2009

Numerical solution of the two-phase incompressible navier-stokes equations using a gpu-accelerated meshless method

Jesse Kelly
University of Central Florida, jessemkelly@gmail.com

NUMERICAL SOLUTION OF THE TWO-PHASE INCOMPRESSIBLE NAVIER-STOKES EQUATIONS USING A GPU-ACCELERATED MESHLESS METHOD

by

JESSE M. KELLY

A thesis submitted in partial fulfillment of the
requirements for the Honors in the Major Program in
Mechanical Engineering in the College of Engineering and
Computer Science and in The Burnett Honors College at
the University of Central Florida
Orlando, Florida

Fall Term 2009

Thesis Chair: Dr. Eduardo Divo

Abstract

This project presents the development and implementation of a GPU-accelerated meshless two-phase incompressible fluid flow solver. The solver uses a variant of the Generalized Finite Difference Meshless Method presented by Gerace et al. [1]. The Level Set Method [2] is used for capturing the fluid interface. The Compute Unified Device Architecture (CUDA) language for general-purpose computing on the graphics-processing-unit is used to implement the GPU-accelerated portions of the solver. CUDA allows the programmer to take advantage of the massive parallelism offered by the GPU at a cost that is significantly lower than other parallel computing options.

Through the combined use of GPU-acceleration and a radial-basis function (RBF) collocation meshless method, this project seeks to address the issue of speed in computational fluid dynamics. Traditional mesh-based methods require a large amount of user input in the generation and verification of a computational mesh, which is quite time consuming. The RBF meshless method seeks to rectify this issue through the use of a grid of data centers that need not meet stringent geometric requirements like those required by finite-volume and finite-element methods. Further, the use of the GPU to accelerate the method has been shown to provide a 16-fold increase in speed for the solver subroutines that have been accelerated.

Acknowledgments

I would like to thank Dr. Eduardo Divo, Dr. Alain Kassab, and Dr. Hassan Foroosh for their help, guidance, and inspiration. I would also like to thank Sebastian Sotelo for his help with running Fluent verification cases, Sal Gerace and Dr. Kevin Erhart for their advice and help debugging the fluid solver code, and my desk-neighbor Craig Rogers for providing continual feedback. I very much appreciate all of the staff and faculty at UCF who encourage undergraduate research, especially those responsible for the RAMP program, which provided me with the essential funding needed to focus on my research. Finally, I'd like to send a big "thank you" to my family and friends for traveling to support me at my thesis defense, particularly to my mom Patty for all of her help and encouragement.

Table of Contents

List of Tables

List of Figures

List of Symbols/Abbreviations

σ	Coefficient of surface tension
κ	Curvature of interface
ρ_h, ρ_l	Density of a heavier and lighter fluid, respectively
∇	Gradient operator ($\frac{\partial}{\partial x}\hat{i} + \frac{\partial}{\partial y}\hat{j}$)
φ	Helmholtz potential
∇^2	Laplace operator. $\nabla \cdot \nabla$ for scalar; for vector $\vec{a}$, $\nabla^2 \vec{a} = \nabla^2 a_x \hat{i} + \nabla^2 a_y \hat{j}$
ϕ	Level set function
p	Pressure
$\vec{V}$	Velocity
μ_h, μ_l	Viscosity of a heavier and lighter fluid, respectively
CFD	Computational fluid dynamics
CUDA	Compute Unified Device Architecture
FDM	Finite-differencing method
GFDM	Generalized Finite Difference Meshless method
GPGPU	General purpose graphics-processing-unit programming
GPU	Graphics-processing-unit
GUI	Graphical user-interface
HLSL	High Level Shader Language
KB	Kilobytes

LBM	Lattice-Boltzman method
LRBF	Localized Radial-Basis Function Collocation Meshless method
MAC	Marker-and-Cell
MB	Megabytes
NSE	Navier-Stokes equations
RBF	Radial-basis function

Introduction

Speed is always a central issue in computational fluid dynamics (CFD). The numerical solution of the Navier-Stokes equations (NSE) is a complex and computationally-intensive process. Most modern methods in CFD require the solution of a large linear system of equations at each time step, which is a significant bottleneck in solution time. Additionally, popular methods for the solution of the NSE such as the Finite Volume Method [3, 4], Finite Element Method [5, 6] and Finite Difference Method [7] require a structured mesh or grid of the problem domain. Mesh generation involves considerable user input, and is often the most time consuming stage in the solution process due to the strong dependence of solution accuracy on mesh quality [8].

Substantial acceleration of CFD codes can be achieved by solving the NSE in parallel using distributed or multi-core computing [9-11]. Often, however, the hardware required for this approach is prohibitively expensive, and programming such systems requires extensive training. A common, efficient, and affordable embodiment of an integrated, massively-parallel multi-core processor is the graphics-processing-unit (GPU). Scientists and engineers have been harnessing the parallelism of the GPU for over a decade [12]. It is low in cost, high in processor count, and inherently parallel in nature, making it an excellent choice for many problems in scientific computing. Until recently, the GPU was not completely programmable. Programmers had to interface with the GPU using graphics APIs such as OpenGL and shader languages such as GLSL, and Cg. With the advent of completely programmable GPUs and the first general-purpose

programming language for the GPU, NVIDIA's Compute Unified Device Architecture (CUDA) language [13], the GPU has become an extremely effective and economical tool for numerical analysis.

Mesh generation bottlenecks may be assuaged by using adaptive meshing [14, 15] or unstructured grid solvers [16-20], though a more straightforward alternative is to remove the necessity for a mesh altogether by using a meshless method. Meshless methods have been in development since the early 1990s, and include methods such as Diffuse Element Methods [21], Element-Free Galerkin Methods [8], and Local Petrov-Galerkin Methods [22]. These methods are not truly meshless, however, since they all require some type of background mesh or shadow elements. The Localized Radial-Basis-Function (LRBF) Collocation method presented by Divo et al. [23], is a truly meshless method in the sense that it requires no order among computational nodes, no polygonization, and no background mesh. The LRBF method uses expansion functions to interpolate for field variables over local, automatically constructed topologies. The Generalized Finite Difference Meshless Method (GFDM) presented by Gerace et al. [1], uses the principles of the LRBF method to interpolate for virtual finite-differencing nodes on an unstructured distribution of data centers. The derivatives of field variables can then be computed at the data centers by applying finite-differencing stencils to the virtual nodes. The data center distribution can be efficiently and completely generated programmatically.

This project will seek to significantly reduce the time and cost associated with CFD analysis through the combined usage of GPU-acceleration using CUDA and the

GFDM meshless method. A two-phase incompressible unsteady fluid flow solver will be programmed using the methods discussed, and the accuracy and robustness of the solver will be examined.

1 Background

In this chapter, the methods that inspired and contributed to the method presented in this project will be surveyed. Mesh-based and meshless methods for CFD will be discussed, as will the use of the Level Set Method for the simulation of two-phase incompressible flows. Finally, the advent of the GPU as a general purpose processor and existing applications of the GPU in CFD will be reviewed.

1.1 Mesh-Based Methods

Classical methods in CFD use a mesh, or a grid of distributed points that have a certain predefined connectivity. Three of the most popular and widely used classical methods are the Finite Differencing Method (FDM), the Finite Volume Method (FVM), and the Finite Element Method (FEM) [24]. Each of these methods requires the user to generate a mesh in order to discretize the solution domain, and the accuracy and convergence of the solution is strongly dependent on mesh quality [8]. The FVM and FEM both require triangular meshes in two spatial dimensions, and tetrahedral meshes in three spatial dimensions (in the case of the FEM, different element geometries are possible, but these two are the ones most commonly used). A significant contributor to overall mesh quality is the aspect ratio of the individual triangles or tetrahedra. Generally, the workflow for a mesh-based method is as follows:

1. Create the geometry to be modeled using a CAD software package.
2. Import the geometry into meshing software. Initialize the mesh by using an automated mesh generation procedure.

3. Manually correct errors in the mesh.
4. Run CFD analysis with the mesh.
5. Identify flow regions that are under-resolved. Add mesh elements to these regions. Adding mesh elements in a mesh-based method requires re-meshing the entire problem domain.
6. Perform several iterations of Steps 4 and 5 until an accurate solution is obtained.

Idelsohn et al. [25] identify the main difficulties in mesh building as the need for a conforming mesh, in which all nodes must lie at element vertices, the need to adhere to boundary contours, and the need to have well-shaped (non-degenerate) elements. Meshless methods eliminate two of these issues: the connectivity between nodes does not need to be conformant, and since no elements are used the presence of degenerate elements is not an issue. There is still some difficulty, as Idelsohn points out, in conforming to boundary contours in meshless methods, but this is typically not a critical issue.

1.2 Meshless Methods

Meshless methods were first introduced in the 1990s in order to avoid the issues prevalent in mesh-based methods. The identifying feature of a meshless method is that the shape functions used to represent the field variables depend on the nodal distribution alone and are independent of connectivity [26]. Examples of existing meshless methods include the Element-Free Galerkin [8] method, Diffuse Element Methods [21], Partition of Unity Methods [27], Local Petrov-Galerkin Methods [22], and H-p Cloud Methods [28]. In addition, several particle-based Lagrangian methods have

been applied to fluid flow problems, most notably Smoothed Particle Hydrodynamics [29] and the semi-Lagrangian Particle Level Set method [30], however particle-based method will not be discussed here in detail.

A common feature of the aforementioned meshless methods is that they all still require some sort of background mesh, effectively making them not truly meshless. As discussed, the novelty of the LRBF method is that it is a truly meshless method. The heart of the LRBF is its use of inverse-multiquadric radial-basis functions as interpolating functions. The method only requires a nodal distribution, and has been shown to be robust for irregular point distributions, abating the need for user intervention and making the mesh-generation process entirely automated. This eliminates the main bottleneck in CFD analysis. The LRBF performs a local collocation among radial-basis expansion functions over multiple local domains. This leads to a linear system of equations that can be solved for the coefficients of the expansion functions, leading to interpolation vectors that can be stored for each node. Any linear differential operator can be applied to the expansion functions, thus differential operators are reduced to vectors and the application of such operators becomes a simple vector-vector operation.

The latest development in the LRBF family of meshless methods is the use of radial-basis functions to enhance a traditional finite differencing scheme [1, 31-33]. This method combines the simplicity and efficiency of the Finite Differencing Method with the flexibility and ease of use of the LRBF method. Essentially, when the local grid distribution is regular enough to allow for a finite-differencing approximation of derivatives, such an approximation is used. Where the grid is irregular or poorly

structured for finite-differencing, virtual nodes are introduced in a regular stencil pattern around the nodal position of focus, and these virtual nodes are subsequently used for virtual RBF-enhanced finite differencing. More technical details of the LRBF method and the LRBF virtual finite-differencing method will be presented in Section 2.5.

1.3 Two-Phase Flow Methods

Methods for free-surface and two-phase flow have been studied since the genesis of CFD. Methods used to track the free-surface boundary have included the use of marker-particles, height functions, line segments, and extra field variables used as flags. Examples of each of these methods will be presented in turn.

One of the earliest and most popular free-surface flow methods is the Marker-and-Cell (MAC) method introduced by Harlow and Welch [7]. In the MAC method, a regular grid and finite-differencing is employed to solve the Navier-Stokes equations. The location of the fluid is determined by mass-less Lagrangian marker particles distributed throughout the computational domain. The marker particles are simply advected using the fluid velocity field. The defining characterizing of the MAC grid is that it is non-collocated, that is the pressure and the velocity of the fluid are defined at different points. Each cell on the grid is made up of a cell center, at which the pressure is defined,

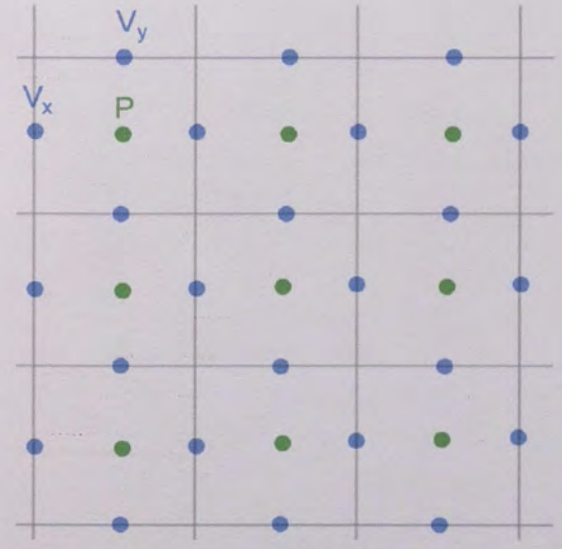

Figure 1. Collocated MAC grid.

and four cell edges, on whose centers the velocity is defined. An example of the MAC grid is shown in Figure 1. The purpose of the non-collated grid was to ease the introduction of boundary conditions and to eliminate spurious oscillations that can occur when a finite-differencing approximation is used when the velocity and pressure fields are collocated. The MAC method is a one-phase method. While it allows the simulation of free surface flow, the lighter fluid is simply implied; it is not physically present in the simulation and so can not affect the evolution of the flow of the denser fluid.

Height functions can also be used to track the free surface of a fluid. Hirt et al. [34] used a height function of the form shown in equation (1), which was evolved according to equation (2).

$$h = f(x,t) \tag{1}$$

$$\frac{\partial h}{\partial t} + u\frac{\partial h}{\partial x} = v \tag{2}$$

The main disadvantage of this formulation is that non-function representations of the free surface are not possible, meaning breaking waves or droplet formation cannot be handled by the method.

The use of line segments to track the fluid interface is a complex method that is difficult to program, often requiring programmatic surgical modification of the interface in order to prevent tangling of the line segments and their Lagrangian endpoints on the free surface. An example of this approach can be found in Nichols et al. [35].

The introduction of the Volume of Fluid (VOF) method by Hirt [36] allowed free surface flow including non-functional representation of the free-surface to be easily simulated. The VOF method uses an additional field variable to represent the volume

fraction of the fluid in each computational cell. A value of 1 indicates that the cell is full of the fluid, and a value of 0 indicates that there is no fluid in the cell. The fluid volume fraction variable *f* is evolved according to equation (3).

$$\frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} + v\frac{\partial f}{\partial y} = 0 \tag{3}$$

Equation (3) indicates that the fluid volume fraction is advected along by the velocity field of the fluid, and that it is conserved in each computational cell.

1.3.1 Level-Set Method

Most of the methods discussed above are one-phase methods, meaning the free surface flow is computed as if the fluid under examination is in a vacuum. In order to model free-surface flow as it actually occurs on earth, a two-phase method is needed that will capture the physical interaction between the lighter and denser fluids. One such method is formulated by using the Level Set Method (LSM), introduced by Osher and Sethian [2], to define the two phases of the flow. The LSM is an interface-capturing method similar to VOF in that it uses a scalar function to represent an interface. The LSM represents an implicit surface as the zero-level set of a scalar function that is on the order of the number of dimensions plus one. For example, in two dimensions an arbitrary interface (which will be a curve) can be represented by the zero level set of a three-dimensional function. Let I represent the interface, then I is composed of all the points (x, y) such that $\phi(x, y) = 0$, where ϕ is a function of x and y in three dimensions. The interface can be transported by any arbitrary velocity field using the following equation:

$$\frac{\partial \phi}{\partial t} + (\vec{V} \cdot \nabla)\phi = 0 \tag{4}$$

Equation (4) is typically referred to as the Level Set Equation, and expresses that the level set function ϕ is advected and conserved according to an arbitrary velocity field $\vec{V}$. Often ϕ is set to a signed distance function from the interface because of the many useful properties of signed distance functions, which will be discussed in detail in Section 2.4. A signed distance function is a representation of the form shown in equation (5), where d is the distance from any point in the domain (x, y) to the nearest point on the interface.

$$\phi(x, y) = \pm d \tag{5}$$

The sign in equation (5) is specified in different closed regions of the domain, and must be consistent such that the sign changes whenever the interface is crossed. For example, a circle can be represented by the LSM using a cone. The region inside of the circle is defined where the level set is negative, the region outside of the circle is defined where the level set is positive, and the edge of the circle is defined where the level set is zero, as shown in Figure 2.

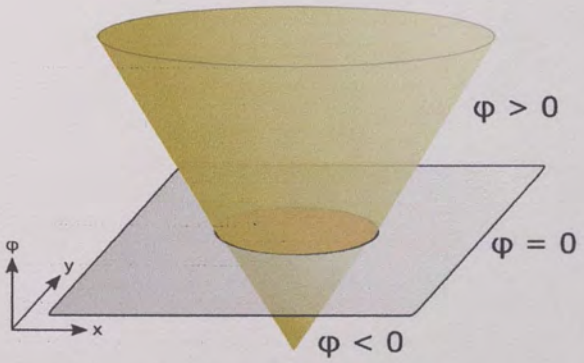

Figure 2. Level-set representation of a circle.

In two-phase incompressible flow, the Level Set Method can be used to implicitly capture the fluid interface. The fluid velocity is used to advect the interface according to equation (4), and the density and viscosity of the fluid is updated at each time step using

the sign of the level set function. More details will be presented in Section 2.4.

The LSM has been well studied in the simulation of two-phase incompressible flow (See Ref. [37-39]). Most of these projects have employed either the MAC grid or some other form of finite-differencing approximation. This project marks the first time the level set equation has been solved using a radial-basis function meshless method for two-phase incompressible fluid flow.

1.4 GPU-Acceleration

The modern central processing unit is a serial computing device, executing one instruction after the other as quickly as possible. While this makes the CPU a good general-purpose computing device, it does not allow the programmer to take advantage of potential parallelism in the execution of codes. Another piece of modern computer hardware is the graphics processing unit, which is the central component in the graphics card in a personal computer. The GPU is a highly parallel processor, containing multiple cores or processors that individually are much less powerful than the CPU. When implemented to take advantage of parallelism, however, a program may run orders of magnitudes more quickly on the GPU due to the distribution of processing among the GPU cores. Not all algorithms have the potential to be parallelized, and certain tasks may not run faster on the GPU due to their inherently serial nature.

Most computational methods used in CFD have a high degree of potential parallelism, making CFD codes excellent candidates for being ported to the GPU.

Traditionally, graphics hardware was based on a fixed-function pipeline intended to be extremely efficient for 3-D graphics operations. As such, there was little to no

flexibility in the programs available on the GPU. In the early 1990s as graphics hardware began to evolve, the GPU began to be explored as a general-purpose processor. Several early applications included robot path-planning [40], neural networks [41], and interference detection [42]. As the 21st century approached, graphics hardware became more programmable. 2001 saw the introduction of fully programmable graphics hardware and the creation of an assembly language that could be implemented at the vertex-level [12]. This prompted an explosion in applications for general purpose processing on the GPU (see Ref. [43] for an in-depth survey of modern general-purpose GPU programming applications). At this point, however, programs on the GPU still had to be written using the 3-D graphics paradigm in what were termed "shader languages". A shader language is a high-level programming language used to control per-pixel or per-vertex operations on the GPU. Modern shader languages include NVIDIA's Cg (C for graphics) [44] and Microsoft's High Level Shader Language (HLSL) [45]. These languages abstract the graphics hardware in terms of graphics data structures such as vertices, pixels, textures, and render buffers. For a programmer unfamiliar with graphics programming, general-purpose programming on the GPU (GPGPU) using a shader language is a confusing process that requires reinterpreting the problem in terms of the graphics pipeline (see Figure 3). Until recently, this was the only option for GPGPU. The Compute Unified Device Architecture language, created by the NVIDIA Corporation, introduces a C-like language for general-purpose computing that compiles to run on NVIDIA graphics cards, and allows the programmer to implement highly parallel programs without needing to explicitly manage thread distribution. The CUDA interface

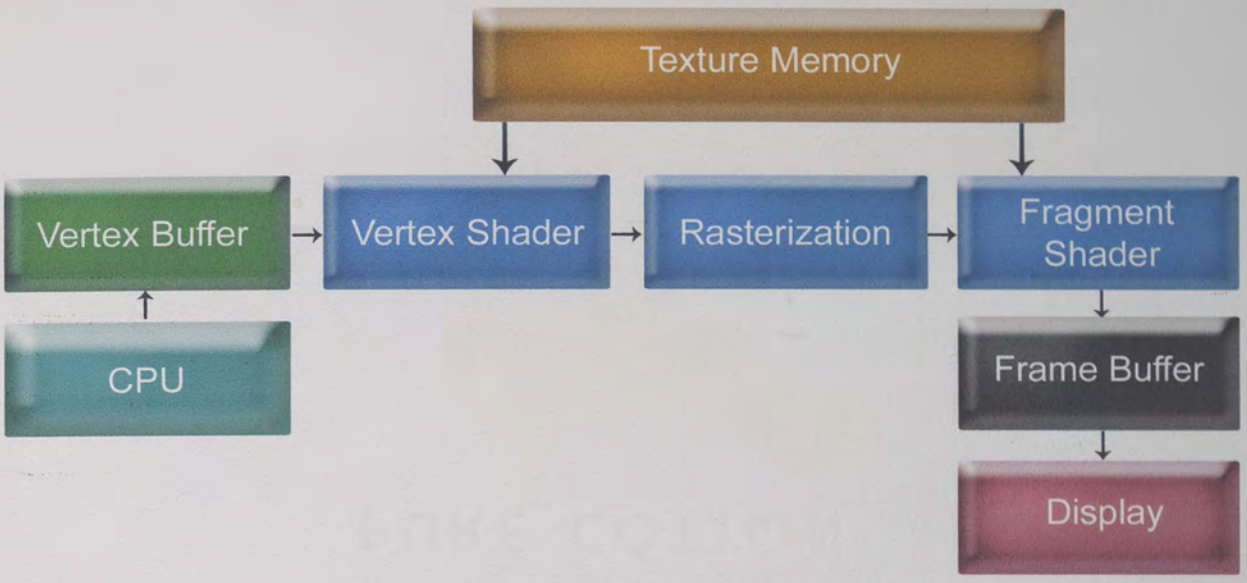

Figure 3. Programmable graphics pipeline.

takes care of thread management and assignment to processors on the GPU, so the programmer has only to specify how many threads are needed along with a few other parameters describing how the program should execute.

The GPU is a Single-Instruction Multiple-Data (SIMD) device, meaning it executes the same or near-same code in each thread, but on different pieces of data. CUDA organizes thread execution on three levels: threads, blocks, and grids. A block is a collection of threads, and threads within a block may be indexed with up to 3 dimensions. A grid is a collection of blocks, and may also be indexed with up to 3 dimensions. Instead of functions, code that executes on the GPU is written in kernels. The user specifies an execution configuration for each kernel launch, which determines the number of threads and blocks used to execute the kernel. The kernel code is executed simultaneously in each thread of each block. Branching and divergence among threads is allowed, such as would occur in conditional statements, however this should be avoided as it decreases the performance of the code considerably. Each thread has

access to a unique thread and block identification, telling it where it is in the computational grid. These IDs can be used to access data, so that each thread accesses a different chunk of data. For example, in C a user could populate an array using a "for" loop as shown below:

```
void FillArray(float* array, int array_size)
{
      for (i = 0; i < array_size; i++)
      {
            array[i] = i;
      }
}
```

In CUDA, however, each element of the array could be populated simultaneously using a kernel execution:

```
__global__ void gpu_FillArray(float* array, int array_size)
{
      int ind = blockIdx.x * blockDim.x + threadIdx.x;
      array[ind] = ind;
}
```

The "__global__" flag in the CUDA code above indicates to the compiler that it is a kernel to be executed on the GPU, "blockIdx" and "threadIdx" are structures containing the indices of the thread within the thread block and the block within the grid, and "blockDim" is a structure containing the number of threads in the block. A simple one-dimensional grid can be visualized as shown in Figure 4 below.

