GPU Accelerated Three Dimensional Unstructured Geometric Multi-grid Solver

Jin Sebastian

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Master of Technology

भारतीय प्रौद्योगिकी संस्थान हैदराबाद

Indian Institute of Technology Hyderabad

Department of Computer Science and Engineering

July 2013

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

[illegible]

(Signature)

(Jin Sebastian)

CS11M11

(Roll No.)

Approval Sheet

This Thesis entitled “GPU Accelerated Three Dimensional Unstructured Geometric Multi-grid Solver” by Jin Sebastian is approved for the degree of Master of Technology from IIT Hyderabad

[illegible]

(Dr. Raja Ban$\overset{e}{a}$rjee) Examiner
Dept. of Mechanical Engineering
IITH

[illegible]

(Dr. M. V. Panduranga Rao) Examiner
Dept. of Computer Science and Engineering
IITH

[illegible]

(Dr. Naveen Sivadasan) Adviser
Dept. of Computer Science and Engineering
IITH

[illegible]

(Dr. C. Krishna Mohan) Chairman
Dept. of Computer Science and Engineering
IITH

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my adviser Dr. Naveen Sivadasan for his valuable guidance, constant encouragement, motivation, enthusiasm, and immense knowledge. Furthermore I would like to thank Dr. Raja Banerjee for introducing me to the topic as well for the valuable guidance on the way. Many individuals contributed in many different ways to the completion of this thesis. I am deeply grateful for their support, and thankful for the unique chances they offered me. Finally, I thank my family for supporting me throughout all my studies at the institute.

I would like to make a special mention of the excellent facility provided by my institute, IIT Hyderabad.

Abstract

Consider a set of points P in three dimensional euclidean space. For each point p, the neighbourhood $N(p)$, is defined as the set of points in P, which are voronoi neighbours. Each point in P represents a variable and its value is dependent on the value of its neighbourhood. Its value is given by the sum of the values of points in its neighbourhood scaled by predefined constants. The constants depend on the spacing between the points. The problem is to solve all the variables. Such representations arise naturally in solving flow equations in Computational Fluid Dynamics with domains represented using unstructured meshes. The problem reduces to solve a system of linear equations. In this work geometric multigrid method is implemented for solving the problem faster. Solving this problem on very large input is a time consuming process. The inputs considered here are having size of the order of millions. Graphics Processing Units(GPU) are dedicated parallel processors which serves both as a programmable graphics processor and a scalable parallel computing platform. The parallelization of this problem for GPUs is not straight forward because of the irregularity. The CFD problem used for experiment is the steady and unsteady heat transfer problem in 3D unstructured meshes.The combination of multigrid algorithm and GPU implementation for the steady problem on a 1.6 million mesh gives 1630 times speed up compared to non-multigrid CPU implementation.

Contents

Declaration ii
Approval Sheet iii
Acknowledgements iv
Abstract v

Nomenclature **vii**

1 Introduction **1**
1.1 3D Unstructured Flow Problem 1
1.2 Related Work 2
1.3 Overview of our work 3
1.4 Thesis Outline 3

2 GPU Computing Model **4**
2.1 Hardware Architecture 4
2.2 CUDA Programming Model 5
2.2.1 Compiler model 6
2.2.2 Execution Model 7
2.2.3 Memory Model 8
2.3 Performance Optimizations 9

3 Geometric Multigrid Solver **10**
3.1 Structured and Unstructured Multigrid 11
3.2 Unstructured Multigrid Level Generation Schemes 12
3.2.1 Maximal Independent Set 12
3.2.2 Dominating Set 13
3.2.3 Function Based Coarsening 15
3.3 Gauss Seidel Iterative Method for Relaxation 15

4 GPU Implementation **17**
4.1 Structured grid 17
4.2 Unstructured grid 17
4.3 Graph Coloring 18
4.4 Graph Representation 19
4.5 Multigrid Implementation 21

4.5.1 Restriction 21
4.5.2 Prolongation 21

5 Experiments and Results **22**
5.1 Steady and unsteady heat transfer problem 22
5.2 Experimental setup 23
5.3 Results 23

6 Discussion **28**

References **29**

Chapter 1

Introduction

Graphical Processing Units (GPUs) are emerged as a main component of the High Performance Computing (HPC) systems. GPUs are originally designed as dedicated parallel processor for accelerating graphics rendering applications used in visual effects, gaming, high resolution display etc. Recently GPUs are evolved as both a programmable graphics processor and a scalable parallel computing platform, whch can be used for parallelizing general purpose applications. Current GPUs are incorporated with hundreds of lightweight cores which can accelerate compute intensive applications substantially. Compared to CPU, GPU architecture dedicates more transistors for creating processing elements than control circuits and as a result the GPUs are less programmable. The GPUs are efficient for data parallel applications as it has a Single Instruction Multiple Data (SIMD) device architecture. Scientific computing areas such as Computational Fluid Dynamics (CFD), Computational Biology and Chemistry, Weather and Climate Forecasting etc. have problems which are highly data parallel and deals with input of very large size. These applications, that have an ever-growing demand for the power of high performance computing infrastructure can make use of this power efficient and less expensive GPUs. Recently many programming interfaces for GPU computing in which Compute Unified Device Architecture(CUDA) introduced by NVIDIA and OpenCL introduced by Khronos group are mainly used interfaces. In this work we use CUDA programming model for the GPU implementation.

1.1 3D Unstructured Flow Problem

Consider a set of points P in three dimensional euclidean space. For each point p, the neighbourhood $N(p)$, is defined as the set of points in P, which are voronoi neighbours. Two points are voronoi neighbours if there is a common voronoi edge in the voronoi cells of the two points. Voronoi cell of a point p is the region around the point p such that every point in the region is farther to any other point in P than p. Each point in P represents a variable and its value is dependent on the value of its neighbourhood. Its value is given by the sum of the values of points in its neighbourhood scaled by predefined constants. The constants depend on the spacing between the points. The value at q,

$$v(q) = \sum_{p \in N(q)} a_{qp} v(p) \qquad \text{and} \qquad \sum_{p \in N(q)} a_{qp} < 1$$

where a_{qp} is the constant by which $v(p)$ is scaled for finding the value of q. The problem is to solve all the variables. Such representations arise naturally in Computational Fluid Dynamics while solving

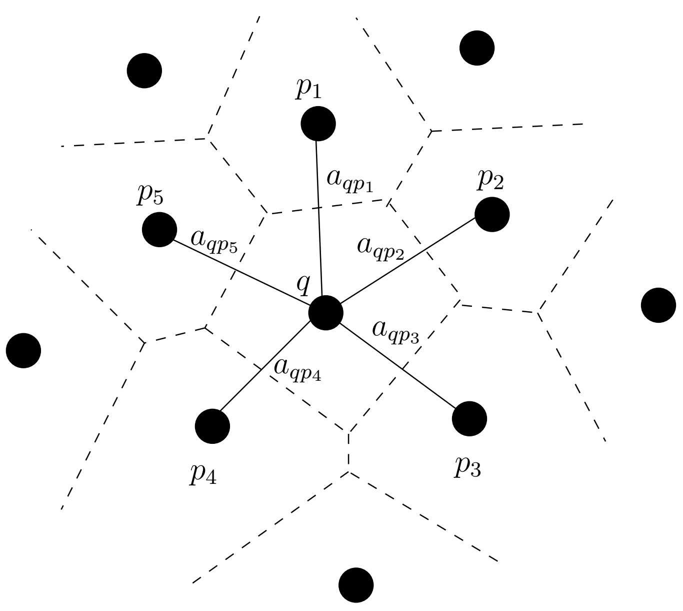

Figure 1.1: Point q with neighbourhood connectivity

flow governing equations with domains represented using unstructured three dimensional meshes. The problem reduces to solve a system of linear equations. There are many numerical methods to solve system of linear equations, out of which Gauss-Seidel iterative solver is an efficient solver to get a good approximation. In Gauss-Seidel method all variables start with an initial guess and solve the equation at each vertex iteratively using the updated values of neighboring vertices until the mean change among all the vertices comes below a certain tolerance. Geometric multigrid is a method to accelerate the iterative solver. In this method the problem is solved in a coarser representation first and this solution is interpolated back to finer graphs to get better approximation. Solving the coarser representation is faster because there are lesser number of vertices. This process is done recursively, which makes the solver converge faster. Solving this problem on very large input is a time consuming process. The inputs considered here are having size of the order of millions. Since the problem is data intensive and data parallel, GPUs can be used to accelerate the execution time. The simple version of this problem is that the points are aligned on a structured grid, in which the connectivity for every point is regular. In general case the points can be anywhere in the space. The parallelization of this problem for GPUs is not straight forward because of the irregularity.

1.2 Related Work

The unstructured geometric multigrid for GPUs are less studied and implemented in the past years. There are several works in which GPUs are used to solve problems on unstructured grids[1, 2]. The multgrid methods are recived moderate attention in the past years, but most of them are using structured grid[3, 4]. The multigrid methods for unstructured grids are studied and implemented in [5, 6, 7, 8, 9]. Among these an efficient method is developed by Miller et. al.[8] in which the multigrid levels are created in such a way that the points selected from the fine grid are well spaced. The work by Brune et. al.[9] simplified the method and improved the preprocessing time. A geometric multigrid solver on unstructured grid on GPUs is done by Geveler et. al.[10]. This work uses the Jacobi method for solving, which is very computational intensive compared to Gauss-Seidel iterative solver.

