METHODS FOR MULTILEVEL PARALLELISM ON GPU CLUSTERS: APPLICATION TO A MULTIGRID ACCELERATED NAVIER-STOKES SOLVER

by

Dana A. Jacobsen

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2011

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Dana A. Jacobsen

Thesis Title: Methods for Multilevel Parallelism on GPU Clusters: Application to a Multigrid Accelerated Navier-Stokes Solver

Date of Final Oral Examination: 13 October 2010

The following individuals read and discussed the thesis submitted by student Dana A. Jacobsen, and they evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Inanc Senocak, Ph.D.	Chair, Supervisory Committee
Amit Jain, Ph.D.	Member, Supervisory Committee
Timothy Barth, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Inanc Senocak, Ph.D., Chair of the Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

ACKNOWLEDGMENTS

I would like to thank my advisor, İnanç Şenocak, for his continuous guidance and support throughout this research. I would also like to thank the other committee members, Amit Jain of Boise State University and Timothy Barth of NASA Ames Research Center, for serving on my committee — not always an easy task with a multidisciplinary thesis topic. Thanks as well to Boise State University and Marty Lukes in particular for help with our local GPU computing infrastructure.

Lastly, I would like to thank my children Alexander, Catherine, and Sophia for sharing their father with this project.

I am grateful for funding during my graduate studies, partially provided by research initiation grants from NASA-Idaho EPSCoR and the NASA Idaho Space Grant Consortium. We utilized the Lincoln Tesla Cluster at the National Center for Supercomputing Applications under grants number ASC090054 and ATM100032, without which the multi-GPU plots would have many fewer GPUs shown. Computing resources from the Texas Advanced Computing Center (TACC) at The University of Texas at Austin were also used via the National Science Foundation's TeraGrid infrastructure.

ABSTRACT

Computational Fluid Dynamics (CFD) is an important field in high performance computing with numerous applications. Solving problems in thermal and fluid sciences demands enormous computing resources and has been one of the primary applications used on supercomputers and large clusters. Modern graphics processing units (GPUs) with many-core architectures have emerged as general-purpose parallel computing platforms that can accelerate simulation science applications substantially. While significant speedups have been obtained with single and multiple GPUs on a single workstation, large problems require more resources. Conventional clusters of central processing units (CPUs) are now being augmented with GPUs in each compute-node to tackle large problems.

The present research investigates methods of taking advantage of the multilevel parallelism in multi-node, multi-GPU systems to develop scalable simulation science software. The primary application the research develops is a cluster-ready GPU-accelerated Navier-Stokes incompressible flow solver that includes advanced numerical methods, including a geometric multigrid pressure Poisson solver. The research investigates multiple implementations to explore computation / communication overlapping methods. The research explores methods for coarse-grain parallelism, including POSIX threads, MPI, and a hybrid OpenMP-MPI model. The application includes a number of usability features, including periodic VTK (Visualization Toolkit) output, a run-time configuration file, and flexible setup of obstacles to

represent urban areas and complex terrain. Numerical features include a variety of time-stepping methods, buoyancy-driven flow, adaptive time-stepping, various iterative pressure solvers, and a new parallel 3D geometric multigrid solver. At each step, the project examines performance and scalability measures using the Lincoln Tesla cluster at the National Center for Supercomputing Applications (NCSA) and the Longhorn cluster at the Texas Advanced Computing Center (TACC). The results demonstrate that multi-GPU clusters can substantially accelerate computational fluid dynamics simulations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API – Application Programming Interface

CFD – Computational Fluid Dynamics

CFL – Courant-Friedrichs-Lewy

CPU – Central Processing Unit

CUDA – Compute Unified Device Architecture

FLOPS – Floating-point Operations Per Second

GPGPU – General Purpose GPU

GPU – Graphics Processor Units

HPC – High Performance Computing

MPI – Message Passing Interface

PDE – Partial Differential Equations

RBGS – Red-Black Gauss-Seidel

SDK – Software Development Kit

SIMD – Single Instruction Multiple Data

SOR – Successive Overrelaxation

CHAPTER 1

INTRODUCTION

Graphics processing units (GPUs) have enjoyed rapid adoption within the high-performance computing (HPC) community. GPU clusters, where fast network connected compute-nodes are augmented with latest GPUs [98], are now being used to solve challenging problems from various domains. Kindratenko et al. [65] describe applications being run on GPU clusters in cosmology, molecular dynamics, and quantum chemistry, among others. To be specific, *multi-GPU clusters* are defined in this thesis as those where each compute-node of the cluster has at least two GPUs. Examples include the 384 GPU Lincoln Tesla cluster deployed in February 2009 by the National Center for Supercomputing Applications (NCSA) at University of Illinois at Urbana Champaign [79], the 512 GPU Longhorn cluster deployed in January 2010 by the Texas Advanced Computing Center (TACC) [78], the 680 GPU TSUBAME 1.2 cluster deployed in October 2008 at the Tokyo Institute of Technology [73], the 4640 GPU Dawning Nebulae cluster at the National Supercomputing Centre in Shenzhen [58], and the 7168 GPU Tianhe-1A cluster at the National Supercomputing Center in Tianjin [59]. Many more systems are planned, such as the planned 4224 GPU TSUBAME 2.0 system and DARPA's recent $25 million research grant for a Cray XE6 with NVIDIA GPU accelerators. Most major HPC system manufacturers now have roadmaps for GPU acceleration on their large systems.

Many multi-GPU clusters such as the previous examples use two GPUs per node. On these systems, it can be efficient to use a dual-level parallel method using NVIDIA's Compute Unified Device Architecture (CUDA) [81], or Open Computing Language (OpenCL) [63] for fine-grain GPU parallelism and the Message-Passing Interface (MPI) [52] for coarse-grain parallelism. The overhead of inter-node communication between the two GPUs is generally not a first order effect. Some systems have densities as high as eight GPUs per node [56, 57], wherein the inter-node overhead of MPI can be substantial. Therefore, there is a need to investigate multilevel parallelism on emerging GPU clusters using advanced simulation science software.

The use of graphics hardware for general-purpose computation has been pursued since the late 1970s using hardware rendering pipelines to perform general computing processing. GPU computing has evolved in the present decade to the modern General Purpose Graphics Processing Unit (GPGPU) paradigm [82]. Owens et al. [83] survey the early history (1978-2003) as well as the state of GPGPU computing to 2007. Early work on GPU computing is extensive and used custom programming to reshape the problem in a way that could be processed by a rendering pipeline, often one without 32-bit floating-point support. The advent of DirectX 9 hardware in 2003 with floating-point support, combined with early work on high-level language support such as BrookGPU, Cg, and Sh, led to a rapid expansion of the field [12, 43, 51, 67, 70, 109]. The use of GPUs for Euler solvers and incompressible Navier-Stokes solvers has been well documented [17, 26, 29, 41, 62, 95, 105].

A recent report from the United States Department of Energy investigates exascale systems for computational science [100]. It highlights the need for computational performance orders of magnitude greater than those delivered by systems today. Areas including energy, climate modeling, biological modeling, and astrophysics are dis-

cussed in detail. With challenges in performance, cost, and power, GPU acceleration looks well poised to help achieve the necessary goals. The Defense Advanced Research Projects Agency (DARPA) recently awarded a $25 million research grant to NVIDIA, Cray, Oak Ridge National Labs, and six universities to develop exascale computing systems for HPC using GPUs. This is an area still seeing rapid development in both hardware and software.

1.1 Thesis Statement

The prime objective of the present research is to focus on ideas related to exploiting the multiple levels of parallelism in a multi-GPU cluster. The research investigates and develops multilevel parallel computing strategies for multi-GPU clusters using advanced numerical methods within a Navier-Stokes solver. Numerous parallelization strategies, including dual-level MPI-CUDA and tri-level MPI-OpenMP-CUDA, implementations are shown. Challenges to achieving scalable performance on multi-GPU clusters are presented. A systematic assessment of computational performance of the implementations on the National Center for Supercomputing Applications (NCSA) Lincoln Tesla cluster are shown. Relevant literature reviews are included in each chapter.

The results include:

- Multigrid solver for the pressure Poisson equation.
- Dual-level parallel MPI-CUDA cluster implementation that enables larger problems and increased performance.
- Tri-level parallel MPI-OpenMP-CUDA implementation.

- Investigation of the challenges and impacts of the different multilevel parallelization methods (Pthreads-CUDA, MPI-CUDA, and MPI-OpenMP-CUDA).
- Scaling and efficiency investigation of the cluster implementations.
- Temperature physics to calculate buoyancy driven flow.
- Validation of the accuracy of the multilevel parallel implementations.

1.2 Works Published

Works published during the course of study:

- D. Jacobsen, I. Senocak, "Massively Parallel Incompressible Navier-Stokes Computations on the NCSA Lincoln Tesla Cluster," Poster session presented at the NVIDIA GPU Technology Conference, September 2009.
- D. Jacobsen, J. Thibault, I. Senocak, "An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters," 48th AIAA Aerospace Sciences Meeting, January 2010.
- I. Senocak and D. Jacobsen, "Acceleration of Complex Terrain Wind Predictions Using Many-Core Computing Hardware," 5th International Symposium on Computational Wind Engineering, May 2010.
- D. Jacobsen, I. Senocak, "Parallel 3D Geometric Multigrid Solver on GPU Cluster," Poster session presented at the NVIDIA GPU Technology Conference, September 2010.

- D. Jacobsen and I. Senocak, "Scalability of Incompressible Flow Computations on Multi-GPU Clusters Using Dual-Level and Tri-Level Parallelism," 49th AIAA Aerospace Sciences Meeting, January 2011 (to appear).

- D. Jacobsen and I. Senocak, "Dual-Level Parallel Geometric Multigrid Solver for GPU Clusters," 49th AIAA Aerospace Sciences Meeting, January 2011 (to appear).

CHAPTER 2

BACKGROUND

2.1 Clusters and GPU Hardware

Many problems in high performance computing require very large amounts of computational performance, and are often run on supercomputers. The advent of general purpose GPU programming has brought immense changes to the field, with some applications requiring a Top 500 supercomputer 10 years ago now able to execute on a desktop machine equipped with one or more GPUs. GPU clusters hold promise to greatly reduce the time taken for some models as well as enable finer scale or larger problems to be considered. They are likely to be a primary component in upcoming exascale HPC systems [100].

Both NVIDIA and AMD make general purpose computing hardware and supply associated software. The present research targets NVIDIA hardware and the CUDA programming model for a number of reasons. The AMD software solutions were split into HAL (very low level), CAL (higher level constructs), and Brook, which is a C-syntax for writing data parallel programs. The unity offered by CUDA was attractive, as well as the software ecosystem of examples, forums, and research papers using CUDA. For instance, the recently announced Portland Group CUDA C compiler for x86 platforms allows the same CUDA code to be take advantage of both GPUs and multi-core CPUs [60]. More importantly, a wider variety of NVIDIA hardware was

available to the researcher, including a large cluster, which is the focus of the research. Hence, the application and research focuses on NVIDIA hardware using CUDA. With OpenCL now supported on both platforms, this split between hardware vendors is not as important in 2010 as it was when the research was started in late 2008.

NVIDIA produces multiple types of GPU hardware that can be used for scientific computation, however, all use the CUDA programming model. The Tesla series of computing servers are distinguished by their large memory (1.5GB in the older C870 model, 4GB in the C1060 model, and 6GB in the latest Tesla C2070 model) and lack of video output. The large memory size in particular makes the Tesla line of GPUs attractive for scientific computation. Tesla series have more rigorous testing, which may result in less downtime and lost results. Tesla S-series (S870, S1070, and S2070) are composed of four GPUs and have a rack mount 1U form factor that makes installation in large clusters much easier. Consumer models such as the GTX 200 and GTX 400 series are targeted for general graphics tasks at a reduced price. Consumer models can also be used for scientific computation with no software modifications. However, the amount of device memory (0.5 to 1.5GB on most models) on consumer models can be a limiting factor for large computational problems. Larger memory on each individual GPU reduces the total number of nodes needed for a given problem size, which means fewer or smaller network switches and reduced system cost. The price of fast networking (e.g. Infiniband) can be significant compared to compute hardware, and hence is a major item in cost analysis of a GPU cluster.

Users of high performance computing systems expect reliable hardware. One benefit of the Tesla series is its rigorous testing, which may result in less downtime and lost results. The error correcting (ECC) memory available in the Tesla C20x0 and S20x0 models provides a degree of reliability desired for very large scale installations.

Another benefit for cluster designs is the rack mount 1U form factor on some models (Tesla S870, S1070/S1075, and S2050/S2070) that makes installation in large clusters much easier. Finally, the larger memory on each individual GPU may reduce the total number of nodes needed for a given problem size, which will mean fewer or smaller network switches may be needed, thus reducing the system cost. The price of large Infiniband switches can be very significant, and hence is a major item in cost analysis of a GPU cluster.

