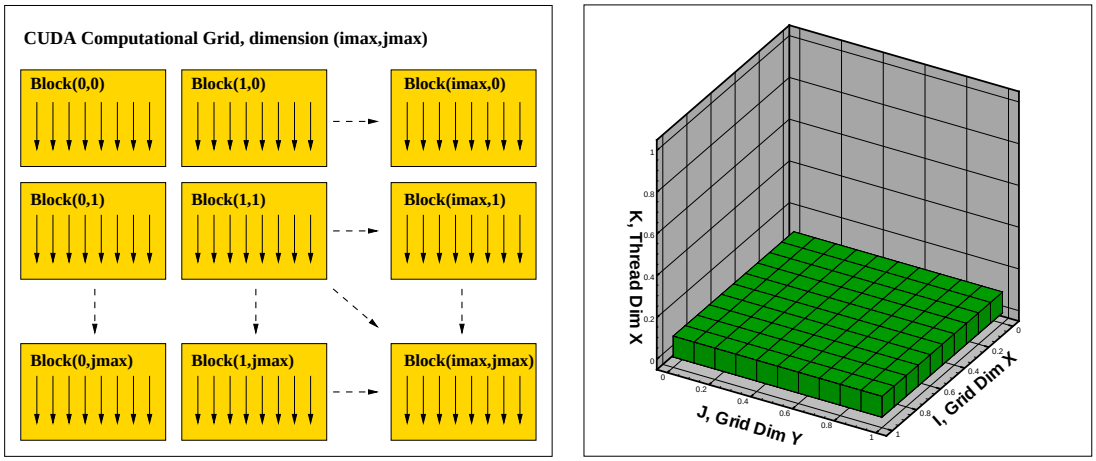

Figure 4.5 Mapping of a 2-dimensional CUDA grid to an (i, j) slice in the computational domain.

This feature permits a very direct mapping of the 3-dimensional structured computational domain onto the parallel CUDA hierarchy of blocks and threads for a majority of the program. First, we consider a slice in the $i-j$ plane. By mapping this to the concept of the 2-dimensional CUDA grid as in Figure 4.2.1, the internal block index variables `(blockIdx.x,blockIdx.y)` match directly with the computational domain variables (i, j).

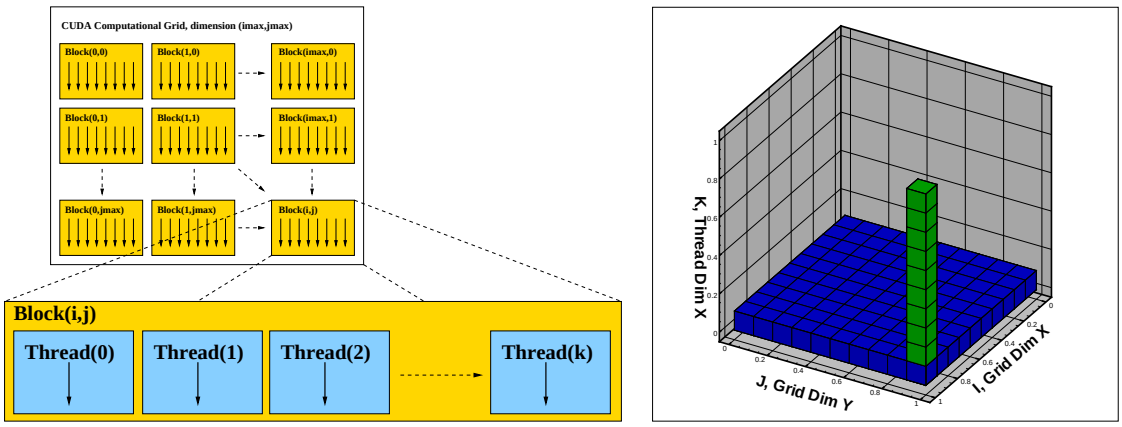

Figure 4.6 Mapping of 1-dimensional CUDA thread block in the k-direction for an (i, j) grid location onto the computational domain.

Next we consider the thread arrangement within the computational blocks. Choosing a 1-dimensional arrangement as in Figure 4.2.1, each thread corresponds to a k coordinate in the block. Using this simple scheme, kernels can address a grid element (i, j, k) at `(blockIdx.x, blockIdx.y, threadIdx.x)`.

