Parallel AMG Solver for Three Dimensional Unstructured Grids Using GPUs

RaviTej Kamakolanu

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

June 2014

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)

K RAVITEJ

(RaviTej Kamakolanu)

CS12M1004

(Roll No.)

Approval Sheet

This Thesis entitled Parallel AMG Solver for Three Dimensional Unstructured Grids Using GPUs by RaviTej Kamakolanu is approved for the degree of Master of Technology from IIT Hyderabad.

[illegible]

(U. RAMAKRISHNA) Examiner
DEPT. OF CSE
IITH

[illegible]

(Dr. Kirti Sahu) Examiner
Dept. Of. Chemical Engineering
IITH

[illegible]

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

[illegible]

(T. Bheemarjuna Reddy) Chairman
Dept. of CSE
IITH

Acknowledgements

At the outset, I would like to express my sincere gratitude to my thesis adviser Dr. Naveen Sivadasan for his constant encouragement, patience and immense knowledge. Without his guidance and persistent help, this dissertation would not have been possible. I would like to thank Dr. Raja Banerjee and Vatsalya Sharma for their support and help in understanding the domain related concepts. Further, I would also like to thank Dr. Ramakrishna Upadrasta for introducing me to many interesting topics in compiler optimization. I am also thankful to Jin Sebastian for the numerous fruitful conversations on GPU computing. Finally, I thank almighty, my family and friends for their support and constant encouragement.

Abstract

Consider a set of points P in three dimensional euclidean space. Each point in P represents a variable and its value is dependent on the value of its neighborhood scaled by predefined constants. The problem is to solve all the variables which reduces to solving a large set of sparse linear equations. This kind of representation arises naturally while solving flow equations in Computational Fluid Dynamics (CFD). Graphics Processing Units (GPUs), over the years have evolved from being graphics accelerator to scalable co-processor. We implement an algebraic multigrid solver for three dimensional unstructured grids using GPUs. Such a solver has extensive applications in Computational Fluid Dynamics. Using a combination of vertex coloring, optimized memory representations, multi-grid and improved coarsening techniques, we obtain considerable speedup in our parallel implementation. For our implementation, we used Nvidia's CUDA programming model. Our solver is used to accelerate solutions to various problems like heat transfer, Navier-Stokes etc. Our solver achieves 2157 and 29 times speed up for steady state and unsteady state head transfer problem respectively on a grid of size 2.3 million, compared to serial non-multigrid implementation. Our solver provides significant acceleration for solving pressure Poisson equations, which is the most time consuming part while solving Navier-Stokes equations. In our experimental study, we solve pressure Poisson equations for flow over lid driven cavity, laminar flow past square cylinder and plain jet problems. Our implementation achieves 915 times speed up for the lid driven cavity problem on a grid of size 2.6 million and a speed up of 1020 times for the laminar flow past square cylinder problem on a grid of size 1.7 million, compared to serial non-multigrid implementations. For plain jet problem, our solver achieves a speed up of 47 times, compared to serial non-multigrid implementation on a grid of size 2.7 million. We also implement multi GPU AMG solver which achieves a speed up of 1.5 times, compared to single GPU solver for heat transfer problem.

Contents

Chapter 1

Introduction

Over the years, Graphics Processing Units (GPUs) have transformed from being hardwired graphics accelerator to programmable devices for general purpose computing known as General Purpose Computation on GPUs (GPGPU). Increase in computing capability coupled with decrease in cost resulted in GPUs becoming cost-effective scalable co-processors and an integral part of high performance computing (HPC). Modern day GPUs which come with thousands of cores are being used to accelerate compute intensive tasks of applications in various fields such as Computational Fluid Dynamics [1], Computer Vision, Linear Algebra [2] and Digital signal Processing [3]. The ease of GPU programming increased especially with interfaces like CUDA from NVIDIA and OpenCL from Khronos group.

Computational Fluid Dynamics (CFD) deals with solving and analyzing fluid flows using various numerical methods and algorithms. CFD simulation of complex problems is highly computationally intensive and is usually done on super computers. Navier-Stokes equations are central to flow equations that are used to solve the velocity-pressure field. Solving pressure Poisson equations consumes most of the computing time in Navier-Stokes simulations [4]. Multi-fold speed up in many CFD applications can be gained by accelerating solvers for these equations.

1.1 Three Dimensional Unstructured Grid Problem

Consider a set of points P in a three dimensional Euclidean space. For each point $i \in P$, its neighborhood $N(i) \subseteq P$ is defined. The neighborhood definition depends on type of discretization used. Each point $i \in P$ represents a variable, whose value is dependent on the values of its neighborhood $N(i)$. The value at i is given by the sum of values at points in $N(i)$ scaled by some pre-determined constants as shown in Fig 1.1. The value $v(i)$ at i is given by,

$$v(i) = \sum_{j \in N(i)} a_{ij} v(i) \qquad \text{and} \qquad \sum_{j \in N(i)} a_{ij} < 1$$

where a_{ij} is the constant by which $v(i)$ is scaled.

The goal is to solve $v(i)$ for all $i \in P$. These kind of representations are frequently encountered in Computational Fluid Dynamics while solving flow governing equations. Three dimensional unstructured grid defined by point set P forms the domain. The problem reduces to solving large set

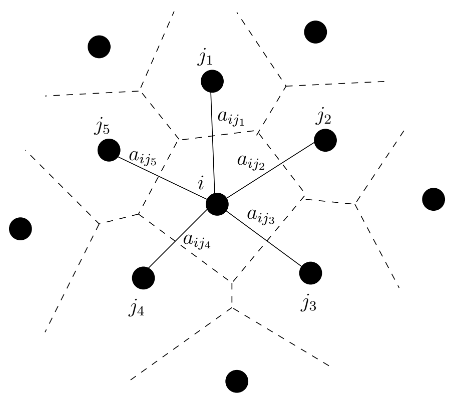


Figure 1.1: Point i with neighborhood connectivity

of sparse linear equations of the form

$$Au = f$$

where A is $n \times n$ matrix with real entries a_{ij}, u and f are vectors in $\mathbb{R}^n$. The numerical methods that exist to solve set of linear equations can be categorized into direct solvers like Gauss elimination, LU factorization etc. and iterative solvers like Gauss Seidel, Jacobi, Conjugate Gradient etc. Though direct solvers give exact solution, they are inefficient for solving large set of sparse linear equations. Hence, iterative methods which give reasonably accurate approximate solution are preferred over direct solvers. Gauss Seidel is an efficient and most commonly used iterative solver. It starts with an initial guess and produces series of improving results till convergence. Multigrid (MG) method is an efficient and scalable approach that accelerates the convergence of the iterative solvers. In multigrid method, the problem is solved on coarser representation and the solution is interpolated back to the finer representation to get a better approximation faster. This is recursively applied which creates an hierarchy of coarser grids. Algebraic multigrid is a multigrid technique which derives the hierarchy of grids from the information available in the set of linear equations. The typical size of the problems considered in the work is in the order of millions which makes it a computationally intensive and time consuming task. Hence, GPUs can be used to accelerate the solution.

1.2 Related Work

Since the introduction of algebraic multigrid in 1980's [5, 6], many improvements to classical AMG have been proposed [7,8]. There are research works [9,10] that use GPU to solve the unstructured grid problems. They are mainly aimed at parallelization of unstructured solvers but not on combining them effectively with multigrid methods. Considerable work is done on parallelizing AMG [11–16] which includes implementing various parallel coarsening and smoothing techniques on parallel computers. They focused mainly on efficient solvers for structured grids and using coalesced memory access with reported speed up ranging from ten to thousand times. The usability of these solvers is limited to problems with simple geometry. A conjugate gradient solver on GPU was given by Gundolf Haase et al. [17]. They use conjugate gradient algorithm with algebraic multigrid preconditioner. Our focus is algebraic multigrid solver with Gauss-Seidel iterative smoother.

In AMG, the grid is represented as a graph and it is critical to have a GPU aware graph representation for improved performance. Different graph representations for GPU processing have been proposed in [18–22] to implement number of graph algorithms on GPU.

The *polyhedral model* [23] is a formal framework by which parallelism in input `for`-loop programs of a specific variety, called Affine Control Programs (ACLs) could *automatically* be found using Rational and Integer Linear Programming techniques. Examples of ACLs are dense matrix programs like matrix-matrix multiplication, matrix-vector multiplication, stencil computations etc. Polyhedral frameworks also have developed advanced code generation tools and techniques suitable for modern heterogeneous architectures with multi-cores or GPUs. PluTo [24] is a well known source-to-source polyhedral compiler with both input and output languages being C. It however cannot be used with our code which is in C++ which has dependencies spanning across functions, sparse matrices and pointer accesses. Another new popular polyhedral compiler is the Polly framework [25] of the LLVM compiler infrastructure. Though Polly applies transformation on an intermediate representation, it also suffers from similar limitations making it unusable for our work.

1.3 Overview of the Work

As part of this work, we implement a parallel algebraic multigrid solver for three dimensional unstructured grids using GPUs. Our main contributions are (a) efficient parallel implementation of AMG solver for 3D unstructured grids on GPU (b) improved AMG coarsening techniques for accelerated GPU performance and (c) Multi GPU implementation. To evaluate the performance of the solver, we solve (i) Steady and unsteady state heat transfer problem (ii) Navier-Stokes problem by accelerating pressure Poisson using our solver. We also validate the results obtained using the solver by comparing them against standard experiment or commercial software generated results.

1.4 Thesis Outline

The thesis is structured as follows. In chapter 2 we give a brief overview of GPU architecture, CUDA and performance optimizations. We discuss the algebraic multigrid solver in chapter 3, its parallelization and our proposed improvements to AMG coarsening in chapter 4. Chapter 5 describes the GPU implementation details. We discuss experimental results in chapter 6. Chapter 7 describes the multi GPU implementation details and results. We discuss future work in chapter 8.

Chapter 2

GPU Architecture and CUDA Programming Model

Graphics Processing Units (GPUs) which were primarily designed for accelerating video or graphics rendering, had all its functionalities hardwired. Over the last few years, GPUs became programmable and are being used to solve general purpose programs, also called as General Purpose Computation on GPUs (GPGPU). GPUs can be used both as a programmable graphics processor and a scalable parallel computing platform. The ease of GPU programming increased especially with interfaces like CUDA from NVIDIA and OpenCL from Khronos group. Power efficient and less expensive modern days GPUs which come with thousands of cores offer massive computational capacity and have become an integral part of High Performance Computing (HPC). GPUs are being used to accelerate parts of applications spanning across different fields that have an ever-increasing demand for computing power. This chapter presents a brief overview on GPU architecture, CUDA programming model and performance optimizations. The reader is referred to [26–29] for details.

2.1 GPU Architecture

GPUs are specially designed hardware devices to cater the needs of highly parallel and compute intensive applications. CPU and GPU are designed using two completely different philosophies. Figure 2.1 compares and contrasts the CPU, GPU architectures.

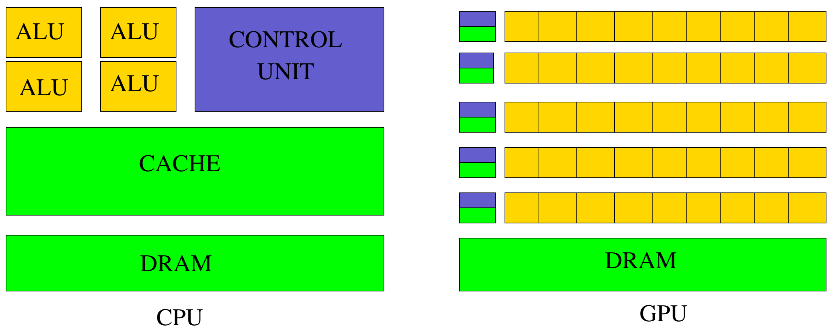


Figure 2.1: Comparison between CPU and GPU architectures

CPU aims at minimizing the latency where as GPU tries to hide the latency. CPU has large cache memory and does sophisticated things like out of order instruction execution, branch prediction etc. It is for this reason, more transistors are dedicated to control unit than arithmetic logic units (ALUs). CPU is well suited for sequential/serial code execution. On the flip side, GPU has relatively smaller cache and more transistors are dedicated to ALUs than the control unit. GPU doesn't support speculative execution and branch predictions. GPU can execute large number of threads in parallel and is well suited for compute intensive tasks. To execute large number of threads in parallel, GPU uses an architecture called SIMT (Single Instruction Multiple Threads). It is closely related to SIMD (Single Instruction Multiple data) where different processing elements execute same instruction but on different data items. In SIMD all the threads follow same execution path where as SIMT facilitates threads to take different execution paths. A typical GPU contains ALU, Control Unit, cache memory and DRAM. GPU cores are organized as an array of Streaming Multiprocessors (SMs). Each SM contains number of Streaming Processors (SPs or simply GPU cores), instruction cache and control unit. In GPU computing model, the terms *host* and *device* are used to refer to CPU and GPU respectively. Each SM creates, manages and executes threads in group (typically of size 32) called warps. Warp Scheduler, which is also a part of SM schedules these warps for execution.

2.2 CUDA Programming Model

Compute Unified Device Architecture (CUDA) is an interface that enables programmer to utilize the massive parallel computing capability provided by the GPU for general purpose computing. CUDA also provides developers a set of libraries and extensions to standard programming languages like C, C++ etc. We briefly discuss CUDA programming, compiler, execution models and performance optimizations in the following sections:

2.2.1 Compiler Model

A CUDA source file will be a mixture of host code (which runs on the CPU) and device code (which runs on the GPU). The CUDA compiler segregates the code into host and device code. Nvidia's *nvcc* compiler translates the device code into pseudo-assembly code known as Parallel Thread Execution (PTX) code. PTX code can either be converted to binary form called *cubin* object or can be loaded by the application at the runtime and get compiled using *just-in-time compilation.* Just-in-time compilation enables the application to benefit from latest compiler improvements but increases the application load time. CUDA compiler replaces the constructs in host code used for device code invocations by CUDA run time functions. The host code is compiled using a CPU compiler (C or C++) and the Cubin, CPU object files are linked to get an CPU-GPU executable file as shown in Fig 2.2

2.2.2 Execution Model

Using CUDA, the compute intensive and data parallel parts of an application are parallelized by launching large number of concurrent threads on GPU. Each thread executes same instruction but on different data. For this purpose, users define kernel which contains the code to be executed by each thread. Kernel configuration specifies the number, organization of the threads and can

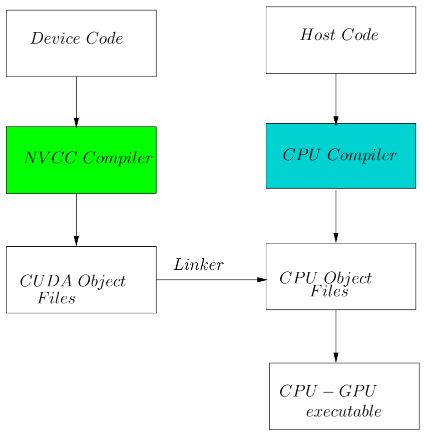


Figure 2.2: CUDA Compilation Model

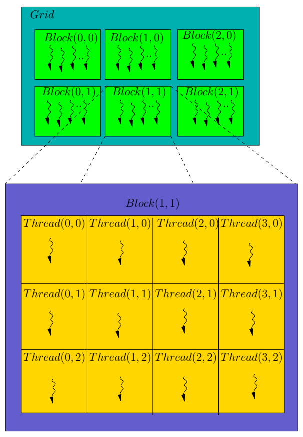


Figure 2.3: Thread Organization in CUDA

be determined either at compile time or run time. In CUDA, threads are organized in two level hierarchy namely blocks and grids as shown in Fig 2.3.

At the first level, the threads are grouped into thread blocks. The block size is a multiple of warp size and is decided by the programmer. Block size of 128 or 256 is most frequently used and often provide optimal performance. Each thread block can run independent of other and hence can be scheduled across any SM. The thread blocks are further grouped into grids. The size of grid is determined by the size of data that the application is dealing with. CUDA allows user to organize blocks and grids in one, two or three dimensions thereby allowing easy mapping of threads to multi-dimensional data structures. Each block within the grid is uniquely identified using the built-in *blockIdx* variable, from the kernel. Similarly, each thread in a thread block is uniquely identified using the built-in *threadIdx* variable. Both blockID and threadID are built in structures that contains three components to store index in each dimension. The thread and block size are stored in built-in variables *blockDim* & *gridDim.* CUDA maps each software thread block to a hardware SM. Multiple blocks can be mapped to same SM and are executed in time sharing fashion.