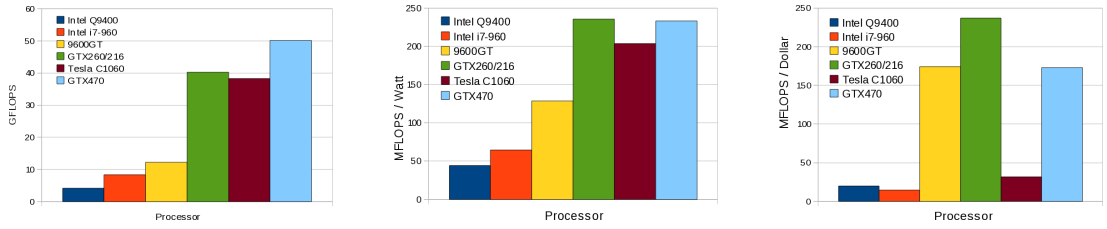
	Intel Q9400	Intel i7-960	9600GT	GTX260/216	Tesla C1060	GTX470
Type	CPU	CPU	GPU	GPU	GPU	GPU
Date Introduced	Aug 2008	Oct 2009	Feb 2008	Dec 2008	June 2008	Mar 2010
Processing Units	4	4 (8)	64	216	240	448
Compute Capability	N/A	N/A	CC1.1	CC1.3	CC1.3	CC2.0
Device Memory	N/A	N/A	512MB	896MB	4096MB	1280MB
Power	95W	130W	95W	171W	188W	215W
Street Price	$210	$570	$70	$180	$1,200	$290
Peak GFLOPS	85.3	102.4	336	805	936	1089
Sustained GFLOPS	3.2	8.4	12.2	40.3	38.3	50.1
MFLOPS/Watt	33.7	64.6	128.4	235.7	203.7	233.0
MFLOPS/Dollar	15.2	14.7	174.3	237.1	32.9	172.8

Table 2.1: Six selected computing platforms. Street price in US dollars as of 15 September 2010. Sustained GFLOPS numbers are based on the performance of a $512 \times 32 \times 512$ incompressible flow lid-driven cavity simulation with single precision.

The cost and energy consumption of modern supercomputers (e.g. Roadrunner at Los Alamos National Laboratory and Jaguar at Oak Ridge National Laboratory) are substantial. In addition to the raw computational power, metrics such as price/performance and power/performance are valuable in assessing the potential of new computing hardware. Table 2.1 shows configuration data and single device computational performance of the CFD code used in the present study on a selection of contemporary CPU and GPU platforms. In the comparisons, we consider a lid-driven cavity benchmark problem [37] that is large enough to show sustained throughput on these devices ($512 \times 32 \times 512$). For CPUs, the Intel Q9400 (Core2)

and i7-960 (Nehalem) were considered, using all cores. Hyperthreading was tried on the i7-960 but showed no advantage with the CFD code. On the GPU side, the consumer models NVIDIA 9600GT, NVIDIA GTX 260 Core 216, and NVIDIA GTX 470 were compared as well as the high performance computing model C1060. Pthreads was used to exploit the multiple cores of the CPU devices, while CUDA utilized the multiple processors on the GPUs. All calculations were done using single-precision floating-point.

The peak single-precision GFLOPS (billions of floating-point operations per second) based on manufacturer specifications are also given in Table 2.1. NVIDIA GPUs are capable of one single-precision multiply-add and one single-precision multiply per clock cycle per thread processor. The theoretical peak flops is therefore 3 flops $\times$ clock rate $\times$ number of thread processors. The Intel Core2 and Nehalem architectures are capable of 4 single-precision multiply-adds per cycle per core using SSE4, therefore the Intel Q9400 can sustain 4 cores $\times$ 2.667 GHz $\times$ 8 flops = 85.3 single-precision GFLOPS. The Intel i7-960 sustains 4 cores $\times$ 3.2 GHz $\times$ 8 flops = 102.4 single-precision GFLOPS. Intel gives reference numbers for double precision [61].



(a) (b) (c)

Figure 2.1: Three performance metrics on six selected CPU and GPU devices based on incompressible flow computations on a single device. Actual sustained performance for the single-node CFD code used in the present research is used rather than peak device performance. (a) Sustained GFLOPS, (b) MFLOPS/Watt, (c) MFLOPS/Dollar

Figure 2.1a shows three performance metrics on each platform using an incompressible flow CFD code. The GPU version of the CFD code is clearly an improvement over the Pthreads shared-memory parallel CPU version. Both of these implementations are written in C and use identical numerical methods [105]. The main impact on individual GPU performance was the introduction of compute capability 1.3, which greatly reduces the memory latency in some computing kernels due to the relaxed memory coalescing rules [81]. Compute capability 2.0 brought a number of important changes, with a small L1 cache being perhaps the most important for the current research. While the cache is quite small in current products (16-48k shared between all threads), it narrows the performance gap between global memory and shared memory code for many kernels. Support for double precision was added with compute capability 1.3, and the performance enhanced with compute capability 2.0 on professional devices.

Figure 2.1b shows the performance relative to the peak electrical power of the device. GPU devices show a definite advantage over the CPUs in terms of energy-efficient computing. The consumer video cards have a slight power advantage over the Tesla series, partly explained by having significantly less active global memory. The recent paper by Kindratenko et al. [65] details the measured power use of two clusters built using NVIDIA Tesla S1070 accelerators. They find significant power usage from intensive global memory accesses, implying CUDA kernels using shared memory not only can achieve higher performance but can use less power at the same time. Approximately 70% of the S1070's peak power is used while running the molecular dynamics program NAMD [87]. Figure 2.1c shows the performance relative to the street price of the device, which sheds light on the cost effectiveness of GPU computing. The consumer GPUs are better in this regard, ignoring other

factors such as the additional memory present in the compute server GPUs.

The rationale for clusters of GPU hardware is identical to that for CPU clusters – larger problems can be solved and total performance increases. Figure 2.1c indicates that clusters of commodity hardware can offer compelling price/performance benefits. By spreading the models over a cluster with multiple GPUs in each node, memory size limitations can be overcome such that inexpensive GPUs become practical for solving large computational problems. Today's motherboards can accommodate up to 8 GPUs in a single node [57], enabling large-scale compute power in small to medium size clusters. However, the resulting heterogeneous architecture with a deep memory hierarchy creates challenges in developing scalable and efficient simulation applications. This thesis focuses on maximizing performance on a multi-GPU cluster through a series of mixed MPI-CUDA implementations.

2.2 Programming Models

To achieve high throughput and scalable results from a Computational Fluid Dynamics (CFD) model on a multi-GPU platform currently requires the use of multiple programming APIs along with a domain decomposition strategy for data-parallelism. For small problems running on a single GPU, execution time is minimized as no GPU/host communication is performed during the computation, and all optimizations are done within the GPU code. CUDA does not address multiple GPUs, so other APIs must be used to support multi-GPU programming. When more than one GPU is used, cells at the edges of each GPU's computational space must be communicated to the GPUs sharing the domain boundary so they have the current data necessary for their computations. This injects additional latency into the implementation, which

may restrict scalability if not properly handled.

CUDA is the API used by NVIDIA for their GPUs. CUDA programming consists of kernels that run on the GPU and are executed by all the processor units in a SIMD (Single Instruction Multiple Data) fashion. The CUDA API also extends the host C API with operations such as `cudaMemcpy()`, which performs host/device memory transfers. Memory transfers between GPUs on a single host are done by using the host as an intermediary – there are no CUDA commands to operate between GPUs.

OpenCL is a unified API which can take a single-source application and run it effectively on multi-CPU machines, GPUs from NVIDIA, AMD, and Intel, as well as many future devices. With both major graphic card vendors (AMD and NVIDIA) on board and shipping software development kits (SDKs) and drivers, it looks very promising for the future of general purpose GPU programming. In many ways, programming with OpenCL is similar to CUDA, as most of the CUDA kernels used in this research would have very minor changes in an OpenCL system, as no CUDA-specific features are used. However, portability is still somewhat limited, as some ATI hardware from as recent as 2008 are not supported, including all the ATI hardware available to the researcher. More importantly, performance measurements in early 2010 showed identical stencil kernel performance on NVIDIA hardware to be superior with CUDA to OpenCL. Future software development tools should eliminate or alleviate this difference, but at the present time the difference exists and was as much as 2× on some kernel / hardware combinations.

POSIX Threads [20] (Pthreads) and OpenMP [103] are two APIs used for running parallel code on shared-memory computers. These APIs both use a shared memory space model. Combined with CUDA, multiple GPUs can perform computation, copy their neighboring cells to the host, synchronize with their neighbor threads, and copy

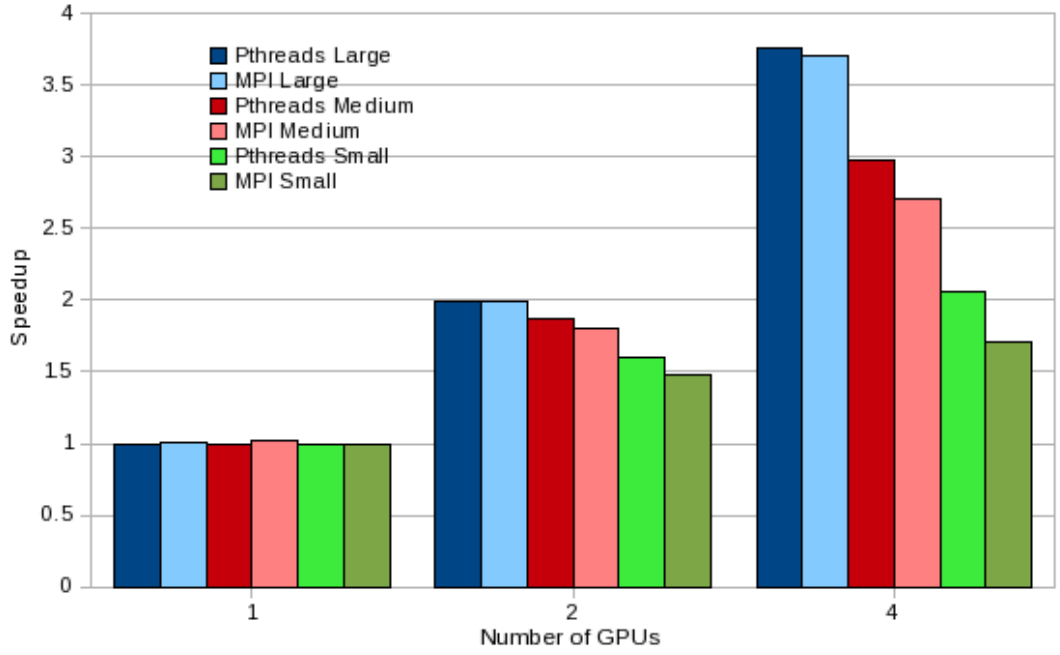


Figure 2.2: Relative performance of the Pthreads and baseline MPI models on a single machine with 1, 2, and 4 GPUs. Domain sizes are $128 \times 32 \times 128$ ("Small"), $256 \times 32 \times 256$ ("Medium"), and $512 \times 32 \times 1024$ ("Large"). For each domain size, all results are relative to the single GPU Pthreads implementation, which is assigned a value of 1.0.

the received boundary cells to the GPU for use in the next computational step.

The MPI (Message Passing Interface) API is widely used for programming clusters, and works on both shared and distributed memory machines. In general, it will have some performance loss compared to the shared-memory model used by POSIX, but in return it offers a highly portable solution to writing programs to work on a wide variety of machines and hardware topologies.

To investigate the efficiency differences of the coarse-level parallelism model in a single node, a simple baseline implementation was chosen and a lid-driven cavity simulation was performed. The expectation is to measure the difference in performance between a Pthreads-CUDA and an MPI-CUDA application running on a single shared

memory machine. Figure 2.2 shows the results, which indicate little to no efficiency loss using the MPI message passing API as the coarse-level parallelism, as long as the problem size is large enough per GPU (roughly 1 million cells per GPU in this example). More details about this test are described in the following paragraphs. It appears that an MPI-CUDA implementation is versatile enough to operate on desktop machines without performance loss for most problems. Since the same application performs well on single-GPU workstations, multi-GPU workstations, and GPU clusters, this is proposed as the preferred method to use for HPC applications.

The test involved a baseline MPI implementation of the application simulating a lid-driven cavity running on a single machine attached to an NVIDIA Tesla S870 unit with 4 GPUs (each of the two pairs of GPUs share a single PCI Express $\times 16$ slot). Three domain sizes are used: a small domain ($128 \times 32 \times 128$), a medium domain ($256 \times 32 \times 256$), and a large domain ($512 \times 32 \times 1024$). The computational problem is large only in the context of a single low-memory GPU. The Pthreads implementation is that of Thibault and Senocak [105], while the MPI implementation is a baseline version that does not overlap computation with exchanges and does not include many of the performance improvements later added. This is done to keep the comparison between the coarse-parallelism models.