1.3 Overview of our work

In this work an unstructured geometric multigrid solver is implemented for two and three dimensional domains. For multigrid implementation on unstructured grids different graph coarsening schemes are investigated. The graph coarsening scheme used in the final implementation is based on the works by Miller et. al.[8] and Brune et. al.[9]. Also we implemented and showed the speed up got in two dimensional geometric multigrid on structured grid. These structured and unstructured geometric multigrid solver is implemented in GPU using an improved graph representation.The CFD problem used for experiment is the steady and unsteady heat flow problem in unstructured meshes. The combination of multigrid algorithm and GPU implementation for the steady problem on a 1.6 million mesh gives 1630 times speed up compared to non-multigrid CPU implementation.

1.4 Thesis Outline

The thesis is structured as follows. Section II describes about the GPU architecture, programming model and performance optimizations. The geometric multigrid solver is explained in Section III. In Section IV, we describes the GPU implementation of the problem and the data structures used for this. Section V explains the experiments carried out and the results. Conclusion and future work are presented in Section VI.

Chapter 2

GPU Computing Model

Graphics Processing Units (GPU) were specialized device designed to reduce the workload on the Central Processing Units (CPU) when computing graphic or video intensive tasks. Over the past few years GPU has evolved from a graphic rendering chip to a parallel programming processor which can be used for general purpose programming, often called GPU Computing. Initially GPUs had all its functions hard-wired, but over time it is made programmable by introducing CPU-like light weight processors. As the processing power increased massively, the research community developed methodologies to use these resources for compute intensive task called General-Purpose computation on Graphics Processing Units (GPGPU)[11]. The implementation overhead occurred while using the graphics accelerator for non-graphics application motivated the GPU manufacturers, Nvidia and AMD/ATI to modify their hardware and to create APIs for better support to the GPGPU community. This advancement made the GPUs to change from graphics accelerator to general-purpose application accelerators, which can accelerate data parallel compute intensive part of any application. CUDA is a hardware and software co-processing architecture for parallel computing developed by NVIDIA. CUDA-C is a programming interface provided by NVIDIA for easy development of programs to execute on GPUs. It consist of a small set of extensions to C and a runtime library. The reader is referred to the CUDA programming guide[12] for more details. In GPU computing the term *host* is used to refer to the CPU and the term *device* is used to refer to the GPU. This section explains the GPU architecture, CUDA programming model and performance considerations.

2.1 Hardware Architecture

The major difference between CPU and GPU is that, CPU is latency oriented while GPU is throughput oriented. CPU focuses on minimizing the latency of a single sequential task and GPU focuses on maximizing total throughput. Since GPU is specially designed for compute intensive, highly parallel applications, more transistors are devoted to data processing rather than data caching and flow control. A typical GPU consist of same elements as a normal CPU; Arithmetic Logic Unit (ALU), Control Unit, Cache and DRAM. But the architecture of GPU is different from CPU. GPU consists of a set of streaming multiprocessors (SM). Each SM contains a number of simpler processor cores called CUDA cores. An SM is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called Single-Instruction

Multiple-Thread (SIMT). It is based on the Single-Instruction Multiple-Data model (SIMD), which describes multiple processing elements that perform the same operation on multiple data simultaneously. Figure 2.1 shows the layout of a GPU in Nvidia Fermi architecture. It consists of 512

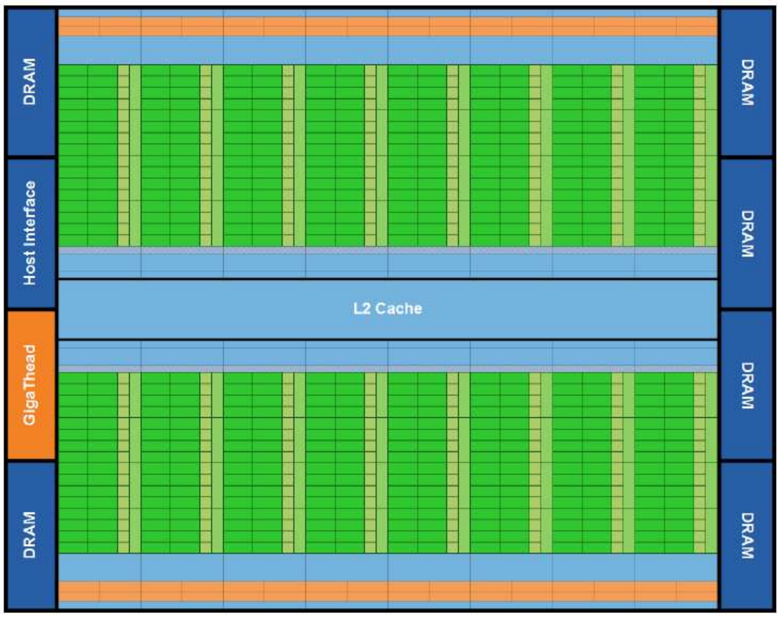

Figure 2.1: Nvidia Fermi Architecture(Source: FERMI White paper[13])

CUDA cores organized in 16 SM of 32 cores each. A CUDA core execute a floating point or integer instruction per clock for a thread. The GPU has 6 GB of DRAM and the host interface connects the GPU to the CPU via PCI-Express. The GigaThread global scheduler distribute thread blocks to SM thread schedulers.

Each streaming multiprocessor contains 32 CUDA cores. The detailed architecture of a streaming multiprocessor is shown in Fig 2.2. Each SM has 16 load/store units which calculate source and destination address to be calculated for sixteen threads per clock. The SM contains 4 Special Function Units (SFU) for computing transcendental function such as trigonometric, exponential and logarithmic functions. The SM has 64KB of on-chip shared memory which enables threads within same thread block to co-operate and to reduce off-chip traffic. The SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Each SM has two wrap schedulers and two instruction dispatch units. The SM groups the threads into warps and each warp gets scheduled by a warp scheduler for execution. The threads in a warp executes one common instruction at a time. Many warps can reside in a multiprocessor and the warp scheduler switches between warps to hide latency. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called latency.

2.2 CUDA Programming Model

The CUDA programming model is designed to overcome the challenges faced in using GPUs for general purpose programming. The CUDA programming model enforces co-operation between hard-

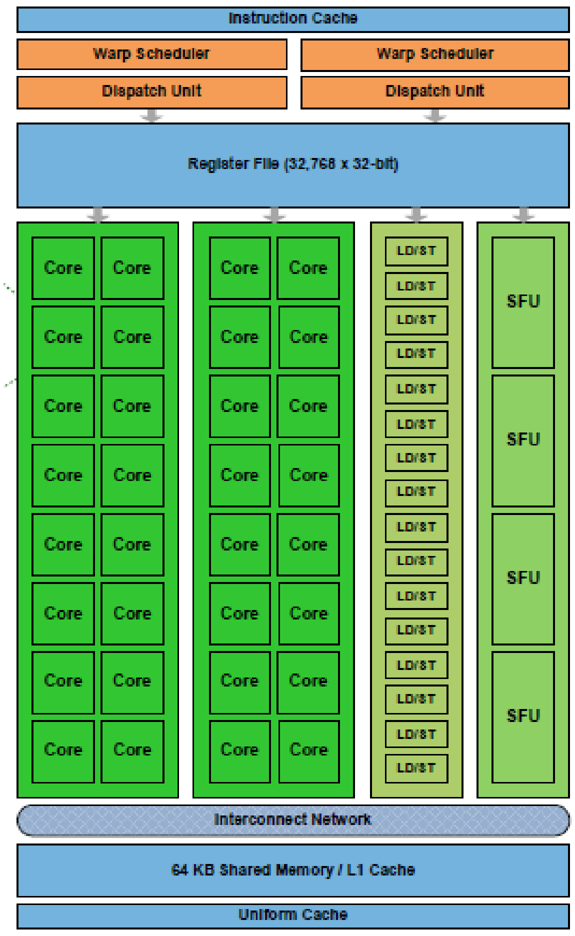

Figure 2.2: Fermi Streaming Multiprocessor (SM)(Source: FERMI White paper[13])

ware, software and different tools to compile, link and execute programs on GPU. In the CUDA programming model, compute-intensive tasks of an application are grouped into an instruction set and passed on to the GPU such that each core works on different data but executes the same instruction.

2.2.1 Compiler model

A CUDA source file contains a mixture of host code, the code that execute on the CPU and device code, the code that execute on the GPU. The CUDA compiler separates the host and device code. The device code is compiled to an assembly form called Parallel Thread Execution (PTX) code. This PTX code can either be converted to binary form, which is called a *cubin* object or loaded by the application at runtime and get compiled by a mechanism called *just-in-time compilation*. Just-in-time compilation increases application load time, but allows application to benefit from latest compiler improvements. The compiler then replaces the CUDA constructs for device code invocations used in the host code with necessary CUDA C runtime function calls to load and launch the device code from PTX code or *cubin* object. The host code is then compiled and linked by the programmers C/C++ compiler.