The graphics hardware organizes threads executing in a block into "warps". Each

Figure 4. Layout of CUDA threads, blocks, and grid.

warp is made up of 16 threads. The memory access architecture on the GPU is designed to give optimal memory bandwidth for sequential memory reads of 32-bit words by each thread in a warp. The GPU will be most efficient if all warps are fully populated, meaning that each block size is a multiple of 16. The GPU is generally expected to show a better performance gain over the CPU when all of its multiprocessors are occupied, so a larger percent increase in speed should be expected for larger computational grid sizes. CUDA extends the range of possibilities for GPGPU. Additionally, it is superior to shader languages in that it is accessible to all programmers regardless of previous graphics programming experience.

1.4.1 CFD on the GPU

In the early 2000s, researchers began implementing solutions of the incompressible Navier-Stokes equations on the GPU using shader languages. Bolz et al. implemented Conjugate Gradient and Multigrid solvers on the GPU in order to solve the pressure Poisson equation derived from the NSE [46]. They reported a speed increase factor of 1.6 for multigrid implemented on the GPU for an unstructured matrix system, and a 1.8 times speedup for a structured system. The solution of the NSE was carried out on a structured grid following a finite-difference formulation. Another multigrid implementation on the GPU appears in Goodnight et al. [47]. In this case, multigrid on the GPU was used to accelerate the solution of the vorticity-stream function formulation of the NSE; a maximum speed increase factor of about 2.6 was reported.

Liu et al. used the GPU to solve the three-dimensional Navier-Stokes equations [48] following the semi-Lagrangian formulation of Stam [49]. While their method was

able to handle complex boundaries, the semi-Lagrangian method Stam presents is not physically accurate enough for engineering applications. Liu did not implement a corresponding solver on the CPU, and provides no data for the level of acceleration achieved using the GPU.

Li et al. implemented a fluid solver using the Lattice-Boltzman method (LBM) and assembly language and reported speedup factors as high as 15 for large lattice sizes. In addition, the solver was reported to be second order accurate in space and time [50].

All of the projects discussed so far have been implemented using shader languages or assembly language for the GPU. Recent implementations using CUDA will now be briefly reviewed. Riegel et al. implemented the LBM using CUDA, observing a speedup factor of 9 over a multi-core CPU [51]. Thibault et al. observed a speedup factor of 13 using a single GPU and a speedup factor of 100 using a quad-GPU setup for solution of the NSE using a MAC grid and finite-differencing, both speedups being with respect to a single-core CPU [52]. Cohen and Molemaker reported an 8 times speedup with respect to a high-end eight-core CPU of the GPU-accelerated solution of the NSE using the Boussinesq approximation and a finite-volume method [53]. Brandvik and Pullan implemented a finite-volume solution of the 3-D Euler equations using CUDA and reported a speedup factor of 16 over a single-core CPU [54].

Other methods that have been ported to the GPU that will not be discussed here in detail include particle-based methods [55] and smoothed-particle hydrodynamics [56].

This project is unique in that it is the first to implement a radial-basis function

collocation meshless method for CFD on programmable graphics hardware. More specifically, it presents the first GPU-accelerated version of the Generalized Finite Difference Meshless Method. In addition, this project joints other researchers in being at the forefront of exploring both scientific computing on the GPU and CFD applications using CUDA.

2 Development of Method

This chapter will detail the theoretical development of the method used for the solution of the two-phase incompressible Navier-Stokes equations. An explicit first forward Euler time-stepping scheme is employed for integration of the NSE. Approximation of derivatives is carried out using the Generalized Finite Difference Meshless Method (GFDM). The interface between the lighter and denser fluid is captured using the Level Set Method, the solution of which is also performed using GFDM. Finally, automatic segmentation of the domain used in the solution on the GPU is performed using recursive bisection.

2.1 The Navier-Stokes Equations

The incompressible Navier-Stokes equations, given by (6) and (7), are fully coupled partial differential equations, with (6) being nonlinear. Their solution requires a full and complete set of boundary conditions.

$$\frac{\partial \vec{V}}{\partial t} = -(\vec{V} \cdot \nabla)\vec{V} - \frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2 \vec{V} + \vec{g} \tag{6}$$

$$\nabla \cdot \vec{V} = 0 \tag{7}$$

We will now examine each term in (6) in turn.

2.1.1 External Forces ($\vec{g}$)

The $\vec{g}$ term corresponds to any external forces, including gravity, that act on the fluid. These may be due to external accelerations, electromagnetic forces, or artificially generated force fields. Although the fluid solver presented here was written with the

capability of applying an externally generated spatially-varying force field, the only force applied in practice was gravity, and so from here the $\vec{g}$ term will be assumed to act in the negative y-direction with a magnitude of 9.81 m/s^2.

2.1.2 *Diffusion* ($\frac{\mu}{\rho}\nabla^2\vec{V}$)

This term is the contribution of viscous effects, and acts to diffuse the velocity of the fluid. As can quickly be seen from an examination of this term, the velocity fields of fluids with a higher kinematic viscosity will diffuse more quickly. For example, a disturbance in the velocity of thick syrup will be smoothed out more quickly than a disturbance of equal magnitude in a light gas. The $\nabla^2\vec{V}$ term represents the vector Laplacian of the velocity field. In two dimensions, this is given by (8):

$$\nabla^2\vec{V} = \nabla^2 V_x\hat{i} + \nabla^2 V_y\hat{j} \tag{8}$$

2.1.3 *Convective Acceleration* ($(\vec{V}\cdot\nabla)\vec{V}$)

The $(\vec{V}\cdot\nabla)\vec{V}$ term is due to convective acceleration in the fluid. Essentially, it represents the fact that the property of velocity at any given point in the fluid is carried along by the fluid as it moves. In steady flow, a consequence of this term is that though the overall flow may not by varying with time, acceleration may still take place with respect to space because the velocity field varies spatially. This term is nonlinear and requires care in order to be correctly resolved numerically. In two dimensions, this term may be expanded as (9):

$$V_x \frac{\partial \vec{V}}{\partial x} + V_y \frac{\partial \vec{V}}{\partial y} \tag{9}$$

$\vec{V} \cdot \nabla$ is referred to as the convective derivative operator. As (9) shows, the information due to the derivative of the velocity field with respect to the x-direction at any point in the fluid will be transported in whatever direction the x-velocity is flowing at that point (which will be either in the positive or negative x-direction). This introduces the need for upwinding the spatial derivative operators when evaluating the convective acceleration term. Upwinding alters the derivative operator based on the direction of the velocity at the point on which the derivative operator will act. More details are presented in Section 2.6.

2.1.4 Pressure ($\frac{1}{\rho}\nabla p$)

An important feature of incompressible flow is that it is driven by the gradient of the pressure; the absolute pressure is unimportant. Solution of the pressure field is the most time-consuming process in the numerical solution of the NSE, and will be discussed shortly.

2.2 Two-Phase Incompressible Navier-Stokes Equations

In two-phase flow a moving boundary between the two phases will exist, and the flow will be governed by two equations, one for each phase of the flow (along with the continuity equation, which holds over the entire fluid domain):

$$\rho_h[\frac{\partial \vec{V}_h}{\partial t} + (\vec{V}_h \cdot \nabla)\vec{V}_h] = \nabla p_h + \mu_h \nabla^2 \vec{V}_h + \vec{g} \qquad \vec{x} \in \text{denser fluid} \tag{10}$$

$$\rho_l[\frac{\partial \vec{V}_l}{\partial t} + (\vec{V}_l \cdot \nabla)\vec{V}_l] = \nabla p_l + \mu_l \nabla^2 \vec{V}_l + \vec{g} \qquad \vec{x} \in \text{lighter fluid} \tag{11}$$

In equations (10) and (11), the subscript h indicates the denser fluid, and the subscript l indicates the lighter fluid. An additional boundary condition is needed at the fluid interface, and is given by equations (12) and (13), where I denotes the fluid interface, $\vec{n}$ denotes a unit vector normal to the interface, σ is the coefficient of surface tension and κ is the curvature of the interface (see Batchelor [57] for details).

$$(\mu_h \nabla^2 \vec{V} - \mu_l \nabla^2 \vec{V}) \cdot \vec{n} = (\rho_h - \rho_l + \sigma\kappa)\vec{n} \qquad \vec{x} \in I \tag{12}$$

$$\vec{V}_h = \vec{V}_l \qquad \vec{x} \in I \tag{13}$$

We can then define:

$$\vec{V} = \begin{cases} \vec{V}_h & \vec{x} \in \text{denser fluid} \\ \vec{V}_l & \vec{x} \in \text{lighter fluid} \end{cases} \tag{14}$$

$$\mu = \begin{cases} \mu_h & \vec{x} \in \text{denser fluid} \\ \mu_l & \vec{x} \in \text{lighter fluid} \end{cases} \tag{15}$$

$$\rho = \begin{cases} \rho_h & \vec{x} \in \text{denser fluid} \\ \rho_l & \vec{x} \in \text{lighter fluid} \end{cases} \tag{16}$$

and combine equations (10) and (11) into an equation which is valid over the entire domain (See Chang et al. [37] for a derivation):

$$\frac{\partial \vec{V}}{\partial t} = -(\vec{V} \cdot \nabla)\vec{V} - \frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2 \vec{V} + \vec{g} - \sigma\kappa\delta(d)\vec{n} \tag{17}$$

In equation (17), $\delta(d)$ is the Dirac delta function and d is the signed distance from the fluid interface. We define the signed distance d as the positive distance from the interface when $\vec{x}$ is in the area of the domain occupied by the lighter fluid, and as

the negative distance from the interface when $\vec{x}$ is in the area of the domain occupied by the heavier fluid.

2.3 Numerical Solution of the Navier-Stokes Equations

Equation (17) is solved using a fractional step method similar to the method presented by Chorin [58]. Integration in time is carried out using a first-order Euler time stepping scheme. We begin with an initial velocity field that satisfies the continuity equation (7).

$$\nabla \cdot \vec{V}^0 = 0 \tag{18}$$

We define three time levels, k, $k+1$, and * such that $t^{k+1} = t^k + \Delta t$. An intermediate velocity field $\vec{V}^*$ can be calculated using:

$$\frac{\vec{V}^* - \vec{V}^k}{\Delta t} = -(\vec{V}^k \cdot \nabla)\vec{V}^k - \frac{1}{\rho}\nabla p^k + \frac{\mu}{\rho}\nabla^2 \vec{V}^k + \vec{g}^k - ST^k$$

$$\vec{V}^* = \vec{V}^k + \Delta t[-(\vec{V}^k \cdot \nabla)\vec{V}^k - \frac{1}{\rho}\nabla p^k + \nu\nabla^2 \vec{V}^k + \vec{g}^k - ST^k] \tag{19}$$

where ST^k represents the surface tension effects:

$$ST^k = \sigma\kappa^k \delta(d^k)\vec{n}^k \tag{20}$$

Since $\vec{V}^*$ has been calculated without considering the continuity equation, $\vec{V}^*$ will not be divergence-free. The field $\vec{V}^*$ can be decomposed into divergence-free and irrotational components as a consequence of the Helmholtz-Hodge decomposition theorem (also called the Ladyzhenskaya decomposition theorem after Russian mathematician and fluid dynamicist Olga Aleksandrovna Ladyzhenskaya). We define the divergence-free component of $\vec{V}^*$ as $\vec{V}^{k+1}$, resulting in:

$$\vec{V}^* = \vec{V}^{k+1} + \nabla\varphi^{k+1} \tag{21}$$

where φ^{k+1} is a scalar field called the Helmholtz potential. Dotting equation (21) with the divergence operator and using the fact that $\nabla \cdot \vec{V}^{k+1} = 0$, we have:

$$\nabla \cdot \vec{V}^* = \nabla \cdot \vec{V}^{k+1} + \nabla \cdot \nabla\varphi^{k+1}$$

$$\nabla^2\varphi^{k+1} = \nabla \cdot \vec{V}^* \tag{22}$$

Equation (22) is a Poisson equation for the Helmholtz potential, and is solved iteratively using Jacobi iteration as discussed in section 3.2.2 using the boundary conditions:

$$\nabla\varphi \cdot \vec{n} = 0 \quad \text{at walls and inlets} \tag{23}$$

$$\varphi = 0 \quad \text{at outlets} \tag{24}$$

Once the Helmholtz potential has been obtained, the velocity field can be updated by rearranging (21):

$$\vec{V}^{k+1} = \vec{V} - \nabla\varphi^{k+1} \tag{25}$$

We then consider equation (6) at time t^{k+1}, and dot the equation with the divergence operator to obtain:

$$\nabla \cdot \frac{\partial \vec{V}^{k+1}}{\partial t} = -\nabla \cdot (\vec{V}^{k+1} \cdot \nabla)\vec{V}^{k+1} - \nabla \cdot (\frac{1}{\rho}\nabla p^{k+1}) + \nabla \cdot (\nu\nabla^2\vec{V}^{k+1}) + \nabla \cdot \vec{g}^{k+1} - \nabla \cdot ST^{k+1}$$

$$\frac{\partial}{\partial t}(\nabla \cdot \vec{V}^{k+1}) = -\nabla \cdot (\vec{V}^{k+1} \cdot \nabla)\vec{V}^{k+1} - \nabla \cdot (\frac{1}{\rho}\nabla p^{k+1}) + \nabla \cdot (\nu\nabla^2\vec{V}^{k+1}) + \nabla \cdot \vec{g}^{k+1} - \nabla \cdot ST^{k+1}$$

$$\nabla \cdot (\frac{1}{\rho}\nabla p^{k+1}) = -\nabla \cdot (\vec{V}^{k+1} \cdot \nabla)\vec{V}^{k+1} + \nabla \cdot (\nu\nabla^2\vec{V}^{k+1}) - \nabla \cdot ST^{k+1} \tag{26}$$

Since we are considering two-phase flow, density is non-constant, so expanding the left hand side of (26) results in:

$$\nabla(\frac{1}{\rho})\nabla p^{k+1} + \frac{1}{\rho}\nabla\cdot\nabla p^{k+1} = -\nabla\cdot(\vec{V}^{k+1}\cdot\nabla)\vec{V}^{k+1} + \nabla\cdot(\nu\nabla^2\vec{V}^{k+1}) - \nabla\cdot ST^{k+1}$$

$$-(\frac{\nabla\rho}{\rho^2})\nabla p^{k+1} + \frac{1}{\rho}\nabla^2 p^{k+1} = -\nabla\cdot(\vec{V}^{k+1}\cdot\nabla)\vec{V}^{k+1} + \nabla\cdot(\nu\nabla^2\vec{V}^{k+1}) - \nabla\cdot ST^{k+1}$$

$$\nabla^2 p^{k+1} - (\frac{\nabla\rho}{\rho})\nabla p^{k+1} = -\rho[\nabla\cdot(\vec{V}^{k+1}\cdot\nabla)\vec{V}^{k+1} + \nabla\cdot(\nu\nabla^2\vec{V}^{k+1}) - \nabla\cdot ST^{k+1}] \quad (27)$$

Equation (27) is an advection-diffusion equation for pressure, and it is solved iteratively using Jacobi iteration (see section 3.2) using the boundary conditions:

$$\nabla p\cdot\vec{n} = -\rho[\frac{\partial\vec{V}}{\partial t} + (\vec{V}\cdot\nabla)\vec{V} - (\nu\nabla^2\vec{V}) - \vec{g}n_y]\cdot\vec{n} \quad \text{at walls and outlets} \quad (28)$$

$$p = \hat{p} \quad \text{at inlets} \quad (29)$$

Equation (28) is derived directly from the Navier-Stokes equations and $\hat{p}$ in equation (29) is a prescribed pressure value.

Once the pressure field has been obtained, the iteration count is advanced to $k+1$ and the solution proceeds from equation (19). Using this solution method, an initial pressure field p^0 must be obtained to start the solution. As long as the initial velocity of the fluid is zero throughout the entire domain, the pressure can be initialized to the hydrostatic pressure by solving equation (30) below using the boundary conditions given in equations (31) and (32), where h_{ref} denotes a prescribed reference height.

$$\nabla^2 p^{k+1} - (\frac{\nabla \rho}{\rho})\nabla p^{k+1} = 0 \tag{30}$$

$$\nabla p \cdot \vec{n} = -\rho g n_y \quad \text{at boundaries} \tag{31}$$

$$p = 0 \quad \text{when } y = h_{ref} \tag{32}$$

Boundary conditions are also required for the velocity, and are implemented at each time step as:

$$(\nabla \cdot \vec{V}) \cdot \vec{n} = 0 \qquad \text{at outlets} \tag{33}$$

$$\vec{V} = 0 \qquad \text{at walls (no slip)} \tag{34}$$

$$(\frac{\partial \vec{V}}{\partial t}) \cdot \vec{n} = \left[-(\vec{V} \cdot \nabla)\vec{V} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \vec{V} + \vec{g} - ST \right] \cdot \vec{n} \qquad \text{at outlets} \tag{35}$$

2.4 Numerical Treatment of the Fluid Interface

The solution of two-phase incompressible flow requires the treatment of density and viscosity as additional field variables. The interface between the two phases is theoretically represented as a discontinuity in the density and pressure fields, and must be tracked somehow. This project uses the Level Set Method (LSM) introduced by Osher and Sethian [2] to capture the fluid interface.

The LSM is an interface capturing method, meaning it does not explicitly track the interface but implicitly keeps track of it as the zero-level set of a higher dimensional function (see section 1.3.1 for more details).

Let ϕ represent a scalar function of x and y, called the level set function, and let a two-dimensional curve I be represented by the set of points (x, y) for which $\phi(x, y) = 0$. We can advect the level set function, and thus the interface, using an

arbitrary velocity field $\vec{V}$ using the equation presented in Section 1.3.1:

$$\frac{\partial \phi}{\partial t} + (\vec{V} \cdot \nabla)\phi = 0 \tag{4}$$

If we allow $\phi(x, y)$ to be a function that takes on values greater than, less then, and equal to zero, then we may divide the domain into three parts corresponding to whether ϕ is nonzero positive, nonzero negative, or zero. Let $\phi(x, y) < 0$ in the region occupied by the fluid with density ρ_h, $\phi(x, y) > 0$ in the region occupied by the fluid with density ρ_l, and $\phi(x, y) = 0$ at the interface between the two fluids. Following Sussman's formulation [38], we can then define the density and the viscosity at any point in the domain using:

$$\rho(\phi) = H(\phi)(\rho_l - \rho_h) + \rho_h \tag{36}$$

$$\mu(\phi) = H(\phi)(\mu_l - \mu_h) + \mu_h \tag{37}$$

where $H(\phi)$ is the Heaviside function, given by:

$$H(\phi) = \begin{cases} 1 & \phi > 0 \\ \frac{1}{2} & \phi = 0 \\ 0 & \phi < 0 \end{cases} \tag{38}$$

The use of the Heaviside function leads to a sharp discontinuity at the fluid interface that is difficult to resolve numerically. In order to provide an easier numerical treatment of the interface, the Heaviside function is replaced by the smoothed Heaviside function that smears the interface over a predefined interface thickness.

$$H_s(\phi) = \begin{cases} 1 & \phi > \varepsilon \\ \frac{1}{2}[1 + \frac{\phi}{\varepsilon} + \frac{1}{\pi}\sin(\frac{\pi\phi}{\varepsilon})] & |\phi| \le \varepsilon \\ 0 & \phi < -\varepsilon \end{cases} \tag{39}$$

The Heaviside function H in equations (36) and (37) is replaced with H_s, giving the interface an approximate thickness of:

$$i = \frac{\varepsilon}{|\nabla\phi|} \tag{40}$$

Additionally, the Dirac delta function in equation (17) is replaced with the mollified delta function, defined as:

$$\delta_s(\phi) = \frac{\partial H_s}{\partial \phi}$$

$$\delta_s(\phi) = \begin{cases} \frac{1}{2\varepsilon}\cos(\frac{\pi\phi}{\varepsilon}) & |\phi| \le \varepsilon \\ 0 & otherwise \end{cases} \tag{41}$$

The curvature and normal unit vector of the interface are also required in order to solve equation (17). As already discussed, a signed distance function is a function whose value is either the positive or negative distance from the interface. If the level set function ϕ is a signed distance function, several useful properties arise which ease the computation of the interface curvature and normal unit vector. For a signed distance function, and unit normal vector and curvature of the interface can be found using equations (42) and (43), respectively (see Osher and Fedkiw [59]).

$$\vec{n} = \frac{\nabla \phi}{|\nabla \phi|} \tag{42}$$

$$\kappa = \nabla \cdot \frac{\nabla \phi}{|\nabla \phi|} \tag{43}$$

On order for equations (42) and (43) to remain valid for the entire solution, the level set function must be periodically reinitialized to a signed distance function. This is accomplished using the reinitialization equation presented in [60], given by equation (44).

$$\frac{\partial \phi}{\partial t} = sign(\phi)(1 - |\nabla \phi|) \tag{44}$$

Equation (44) is evolved to steady-state in fictitious time using a first order Euler's method. Generally, only a few iterations are necessary as ϕ is already sufficiently close to a signed distance function.

2.5 Radial-Basis Function Interpolated Generalized Finite-Differencing

In order to discretize equation (17), a generalized finite-differencing scheme using radial-basis function interpolation of field variables at virtual nodes is used, following the method presented by Gerace et al. [1]. This method builds off of the Localized Radial Basis Function Collocation meshless method of Divo and Kassab [23]. The essence of both methods is the use of radial-basis functions to approximate field variable distributions.

2.5.1 Radial-Basis Function Interpolation

A radial-basis function is a scalar function whose value at a point depends on the distance from the point to a predefined data center. The functions used in the solver that

is presented here follow the form of the Hardy Multiquadrics RBF:

$$\chi(\vec{x}) = [r_j^2(\vec{x}) + c^2]^{n-(3/2)} \tag{45}$$

where n is a positive integer, c is a predefined shape parameter, and r_j is the Euclidean distance between point $\vec{x}$ and some predefined point $\vec{x}_j$. In all of the radial-basis functions used in the fluid solver presented here, n = 1, leading to the inverse-multiquadric RBF:

$$\chi(\vec{x}) = \frac{1}{\sqrt{r_j^2(\vec{x}) + c^2}} \tag{46}$$

Consider Figure 5 below, which demonstrates how the shape of equation (45) changes with different values of c. Higher values of c smooth out the distribution of the inverse-multiquadric RBF, while lower values of c yield a steeper distribution and a larger

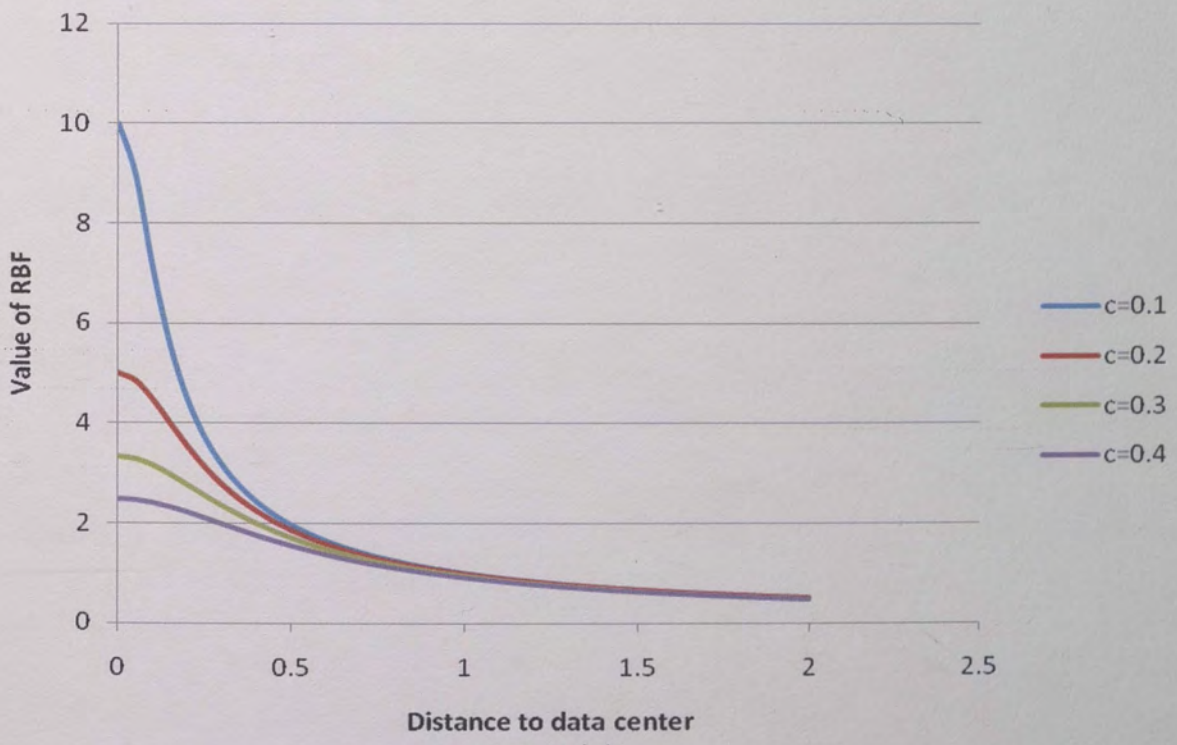

Figure 5. Variation of RBF shape with c.

magnitude when the distance to the data center is zero.