The Pthreads implementation accomplishes the neighbor cell transfers with a device to host copy of all cells to transfer, a barrier, then a host to device copy of the cells to update. The MPI implementation attempts to overlap MPI communication and data transfer by posting non-blocking receives early and does non-blocking sends as soon as each piece of data is on the host. Neither implementation does any overlapping of computation with other work, such as host/device data transfer or message passing. Computational overlapping is investigated in more detail in Section 4.2.3, and is

critical for good performance on a cluster.

The scaling of both implementations on a single machine is limited by the data transfer as the problem size is reduced. The memory copies using the PCI Express bus are limited to 2-5 GB/s on the hardware utilized. As expected, the extra work due to the MPI communications causes a performance loss on this single shared-memory machine for smaller problems. With larger problem sizes, the asynchronous calls succeed in hiding more of the MPI communication, and therefore the Pthreads and MPI implementation for multiple GPUs show little difference.

A number of studies have shown the benefits of combining MPI with a threading model when writing programs to operate on clusters of multi-core machines [22, 23], including many specific to scientific computing and CFD applications [14, 28, 53, 71, 93]. Since the GPU version of the CFD application requires the boundary cells to be copied from the GPU to the host and back again, the shared-memory model requires only a synchronization between neighbors as the memory is immediately accessible to the neighbor once on the host. In contrast, the message passing model requires the same work plus a message passed, which may involve sending the data through network interfaces, which can be quite expensive. Therefore, a hybrid approach takes advantage of low-overhead shared memory when possible and message passing when necessary.

A tri-level MPI-OpenMP-CUDA implementation is investigated in Section 4.2.5. MPI is used for communication between nodes, while OpenMP maximizes efficiency within each node, and CUDA is used for data parallelism on each GPU. Implementation details are discussed, and performance results are shown. The main question to be answered is whether the additional complication is worth the performance benefits in this type of application running on a GPU cluster.

CHAPTER 3

TECHNICAL BACKGROUND

3.1 Governing Equations

The Navier-Stokes equations for buoyancy-driven incompressible fluid flows can be written in the following compact form:

$$\nabla \cdot \mathbf{u} = 0, \tag{3.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{1}{\rho}\nabla P + \nu \nabla^2 \mathbf{u} + \mathbf{f}, \tag{3.2}$$

where $\mathbf{u}$ is the velocity vector, P is the pressure, ρ is the density, ν is the kinematic viscosity, and $\mathbf{f}$ is the body force. The Boussinesq approximation, which applies to incompressible flows with small temperature variations, is used to model the buoyancy effects in the momentum equations [68]:

$$\mathbf{f} = \mathbf{g} \cdot (1 - \beta(T - T_\infty)), \tag{3.3}$$

where $\mathbf{g}$ is the gravity vector, β is the thermal expansion coefficient, T is the calculated temperature at the location, and T_∞ is the steady state temperature.

The temperature equation can be written as [44, 102]

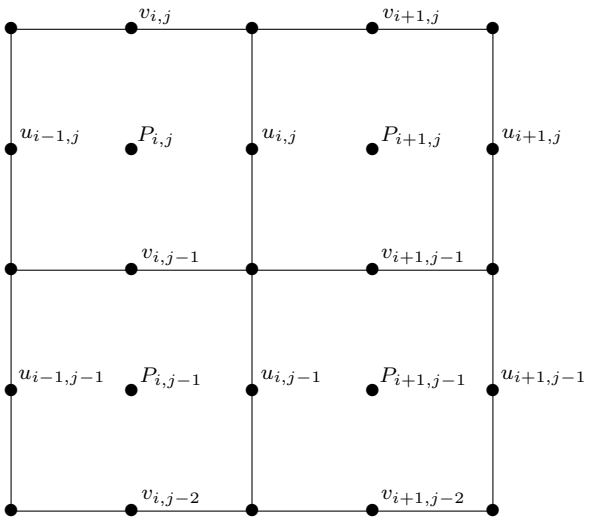


Figure 3.1: Staggered grid arrangement (Arakawa type C grid [1]), showing placement of momentum and pressure spatial variables. Pressure P is located in the cell center, while momentum components u and v are located in the edge midpoints. Temperature T is cell centered similar to pressure.

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{u}T) = \alpha \nabla^2 T + \Phi, \tag{3.4}$$

where α is the thermal diffusivity and Φ is the heat source.

3.2 Numerical Approach

3.2.1 Projection Algorithm

The incompressible Navier-Stokes equations (Eqs. 3.1-3.2) do not have explicit separate equations for pressure or momentum. A variety of methods have been proposed for splitting the solution into fractional steps where the momentum and pressure are independently solved. These include the projection algorithm of Chorin [24], Patankar's SIMPLE scheme [84, 85] and its variants, and others. Many of these fractional-step methods are reviewed and contrasted in the survey of Guermond et al. [48].

A second-order accurate central difference scheme is used to discretize the advection and diffusion terms of the Navier-Stokes equations on a uniform staggered grid [34] as shown in Figure 3.1. Various time-stepping methods are implemented, including the first-order accurate explicit Euler scheme and the second-order accurate Adams-Bashforth scheme. The projection algorithm [24] is then adopted to find a numerical solution to the Navier-Stokes equation for incompressible fluid flows.

In the projection algorithm, the velocity field $\mathbf{u}^*$ is predicted using the momentum equations without the pressure gradient term [24, 34]. With a first-order accurate Euler time-stepping method, the velocity field is predicted as

$$\mathbf{u}^* = \mathbf{u}^t + \Delta t(-\mathbf{u}^t \nabla \cdot \mathbf{u}^t + \nu \nabla^2 \mathbf{u}^t + \mathbf{f}), \tag{3.5}$$

where the index t and Δt represent the time level and time-step size, respectively. $\mathbf{u}^t \nabla \cdot \mathbf{u}^t$ is the advective term, $\nu \nabla^2 \mathbf{u}^t$ is the diffusive term, and $\mathbf{f}$ is the calculated buoyancy effect. For the second-order Adams-Bashforth time-stepping method, the velocity field is predicted as

$$\mathbf{u}^* = \mathbf{u}^t + \Delta t\Big(\frac{3}{2}(-\mathbf{u}^t \cdot \nabla \mathbf{u}^t + \nu \nabla^2 \mathbf{u}^t + \mathbf{f}) - \frac{1}{2}(-\mathbf{u}^{t-1} \cdot \nabla \mathbf{u}^{t-1} + \nu \nabla^2 \mathbf{u}^{t-1} + \mathbf{f})\Big). \tag{3.6}$$

The predicted velocity field $\mathbf{u}^*$ does not satisfy the divergence free condition because the pressure gradient term is not included in Eq. 3.5. By enforcing the divergence free condition on the velocity field at time $t + 1$, the following pressure Poisson equation can be derived from the momentum equations given in Eq. 3.2

$$\nabla^2 P^{t+1} = \frac{\rho}{\Delta t} \nabla \cdot \mathbf{u}^*. \tag{3.7}$$

In our implementation, Eq. 3.7 is solved with either an iterative solver (Jacobi or Red-Black Gauss-Seidel) or a geometric multigrid method. The pressure field at time $t+1$ is then used to correct the predicted velocity field $\mathbf{u}^*$ as follows

$$\mathbf{u}^{t+1} = \mathbf{u}^* - \frac{\Delta t}{\rho} \nabla P^{t+1}. \tag{3.8}$$

The temperature equation is implemented using the second-order accurate central difference method to compute the advection and diffusion terms in Eq. 3.4 at each time step. Buoyancy effects are are added to the momentum term following Eq. 3.3.

Most thermo-fluid applications involve turbulence. Modeling of turbulence is beyond the scope of this thesis study. However, for future applications, the original Smagorinsky large-eddy simulation model was implemented, where the turbulent eddy viscosity can be calculated per cell prior to calculating the momentum at each time step. The turbulence model was not validated, and further discussion is in Appendix B. Results and validation for models shown in this thesis were done with essentially laminar flow that does not require turbulence modeling.

It is noted that the projection algorithm [24] used is only first-order in time. Guermond et al. [48] review alternate projection algorithms, including the second-order rotational form of Brown et al. [19] and the fractional-step method of Kim and Moin [64]. Implementing the higher-order algorithms requires special care with boundary conditions, especially with non-Dirichlet conditions. The method used should not have an impact on the conclusions of the applicability of the projection method to GPU clusters, and it was decided not to pursue this further.

3.2.2 Time-Stepping Methods

The momentum, turbulence, and temperature operations allow coupled operation of multi-stage time-stepping routines such as Runge-Kutta methods as well as multi-step methods such as Euler and Adams-Bashforth. The multi-step methods use a lower-order version to initialize, hence second-order Adams-Bashforth uses forward Euler on the first time step. A Runge-Kutta method could alternately be applied.

Higher-order Runge-Kutta methods have many advantages when solving systems of PDEs [34]. Their stability ranges are quite large, and typically the increase in allowable time-step size outweighs the extra work performed in the multiple stages, which leads to an overall reduction in time taken to reach a given simulation time. A systematic study of time-stepping routines is not pursued, and discussion of the methods implemented is included in Appendix A. The second-order Adams-Bashforth method is used for all results in this thesis.

By default, static time stepping is used, where an appropriate δt is calculated from the input parameters and used throughout the computation. This calculation is as follows

$$\Delta t_{\text{convective}} = (\min\Delta_{xyz} \cdot \text{CFL}) \ / \ \text{velmax}, \tag{3.9}$$

$$\Delta t_{\text{viscous}} = (\min\Delta_{xyz} \cdot \min\Delta_{xyz}) \ / \ \nu, \tag{3.10}$$

$$\Delta t_{\text{temperature}} = (0.5 \cdot \text{Pr} \cdot \text{Re}) \ / \ (\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}), \tag{3.11}$$

$$\Delta t = \tau \cdot \min(\Delta t_{\text{convective}}, \Delta t_{\text{viscous}}, \Delta t_{\text{temperature}}), \tag{3.12}$$

where $\min\Delta_{xyz} = \min(\Delta x, \Delta y, \Delta z)$, CFL is the Courant–Friedrichs–Lewy number

specified in the configuration file, velmax is the inlet velocity magnitude, ν is the kinematic viscosity, Pr is the Prandtl number (the ratio of kinematic viscosity to thermal diffusivity), Re is the Reynolds number, and τ is a limiting factor between 0 and 1. The temperature limit for time stepping is identical to that shown in Griebel et al. [44]

Adaptive time stepping is also supported, where the the maximum velocity is computed across the entire domain, and used in Eq. 3.9 if it is higher than the inlet velocity. This practice enables stable computations for configurations where the maximum velocity can fluctuate as the simulation progresses, which typically is the case in many complex geometry configurations.

3.2.3 Pressure Poisson Solver

The pressure Poisson Eq. 3.7 is the most time consuming step in incompressible flow solvers, hence using efficient methods is worth pursuing. It may be solved by a variety of methods suitable for linear second-order PDEs. The initial solver used for the single-node flow solver [104] was a Jacobi iterative solver, which runs rapidly on the GPU and reduces the initial error quickly. This is adequate for many simple problems where an approximate solution is all that is needed. However, convergence to machine-precision levels is very slow when more precise results are desired. Convergence is also dependent on the grid size, which is a concern for cluster implementations targeting very large problems.