2.2.2 Execution Model

The core of the CUDA programming model is the special user defined functions called kernels, that, when called, are executed N times in parallel by N different CUDA threads using different data streams. The number and organization of the threads to be created can be determined at compile time or runtime and is called the execution configuration of that kernel. The threads are organized into a two level hierarchy block and grid as shown in Fig 2.3. At the first level threads

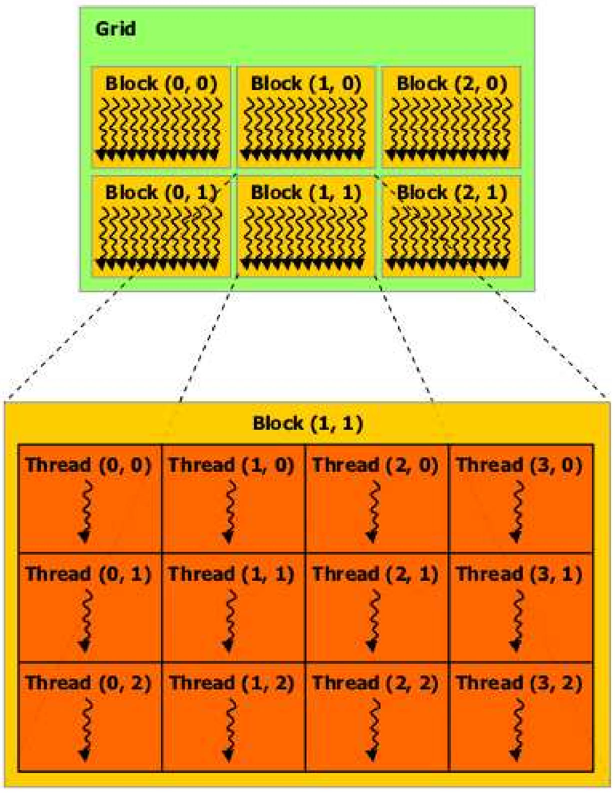

Figure 2.3: CUDA thread organization(Source: CUDA Programming Guide[12])

are grouped into thread blocks. The size of the thread block is determined by the programmer and is arbitrary. Each block run independent of each other, so that they can be scheduled across any streaming multiprocessor. Thread blocks are in turn grouped into grids. The number of blocks in a grid is determined by the size of data the application dealt with. The blocks and grids can have one- or two- or three-dimensional organization. This organization allows the programmer to easily map the threads to multi-dimensional data structures. Each block within the grid is given a unique blockID and is identified through the built in *blockIdx* variable, from the kernel. Also each thread in a thread block is given a unique threadID and is identified through the built in *threadIdx* variable, from the kernel. Both *blockIdx* and *threadIdx* is a 3-component vector describes the index in three-dimensions. The dimension of the grid and thread block are available in the built in *gridDim* and *blockDim* variables respectively. The kernel invocation is accompanied by the execution configuration and has the following syntax: kernel_name <<<grid_size, block_size>>>(). All the threads in a particular thread block is executed in a single multiprocessor. The maximum number of threads that a thread block can contain depends on GPU model, since all threads of a block are expected to reside on the same multiprocessor and must share the limited memory resources of that multiprocessor. All the threads from a particular thread block have access to the same shared memory and can synchronize together through __syncthread() function call. On the other hand, threads from different thread blocks cannot synchronize and can exchange data only through the global memory. The data that the kernel functions access is needed to be transferred to the device global memory before the kernel is invoked. CUDA API provide two functions for this purpose, cudaMalloc() and cudaMemcpy(). These functions allocate memory on the GPU and copy data from the CPU memory into the device memory respectively. cudaFree() function is used to free memory on the device.

2.2.3 Memory Model

CUDA threads use a seperate memory space which resides on GPU. The GPU memory is organized into a three level hierarchy as shown in Fig 2.4.

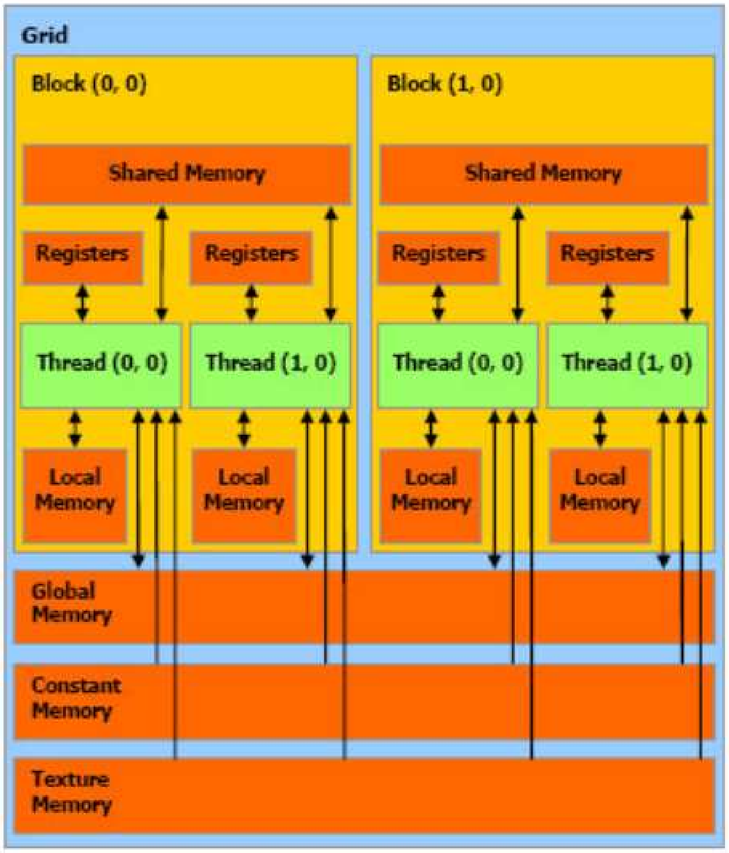

Figure 2.4: CUDA memory model(Source: CUDA Programming Guide[12])

Device Memory

This is the largest memory in the GPU and also the memory with the highest latency. It can be compared with the DRAM of a normal CPU. The device memory is virtually divided into global memory, local memory, constant memory and texture memory. The global memory is the memory which every thread can access and it is the only part of the device memory CPU can read and write. The local memory space resides in device memory and is a private unit for each thread processor. Variables are automatically placed in local memory by the compiler if they do not fit in register. The constant memory provides faster and more parallel data access paths for CUDA kernel execution than the global memory. The texture memory on the other hand is optimized for 2D spatial locality and should be preferred over global device memory when coalesced read cannot be achieved. The GPU can only read from constant and texture memory while CPU can only write.

Shared Memory

Every SM has its own shared memory that is shared between all threads in a block. Because shared memory resides on-chip, it is much faster than global memory. The data that is reused by the threads in a thread block can be loaded to shared memory for improving the performance. It can also be used to share the data between threads in same thread block.

Registers

Every thread has its own set of registers. This is the fastest memory in the memory hierarchy. Basically, every access to a register by the thread processor is immediate, ie, it takes zero extra clock cycles per instruction.

2.3 Performance Optimizations

The challenge for a CUDA developer is not only the parallelization of the code, but also the optimization of the code for best performance[14]. The important optimization strategies are maximizing processor utilization, maximizing memory throughput and maximizing instruction throughput. Detailed performance optimization practices are given in Nvidia's Best Practices Guide[15].

The GPU takes on the order of hundreds of clock cycles to fetch a single element from global memory. The massively threaded architecture of GPU is used to hide this memory latency by switching between warps. Once a warp stalls for a memory fetch, the GPU switches to the next warp which is ready to execute. So a good strategy is to create a lot of threads to completely hide the memory latency, which in turn maximize the processor utilization. When all threads in a warp execute a load instruction, the hardware detects whether the threads access consecutive global memory locations. In that case, the hardware combines all of these accesses into a consolidated access to the device global memory. This is called memory coalescing. If the threads in a warp access data scattered in the memory, then more unused data are transferred between global memory and the cache in addition to the data accessed by the threads. So a good strategy is to store data accessed by threads in a warp, in consecutive global memory locations to maximize the memory throughput. The GPU executes instructions in such a way that same instruction is simultaneously executed by threads in a warp. If there is any branching instruction in the kernel, it affect the instruction throughput by causing the threads of the same warp to diverge, ie, to follow different execution paths. If this happens, the threads in a warp execute different execution path in serial. One technique to avoid expensive branching is to arrange the elements according to the branch and thus make sure the threads within each warp will execute them without branching.

Chapter 3

Geometric Multigrid Solver

The generally used neumerical methods to solve system of linear equation are direct methods like Gaussian elimination, LU factorization etc and iterative methods like Jaccobi method, Gauss-Seidel method, Conjugate gradient method etc. Direct method compute the solution in a finite number of operations. In iterative, starting from an initial guess the solution is improved step by step until it reaches the sufficient accuracy. Iterative methods are not expected to terminate in a fixed number of steps. For large systems the direct methods are inefficient where the iterative methods perform very well. Multigrid (MG) offer an alternative to these methods for efficiently solving large sparse linear systems of equations, in particular for those arising in the finite element, finite difference and finite volume discretization of partial differential equations (PDEs). Multigrid methods enable solution to be obtained in a given level of convergence in O(N) operations, where N represents the number of unknowns[16, 17, 18]. The theoretical foundation of multigrid methods is that high-frequency error components can be eliminated quickly using elementary iterative methods (like Jacobi or Gauss-Seidel), while low-frequency error components are more persistent.The iterative methods mentioned above are good in relaxing highly oscillating errors efficiently but it will relax the smooth errors very slowly. However, low-frequency errors on a fine mesh resolution appear as high-frequency errors on a coarser mesh. Multigrid is thus an inherently recursive defect correction method acting on a hierarchy of different mesh resolutions: Starting on a fine mesh, the error is smoothed with a few iterations of an elementary iterative scheme. Then, the resulting defect is restricted to a coarser mesh, where the algorithm is recursively applied. At the coarsest mesh, the so-called coarse grid problem is solved exactly, and the resulting defect correction is prolongated to the next finer level where any remaining high-frequency errors are eliminated by post-smoothing. One such solver iteration is called a multigrid cycle, and different cycle types exist, depending on the order in which different mesh resolutions are traversed. Thus multigrid strategies are generally considered as convergence accelerator technique rather than solution methods.