The RBF can be used as an interpolating function given the values of a function at several points in space. Consider a general function $f(\vec{x})$, of which several values f_i are known at several points $\vec{x}_i$. Let χ_i be an inverse-multiquadric radial-basis function whose data center is the point $\vec{x}_i$. The value of the function can be estimated at any point $\vec{x}$ using the known values of the function by performing a global RBF expansion according to:

$$f(\vec{x}) = \sum_i \alpha_i \chi_i(\vec{x}) \tag{47}$$

where the α_i's are scalar expansion coefficients. The expansion coefficients may be determined by collocation of the radial-basis functions at each known point $\vec{x}_i$. For example, for the known function value f_k at point $\vec{x}_k$, an equation for the collocated RBF expansion at $\vec{x}_k$ is determined using (47). This is shown in (48), where N is the total number points at which values of f are known.

$$f_k(\vec{x}_k) = \sum_i \alpha_i \chi_i(\vec{x}_k) = \alpha_1 \chi_1(\vec{x}_k) + \alpha_2 \chi_2(\vec{x}_k) + \ldots \alpha_N \chi_N(\vec{x}_k) \tag{48}$$

All of the χ_i's can be explicitly calculated using equation (46). If equation (48) is assembled for all of the known points (i.e. for k = 1 to N), then a system of linear equations to determine the expansion coefficients α_i can be formulated.

$$\begin{bmatrix} \chi_1(\vec{x}_1) & \chi_2(\vec{x}_1) & \cdots & \chi_N(\vec{x}_1) \\ \chi_1(\vec{x}_2) & \chi_2(\vec{x}_2) & \cdots & \chi_N(\vec{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \chi_1(\vec{x}_N) & \chi_2(\vec{x}_N) & \cdots & \chi_N(\vec{x}_N) \end{bmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{pmatrix}$$

$$[\chi](\alpha) = (f) \tag{49}$$

The linear system of equations in (49) is solved for the expansion coefficients.

$$(\alpha) = [\chi]^{-1}(f) \tag{50}$$

Equation (48) can be recast in terms of a vector operation as:

$$f_k = (\chi_k)^T(\alpha) \tag{51}$$

$$(\chi_k) = \begin{pmatrix} \chi_1(\vec{x}_k) \\ \chi_2(\vec{x}_k) \\ \vdots \\ \chi_N(\vec{x}_k) \end{pmatrix} \quad (\alpha) = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{pmatrix}$$

Substituting equation (50) into (51) yields:

$$f_k = (\chi_k)^T[\chi]^{-1}(f)$$

$$f_k = (\Phi_k)^T(f) \tag{52}$$

where the interpolation vector (Φ_k) can be pre-computed as:

$$(\Phi_k) = (\chi_k)^T[\chi]^{-1} \tag{53}$$

It is here that the observation made about the shape parameter c in equation (46) becomes important. The value of c determines the range of the RBF for the domain of known points. If c is small, the RBF will have a large range; if c is large, the RBF will have a small range. The range of each of the expansion functions χ_i will impact the conditioning number of the matrix in (49). A shape parameter that is too large or too

small will cause $[\chi]$ to have a large conditioning number and be difficult to invert numerically. For this reason, when the interpolation vectors are calculated in the solver presented here, the shape parameter is optimized using a simple search algorithm that targets a conditioning number between 100 and 1000 for the matrix $[\chi]$. The conditioning number is calculated using the maximum and minimum eigenvalues of the matrix provided by Singular Value Decomposition, and the matrix is inverted directly using LU decomposition and back-substitution.

Localized radial-basis function interpolation is similar to global RBF interpolation, except the points used in the RBF expansion about a given data center are limited to those that are within a certain distance of the data center. Consider the data center distribution consisting of N data centers in Figure 6. We can apply an RBF expansion at point k using only the values of the field variable and nodal locations that are within the area of influence for k, instead of collocating at all N data centers. This yields a collocation matrix that is much more well-conditioned and easier to invert, and also greatly reduces the number of operations that must be performed for the interpolation of the field variable at the data center. The influence points for each data center can be found programmatically using a nearest-neighbor search, where nodes are added to influence topologies in sequence, with the next closest node to the data center being added at each step until a prescribed number of influence points have been added to the topology. In Figure 6, the red circle encapsulates all of the influence nodes in the topology for the red data center.

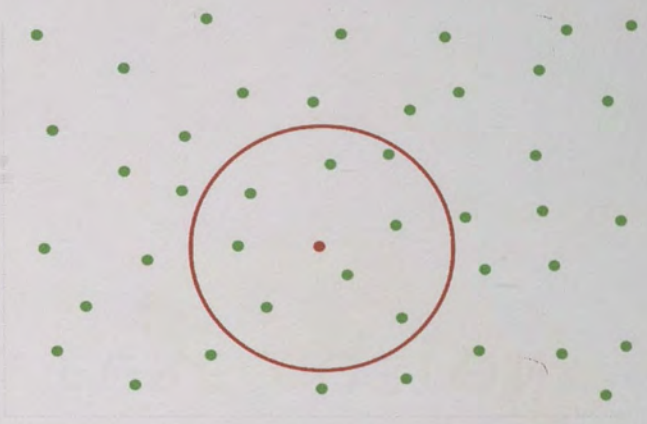

Figure 6. Local RBF influence topology.

2.5.2 *Virtual Finite-Differencing*

Consider the point distribution shown in Figure 7. Imagine that a field variable f is known at each of the points in the figure, and we wish to calculate the derivative of f with respect to the x- and y-coordinate directions at the red point k using finite differencing. Obviously the point distribution is too irregular to directly apply any of the common finite-difference stencils. Let us introduce several virtual nodes, denoted by *'s in Figure 8. The virtual nodes are spaced Δx distance apart in the x-direction, and Δy distance apart in the y-direction about the data center k. Let us assume that the value of the field variable f is also known at these virtual nodes. We can then apply standard finite-differencing to approximate the derivatives of f with respect to the x- and y-coordinate directions.

$$\frac{\partial f}{\partial x} = \frac{f(x_{k+1}) - f(x_{k-1})}{2\Delta x} \qquad \frac{\partial f}{\partial y} = \frac{f(y_{k+1}) - f(y_{k-1})}{2\Delta y} \tag{54}$$

The derivative approximations in (54) represent second order central differences.

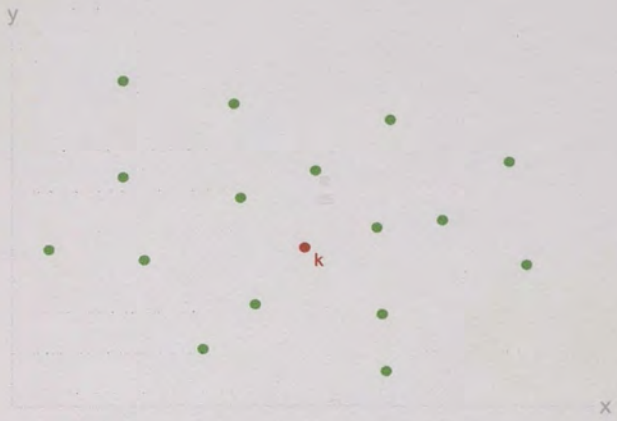

Figure 7. Irregular point distribution.

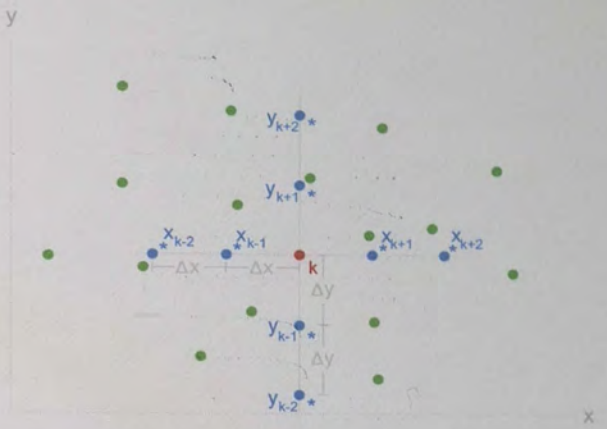

Figure 8. Virtual node distribution.

Any finite-difference stencil can be used, such as forward or backward differences. Let the values of the field variable at any general virtual points separated by any spacing vector $\vec{d}$ be:

$$f_{k-2} = f(\vec{x} - 2\vec{d}),\ f_{k-1} = f(\vec{x} - \vec{d}) \ldots f_{k+1} = f(\vec{x} + \vec{d}),\ f_{k+2} = f(\vec{x} + 2\vec{d}) \tag{55}$$

We can then apply a radial-basis function expansion at each of the virtual locations using equation (48), selecting other points in the vicinity of the data center to act as influence points to aid in the interpolation of the field variable at the virtual nodes. Note that all virtual nodes for a given data center use the same influence topology. Performing collocation with all of the influence points for each of the virtual nodes and following the procedure described in the previous subsection, we can obtain interpolation vectors for each of the virtual nodes, which are given by (53), which is repeated here for convenience.

$$(\Phi_k) = (\chi_k)^T [\chi]^{-1} \tag{53}$$

The value of the field variable at each of the virtual nodes is then given by:

$$f_v = (\Phi_v)^T (f) \quad v = k-2, k-1, \ldots, k+2 \tag{56}$$

where (f) is a vector composed of all the known values of the field variable in the domain. Substituting (56) into the standard central-difference formula, we can obtain an expression for the derivative of f with respect to any direction $s = \vec{d} / \Delta s$, where $\Delta s = \left|\vec{d}\right|$ as:

$$\left.\frac{\partial f}{\partial s}\right|_k = \frac{f_{k+1} - f_{k-1}}{2\Delta s}$$

$$\left.\frac{\partial f}{\partial s}\right|_k = \frac{(\Phi_{k+1})^T (f) - (\Phi_{k-1})^T (f)}{2\Delta s}$$

$$\left.\frac{\partial f}{\partial s}\right|_k = \left[\frac{(\Phi_{k+1}) - (\Phi_{k-1})}{2\Delta s}\right]^T (f)$$

$$\left.\frac{\partial f}{\partial s}\right|_k = (df_k)^T (f) \tag{57}$$

where (df_k) is a vector representing the first derivative operator with respect to the direction s, and can be pre-computed as:

$$(df_k) = \frac{(\Phi_{k+1}) - (\Phi_{k-1})}{2\Delta s} \tag{58}$$

Vectors similar to (58) can be derived for any finite-difference stencil of any order.

2.6 Upwinding

The convective acceleration term in the incompressible Navier-Stokes equations must be treated with care. As discussed in Section 2.3.1, this term is given by:

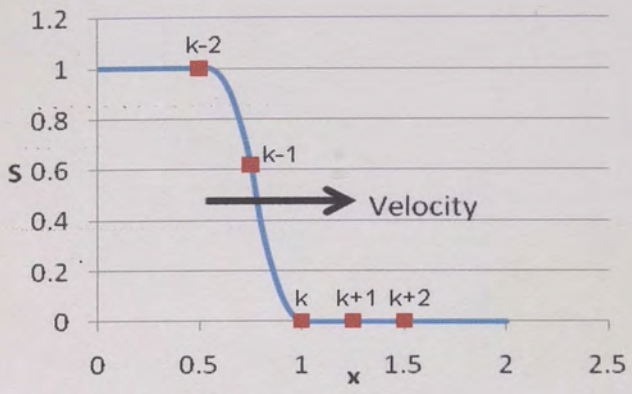

Figure 9. Distribution at t = 0.

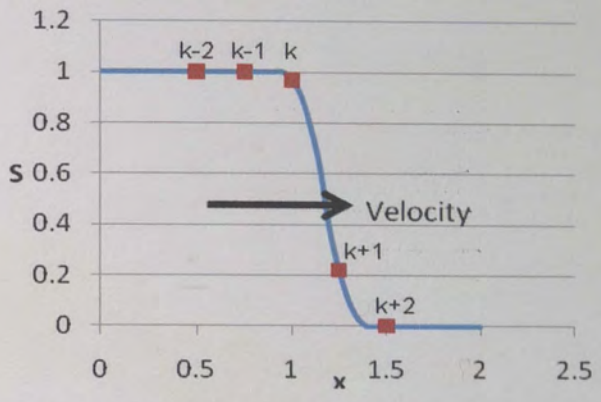

Figure 10. Distribution at t = Δt.

$$V_x \frac{\partial \vec{V}}{\partial x} + V_y \frac{\partial \vec{V}}{\partial y} \tag{9}$$

To illustrate the need for upwinding, consider a general one-dimensional advection equation for a scalar property S. The property is transported according to:

$$\frac{\partial S}{\partial t} = V_x \frac{\partial S}{\partial x} \tag{59}$$

Imagine that the spatial distribution of S is that shown in Figure 9 at time t = 0. At a time Δt, the velocity will have transported S over some distance, as shown in Figure 10. Consider point k in Figure 9. Using a first order forward difference for discretization of the time derivative in (59), we obtain an expression for S at point k at time Δt in terms of S at point k at time 0.

$$S_k(\Delta t) = S_k(0) + \Delta t V_x \left.\frac{\partial S}{\partial x}\right|_{t=0} \tag{60}$$

Notice that the derivative of S with respect to x is evaluated at time t = 0. Examining Figure 10, we see that $S_k(\Delta t) = 0.97$. In order to evaluate the spatial derivative $\left.\frac{\partial S}{\partial x}\right|_{t=0}$ we may use either a forward, backward, or central difference

approximation.

$$\left.\frac{\partial S}{\partial x}\right|_{t=0} = \frac{S_{k+1} - S_k}{\Delta x} \qquad \text{forward difference} \tag{61}$$

$$\left.\frac{\partial S}{\partial x}\right|_{t=0} = \frac{S_{k+1} - S_{k-1}}{2\Delta x} \qquad \text{central difference} \tag{62}$$

$$\left.\frac{\partial S}{\partial x}\right|_{t=0} = \frac{S_k - S_{k-1}}{\Delta x} \qquad \text{backward difference} \tag{63}$$

Table 1 shows the values of the derivatives of S at point k and time t = 0 obtained using equations (61) through (63), and the corresponding values of $S_k(\Delta t)$ calculated using these derivative values in equation (60). The backward difference approximation yields an $S_k(\Delta t)$ value that is closest to the actual value. This is because the information in this distribution is propagating from the left due to the positive advection velocity.

Table 1. Derivative approximations.

Finite-Difference Stencil	$\left.\frac{\partial S}{\partial x}\right\|_{t=0}$	$S_k(\Delta t)$
$\frac{S_{k+1} - S_k}{\Delta x}$ (forward)	0	0
$\frac{S_{k+1} - S_{k-1}}{2\Delta x}$ (central)	-1.24	0.496
$\frac{S_k - S_{k-1}}{\Delta x}$ (backward)	-2.48	0.992

For a general field variable f being advected by a general velocity field $\vec{V}$, the derivative operators used to calculate the spatial derivatives of f in each of the coordinate directions can be upwinded using the components of $\vec{V}$ in those coordinate directions. In the solver presented here, several spatial derivative operators in each coordinate direction are pre-calculated for each data center using forward, backward,

and central differencing.

For example, for a point k, three x-derivative operators are pre-computed using the virtual finite-differencing method discussed in Section 2.5:

$$(\delta for_x)_k = \frac{(\Phi_{x(k+1)}) - (\Phi_{x(k)})}{\Delta x} \tag{64}$$

$$(\delta cen_x)_k = \frac{(\Phi_{x(k+1)}) - (\Phi_{x(k-1)})}{2\Delta x} \tag{65}$$

$$(\delta back_x)_k = \frac{(\Phi_{x(k)}) - (\Phi_{x(k-1)})}{\Delta x} \tag{66}$$

Three y-derivative operators are computed in a similar fashion. Although the operators shown here are first-order, any order of finite-differencing can be used. The solver presented here is capable of up to second-order derivative approximations for forward and backward differencing, and up to fourth-order approximations for central differencing.

When the derivative of a field variable is required in order to transport the field variable with the velocity of the fluid, the derivative operator used is determined using the following upwinding scheme:

$$(\delta_x)_k = \begin{cases} (\delta for_x)_k & if\ V_x < 0 \\ (\delta cen_x)_k & if\ V_x = 0 \\ (\delta back_x)_k & if\ V_x > 0 \end{cases} \tag{67}$$

Upwinding in the y-direction is performed analogously. Upwinding is also used in the iterative solution of the advection-diffusion equations for pressure (equations (27) and (30)), where the advection velocity used for upwinding is $(\frac{\nabla \rho}{\rho})$.

2.7 Parallelization

Parallelization of the solution process for implementation on the GPU requires the segmentation of the domain into groups of nodes, or segments. The segmentation of the nodes is not a trivial process. In order for the solution to operate efficiently in parallel, each segment should contain nodes that are within close physical proximity to one another, ensuring memory and communication requirements are kept to a minimum for each segment. Additionally, efficient load balancing will occur when each segment contains approximately the same number of nodes. Communication between segments occurs along the boundaries of the segments, and so the ratio of boundary to interior points should be minimized.

Recall from Section 1.4 that the GPU operates most effectively when block sizes are multiples of 16. Segmentation will be performed such that each segment produced is directly mapped to a thread block on the GPU, so each segment should contain some number of points n that is a multiple of 16.

Segmentation is accomplished using a recursive bisection algorithm [61], initialized using the entire domain as the first segment, the x-direction as the initial direction, and the desired number of segments equal to p. The algorithm begins by dividing the domain into two segments using some heuristic to determine the optimal layout of the segments. Each of these segments is then recursively split using the same heuristic until the desired number of segments has been generated. After the initial bisection of the domain, splitting of segments is performed either by bisection if the number of segments left to be generated p is a multiple of two, or by splitting the

segment into p segments if p is not a multiple of two. The heuristic used in this project splits each segment according to the positions of the nodes in alternating coordinate directions. For example, if the first cut is specified to be in the x-direction, then all of the points in the domain are sorted according to their x-coordinates, and then the domain is split into two segments by placing all the points to the left of the median in the left segment, and all the points to the right of the median in the right segment. Each of these segments is then recursively split beginning in the y-direction, and then alternating between cuts in the x and y directions. Figure 11 shows the flow chart for this algorithm.

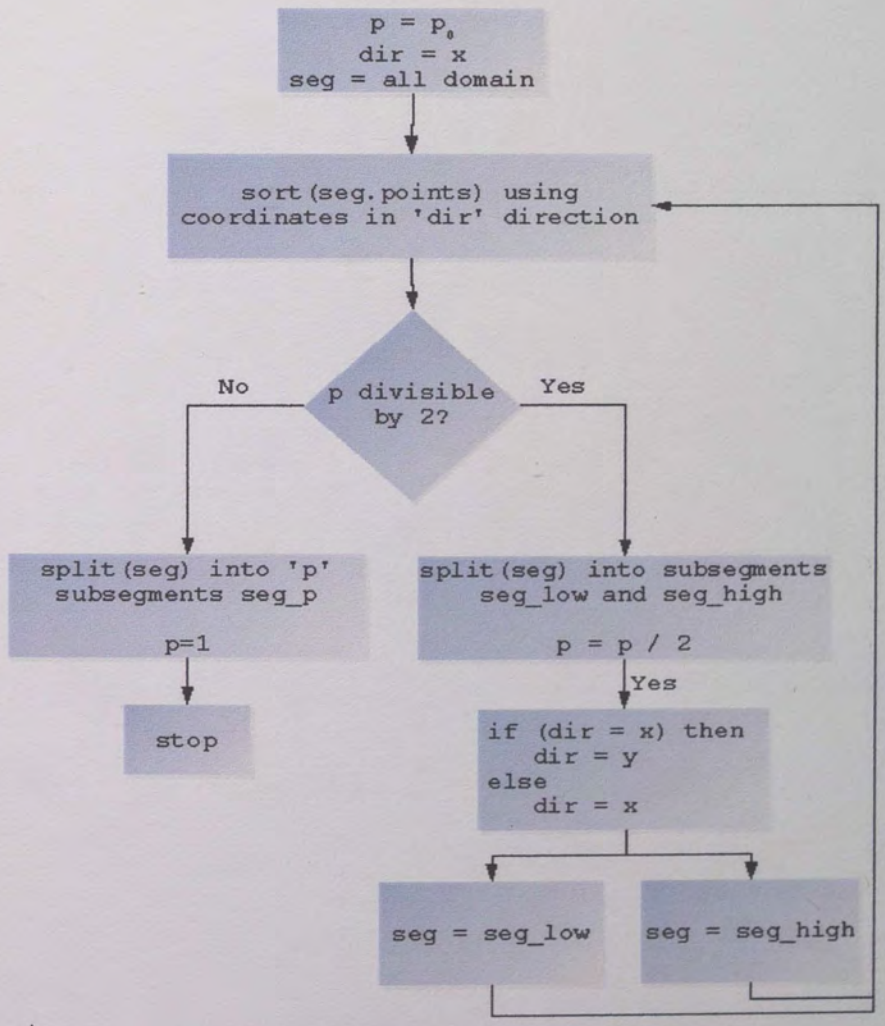

Figure 11. Segmentation algorithm flow chart.