Threads within a block can communicate with each other using shared memory and can synchronize using *_syncthread()* method. However, threads across the blocks can't synchronize with each other and can communicate only using global memory. Any data that kernel operates on should reside in device global memory. CUDA API provides three functions for this purpose: (a) cudaMalloc allocates memory on the device (b) cudaMemcpy transfers the data from host to device and vice-versa and (c) cudaFree is used to free the memory on the device.

2.2.3 Memory Hierarchy

The GPU memory is organized as three level hierarchy as shown in Fig 2.4

(a) **Device Memory :** It is the the largest memory in the hierarchy and also the one with highest latency. Device Memory is to GPU what DRAM is to a CPU. Device memory is logically further divided into global memory, local memory, constant and texture memory. All threads can access global memory and is the only part of device memory which CPU can read as well as write. The local memory which is private to each thread also resides on device memory. Constant memory is used to store read only data such as constant tables etc. and is cached into constant cache.

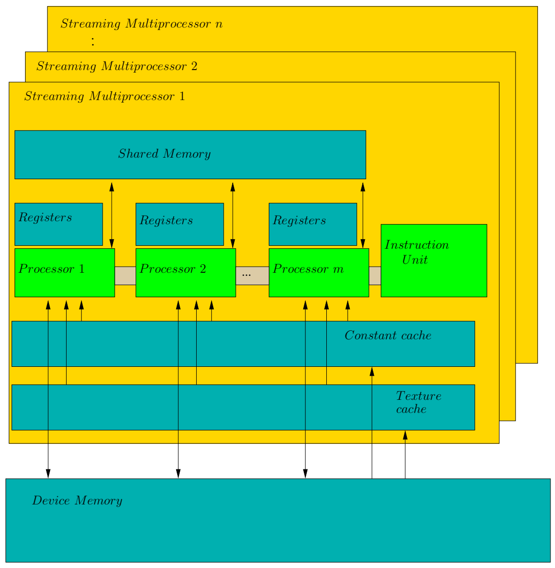


Figure 2.4: CUDA Memory Hierarchy

Texture memory which is cached into texture cache is optimized for 2D spatial locality and is preferred over global memory when there is no access pattern to do memory coalescing. GPU can only read from constant and texture memory where as CPU can only write to them.

(b) **Shared Memory :** It is per SM memory that resides on-chip and is shared by all the threads within a thread block. Access to shared memory is very fast when compared to that of global memory. Any data that is shared or reused by the threads within a block can be transferred to shared memory for improved performance. Shared memory can also be used to share data among threads of same thread block.

(c) **Registers :** Each thread has its own set of registers. Accessing data in the registers is extremely fast and the CUDA compiler automatically tries to place the frequently accessed variables by the thread into registers.

2.3 Performance Optimizations

In addition to effective parallelization of the code, it is crucial to optimize the implementation with respect to the underlying GPU architecture to extract maximum performance [28]. The optimizations include maximizing SM utilization, memory and instruction throughput [29]. Increasing occupancy, coalesced memory access and avoiding warp divergence greatly increase the performance of the applications. Occupancy is defined as the ratio of number of active warps to maximum number of warps supported by SM. Access to global memory data requires hundreds of clock cycles and the warp scheduler switches between warps to hide this latency. Increasing thread pool size i.e., occupancy of SM helps in hiding the latency and also maximizes SM utilization. The hardware also checks if all the threads in a warp are accessing collocated global memory locations. In such scenario, all the accesses can be consolidated and is known as *Coalesced Memory Access*. Scattered memory access by threads in a warp will results in unnecessary data transfer from global memory to cache. Hence, storing the data accessed by thread warp in collocated global memory locations results in increased memory throughput. Conditional statements in the kernel may cause threads of same warp to follow different execution paths, called as *Warp Divergence*. Warp divergence causes delay in execution of entire warp and can be avoided by re-ordering the data so that all the threads in warp take same branch.

Chapter 3

Algebraic Multigrid

The numerical methods that exist to solve set of linear equations can be categorized into direct solvers like Gauss Elimination, LU factorization etc. and iterative solvers like Gauss Seidel, Jacobi, Conjugate Gradient etc. Direct methods compute the solution in finite number of steps and usually provide an exact solution (assuming no rounding errors exist). On the flip side, iterative methods provide only an approximate solution. Iterative methods start with an initial guess and produce a sequence of improving solutions. The method terminates when the solution reaches desired accuracy. Iterative methods may or may not terminate in finite number of steps and are called convergent if they terminate for given initial guess. Understanding the problem background helps us in choosing good initial guess which reduces the number of iterations required for convergence. To solve large system of linear equations, iterative methods are preferred over direct methods which are too expensive. In iterative methods, error is defined as the difference between exact solution and the current estimate. Iterative methods like Jacobi or Gauss Seidel are very effective at smoothing high frequency error component (rough error) in the system of equations and typically take only few iterations to do so. However, they are not so effective in smoothing low frequency error (smooth error) and require more number of iterations.

3.1 Multigrid Methods

Multigrid (MG) method offers an efficient way of solving large system of linear equations especially those from finite volume, finite difference and finite element discretization of governing partial differential equations(PDEs). Multigird methods are known to scale linearly with respect to number of unknowns i.e., for a given level of convergence multigrid methods provide a solution in $O(n)$ time where n is the number of unknowns [30]. Instead of working on a single mesh, multigrid method works on hierarchy of meshes, which are carefully constructed in such a way that the low frequency error in finer mesh turns out to be high frequency level in the coarse mesh, which can again be effectively smoothed using an iterative method. Multigrid method is a recursive error correcting method and has following steps:

Smoothing: Reduce high frequency error component using iterative methods like Jacobi or Gauss Seidel.

Restriction: Transfer the residual from finer mesh to coarser mesh.

Prolongation: Transfer the error correction calculated on coarser mesh to finer mesh.

Defining MG components include constructing hierarchy of grids and defining inter-grid transfer operations i.e., restriction and prolongation. Two different multigrid approaches exist namely, Geometric multigrid (GMG) and Algebraic multigrid (AMG) [31]. Geometric multigrid, uses the geometry of the problem (grid) to define various multigrid components. On the other hand, algebraic multigrid uses only the information present in the set of linear equations obtained by discretizing the governing PDEs to define various multigrid components.

Though GMG is more natural or intuitive, its applicability is restricted due to requirement of explicit knowledge about problem geometry. Also, the coarsening becomes very complicated/impossible for complex and concave grids. AMG is preferred over GMG due to following advantages:

- It is purely a matrix based approach and doesn't use any geometric information
- No special handling is required for concave grids during coarsening
- AMG can be used as a black-box to solve problems, provided the underlying matrices have certain properties [32].

3.2 Algebraic Multigrid

In AMG, it is often very helpful to visualize the $n \times n$ matrix A as a graph G on the vertex set $\{1, \dots, n\}$. Each variable corresponds to a vertex in G and each non-zero matrix entry a_{ij} in the matrix A (which is assumed to be symmetric positive definite) corresponds to a directed edge between vertices i and j. In the rest of the paper, the terms grid, mesh, graph and mesh graph are used interchangeably. So, are the terms nodes, points and vertices. If there is a directed edge from vertex u to vertex v then we say that u *depends* on v and that v *influences* u. AMG works on the heuristic that the smooth error varies slowly in the direction of relatively large negative coefficients of the matrix A [33].

Definition 1 (Strength of Connection, [33]) *Given a threshold $0 < \theta \leq 1$, the variable i strongly depends on variable j if*

$$-a_{ij} \geq \theta \max_{k \neq i} \{-a_{ik}\}$$

Strength of connection is always measured relative to the largest off-diagonal entry. Off diagonal entries which do not satisfy above condition are considered *weak* connections. The matrix obtained by deleting *weak* connections in A is called *Strength Matrix A_s*. We note that strength of connection need not be symmetric i.e., a variable i can strongly depend on j but not vice-versa.

Each level of AMG uses a prolongation matrix P, and the corresponding restriction matrix P^T which is the transpose of P. These matrices are defined based on corresponding strength matrix and is discussed in Section 3.2.2. The coarser system will have lesser number of variables, say $n_c < n$ where n is number of variable in the finer system. Hence P is an $n \times n_c$ matrix. Let $Au = f$ be the equations governing the finer system. Main steps in a two level AMG (which can be extended to multi-level) can be summarized as:

Compute estimate u^* for u in $Au = f$;

Compute the residual $r = f - Au^* = Ae$;

Solve for e_c in the coarser system $A_c \cdot e_c = P^T \cdot r$, where $A_c = P^T AP$;

Correct $u^* \leftarrow u^* + P \cdot e_c$.

Couple of smoothing steps are executed while computing the initial estimate for u^* and after obtaining the correction from the coarser system.

3.2.1 Multigrid Generation

In classical AMG, hierarchy of grids are created from the initial grid by applying a coarsening algorithm recursively. Coarsening algorithm partitions the points into two disjoint sets. One is set of C-points i.e., points that are part of coarse grid as well and the other is F-points i.e., points that are not part of the coarse grid. To compute C, the coarsening algorithm [31] considers the strength matrix A_s and the corresponding mesh graph G_s. Each vertex u is assigned a weight which is the total number of vertices that depend on u. The algorithm proceeds iteratively and at each step, a vertex u with highest weight is chosen as a C point and all vertices depending on u are marked as F points. The weights are updated for vertices that are connected by outgoing edges from the new set of C and F vertices. Weights of all points that influence the new C point is decremented by one. For each new F point u, weights of all points that influence u is incremented by one. Figure 3.1 illustrates the coarsening process.

3.2.2 Computing Matrices P and A_c

Given the C/F splitting of points, the goal is to define P and thereby compute A_c. Let n_c denote the size of C and let n denote the size of $C \cup F$. We follow the approach in [31] to define the $n \times n_c$ matrix P. Let $u_1, u_2, \ldots, u_{|C|}$ be an ordering of the vertex set C. Let C_i denote subset of C that strongly influence vertex i. For each $i \in C \cup F$ and each $j \in \{1, \ldots, |C|\}$, the entry w_{ij} of P is define as :

$$w_{ij} = \begin{cases} 1 & \text{if } \ i \in C \ \text{ and } \ i = u_j; \\ a_{ij} / \sum\limits_{k \in C_i} a_{ik} & \text{if } \ i \notin C \ \text{ and } \ u_j \in C_i; \\ 0 & \text{otherwise.} \end{cases}$$

The coarser system A_c is obtained using the Galerkin operator

$$A_c = P^T AP.$$

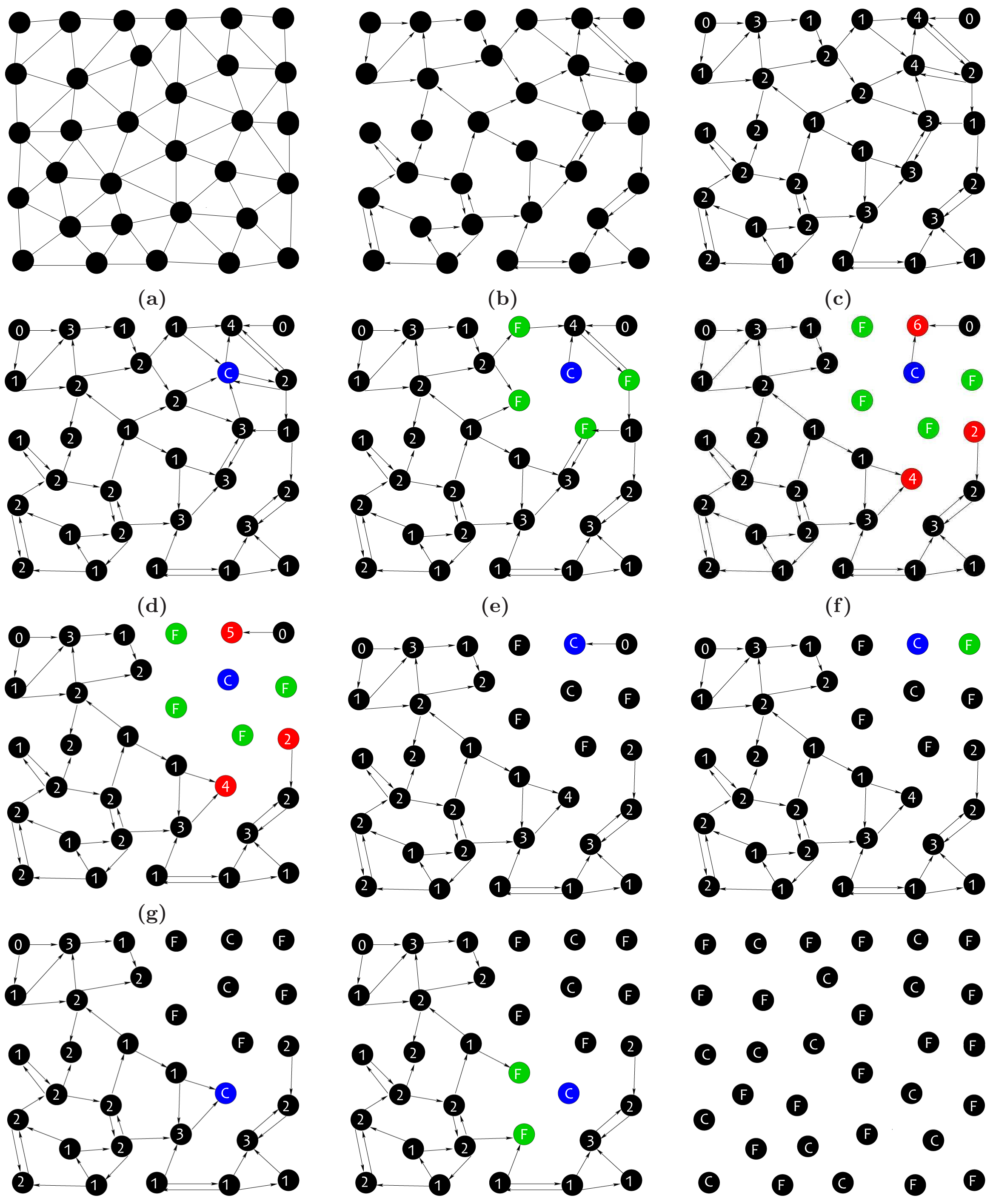


Figure 3.1: Illustration of the coarsening algorithm. (a) The graph corresponding to the A matrix. (b) The graph after deleting weak connections. (c) Nodes of the graph are assigned a weight equal to the number of nodes that depend on it. (d) A point with maximal weight is chosen as a C-point. (e) The neighbors of the new C-point are marked as F-points. (f) For each new F-point, increment the weights of nodes that influence it to make them more likely to be C-points. (g) For new C-point, decrement the weights of nodes that influence it. The algorithm continues in this way until all points are either C or F points.

Chapter 4

Algorithmic Improvements and Parallelization of AMG

In the following, we discuss the specific algorithmic improvements that we incorporate for faster GPU implementation.

4.1 Improved Coarsening

The accuracy of the solver also depends on the quality of the coarsening. Each level in the multigrid should retain adequate number of boundary nodes and the coarsening algorithm as such will not ensure this. To overcome this, we modify the coarsening (Algorithm 1) and at each stage of coarsening, the boundary nodes are assigned a weight which is α times the number of points that depend on it, for a predefined $\alpha > 1$. As shown in the experiments, by this coarsening, more number of boundary nodes become part of highly coarser grids and thereby improving the coarsening quality.