Two fundamentally different multigrid schemes exist, Algebraic multigrid (AMG) and Geometric multigrid (GMG). In algebraic multigrid no information concerning the grid is used on which the governing partial differential equations are discretized. In geometric multigrid, coarse grids are constructed from the given fine grid and coarse grid corrections are computed using discrete systems constructed on the coarse grid. Although the GMGs applicability is difficult as it requires explicit information on the hierarchy of the discrete system, when it can be applied, GMG is far more efficient than its algebraic version, the AMG method. GMG method comprise a group of algorithms for solving differential equations using a hierarchy of discretizations. The key steps in the multigrid method are as follows:

- Relaxation or Smoothing: Reduce high-frequency errors using one or more smoothing steps based on a simple iterative method, like Jacobi or Gauss-Seidel.
- Restriction: Restrict the residual on a finer grid to a coarser grid.
- Prolongation: Represent the correction computed on a coarser grid to a finer grid.

3.1 Structured and Unstructured Multigrid

The important part of geometric multigrid is the creation of hierarchy of grids. Each coarser grid should approximate the topology of the higher level finer mesh. In CFD the domains are represented in two ways: one is using structured grid, where every interior node is connected to same number of nodes and other is using unstructured grid, which has no regularity in the connectivity. In structured grid the creation of topologically approximated coarse grids is easy. For a coarse grid, select alternate nodes in each dimension as shown in Fig 3.1.

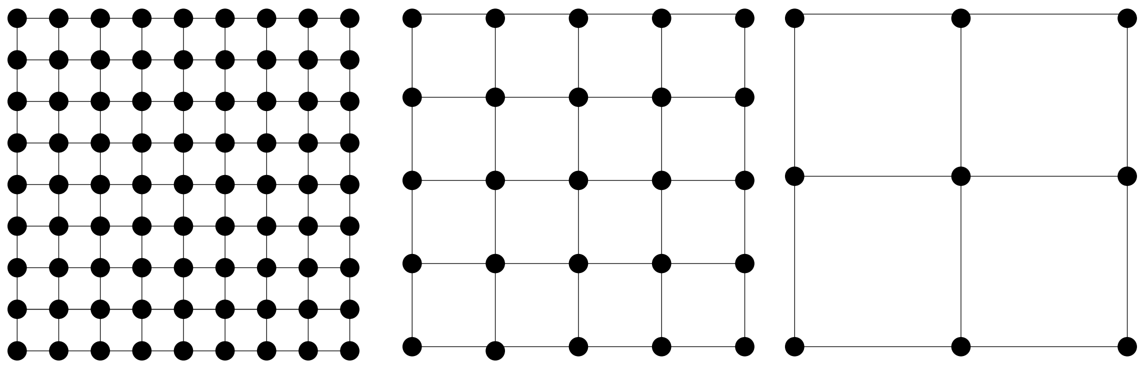

Figure 3.1: Structured Multigrid Levels

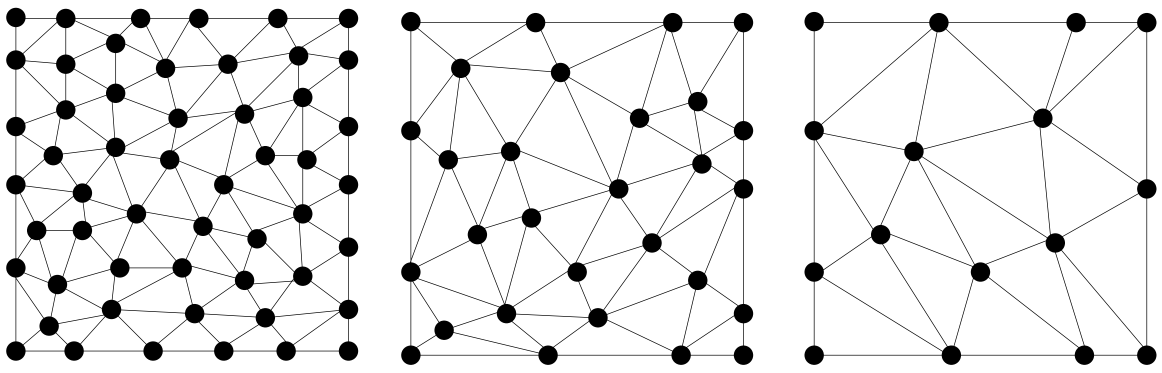

Figure 3.2: Unstructured Multigrid Levels

For unstructured grids the generation multigrid levels is difficult. Unless the coarser grid topologically approximates the finer grid the multigrid algorithm does not give large improvement. Figure 3.2

shows the multigrig levels for an unstructured grid. Different schemes can be used to create coarser grid for unstructured grids.

3.2 Unstructured Multigrid Level Generation Schemes

The different levels for geometric multigrid is generated by coarsening the initial fine grid using a graph coarsening algorithm recursively. The general method is to select a subset of vertices from the fine graph and triangulate(2D) or tetrahedralize(3D) to create the next level coarse graph. The effectiveness of the geometric multigrid depends heavily on the vertices which are selected for coarser level. The vertices selected for the coarser level should approximate the topology of the finer level graph. Once the vertices for the coarse level is selected it is re-meshed to generate the coarse level grid. The remeshing method used here is delaunay tetrahedralization of the coarse subset of vertices. In a delaunay tetrahedralization of a set P of points, no point in P is inside the circumsphere of any tetrahedron of the tetrahedralization. Delaunay method maximize the minimum angle and they tend to avoid skinny triangles or tetrahedrons. The *TetGen*[19] library is used for delaunay tetrahedralization. In this section different schemes for selecting the coarse level vertices is discussed.

3.2.1 Maximal Independent Set

The vertices are selected with the goal of reducing graph size while maintaining a good distribution of vertices. The most widely used method for this purpose is Maximal Independent Set (MIS)[7, 6, 5]. Given an undirected graph $G(V, E)$, a set $V' \subseteq V$ is an independent set if $u, v \in V' \Rightarrow (u, v) \notin E$. An independent set is maximal if no node in $V - V'$ can be added to V' without violating independence property. This scheme select coarse vertex set such that no two vertices in the coarse grid are adjacent in the fine grid. This can be implemented using a simple linear time algorithm where at each stage a vertex is selected for inclusion in MIS and its neighbours are marked for exclusion.

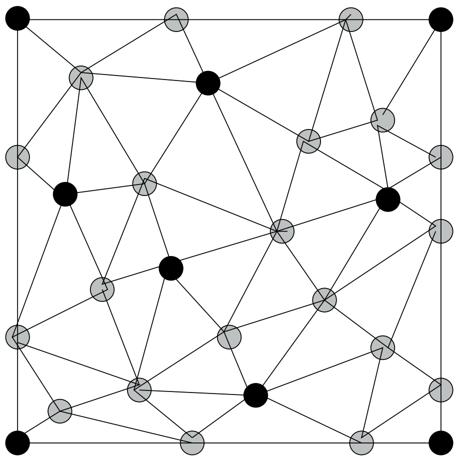
Figure 3.3: Maximal Independent Set(MIS)

In this coarsening scheme all vertices which are not selected for coarse grid has at least one adjacent vertex in the coarse grid so that every non selected vertex can get interpolated with the correction from the coarse grid while prolongation. The maximal independent set of the graph is a good choice for selecting coarse graph vertices as it is simple to implement and gives good results.

Algorithm 1 Arbitrary MIS

```
Require: Graph $G(V, E)$
Output: MIS
1: for all $v \in V$ do
2:     if all $Adj(v) \notin MIS$ then
3:         $MIS \leftarrow MIS \cup \{v\}$ {Include $v$ in $MIS$}
4:     end if
5: end for
```

Another method for finding MIS is using a greedy heuristic algorithm. This algorithm uses a particular ordering of the vertices for finding the MIS. In this method the vertex with minimum degree is removed from the graph and added to a stack. This operation is repeated until all vertices are added to the stack. Then the vertices are considered for inclusion into MIS in the popping order from the stack.

Algorithm 2 Greedy MIS

```
Require: Graph $G(V, E)$
Require: Stack $S$
Output: MIS
1: $MIS \leftarrow \phi$
2: while $V \neq \phi$ do
3:     $v \leftarrow$ Minimum degree vertex in $G$
4:     Push $v$ onto stack $S$
5:     $V \leftarrow V - \{v\}$ {Remove $v$ from graph $G$}
6: end while
7: while $S \neq \phi$ do
8:     $v \leftarrow$ Pop from stack $S$
9:     $V \leftarrow V \cup \{v\}$ {Add $v$ to graph $G$}
10:    if all $Adj(v) \notin MIS$ then
11:        $MIS \leftarrow MIS \cup \{v\}$ {Include $v$ in $MIS$}
12:    end if
13: end while
```

The greedy MIS algorithm gives better multigrid result compared to the arbitrary MIS.

3.2.2 Dominating Set

In this scheme the concept of domination[20, 21] in graph theory is used for coarsening. The idea behind in the scheme is to improve the interpolation during the prolongation operation and thus to improve the performance of multigrid. This scheme gives a lower bound on the number of nodes adjacent to a node in a coarser grid from a node in the finer grid which is not selected. There are two versions of this scheme, k-dominating set and α-dominating set[22].

k-Dominating Set

A set X is called a k-dominating set if every vertex not in X has at least k neighbours in X. The vertices in the k-dominating set is selected as the coarser vertex set. So every dropped nodes has at least k adjacent nodes in the coarser grid.