Figure 12 provides a detailed example of the segmentation algorithm on a simple domain. In this example, the target number of segments is p = 4. Segment 0, which consists of the entire domain, is first split according to the x-coordinates of the points in the segment to form Segment 1. At this level, p = 4 / 2 = 2. Segment 1 is then split according to the y-coordinates of the points in the segment, forming Segment 2. At this level, p = 2 / 2 = 1. Since p is no longer divisible by 2, Segment 2 is split into 'p'

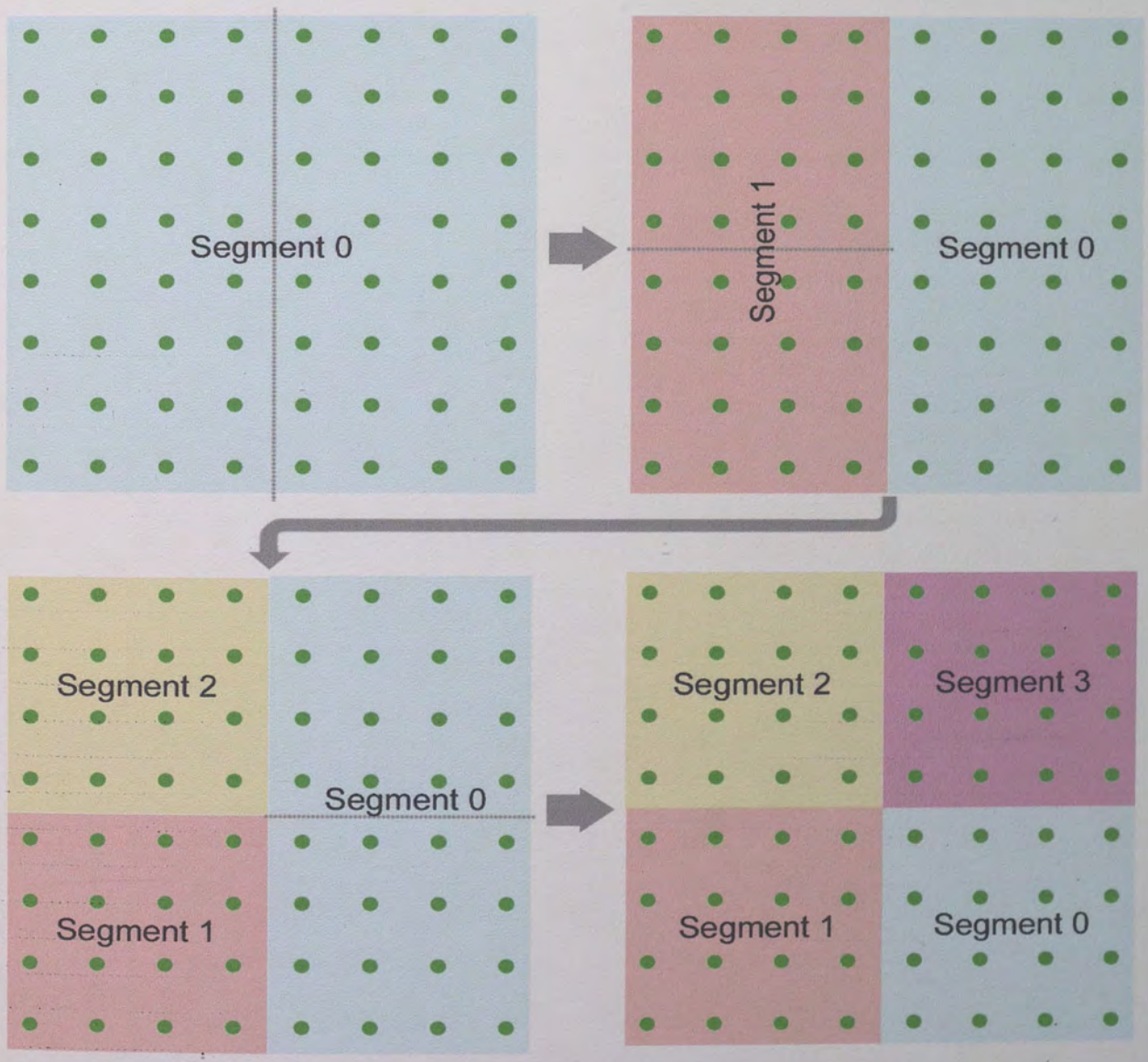

Figure 12. Example of segmentation procedure.

segments, but in this case p = 1, so Segment 2 is simply retained. The same applies to Segment 1. Segment 0 is then divided according to the y-coordinates of the points in the segment, forming Segment 3, and again p = 2 / 2 = 1 for Segments 3 and 0 so these segments are retained.

3 Implementation

In this chapter the implementation of the fluid solver will be discussed. First, performance profiling results for the serial code will be discussed. Profiling was used to determine which routines in the solver were bottlenecks; these routines became the targets for GPU-acceleration. Next, the graphics hardware used to implement the GPU-acceleration will be discussed, including how the hardware relates to software optimization issues. Finally, the segmentation algorithm used to map the problem to the GPU will be detailed.

3.1 Profiling of Serial Code

Two versions of the fluid solver were programmed, one that was entirely serial and executed solely on the CPU, and one that was GPU-accelerated that had portions of the code ported to the GPU. In order to determine which routines in the solver should be ported to the GPU, the serial code was analyzed for bottlenecks.

In order to test the serial code, two problems were run on four grid sizes each, and the execution times of each of the primary subroutines in the solver were recorded. Each solution was run for 100 time steps, and the execution times were recorded every 10 time steps. The average execution times across all of the problems run were then calculated for each of the primary subroutines. These average times were normalized according to the average time step for all of the problems, and the results are shown in Figure 13.

The solution of the Poisson equation for Helmholtz potential (equation (22)) and

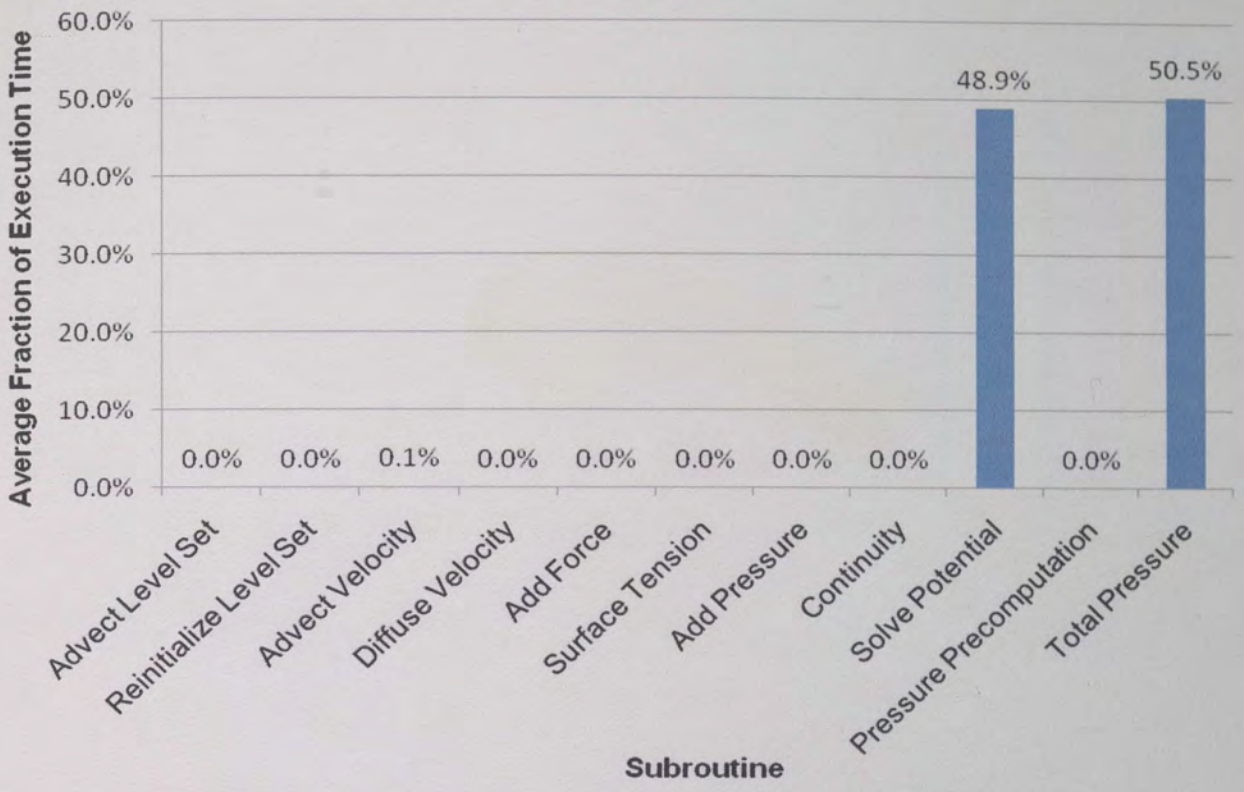

Figure 13. Average fraction of execution time for each subroutine.

the solution of the advection-diffusion equation for total pressure (equation (27)) were found to take up 99% of the total execution time of each time step of the solver. These were the two subroutines that were targeted for GPU-acceleration.

3.2 Software Implementation

The first stage of software implementation consisted of programming the serial solver in C++. Next, a graphical pre-processor was programmed in C# to facilitate creation of problem geometry and initial conditions. Finally, the subroutines chosen for GPU-acceleration were implemented for parallel execution on the GPU using CUDA.

3.2.1 Serial Solver

The core of the serial solver was structured in a procedural fashion. Four main

pre-processing routines were programmed, which are described in Table 2, and within the main iteration routine ten main subroutines were called, which are described in Table 3.

While object-oriented programming was used for reading and writing files, segmentation, and code profiling, the choice to write the bulk of the solver using a procedural programming paradigm was made because of its ease of translation to the GPU. CUDA does not support classes, and data is most efficiently transferred to and from the GPU as raw sequential arrays in memory. While the use of classes to represent points or vector operators might have eased the programming of the serial code, using an object-oriented design for all of the serial code would have hindered tight integration of CUDA kernels and decreased the performance of the solver overall.

The solver graphical-user-interface (GUI), or shell, was written in C# using a strongly object-oriented approach. The shell is composed of two main workspaces: the pre-processing workspace and the batch-execution workspace, shown in Figure 14 and Figure 15, respectively.

The pre-processing workspace allows the user to generate problem geometry, specify initial and boundary conditions, draw regions of the domain occupied by the two fluids being solved for, and provide a detailed specification of the way the solver will execute. There is a large drawing space in the pre-processing workspace that allows the user to use various tools to create the nodal distribution, create geometric primitives representing the two fluid regions, and draw vectors specifying the initial velocity and force fields.

Table 2. Main pre-processing routines.

Function	Description
ReadInput()	Reads solution parameters regarding input and output, order of derivative approximations, and temporal discretization from an input file.
Initialize()	Reads problem geometry from file and initializes all arrays used for storage of field variables, operators, connectivity, segmentation, and nodal positions.
OperatorsConnectivity()	Reads from file or generates derivative operators and influence point topologies (connectivity) for each of the nodes in the domain. If operators and connectivity were generated, they are saved to file.
Segmentation()	Reads segmentation data from file or performs segmentation of the domain if the problem is going to run with GPU acceleration. If segmentation data was generated it is written to file. For purely serial cases this routine is ignored.

Table 3. Primary iteration subroutines.

Function	Description
Advect(scalar_field *f*, vector_field *v*)	Transports the scalar field *f* according to the velocity field *v* using equation (9). Utilizes upwinding to determine which derivative operator to use.
ReinitializeLevelSet()	Reinitializes the level set to a signed distance function using equation (44).
Diffuse()	Diffuses the velocity according to equation (8).
AddForce()	Adds force field and gravity to velocity field.
SurfaceTension()	Calculates surface tension effects according to equation (20).
AddPressure()	Calculates the gradient of the pressure field and adds pressure effects to the velocity field.
Continuity()	Ensures that continuity is satisfied at the boundaries.
SolvePoisson()	Solves the Poisson equation for Helmholtz potential, equation (22), using Jacobi iteration.
PressurePrecomputation()	Pre-computes vector operators that when applied to the pressure field is equivalent to evaluating the left-hand-side of equation (27).
SolveADE()	Solves the advection-diffusion equation for pressure, equation (27), using Jacobi iteration.

Both the solution of the Poisson equation and the solution of the advection-diffusion equation implement Jacobi iteration to solve the sparse matrix system formed by discretization of the differential operators.

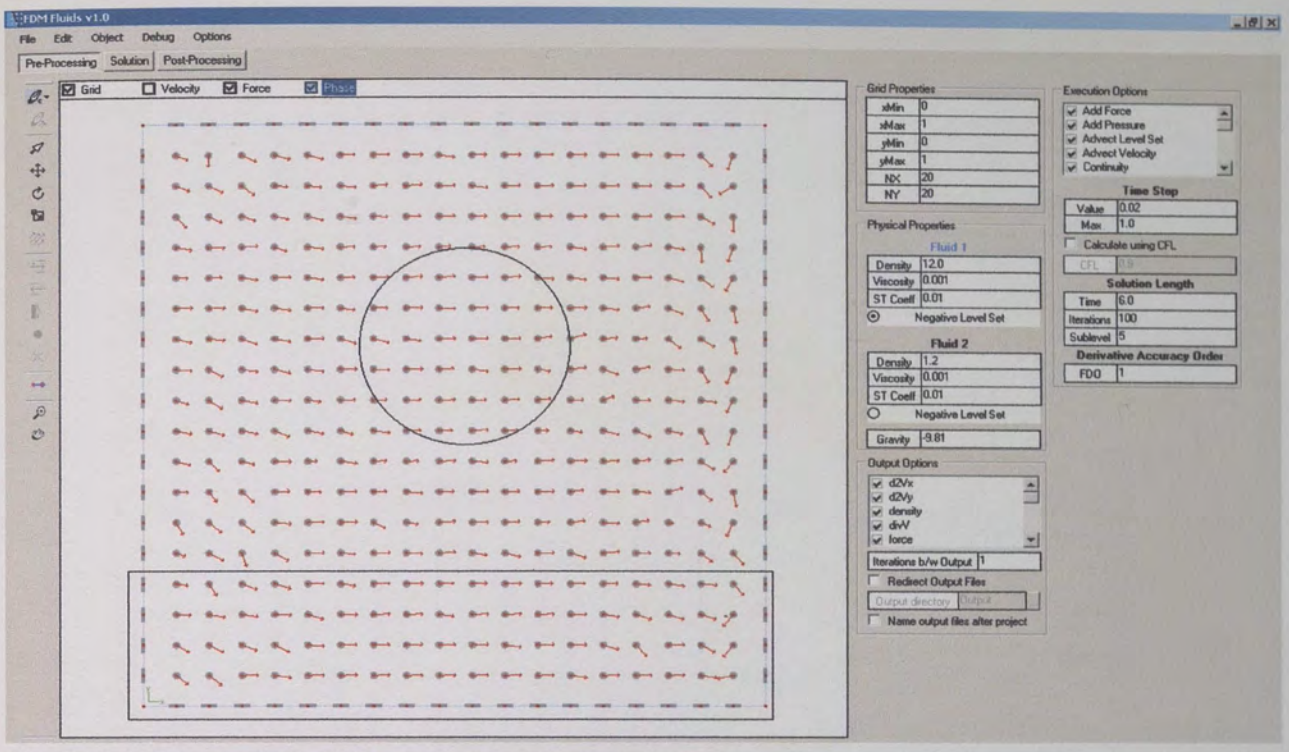

Figure 14. Screenshot of the pre-processing workspace.

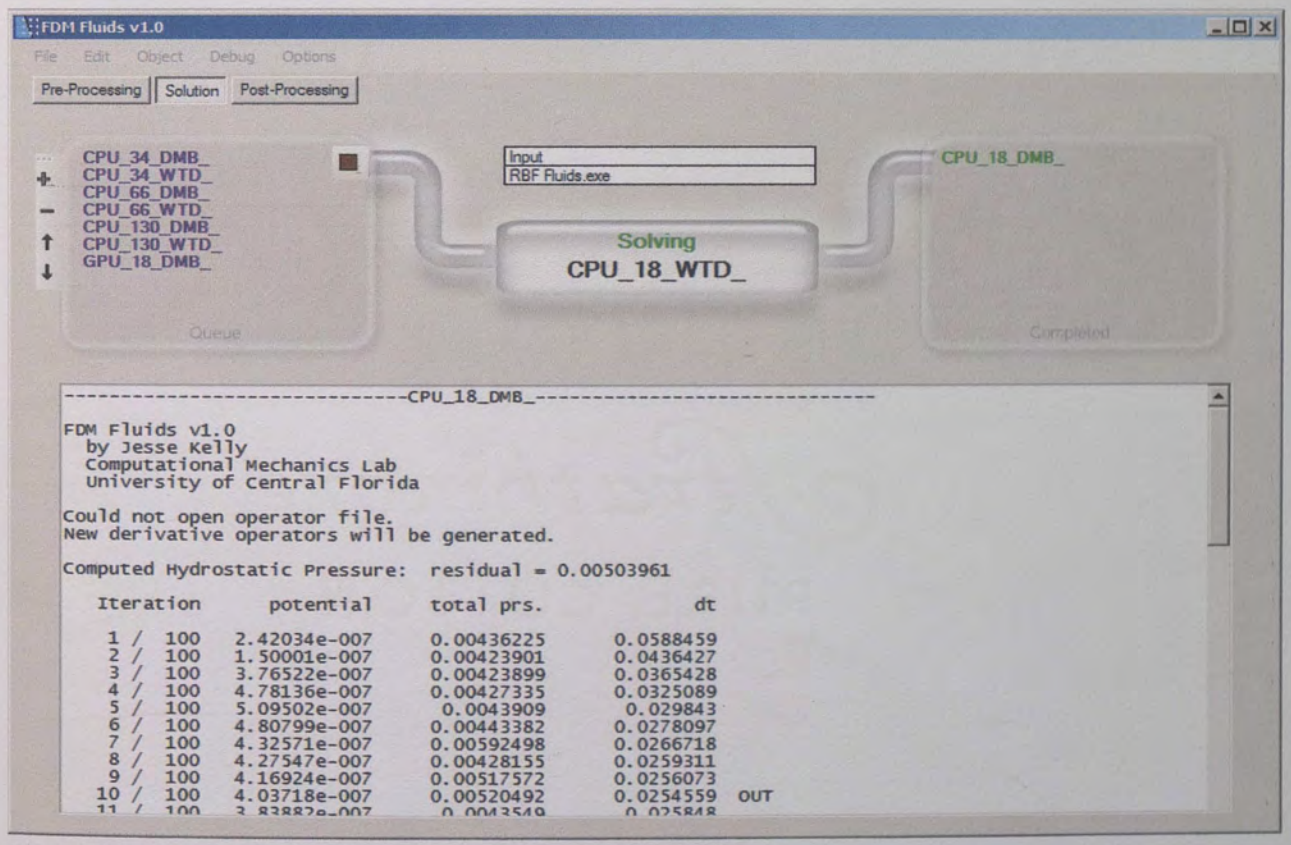

Figure 15. Screenshot of the batch-processing workspace.

3.2.2 *Jacobi Iteration*

The SolvePoisson() and SolveADE() routines both implement Jacobi iteration for the solution of the Poisson and advection-diffusion equations, respectively. Consider an equation of the general form:

$$\Psi x = \beta \tag{68}$$

where Ψ is some linear differential operator, x is some field variable, and β is some constant right-hand side. This equation applies at every interior data point within a given domain. At a specific internal data center *k*, this equation takes the form:

$$(\Psi)_k^T (x)_k = \beta_k \tag{69}$$

where $(\Psi)_k$ is the RBF-enhanced finite difference operator version of Ψ for node *k*, $(x)_k$ is a vector containing the values of the field variable at each of the influence points for node *k*, and β_k is some right-hand side value corresponding to node *k*. The operators for all of the interior data centers can be assembled into the following linear system of equations, where the rows prior to x_0 correspond to boundary nodes:

$$\begin{bmatrix} 0 \\ 0 \\ \vdots \\ (\Psi e)_0^T \\ (\Psi e)_1^T \\ \vdots \\ (\Psi e)_{NI}^T \end{bmatrix} \begin{pmatrix} x_{B0} \\ x_{B1} \\ \vdots \\ x_0 \\ x_1 \\ \vdots \\ x_{NI} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{NI} \end{pmatrix} \tag{70}$$

where $(\Psi e)_k$ corresponds to the k^{th} expanded differential operator, expanded meaning that each expanded operator is composed of *NI* elements, where *NI* is the total number

of interior data centers. Each i^{th} element in $(\Psi e)_k$ corresponds to the i^{th} data center in the distribution, and for data centers whose influence topologies do not include the i^{th} data center, that element of $(\Psi e)_k$ is zero. For example, say there are 10 total points, and the influence topology and differential operator of the 3rd interior data center are given by:

$$(\iota)_2^T = (2 \quad 5 \quad 3 \quad 9) \tag{71}$$

$$(\Psi)_2^T = (\Psi_{2,0} \quad \Psi_{2,1} \quad \Psi_{2,2} \quad \Psi_{2,3}) \tag{72}$$

The influence topology vector $(\iota)_2$ stores the indices of the nodes that are in the influence topology of the 3rd node. Assembling a vector of influence values using the value of the field variable x at these indices will result in $(x)_2$. So $(\Psi)_2^T(x)_3$ will evaluate to:

$$(\Psi)_2^T(x)_2 = \Psi_{2,0} \cdot x_2 + \Psi_{2,1} \cdot x_5 + \Psi_{2,2} \cdot x_3 + \Psi_{2,3} \cdot x_9 \tag{73}$$

The first node in the influence topology for any given data center is always the data center itself. Note that the indices of the data centers are stored in zero-offset format, which is why the subscript 2 appears in equations (71) thru (73) even though the 3rd interior data center is being considered. The expanded form of the differential operator $(\Psi e)_2$ is given by:

$$\begin{array}{rcccccccccc} (\Psi e)_2^T = (0 & | & 0 & | & \Psi_{2,0} & | & \Psi_{2,2} & | & 0 & | & \Psi_{2,1} & | & 0 & | & 0 & | & 0 & | & \Psi_{2,3}) \\ \text{i: } 0 & | & 1 & | & 2 & | & 3 & | & 4 & | & 5 & | & 6 & | & 7 & | & 8 & | & 9 \end{array} \tag{74}$$

The numbers below each element in (74) correspond to the global indices of the points whose field variable values each element multiplies. (74) would appear in the

matrix in (70) as shown in (75).

$$\begin{bmatrix} & & & & \vdots & & & & & \\ 0 & 0 & \Psi_{2,0} & \Psi_{2,2} & 0 & \Psi_{2,1} & 0 & 0 & 0 & \Psi_{2,3} \\ & & & & \vdots & & & & & \end{bmatrix} \begin{pmatrix} \vdots \\ x_0 \\ \vdots \\ x_{NI} \end{pmatrix} = \begin{pmatrix} \vdots \\ \beta_0 \\ \vdots \\ \beta_{NI} \end{pmatrix} \tag{75}$$

The linear system (70) is solved using Jacobi iteration, which is an iterative solution technique. At any iteration *k*, the estimate for any unknown x_j is calculated using the equation:

$$x_j^{k+1} = \frac{\beta_j - \sum_{i \neq j} \left(\Psi e_j\right)_i x_i^k}{\left(\Psi e_j\right)_j} \tag{76}$$

In practice, the sparse matrix in (70) is not assembled. The actual code in the solver simply circumvents all of the zero elements in the matrix by using equation (77) in place of equation (76).

$$x_j^{k+1} = \frac{\beta_j - \sum_{i=1}^{NF} \left(\Psi_j\right)_i \left(x_j\right)_i}{\left(\Psi_j\right)_0} \tag{77}$$

where NF is the number of influence points in node j's influence topology.