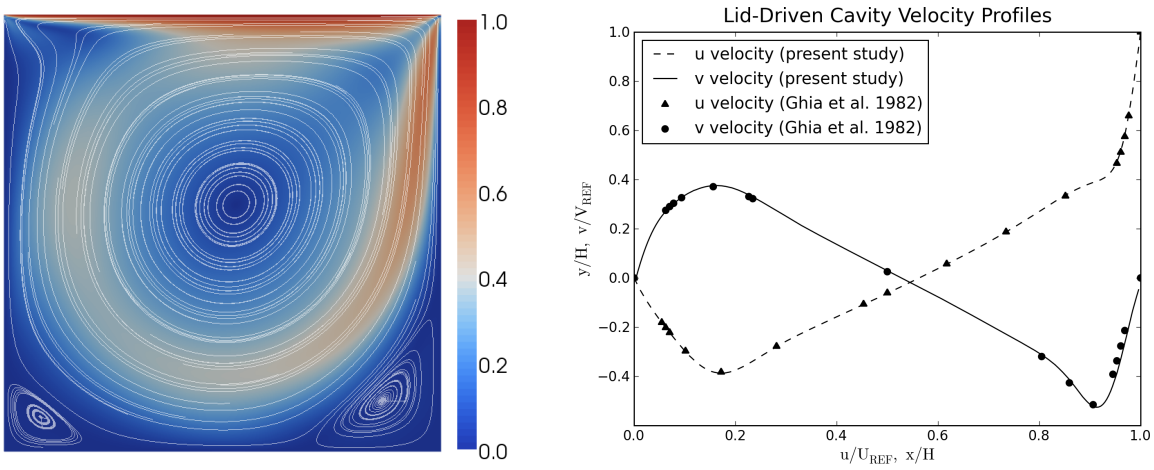
Another choice for an iterative solver is the method of Gauss-Seidel, and the weighted version known as Successive Overrelaxation (SOR). Data parallel versions of this solver using the standard lexicographic order are complicated due to data dependencies, resulting in frequent communications and limited parallelism. By using

Method	Operations (3D)	Operations (2D)	Storage (3D)
Dense LU (Gaussian Elimination)	$O(N^3)$	$O(N^3)$	$O(N^2)$
Band LU	$O(N^{7/3})$	$O(N^2)$	$O(N^{5/3})$
Jacobi iteration	$O(N^{5/3})$	$O(N^2)$	$O(N)$
Gauss-Seidel	$O(N^{5/3})$	$O(N^2)$	$O(N)$
Successive Overrelaxation (SOR)	$O(N^{4/3})$	$O(N^{3/2})$	$O(N)$
Conjugate Gradient	$O(N^{4/3})$	$O(N^{3/2})$	$O(N)$
Fast Fourier Transform	$O(N \log N)$	$O(N \log N)$	$O(N)$
Multigrid	$O(N)$	$O(N)$	$O(N)$

Table 3.1: Run-time and memory complexity of different solvers for the discrete Poisson equation, with convergence to a fixed small error with iterative methods. $N = n^3$ for a $n \times n \times n$ 3D grid, $N = n^2$ for a $n \times n$ 2D grid. From Demmel [27].

the Red-Black ordering for the evaluation order, the algorithm is readily parallelizable. In the Red-Black ordering, the grid points are colored alternately and data parallel operations are done on each color successively. This method is very common for parallel implementations, and is easily implemented in CUDA using one kernel invocation per color.

As indicated in Table 3.1, the computational complexity of both the Jacobi and Gauss-Seidel methods is $O(N^2)$ in 2D, though with an optimal choice of weighting the SOR method is $O(N^{3/2})$. Another solver choice is the method of conjugate gradients, but it is not pursued in this study. Instead, the geometric multigrid method [107], which is computationally $O(N)$, was implemented. This is one of the most efficient methods when more precise results are desired. One advantage the multigrid method has over the methods considered is grid-size independent convergence rate. Assuming other factors held constant, as the domain size grows the convergence rate does not decrease as it does with the other methods. The geometric multigrid method and its implementation on GPU clusters is examined in detail in Chapter 5.



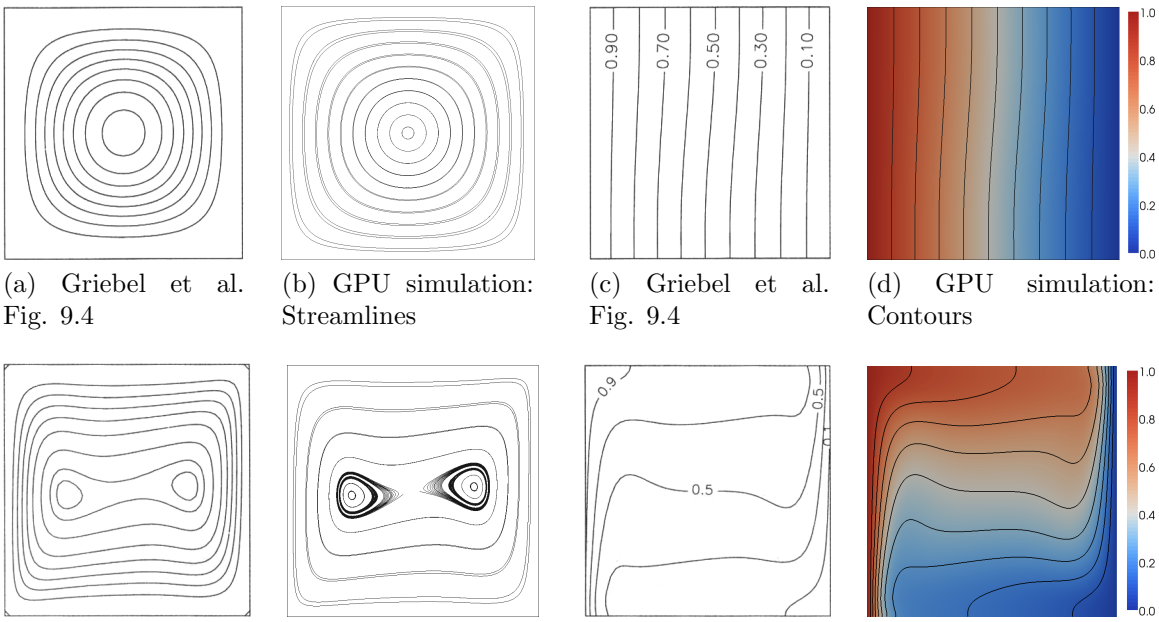
(a) (b)

Figure 3.2: Lid-driven cavity simulation with Re = 1000 on a $256 \times 32 \times 256$ grid. 3D computations were used and a 2D center slice is shown. (a) Velocity streamlines and velocity magnitude distribution. (b) Comparison to the benchmark data from Ghia et al. [37].

3.3 Validation

A test suite has been created and the various implementations have been validated using the tests. The suite includes the well-known lid-driven cavity and natural convection in heated cavity problems [37, 108]. Figure 3.2 presents the results of a lid-driven cavity simulation with a Reynolds number 1000 on a $256 \times 32 \times 256$ grid. Fig. 3.2a shows the velocity magnitude distribution and streamlines at mid-plane. As expected, the computations capture the two corner vortices at steady-state. In Fig. 3.2b, the horizontal and vertical components of the velocity along the centerlines are compared to the benchmark data of Ghia et al. [37]. The results agree well with the benchmark data.

Natural convection in a cavity with heated lateral walls is simulated to test the buoyancy-driven incompressible flow computations on a $128 \times 16 \times 128$ grid. A Prandtl



(a) Griebel et al. Fig. 9.4

(b) GPU simulation: Streamlines

(c) Griebel et al. Fig. 9.4

(d) GPU simulation: Contours

(e) Griebel et al. Figure 9.5

(f) GPU simulation: Streamlines

(g) Griebel et al. Figure 9.5

(h) GPU simulation: Contours

Figure 3.3: Natural convection in a cavity using a $128 \times 16 \times 128$ grid and Prandtl number 7, with a 2D center slice shown. a-d) Streamlines and temperature isotherms for $Pr = 7$ and $Ra = 140$. e-f) Streamlines and temperature isotherms for $Pr = 7$ and $Ra = 200,000$. The simulation uses parameters shown in Griebel et al. [44, Section 9.7.1], and results can be seen to closely match.

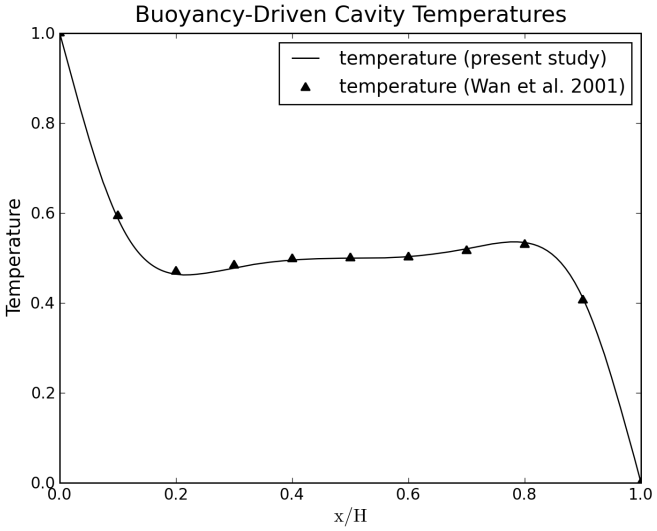


Figure 3.4: Centerline temperature for natural convection in a cavity with Prandtl number 7 and Rayleigh number 100,000, using a $256 \times 16 \times 256$ grid with a 2D center slice used. Comparison is shown to data from Wan et al. [108].

number of 7 was used for each heated cavity simulation. Figure 3.3 presents the natural convection patterns and isotherms for Rayleigh numbers of 140 and 200,000. Lateral walls have constant temperature boundary conditions with one of the walls having a higher temperature than the wall on the opposite side. Top and bottom walls are insulated. Fluid inside the cavity is heated on the hot lateral wall and rises due to buoyancy effects, whereas on the cold wall it cools down and sinks, creating a circular convection pattern inside the cavity. The results agree with those presented in Griebel et al. [44]. Figure 3.4 presents a comparison of the horizontal centerline temperatures for a heated cavity with $\mathrm{Ra} = 100,000$ along with reference data from Wan et al. [108]. These results are also in good agreement.

Rayleigh-Bénard convection is a cell-like arrangement of rising and descending fluid (Bénard cells) that occurs when fluid is heated from the bottom. It is a well studied phenomenon and is often used for testing temperature-driven flow in

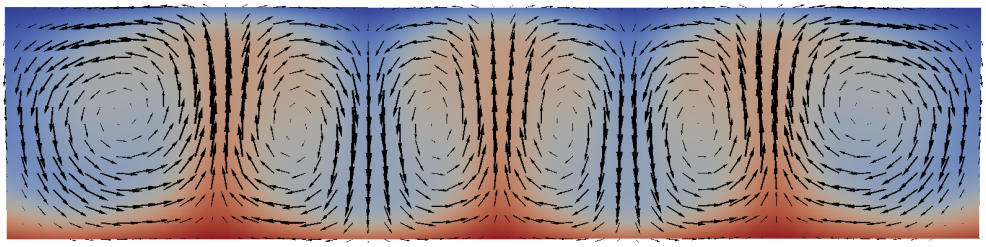


Figure 3.5: Natural convection in a 3D cavity with aspect ratio 8:1:2, following the example of Griebel et al. [44, Section 11.4.4]. Parameters are set to $Pr = 0.72$, $Ra = 30000$, and $Re = 4365$. The mesh used was $65 \times 9 \times 17$ with a 2D center slice shown.

simulations. Figure 3.5 shows a simulation using the example of Griebel et al. [44, Section 11.4.4]. This simulates air in a three-dimensional cavity with an aspect ratio of 8:1:2. The dimensionless parameters are $Pr = 0.72$, $Ra = 30000$, and $Re = 4365$ in each case. Six cells are formed, as expected.

3.4 Flow in Urban Environments and Complex Terrain

Modeling airflow in urban environments is an important application that has seen much recent work [2, 30, 33, 35, 50, 91, 97]. The present CFD model includes the ability to read solid rectangular obstacles into the initial conditions, allowing for solution of more complex problems, such as the simulation of air flow through an urban environment. The urban geometry is typically available in the form of a Geographic Information Systems (GIS) database. Figure 3.6 shows the velocity field around buildings from the Oklahoma City downtown area. The flow field is visualized with streamlines at time step 4000. Figure 3.7 shows air flow around complex terrain, where data was imported from a USGS Digital Elevation Model. These simulations are preliminary, as complex terrain physics has a number of additional factors that are not addressed in the present modeling effort.

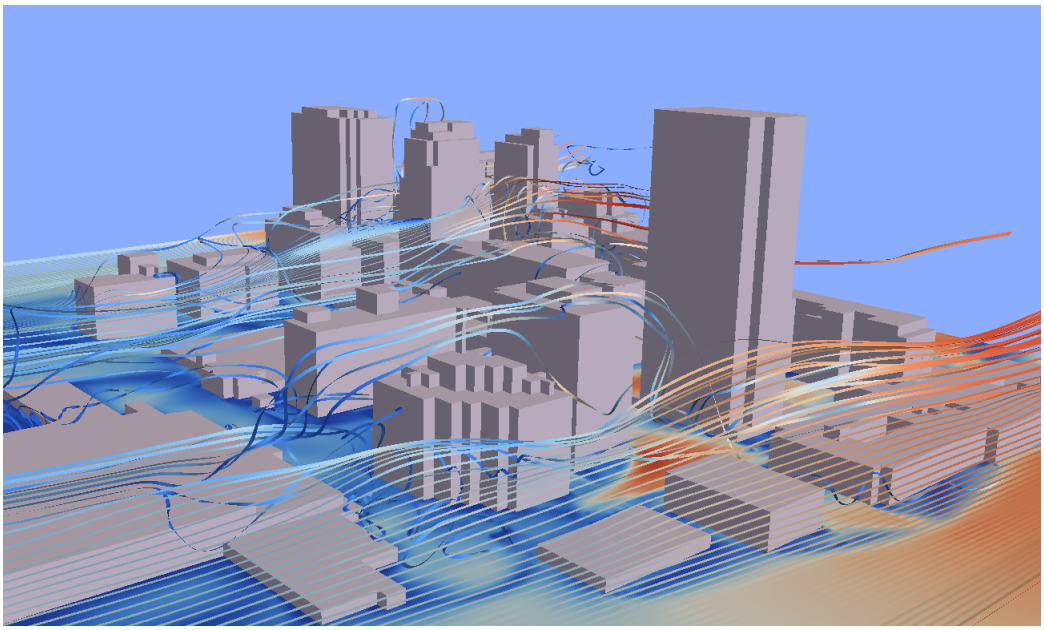


Figure 3.6: A view of the simulation showing air flow in the Oklahoma City downtown area at time step 4000.