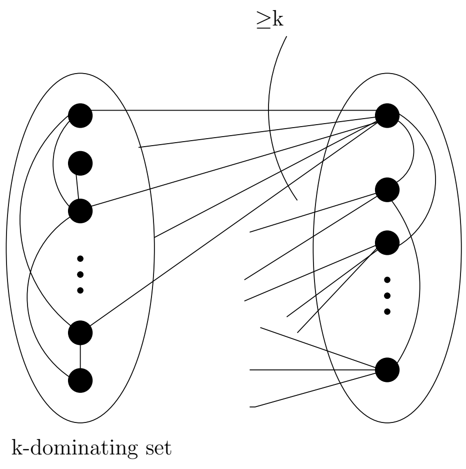

Figure 3.4: k-Dominating Set

α-Dominating Set

Let α be a real number satisfying $0 < \alpha \leq 1$. A set $X \subseteq V(G)$ is called an α-dominating set of G if $|N(v) \cap X| \geq \alpha d_v$ for every vertex $v \in V(G)X$, i.e. v is adjacent to at least $\lceil \alpha d_v \rceil$ vertices of X. he vertices in the α-dominating set is selected as the coarser vertex set. In k-dominating set fixing a k value is difficult. So by using α-dominating set α fraction of a dropped node's adjacent nodes will be there in the coarser grid.

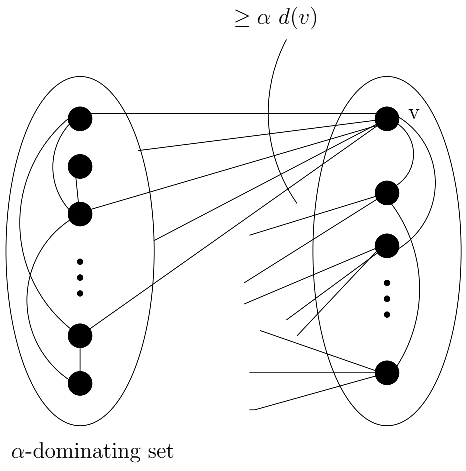

Figure 3.5: alpha-Dominating Set

This coarsening scheme fails to give a good topological approximation to the finer grid.

3.2.3 Function Based Coarsening

For getting good multigrid performance the observation is that the coarse representation should topological approximate the finer level structure. The coarse level points should be a well spaced subset of fine level points. In the above mentioned schemes the coarse node selection is solely based on the node connectivity in the fine grid. The geometric spacing of the points are not considered in the coarse points selection.

This graph coarsening algorithm was developed by Miller et. al.[8] considering the geometric spacing of nodes. This method defines a spacing function, $Sp(v)$, for describing the size and spacing of the grid. This function is defined at the grid points. In practice the nearest neighbour distance to each node is a good spacing function. It then defines a constant C, $1 < C \leq 2$, by which the spacing function at each point is multiplied for coarsening the fine grid. A *conflict graph* is created for the fine grid such that, the vertex set is same as the vertex set of the grid and the edge set as,

$$E^c = \{(i,j) : C(Sp(i) + Sp(j)) > \|i - j\|, \forall i, j \in V\}$$

where V is the vertex set. The coarse vertex set is found by selecting maximal independent set on the conflict graph. The complexity for constructing conflict graph is $O(|V|^2)$ because all pairs of vertices should be compared.

Brune et. al.[9] modified the conflict graph definition and improved the running time. The conflict condition is simplified in such a way that the spacing condition is restricted to take into account only distance between vertices connected by graph edges rather than all pairs of vertices. In this method the conflict graph edge set is given by,

$$E^c = \{(i,j) : C(Sp(i) + Sp(j)) > \|i - j\|, \forall (i,j) \in E\}$$

where E is the edge set of fine graph. The The complexity for constructing conflict graph in this method is $O(|V| + |E|)$. In both case the MIS is found in an arbitrary way together with the construction of conflict graph.

In this work the conflict graph is explicitly computed and the MIS is found using the greedy MIS algorithm. The use of this algorithm shows improvement in the multigrid performance compared to arbitrary MIS algorithm. A hierarchy of graphs then be created by recursively applying the coarsening procedure to each newly created coarse graph with constant C and some recalculated Sp. This may be done until the graph fits some minimum size requirement, or until the creation of a desired number of graphs for the hierarchy.

3.3 Gauss Seidel Iterative Method for Relaxation

At each level in the multigrid a smoothing operator is needed for reducing high-frequency errors. For matrices resulting from unstructured grids we must use general update methods than direct methods such as Gauss elimination, LU decomposition etc. The simplest of these are the Jacobi and Gauss-Seidel methods. In both cases, the nodes are visited in sequence and at each node the value is updated. The two methods differ in the values of the neighboring nodes that are employed. In case of the Jacobi method, the old values of all the neighbors are used whereas in the Gauss-Seidel method, the latest values of all the neighbours that have already been updated during the current sweep and old values of the neighbours that are yet to be visited are used. In general the Gauss-Seidel method has better convergence characteristics than the Jacobi method and is therefore most widely used. For vector and parallel hardware Jacobi method is generally used due to the ease of parallelization. In this work we are using Gauss-Seidel method smoothing operator. In Gauss-Seidel method all nodes cannot be updated simultaneously because of the dependency of a node to its neighbours that are updated in the current sweep. By using graph coloring technique the nodes are divided into independent sets such that nodes in same set can be updated simultaneously.

Chapter 4

GPU Implementation

This chapter explains the methods used to implement the problem in structured and unstructured grids.

4.1 Structured grid

In Gauss-Seidel iterative method each node is dependent on the adjacent nodes. So all nodes cannot updated at the same time. In structured grid the red-black scheme can be used to solve this issue[23, 24]. The nodes are alternatively colored with as red and black as shown in Fig 4.1.

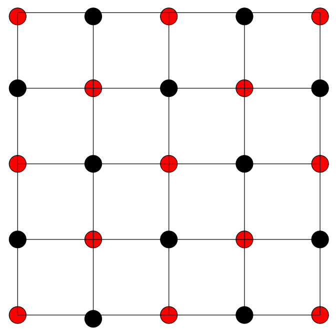

Figure 4.1: Red-Black scheme

All red nodes have only black nodes as adjacent and for black nodes have only red nodes as adjacent. So the nodes with same color can be updated simultaneously. One Gauss-Seidel iteration will happen in two steps. This method of parallelizing structured grid for GPU is used for accelerating single and two phase CFD flow solvers[25, 26].

4.2 Unstructured grid

Solving graph processing problems in GPUs are challenging because that require highly irregular data access.Graph problems represent a worse case scenario for coalescing parallel memory access. on GPUs. The GPU memory layout is optimized for graphics rendering and cannot support user defined data structures efficiently. However in CUDA programming model the memory is treated as general arrays and can support more efficient data structures. It is critical for the efficiency of GPU method that parallel data coalescing is maximized and that number of conditional breaking point in the algorithm is minimized. In the implementation three different GPU kernel functions are created. First one is KERNEL_ITERATE, which updates all points of the graph once. Second one is KERNEL_PROLONGATE, which transfers the coarse graph corrections to fine graph. The third kernel is KERNEL_SUM which finds the sum of values in an array and is used for finding root mean square value of error, which is used in the convergence condition.

4.3 Graph Coloring

Coloring of a graph is labelling of the graph's vertices with colors such that no two vertices sharing the same edge have the same color. A subset of vertices assigned to the same color is called a color class, every such class forms an independent set, ie, no two of which are neighbours. A coloring using at most k colors is called a k-coloring. The smallest number of colors needed to color a graph G is called its chromatic number, and is often denoted χ(G). Since the updates of nodes can be carried out in any order in Gauss-Seidel method, the order can be of the color classes, ie, each color class one after other. In a color class all nodes are independent to each other. So all nodes in a particular color class can be updated in parallel.

The choice of the coloring algorithm is determined only by practical considerations. Among these are the performance of the algorithm, ie, the number of colors being used, and its efficiency, ie, the time needed to color the graph. A χ(G)-coloring might be the optimal solution for the given problem. But the determination of χ(G) is known to be a NP-complete problem for general graphs. Practically, one is interested in finding good approximations for χ(G) in polynomial time. One can try to use a simple coloring procedure known as greedy coloring, which is a coloring of the vertices of a graph formed by a greedy algorithm that considers the vertices of the graph in sequence and assigns each vertex its first available color. A commonly used heuristic ordering strategy for greedy coloring is to choose a vertex v of minimum degree, order the remaining vertices, using same strategy and then place v last in the ordering. For a graph of maximum degree Δ, any greedy coloring will use at most Δ + 1 colors.

Algorithm 3 Greedy graph coloring algorithm

```
Require: Graph $G(V, E)$
Require: Stack $S$
 1: while $V \neq \phi$ do
 2:     $v \leftarrow$ Minimum degree vertex in $G$
 3:     Push $v$ onto stack $S$
 4:     $V \leftarrow V - \{v\}$ {Remove $v$ from graph $G$}
 5: end while
 6: while $S \neq \phi$ do
 7:     $v \leftarrow$ Pop from stack $S$
 8:     $V \leftarrow V \cup \{v\}$ {Add $v$ to graph $G$}
 9:     Assign lowest numbered color to $v$ which is not same as that of its neighbors
10: end while
```

4.4 Graph Representation

The memory representation of graph heavily affects the performance of GPU implementation. There are two types of data needed to be stored in the device memory. One is data for each vertex such as value at the vertex, degree of the vertex etc. and other is data for each edge such as the adjacent vertex indexes, scale factor for each neighbour, distance to each neighbour etc. The vertex data is reordered according to the color of the vertices, ie, the vertices with same color is kept adjacent in the memory. A set of pointers are kept for storing the starting of each color. The KERNEL_ITERATE function takes the start index and end index of each color and is repeatedly invoked for each color. The implementation creates as many threads as the number of vertices in the particular color class. The reordering of vertex data according to the color is needed for coalesced memory access. Otherwise the vertex data of a particular color class will be scattered in the memory resulting the un-coalesced memory access by thread warp. The reordering also makes the mapping of threads to the data easy. The data access for non-reordered and reordered array is shown in Fig 4.2 and Fig 4.3 respectively.