Applying this framework to the solution of the Poisson equation for Helmoltz potential (22), we have:

$$\left(\Psi\right)_k = \left(\nabla^2\right)_k \quad \left(x\right)_k = \left(\varphi\right)_k \quad \left(\beta\right)_k = \nabla \cdot \vec{V}_k^* \tag{78}$$

Applying the framework to the solution of the advection-diffusion equation for pressure (27) and hydrostatic pressure (30), we have:

$$(\Psi)_k = \left[(\nabla^2)_k - (\frac{\nabla \rho_k}{\rho_k})(\nabla)_k \right] \quad (x)_k = (p)_k \tag{79}$$

$$(\beta)_k = -\rho_k [\nabla \cdot (\vec{V}_k \cdot \nabla)\vec{V}_k + \nabla \cdot (\nu \nabla^2 \vec{V}_k) - \nabla \cdot ST_k \text{ for total pressure} \tag{80}$$

$$(\beta)_k = 0 \text{ for hydrostatic pressure} \tag{81}$$

In order to solve (70), boundary conditions are required. The boundary conditions expressed in equations (23) and (24) for Helmholtz potential are implemented using equations (82) and (83) below, where (dn_k) is the boundary normal derivative operator.

$$\phi_k = \frac{-\sum_{i=1}^{NF} (dn_k)_i (\phi_k)_i}{(dn_k)_0} \quad \phi_k \in \text{walls and inlets} \tag{82}$$

$$x_k = 0 \quad x_k \in outlets \tag{83}$$

The boundary conditions expressed in equations (28) through (29) for total pressure are implemented using equations (84) through (86) below.

$$p_k = \frac{-\rho_k [\nabla \cdot (\vec{V}_k \cdot \nabla)\vec{V}_k + \nabla \cdot (\nu \nabla^2 \vec{V}_k)] \cdot \vec{n} - \sum_{i=1}^{NF} (dn_k)_i (p_k)_i}{(dn_k)_0} \quad p_k \in \text{walls} \tag{84}$$

$$p_k = 0 \quad p_k \in \text{outlets} \tag{85}$$

$$p_k = \hat{p}_k \quad p_k \in \text{inlets} \tag{86}$$

The boundary conditions for hydrostatic pressure expressed in equations (31) and (32) are implemented using equations (87) and (88) below.

$$p_k = \frac{-\rho_k g n_{k,y} - \sum_{i=1}^{NF} (dn_k)_i (p_k)_i}{(dn_k)_0} \quad p_k \in \text{walls} \tag{87}$$

$$p_k = 0 \quad p_k \in \text{reference height} \tag{88}$$

Note that the subscript *k* in equations (78) through (88) does not refer to the time step, but to the value of the field variables at the k^{th} data center. In the equations corresponding to Helmholtz potential, the velocity is evaluated at the intermediate time step '*'. In the equations corresponding to the pressure, the velocity is evaluated at time step *k+1*. The velocity boundary conditions given by equations (33), (34) and (35) are implemented in the same manner as the pressure boundary conditions.

Iteration can be stopped when the residual norm of (70) satisfies some convergence criteria, or when a prescribed number of iterations have been reached.

3.2.3 Boundary Continuity

The Continuity() subroutine in Table 3 enforces continuity at the boundaries. This is necessary when the discrete form of the Poisson equation for the Helmholtz potential does not explicitly satisfy continuity. Boundary continuity is enforced by summing the mass flux at the inlets and the outlets, calculating the net flux flowing into the domain, and correcting the inlet and outlet velocities so that the net flux becomes zero. This process ensures that equation (89) is satisfied.

$$\int \rho(\vec{V} \cdot \vec{n}) dA = 0 \qquad \vec{x} \in \text{outlets and outlets} \tag{89}$$

3.3 Hardware Implementation

NVIDIA GPUs are made up of a collection of multiprocessors, each of which

contains eight core processors. When a CUDA kernel is launched with a given execution configuration, thread blocks are mapped to multiprocessors on the device. Each multiprocessor can handle multiple thread blocks, and the number of thread blocks the multiprocessor is capable of handling is limited by the number of registers used and the amount of shared memory used by the kernel being executed. Within each thread block, threads are organized into warps. A warp is a physical collection of threads that executes simultaneously on the GPU; there is no abstraction for a warp in CUDA. Warp size is constant for a given device, and varies depending on the device used.

Each multiprocessor has access to several memory spaces, including global, local, shared, texture, and constant memory. Each of these memory spaces has advantages and disadvantages, outlined in Table 4 below. Global memory may be read or written to by both the code running on the computer that hosts the graphics card, called the host, and by the graphics device on which the CUDA kernels are being executed, called the device. Shared and local memory may only be accessed by the device, although the amount of shared memory allocated to each block is controlled by the host. Texture and constant memory are similar in that they may only be written to by the host, and may only be read by the device. Texture memory differs from constant memory in that it is available in a much larger amount on the device. The name "texture" memory derives

Table 4. Memory spaces on NVIDIA graphics hardware.

Memory Space	Location	Cached	Host Accessibility	Device Accessibility
Global	Off-chip	No	Read & write	Read & write
Local	Off-chip	No		Read & write
Shared	On-chip	No		Read & write
Texture	Off-chip	Yes	Write	Read
Constant	On-chip	Yes	Write	Read

from the texture data structures used in graphics programming, and refers to a one, two, or three-dimensional array of either integers or floats.

Global memory and local memory reside off the microprocessor chips (they are off-chip), while shared memory and constant memory are on-chip. On-chip memory accesses are much faster than off-chip, so the first optimization strategy that should be observed is the avoidance of global and local memory accesses in favor of shared and constant memory accesses. In the CUDA language abstraction, shared memory may only be shared amongst threads in a block, so a practical limitation of shared memory is its inability to be used to communicate data amongst threads of different blocks. For inter-block communication, global memory is the only choice. The second optimization strategy, then, is to store all data that must not be shared with other blocks in shared memory, and access global memory only when needed to communicate with other blocks. In order to minimize global memory access further, each block should perform operations on shared memory for as much time as possible before having to write results to global memory to be shared with other blocks. This forms the third optimization strategy, which may be re-stated in terms of arithmetic intensity, or the percentage of total computational time that is used for arithmetic operations versus memory operations: maximize arithmetic intensity.

Texture memory is another alternative to global memory for data that only needs to be read by the device. Texture memory accesses are much faster than global memory accesses because they are cached. The texture memory space is also optimized for two-dimensional spatial locality, meaning memory operations that access texture elements

that are close to one another (i.e. have similar indices) will be coalesced. The fourth optimization strategy is to store global data in textures whenever it can remain constant throughout the execution of a single kernel.

Coalescence on the GPU refers to the execution of multiple memory operations simultaneously. Coalescence only occurs under certain conditions, which differ for each memory space, and which also depend on the device used. For global memory operations, whether or not a specific operation is coalesced depends on the behavior of a half-warp of threads requesting the memory operation. Recall that a warp is a physical organizational unit of threads. The GPU used in this project was an NVIDIA GeForce 9800 GT. The specifications for this device as relate to the current discussion are shown in Table 5.

For the GeForce 9800 GT, global memory coalescence occurs when the following three conditions are met:

1. All threads in a half-warp (which is 16 threads) must access words composed of 4, 8, or 16 bytes.
2. All of the words read must reside in a segment of global memory that is equal in

Table 5. GeForce 9800 GT specifications.

Multiprocessor count	14
Maximum number of threads per block	512 threads
Maximum number of blocks per grid	65535 blocks
Warp size	32 threads
Register count per multiprocessor	8192
Shared memory per multiprocessor	16 KB
Constant memory available (total)	64 KB
Local memory available (total)	16 KB
Global/texture memory available (total)	512 MB
Maximum active blocks per multiprocessor	8
Maximum active warps per multiprocessor	24
Maximum active threads per multiprocessor	768

size to the total amount of memory requested.

3. The threads must access the memory sequentially, i.e. the i^{th} thread must access the i^{th} word.

When global memory accesses are coalesced, memory bandwidth is significantly increased (up to an order of magnitude). Thus, the fifth optimization strategy is to maximize the amount of memory operations that are coalesced by adhering to the three conditions outlined above whenever possible. The first condition is most effectively met by specifying an execution configuration in which the number of threads per block is a multiple of 16. The second and third conditions are usually handled by the way in which kernels are written. Memory accesses should be sequential whenever possible. A consequence of this is that the GPU does not handle random access of global memory well. Sometimes the data access patterns necessary for a specific kernel must be random, in which case the use of texture memory may significantly increase performance.

As discussed, texture memory reads are already significantly cheaper than global memory reads because they are cached. Another benefit of reading from texture memory is that reads can be coalesced even when threads read data in a random, non-sequential way, provided there is some spatial locality among the memory accesses. As mentioned, the main disadvantage of texture memory is its inability to be written to by the device. Thus, it is best suited for large data sets that are able to remain constant throughout the entire execution of a kernel.

The optimal memory space for large, non-constant amounts of data is the shared

memory space. Shared memory resides on-chip and is cached, so shared memory latency is much lower than that of global, local, and texture memories. Optimizing shared memory performance is similar to global memory performance in that alignment of memory accesses is key. On the GeForce 9800 GT, there is 16 KB of shared memory available per multiprocessor, organized into 16 memory banks per multiprocessor. Memory is stored such that two 4-byte words with adjacent addresses (i.e. $address_0$ and $address_0 + 4$) are stored in adjacent banks. This is illustrated in Figure 16 below.

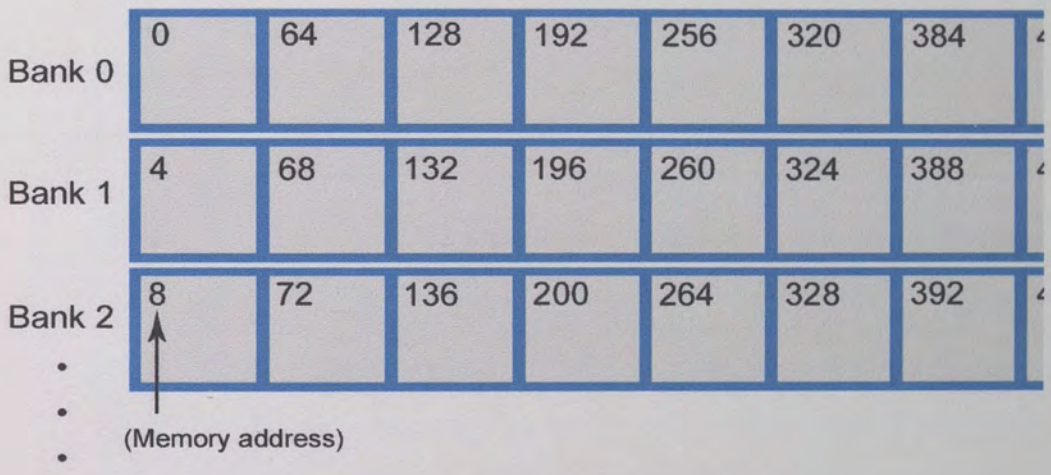

Figure 16. Shared memory layout.

In order for a shared memory access to be coalesced, each thread in a half-warp must access a different memory bank. The memory accesses do not necessarily have to be sequential, so a certain level of random access is allowed for shared memory before performance is hindered. If memory access is sequential in the case of 4-byte words, no thread will access the same bank. As long as no data type that is larger or smaller than 4 bytes is used, sequential shared memory accesses are guaranteed to be coalesced into one memory operation. For non-sequential requests, as long as two threads do not access the same address in shared memory, and as long as the range of data accessed is

no larger than 16 4-byte words (since memory operations are performed by the half-warp), non-sequential operations are guaranteed to be coalesced into a single operation.

In situations where two or more threads in a half-warp access the same address in shared memory, the data at that address can be broadcast to each of the requesting threads, resulting in a single memory operation. Any remaining threads that do not require this data to be broadcast to them then have their memory operations serviced according to the coalescing conditions for shared memory already discussed. In the event of two or more threads accessing different data from the same bank, such a memory operation must be performed serially, with one memory operation performed per thread, reducing bandwidth. The sixth optimization strategy is the allowance of sequential or non-bank-conflicting memory accesses from shared memory whenever possible.

The last two memory spaces to be discussed, local and constant memory, are not as crucial as global, texture, and shared memory. As Table 5 shows, there is only 64 KB of constant memory available on the entire device, and only 16 KB of local memory available. Constant memory reads are only as expensive as reading from a register as long as all threads in a half-warp read from the same constant memory address. Constant memory is useful for storing scalar execution parameters required by all the threads in the grid. Local memory is non-cached, so it is as expensive as global memory, and it is not directly allocated by the programmer. Automatic variables in kernels may be relegated to local memory if the CUDA compiler determines that there is no room for them in on-chip memory. In order to avoid having to access local memory, the amount of

memory used by automatic variables in kernels should be minimized. The limitations of the last two memory spaces are addressed in the first optimization strategy, which has already been presented and which will be repeated shortly.

The six optimization strategies for CUDA programming presented in this section are repeated here for emphasis and convenience:

1. Avoid the use of global and local memory in favor of shared and constant memory.
2. Store all data that does not need to be shared with other blocks in shared memory, and access global memory only when needed to communicate with other blocks.
3. Maximize arithmetic intensity.
4. Store global data in textures whenever it can remain constant throughout the execution of a single kernel.
5. Maximize the number of global memory operations that are coalesced by satisfying these three conditions whenever possible:
    a. All threads in a half-warp should access 4, 8, or 16-byte words.
    b. All words accessed should reside in a segment of global memory that is equal in size to the total size of the memory requested.
    c. Threads should access memory sequentially.
6. Access shared memory sequentially or in a way that does not cause the same bank to be accessed by two different threads.

The information provided in this section has been summarized from the CUDA

programming guide published by NVIDIA [62].

3.4 Optimization of Execution Configuration

In order to execute a kernel on the GPU, an execution configuration is required at runtime. An execution configuration is an abstraction of the way the execution of a kernel is mapped to the actual graphics hardware. The execution configuration includes the block dimensions and the grid dimensions, which together specify the total number of threads executed (see Section 1.4). The execution configuration is limited by the number of registers available per multiprocessor, which we shall denote R_{avail}, and depends on the specific graphics card being used. As shown in Table 5, $R_{avail} = 8192$ for the GeForce 9800 GT. Execution configuration is also limited by the amount of shared memory available per multiprocessor, which we shall denote S_{avail}, and which is equal to 16 KB as shown by Table 5. In order for a kernel launch to be successful, the execution configuration must be such that the number of registers and amount of shared memory required for all blocks executing on a single multiprocessor is below R_{avail} and S_{avail}, respectively. We shall denote the number of registers per block R and the amount of shared memory per block S for a given kernel that executes using B blocks on M multiprocessors. A necessary condition for execution of the kernel is then given by:

$$(\frac{S \cdot B}{M} \leq S_{avail}) \wedge (\frac{R \cdot B}{M} \leq R_{avail}) \tag{90}$$

For the GeForce 9800 GT, M = 14, so in order for a kernel launch to be successful, the relation given by (90) becomes:

$$(S \cdot B \leq 224 \text{ KB}) \wedge (R \cdot B \leq 114688)$$

$$(S \leq \frac{224}{B} \text{KB}) \wedge (R \leq \frac{114688}{B}) \tag{91}$$

So the amount of shared memory allocated per block must be no larger than 224 kilobytes divided by the number of blocks in the execution configuration, and the number of registers required per block must be no more than 114688 divided by the number of blocks in the execution configuration.

Each multiprocessor on the GPU hides memory latency by rotating the blocks that are being actively executed, running blocks whose memory accesses have been served while those that have incomplete memory operations wait to become active. Thus, it is good to have at least twice as many blocks as there are multiprocessors available on the device in order to mask memory access overhead. As Table 5 shows, the maximum number of threads per block is 512, leading to the following constraint:

$$T \leq 512 \tag{92}$$

As Table 5 shows, the maximum number of warps that can actively execute on a single multiprocessor for the GeForce 9800 GT is 24, which corresponds to 768 active threads, and the maximum number of blocks that may concurrently execute on a multiprocessor is 8. Consider an execution configuration in which there are T threads per B blocks, yielding a total of T·B threads in the entire grid. In order to maximize the available resources of the GPU, we can calculate the theoretically optimal execution configuration as:

$$\frac{B}{M} = 8 \Rightarrow B = 8 \cdot 14 = 112 \text{ blocks per grid}$$

$$T = \frac{768}{8} = 96 \text{ threads per block}$$

$$T \cdot B = 10752 \text{ threads per grid}$$

For general problems, however, the number of threads per grid will typically be prescribed and the optimal execution configuration can only be approximately satisfied in terms of either B or T. Consider the case where the total number of threads is specified as T_{total}, and the number of blocks, and thus the number of threads per block, is allowed to vary. Based on the discussion of memory coalescence in the previous section, T should be a multiple of 16 in order to facilitate increased memory bandwidth for both global and shared memory. Since T is a function of T_{total} and B, this means that B should also be a multiple of 16. Additionally, the amount of shared memory allocated per block must satisfy (91). The algorithm for selecting the optimal execution configuration for a given number of threads per grid is then as follows, and is shown in Figure 17.

1. Begin by setting T = 512, which will ensure that (92) is satisfied. Calculate B = T_{total} / T and round up to the nearest integer.
2. Calculate T = T_{total} / B and round up to the nearest multiple of 16.
3. Calculate B = T_{total} / T.
4. If B is not an integer, decrease T by 16 and return to Step 3.
5. Calculate S.
6. Let Z represent the initial satisfaction of (91). If Z is true, then continue until B is 112 or until (91) is no longer satisfied. If Z is false, continue until S satisfies (91) and, if possible, until B is greater than 112.

7. Increase B by 16 and return to Step 2.

The amount of shared memory per block S depends largely on the global data center distribution. In order to provide an estimate for S as a function of the number of threads per block T, S was recorded for several different selections of T for a regular data

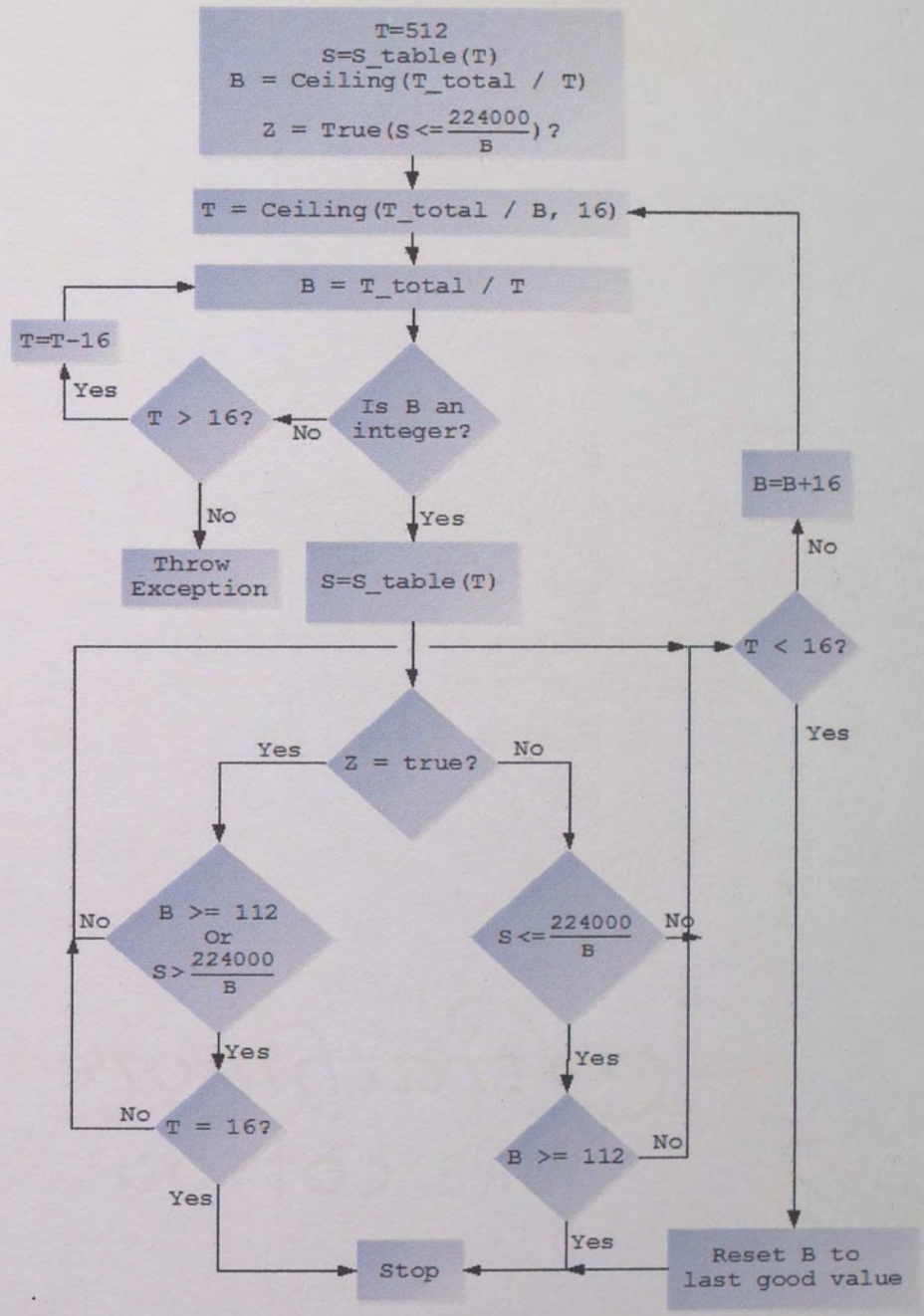

Figure 17. Flow chart for execution configuration optimization algorithm.

center distribution of 130 x 130 equally spaced data centers. The results are shown in Figure 18.

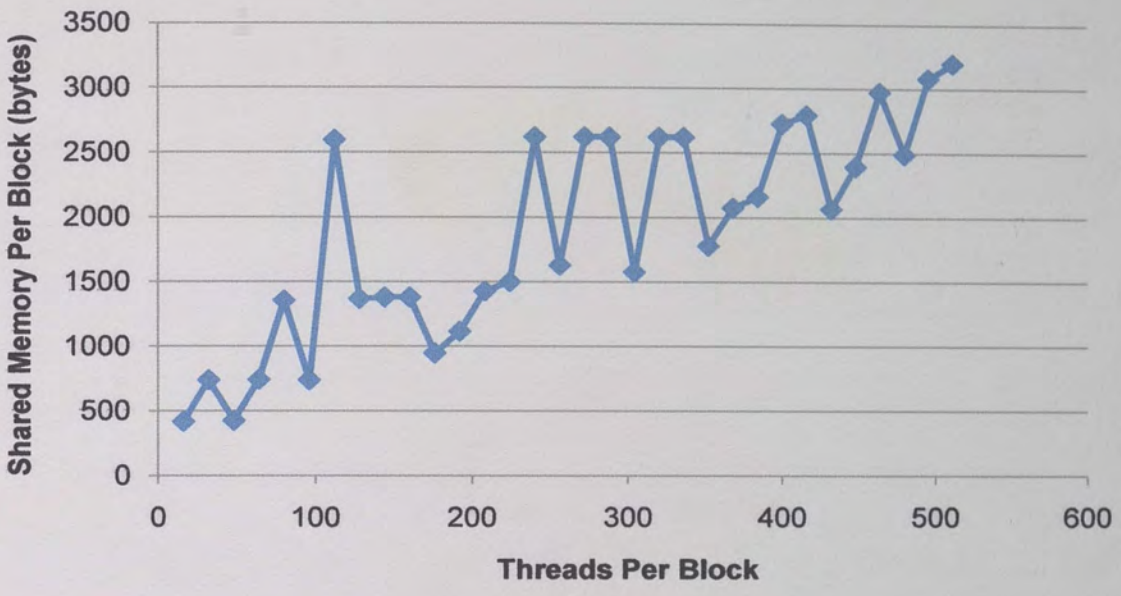

Figure 18. Shared memory requirements.