Algorithm 1 Improved Coarsening

```
Require: Graph representation of matrix $A$
1: Delete weak connections in the graph
2: For each non-boundary point $u$, assign a weight equal to the number of points that depend on $u$.
3: For each boundary point $v$, assign a weight equal to $\alpha$ times the number of points that depend on $v$.
4: Choose a point $p$ with maximum weight as $C$ point.
5: Mark points depending on $p$ as $F$ points.
6: For each new $F$ point $u$, increment weights of all points that influence $u$ by one.
7: Decrement the weights of all points that influence $p$ by one.
8: Repeat steps (4) to (7) till all the points are marked as $C$ or $F$.
```

4.2 Modifying A_c for Faster Smoothing in Coarser Grids

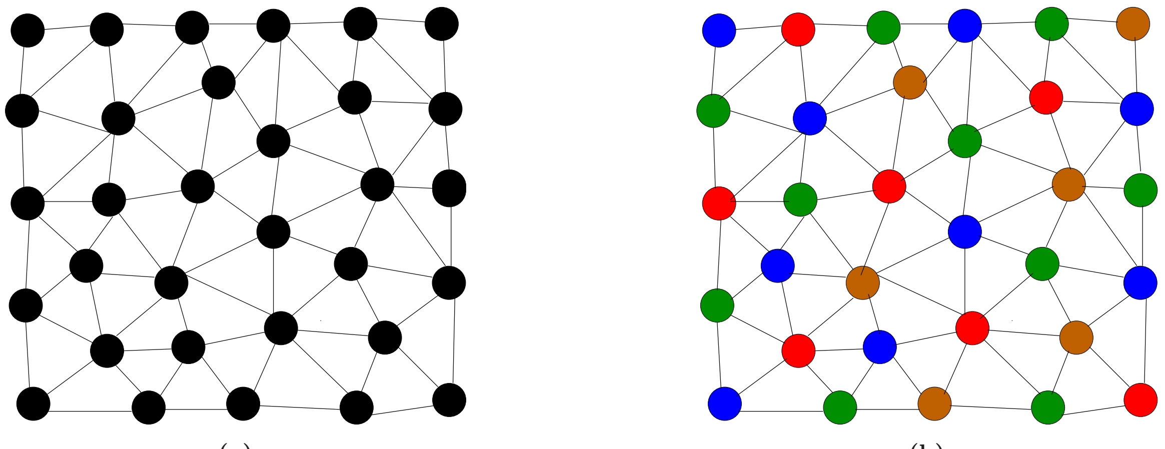
We incorporate the following transformation to matrix A_c in our coarsening procedure for improving the performance of GPU implementation. As we go down the AMG hierarchy, the number of neighbors for each node in the coarser graphs increases rapidly making the coarse systems more denser. Large degrees result in fetching more data from global memory during smoothing operation in GPU and thereby degrading the GPU performance on coarser systems. To overcome this, our coarsening procedure modifies matrix A_c in such a way that the neighbors with insignificant influence in the corresponding graph is ignored. Entries a_{ij} in the ith row of A_c are modified as follows. Let δ denote the average value of off-diagonal entries in row i (they are negative valued in A_c and positive valued in the graph). Let J' denote the subset of columns such that for each $j \in J'$, $a_{ij} \leq \beta \cdot \delta$, where β is a user defined constant. Let $|J'| = n'$ and let $\epsilon = \sum_{j \notin J'} a_{ij}/n'$. Modified a_{ij} is given by

$$a_{ij} \leftarrow a_{ij} + \epsilon \text{ if } j \in J', \text{ and } a_{ij} = 0 \text{ otherwise.}$$

By the above modification, we ignore all the neighbors whose influence is less than β times the average influence, and their total influence is distributed among the remaining neighbors of i. Though this might slightly slow down the convergence, it is compensated by the reduced smoothing time in coarser grids.

4.3 Parallelization of Gauss Seidel Iterative Method

To smooth high frequency error component at each level in the multigrid, iterative solvers like Jacobi or Gauss-Seidel can be employed. Both these methods assume an initial guess and visit nodes in an arbitrary order to update the value at the node. However, they differ in the values of neighboring nodes that are used during updating. Jacobi method is preferred if vector or parallel processor is available at disposal due to its ease of parallelization. However, Gauss-Seidel method has faster convergence than Jacobi methods and hence is used in this work. Gauss-Seidel is inherently sequential as we can't update all the inter-dependent nodes simultaneously. Graph vertex coloring in the corresponding mesh graph is used to obtain independent sets corresponding to the color classes (Fig 4.1). All points in one color class can be updated in parallel [34,35]. We discuss the details in the next chapter.



(a) **(b)**

Figure 4.1: Multi Colored Gauss Seidel Smoother (a) Graph corresponding to the matrix A (b)The graph is colored to get independent sets of nodes

Chapter 5

GPU Implementation

We use CUDA programming model for our implementation. Implementing graphs algorithms on GPU is challenging due to irregular data access pattern associated with graphs. Using appropriate data structures and data organization/arrangement that maximizes coalesced memory access is the key for effective GPU implementation. A total of seven GPU kernels are used in our implementation: One kernel to perform smoothing, two kernels each for restriction and prolongation operations. A kernel for array reduction is used to get root mean square error for convergence testing. The different algorithmic techniques and data structures used for GPU implementation of the solver are discussed in the following.

5.1 Vertex Coloring

To parallelize Gauss-Seidel iterative smoothing, we use standard vertex coloring technique to get independent sets of nodes in the graph. As no two adjacent nodes have same color, each color class forms an independent set. The minimum number of colors required to color a graph G is called its chromatic number denoted by $\chi(G)$. As Gauss-Seidel method allows us to update nodes in any arbitrary order, we update them in the order of color class i.e., update nodes in one color class after the other. Within a color class, all the nodes can be updated in parallel as they form an independent set. Though an easy $\Delta + 1$ coloring is possible for any graph, where Δ is maximum degree, a $\chi(G)$ coloring is known to be NP-hard in general. We use the standard greedy coloring algorithm employed in [35], which gives a 6 coloring for planar graphs. Let the vertices of the graph be ordered as $u_1, u_2, \ldots, u_n$, in such a way that u_i is a minimum degree vertex in the graph induced by vertices $\{u_1, u_2, \ldots, u_i\}$. Now color each vertex with a free color in the order $u_1, u_2, \ldots, u_n$.

5.2 Graph Representation

The memory representation of graph used for GPU processing has significant impact on the performance. Graph data includes (a) Data corresponding to each vertex - degree, value at vertex etc. (b) Edge information - indices of neighboring vertices and their corresponding scale factors etc. Vertex data is re-ordered according to the color of vertices i.e., data of all the vertices having same color will be co-located. An array of pointers is maintained to store the starting index of each color class.

The implementation processes the vertices of each color in sequence and for each color, creates as many threads as the number of vertices in the color class. Re-ordering the data according to color results in coalesced memory access [35] as show in Fig 5.1 (adapted from [35]).

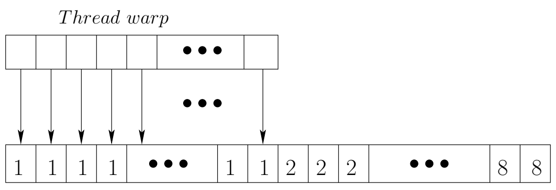


Figure 5.1: Re-ordered for coalesced memory access

To store edge data, we use the semi-compact column major matrix representation as in [35] which requires $O(\Delta \cdot |V|)$ space for a graph with maximum degree Δ, which is generally small for many practical problems. Each column stores the adjacency information of a single vertex. The edge data accessed by threads will be collocated and hence results in a coalesced access as shown in Fig 5.2 (adapted from [35]).

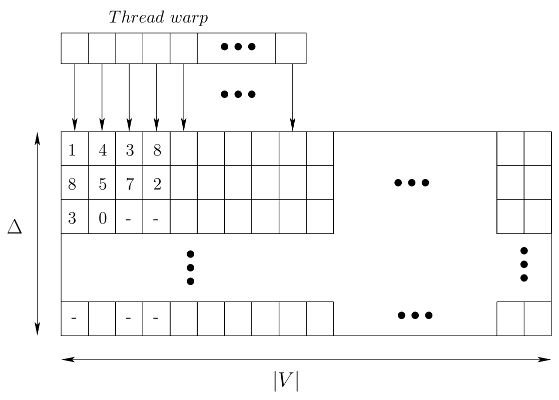


Figure 5.2: Coalesced memory access in column major adjacency

5.3 Multigrid Implementation

Hierarchy of grids created during pre-processing phase are stored in device memory. The inter-grid transfer operators which include prolongation and restriction matrices (stored in column major matrix representation) are also stored as part of grid. Following GPU kernels are used for implementing different steps in the multigrid method.

- Smoothing The kernel takes the starting and ending index of each color class, creates as many threads as the number of vertices in the color class and updates the value at each vertex.

- Restriction Two kernels are used for implementing restriction operation. One of the kernels creates as many threads as the number of vertices in the finer mesh and calculates residual at each vertex. The other kernel creates as many threads as the number of vertices in the coarser mesh and updates residual at each vertex using restriction matrix.

- Prolongation Two kernels are used for implementing prolongation operation as well. One of them creates as many threads as the number of vertices in the coarser mesh and calculates error correction at each vertex. The other kernel creates as many threads as the number of vertices in the finer mesh and updates the value at each vertex using prolongation matrix.

We use V-cycle multigrid, which is made up of a down cycle and up cycle. *Down cycle* is a sequence of smoothing and restriction operations performed alternately starting from finest grid till we reach coarsest grid. *Up cycle* is a combination of prolongation and smoothing operations performed alternately starting with the coarsest grid till we reach finest grid. The multigrid V-cycle is repeated till the desired convergence is reached.

5.4 Integration with CFD Solver

In order to solve Navier-Stokes equation, the solver has been integrated with in-house developed CFD software. The solver accelerates pressure Poisson equation solving and the block diagram of the CFD solver is shown in Fig 5.3.

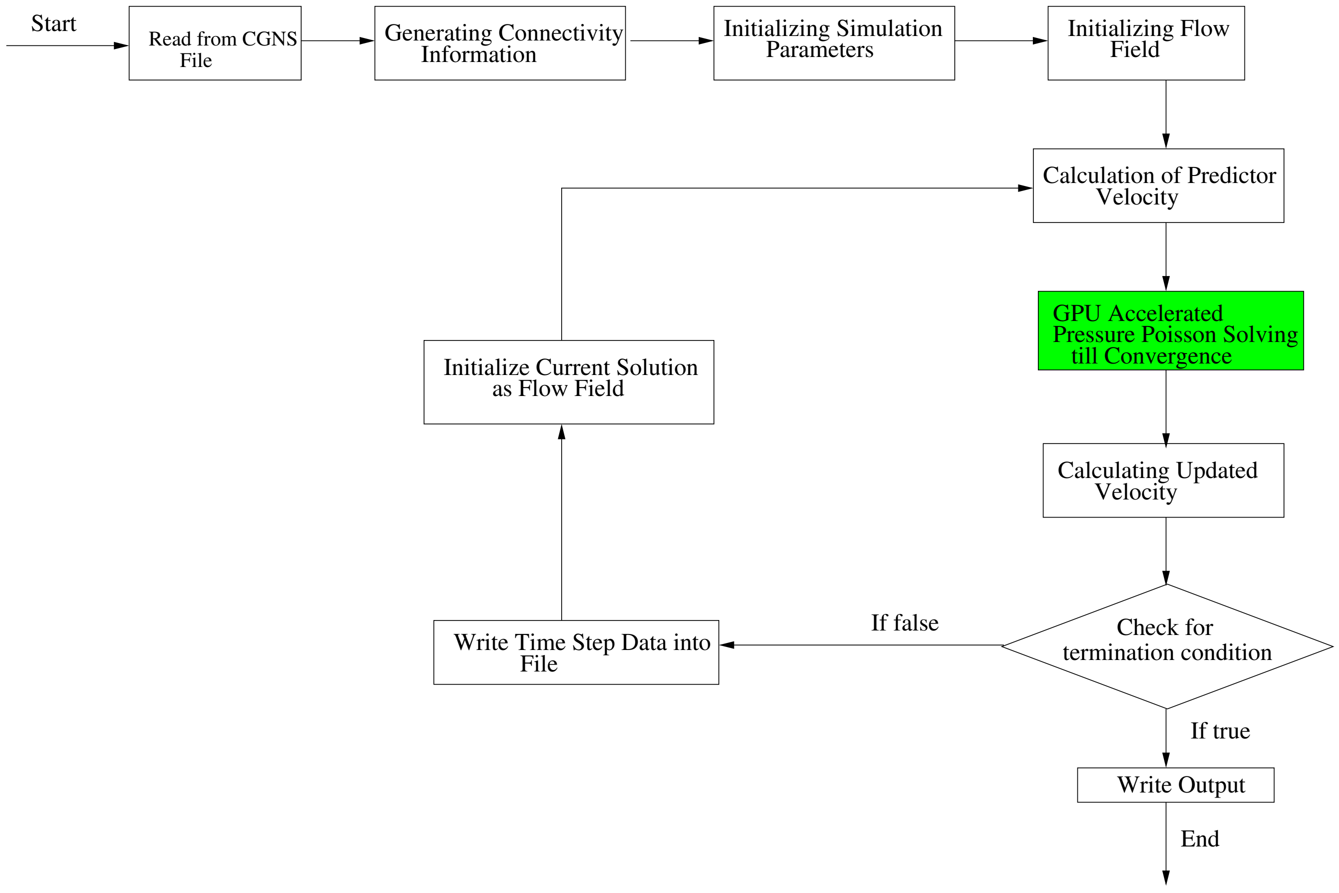

Start
Read from CGNS File
Generating Connectivity Information
Initializing Simulation Parameters
Initializing Flow Field
Calculation of Predictor Velocity
GPU Accelerated Pressure Poisson Solving till Convergence
Calculating Updated Velocity
Check for termination condition
If false
Write Time Step Data into File
Initialize Current Solution as Flow Field
If true
Write Output
End

Figure 5.3: Block Diagram of CFD Solver with GPU Accelerated Pressure Poisson Solving

Chapter 6

Experiments and Results

6.1 Performance Metrics

The usual performance metric used for non-multigrid solver is number of iterations for convergence. However, the same can't be used for multigrid as the grids are of different size and the amount of work done per iteration is not same across all levels. Hence *work units* [36] is generally used as the performance metric for multigrid solvers. Work units is defined as the sum total of number of updates in all levels normalized to number of points in the finest grid

$$Work\ units = \frac{Total\ no.\ of\ updates\ in\ all\ levels}{No.\ of\ points\ in\ finest\ grid}$$

The speed up achieved is calculated relative to non-multigrid serial implementation. Pre-processing time is not considered for result comparison as it is a one time activity. Often, different analysis/simulations are carried on same mesh and pre-processing need not be repeated. The pre-processed multigrid can also be stored persistently on the disk for further simulations.

6.2 Heat Transfer Problem

Heat transfer, as name suggests is the transfer of thermal energy from a body at a high temperature to another at a lower temperature. In unsteady state heat transfer problem, the temperature within the system varies with time. The unsteady state heat transfer problem is one of the fundamental problems in CFD and many other physical processes like potential flow, mass diffusion, flow through porous media etc. are governed by similar mathematical equations.

6.2.1 Steady and Unsteady State Heat Transfer Problem

The steady state heat transfer is governed by the equation,

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

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

$$a_i T_i = \sum_{j \in N(i)} a_{ij} T_j$$

$$a_{ij} = \frac{k_{ij} s_{ij}}{d_{ij}} \qquad a_i = \sum_{j \in N(i)} a_{ij}$$

where, k_{ij} = mean conductivity of i and j, s_{ij} = interface area between i and j and d_{ij} = distance between i and j.

The unsteady state 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 the specific heat and k is conductivity of the material. The discretization for the above governing equation is,

$$a_i T_i = \sum_{j \in N(i)} a_{ij} T_j + a_i^0 T_i^0$$

$$a_i^0 = \frac{\rho c v_i}{\Delta t} \qquad a_{ij} = \frac{k_{ij} s_{ij}}{d_{ij}} \qquad a_i = \sum_{j \in N(i)} a_{ij}$$

where, k_{ij} = mean conductivity of i and j, s_{ij} = interface area between i and j, d_{ij} = distance between i and j and v_i = control volume around i and Δt is the time step size [37].

6.2.2 Experimental Setup

Serial implementations which include non-multigrid as well as multigrid implementations are run on Intel Xeon E5-2600 2.60 GHz processor. The operating system used is 64-bit Ubuntu 12.04 LTS. Parallel implementation which includes non-multigrid and multigrid GPU implementations of AMG solver are run on NVIDIA Kepler K20Xm GPU with CUDA driver version 5.5. The GPU has 2668 cores, 6GB device memory. The solver has been written in C++ and is compiled using g++ 4.6.3, nvcc 5.5 compilers.

6.2.3 Results

The steady and unsteady state problems are solved on an unstructured unit cube as show in Fig 6.1. For steady state problem, the temperature at all faces except one is set constant at 300 and one face is set constant at 600. For unsteady state problem, the temperature at all faces except one is set constant at 300 and one face is kept sinusoidally varying, starting from 600 at time zero. The time step size of 0.01 is used and the experiment is run for 24000 time steps. The temperature variation given for each time step is

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

The multigrid parameters for different grids used in experiment are shown in Table 6.1. At each level in the multigrid, two iterations of Gauss Seidel is used to smooth the error. The α value used is 5. The β values starts with value of 1 in the first level and is incremented for each level.