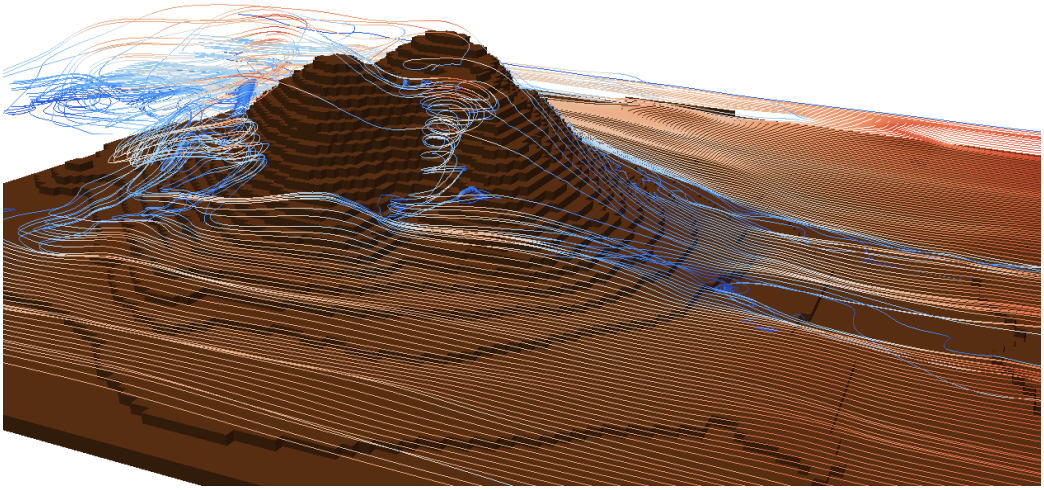


Figure 3.7: A view of instantaneous airflow around complex terrain.

CHAPTER 4

MULTILEVEL PARALLELISM ON GPU CLUSTERS

4.1 Introduction

Graphics processing units used for general purpose computing enable high levels of fine-grain data parallelism. However, many tasks in the scientific community require more resources than a single device can provide, whether in the problem size or the required performance. Adding more GPUs to a workstation can be useful but is limited in scalability. At this time, it is rare to see more than 4 GPUs on a single workstation. Emerging GPU clusters allow large numbers of GPUs to be used to solve single problems. Examples include the 7168 GPU Tianhe-1A cluster, the 4640 GPU Dawning Nebulae cluster, the 680 GPU TSUBAME 1.2 cluster, the 512 GPU Longhorn cluster, the 284 GPU NCSA Lincoln Tesla cluster, as well as many planned petascale GPU platforms. These new systems are designed for high performance as well as high power efficiency, which is a crucial factor in future exascale computing [100].

Shared memory models, such as POSIX Threads and OpenMP, are well suited for single-node multi-GPU machines; however, taking advantage of GPU clusters in an efficient manner requires a message passing API such as MPI or a higher level language. While research is ongoing in cluster-aware parallel languages [10, 31, 36, 80], the majority of applications use explicit commands for coarse-grain parallelism. The

present work concentrates solely on investigating using combinations of CUDA for the GPU, Pthreads or OpenMP for intra-node parallelism, and MPI for inter-node parallelism.

Brandvik and Pullan [17] show the implementation of 2D and 3D Euler solver on a single GPU, showing 29× speedup for the 2D solver and 16× speedup for the 3D solver. One unique feature of their paper is the implementation of the solvers in both BrookGPU and CUDA. Elsen et al. [29] show the conversion of a subset of an existing Navier-Stokes solver to arrive at a BrookGPU version of a 3D compressible Euler solver on a single GPU. Significant effort went into efficiently handling the non-uniform mesh as well as geometric multigrid on an irregular mesh. Measured speedups of 15× to 40× on complex geometries were obtained from a NVIDIA 8800GTX compared to a single core of a 2.4GHz Intel Core 2 Duo E6600. Tölke and Krafczyk [106] describe a 3D Lattice Boltzmann model (D3Q13) in detail along with a CUDA implementation. Their single GPU implementation on an NVIDIA 8800 Ultra achieves a speedup of 100× over an Intel Xeon (noting that the CPU calculation was done in double precision and with an implementation of a more detailed model, making the speedup value not directly comparable). Simek et al. [99] detail performance gains on a variety of single GPU platforms for atmospheric dispersion simulations, achieving speedups as high as 77× compared to a CPU implementation. Cohen and Molemaker [26] describe the implementation and validation of an incompressible Navier-Stokes solver with Boussinesq approximation that supports double precision. The numerical approach is very similar to that used in this thesis. Since it is a single GPU implementation, domain decomposition and overlapping methods are not examined. Performance of their implementation on a Tesla C1060 was 8× faster than comparable multi-threaded code running on an 8-core Intel Xeon E5420 at 2.5GHz.

Double precision was 46% to 66% slower than single precision, which is in line with results we have seen with our model running in double precision.

Modern motherboards can accommodate multiple GPUs in a single workstation with several TeraFLOPS of peak performance. Currently, GPU programming models, such as CUDA and OpenCL, do not address parallel implementations for multi-GPU platforms. Therefore, GPU programming models have to be interleaved with MPI, OpenMP, or Pthreads. In the multi-GPU computing front, Thibault and Senocak [105] developed a single-node multi-GPU 3D incompressible Navier-Stokes solver with a Pthreads-CUDA implementation. The GPU kernels from their study forms the internals of the present cluster implementation. Thibault and Senocak demonstrated a speedup of 21× for two Tesla C870 GPUs compared to a single core of an Intel Core 2 3.0 GHz processor, 53× for two GPUs compared to an AMD Opteron 2.4 GHz processor, and 100× for four GPUs compared to the same AMD Opteron processor. Four GPUs were able to sustain 3× speedup compared to a single GPU on a large problem size. The multi-GPU implementation of Thibault and Senocak does not overlap computation with GPU data exchanges. Overlapping features are introduced in the present study.

Micikevicius [75] describes both single- and multi-GPU CUDA implementations of a 3D 8^{th}-order finite difference wave equation computation. The wave equation code is composed of a single kernel with one stencil operation, unlike CFD computations, which consist of multiple inter-related kernels. MPI was used for process communication in multi-GPU computing. Micikevicius uses a two stage computation where the cells to be exchanged are computed first, then the inner cells are computed in parallel with asynchronous memory copy operations and MPI exchanges. With efficient overlapping of computations and copy operations, Micikevicius achieves superlinear

speedup on 4 GPUs running on two Infiniband connected nodes with two Tesla 10-series GPUs each, when using a large enough dataset.

Before the introduction of CUDA in 2007, several early studies demonstrated the potential of GPU clusters to tackle computationally large problems. Fan et al. [30] investigated GPU cluster use for scientific computation in their 2004 paper. Fan et al. showed an urban dispersion simulation implemented as a Lattice Boltzmann model (LBM) run on a GPU cluster. Their cluster consisted of 32 nodes each with a single GPU and uses MPI for inter-node communication. The paper emphasizes the importance of minimizing communication costs, and the authors give excellent views of overlapping communication and computation time in their model. Göddeke et al. [40] in 2007 surveyed cluster approaches for scientific computation and shows how GPU clusters from commodity components can be a practical solution. Phillips et al. [88] in 2008 also investigated GPU clusters, detailing many of the complications that arise that are unique to the system. Schive et al. [94] presented a 16 node, 32 GPU cluster running a parallel direct N-body simulation system, achieving 7.1 TeraFLOPS and over 90% parallel efficiency. Performance on GPU clusters running well-parallelizable algorithms such as N-body simulations are better than those reported for fluid dynamics solvers which are implemented with several kernels, as noted in the more recent article by Schive et al. [95] that describes an Euler solver using adaptive mesh refinement. Clearly, the sustained performance on GPU computing platforms depends on the application.

Göddeke et al. [41] explored coarse- and fine-grain parallelism in a finite element model for fluids or solid mechanics computations on a GPU cluster. Göddeke et al. [39] described the application of their approach to a large-scale solver toolkit. The Navier-Stokes simulations in particular exhibited limited performance due to

memory bandwidth and latency issues. Optimizations were also found to be more complicated than simpler models such as the ones they previously considered. While the small cluster speedup of a single kernel is good, unfortunately acceleration of the entire model is only a modest factor of two. Their model uses a non-uniform grid and multigrid solvers within a finite element framework for relatively low Reynolds numbers.

Phillips et al. [88] describe many of the challenges that arise when implementing scientific computations on a GPU cluster, including the host/device memory traffic and overlapping execution with computation. A performance visualization tool was used to verify overlapping of CPU, GPU, and communication on an Infiniband connected 64 GPU cluster. Scalability is noticeably worse for the GPU accelerated application than the CPU application, as the impact of the GPU acceleration is quickly dominated by the communication time. However, the speedup is still notable. Phillips et al. [86] describe a 2D Euler Equation solver running on an 8-node cluster with 32 GPUs. The decomposition is 1D, but GPU kernels are used to gather/scatter from linear memory to non-contiguous memory on the device.

While MPI is the API typically used for network communication between compute-nodes, it presents a distributed memory model, which makes it less efficient for processes running on the same shared-memory compute-node [9, 66]. For this reason, hybrid programming models combining MPI and a threading model such as OpenMP or Pthreads have been proposed with the premise that message passing overhead can be reduced, increasing scalability. With two to four GPUs per compute-node, this hybrid method warrants further investigation.

Cappello, Richard, and Etiemble [21, 22, 23] were among the first to present the hybrid programming model of using MPI in conjunction with a threading model such

as OpenMP. They demonstrated that it is sometimes possible to increase efficiency on some code by using a mixture of shared memory and message passing models. A number of other papers followed with the same conclusions [14, 28, 53, 71, 77, 89, 92, 93]. Many of these papers also point out a number of cases where the applications or computing systems are a poor fit to the hybrid model, and in some cases performance decreases. Lusk and Chan [72] describes using OpenMP and MPI for hybrid programming on three cluster environments, including the effect the different models have on communication with the NAS benchmarks. They believe this combination of programming models is well fitted to modern scalable high performance systems.

Hager, Jost, and Rabenseifner [49] give a recent perspective on the state of the art techniques in hybrid MPI-OpenMP programming. Particular attention is given to mapping the model to domain decomposition as well as overlapping methods. Results with hybrid models of the BT-MZ benchmark (part of the NAS Parallel Benchmark suite) on a Cray XT5 using a hybrid approach showed similar performance at 64 and fewer cores, but greatly improved results for 128, 256, and 512 cores, where a good combination of OpenMP fine-grain parallelism combined with MPI coarse-grain parallelism can be found that matches well with the hardware. These examples also take advantage of the loop scheduling features in OpenMP. Advantages in fine-grain parallelism like this will not be able to be taken advantage of in a model where OpenMP is only used for coarse-grain data transfer and synchronization.

Balaji et al. [4] discuss issues arising from using MPI on petascale machines with close to a million processors. A number of MPI collective operations are shown to have exponential time with respect to the number of processors. The tested MPI implementations also allocate some memory proportional to the number of processes, limiting scalability. These, as well as other limitations, lead the authors to suggest a

hybrid threading / MPI model as one way to mitigate the issue. However, in the case of a typical GPU system, the situation is not as bad. In this case, the CUDA model for fine-grain parallelism manages 256 to 512 processing elements within a single process, and this number will likely increase with future GPUs. Hence, a one million processing element GPU cluster using just MPI-CUDA may have fewer than 4000 MPI processes. This indicates that clusters enhanced with GPUs look well suited for petascale and emerging exascale architectures. On the other hand, it also indicates that the hybrid model has less potential benefit on multi-GPU clusters.

Nakajima [76] describes a three-level hybrid method using MPI, OpenMP, and vectorization. This approach uses MPI for inter-node communication, OpenMP for intra-node communication, and parallelism within the node via the vector processor. It closely matches the rationale behind the present approach for the multi-GPU cluster implementation. Weak scaling measurements showed worse results for 64 and fewer SMP nodes, but improved with 96 or more. GPU clusters with 128 or more compute-nodes (256 or more GPUs) are rare at this time but trends indicate these machines will become far more common in the high performance computing field [100].