The number of threads per block was varied from 16 to 512 in multiples of 16, so no matter what the value of T in the algorithm presented in Figure 17, a value of S will be available through a table lookup.

Segmentation of the domain in the fluid solver presented here first begins with the execution configuration optimization discussed in this section, which determines the optimal number of thread blocks and number of threads per block for execution on the GPU. As previously discussed, the terms "segment" and "thread block" as applies to this project are interchangeable, so once the execution configuration has been determined the domain is segmented according to the procedure discussed in Section 2.7. The recursive bisection algorithm is started with p set equal to the number of thread blocks computed from the execution configuration optimization.

3.4.1 Mapping to the GPU

Since each thread block on the GPU references threads according to a local index, mapping is required to map thread indices within each block to the global indices of the data centers. Additionally, the influence topologies that are transferred to the GPU are mapped to local block indices so each block can make efficient use of shared memory rather than using global indices to reference global memory. Once segments have been constructed, the mapping process detailed below is applied to each segment.

First, the influence topologies of each of the nodes in a segment are compared. A unique list of indices that the nodes within the segment will reference is constructed. For example, if multiple nodes in a segment both reference node *n*, only one entry in the unique list exists for node *n*. In pseudo-code, this process looks like:

```
AssembleUniqueList(Segment s)
{
      s.UniqueList = empty
      For each node n in s
            For each node f in n.InfluencePoints
                  If (s.UniqueList.DoesNotContain(f))
                        s.UniqueList.Add(f)
                  End If
            End For
      End For
}
```

Once the unique list has been assembled for each segment, a new influence topology is generated for each point within the segment that maps the point's global influence topology to local segment indices. In pseudo-code, this process looks like:

```
AssembleLocalTopologies(Segment s)
{
      For each node n in s
            n.LocalInfluencePoints = empty
            For f=0 to n.InfluencePoints
                  G_idx = n.InfluencePoints(f)
                  For u=0 to s.UniqueList.Size
```

```
                        If (G_idx == s.UniqueList(u))
                            n.LocalInfluencePoints(f) = u
                        End If
                    End For
                End For
            End For
}
```

On the GPU, each thread uses its local influence point array to perform differential operations using the shared memory of the block.

For the solution of the Poisson and advection-diffusion equations, all data such as differential operators and boundary node identifiers is transferred to the GPU before iteration begins. At each time step, the right-hand sides of the equations and the boundary conditions are transferred to texture memory on the GPU, and the global unknown vectors are transferred to and from global memory on the GPU. Each thread block loads the nodal values it requires from global to shared memory, performs operations using the shared data, and then writes all values required by other thread blocks to global memory. Table 6 provides a summary of GPU data transfer and storage modes.

Table 6. Summary of GPU data transfer and storage modes.

CPU data	GPU data structure	Transfer
Right-hand side vector	1-D texture	Every time step (write)
Boundary conditions	1-D texture	Every time step (write)
Advection-diffusion operators	2-D texture	Every time step (write)
Normal derivative operators	2-D texture	Once (write)
Laplacian operators	2-D texture	Once (write)
Map of local to global indices	2-D texture	Once (write)
Local influence topologies	2-D texture	Once (write)
Node sharing ID	2-D texture	Once (write)
Boundary node type ID	1-D texture	Once (write)
Unknown vector	Array in global memory	Every time step (write/read)

4 Results

This chapter will present the results obtained using the fluid solver. First, the accuracy of the solver will be verified. Next, the robustness of the solver will be demonstrated. Finally, execution times of the GPU-accelerated solver will be compared to that of the serial solver.

4.1 Verification of Accuracy

The first verification problem that will be presented is the solution of flow between infinite parallel plates. This problem served as a good test bed for the accuracy of the solver since an analytical solution for the flow was available against which the numerical results may be compared. The problem setup is shown in Figure 19 below, and consists of two plates, each L = 3 m long, spaced a = 0.5 m apart. A meshless grid of 18 x 66 nodes was used in the solution of the flow. An inlet was placed at the left side of the domain that had a prescribed pressure of zero and a prescribed velocity of 0.02 m/s. The no-slip condition was applied at the walls. The density of the fluid was 1.0 kg/m^3, and its viscosity was 1 x 10^{-4} kg/m·s. An outlet was placed at the rightmost side of the domain. Using the spacing between the plates as the characteristic length, and using the mean

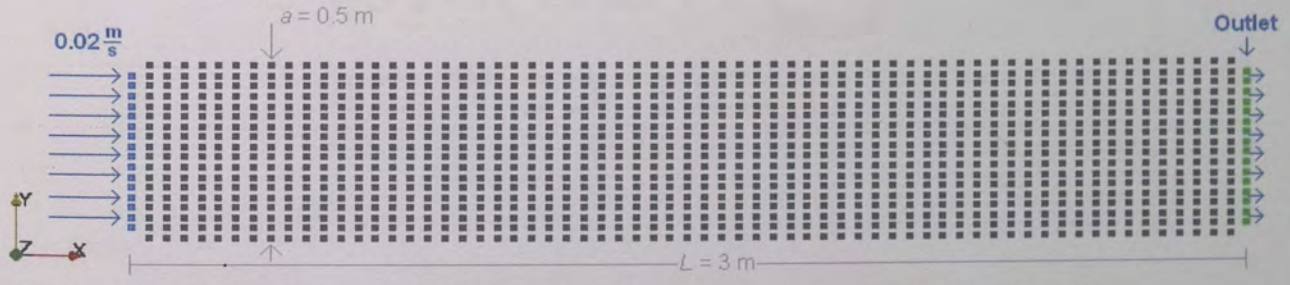

Figure 19. Point distribution for flow between two infinite parallel plates.

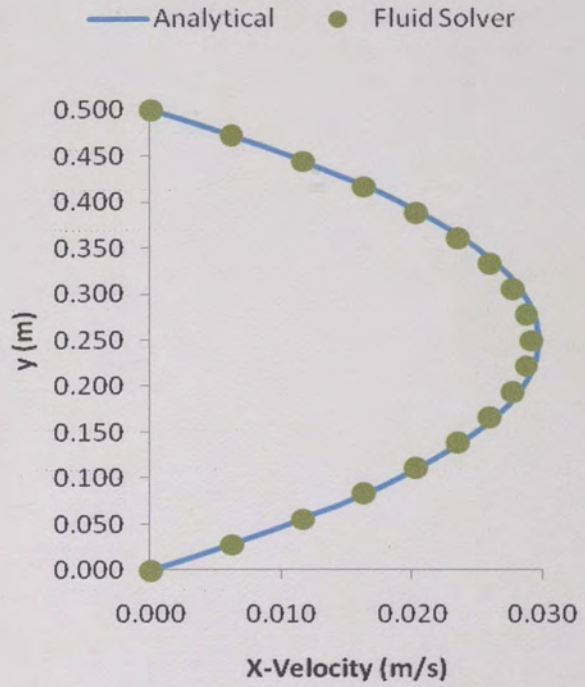

Figure 20. Calculated and exact velocity.

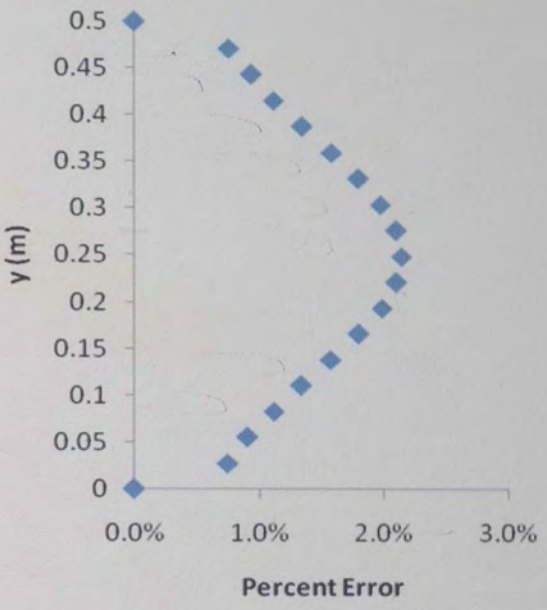

Figure 21. Percent error in solution.

velocity near the inlet as the characteristic velocity, the Reynolds number of the flow is given by equation (93).

$$\mathrm{Re} = \frac{\rho \bar{V} L}{\mu} = 100 \tag{93}$$

An analytical expression for the x-velocity of the fluid in the fully developed region as a function of the y-coordinate is given by equation (94), and the x-position at which the flow may be considered fully developed is calculating with equation (95) (see [63]).

$$u(y) = -4u_{\max}\left[\left(\frac{y}{a}\right)^2 - \frac{y}{a}\right] \tag{94}$$

$$x_{fd} = 0.05a \cdot \mathrm{Re} = 2.475 \tag{95}$$

The problem was evolved in time until steady-state was reached. The x-velocity profile was sampled at each of the computational nodes, and the computed values are

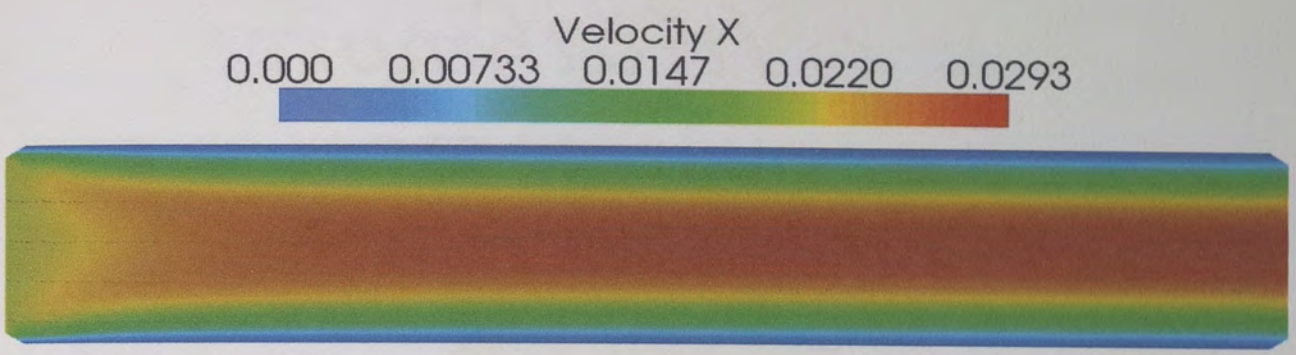

Figure 22. Velocity contours for flow between infinite parallel plates.

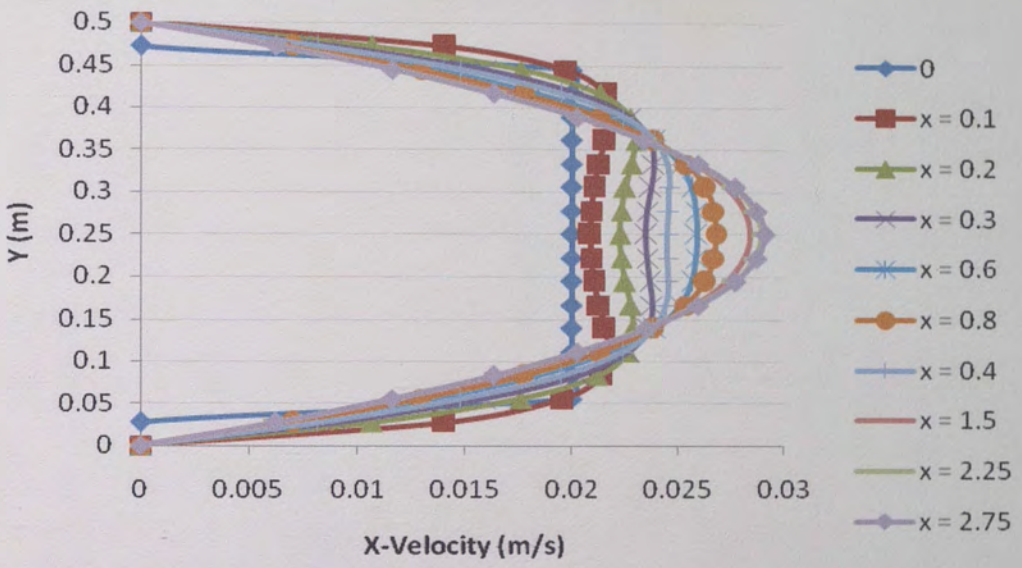

Figure 23. Velocity profile for several x-positions along the plates.

shown along with the exact result in Figure 20. The percent error for the calculated solution is shown as a function of y in Figure 21, and illustrates that the solver was able to represent the exact solution to within an average of 1.3% accuracy. The x-velocity is shown for the entire domain in Figure 22. The horizontal axis is an axis of symmetry, and the y-velocity is zero everywhere in the domain. Figure 23 shows the evolution of the velocity profile at several different x-locations. The propagation of the boundary layer towards the fully developed region is evident from examination of the velocity curves.

A lid-driven cavity flow was also used to verify the accuracy of the solver. The

problem setup is shown in Figure 24. The cavity flow is a single-phase steady-state problem in which all boundaries on a square domain are no-slip walls and the topmost wall is moving to the right with a constant velocity. The density of the fluid was set to 1 kg/m^3 and its viscosity was set to 10^{-4} m^2/s. The Reynolds number for the problem based on the length of one of the sides of the domain was 100. Figure 25 shows the resulting steady-state velocity contours and vectors, revealing the large-scale circulatory behavior of the fluid. The normalized x-velocity along the vertical geometric centerline of the domain and the normalized y-velocity along the horizontal geometric centerline of the domain were compared to the results of Ghia et al. [64], and showed good agreement. The velocities were normalized to the velocity of the top wall. Figure 26 and Figure 27 plot the normalized velocities along the horizontal and vertical centerlines, respectively, and compare the computed results to Ghia's results.

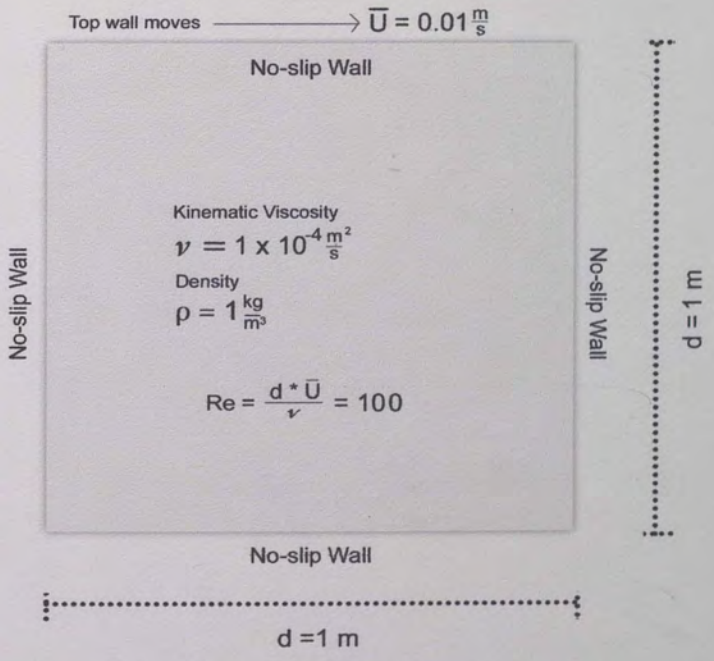

Figure 24. Setup for lid-driven cavity flow.

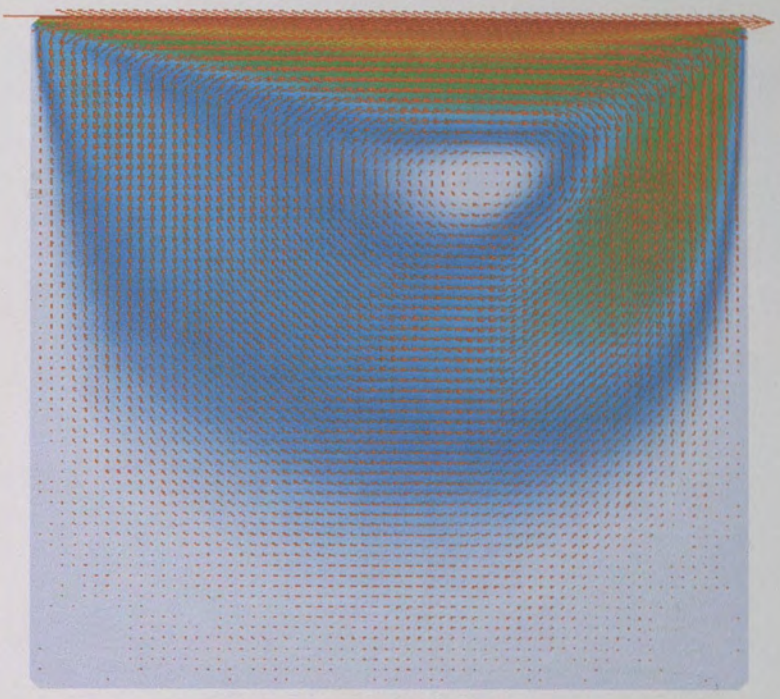

Figure 25. Steady-state velocity for lid-driven cavity.

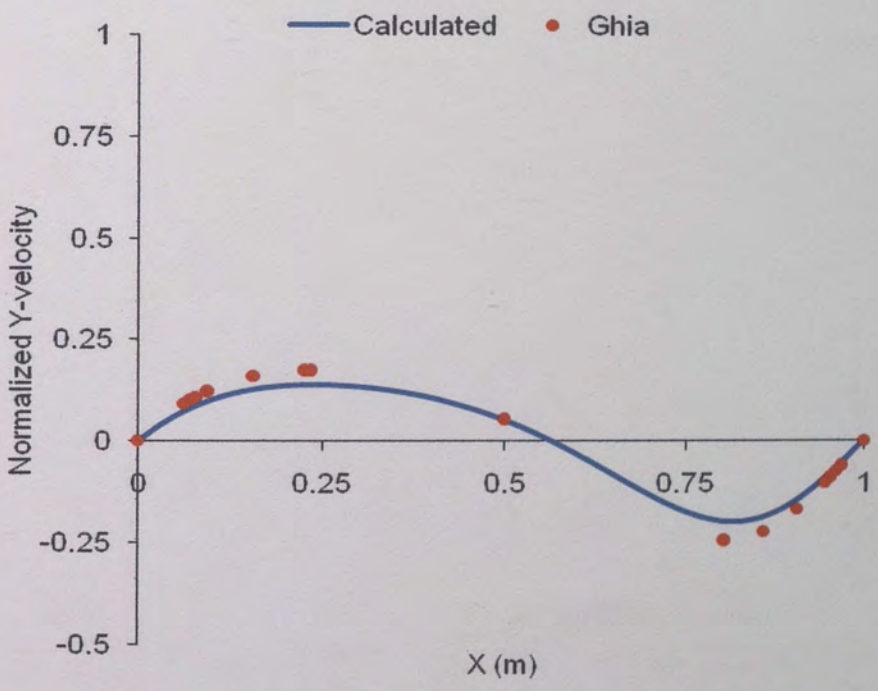

Figure 26. Velocity profile along horizontal centerline for lid-driven cavity.

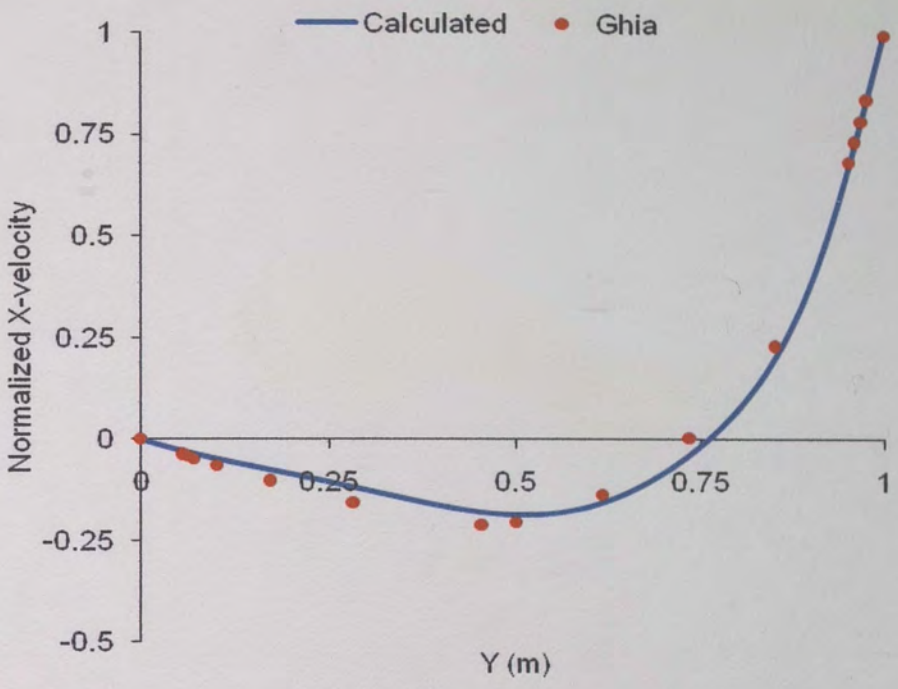

Figure 27. Velocity profile along vertical centerline for lid-driven cavity.

The good agreement with Ghia's canonical data for the lid-driven cavity flow further demonstrates the accuracy of the solver.

Comparison of two-phase results to existing numerical or experimental results was difficult. Numerical solution of two-phase flow is still a problem under much development in the field of computational fluid dynamics, and so canonical data is difficult to come by. Additionally, experimental data for problems such as the dam break or droplet problems are lacking. For this reason, verification of two-phase accuracy is omitted here, although future work will include efforts to demonstrate the accuracy of the two-phase results.

4.2 Robustness

The robustness of the meshless solver was tested through the use of successively more disordered grids in the solution of the same problem. Figure 28 below shows the

regular nodal distribution near the inlet of the infinite parallel plate flow problem. A random disordering term was added to the positions of the interior nodes, resulting in the perturbed nodal distribution shown in Figure 29. The problem was run using both nodal distributions using the same boundary conditions and iteration parameters, and the results agree very closely. Figure 30 and Figure 31 show the velocity distributions of the two solutions, while Figure 32 compares the profile of x-velocity in the fully-developed region of the flow for both grids used.

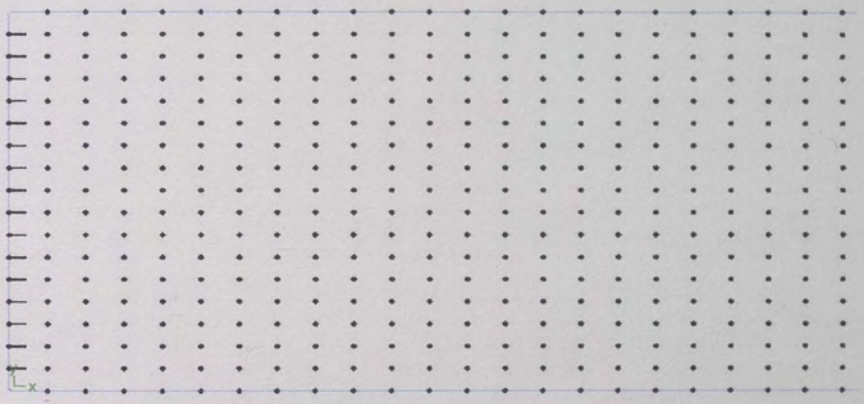

Figure 28. Regular nodal distribution near inlet.

Figure 29. Perturbed nodal distribution near inlet.