When considering efficient use of the hardware, two dominant and often conflicting traits must be addressed. First, the number of blocks per grid should be far in excess of the number of multiprocessors available for them to be executed on. Second, the number of threads per block should be in excess of the number (8) of physical thread processors on the multiprocessor [34]. In the case of a $128 \times 128 \times 128$ grid, a total of 16384 blocks consisting of 128 threads are used. With a typical subroutine requiring anywhere from 2 to 8 variables per node with each variable requiring 4 bytes, each grid line calculation can comfortably fit within the 16KB shared memory limit of the current generation hardware. By treating the full 3-dimensional grid in terms of these 2-d and 1-d chunks, a simple but efficient way of utilizing the large degree of parallelism on the GPU is achieved.

4.2.2 Parallel solution kernel

After collecting the coefficients, the linear system of equations must be solved. Considerable care must be taken here as parallelization of a solution algorithm is not as straightforward as for coefficient collection. The original serial solution was achieved through an efficient line-based alternating direction (ADI) method. In this method, the unknowns along a line in the computational domain are solved for implicitly, while neighboring values are treated explicitly based on a previous guess. This forms a tridiagonal system which can be solved by any means available. By subsequently alternating the solution direction, e.g. solve along k grid lines, followed by j lines, then i, errors are quickly smoothed out and a converged solution is found. By treating solution kernels in terms of 1-dimensional thread blocks and grids in terms of 2-dimensional planes, flexibility is gained that allows alternating the sequence of solution by simply changing the grid and block parameters as in Figure 4.2.2..

While treating each line involved in the sweeps as an individual CUDA block as previously

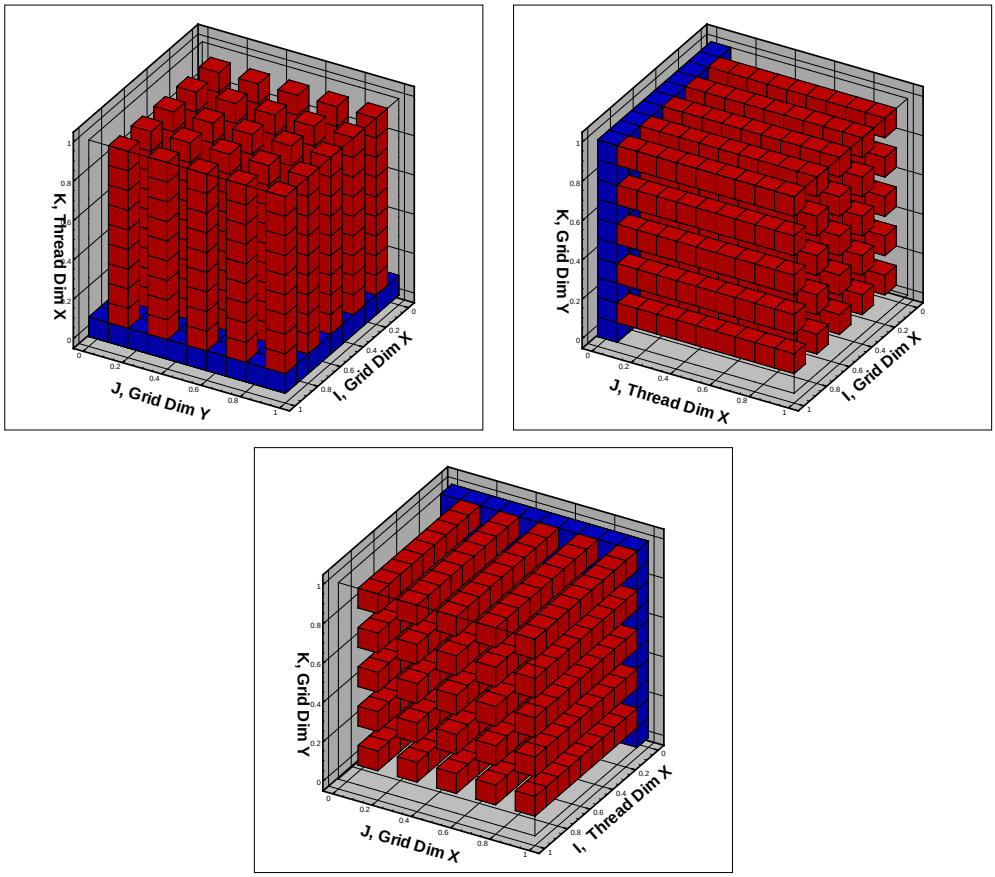

Figure 4.7 Alternating directions for tridiagonal solver. Blue represents the CUDA grid, while red represents the thread blocks and direction of implicit solution.