While these articles show some potential benefits for using the hybrid model on CPU clusters, a question is whether the same benefits will accrue to a tri-level CUDA-OpenMP-MPI model, and whether they will outweigh the added software complexity. With high levels of data parallelism on the GPU, separate memory for each GPU, low device counts per node, and currently small node counts, the GPU cluster model has numerous differences from dense-core CPU clusters. The present thesis aims to address some of these questions.

4.2 Multilevel Parallel Implementations

4.2.1 Common Implementation Details

A large portion of the flow solver is identical across all parallel implementations, and on a single GPU system they run identically. There are a number of steps taken before the model is started, including:

- Reading and parsing the configuration file.
- Reading and parsing an obstacle file if specified.
- Adjusting configuration parameters based on obstacle file, if needed.
- Determining process / thread / GPU mapping, and doing domain decomposition.
- Selecting GPU context and ensuring required resources will be available.
- Printing configuration text to screen and log.
- Allocating host memory.
- Optionally, initializing host memory with an initial condition.
- Spawning parallel processes / threads, each responsible for a single GPU.

The projection algorithm described in Section 3.2.1 is implemented here, with pseudo-code shown in Figure 4.1. At each time step there is also optional progress status output, and VTK visualization output. When the time-step loop ends, the process writes the final output and clears GPU memory, then returns to the common code, which can exit.

```
for (t=0; t < time_steps; t++)
{
    adjust_timestep();  // Adaptive timestepping

    for (stage = 0; stage < num_timestep_stages; stage++) {
        temperature <<<grid,block>>> (u,v,w,phiold,phi,phinew);
        ROTATE_POINTERS(phi,phinew);
        temperature_bc <<<grid,block>>> (phi);
        EXCHANGE(phi);

        turbulence <<<grid,block>>> (u,v,w,nu);
        turbulence_bc <<<grid,block>>> (nu);
        EXCHANGE(nu);

        momentum <<<grid,block>>> (phi,uold,u,unew,vold,v,vnew,wold,w,wnew);
        momentum_bc <<<grid,block>>> (unew,vnew,wnew);
        EXCHANGE(unew,vnew,wnew);
    }

    divergence <<<grid,block>>>(unew,vnew,wnew,div);

    // Iterative or multigrid solution
    pressure_solve(div,p,pnew);

    correction <<<grid,block>>> (unew,vnew,wnew,p);
    momentum_bc <<<grid,block>>> (unew,vnew,wnew);
    EXCHANGE(unew,vnew,wnew);

    ROTATE_POINTERS(u,unew);  ROTATE_POINTERS(v,vnew);  ROTATE_POINTERS(w,wnew);
}
```

Figure 4.1: Host code for the projection algorithm to solve buoyancy-driven incompressible flow equations on multi-GPU platforms. The outer loop is used for time stepping, and indicates where the time-step size can be adjusted. The `EXCHANGE` step updates the ghost cells for each GPU with the contents of the data from the neighboring GPU.

CUDA Kernels

All implementations use similar CUDA kernels. Since most kernels perform stencil calls, a common framework is used. Figure 4.2 shows a template for a kernel. A number of thread sizes were tried, and Table 4.1 shows the best results over a variety of simulations. These are selected at compile time for all kernels, although it is possible these could be chosen independently per kernel or even modified at run-time.

	CC 1.3	CC 2.0
Single Precision	16×16	32×16
Double Precision	16×8	32×8

Table 4.1: Thread block sizes used for compute capability 1.3 and compute capability 2.0 devices.

Boundary conditions are set via kernels, which also use a common template. The number of threads is chosen to be $threads = threads_x \times threads_y$, where $threads_x = \max(nx, ny)$ and $threads_y = \max(ny, nz)$. In the kernel, this results in each of the six faces having a thread per face cell. Each thread is then responsible for up to six faces. When compared with a typical method of spawning one thread per domain cell with only the threads on a face performing work, there are many fewer threads spawned and a resulting increase in performance on large domains.

Tables 4.2 and 4.3 list the main kernels, their performance on an example problem, and the percent of the total time taken in that operation. The percent includes the boundary-condition kernel. The temperature and turbulence kernels are used in this example, but are typically not used for performance measurements elsewhere in the thesis. The momentum kernel is very dense, as each thread performs advection and diffusion on all three momentum vectors of a cell. This results in high register use,

```
__global__ void kernel(
    int sections,
    const REAL* d_u,    const REAL* d_v,    const REAL* d_w,
    REAL* ...)
{
   unsigned int xpos = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned int ypos = blockIdx.y*blockDim.y + threadIdx.y;
   unsigned int I =   DCONSTANT_PADNY*DCONSTANT_PADNX
                    + ypos*DCONSTANT_PADNX + xpos;

   if (    (xpos == 0) || (xpos >= (DCONSTANT_NX-1))
        || (ypos == 0) || (ypos >= (DCONSTANT_NY-1))
        || (sections == 0)
      )  return;

   unsigned int kbeg = (sections & SECTION_BOT)  ?  0        :     1;
   unsigned int kend = (sections & SECTION_TOP)  ?  nlayers  :  nlayers-1;
   bool do_mid = (sections & SECTION_MID);
   unsigned int k = kbeg;

   while (k < kend) {
      unsigned int Ik = I + k*DCONSTANT_PADNX*DCONSTANT_PADNY;

      // Do operations here using Ik as the index for the center pixel

      if ( (!do_mid) && (k == kbeg) ) {
         k = nlayers-1;
      } else {
         k++;
      }
   }
}
```

Figure 4.2: Template for CUDA kernel performing stencil operation. A bit mask is handed in, indicating which sections (top, bottom, middle) should be computed. The borders will never be computed in this kernel. Note the use of constant memory, which can alternately be done as variables passed into the kernel.

	registers	GFLOPS	percent time
temperature	33	24.8	6%
turbulence	28	38.2	17%
momentum	51	56.9	22%
divergence	14	26.6	2%
correction	27	29.0	3%
pressure (Jacobi)	18	40.7	51%

Table 4.2: Kernels with 30 Jacobi iterations for the pressure Poisson solver. Example problem is $384 \times 384 \times 384$ on a single Tesla S1070 GPU. Using the compute capability 1.3 thread-block size from Table 4.1, 32 registers are available with full occupancy. Measurements indicated that for the two kernels using more than that number of registers, performance decreased if the compiler was told to limit usage to 32.

	registers	GFLOPS	percent time
temperature	33	21.1	4%
turbulence	28	40.9	9%
momentum	51	55.5	13%
divergence	14	23.6	1%
correction	27	22.8	2%
smoother (Jacobi)	17	30.0	52%
prolongation	19	10.3	4%
laplacian	18	15.0	6%
restriction	22	30.2	7%

Table 4.3: Kernels with 4 multigrid V-cycles for the pressure Poisson solver. Example problem is $385 \times 385 \times 385$ on a single Tesla S1070 GPU.

which might indicate scheduling issues, but because of the number of operations performed, the GFLOPS is higher than other kernels. As expected, the pressure Poisson solver takes the majority of the time, and it is mostly bandwidth bound.

Other Features

The implementation supports double-precision computation as a compile time option. Current CUDA compute capability 1.3 and higher devices support double-precision, but at a substantially reduced throughput (8 to 12×), unless one uses the Tesla-model Fermi cards (e.g. C2050), where it is half the speed of single-precision. While this would indicate a potential performance decrease of 8× on most cards, in practice we

see 2× to 2.6×. This is in line with reports from other developers [5, 26, 38] for CFD code, which is at least partially memory bandwidth bound.

All configuration in the current implementations use a run-time ASCII text configuration file for most decisions, with only the single-precision / double-precision and the MPI overlapping methods remaining as compile-time parameters. Output formats including the full model VTK output and periodic output are controlled by the configuration, as well as intermediate data output. An example configuration file is shown in Appendix C.

Multilevel Parallelism

All the features of the model described are applicable to a single GPU, and results from Section 4.2.4 indicate a 13× performance increase over Pthreads-parallelized CPU code using eight 2.33GHz Intel64 cores. Further increases can be achieved by running on multiple GPUs, and even more when run on a cluster. Moreover, the domain size can be substantially larger than that allowed by a single GPU.

Multiple programming APIs along with a domain decomposition strategy for data-parallelism is required to achieve high throughput and scalable results from a CFD model on a multi-GPU platform. For problems that are small enough to run on a single GPU, overhead time is minimized as no GPU/host communication is performed during the computation, and all optimizations are done within the GPU code. When more than one GPU is used, cells at the edges of each GPU's computational space must be communicated to the GPUs that share the domain boundary so they have the current data necessary for their computations. Data transfers across the neighboring GPUs inject additional latency into the implementation, which can restrict scalability if not properly handled. Therefore, we investigate several implementations of mul-

tilevel parallelism to improve the performance and scalability of the Navier-Stokes solver.

4.2.2 Dual-Level Pthreads-CUDA Implementation

The work by Thibault and Senocak [105] showed how an incompressible Navier-Stokes solver written for a single GPU can be extended to multiple GPUs by using Pthreads. The implementation and results are described in detail in Julien Thibault's thesis [104]. The full 3D domain is decomposed across threads in one dimension, splitting on the Z axis. The resulting partitions are then solved using one GPU per thread, with CUDA used for fine-grain parallelism. No effort in this model is made to hide latencies arising from GPU data transfers or Pthreads synchronization.

Scalability results were presented in those papers, with results obtained on an AMD system with eight dual core CPUs and an attached NVIDIA S870 server holding four GPUs, using two PCIe $\times 16$ adapters. In the implementation used for performance testing, no overlapping or asynchronous calls were performed. Ghost cell exchanges were done using `cudaMemcpy()` to copy the edge layers from each GPU, a barrier to synchronize the GPUs, followed by `cudaMemcpy()` to copy the appropriate edges to each GPU. Note that since the first copy puts the data in shared memory, each thread has access to the edge memory needed without any need for message passing. Performance results with a relatively large $1024 \times 32 \times 1024$ problem show $1.6\times$ speedup of two GPUs $3.0\times$ speedup of four GPUs, relative to a single GPU. Since the number of GPUs is increased while the problem size remains constant, this is known as *strong scaling*. The parallel efficiency is 80% and 75% at 2 and 4 GPUs, respectively.

These results show the promise of using multiple GPUs for computational fluid dynamic simulations. However, since this method uses a shared-memory model, running it on a network-connected cluster requires a distributed shared-memory system such as Intel Cluster OpenMP [55]. For models with large amounts of data to transfer, such systems are currently very inefficient compared to message passing APIs such as MPI. Without using a network-efficient API, the pure threading models are not appropriate for use on a cluster, and hence are limited in scale.

4.2.3 Dual-Level MPI-CUDA Implementation

To solve the restrictions of the shared-memory model, MPI was adopted as the mechanism for communication between GPUs. A single process is used per GPU, with all ghost cell exchanges being done via `MPI_Isend` and `MPI_Irecv`. As discussed in Section 2.2, with a large enough domain, MPI seems able to closely match the performance of a shared-memory Pthreads-CUDA implementation on a single node. Testing of the final MPI-CUDA implementation on identical hardware and simulation parameters as used in Thibault and Senocak [105] show better scalability, even with no overlapping of GPU memory copies. Therefore, with problem sizes sufficient to use multiple GPUs, it is possible to achieve very efficient MPI solutions compared to Pthreads.

Domain Decomposition

A 3D Cartesian volume is decomposed into 1D layers. These layers are then partitioned among the GPUs on the cluster to form a 1D domain decomposition. The 1D decomposition is shown in Figure 4.3. After each GPU completes its computation,

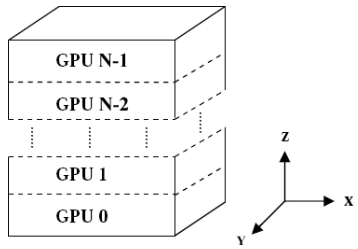


Figure 4.3: The decomposition of the full domain to the individual GPUs.

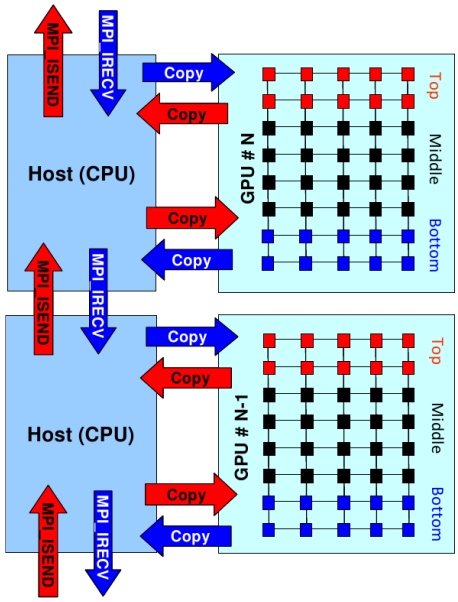


Figure 4.4: An overview of the MPI communication, GPU memory transfers, and the intra-GPU 1D decomposition used for overlapping.

the edge cells ("ghost cells") must be exchanged with neighboring GPUs. Efficiently performing this exchange process is crucial to cluster scalability.