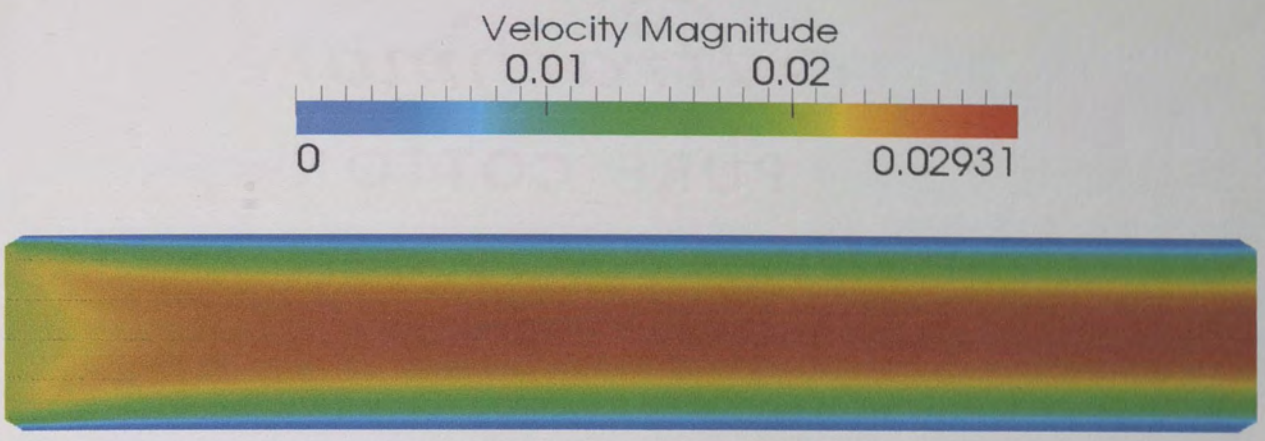

Figure 30. Solution from regular nodal distribution.

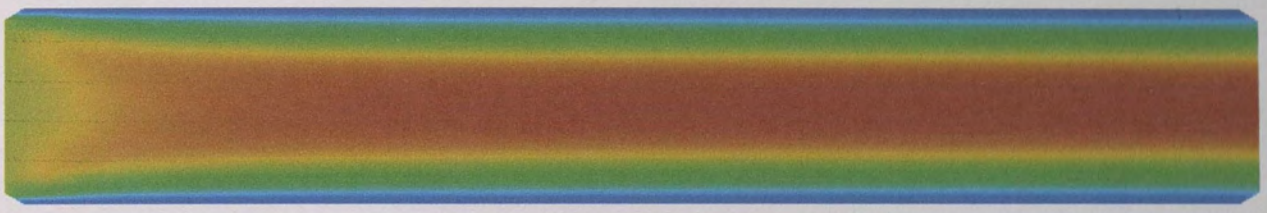

Figure 31. Solution from perturbed nodal distribution.

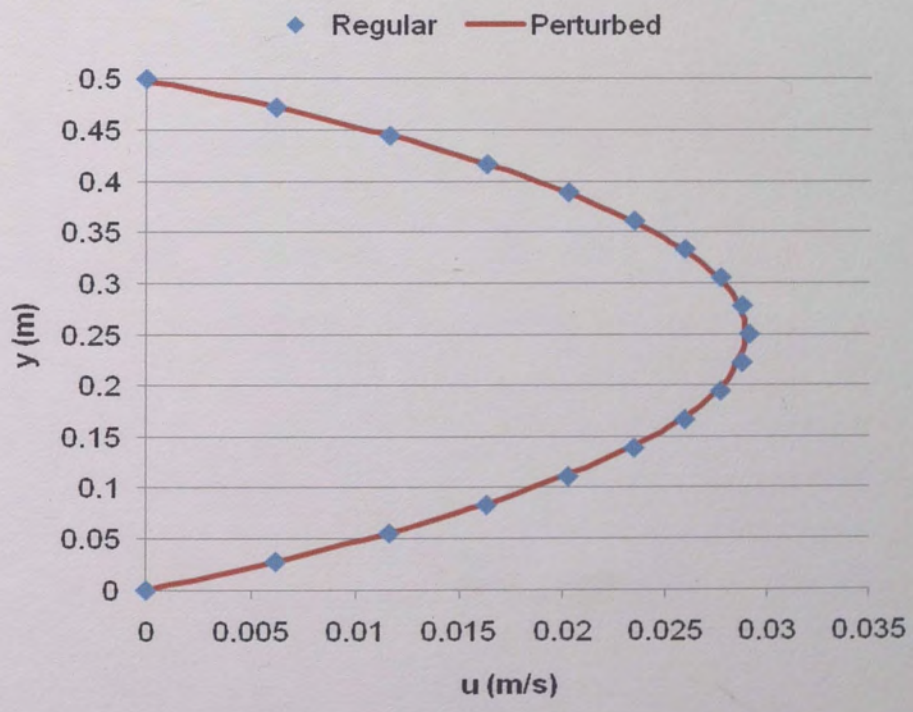

Figure 32. Velocity profiles for both regular and perturbed solutions.

In addition, the solver is robust in handling two-phase flow. A dam break problem was run using three grids. One was regular (Figure 33), one was perturbed (Figure 34), and one was more perturbed (Figure 35). The dam break problem is a canonical (mostly qualitative) problem for two-phase flow, and consists of a wall of water or some denser fluid that is at rest and held in place before the solution begins. At t = 0, the fluid is allowed to collapse under gravity. The geometry used in the dam break simulation is shown in Figure 36.

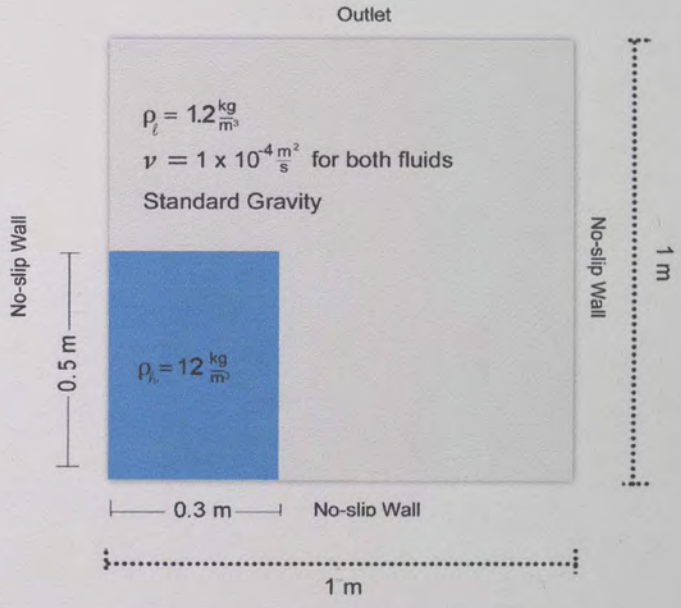

Figure 36. Dam break problem.

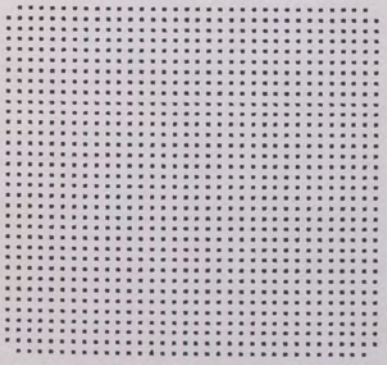

Figure 33. Regular grid.

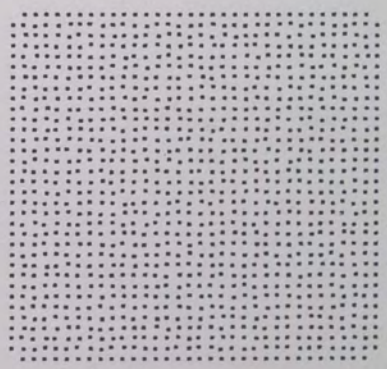

Figure 34. 1st level perturbed.

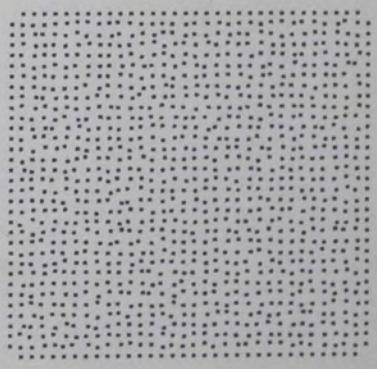

Figure 35. 2nd level perturbed.

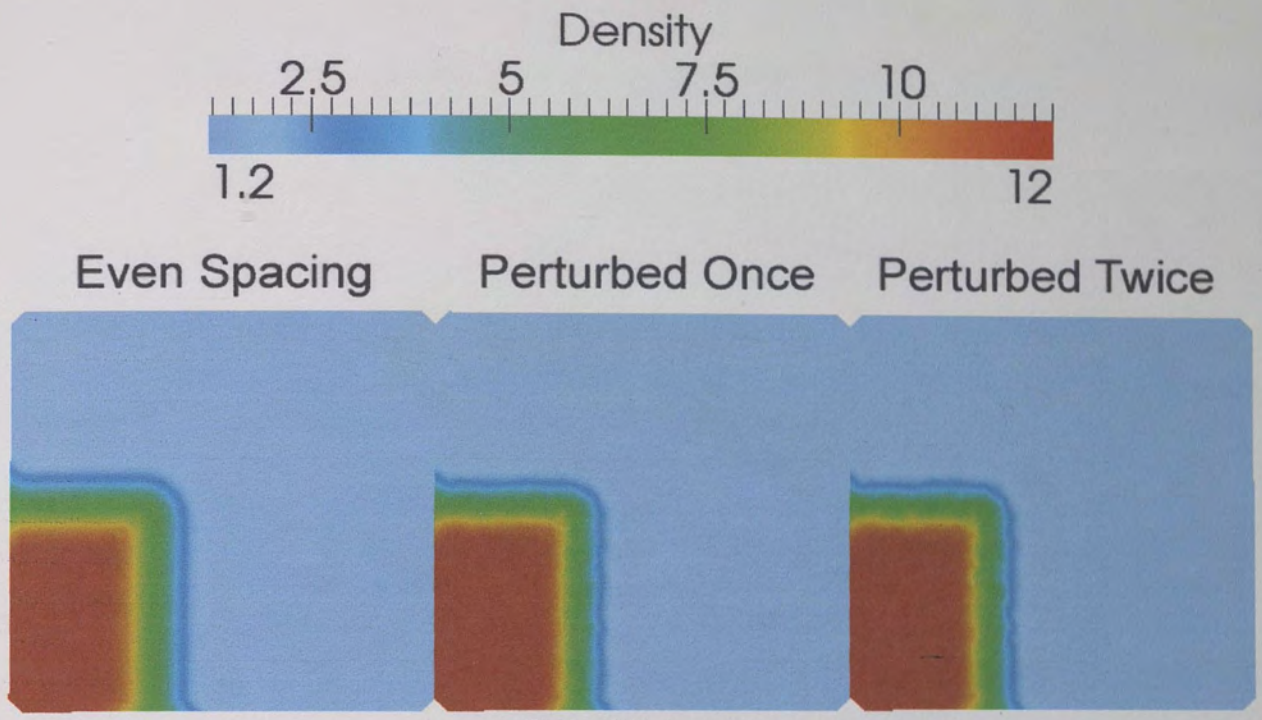

Figure 37. t = 0 s.

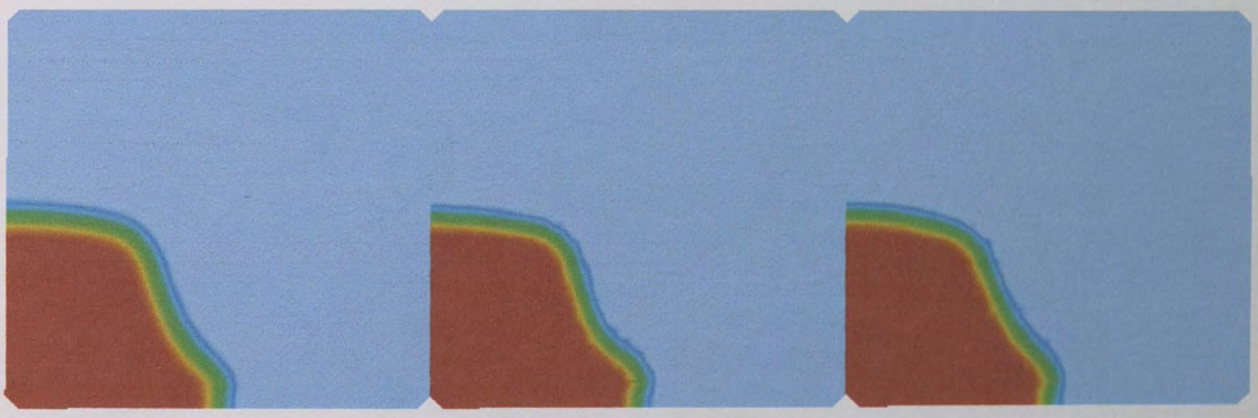

Figure 38. t = 0.1783 s.

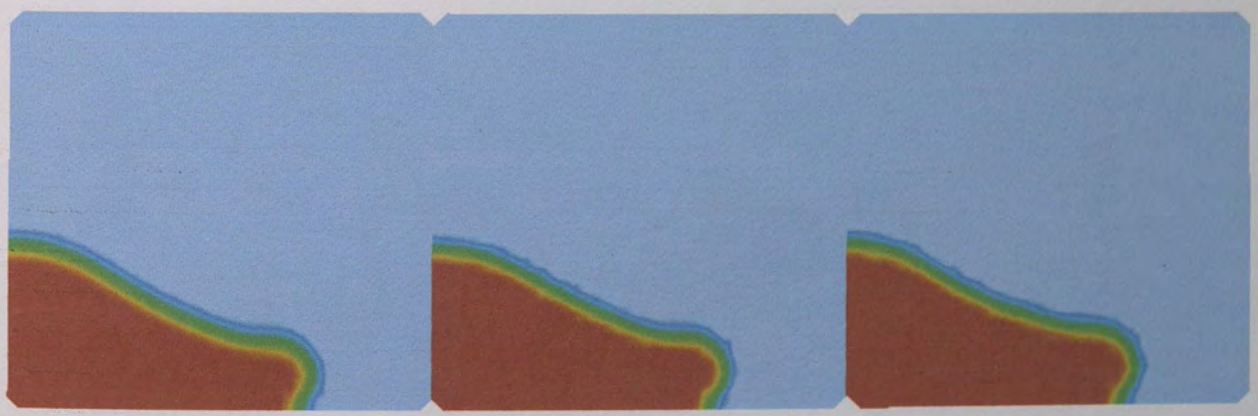

Figure 39. t = 0.2960 s.

Figure 40. t = 0.4047 s.

Figure 41. t = 0.4859 s.

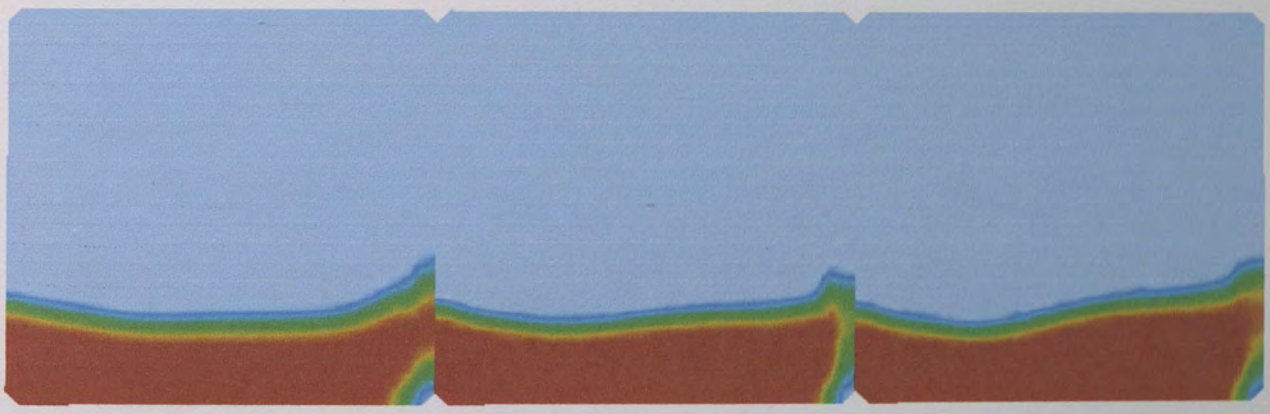

Figure 42. t = 0.5821 s.

As these figures show, the solutions to the dam break problem on successively disordered nodal distributions agree very well. This illustrates that the solver is robust regarding both the discontinuity between the interfaces as well as with the ability to handle irregular nodal distributions.

In order to determine how well mass and volume are conserved in each of the three cases above, the integral of density was performed numerically at each time step shown. The results are shown in Table 7 below.

Table 7. Numerical integral of density at multiple time steps.

Time (s)	Regular Grid	1st Level Perturbed	2nd Level Perturbed
0	3.12097	3.11035	3.11344
0.1783	3.20369	3.17817	3.22279
0.296	3.32571	3.29973	3.34034
0.4047	3.40764	3.39243	3.4398
0.4859	3.43341	3.4258	3.47261
0.5821	3.48225	3.47872	3.59681

Table 7 shows good agreement between the regular and perturbed grids at each time step.

4.3 Two-phase Results

More two-phase results will now be presented. A droplet problem in which a droplet of denser fluid was allowed to fall under the force of gravity into a pool of denser fluid was run. The problem setup is shown in Figure 43.

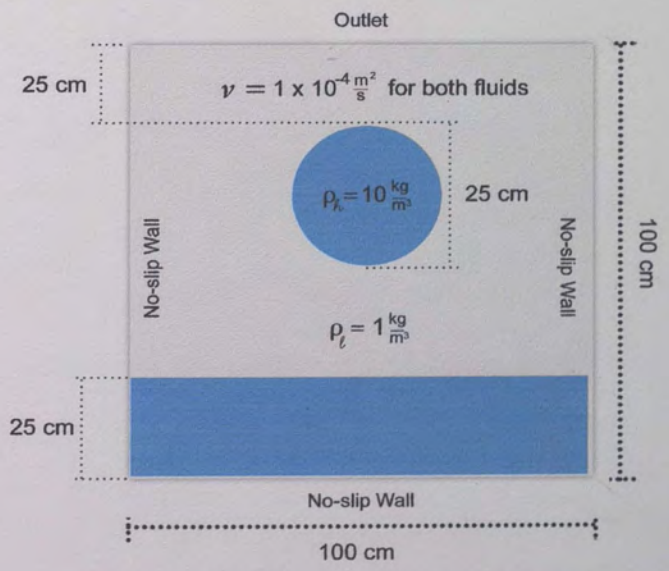

Figure 43. Setup for droplet problem.

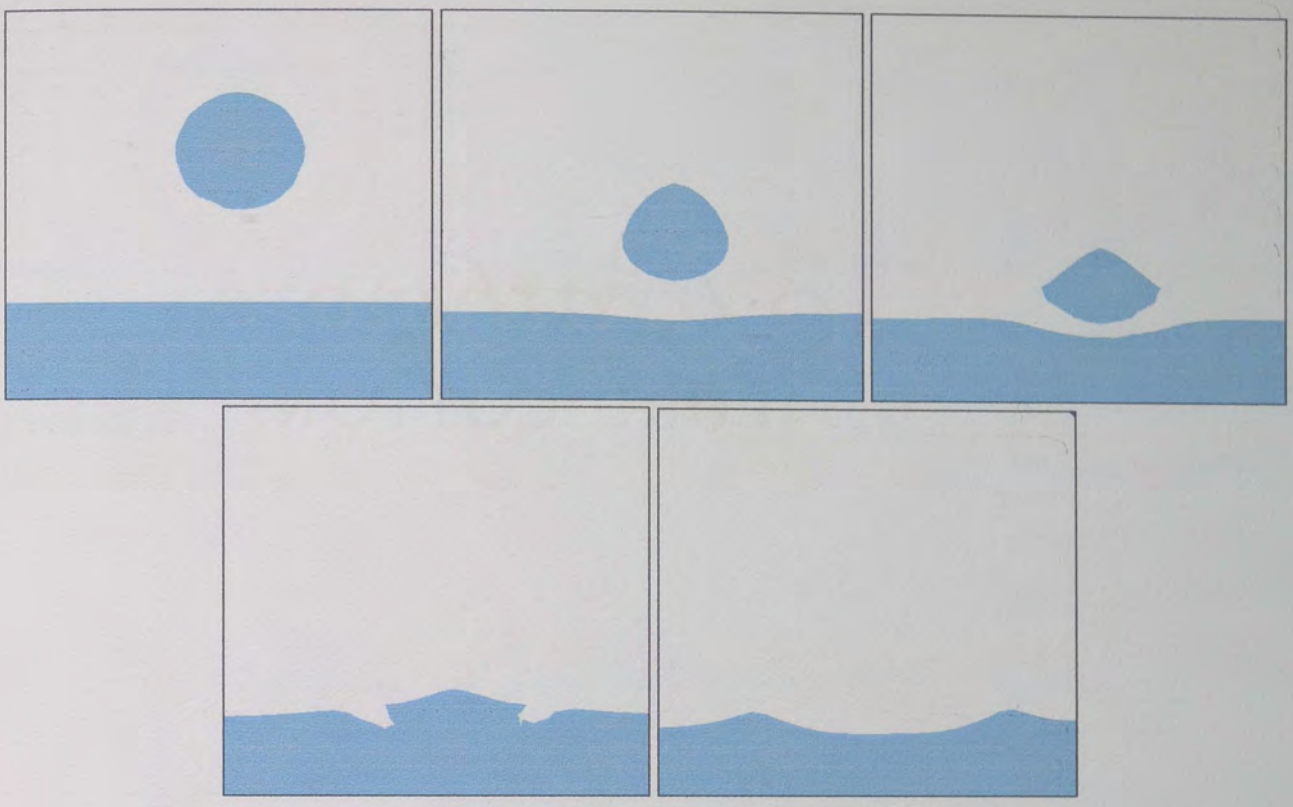

Figure 44. Evolution of droplet.

Figure 44 shows the evolution of the droplet. While the droplet remains relatively symmetric, there are still instabilities present as it nears the surface of the pool of the denser fluid. This problem was a relatively stable two-phase problem. An unstable problem can be found in the simulation of a rising bubble. Both the drop and bubble were solved using 34 x 34 node meshless point distributions.

The rising bubble problem consists of a bubble of lighter fluid submersed in a denser fluid. The only forces present are due to buoyant effects. Figure 45 shows a bubble of a fluid of density 1 kg/m^3 rising in a fluid of density 10 kg/m^3. The viscosity and surface tension coefficient for both fluids were 10^{-4} m^2/s and 0.01 N/m, respectively. The bubble flows off to one side rather than straight up. This is due to the large amount of numerical instability present, especially at due to surface tension at the interface.