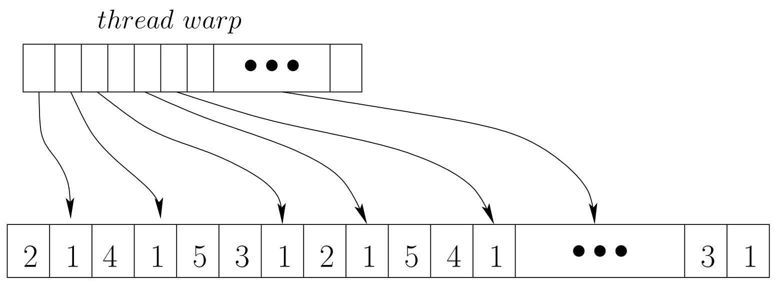

Figure 4.2: Un-coalesced memory access

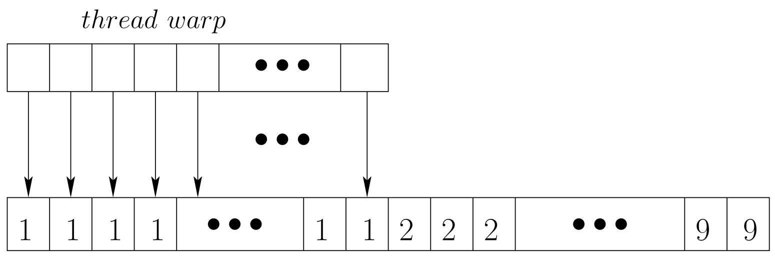

Figure 4.3: Reordered for coalesced memory access

For storing data at edges, commonly used data structure is adjacency matrix. It is not suitable for large sparse graphs because of $O(|V|^2)$ space requirements. Adjacency list is a more practical representation for sparse graphs requiring $O(|V| + |E|)$ space. A graph representation found in [27, 28, 29, 30, 31] is using a compact adjacency list representation, with edge data lists packed into a single large array. A pointer array of $O(|V|)$ is also kept, which stores for each vertex the index in compact adjacency list where its adjacency list starts.

But in the compact adjacency list representation the memory access is highly un-coalesced. All threads in a warp executes the same instruction at a time. While iterating through the edge data, all threads access the respective edge list. So the access to the memory by these threads are scattered as shown in Fig 4.4.

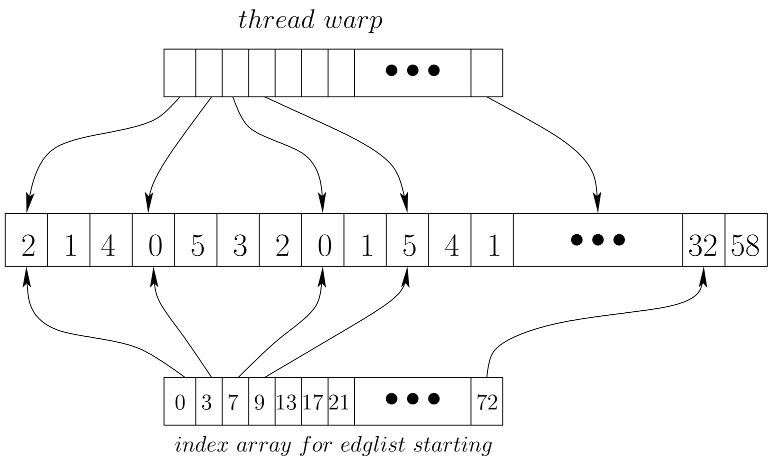

Figure 4.4: Un-coalesced memory access for compact adjacency list

In this work we are using a modified edge data representation, a column major matrix representation. In this representation the edge data is stored in a matrix of size $\Delta \times |V|$. The first edge data of all the vertices is stored in the first row of the matrix, second edge data in second row of the matrix and so on. The memory access to this representation is coalesced as the concurrently accessed edge data are co located as shown in Fig 4.5. This graph representation can be used for memory coalescing in GPU implementation of other graph algorithms. The space requirement for this representation is $O(\Delta V)$, which is higher compared to compact adjacency list representation. This can be reduced to $O(|V|+|E|)$ by storing vertices in the descending order of degree and storing the rows as separate arrays. But in this work the vertices are ordered according to the color. So the sorted representation is not possible to implement. Also practical problems that uses this model have very small Δ value.

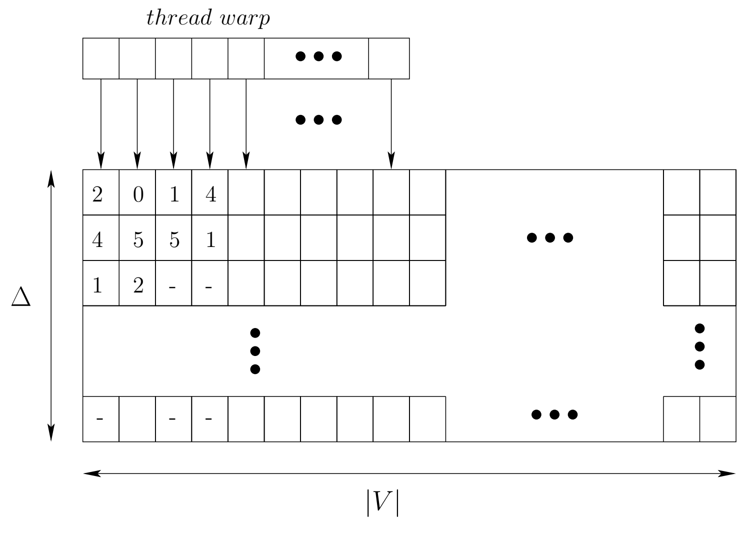

Figure 4.5: Coalesced memory access for column major adjacency list

4.5 Multigrid Implementation

The multigrid levels are created in the preprocessing part as described in section III.A. All grid data is stored in the device global memory in series as described in the previous section. For vertices which are present in the next coarser level, a index for the corresponding vertex in the coarser graph is stored in the finer graph for restriction and prolongation.

4.5.1 Restriction

The restriction is implemented in the KERNEL_ITERATE function. While updating the value of a vertex, if the vertex is present in the next coarser graph it find the residual for that vertex and store it in the coarser graph vertex.

4.5.2 Prolongation

The prolongation operation is done by the KERNEL_PROLONGATE function. The number of thread created for prolongation operation is equal to the number of vertices in finer graph. For a finer graph vertex if that vertex is present in the coarser graph, its correction is directly added to the finer graph vertex. Otherwise the corrections of its selected neighbours are added up in the ratio of its distance to the correcting vertex.

Chapter 5

Experiments and Results

In this work the steady and unsteady heat transfer problem is used for the experiments. Heat transfer is the transfer of thermal energy from a body at a high temperature to another at a lower temperature. In unsteady heat transfer problem the temperature within the system vary with time. The unsteady heat transfer problem is important in CFD as other physical processes like potential flow, mass diffusion, flow through porous media etc. are governed by similar mathematical equations. Electromagnetic field theory, diffusion models of thermal radiation and lubrication flows are further examples of phenomena governed by this type equation.

5.1 Steady and unsteady heat transfer problem

The steady heat transfer is governed by the equation,

$$\nabla . k \nabla T = 0$$

where T is the temperature and k is conductivity. The discretization for the above governing equation is,

$$a_p T_p = \sum_{i \in N(p)} a_{pi} T_i$$

$$a_{pi} = \frac{k_{pi} s_{pi}}{d_{pi}} \qquad a_p = \sum_{i \in N(p)} a_{pi}$$

where, k_{pi} = mean conductivity of p and i, s_{pi} = interface area between p and i and d_{pi} = distance between p and i

The unsteady heat transfer is governed by the equation,

$$\rho c \frac{\partial T}{\partial t} = \nabla . k \nabla T$$

where T is the temperature, ρ is the density, c is specific heat and k is conductivity. The discretization for the above governing equation is,

$$a_p T_p = \sum_{i \in N(p)} a_{pi} T_i + a_p^0 T_p^0$$

$$a_p^0 = \frac{\rho c v_p}{\Delta t} \qquad a_{pi} = \frac{k_{pi} s_{pi}}{d_{pi}} \qquad a_p = \sum_{i \in N(p)} a_{pi}$$

where, k_{pi} = mean conductivity of p and i, s_{pi} = interface area between p and i, d_{pi} = distance between p and i and v_p = control volume around p and Δt is the time step size[32].