While a 1D decomposition leads to more data being transferred as the number of GPUs increases, there are advantages to the method when using CUDA. In parallel CPU implementations, host memory access can be performed on non-contiguous segments with a relatively small performance loss. The `MPI_CART` routines supplied by MPI allow efficient management of virtual topologies, making the use of 2D and 3D decompositions easy and efficient. In contrast, the CUDA API only provides a way to transfer linear segments of memory between the host and the GPU. Hence, 2D or 3D decompositions for GPU implementations must either use non-standard device memory layouts, which may result in poor GPU performance, or run separate kernels to perform gather/scatter operations into a linear buffer suitable for the `cudaMemcpy()` routine. These routines add significant time and hinder overlapping methods. For this reason, the 1D decomposition was deemed best for moderate size clusters such as the ones used in this study. This is briefly investigated in Section 4.2.4.

To accommodate overlapping, a further 1D decomposition is applied within each GPU. Figure 4.4 indicates how the 1D layers within each GPU are split into a top, bottom, and middle section. When overlapping communication and computation, the GPU executes each separately such that the memory transfers and MPI communication can happen simultaneously with the computation of the middle portion.

Implementations

Three implementations of the MPI-CUDA incompressible flow solver were developed. A single MPI process is started per GPU, and each process is responsible for managing its GPU and exchanging data with its neighbor processes. Overlap of computations

```
// PART 1:  Interleave non-blocking MPI calls with device
//          to host memory transfers of the edge layers.

// Communication to south
MPI_Irecv(new ghost layer from north)
cudaMemcpy(south edge layer from device to host)
MPI_Isend(south edge layer to south)
// Communication to north
MPI_Irecv(new ghost layer from south)
cudaMemcpy(north edge layer from device to host)
MPI_Isend(north edge layer to north)

// ... other exchanges may be started here, before finishing in order

// PART 2:  Once MPI indicates the ghost layers have been received,
//          perform the host to device memory transfers.

MPI_Wait(new ghost layer from north)
cudaMemcpy(new north ghost layer from host to device)
MPI_Wait(new ghost layer from south)
cudaMemcpy(new south ghost layer from host to device)
MPI_Waitall(south and north sends, allowing buffers to be reused)
```

Figure 4.5: An `EXCHANGE` operation overlaps GPU memory copy operations with asynchronous MPI calls for communication.

with inter-node and intra-node data exchanges is accomplished to better utilize the cluster resources. All three of the implementations have much in common, with differences in the way data exchanges are implemented. It is shown in Section 4.2.4 that implementation details in the data exchanges have a large impact on performance.

The projection algorithm is composed of distinct steps in the solution of the fluid flow equations. Figure 4.1 shows an outline of the basic implementation using CUDA kernels to perform each step. The steps marked as `EXCHANGE` are where ghost cells for each GPU are filled in with the calculated contents of their neighboring GPUs. The most basic exchange method is to call `cudaMemcpy()` to copy the edge data to host memory, MPI exchange using `MPI_Send` and `MPI_Recv`, and finally another `cudaMemcpy()` to copy the received edge data to device memory. This is

```
// The GPU domain is decomposed into three sections:
//    (1) top edge, (2) bottom edge, and (3) middle
// Which of them the kernel should process is indicated
// by a flag given as an argument.

pressure <<<grid_edge,block>>> (edge_flags, div,p,pnew);

// The cudaMemcpy calls below will not start until
// the previous kernels have completed.
// This is identical to part 1 of the EXCHANGE operation.

// Communication to south
MPI_Irecv(new ghost layer from north)
cudaMemcpy(south edge layer from device to host)
MPI_Isend(south edge layer to south)
// Communication to north
MPI_Irecv(new ghost layer from south)
cudaMemcpy(north edge layer from device to host)
MPI_Isend(north edge layer to north);

pressure <<<grid_middle,block>>> (middle_flag, div,p,pnew);

// This is identical to part 2 of the EXCHANGE operation.
MPI_Wait(new ghost layer from north)
cudaMemcpy(new north ghost layer from host to device)
MPI_Wait(new ghost layer from south)
cudaMemcpy(new south ghost layer from host to device)
MPI_Waitall(south and north sends, allowing buffers to be reused)

pressure_bc <<<grid,block>>> (pnew);
ROTATE_POINTERS(p,pnew);
```

Figure 4.6: An example Jacobi pressure loop, showing how the CUDA kernel is split to overlap computation with MPI communication.

```
pressure <<<grid_edge,block, stream[0]>>> (edge_flags, div,p,pnew);
// Ensure the edges have finished before starting the copy
cudaThreadSynchronize();

cudaMemcpyAsync(south edge layer from device to host, stream[0])
cudaMemcpyAsync(north edge layer from device to host, stream[1])

pressure <<<grid_middle,block, stream[2]>>> (middle_flag, div,p,pnew);

MPI_Irecv(new ghost layer from north)
cudaStreamSynchronize(stream[0]);
MPI_Isend(south edge layer to south)

MPI_Irecv(new ghost layer from south)
cudaStreamSynchronize(stream[1]);
MPI_Isend(north edge layer to north);

MPI_Wait(south receive to complete)
cudaMemcpyAsync(new south ghost layer from host to device, stream[0])
MPI_Wait(north receive to complete)
cudaMemcpyAsync(new north ghost layer from host to device, stream[1])

// Ensure all streams are done, including copy operations and computation
cudaThreadSynchronize();
pressure_bc <<<grid,block>>> (pnew);
ROTATE_POINTERS(p,pnew);
```

Figure 4.7: CUDA streams are used to fully overlap computation, memory copy operations, and MPI communication in the pressure loop.

straightforward, but all calls are blocking which hinders performance. Therefore, this most basic implementation is not pursued.

Non-Blocking MPI with No Overlapping of Computation

The first implementation uses non-blocking MPI calls [46] to offer a substantial benefit over the blocking approach. This implementation does not overlap computation although it tries to overlap memory copy operations. The basic `EXCHANGE` operation is shown in Figure 4.5. In this approach, none of the device/host memory operations nor any MPI communication happens until the computation of the entire domain has completed. The MPI communication is able to overlap with the CUDA memory operations. When multiple arrays need to be exchanged, such as the three momentum components, the components may be interleaved such that the MPI send and receive for one edge of the first component is in progress while the memory-copy operations for the later component are proceeding. This is done by starting part 1 for each component in succession, then part 2 for each component.

Overlapping Computation with MPI

The second implementation for exchanges aims to overlap the CUDA computation with the CUDA memory copy operations and the MPI communication. We split the CUDA kernels into three calls such that the edges can be done separately from the middle. This has a very large impact on the cluster performance as long as the domain is large enough to give each GPU enough work to do. The body of the pressure kernel loop when using this method is shown in Figure 4.6. Rather than perform the computation on the entire domain before starting the exchange, the kernel is started with just the edges being computed. The first portion of the previously

shown non-blocking MPI `EXCHANGE` operation is then started, which does device to host memory copy operations followed by non-blocking MPI communications. The computation on the middle portion of the domain can start as soon as the edge layers have finished transferring to the host, and operates in parallel with the MPI communication. The last part of the non-blocking MPI `EXCHANGE` operation is also identical and is run immediately after the middle computation is started. While this implementation results in significant overlap, it is possible to improve on it by overlapping the computation of the middle portion with the memory transfer of the edge layers as shown in the final implementation.

Overlapping Computation with MPI Communications and GPU Transfers

The final implementation is enabled by CUDA streams, and uses asynchronous methods to start the computation of the middle portion as soon as possible, thereby overlapping computation, memory operations, and MPI communication. A similar approach is described in Micikevicius [75]. This method has the highest amount over overlapping, and is expected to have the best performance at large scales. The body of the pressure kernel loop when using this method is shown in Figure 4.7.

It is important to note that the computations inside the CUDA kernels need minimal change, and the same kernel can be used for all three implementations. A flag is sent to each kernel to indicate which portions (top, bottom, middle) it is to compute, along with an adjustment of the CUDA grid size so the proper number of GPU threads are created. Since GPU kernels tend to be highly optimized, minimizing additional changes in kernel code is desirable.

Iterative Solvers

Either of two iterative solvers, weighted Jacobi Red-Black Gauss-Seidel, may be chosen at run-time via the configuration file. Both solvers allow optional weights, for damped Jacobi or successive overrelaxation with Gauss-Seidel. Global versions of both kernels are shown in Appendix D. A shared-memory version of the Jacobi solver has also been implemented. While the shared-memory version is faster, it is not an optimal implementation, as there are additional optimizations using `tex1Dfetch` that could be applied.

4.2.4 Dual-Level MPI-CUDA Performance Results with NCSA Lincoln Tesla and TACC Longhorn Clusters

The NCSA Lincoln cluster consists of 192 Dell PowerEdge 1950 III servers connected via InfiniBand SDR (single data rate) [65]. Each compute-node has two quad-core 2.33GHz Intel64 processors and 16GB of host memory. The cluster has 96 NVIDIA Tesla S1070 accelerator units each housing four C1060-equivalent Tesla GPUs. An accelerator unit is shared by two servers via PCI-Express $\times 8$ connections. Hence, a compute-node has access to two GPUs. For this research, performance measurements for 64 of the 192 available compute-nodes in the NCSA Lincoln Tesla cluster are shown, with up to 128 GPUs being utilized. The CUDA 3.0 Toolkit was used for compilation and runtime, gcc 4.2.4 was the compiler used, and OpenMPI 1.3.2 was used for the MPI library.

The TACC Longhorn cluster consists of 240 Dell R610 compute-nodes connected via InfiniBand QDR (quad data rate). Each compute-node has two quad-core 2.53GHz Intel Nehalem processors and 48GB of host memory. The cluster has 128 NVIDIA

QuadroPlex S4 accelerator units each housing four FX5800 GPUs. An accelerator unit is shared by two servers via PCI-Express 2.0 ×16 connections. Performance of the GPU units is similar to the Lincoln cluster, however, the device/host memory bandwidth is more than 2× higher and the cluster interconnect is 4× faster. For this research, performance measurements for 128 of the 240 available compute-nodes in the TACC Longhorn cluster are shown, with up to 256 GPUs being utilized. The CUDA 3.0 Toolkit was used for compilation and runtime, gcc 4.1.2 was the compiler used, and OpenMPI 1.3.3 was used for the MPI library.

Single GPU performance has been studied relative to a single CPU processor in many studies such as those in Section 4.1. Such performance comparisons are adequate for desktop GPU platforms. On a multi-GPU cluster, a fair comparison should be based on all the available CPU resources in the cluster. To partially address this issue, the CPU version of the CFD code is parallelized with Pthreads to use the eight CPU cores available on a single compute-node of the NCSA Lincoln cluster. Identical numerical methods are used in the CPU and GPU code for the tests performed.

A lid-driven cavity problem at a Reynolds number of 1000 was chosen for performance measurements. Measurements were performed for both *strong scaling* where the problem size remains fixed as the number of processing elements increases, and *weak scaling* where the problem size grows in direct proportion to the number of processing elements. Measurements for the CPU application were done using the Pthreads shared-memory parallel implementation using all eight CPU cores on a single compute-node of the NCSA Lincoln cluster. All measurements include the complete time to run the application including setup and initialization, but do not include I/O time for writing out the results. Single precision was used for these

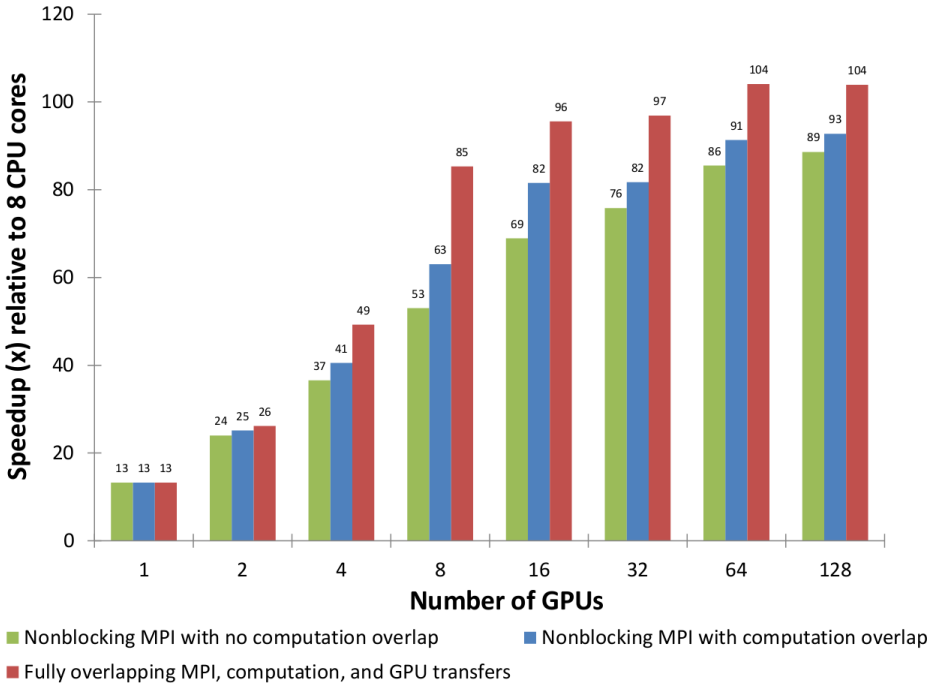


Figure 4.8: Speedup on the NCSA Lincoln Tesla cluster from the three MPI-CUDA implementations relative to the Pthreads parallel CPU code using all 8 cores on a compute-node. The lid-driven cavity problem is solved on a $1024 \times 64 \times 1024$ grid with fixed number of iterations and time steps.

measurements.