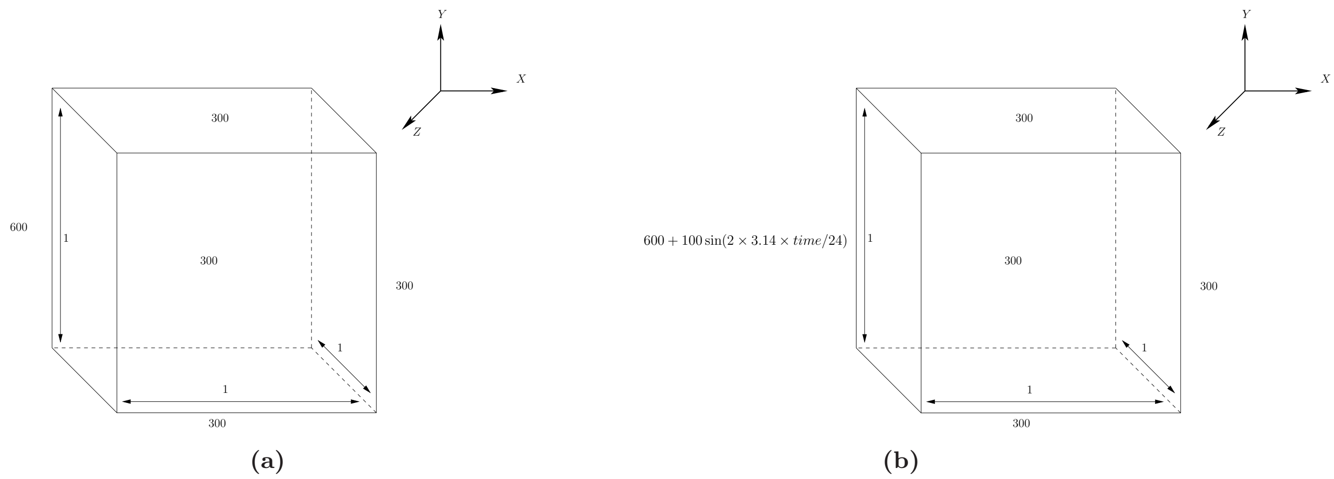


Figure 6.1: Computational Domain for (a) Steady state heat transfer problem (b)Unsteady state heat transfer problem

Grid Size	θ	Number of levels
89126	0.15	4
200337	0.1	4
305334	0.15	5
510940	0.1	5
701161	0.1	5
1699751	0.1	5
2345137	0.2	6

Table 6.1: Multigrid Parameters

Results for Steady State Heat Transfer Problem

Table 6.2 gives the work unit comparison between non-multigrid and multigrid solvers. The solve time for serial, parallel implementations of non-multigrid and multigrid solvers is summarized in Table 6.3. The speed up achieved by serial multigrid, parallel non-multigrid and parallel multigrid solvers is shown in Fig 6.2. Serial multigrid and parallel non-multigrid solvers achieve a speed up of 19x and 41x respectively where as the multigrid solver on GPU achieves speed up close to 2157x.

Grid Size	Without Multigrid	With Multigrid
89126	810	25.88
200337	1293	38.73
305334	1528	23.63
510940	2019	25.44
701161	2369	28.95
1699751	3555	28.76
2345137	4219	29.21

Table 6.2: Work Unit Comparison Between Non-multigrid and Multigrid Solvers

The temperature contour along $Z = 0.5$ plane for a grid of size 0.1 million is shown in Fig 6.3.

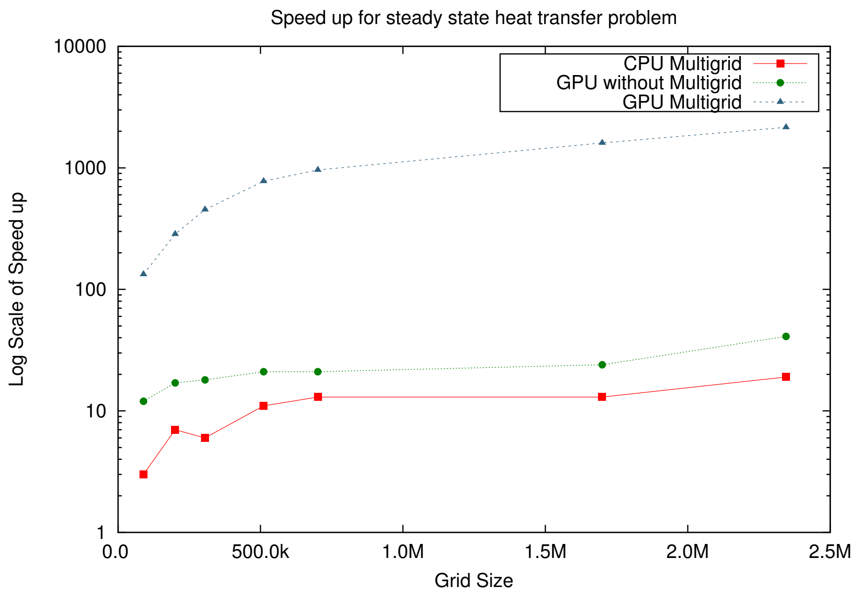


Figure 6.2: Solve time speed up comparison for flow steady state heat transfer problem

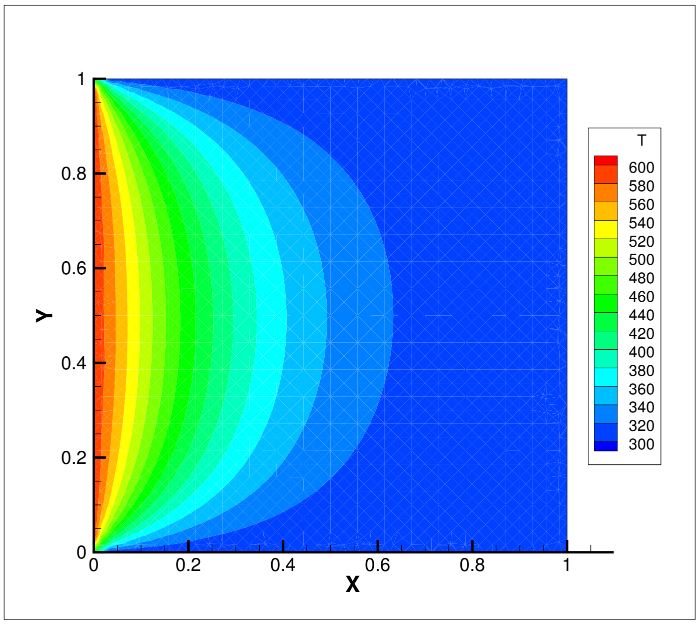


Figure 6.3: Temperature contour along $Z = 0.5$ plane

Results for Unsteady State Heat Transfer Problem

The solve time for serial, parallel implementations of non-multigrid and multigrid solvers is summarized in Table 6.4. The speed up achieved by serial multigrid, parallel non-multigrid and parallel multigrid solvers is shown in Fig 6.4. Parallel non-multigrid solver achieve a speed up of 20x where as the multigrid solver on GPU achieves speed of 29x. The speedup achieved by multigrid GPU implementation is understandably low when compared to that of steady state problem due to the fact that once steady state is reached, the multigrid has little impact as it takes only few iterations

Table 6.3: Solve Time Comparison for Steady State Heat Transfer Problem

Grid Size	CPU	CPU-MG	GPU	GPU-MG
89153	14.68 sec	3.86 sec	1.19 sec	0.11 sec
200337	59.85 sec	7.69 sec	3.43 sec	0.21 sec
305334	1 min 40 sec	15.52 sec	5.4 sec	0.22 sec
510940	3 min 53 sec	19.83 sec	11.09 sec	0.3 sec
701161	6 min 25 sec	27.81 sec	17.62 sec	0.4 sec
1699751	24 min 37 sec	1 min 48 sec	1 min	0.92 sec
2345137	40 min 38 sec	2 min 6 sec	1 min 38 sec	1.13 sec

for convergence at each time step.

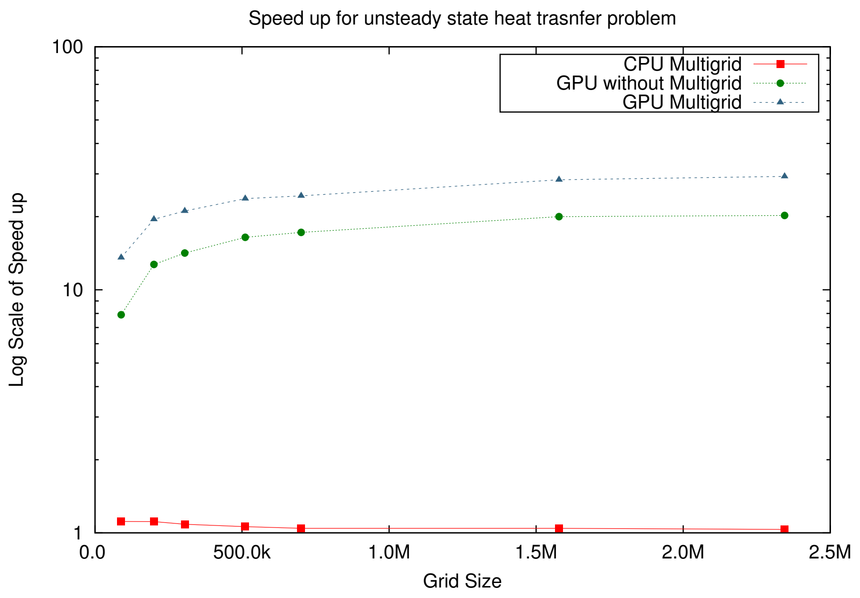


Figure 6.4: Solve time speed up comparison for flow unsteady state heat transfer problem

Table 6.4: Solve Time Comparison for Unsteady State Heat Transfer Problem

Grid Size	CPU without MG	CPU with MG	GPU without MG	GPU with MG
89153	9 min 5 sec	8 min 9 sec	1 min 9 sec	40.16 sec
200337	23 min 46 sec	21 min 20 sec	1 min 52 sec	1 min 13 sec
305334	33 min 46 sec	31 min 6 sec	2 min 23 sec	1 min 36 sec
510940	58 min 9 sec	54 min 49 sec	3 min 32 sec	2 min 27 sec
701161	1 hr 20 min 25 sec	1 hr 17 min	4 min 40 sec	3 min 18 sec
1699751	3 hr 34 min 43 sec	3 hr 25 min 43 sec	10 min 44 sec	7 min 34 sec
2345137	4 hr 54 min 47 sec	4 hr 45 min 14 sec	14 min 34 sec	10 min 4 sec

Correctness

The results of steady and unsteady problem obtained using our solver are compared against those obtained using commercial software ANSYS FLUENT. For steady state problem, Fig 6.5 compares the temperature on a line along Z dimension at $X = 0.1$ *and* $Y = 0.5$. For unsteady state problem, Fig 6.6 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 1.7% with respect to ANSYS results.

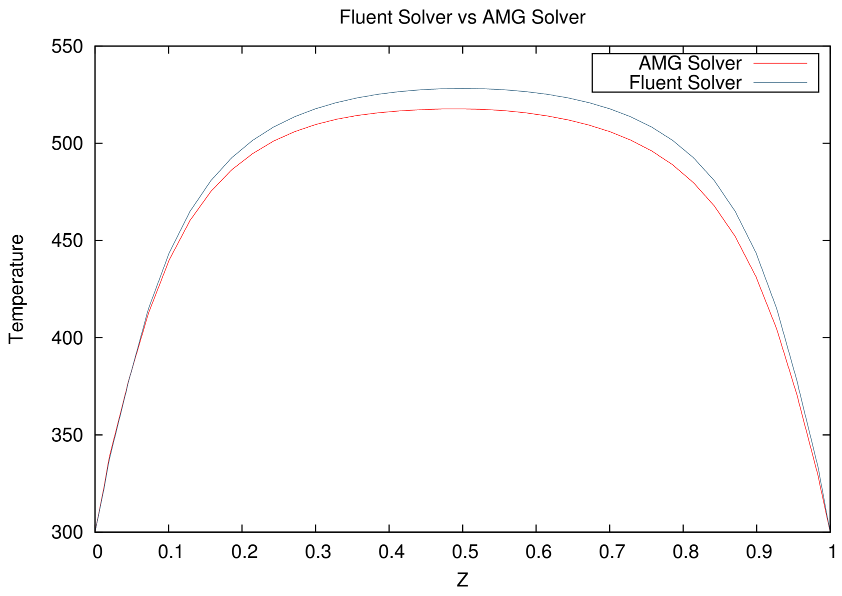


Figure 6.5: Temperature on a line along Z dimension at $X = 0.1\ and\ Y = 0.5$

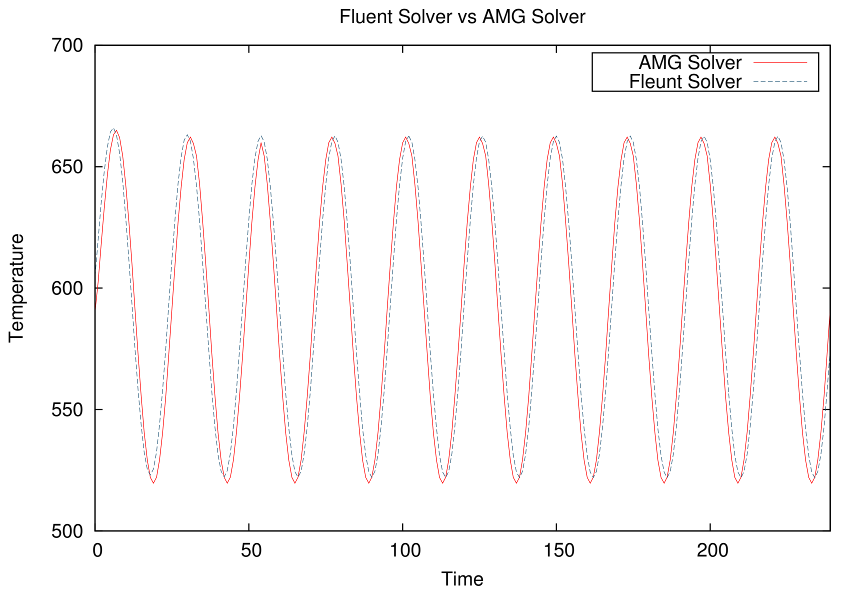


Figure 6.6: Temperature at point $X = 0.01,\ Y = 0.5\ and\ Z = 0.5$

6.3 Accelerating Pressure Poisson Solving in Navier-Stokes Problem

In our experiments, we consider Navier-Stokes equations that are solved to obtain velocity and pressure field using semi-implicit predictor-corrector approach [38]. The governing equations are discretized using finite volume approach. We use our GPU AMG solver to accelerate solving pressure Poisson equations, which is the most time consuming task in these simulations. The pressure Poisson equation is given by

$$\nabla^2 . P = \frac{\rho}{\Delta t} \nabla . \vec{U}$$

where P is the pressure, ρ is density of the fluid, t is the time step and $\vec{U}$ is the velocity field vector. Finite volume discretization of the equation is given by

$$\sum_f \nabla P_f . S_f = \frac{\rho}{\Delta t} \sum_f F_f$$

where ∇P_f is gradient of pressure at each face of the cell, S_f is the surface area of respective face of the cell and F_f is the flux through each face of the cell. Expanding the above equations further will result in equation of the form

$$a_i P_i = \sum_{j \in N(i)} a_{ij} P_j - \sum_{j \in N(i)} b_{ij} F_j$$

We solve Navier-Stokes equations for (a) 3D flow over lid driven cavity problem (convex grid) (b) 3D laminar flow past square cylinder problem (concave grid) and (c) Plain Jet Problem (concave grid).

6.3.1 Experimental Setup

Serial implementations which include non-multigrid as well as multigrid implementations are run on Intel Xeon CPU X3430 2.40 GHz. The operating system used is 64-bit CentOS 5.10. Parallel implementation which includes non-multigrid and multigrid GPU implementations of AMG solver are run on NVIDIA Kepler K20Xm GPU with CUDA driver version 5.5. The GPU has 2668 cores, 6 GB device memory and is controlled by host with Intel Xeon E5-2600 2.60 GHz processor. The operating system used is 64-bit Ubuntu 12.04 LTS. The solver has been written in C++ and is compiled using g++ 4.6.3, nvcc 5.5 compilers.