described is straightforward, utilizing the multiprocessor to solve the tridiagonal systems that arise is not. By implementing the parallel cyclic reduction algorithm of section 3.2.2 for the solutions of the per-block tridiagonal systems, all available thread processors can be used at near full-capacity.

4.2.3 Legacy and CPU code integration

While it is generally preferable to offload all computation to the GPU in terms of speed, this may not be possible or desirable. Interoperability with an existing CPU code base may be desired, with the GPU software used simply as a plugin to accelerate certain parts formerly done on the CPU. Or as is the case for the rotor simulation later in this work, a mostly GPU-

based code may wish to occasionally use a CPU-based function. Either way, interoperability through array transfers between the device and the main system memory is performed in the same manner.

In the present implementation, all input/output functionality is performed by CPU code, as the GPU is incapable of these tasks. The CPU is also used for all initialization tasks such as computing the various grid constants that follow from the basic grid cell inputs. After this initialization, memory on the GPU is allocated, and all necessary data is copied from main memory to the device, after which the iterative scheme can be performed. Upon completion, the desired flowfield information is copied back to main memory, where disk output can be performed and execution terminated.

Of interest in this work is the implementation of the rotor model of Rajagopalan [35]. This procedure utilizes the flowfield information in conjuntion with data about a rotor's characteristics in order to generate momentum source terms which are added to the SIMPLER routine in order to simulate a rotor. The existing code consists of CPU-based Fortran, requiring communication of the flowfield at each iteration. This necessitates transferring the pressure field and velocity component arrays from device memory to main memory, calling the external Fortran routines, and transferring the computed momentum source arrays onto the device.

Typically this behavior of large, repeated memory transfers is undesirable, due to the relatively slow bandwidth and latency compared to on-device memory. This is mitigated by noting that this rotor model, and the associated transfers, can be performed at any time after the last iteration finishes and before the current iteration solves for the velocity field. Through the use of asynchronous memory transfers, this work can generally be accomplished without reaching a synchronization barrier, minimizing loss of performance.

CHAPTER 5 RESULTS

Presented here are some sample test cases performed by the newly implemented GPU, with validation through comparison with an existing CPU-based Fortran method. Algorithmically, the two different approaches perform identical computation, with the exception of the method for solving the tridiagonal system as described in 4.2.2. In terms of convergence rate per iteration, these alternate solution methods are nearly identical, so this difference is ignored, and "full convergence" times are utilized for acceleration comparisons.

5.1 Driven Cavity

Internal flow for a 3-dimensional driven cavity is computed. The domain is a cube consisting of no-slip walls, with a "lid" at the maximum y-direction moving at a chosen velocity in the x-direction. A configuration corresponding to a Reynold's number of $Re = 100$ based on box diameter and lid velocity is chosen here.

Table 5.1 Convergence acceleration for GPU implementation on 3D driven cavity.

Grid size	GPU (s)	CPU_{serial} (s)	$\text{Speed-up}_{\text{serial}}$
126x126x126	2130	48600	22.8x
64x64x64	214	4313	20.1x
30x30x30	33	102	3.09x