The steady problem is solved on structured and unstructured unit square grid in two dimension and unstructured unit cube in three dimension. All sides or faces except one is set constant at 300 and one side or face is set constant at 600. The unsteady problem is solved on unstructured unit cube in three dimension. All sides or faces except one is set constant at 300 and one side or face is kept sinusoidally varying, starting from 600 at time zero. The time step size given is 0.01 and the experiment is run for 24000 time. The temperature variation given for each time step is

$$T = 600 + 100 \sin \frac{2\pi t}{24}$$

5.2 Experimental setup

The CPU configuration used for executing serial implementation is Intel Xeon CPU X5675 3.07 GHz. The GPU used is Nvidia GeForce GTX 480 with CUDA driver version 5.0. Its clock rate is 1.40 GHz and having 480 CUDA cores.

5.3 Results

In this section the speed up results of the problem in different settings is discussed. The different settings are steady state problem on 2D structured grid, steady state problem on 2D ustructured grid, steady state problem on 3D ustructured grid and unsteady problem on 3D unstructured grid. In the results both the performance improvement by multigrid method and GPU implementation is compared. The multigrid performance got by using different graph coarsening scheme is also compared. The output of our implementation is validated with the output of a standard commercial software.

The metric used to compare the performance of multigrid is *work units*[33]. In normal case the number of iterations can be used as the metric. But in the multigrid method the number of iterations will not be a good metric because the size of grids in different levels are different. So instead of number of iterations, total number of updates normalized to the size of fine grid, called *work units* is used as the metric. The work units are calculated by,

$$work\ units = \frac{Total\ number\ of\ updates}{Number\ of\ points\ in\ fine\ grid}$$

The work units taken for solving the problem with multigrid and without multigrid are compared separately.

Steady state 2D structured multigrid

The steady state problem is solved on a structured 2D grid and the speed up results are shown below. Table 5.1 compares the work units for implementation with out multigrid and with multigrid and the speed up achieved by both GPU implementation and multigrid algorithm as shown in Table 5.2.

Table 5.1: With multigrid vs Without mutigrid (work units)

Grid Size	Without MG	With MG
512 x 512	205520	1781.48
1024 x 1024	674693	2917.46
2048 x 2048	2109429	4563.45

Table 5.2: 2D structured grid time comparison

Grid Size	CPU	CPU-MG	GPU	GPU-MG	Speed up
512 x 512	46 min 37 sec	26 sec	1 min 63 sec	1.26 sec	2219x
1024 x 1024	10 hr 18 min 41 sec	2 min 53 sec	18 min 40 sec	7 sec	5303x
2048 x 2048	128 hr 57 min 15 sec (estimated)	19 min	3 hr 46 min 10 sec	39.42 sec	11776x

Graph Coarsening Scheme Comparison

The different methods based on function based coarsening is compared in Fig 5.1. The method by Brune et. al. improves the preprocessing time compared to the Miller et. al. algorithm and the multigrid performance is same. The method used in this work improves the multigrid performance.

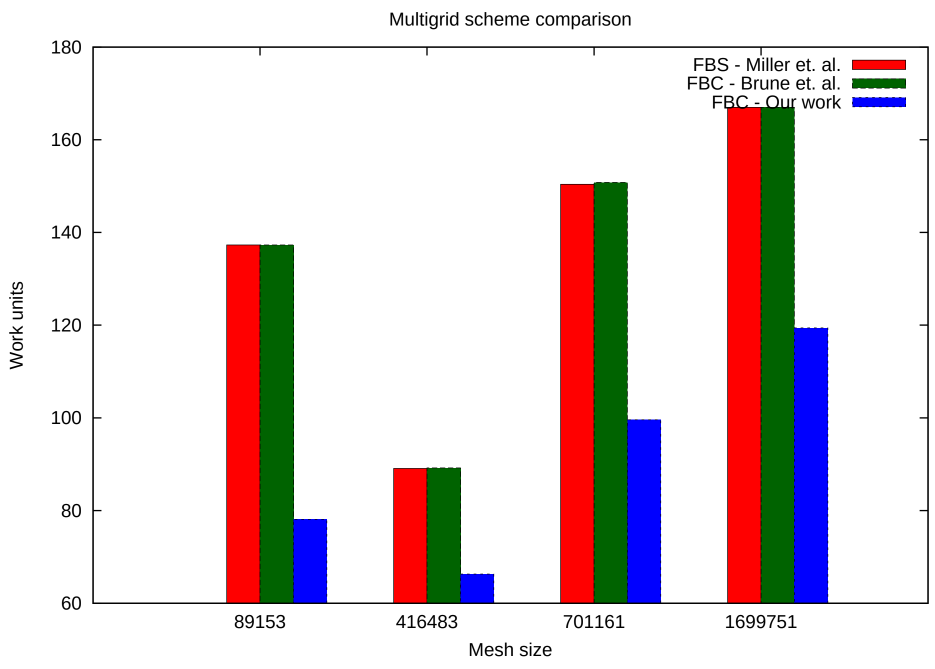

Figure 5.1: Graph Coarsening Comparison

Steady state 2D unstructured multigrid

The steady state problem is solved on a unstructured 2D grid and the speed up results are shown below. Table 5.3 compares the work units for implementation with out multigrid and with multigrid and the speed up achieved by both GPU implementation and multigrid algorithm as shown in Table 5.4.

Table 5.3: With multigrid vs Without mutigrid (work units)

Grid Size	Without MG	With MG
145537	51772	772.68
1866609	61843	504.1

Table 5.4: 2D unstructured grid time comparison

Grid Size	CPU	CPU-MG	GPU	GPU-MG	Speed up
145537	29 min 46 sec	32 sec	1 min	3 sec	595x
1866609	10 hr 9 min 5 sec	5 min 39 sec	12 min 39 sec	9 sec	4060x

Correctness Comparison

The results of steady and unsteady problem implementation in this work is compared with the results of a commercial software as shown in Fig 5.2 and Fig 5.3 respectively. Figure 5.2 compares the temperature in a line along z dimension at $x = 0.1$ and $y = 0.5$. Figure 5.3 compares the temperature at point $x = 0.01$, $y = 0.5$ and $z = 0.5$. The implementation in this work suffers an average error of 1.5%.

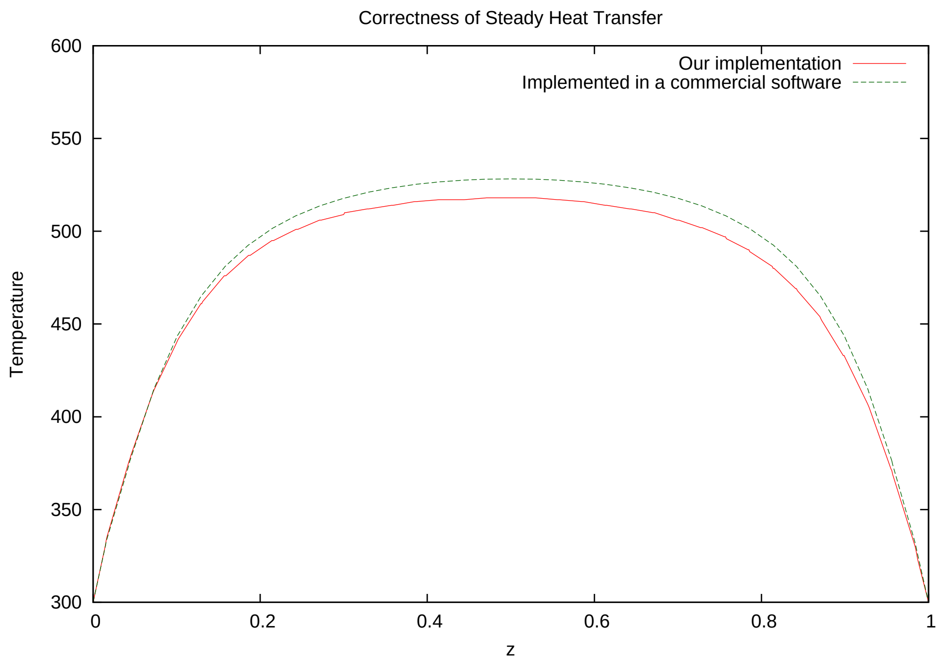

Figure 5.2: Steady state problem correctness

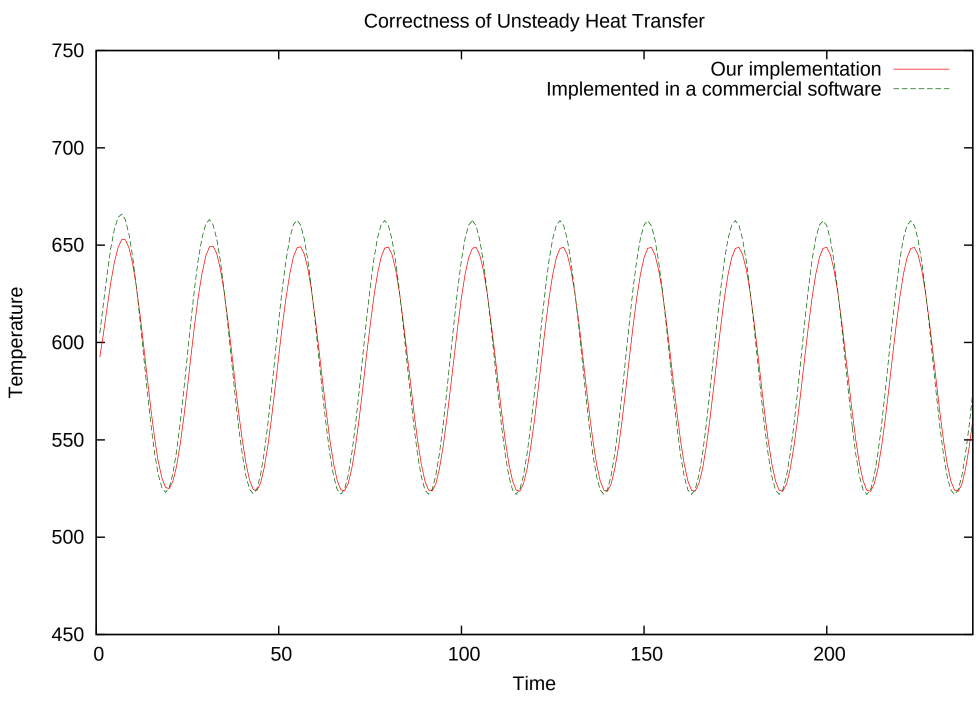

Figure 5.3: Unsteady state problem correctness

Steady state 3D unstructured multigrid

The steady state problem is solved on a unstructured 3D grid and the speed up results are shown below. Table 5.5 compares the work units for implementation with out multigrid and with multigrid and the speed up achieved by both GPU implementation and multigrid algorithm as shown in Table 5.6. The speed up shown in Table 5.6 is the speed up in computation time.