6.3.2 Results

Solving Navier-Stokes consists of solving pressure Poisson equations for multiple discrete time steps. Reaching Navier-Stokes steady state will take around thousands of such time steps. For our result comparisons, we use the time taken by our AMG solver to solve pressure Poisson equations for one such time step.

Results for Flow Over Lid Driven Cavity Problem

We solve Navier-Stokes equation for 3D flow over lid driven cavity. The computational domain is unit cube as shown in Fig 6.7. We use Dirichlet boundary condition for velocities at all the surfaces ($u = 1,\ v = 0\ and\ w = 0$ for lid and $u = 0,\ v = 0\ and\ w = 0$ for all other surfaces) and Neumann boundary condition for pressure at all surfaces ($\frac{\partial P}{\partial n} = 0$). The multigrid parameters for different grids used in experiment are shown in Table 6.5. At each level in the multigrid, two iterations of Gauss-Seidel is used to smooth the error.The β values starts with value of 1 in the first level and is incremented for each level.

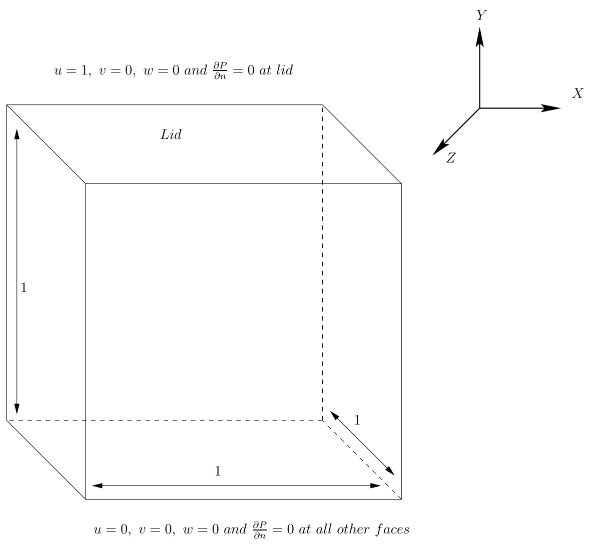

$u = 1,\ v = 0,\ w = 0\ and\ \frac{\partial P}{\partial n} = 0\ at\ lid$
Lid
1
1
1
Y
X
Z
$u = 0,\ v = 0,\ w = 0\ and\ \frac{\partial P}{\partial n} = 0\ at\ all\ other\ faces$

Figure 6.7: Computational domain for flow over lid driven cavity problem

Table 6.5: Multigrid Parameters for Flow Over Lid Driven Cavity Problem

Grid Size	$\boldsymbol{\theta}$	**Number of levels**	$\boldsymbol{\alpha}$
1060000	0.05	4	5
1580000	0.25	4	5
2100000	0.05	4	5
2620000	0.05	4	5

Table 6.6 gives the work unit comparison between non-multigrid and multigrid solvers. The Pressure Poisson solve time for serial, parallel implementations of non-multigrid and multigrid solvers is summarized in Table 6.7. The speed up achieved by serial multigrid, parallel non-multigrid and parallel multigrid solvers is shown in Fig 6.8. Serial multigrid and parallel non-multigrid solvers achieve a speed up of 3x and 600x respectively where as the multigrid solver on GPU achieves speed up close to 915x.

To validate the solutions, we solve Navier-Stokes problem till steady state using our GPU AMG solver to accelerate pressure Poisson solution. The X velocity contour along $Z = 0.5$ plane for grid of 1.06 million cells for RE 100 is shown in Fig 6.9. We also compare $X,\ Y$ velocity plots along center-line on $Z = 0.5$ plane for the same grid with those obtained by Ku et al. [39], using pseudo spectral method for RE 100 and RE 1000. Figures 6.10, 6.11 confirm that the results are in good agreement with the experimental results.

Table 6.6: Work Unit Comparison Between Non-multigrid and Multigrid Solvers for Flow Over Lid Driven Cavity Problem

Grid Size	Without Multigrid	With Multigrid
1060000	19604	1226.86
1580000	26620	1784.54
2100000	36036	3175.43
2620000	47242	8644.2

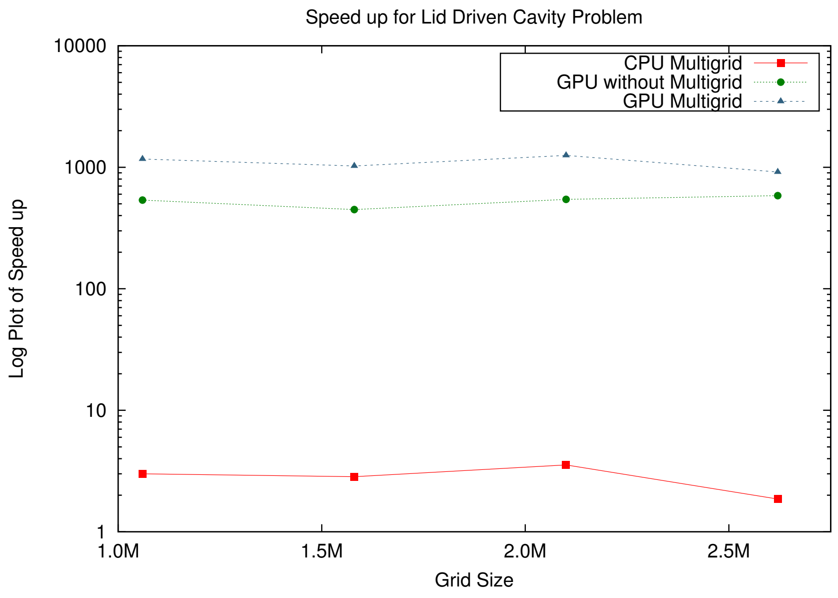


Figure 6.8: Pressure Poisson solve time speed up comparison for flow over lid driven cavity problem

Table 6.7: Pressure Poisson Solve Time Comparison for Flow Over Lid Driven Cavity Problem

Grid Size	CPU without MG	CPU with MG	GPU without MG	GPU with MG
1060000	3 hr 8 min 54 sec	1 hr 2 min 20 sec	21.09 sec	9.69 sec
1580000	5 hr 23 min 33 sec	1 hr 52 min 13 sec	42.89 sec	18.84 sec
2100000	11 hr 30 min 50 sec	3 hr 14 min 20 sec	1 min 16 sec	33.09 sec
2620000	20 hr 8 min 37 sec	10 hr 48 min 34 sec	2 min 4 sec	1 min 19 sec

Results for Laminar Flow Past Square Cylinder Problem

We also solve Navier-Stokes equations for laminar flow past square cylinder problem. The computational domain is concave as shown in Fig 6.12. We use the following boundary conditions:

- At Inlet : Dirichlet boundary condition for velocities ($u = 1,\ v = 0\ and\ w = 0$) and Neumann boundary condition for pressure ($\frac{\partial P}{\partial n} = 0$)

- At Outlet : Dirichlet boundary condition for pressure ($P = 0$) and Neumann boundary condition for velocities ($\frac{\partial u}{\partial n} = 0,\ \frac{\partial v}{\partial n} = 0\ and \frac{\partial w}{\partial n} = 0$)

- At all other surfaces : Dirichlet boundary condition for velocities ($u = 0,\ v = 0\ and\ w = 0$) and Neumann boundary condition for pressure ($\frac{\partial P}{\partial n} = 0$).

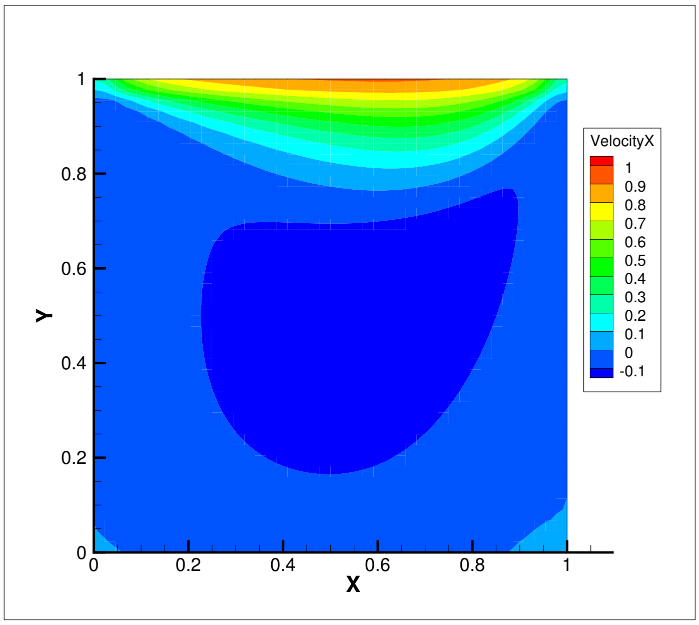


Figure 6.9: X velocity contour along $Z = 0.5$ plane for RE 100

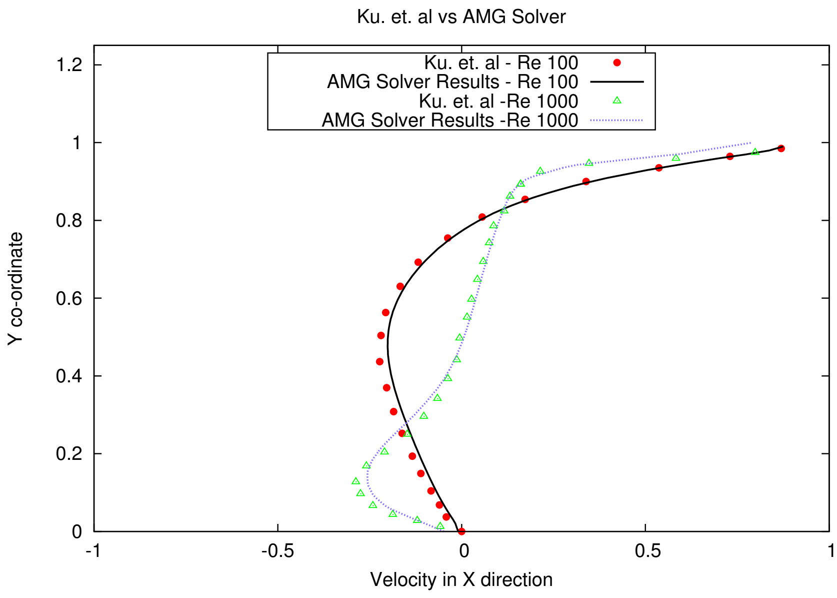


Figure 6.10: Velocity in X direction along the center line

The multigrid parameters for different grids used in experiment are shown in Table 6.8. At each level in the multigrid, two iterations of Gauss-Seidel is used to smooth the error.

Table 6.9 gives the work unit comparison between non-multigrid and multigrid solvers. The pressure Poisson solve time for serial, parallel implementations of non-multigrid and multigrid solvers is summarized in Table 6.10. The speedup achieved by serial multigrid, parallel non-multigrid and parallel multigrid solvers is shown in Fig 6.13. Serial multigrid and parallel non-multigrid solvers achieve a speed up of 2x and 113x respectively where as the multigrid solver on GPU achieves speed up close to 1020x.

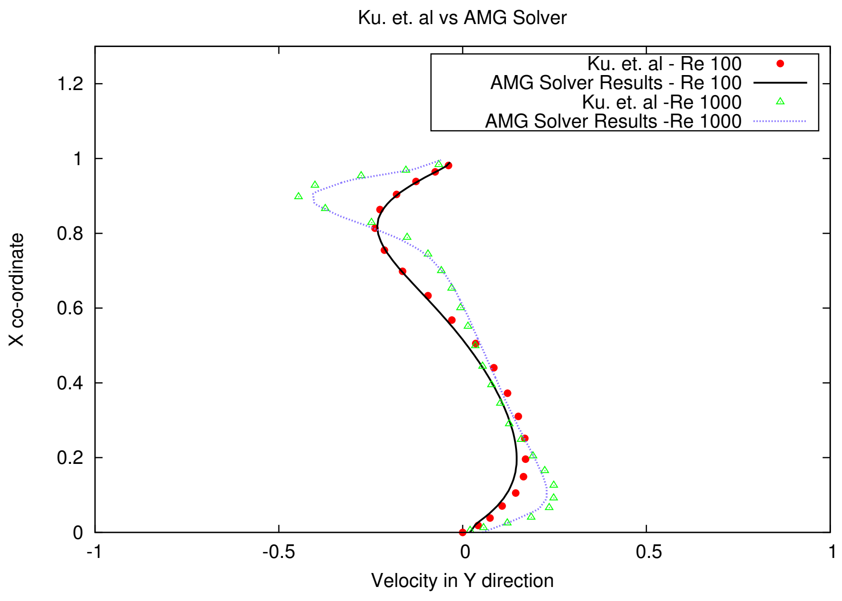


Figure 6.11: Velocity in Y direction along the center line

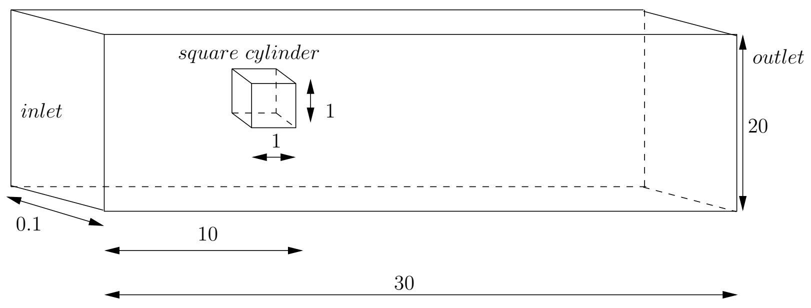


Figure 6.12: Computational domain for laminar flow past square cylinder problem

Table 6.8: Multigrid Parameters for Laminar Flow Past Square Cylinder Problem

Grid Size	θ	Number of levels	α	β
203748	0.12	3	5	0.2
302310	0.1	3	5	0.2
403510	0.01	4	5	0.2
1713160	0.05	4	5	0.2

Table 6.9: Work Unit Comparison between Non-Multigrid and Multigrid Solvers for Laminar Flow Past Square Cylinder Problem

Grid Size	Without Multigrid	With Multigrid
203748	179829	11030
302310	189092	12087.1
403510	326385	15430.6
1713160	634579	31458.3

To validate the solutions, we solve Navier-Stokes problem till steady state using our GPU AMG solver to accelerate pressure Poisson solution. The X velocity contour along $Z = 0.5$ plane for grid

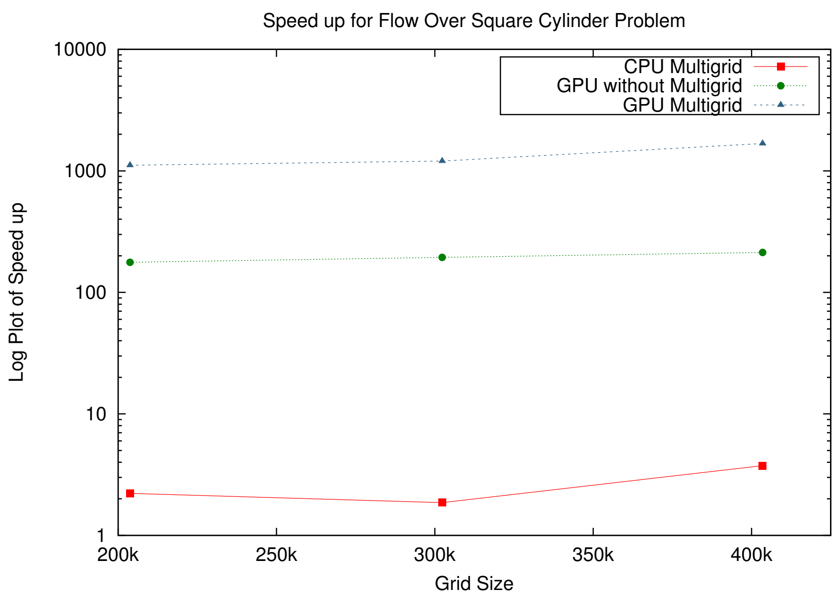


Figure 6.13: Pressure Poisson solve time speed up comparison for laminar flow past square cylinder problem

Table 6.10: Pressure Poisson Solve Time Comparison for Laminar Flow Past Square Cylinder Problem

Grid Size	CPU without MG	CPU with MG	GPU without MG	GPU with MG
203748	2 hr 2 min 25 sec	55 min 7 sec	41.35 sec	6.61 sec
302310	3 hr 4 min 40 sec	1 hr 39 min 10 sec	57.09 sec	9.19 sec
403510	7 hr 13 min 14 sec	1 hr 55 min 22 sec	2 min 1 sec	15.47 sec
1713160	26 hr 19 min 48 sec	14 hr 56 min 10 sec	13 min 53 sec	1 min 33 sec

of 0.3 million cells for RE 30 is shown in Fig 6.14. We also plot re-circulation length against Reynold number (Fig 6.15) and validate the same against those of Breuer M et al. [40].