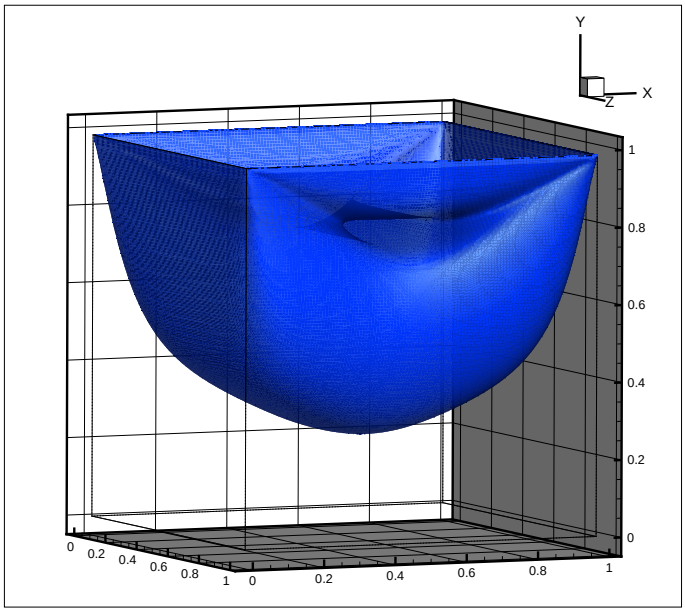

Figure 5.1 Velocity magnitude isosurface for 3D driven cavity, $Re = 100$, mesh size 126^3.

5.2 Rotorcraft simulations

Isolated rotor simulations are performed through integration of a CPU-based rotor model [35]. In coupling with the CPU code, flowfield information from the GPU is copied to main system memory at each iteration, where the rotor model computes momentum source terms which are transferred back to GPU memory. Grid size considered is $93 \times 93 \times 66$.

Table 5.2 Convergence acceleration for GPU implementation on isolated rotor computation.

Simulation	GPU (s)	CPU$_{serial}$ (s)	Speed-up$_{\text{serial}}$
Isolated hover, grid size $93 \times 93 \times 66$	1265	29658	23.4x

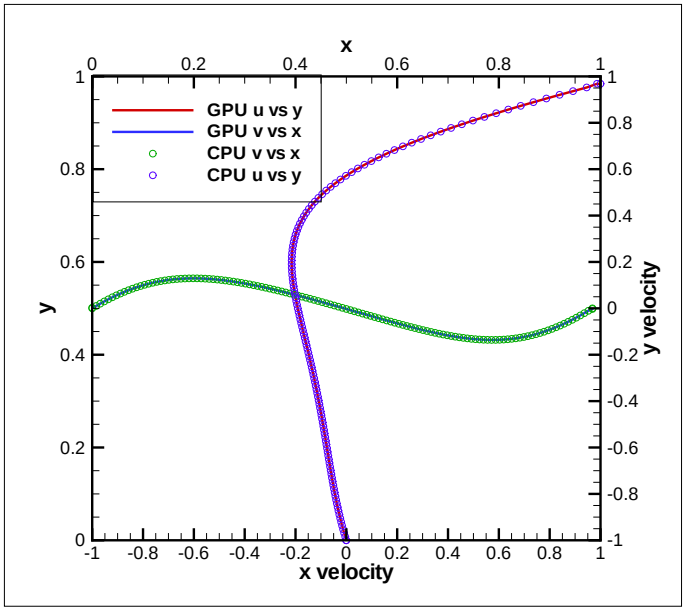

Figure 5.2 Velocity at centerlines for 3D driven cavity, $Re = 100$, mesh size 126^3.

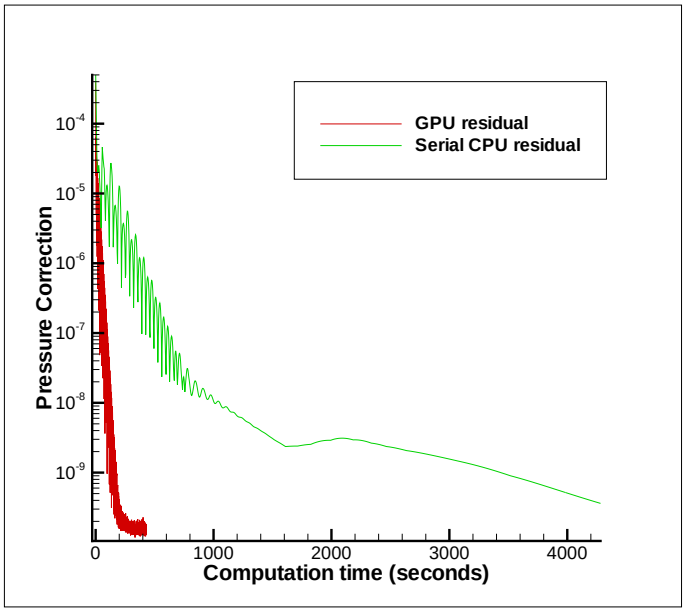

Figure 5.3 Convergence history comparison for 3D driven cavity, mesh size 64^3.