Table 5.5: With multigrid vs Without mutigrid (work units)

Grid Size	Without MG	With MG
416483	1799	66.36
701161	6510	99.63
1699751	10919	119

Table 5.6: 3D unstructured grid time comparison (computation time + preprocessing time)

Grid Size	CPU	CPU-MG	GPU	GPU-MG	Speed up
416483	3 min 54 sec + 46.8 sec	11.7 sec + 58.1 sec	7.5 sec + 65.6 sec	0.68 sec + 83.64 sec	344x
701161	20 min 5 sec + 1 min 57 sec	27.5 sec + 2 min 22 sec	36.45 sec + 2 min 39 sec	1.31 sec + 3 min 27 sec	919x
1699751	1 hr 15 min 20 sec + 10 min 14 sec	1 min 15 sec + 11 min 50 sec	2 min 23 sec + 32 min 45 sec	3.14 sec + 33 min 25 sec	1630x

Speed up comparison between GPU, Multigrid, GPU and Multigrid

The speed up given by multigrid method and GPU implementation is compared separately in Table 5.7.

Table 5.7: Speed up comparison of using Multigrid method and GPU implementation

Grid Size	Multigrid only	GPU only	Multigrid and GPU
416483	20x	30x	344x
701161	43x	33x	919x
1699751	68x	35x	1630x

Unsteady state 3D unstructured multigrid

The unsteady state problem is solved on a unstructured 3D grid and the speed up results are shown below. Table 5.8 shows the speed up achieved by both GPU implementation and multigrid algorithm compared to non-multigrid CPU algorithm. The speed up shown in Table 5.8 is the speed up in computation time. Once the solution reaches steady state the multigrid method has less effect on the convergence because the solution convergence in a few iterations in each time step.

Table 5.8: Unstructured grid time comparison (computation time + preprocessing time)

Grid Size	CPU	CPU-MG	GPU	GPU-MG	Speed up
416483	2 hr 27 min 40 sec + 46.8 sec	2 hr 26 min 10 sec + 58.1 sec	6 min 9 sec + 65 sec	6 min 6 sec + 83 sec	24x
701161	5 hr 12 min 29 sec + 1 min 57 sec	4 hr 41 min 28 sec + 2 min 22 sec	9 min 46 sec + 2 min 39 sec	9 min 19 sec + 3 min 27 sec	33x
1699751	14 hr 28 min 12 sec + 10 min 14 sec	13 hr 2 min 45 sec + 11 min 50 sec	26 min 41 sec + 32 min 45 sec	24 min 40 sec + 33 min 25 sec	35x

Chapter 6

Discussion

In this work a GPU based unstructured geometric multigrid solver is implemented in two and three dimensions. Different schemes for creating coarse level grids for unstructured grids are investigated and a improved coarsening implementation is proposed. The GPU implementation in this work uses a better graph representation which perform well in GPU architecture. Steady and unsteady heat transfer problem from CFD is solved and evaluated the speed up given by multigrid GPU implementation.

The current multgrid implementation works only for convex domains. In future this can be extended to concave domains. In this work the speed up given is focused only on computation time. The preprocessing time is given less attention because it is a one time process. In future the preprocessing time can be improved because for very large input it may take long time and one can think about a better coarse vertex set which improve the multigrid performance. Another future expansion is implementing the solver for multiple GPUs.

References

[1] A. Corrigan, F. F. Camelli, R. Löhner, and J. Wallin. Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids* 66, (2011) 221–229.

[2] J. Waltz. Performance of a three-dimensional unstructured mesh compressible flow solver on NVIDIA Fermi-class graphics processing unit hardware. *International Journal for Numerical Methods in Fluids* .

[3] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In ACM Transactions on Graphics (TOG), volume 22. ACM, 2003 917–924.

[4] C. Feng, S. Shu, J. Xu, and C.-S. Zhang. Numerical Study of Geometric Multigrid Methods on CPU–GPU Heterogeneous Computers. *arXiv preprint arXiv:1208.4247* .

[5] H. Guillard. Node-nested multi-grid method with Delaunay coarsening. Research Report RR-1898, INRIA 1993.

[6] T. F. Chan and B. F. Smith. Domain decomposition and multigrid algorithms for elliptic problems on unstructured meshes. *Electronic Transactions on Numerical Analysis* 2, (1994) 171–182.

[7] M. Adams and J. Demmel. Parallel multigrid solver for 3d unstructured finite element problems. In Supercomputing, ACM/IEEE 1999 Conference. IEEE, 1999 27–27.

[8] G. L. Miller, D. Talmor, and S.-H. Teng. Optimal Coarsening of Unstructured Meshes. *Journal of Algorithms* 31, (1999) 29 – 65.

[9] P. R. Brune, M. G. Knepley, and L. R. Scott. Unstructured Geometric Multigrid in Two and Three Dimensions on Complex and Graded Meshes. *CoRR* abs/1104.0261.

[10] M. Geveler, D. Ribbrock, D. Göddeke, P. Zajac, and S. Turek. Efficient finite element geometric multigrid solvers for unstructured grids on GPUs. Techn. Univ., Fak. für Mathematik, 2011.

[11] General-Purpose Computation on Graphics Hardware. `http://www.gpgpu.org`.

[12] CUDA Programming Guide. Nvidia Corporation, April 2012.

[13] NVIDIA's Next Generation CUDA Compute Architecture: FERMI, White paper. Nvidia Corporation, 2009.

[14] A. R. Brodtkorb, T. R. Hagen, and M. L. SæTra. GPU programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing* .

[15] CUDA C Best Practices Guide. Nvidia Corporation, May 2011.

[16] W. Briggs, V. Henson, and S. McCormick. A Multigrid Tutorial. Miscellaneous Bks. Society for Industrial and Applied Mathematics, 2000.

[17] U. Trottenberg, C. Oosterlee, and A. Schüller. Multigrid. Elsevier Academic Press, 2001.

[18] P. Wesseling. An introduction to multigrid methods. Pure and applied mathematics. John Wiley & Sons Australia, Limited, 1992.

[19] S. Hang. TetGen website. `http://tetgen.berlios.de`.

[20] T. Haynes, S. Hedetniemi, and P. Slater. Fundamentals of Domination in Graphs. Chapman and Hall/CRC Pure and Applied Mathematics Series. Marcel Dekker, 1998.

[21] A. Poghosyan. The Probabilistic Method for Upper Bounds in Domination Theory. Ph.D. thesis, University of the West of England, Bristol 2010.

[22] J. Dunbar, D. Hoffman, R. Laskar, and L. Markus. α-Domination. *Discrete Mathematics* 211, (2000) 11 – 26.

[23] I. Yavneh. On red-black SOR smoothing in multigrid. *SIAM J. Sci. Comput.* 17, (1996) 180–192.

[24] S. P. Vanka, A. F. Shinn, and K. C. Sahu. Computational fluid dynamics using graphics processing units: challenges and opportunities. In Proceedings of the ASME IMECE. 2011 .

[25] S. R. Reddy, J. Sebastian, S. M. Miyyadad, R. Banerjee, and N. Sivadasan. VOF Based Two-Phase Flow Solver On GPU Architecture. In ICCMS. December 2012 .

[26] S. R. Reddy, J. Sebastian, S. M. Miyyadad, R. Banerjee, and N. Sivadasan. Single and Two Phase Flow CFD Solvers Using GPU. In PARCOMPTECH. February 2013 .

[27] P. Harish, V. Vineet, and P. Narayanan. Large graph algorithms for massively multithreaded architectures. *Centre for Visual Information Technology, I. Institute of Information Technology, Hyderabad, India, Tech. Rep. IIIT/TR/2009/74* .

[28] V. Vineet, P. Harish, S. Patidar, and P. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In Proceedings of the Conference on High Performance Graphics 2009. ACM, 2009 167–171.

[29] W. Wang, Y. Huang, and S. Guo. Design and Implementation of GPU-Based Prim's Algorithm. *International Journal of Modern Education and Computer Science (IJMECS)* 3, (2011) 55.

[30] A. Rungsawang and B. Manaskasemsak. Fast PageRank Computation on a GPU Cluster. In Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on. IEEE, 2012 450–456.

[31] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In ACM SIGPLAN Notices, volume 47. ACM, 2012 117–128.

[32] S. V. Patankar. Numerical heat transfer and fluid flow. Taylor & Francis, 1980.

[33] J. C. Tannehill, D. A. Anderson, and R. H. Pletcher. Computational fluid mechanics and heat transfer. Taylor & Francis, 1997.