Results for Plain Jet Problem

We solve Navier-Stokes equations for plain jet problem. The computational domain is concave as shown in Fig 6.16. We use the following boundary conditions:

- At Inlet 1 : Dirichlet boundary condition for velocities ($u = 0,\ v = 0\ and\ w = 1$) and Neumann boundary condition for pressure ($\frac{\partial P}{\partial n} = 0$) .
- At Inlet 2 : Dirichlet boundary condition for velocities ($u = 0,\ v = 0\ and\ w = 0$) and Neumann boundary condition for pressure ($\frac{\partial P}{\partial n} = 0$).
- At Outlet : Dirichlet boundary condition for pressure ($P = 0$) and Neumann boundary condition for velocities ($\frac{\partial u}{\partial n} = 0,\ \frac{\partial v}{\partial n} = 0\ and \frac{\partial w}{\partial n} = 0$).
- At all other surfaces : Dirichlet boundary condition for velocities ($u = 0,\ v = 0\ and\ w = 0$) and Neumann boundary condition for pressure ($\frac{\partial P}{\partial n} = 0$).

The multigrid parameters for different grids used in experiment are shown in Table 6.11. At each level in the multigrid, two iterations of Gauss-Seidel is used to smooth the error.

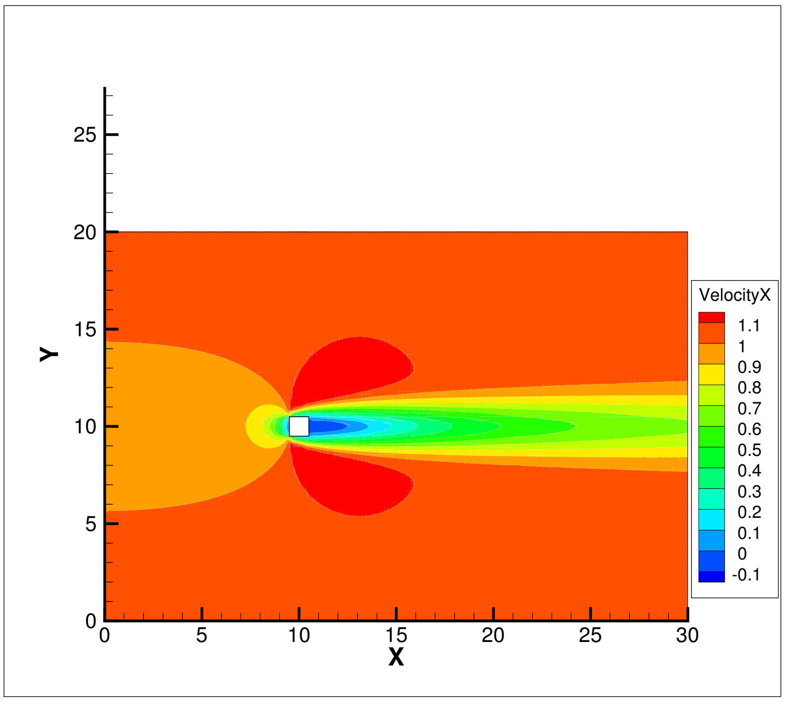


Figure 6.14: X velocity contour along $Z = 0.5$ plane for RE 30

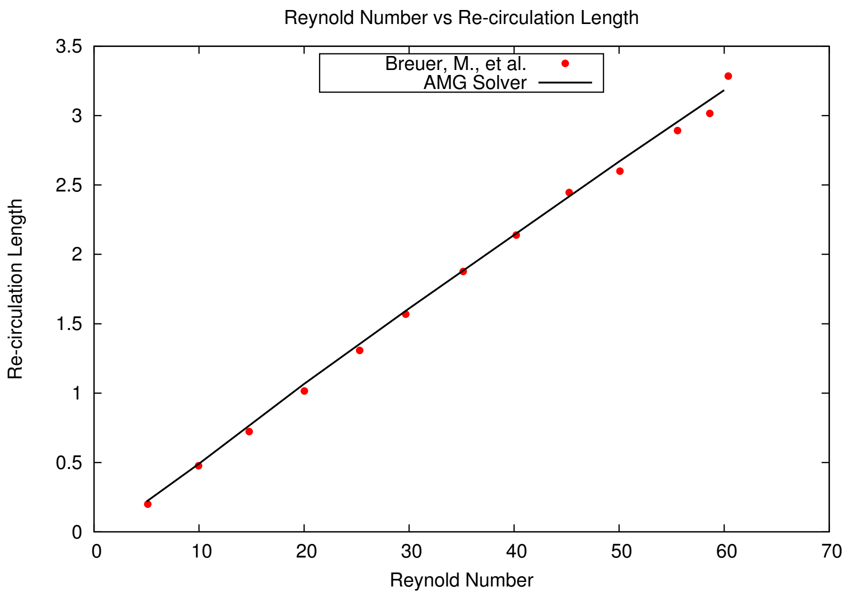


Figure 6.15: Reynold number vs re-circulation length for laminar flow past square cylinder problem

Table 6.12 gives the work unit comparison between non-multigrid and multigrid solvers. The pressure Poisson solve time for serial, parallel implementations of non-multigrid and multigrid solvers is summarized in Table 6.13. The speedup achieved by serial multigrid, parallel non-multigrid and parallel multigrid solvers is shown in Fig 6.17. Serial multigrid is slower than non-multigrid implementation and parallel non-multigrid solvers achieves a speed up of 15x where as the multigrid solver on GPU achieves speed up close to 47x.

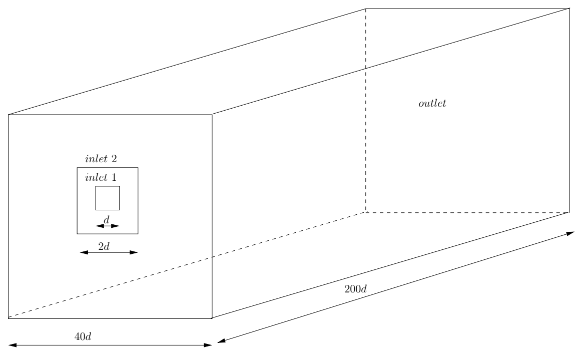


Figure 6.16: Computational domain for plain jet problem

Table 6.11: Multigrid Parameters for Plain Jet Problem

Grid Size	θ	Number of levels	α	β
281328	0.15	3	30	0.2
1547208	0.05	5	30	0.2
1930968	0.05	5	30	0.2
2698488	0.05	5	30	0.2

Table 6.12: Work Unit Comparison between Non-Multigrid and Multigrid Solvers for Plain Jet Problem

Grid Size	Without Multigrid	With Multigrid
281328	518388	79350.9
1547208	924602	51743.5
1930968	1038050	59595.8
2698488	1334640	67704

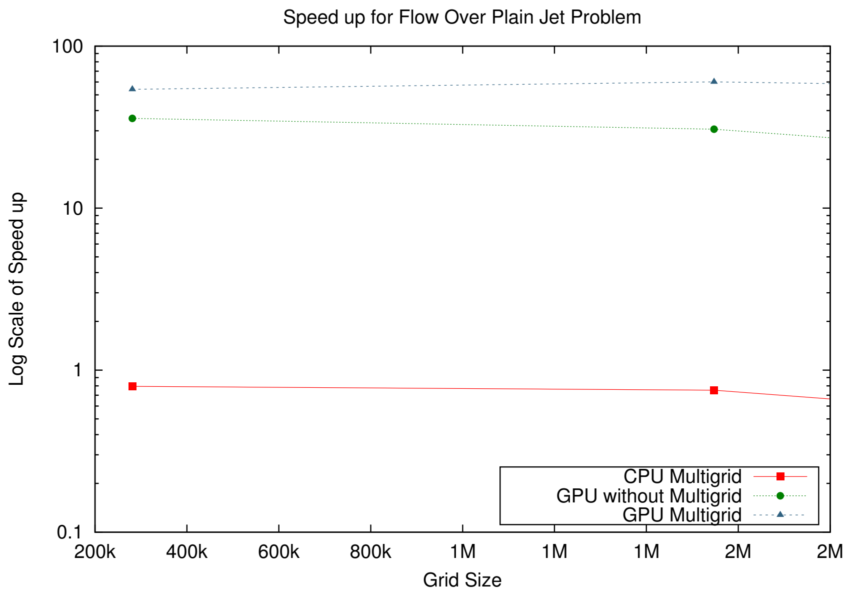


Figure 6.17: Pressure Poisson solve time speed up comparison for plain jet problem

To validate the solutions, we solve Navier-Stokes problem till steady state using our GPU AMG solver to accelerate pressure Poisson solution. The axial velocity in axial and radial directions

Table 6.13: Pressure Poisson Solve Time Comparison for Plain Jet Problem

Grid Size	CPU without MG	CPU with MG	GPU without MG	GPU with MG
281328	5 hr 17 min 7 sec	5 hr 40 min 7 sec	7 min 33 sec	5 min
1547208	1 day 10 hr 8 in 39 sec	1 day 21 hr 24 min 4 sec	1 hr 6 min 42 sec	34 min 5 sec
1930968	1 day 15 hr 46 min 20 sec	2 days 15 hr 52 min 53 sec	1 hr 33 min 37 sec	41 min 7 sec
2698488	1 day 17 hr 53 min 41 sec	4 days 9 hr 23 min 53 sec	2 hr 46 min 53 sec	53 min 55 sec

along $Y = 0.5$ plane for grid of 0.2 million cells for RE 10 is shown in Fig 6.18. We also plot axial distance against axial velocity (Fig 6.19) and validate the same against those generated by commercial software ANSYS FLUENT.

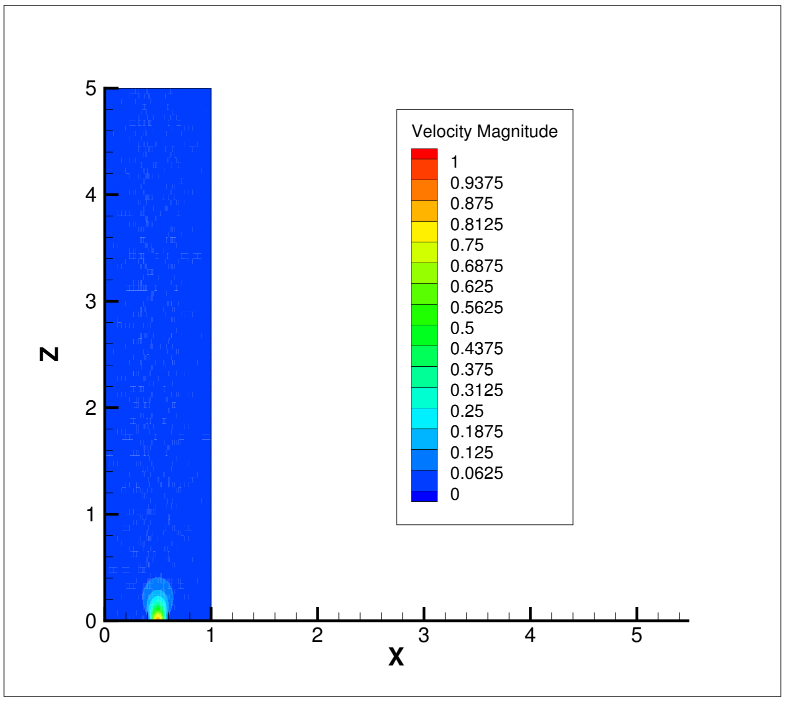


Figure 6.18: Axial velocity contour in axial and radial directions along $Y = 0.5$ plane for RE 10

6.4 Performance Gain Due to Improved Coarsening and A_c Transformation

In this section, we present in detail the performance gain due to improved coarsening algorithm and A_c transformation for faster GPU implementation. For doing so, we compare performance of implementations with and without improved coarsening, A_c transformations for (a) Heat Transfer Problem and (b) Accelerated pressure Poisson solving in Navier-Stokes problem.

Heat Transfer Problem

Tables 6.14, 6.15 gives the work unit and solve time comparison among implementations with and without improved Coarsening and A_c transformation. The implementations that employ original coarsening algorithm doesn't retain adequate boundary nodes at each level and hence diverge for most of the grids considered. Implementations with improved coarsening retain adequate number

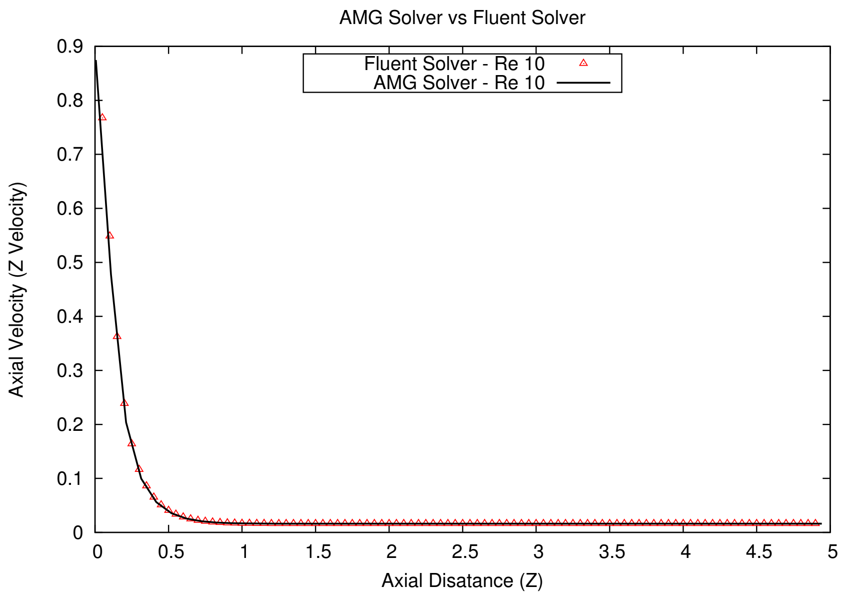


Figure 6.19: Axial Distance vs Axial Velocity for Plain Jet Problem

of boundary nodes per level and hence converge to produces results for all the grids. Tables 6.16, 6.17 compares the number of nodes and boundary nodes at each level for different implementations on a grid of size 2.3 million. Though A_c transformation causes slight increase in work units, it is compensated with reduced solve time due to smaller average degree per level. Figure 6.20 compares the average degree per level for different implementations on a grid of size 2.3 million. Without A_c transformation, the average degree in the coarsest level is as high as 900 where as with A_c transformation, the average degree in the coarsest level is mere 37. On the whole, the implementation that employs improved coarsening and A_c transformation outperforms other implementations.

Table 6.14: Work Unit Comparison for Steady State Heat Transfer Problem

Grid Size	Without A_c Transformation		With A_c Transformation	
	Original Coarsening	Improved Coarsening	Original Coarsening	Improved Coarsening
89126	21.2	18.58	28.11	25.88
200337	Diverging	20.72	Diverging	38.73
305334	62.63	19.79	52.84	23.63
510940	Diverging	17.4	92.73	25.44
701161	Diverging	18.43	Diverging	28.95
1699751	Diverging	33.37	Diverging	28.76
2345137	Diverging	18.39	Diverging	29.21

Accelerating Pressure Poisson Solving in Navier-StokesProblem

We consider plain jet problem to illustrate the performance gain due to improved coarsening and A_c transformation. Tables 6.18, 6.19 give the work unit and solve time comparison among implementations with and without improved Coarsening, A_c transformation. The implementations that employ original coarsening algorithm doesn't retain adequate boundary nodes at each level and

Table 6.15: Solve Time Comparison for Steady State Heat Transfer Problem

Grid Size	Without A_c Transformation		With A_c Transformation	
	Original Coarsening	Improved Coarsening	Original Coarsening	Improved Coarsening
89126	1.97 sec	2 sec	0.11 sec	0.11 sec
200337	Diverging	1.47 sec	Diverging	0.21 sec
305334	17.98 sec	10.04 sec	0.45 sec	0.22 sec
510940	Diverging	3.22 sec	1.05 sec	0.3 sec
701161	Diverging	3.6 sec	Diverging	0.4 sec
1699751	Diverging	14.23 sec	Diverging	0.92 sec
2345137	Diverging	38.26 sec	Diverging	1.13 sec

Table 6.16: Nodes Per Level in 2.3 Million Grid for Steady State Heat Transfer Problem

Level	Number of Nodes per Level			
	Without A_c Transformation		With A_c Transformation	
	Original Coarsening	Improved Coarsening	Original Coarsening	Improved Coarsening
1	2345137	2345137	2345137	2345137
2	596299	613230	596299	613230
3	150016	167499	128159	167931
4	45110	56712	68207	81205
5	16721	23773	24091	27686
6	6819	10950	8566	9578

Table 6.17: Boundary Nodes Per Level in 2.3 Million Grid for Steady State Heat Transfer Problem

Level	Number of Boundary Nodes per Level			
	Without A_c Transformation		With A_c Transformation	
	Original Coarsening	Improved Coarsening	Original Coarsening	Improved Coarsening
1	62683	62683	62683	62683
2	13858	36194	13858	36194
3	5860	18154	4335	18106
4	2159	9796	1584	9726
5	824	5162	606	4805
6	316	2580	268	2314

hence diverge for most of the grids considered. Implementations with improved coarsening retain adequate number of boundary nodes per level and hence converge to produces results for all the grids. Tables 6.20, 6.21 compares the number of nodes and boundary nodes at each level for different implementations on a grid of size 2.69 million. Though A_c transformation causes slight increase in work units, it is compensated with reduced solve time due to smaller average degree per level. Figure 6.21 compares the average degree per level for different implementations on a grid of size 2.69 million. Without A_c transformation, the average degree in the coarsest level is close to 300 where as with A_c transformation, the average degree in the coarsest level is only 85. On the whole, the implementation that employs improved coarsening and A_c transformation outperforms other implementations.