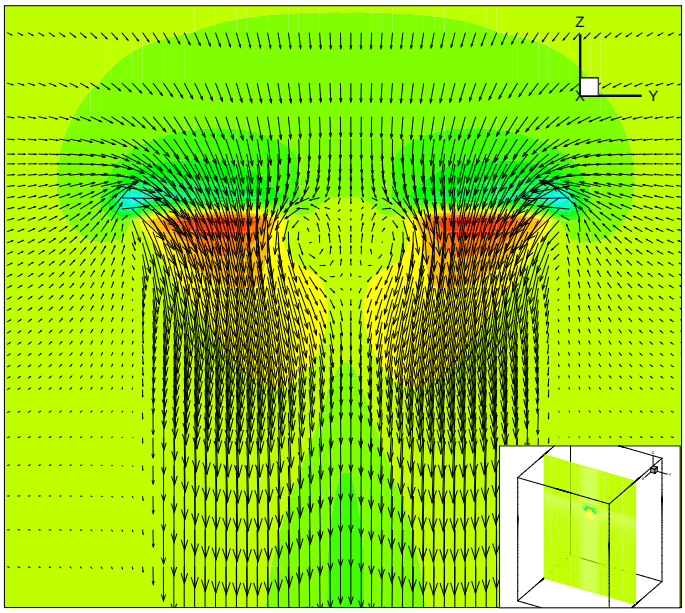

Figure 5.4 Pressure contour and velocity vectors at isolated rotor center in free hover.

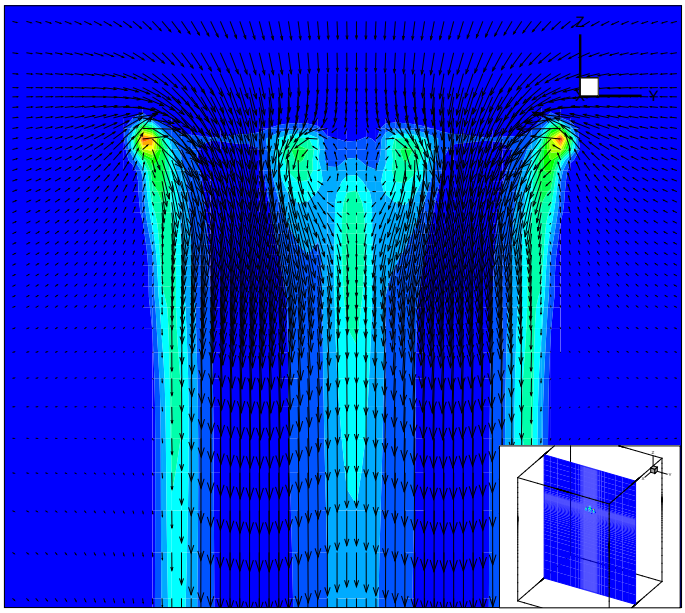

Figure 5.5 Vorticity magnitude at isolated rotor center in free hover.

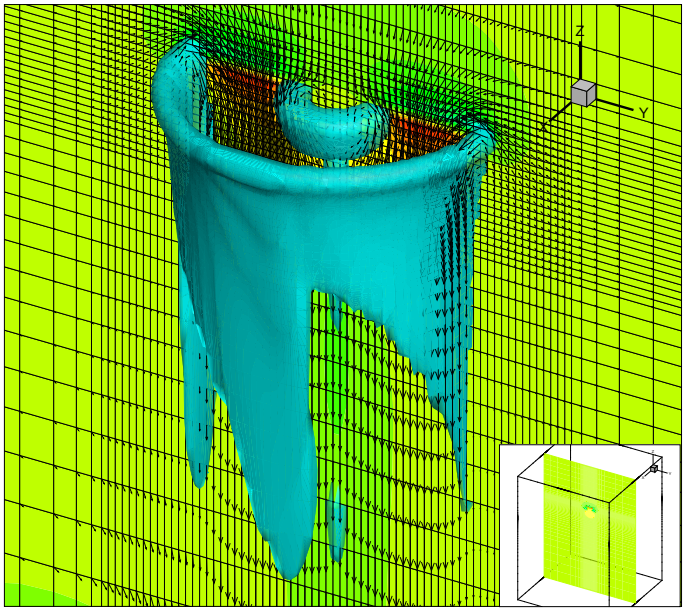

Figure 5.6 Vorticity isosurfaces in free hover.