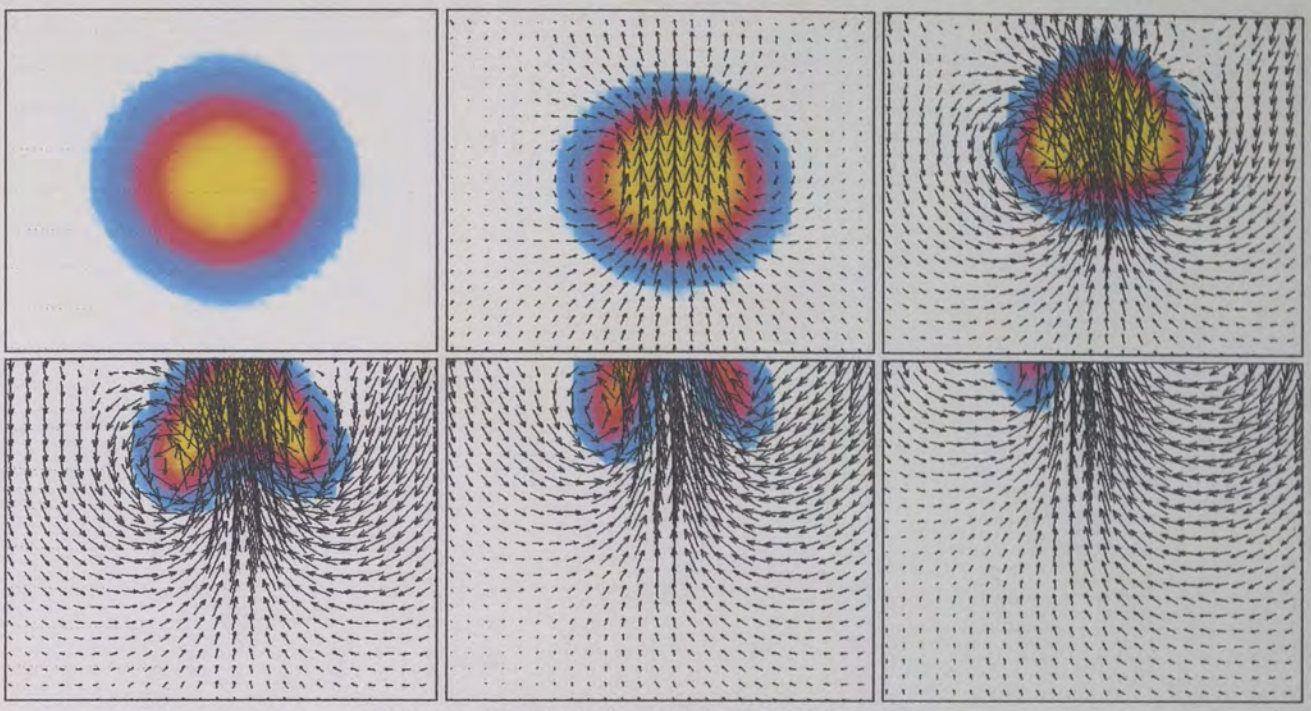

Figure 45. Evolution of rising bubble flow.

4.4 Benchmarking of GPU-Accelerated Routines

Benchmarking of the two GPU-accelerated routines was performed using four different grid sizes for two different problems, one single-phase and one two-phase. The four grid sizes used were 18 x 18, 34 x 34, 66 x 66, and 130 x 130. Figure 46 below shows the speedup factors for each of these grid sizes and problems. As can be seen, the acceleration obtained is independent of the problem being solved, and strongly depends on grid size. It was observed that the acceleration factor increases considerably as the grid size increases. Figure 47 shows the execution time per time step for the serial code and the GPU-accelerated code as a function of grid size. It is interesting to note that the execution time for the GPU-accelerated code scales fairly linearly with grid size, while execution time for the serial code appears to scale quadratically with grid size.

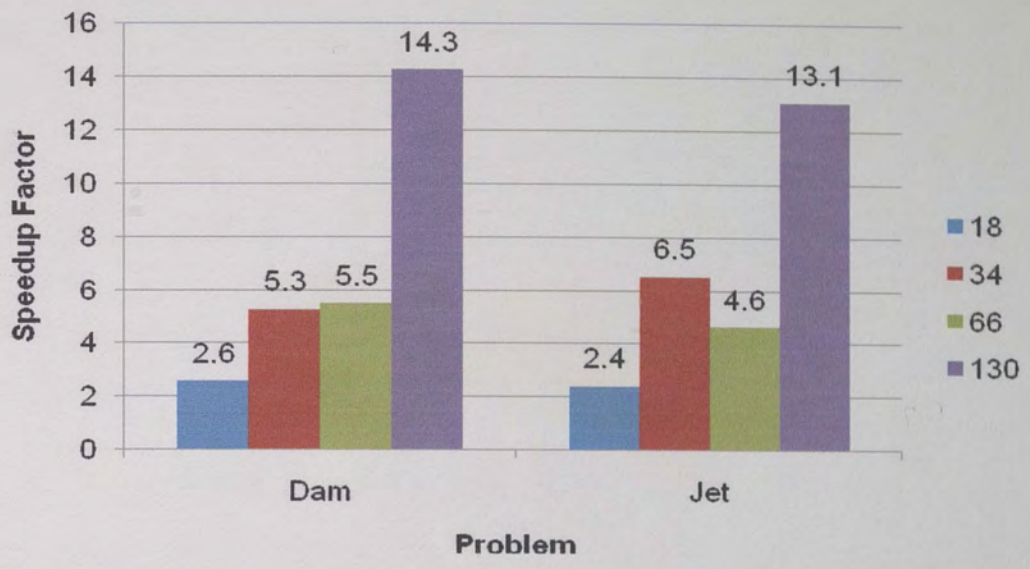

Figure 46. Speedup factors for one and two-phase problems on several grid sizes.

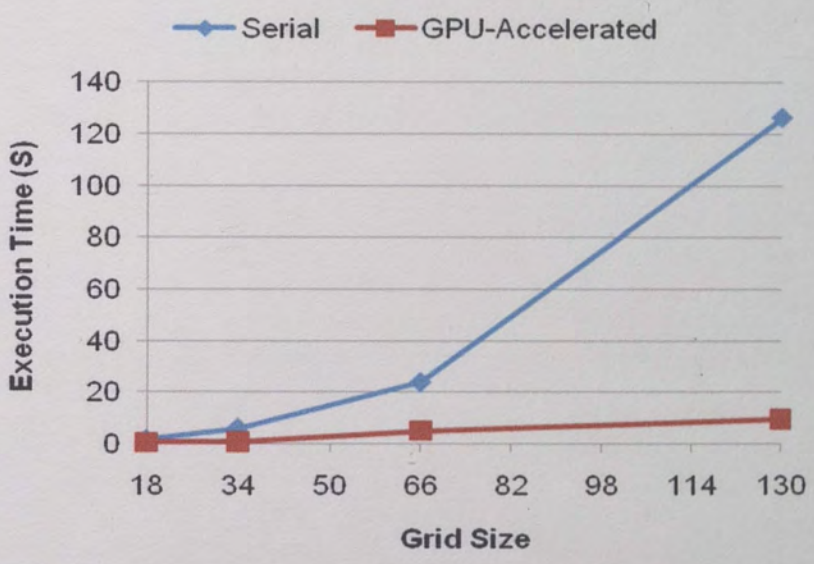

Figure 47. Average execution times per time step.

This is a result of the architecture of the GPU. As more threads are executed on the device, its efficiency increases due to the higher amount of occupancy per multiprocessor than for lower numbers of threads.

As far as accuracy and robustness is concerned, the GPU-accelerated code performs identically to the purely serial code. Many of the examples presented in the

previous sections concerning accuracy and robustness were computed with the GPU-accelerated code. Since all the data is identical for the two codes, it would be trivial to display any comparative results, as they may already be found throughout the solutions previously discussed.

The data discussed above concerns the acceleration of the entire time step. Looking at individual subroutines rather than the time step as a whole, the largest speedup factor observed was for the solution of the Helmholtz potential in a dam break problem on a 130 x 130 grid, as shown in Figure 48 and Figure 49. The speedup factor in this case was found to be 16.07. Table 8 compares the maximum speedup factor observed in this project to the other contemporary GPGPU projects in CFD that were discussed in section 1.4.1. The speedup factor achieved in this project is fairly consistent with other contemporary speedup factors.

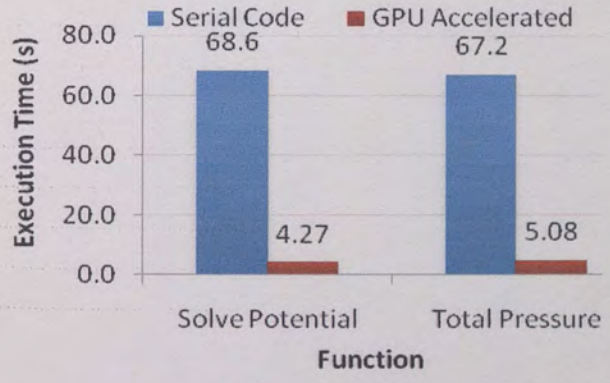

Figure 48. Results for 130 x 130 dam break.

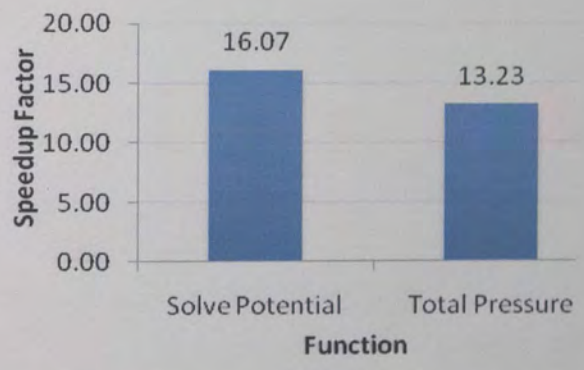

Figure 49. Speedup for 130 x 130 dam break.

Table 8. Speedup factor comparison.

Author	Method	Speedup factor
Brandvik and Pullan	FVM	16
Li et al.	Lattice-Boltzman	15
This project	GFDM	14.3
Thibault et al.	FDM	13
Riegel et al.	Lattice-Boltzman	9

5 Conclusions

In this chapter, important observations and knowledge gained through this study will be reviewed, after which recommendations and possibilities for future research will be discussed.

5.1 Observations

The main goal of this project was to enhance the speed with which the two-phase incompressible Navier-Stokes equations could be solved. This goal was achieved through the novel coincident implementation of a radial-basis function enhanced virtual finite differencing scheme (the GFDM method) using GPU-acceleration and the level set method for interface capturing.

Substantial accelerations were seen for subroutines that were implemented on the GPU. For large grid sizes, the speedup factor varied from 4 to 14 for total time step, and peaked at 16 for the acceleration of the solution of the Poisson equation for Helmholtz potential. Comparing the maximum total time-step speedup factor to those reported in the literature, this project was found to perform as well as or better than several contemporary GPU-accelerated CFD codes.

The GFDM scheme was also observed to be both accurate and robust. Irregular nodal distributions produced solutions that agreed very well with solutions obtained on regular grids. For fluid flow between infinite parallel plates, the computed solution was found to be within 2% of the analytical solution on average. The method was shown to handle an increasingly irregular nodal distribution with ease, even in the computation of

two-phase flow with a fairly large density gradient near the interface. This project has served to illustrate that the GFDM method is an accurate and viable scheme for two-phase flows.

5.2 Future Work

This research could be taken in many directions. Because of its exploration of two-phase flow, an extension to fluid-structure interaction would prove extremely useful in the simulation of processes ranging from the effect of storm surge on coastal architecture to modeling the propagation of an embolism in the human cardiovascular system. Extension of the solver to three dimensions would also prove very useful, especially since mesh generation for traditional CFD methods becomes even more complex in three spatial dimensions.

NVIDIA graphics cards have the ability to be directly interfaced with one another, providing extremely high data throughput rates and increased processing power. This project is a natural jumping-off point for further exploration in parallelization of arithmetically intensive algorithms. The GPU is a tool whose value is beginning to be truly recognized, and its use as a parallel computing device is not limited to its solo operation. Research into collections of high performance GPUs has the potential to transform massively parallel computing. Consider the results presented here for the acceleration of one GPU over one CPU core. Replacing the CPU as the computing device in a cluster environment has the potential to allow the GPU to become the preeminent tool of choice for high performance computing. It is especially important during this early stage in general purpose computing on the GPU that significant research is

conducted concerning the correct and most efficient way to program and make use of all of the resources the GPU has to offer.

References

[1] S. Gerace, K. Erhart, E. Divo, and A. Kassab, "Generalized Finite Difference Meshless Method in Computational Mechanics and Thermofluids," in *ECCOMAS Coupled Problems 2009*, Ischia Island, Italy, 2009.

[2] S. Osher and J. Sethian, "Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations," *J. Comput. Phys.*, vol. 79, pp. 12-49, 1988.

[3] C.-W. Shu, "High-order Finite Difference and Finite Volume WENO Schemes and Discontinuous Galerkin Methods for CFD," *International Journal of Computational Fluid Dynamics,* vol. 17, pp. 107-118, 2003.

[4] H. Johnston and J.-G. Liu, "Finite difference schemes for incompressible flow based on local pressure boundary conditions," *Journal of Computational Physics,* vol. 180, pp. 120-154, 2002.

[5] J. N. Reddy and D. K. Gartling, *The Finite Element Method in Heat Transfer and Fluid Dynamics*: CRC Press, 2000.

[6] V. Girault and P.-A. Raviart, *Finite Element Methods for Navier-Stokes Equations: Theory and Algorithms*. New York: Springer, 1986.

[7] F. Harlow and E. Welch, "Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface," *Physics of Fluids,* vol. 8, pp. 2182-2189, 1965.

[8] T. Belytschko, Y. Y. Lu, and L. Gu, "Element-free Galerkin methods," *International Journal for Numerical Methods in Engineering,* vol. 37, pp. 229-256, 1994.

[9] E. Brakkee, "A parallel domain decomposition algorithm for the incompressible Navier-Stokes equations," *Simulation Practice and Theory,* vol. 3, pp. 185-205, 1995.

[10] E. Schreck and M. Perić, "Computation of fluid flow with a parallel multigrid solver," *International Journal for Numerical Methods in Fluids,* vol. 16, pp. 303-327, 1993.

[11] S. K. Aliabadi and T. E. Tezduyar, "Parallel fluid dynamics computations in aerospace applications," *International Journal for Numerical Methods in Fluids,* vol. 21, pp. 783-805, 1995.

[12] E. Lindholm, M. Kligard, and H. Moreton, "A user-programmable vertex engine," in *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001, pp. 149-158.

[13] T. R. Halfhill, "Parallel processing with CUDA," in *Microprocessor Report*: Reed Electronics Group, 2008.

[14] A. Iske and M. Käser, "Conservative semi-Lagrangian advection on adaptive unstructured meshes," *Numerical Methods for Partial Differential Equations,* vol. 20, pp. 388-411, 2004.

[15] R. F. Vanderwijngaart, "Composite-grid techniques and adaptive mesh refinement in computational fluid dynamics," Stanford Univ., CA., 1990.

[16] J. Paraire, K. Morgan, and J. Peiro, "Unstructured finite element mesh generation and adaptive procedures for CFD," in *AGARD, Applications of Mesh Generation to Complex 3-D Configurations 12 p (SEE N90-21975 15-34)*, 1990.

[17] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Performance Modeling and Tuning of an Unstructured Mesh CFD Application," in *Supercomputing, ACM/IEEE 2000 Conference*, 2006, pp. 34-34.

[18] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Achieving High Sustained Performance in an Unstructured Mesh CFD Application," in *Supercomputing, ACM/IEEE 1999 Conference*, 2006, pp. 69-69.

[19] Y. Ito and K. Nakahashi, "Improvements in the reliability and quality of unstructured hybrid mesh generation," *International Journal for Numerical Methods in Fluids,* vol. 45, pp. 79-108, 2004.

[20] R. Löhner, J. Baum, E. Mestreau, D. Sharov, C. Charman, and D. Pelessone, "Adaptive embedded unstructured grid methods," *International Journal for Numerical Methods in Engineering,* vol. 60, pp. 641-660, 2004.

[21] B. Nayroles, G. Touzot, and P. Villon, "Generalizing the finite element method: Diffuse approximation and diffuse elements," *Computational Mechanics,* vol. 10, pp. 307-318, 1992.

[22] S. N. Atluri and T. Zhu, "A new Meshless Local Petrov-Galerkin (MLPG) approach in computational mechanics," *Computational Mechanics,* vol. 22, pp. 117-127, 1998.

[23] E. Divo and A. Kassab, "An Efficient Localized Radial Basis Function Meshless Method for Fluid Flow and Conjugate Heat Transfer," *Journal of Heat Transfer,* vol. 129, pp. 124-136, 2007.

[24] B. Fraeijs de Veubeke, R. North Atlantic Treaty Organization. Advisory Group for Aeronautical, S. Development, and P. Materials, *Matrix methods of structural analysis*. New York: Published for and on behalf of Advisory Group for Aeronautical Research and Development, North Atlantic Treaty Organization by Macmillan, 1964.

[25] S. Idelsohn and E. Oñate, "To mesh or not to mesh. That is the question..." *Computer Methods in Applied Mechanics and Engineering,* vol. 195, pp. 4681-4696, 2006.

[26] S. Idelsohn, E. Oñate, N. Calvo, and F. Del Pin, "The meshless finite element method," *International Journal for Numerical Methods in Engineering,* vol. 58, pp. 893-912, 2003.

[27] J. Melenk, "The partition of unity finite element method: Basic theory and applications," *Computer Methods in Applied Mechanics and Engineering,* vol. 139, pp. 289-314, 1996.

[28] A. Duarte and T. Oden, "H-p clouds: an h-p meshless method," *Numerical Methods for Partial Differential Equations,* vol. 12, pp. 673-705, 1996.

[29] J. J. Monaghan, "Smoothed Particle Hydrodynamics," *Annual Review of Astronomy and Astrophysics,* vol. 30, pp. 543-574, 2003.

[30] D. Enright, F. Losasso, and R. Fedkiw, "A fast and accurate semi-Lagrangian particle level set method," *Computers & Structures,* vol. 83, pp. 479-490, 2005.

[31] K. Erhart, S. Gerace, E. Divo, and A. Kassab, "An RBF Interpolated Generalized Finite Difference Meshless Method for Compressible Turbulent Flows," in *ASME-IMECE* Lake Buena Vista, FL, 2009.

[32] K. Erhart, S. Gerace, E. Divo, and A. Kassab, "Turbulent Compressible Flow Analysis with Meshless Methods," in *ECCOMAS Coupled Problems 2009*, Ischia Island, Italy, 2009.

[33] S. Gerace, K. Erhart, E. Divo, and A. Kassab, "Local and Virtual RBF Meshless Method for High Speed Flows," in *BEM/MRM 31*, C. A. Brebbia, Ed. Southampton, UK: WIT Press, 2009, pp. 83-94.

[34] C. W. Hirt, B. D. Nichols, and N. C. Romero, "SOLA: A numerical solution algorithm for transient fluid flows," *NASA STI/Recon Technical Report N,* vol. 75, 1975.

[35] B. D. Nichols and C. W. Hirt, "Improved free surface boundary conditions for numerical incompressible-flow calculations," *Journal of Computational Physics,* vol. 8, pp. 434-448, 1971.

[36] C. W. Hirt and B. D. Nichols, "Volume of fluid (VOF) method for the dynamics of free boundaries," *Journal Name: J. Comput. Phys.; (United States); Journal Volume: 39:1,* pp. Medium: X; Size: Pages: 201-225, 1981.

[37] Y. C. Chang, T. Y. Hou, B. Merriman, and S. Osher, "A level set formulation of Eulerian interface capturing methods for incompressible fluid flows," *J. Comput. Phys.,* vol. 124, pp. 449-464, 1996.

[38] M. Sussman, P. Smereka, and S. Osher, "A Level Set Approach for Computing Solutions to Incompressible Two-Phase Flow," *Journal of Computational Physics,* vol. 114, pp. 146-159, 1994.

[39] J. A. Sethian and P. Smereka, "Level set methods for fluid interfaces," *Annual Review of Fluid Mechanics,* vol. 35, pp. 341-372, 2003.

[40] L. Jed, R. Mark, R. D. Bruce, and P. G. Donald, "Real-time robot motion planning using rasterizing computer graphics hardware," in *Proceedings of the 17th annual conference on Computer graphics and interactive techniques* Dallas, TX, USA: ACM, 1990.

[41] C.-A. Bohn, "Kohonen feature mapping through graphics hardware," in *3rd Int. Conference on Computational Intelligence and Neurosciences,* N. Carolina, USA, 1998.

[42] S. Mikio and F. Marie-Claire, "Interference detection through rasterization," *The Journal of Visualization and Computer Animation,* vol. 2, pp. 132-134, 1991.

[43] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrÃ¼ger, A. Lefohn, and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum,* vol. 26, pp. 80-113, 2007.

[44] R. M. William, R. S. Glanville, A. Kurt, and J. K. Mark, "Cg: a system for programming graphics hardware in a C-like language," in *ACM SIGGRAPH 2003 Papers* San Diego, California: ACM, 2003.

[45] O. Michael, "HLSL shader model 4.0," in *ACM SIGGRAPH 2007 courses* San Diego, California: ACM, 2007.

[46] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers,* San Diego, California, 2003, pp. 917-924.

[47] G. Nolan, W. Cliff, L. Gregory, L. David, and H. Greg, "A multigrid solver for boundary value problems using programmable graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* San Diego, California: Eurographics Association, 2003.

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

[49] J. Stam, "Stable fluids," in *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, 1999, pp. 121-128.

[50] W. Li, Z. Fan, X. Wei, and A. Kaufman, "GPU-based flow simulation with complex boundaries," in *GPU Gems 2*, M. Pharr, Ed.: Addison Wesley, 2005, pp. 747-764.

[51] E. Riegel, T. Indinger, and N. Adams, "Implementation of a Lattice–Boltzmann method for numerical fluid mechanics using the nVIDIA CUDA technology," *Computer Science - Research and Development,* vol. 23, pp. 241-247, 2009.

[52] J. C. Thibault and I. Senocak, "CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows," in *47th AIAA Aerospaces Sciences Meeting Including the New Horizons Forum and Aerospace Exposition* Orlando, Florida, 2009.

[53] J. M. Cohen and M. J. Molemaker, "A fast double precision CFD code using CUDA," Santa Clara, CA: NVIDIA Corporation, 2009.

[54] T. Brandvik and G. Pullan, "Acceleration of a 3D Euler solver using commodity graphics hardware," in *46th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, Nevada, 2008.

[55] K. Hegeman, N. Carr, and G. Miller, "Particle-Based Fluid Simulation on the GPU," in *Computational Science â€" ICCS 2006*, 2006, pp. 228-235.

[56] T. Harada, S. Koshizuka, and Y. Kawaguchi, "Smoothed Particle Hydrodynamics on GPUs," in *Proc. of Computer Graphics International*, 2007, pp. 63-70.

[57] G. K. Batchelor, *An Introduction to Fluid Dynamics*. Cambridge, UK: Cambridge University Press, 2000.

[58] A. Chorin, "Numerical Solution of the Navier-Stokes Equations," *Mathematics of Computation,* vol. 22, pp. 745-762, 1968.

[59] S. Osher and R. Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*. New York: Springer-Verlag New York, Inc., 2003.

[60] E. Rouy and A. Tourin, "A Viscosity Solutions Approach to Shape-From-Shading," *SIAM Journal on Numerical Analysis,* vol. 29, pp. 867-884, 1992.

[61] G. C. Fox, "A Review of Automatic Load Balancing and Decomposition Methods for the Hypercube," *Institute for Mathematics and Its Applications,* vol. 13, p. 63, 1988.

[62] "NVIDIA CUDA Compute Unified Device Architecture Programming Guide," Santa Clara, CA: NVIDIA, 2007.

[63] R. W. Fox, P. J. Pritchard, and A. T. McDonald, *Introduction to Fluid Mechanics*, 7 ed. Hoboken, NJ: Wiley, 2009.

[64] U. Ghia, K. Ghia, and C. Shin, "High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method," *Journal of Computational Physics,* vol. 48, pp. 387-411, 1982.