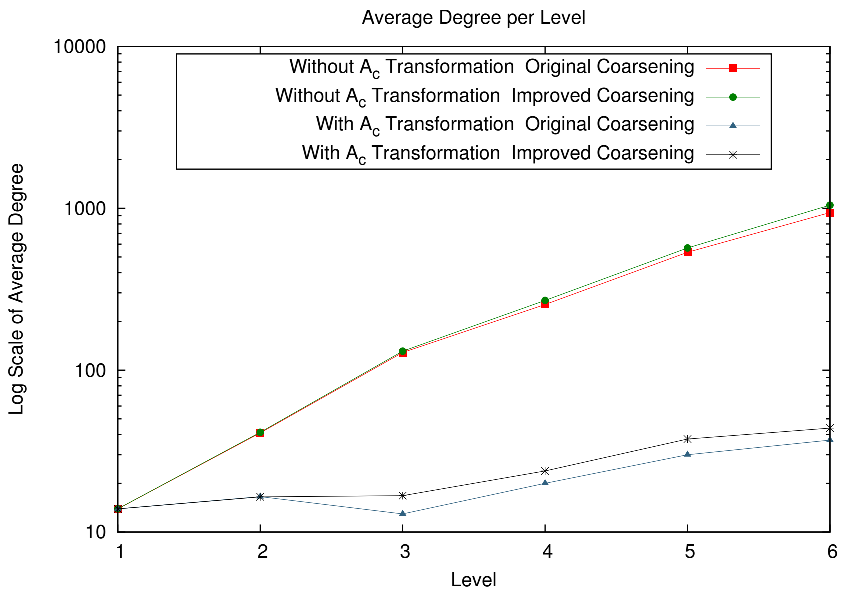


Figure 6.20: Average Degree per Level in 2.3 Million Grid for Steady State Heat Transfer Problem

Table 6.18: Work Unit Comparison for Plain Jet Problem

Grid Size	Without A_c Transformation		With A_c Transformation	
	Original Coarsening	Improved Coarsening	Original Coarsening	Improved Coarsening
281328	75231.4	79789.3	74565.4	79350.9
1547208	Diverging	60800.4	Diverging	51743.5
1930968	Diverging	67581.5	Diverging	59595.8
2698488	Diverging	63386.8	Diverging	67704

Table 6.19: Solve Time Comparison for Plain Jet Problem

Grid Size	Without A_c Transformation		With A_c Transformation	
	Original Coarsening	Improved Coarsening	Original Coarsening	Improved Coarsening
281328	6 min 15 sec	8 min 15 sec	4 min 6 sec	5 min
1547208	Diverging	6 hr 57 min 47 sec	Diverging	34 min 5 sec
1930968	Diverging	5 hr 57 min 15 sec	Diverging	41 min 6 sec
2698488	Diverging	4 hr 51 min 14 sec	Diverging	53 min 55 sec

Table 6.20: Nodes Per Level in 2.69 Million Grid for Plain Jet Problem

Level	Number of Nodes per Level			
	Without A_c Transformation		With A_c Transformation	
	Original Coarsening	Improved Coarsening	Original Coarsening	Improved Coarsening
1	2698488	2698488	2698488	2698488
2	1349244	1357528	1349244	1357528
3	429586	442860	437573	452008
4	141328	152495	142870	154367
5	40817	47930	40947	48170

Table 6.21: Boundary Nodes Per Level in 2.69 Million Grid for Plain Jet Problem

Level	Number of Boundary Nodes per Level			
	Without A_c Transformation		With A_c Transformation	
	Original Coarsening	Improved Coarsening	Original Coarsening	Improved Coarsening
1	143208	143208	143208	143208
2	71604	78000	71604	78000
3	8288	19456	10282	21547
4	2402	11810	2877	13081
5	678	6745	806	7269

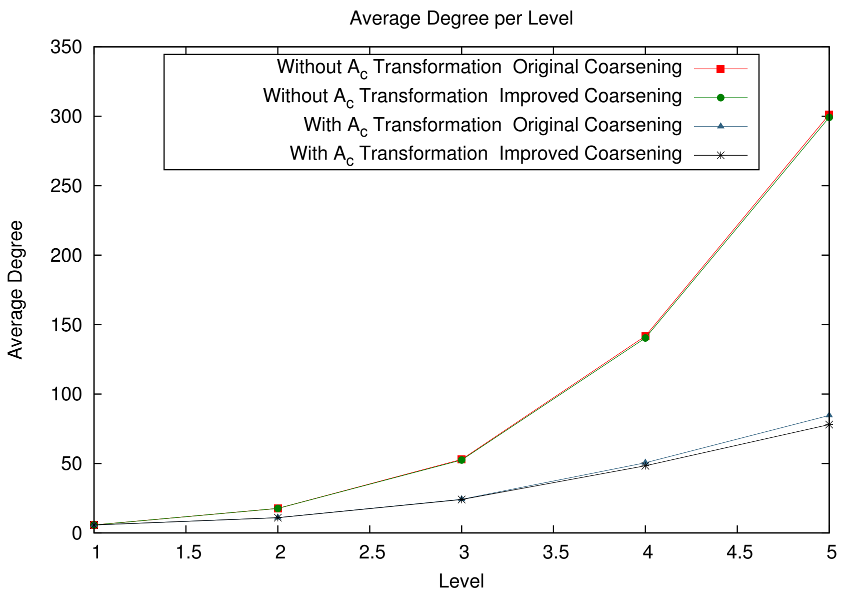


Figure 6.21: Average Degree per Level in 2.69 Million Grid for Plain Jet Problem

Chapter 7

Multi GPU Implementation and Results

The size of problems that can be solved on a GPU are limited by its memory. Similarly, the amount of speed up that can be achieved is also limited by the GPU hardware i.e., the number of GPU cores available. Thus, it is natural to go for multi GPU implementation to further speed up the computations and/or solve problems that do not fit into single GPU memory. In the following, we discuss the multi GPU implementation of AMG Solver.

7.1 Domain Decomposition

The first step towards a multi GPU implementation is to partition the working set to different GPUs. For AMG solver, the mesh graph corresponding to matrix A has be to partitioned among the GPUs. If k is the number of GPUs available, k-way partitioning (as defined in Definition 2) of the mesh graph is required.

Definition 2 (k-way graph partitioning [41]) *Given a graph $G = (V, E)$, partition V into k non-empty and disjoint sets $V_1, V_2, ..., V_k$ such that the number of edges connecting vertices of k groups is minimized*

k-way graph partitioning assigns to each vertex $i \in V$, a label $P(i) \in \{1...k\}$ indicating the partition to which the vertex belongs.

Definition 3 (Partition Boundary Node) *Node that shares an edge with a node belonging to different partition is called a partition boundary node. A node $i \in V$ is partition boundary node iff*

$$\exists_{j \in N(i)} P(i) \neq P(j)$$

Graph partitioning is known to be NP-Complete [42] and hence many heuristic algorithms exist to produce high quality partitioning. In our work, we use METIS [41] which is based on multi level graph partitioning approach for domain decomposition.

7.2 Multi GPU Implementation

7.2.1 Non-Multigrid Solver

Along with the mesh graph, the vertex data should also be partitioned. As the partition boundary nodes require the data from other partition boundary nodes, we store a copy of other partitions boundary data also as part of current partition data. To summarize, each GPU has data pertaining to its partition as well as other partitions boundary node data. We create as many CPU threads as the number of GPUs. Each CPU thread initializes its GPU, copies the corresponding partition data to global memory. The CPU threads simultaneously invokes Gauss Seidel smoothing kernel following which each GPU communicates its partition boundary node updates to other GPUs.

7.2.2 Multigrid Solver

Along with mesh graph and vertex data, the prolongation and restriction matrices are also partitioned. We use METIS only to partition the mesh graph corresponding to original grid (finest level). We make use of the fact that the C-points are subset of points in the original grid and partition the remaining grids in the hierarchy using Algorithm 2.

Algorithm 2 Partitioning Coarse Grids

```
Require: Partition label for all $i \in V$ in mesh graph corresponding to matrix $A$
Require: $n$ is the number of levels in the multigrid
1: for $i = 0$ to $n$ do
2:     $n_c \leftarrow$ no. of nodes in grid corresponding to level $i$
3:     for $j = 0$ to $n_c$ do
4:         $P_i(j) \leftarrow P_0(V_j)$ {partition label of vertex in finest grid which corresponds to vertex $j$ in level $i$ is assigned to $j$}
5:     end for
6: end for
```

We create as many CPU threads as the number of GPUs. Each CPU thread initializes its GPU, copies the corresponding partition data to global memory. The CPU threads simultaneously start the down cycle which involves the following: (a) calling Gauss Seidel kernel on corresponding GPU (b) communicating the partition boundary node updates to other GPUs and (c) calling restriction kernels on corresponding GPU. Once down cycle is completed, the CPU threads simultaneously start the up cycle which involves the following: (a) calling prolongation kernels on corresponding GPU (b) calling Gauss Seidel kernel on corresponding GPU and (c) communicating the partition boundary node updates to other GPUs.

7.3 Experimental Setup

All the experiments were carried out on a machine with two Intel Xeon E5-2600 2.60 GHz processors, each controlling 2 NVIDIA Kepler K20Xm GPUs with CUDA driver version 5.5. The operating system used is 64-bit Ubuntu 12.04 LTS. Each GPU has 2668 cores and 6 GB device memory.

OpenMP [43] directives are used to create and manage CPU threads. The solver has been written in C++ and is compiled using g++ 4.6.3, nvcc 5.5 compilers.

7.4 Results

For evaluation, we consider the same set of experiments that were described in chapter 6 and use solve time as the metric for comparison.

Heat Transfer Problem

Table 7.1 compares the solve time for non-multigrid single GPU and multi GPU implementations. Figure 7.1 shows the speed up achieved by the solver for grids of different size, when compared to that of non-multigrid single GPU implementation. For grids of size greater than 0.7 million, the multi GPU solver achieves a speed up of 3 times compared to single GPU solver.

Table 7.1: Non-Multigrid Multi GPU Solve Time Comparison for Steady State Heat Transfer Problem

Grid Size	1 GPU	2 GPUs	3 GPUs	4 GPUs
89126	1.2 sec	0.97 sec	1.04 sec	1.12 sec
200337	3.43 sec	2.48 sec	2.54 sec	2.15 sec
305334	5.4 sec	3.65 sec	3.44 sec	2.71 sec
510940	11.09 sec	6.94 sec	5.66 sec	4.84 sec
701161	17.62 sec	10.38 sec	8 sec	6.93 sec
859048	23.69 sec	13.91 sec	10.46 sec	8.69 sec
1577761	58.96 sec	33.55 sec	24.45 sec	20.25 sec
1699751	1 min	34.88 sec	25.45 sec	20.75 sec
2345137	1 min 38 sec	55.3 sec	39.77 sec	31.76 sec

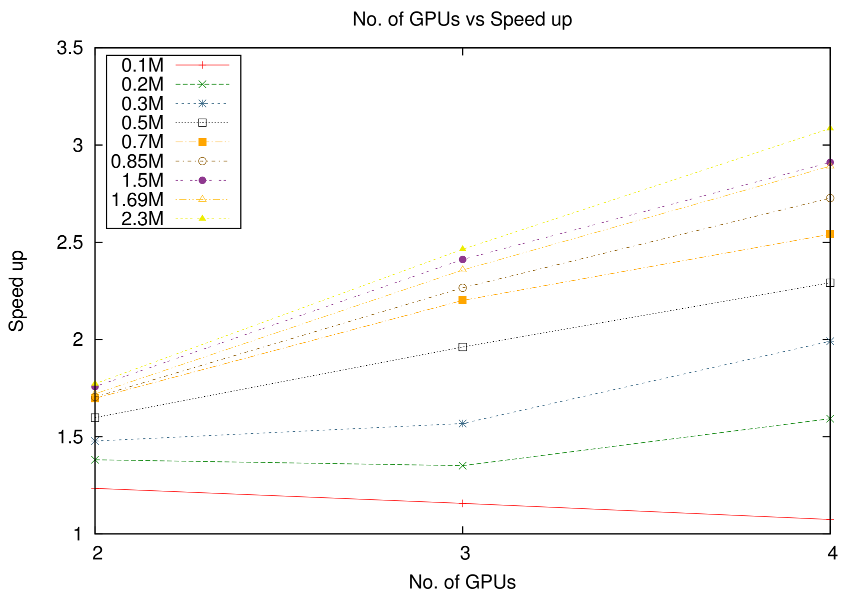


Figure 7.1: Non-Multigrid Multi GPU Solver Speed up for Heat Transfer Problem

Tables 7.2 compares the solve time for single GPU and multi GPU AMG solvers. Figure 7.2 shows the speed up achieved by the solver for grids of different size, when compared to that of single GPU AMG solver. For smaller grids (whose size is less than 1.5 million) the multi GPU solver is slower than single GPU solver due to the fact that the communication cost dominates computation cost. For grids of size greater than 1.5 million, the multi GPU solver achieves a speed up of 1.5 times compared to single GPU solver.

Table 7.2: Multigrid Multi GPU Solve Time Comparison for Steady State Heat Transfer Problem

Grid Size	1 GPU	2 GPUs	3 GPUs	4 GPUs
89126	0.11 sec	0.15 sec	0.18 sec	0.23 sec
200337	0.21 sec	0.25 sec	0.28 sec	0.31 sec
305334	0.22 sec	0.24 sec	0.27 sec	0.29 sec
510940	0.3 sec	0.33 sec	0.36 sec	0.34 sec
701161	0.4 sec	0.41 sec	0.42 sec	0.44 sec
859048	0.46 sec	0.48 sec	0.44 sec	0.46 sec
1577761	1.08 sec	0.86 sec	1.02 sec	1.05 sec
1699751	0.92 sec	0.88 sec	0.61 sec	0.55 sec
2345137	1.13 sec	0.84 sec	0.76 sec	0.74 sec

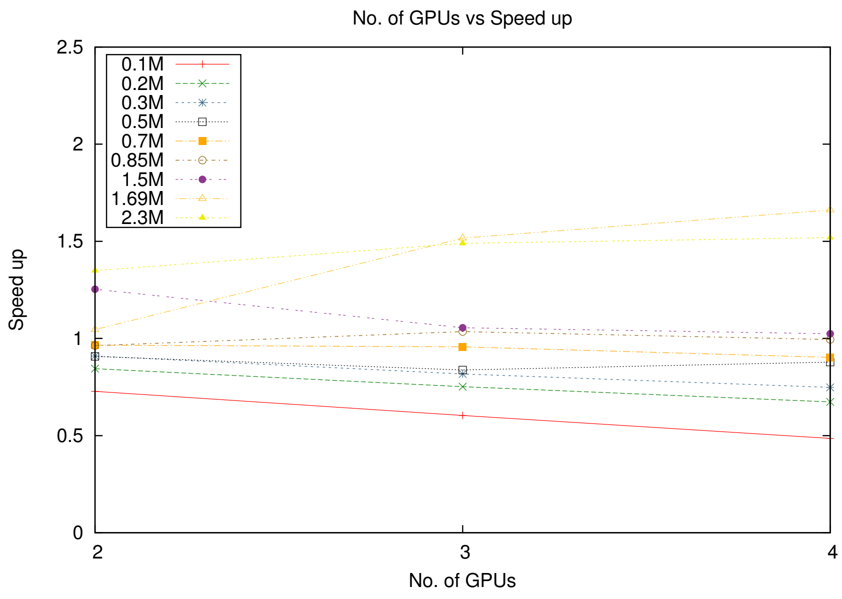


Figure 7.2: Multigrid Multi GPU Solver Speed up for Heat Transfer Problem

Accelerating Pressure Poisson Solving in Navier-StokesProblem

(a) Flow Over Lid Driven Cavity Problem

Table 7.3 compares the solve time for non-multigrid single GPU and multi GPU implementations. Figure 7.3 shows the speed up achieved by the solver for grids of different size, when compared to that of non-multigrid single GPU implementation. For grids of size greater than 2.6 million, the multi GPU solver achieves a speed up of 2 times compared to single GPU solver.