CHAPTER 6 SUMMARY AND DISCUSSION

The recent advent of programmable graphics hardware represents a low cost alternative to traditional high performance computing. For a small investment in hardware, numerical software running on common workstations can be accelerated an order of magnitude or more, rivaling or outperforming parallel CPU servers at a fraction of the cost.

While the approach to efficient software development for these devices differs considerably from CPU-based platforms, the historical context and present trends in computing leads one to believe this change is inevitable. Massively parallel numerical software will be the rule in the future, not the exception. Fully exploiting the potential of these devices requires the development of new algorithms. With the constant demand for higher fidelity simulations, CFD solution algorithms must adapt to new, faster computer architectures.

To that end, this work has devised and implemented a computational fluid dynamics solver with considerable performance gains over the previous CPU implementation. The pressure-based SIMPLER algorithm was used for solution of the incompressible Navier-Stokes equations on a staggered, structured cartesian grid. The resulting discretized equation systems were solved through a block-iterative method resulting in a large number of tridiagonal systems, subsequently solved by a parallel vector method tailored to the unique architecture of the GPU.

Bibliography

[1] John L. Hess and A.M.O. Smith. Calculation of non-lifting potential flow about arbitrary three-dimensional bodies. Technical Report E.S. 40622, Douglas Aircraft Division, 1962.

[2] Tobias Brandvik and Graham Pullan. Acceleration of a 3D Euler solver using commodity graphics hardware. In *46th AIAA Aerospace Sciences Meeting*, 2008.

[3] Erich Elsen, Patrick LeGresley, and Eric Darve. Large calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics*, 227(24):10148–10161, 2008.

[4] Andrew Corrigan, Fernando Camelli, Rainald Löhner, and John Wallin. Running unstructured grid based CFD solvers on modern graphics hardware. In *19th AIAA Computational Fluid Dynamics Conference*, 2009.

[5] Graham Pullan. NVIDIA Tesla Case Study: CFD. Technical report, University of Cambridge, 2009.

[6] Reiji Suda, Rakayuki Aoki, Shoichi Hirasawa, Akira Nukada, Hiroki Honda, and Satoshi Matsuoka. Aspects of GPU for general purpose high performance computing. In *Asia and South Pacific Design Automation Conference*, 2009.

[7] Patrick LeGresley. Case study: Computational fluid dynamics (CFD). In *International Supercomputing Conference*, 2008.

[8] Jonathan M. Cohen and M. Jeroen Molemaker. A fast double precision CFD code using CUDA. In *21st International Conference on Parallel Computational Fluid Dynamics*, 2009.

[9] Joseph M. Elble, Nikolaos V. Sahinidis, and Panagiotis Vouzis. GPU computing with Kaczmarz's and other iterative algorithms for linear systems. *Parallel Computing*, 36(5-6):215 – 231, 2010. Parallel Matrix Algorithms and Applications.

[10] Jack Dongarra, Shirley Moore, Gregory Peterson, Stanimire Tomov, Jeff Allred, Vincent Natoli, and David Richie. Exploring new architectures in accelerating CFD for Air Force applications. In *Proceedings of DoD HPCMP*, 2008.

[11] Chris Edwards. Game on for acceleration. *Engineering and Technology*, 3(11), 2008.

[12] Julien C. Thibault and Inanc Senocak. CUDA Implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. In *47th AIAA Aerospace Sciences Meeting*, 2009.

[13] Inanc Senocak, Julien C. Thibault, and Matthew Caylor. Rapid-response urban CFD simulations using a GPU computing paradigm on desktop supercomputers. In *8th Symposium on the Urban Environment*, 2009.

[14] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2), 2008.