Table 7.3: Non-Multigrid Multi GPU Solve Time Comparison for Flow Over Lid Driven Cavity Problem

Grid Size	1 GPU	2 GPUs	3 GPUs	4 GPUs
1060000	51 sec	55 sec	44 sec	40 sec
1580000	1 min 59 sec	1 min 25 sec	1 min 21 sec	1 min 12 sec
2100000	3 min 43 sec	3 min 3 sec	2 min 32 sec	2 min 2 sec
2620000	5 min 47 sec	4 min 55 sec	3 min 36 sec	3 min 5 sec
3140000	9 min 1 sec	7 min 32 sec	5 min 29 sec	4 min 35 sec
16000000	1 hr 17 min 20 sec	1 hr 2 min 38 sec	44 min 2 sec	34 min 32 sec

.

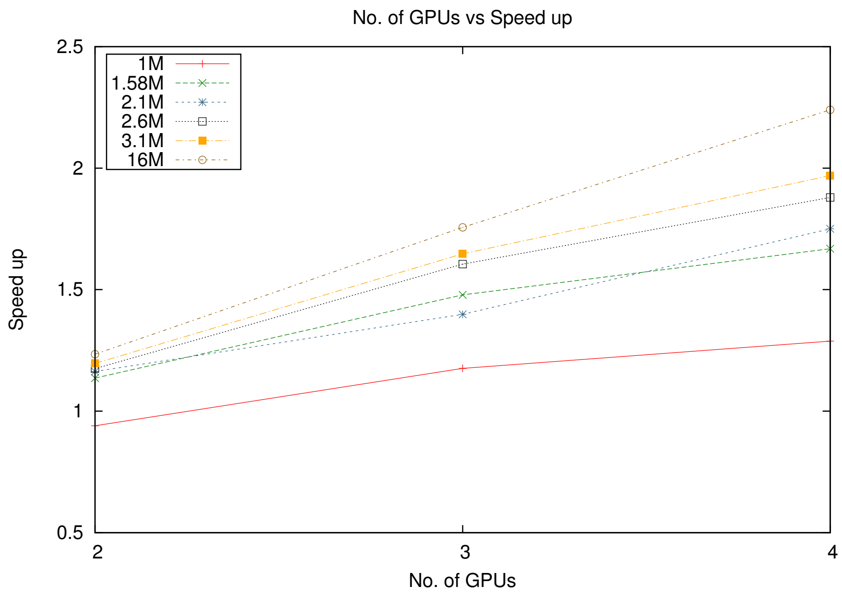


Figure 7.3: Non-Multigrid Multi GPU Solver Speed up for Flow Over Lid Driven Cavity Problem

(b) Laminar Flow Past Square Cylinder Problem

Table 7.4 compares the solve time for non-multigrid single GPU and multi GPU implementations. Figure 7.4 shows the speed up achieved by the solver for grids of different size, when compared to that of non-multigrid single GPU implementation. For smaller grids (of size less than 1.7 million), multi GPU implementation is slower than single GPU implementation due to fact that communication cost dominates the computation cost. For grids of size greater than 1.7 million, the multi GPU solver achieves a speed up of 1.6 times compared to single GPU solver.

Table 7.4: Non-Multigrid Multi GPU Solve Time Comparison for Laminar Flow Past Square Cylinder Problem

Grid Size	1 GPU	2 GPUs	3 GPUs	4 GPUs
203748	1 min 20 sec	1 in 55 sec	1 min 58 sec	2 min 16 sec
302310	1 min 54 sec	2 min 20 sec	2 min 26 sec	2 min 33 sec
403510	4 min 10 sec	5 min 9 sec	4 min 45 sec	5 min 16 sec
1713160	39 min 10 sec	35 min 11 sec	27 min 27 sec	23 min 35 sec
2073800	1 hr 7 min 31 sec	1 hr 18 sec	45 min 13 sec	39 min 5 sec

.

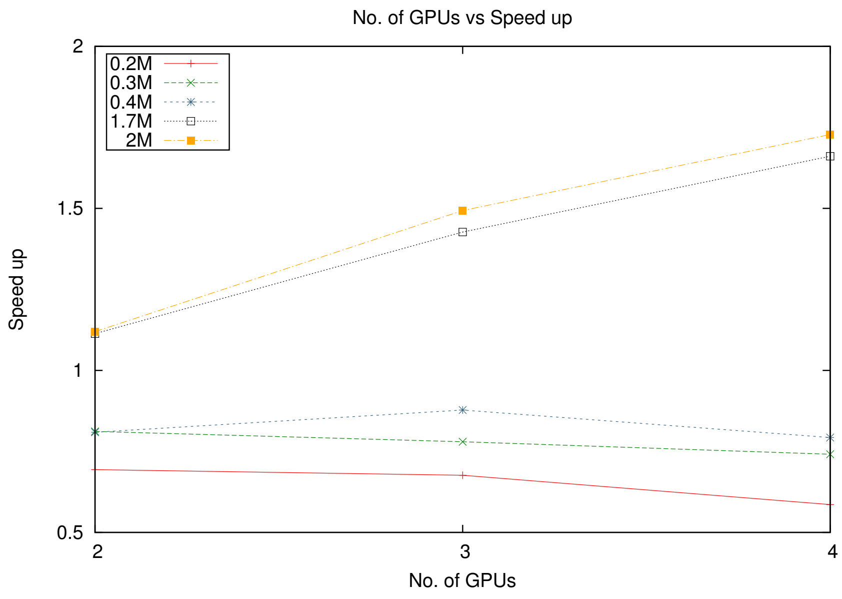


Figure 7.4: Non-Multigrid Multi GPU Solver Speed up for Laminar Flow Past Square Cylinder Problem

(b) Plain Jet Problem

Table 7.4 compares the solve time for non-multigrid single GPU and multi GPU implementations. Figure 7.4 shows the speed up achieved by the solver for grids of different size, when compared to that of non-multigrid single GPU implementation. For grid of size 2.7 million, the multi GPU solver achieves a speed up of 1.9 times compared to single GPU solver.

Table 7.5: Non-Multigrid Multi GPU Solve Time Comparison for Plain Jet Problem

Grid Size	1 GPU	2 GPUs	3 GPUs	4 GPUs
281328	7 min 33 sec	9 min 48 sec	8 min 52 sec	8 min 41 sec
1547208	1 hr 6 min 41 sec	1 hr 51 sec	45 min 33 sec	40 min 8 sec
1930968	1 hr 33 min 37 sec	1 hr 23 min 7 sec	1 hr 1 min 40 sec	52 min 56 sec
2698488	2 hr 46 min 53 sec	2 hr 21 min 50 sec	1 hr 43 min 58 sec	1 hr 26 min 53 sec

For the various experiments carried out, the non-multigrid multi GPU solver achieves a speed up of close to 3 times,compared to single GPU non-multigrid solver. For heat transfer problem, multi GPU AMG achieves a speed up of 1.5 times, compared to single GPU solver. The performance of multi GPU AMG solver depends on various factors like number of levels in the multigrid and the size of coarsest grid in the hierarchy etc. The underlying GPU interconnect also has significant impact on communication latency among the GPUs. To improve the efficiency of multi GPU AMG solver, further study and analysis has to be carried out on effective overlapping of computation and communication on GPUs which increases per GPU SM Utilization

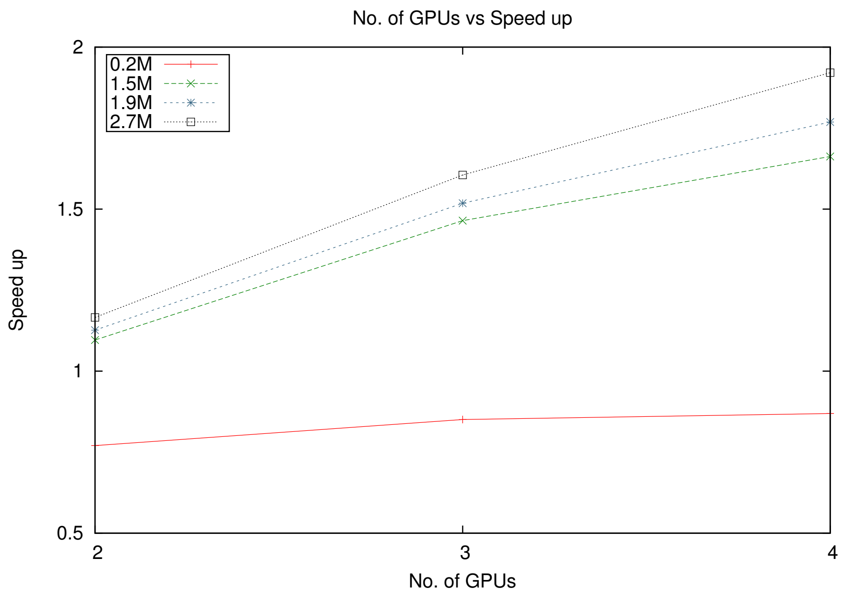


Figure 7.5: Non-Multigrid Multi GPU Solver Speed up for Plain Jet Problem

Chapter 8

Conclusion and Future Work

In this work, we implemented a parallel AMG Solver for three dimensional unstructured grids on GPU. The quality of coarsening and GPU acceleration is improved by retaining more boundary nodes and by reducing high degree nodes in coarse grids. Our GPU implementation uses graph representations that aid coalesced memory access. We also extend the implementation to multiple GPUs using METIS for domain decomposition. Both the solvers (single GPU as well as Multi GPU) are integrated with in-house developed CFD software to solve Navier-Stokesequations. To evaluate the speed up given by our multigrid GPU implementation, we solve heat transfer problem on unit cube and Navier-Stokes problems, with GPU accelerated pressure Poisson solving, on both convex and concave grids of the order of 2 million cells. We also validate the solutions obtained by our implementation against published or commercial software generated results.

The primary focus of the work is on improving solve time and not on pre-processing time as it is considered to be a one time activity. However, pre-processing time can be improved using (a) parallel coarsening techniques (b) parallel graph coloring and (c) parallel graph partitioning in case of multi GPU implementation. Further changes have to be done to coarsening and interpolation to deal with positive off-diagonal entries (due to cross diffusion terms) in matrix A. Different coarsening techniques like aggressive coarsening, aggregate coarsening etc and different interpolation techniques like direct and indirect interpolation etc. can also be tried out. As CPU and GPU executions are asynchronous, it will be worthwhile to have an implementation which splits the work between CPU and GPUs instead of completely offloading the work to GPUs. Further study and analysis has to be done on different mechanism to effectively overlap computation and communication on the GPUs so as to boost the speed up of multi GPU AMG solver.

References

[1] S. P. Vanka, A. F. Shinn, and K. C. Sahu. Computational fluid dynamics using graphics processing units: challenges and opportunities. In ASME 2011 International Mechanical Engineering Congress and Exposition. American Society of Mechanical Engineers, 2011 429–437.

[2] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In ACM Transactions on Graphics (TOG), volume 22. ACM, 2003 908–916.

[3] J. D. Owens, S. Sengupta, and D. Horn. Assessment of graphic processing units (gpus) for department of defense (dod) digital signal processing (dsp) applications. *Department of Electrical and Computer Engineering, University of California, Davis, Tech. Rep. ECE-CE-2005-3* .

[4] G. P. Williams. Numerical integration of the three-dimensional Navier-Stokes equations for incompressible flow. *Journal of Fluid Mechanics* 37, (1969) 727–750.

[5] A. Brandt, S. McCoruick, and J. Huge. ALGEBRAIC MULTIGRID (AMG) FOR SPARSE MATRIX EQUATIONS. *Sparsity and its Applications* 257.

[6] A. Brandt. Algebraic multigrid theory: The symmetric case. *Applied Mathematics and Computation* 19, (1986) 23–56.

[7] G. Golubovici and C. Popa. Interpolation and related coarsening techniques for the algebraic multigrid method. In Multigrid Methods IV, 201–213. Springer, 1994.

[8] A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, T. A. Manteuffel, S. F. McCormick, G. N. Miranda, and J. W. Ruge. Robustness and scalability of algebraic multigrid. *SIAM Journal on Scientific Computing* 21, (2000) 1886–1908.

[9] 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.

[10] 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* .

[11] U. M. Yang. Parallel algebraic multigrid methodshigh performance preconditioners. Springer, 2006.

[12] R. D. Falgout and U. M. Yang. hypre: A library of high performance preconditioners. In Computational ScienceICCS 2002, 632–641. Springer, 2002.

[13] Z. Li, Y. Saad, and M. Sosonkina. pARMS: a parallel version of the algebraic recursive multilevel solver. *Numerical linear algebra with applications* 10, (2003) 485–509.

[14] R. Biswas. Parallel Computational Fluid Dynamics: Recent Advances and Future Directions. DEStech Publications, Inc, 2010.

[15] E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In Proceedings of the 47th AIAA Aerospace Sciences Meeting. 2009 2009–565.

[16] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics* 22, (2008) 443–456.

[17] G. Haase, M. Liebmann, C. C. Douglas, and G. Plank. A parallel algebraic multigrid solver on graphics processing units. In High performance computing and applications, 38–47. Springer, 2010.

[18] 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* .

[19] 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.

[20] 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.

[21] 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.

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

[23] A. Darte, Y. Robert, and F. Vivien. Scheduling and Automatic Parallelization. Birkhaüser, 2000.

[24] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Program Optimization System. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 2008 `http://pluto-compiler.sourceforge.net/`.

[25] T. Grosser, A. Größlinger, and C. Lengauer. Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22. `http://polly.llvm.org/`.

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

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

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

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

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

[31] K. Stüben. Algebraic multigrid (AMG): an introduction with applications. GMD-Forschungszentrum Informationstechnik, 1999.

[32] K. Stuben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics* 128, (2001) 281–309.

[33] R. D. Falgout. An introduction to algebraic multigrid. *Computing in Science and Engineering* 8, (2006) 24–33.

[34] H.-C. Hege and H. Stüben. Vectorization and parallelization of irregular problems via graph coloring. In Proceedings of the 5th international conference on Supercomputing. ACM, 1991 47–56.

[35] J. Sebastian, N. Sivadasan, and R. Banerjee. GPU Accelerated Three Dimensional Unstructured Geometric Multigrid Solver. *Accepted in IEEE HPCS 2014* .

[36] R. H. Pletcher, J. C. Tannehill, and D. Anderson. Computational fluid mechanics and heat transfer. CRC Press, 2012.

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

[38] A. Dalal, V. Eswaran, and G. Biswas. A finite-volume method for Navier-Stokes equations on unstructured meshes. *Numerical Heat Transfer, Part B: Fundamentals* 54, (2008) 238–259.

[39] H. C. Ku, R. S. Hirsh, and T. D. Taylor. A pseudospectral method for solution of the three-dimensional incompressible Navier-Stokes equations. *Journal of Computational Physics* 70, (1987) 439–462.

[40] M. Breuer, J. Bernsdorf, T. Zeiser, and F. Durst. Accurate computations of the laminar flow past a square cylinder based on two different methods: lattice-Boltzmann and finite-volume. *International Journal of Heat and Fluid Flow* 21, (2000) 186–196.

[41] D. LaSalle and G. Karypis. Multi-threaded graph partitioning. In Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE, 2013 225–236.

[42] T. N. Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters* 42, (1992) 153–159.

[43] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5, (1998) 46–55.