[15] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6), 2010.

[16] M. Soria, C. D. Pérez-Segarra, and A. Oliva. A direct parallel algorithm for the efficient solution of the pressure-correction equation of incompresible flow problems using loosely-coupled computers. *Numerical Heat Transfer, Part B: Fundamentals*, 41(2):117–138, 2002.

[17] V. Venkatakrishnan, Horst D. Simon, and Timothy J. Barth. A MIMD implementation of a parallel Euler solver for unstructured grids. *The Journal of Supercomputing*, 6(2):117–137, 1992.

[18] Erik Brakkee and Piet Wesseling. Schwarz domain decomposition for the incompressible Navier-Stokes equations in general coordinates. Technical report, Faculty of Technical Mathematics and Informatics, Delft University of Technology, Delft, 1994.

[19] E. Brakkee, C. Vuik, and P. Wesseling. Domain decomposition for the incompressible Navier-Stokes equations: solving subdomain problems accurately and inaccurately. *International Journal for Numerical Methods in Fluids*, 26(10):1217–1237, 1998.

[20] W. D. Gropp, D. E. Keyes, L. C. McInnes, and M. D. Tidriri. Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD. Technical report, Insitute for Computer Applications in Science and Engineering, NASA Langley Research Center, 1998.

[21] Markus Nordén, Malik Silva, Sverker Holmgren, Michael Thuné, and Richard Wait. Implementation issues for high performance CFD. In *Proceedings of International Information Technology Conference, Colombo, Sri Lanka*, 2002.

[22] Howard Elman, V.E. Howle, John Shadid, Robert Shuttleworth, and Ray Tuminaro. A taxonomy and comparison of parallel block multi-level preconditioners for the incompressible Navier-Stokes equations. *Journal of Computational Physics*, 227(3):1790 – 1808, 2008.

[23] Suhas Patankar. *Numerical Heat Transfer and Fluid Flow*. McGraw-Hill, New York, 1980.

[24] Barry Smith. *Domain Decomposition*. McGraw-Hill, New York, 1996.

[25] Marián Vajteršic. *Algorithms for Elliptic Problems*. McGraw-Hill, New York, 1993.

[26] Sauro Succi. *An Introduction to Parallel Computational Fluid Dynamics*. McGraw-Hill, New York, 1996.

[27] Alan Gibbons. *Lectures on Parallel Computation*. McGraw-Hill, New York, 1993.

[28] Alfio Quarteroni. *Domain Decomposition Methods for Partial Differential Equations*. McGraw-Hill, New York, 1999.

[29] Andrea Toselli. *Domain Decomposition Methods*. McGraw-Hill, New York, Year.

[30] E. Brakkee, A. Segal, and C. G. M. Kassels. A parallel domain decomposition algorithm for the incompressible Navier-Stokes equations. *Simulation Practice and Theory*, 3(4-5):185–205, 1995.

[31] Roger Hockney. *Parallel Computers*. McGraw-Hill, New York, 1983.

[32] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In *Principles and Practice of Parallel Programming*, 2010.

[33] NVIDIA Corporation. *NVIDIA CUDA Programming Guide v3.0*, February 2010.

[34] NVIDIA Corporation. *NVIDIA CUDA Programming Best Practices Guide v3.0*, February 2010.

[35] R. Ganesh Rajagopalan and Sanjay R. Mathur. Three dimensional analysis of a rotor in forward flight. *Journal of the American Helicopter Society*, 38:14–25, 1993.

APPENDIX

Test system

All performance comparisons were performed on the following system:

- **CPU:** Intel Core2 Quad Processor Q6600 (8M Cache, 2.40 GHz, 1066 MHz FSB)
- **GPU:** BFG NVIDIA GeForce GTX 260 OC MAXCORE 55 (896MB RAM, 216 stream processor cores, 590MHz core clock, 1296MHz shader clock)
- **Memory:** 4GB DDR2
- **Operating system:** Ubuntu 9.10 x64, Linux kernel 2.6.31-16
- **CUDA compiler:** NVCC release 3.0, V0.2.1221
- **C++ compiler:** GNU g++ V4.3
- **Fortran compiler:** Intel Fortran 64 Compiler Professional V11.0.083