Strong Scaling

Figure 4.8 shows the speedup of the MPI-CUDA GPU application relative to the performance of the CPU application using Pthreads. The computational performance on a single compute-node with 2 GPUs was 26 times faster than 8 Intel Xeon cores, and 64 compute-nodes with 128 GPUs performed up to 104 times faster than 8 Intel Xeon cores. In all configurations, the fully overlapped implementation performed faster than the first implementation that did not perform overlapping. Additionally, the final fully overlapping implementation performs fastest in all configurations with

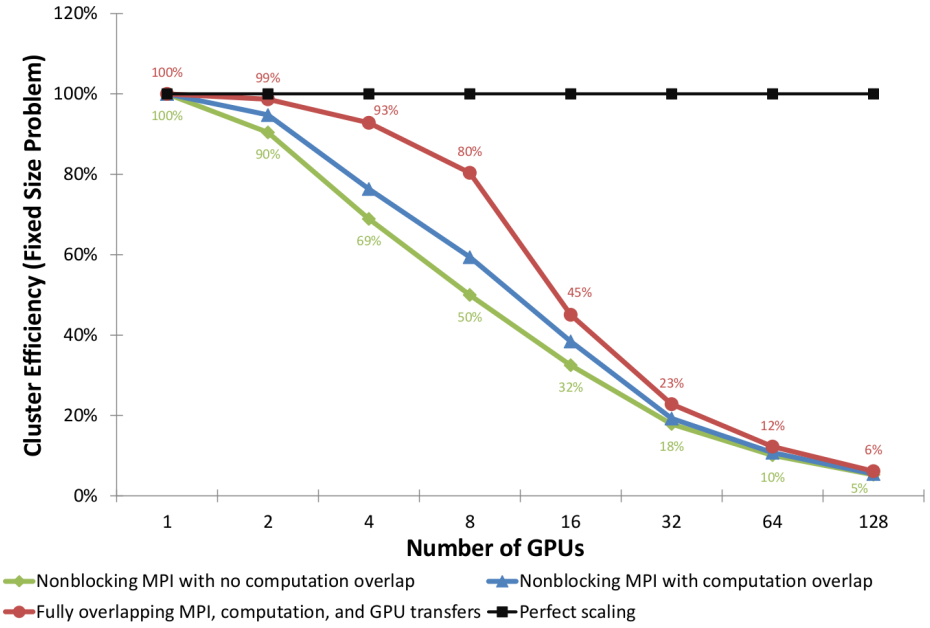


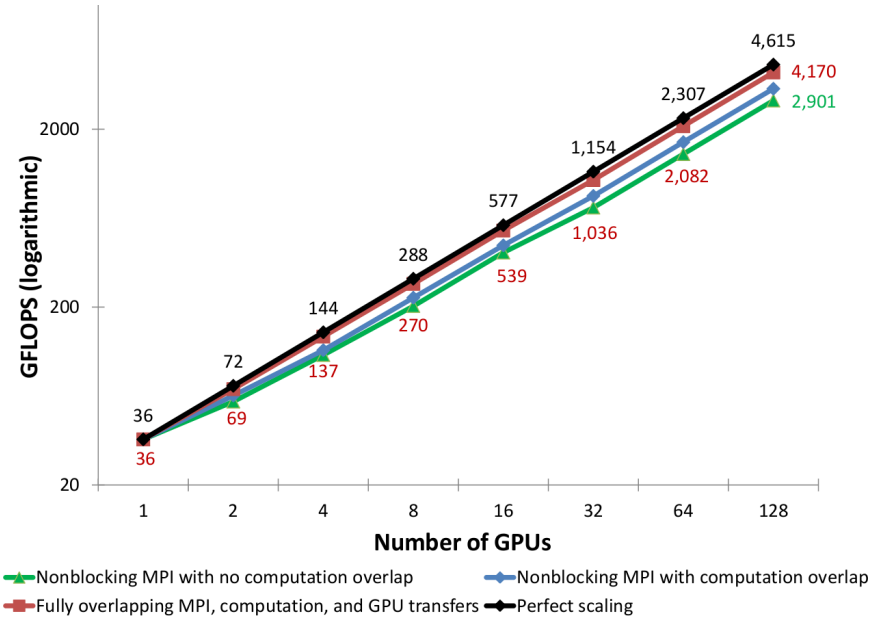
Figure 4.9: Efficiency on the NCSA Lincoln Tesla cluster of the three MPI-CUDA implementations with increasing number of GPUs (strong scalability presentation). The lid-driven cavity problem is solved on a $1024 \times 64 \times 1024$ grid with fixed number of iterations and time steps.

more than one GPU, and shows a significant benefit with more than four GPUs. With the fixed problem size, the amount of work to do on each node quickly drops — on a single GPU a single pressure iteration takes under 10ms of compute time. Little gain is seen beyond 16 GPUs on this fixed-size problem.

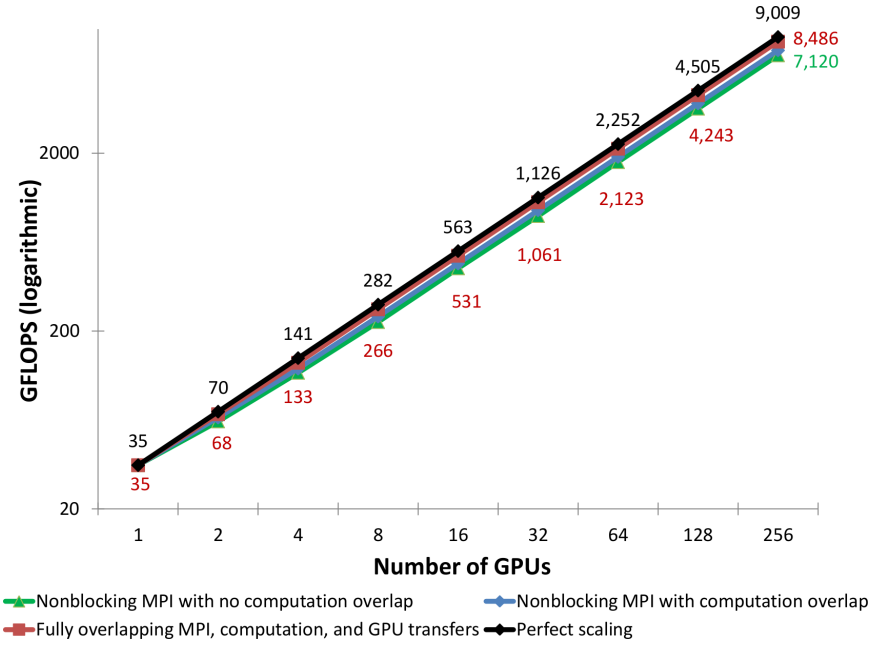
The efficiency numbers for a fixed-size problem in Figure 4.9 indicate the way scaling drops off as the number of nodes is increased. Linear speedup would result in 100% efficiency. Overlapping shows improved efficiency, and the fully overlapped implementation is clearly the best. However, with the fixed-size workload, eventually no amount of overlapping is able to overcome the network overhead once the amount of work per GPU becomes very small. With 128 GPUs, only 30MB of GPU memory is used per GPU for the problem size considered, and the Poisson pressure data is only 2MB. Although as many as 128 GPUs were used for this problem, there appears to be no significant gain in performance beyond 16 GPUs for this fixed-size problem which uses approximately 4GB of total memory.

Weak Scaling in One Dimension

All three MPI-CUDA implementations were also run with increasing problem sizes such that the memory used per GPU was approximately equal. This is commonly referred to as weak scalability. Simulations such as channel flow can lead to extension of the whole domain in one of the three dimensions as the problem size increases. In this case, the height and depth of a channel is fixed, while the width increases relative to the number of GPUs. For the 1D network decomposition performed by this CFD code, the amount of data transferred between each GPU will be constant, as will the domain dimensions on each GPU. Scalability should be excellent.

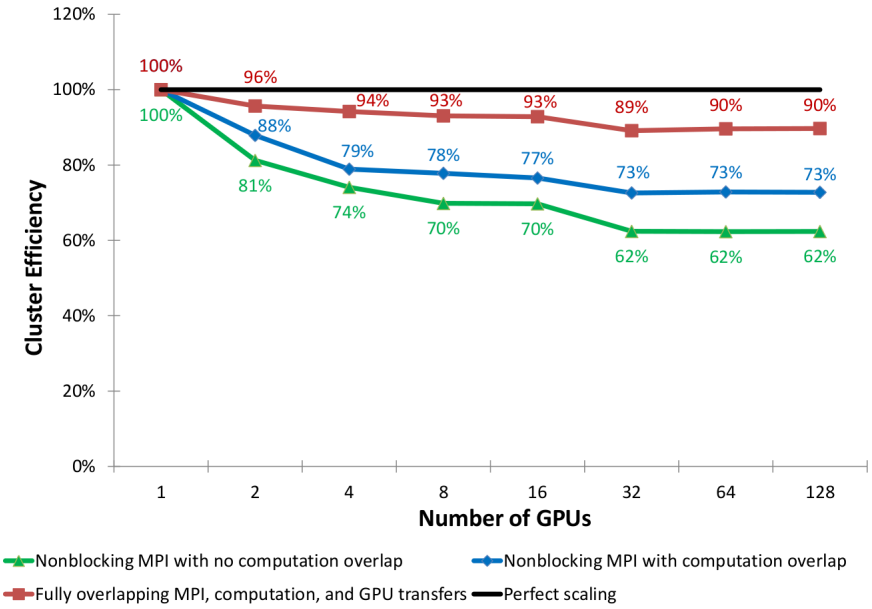


(a) 1D Growth (Lincoln)

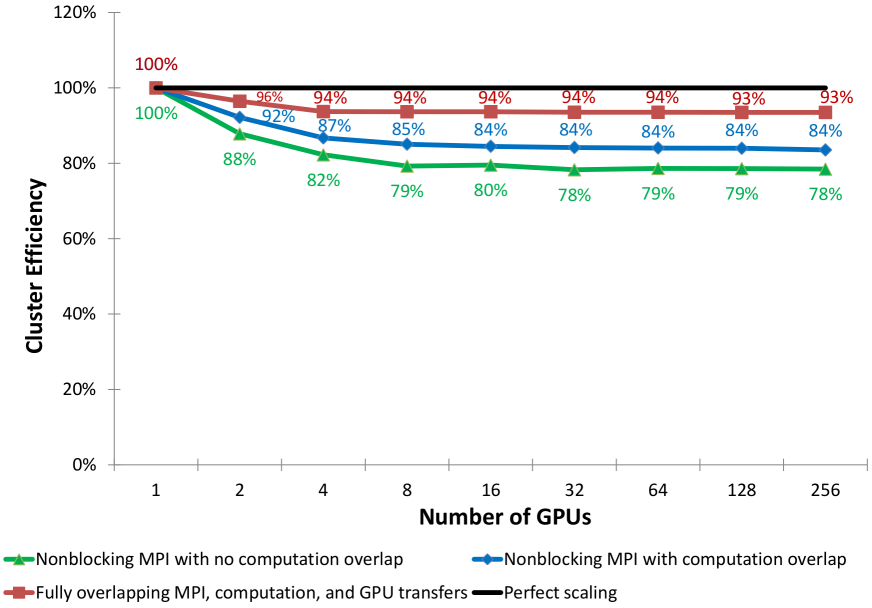


(b) 1D Growth (Longhorn)

Figure 4.10: Sustained cluster performance of the three MPI-CUDA implementations measured in GFLOPS. Growth is in one dimension. The size of the computational grid is varied from $512 \times 512 \times 256$ to $512 \times 512 \times 65536$ with increasing number of GPUs. (a) 4.1 TeraFLOPS with 128 GPUs on the NCSA Lincoln Tesla cluster, (b) 8.4 TeraFLOPS with 256 GPUs on the TACC Longhorn cluster.



(a) 1D Growth (Lincoln)



(b) 1D Growth (Longhorn)

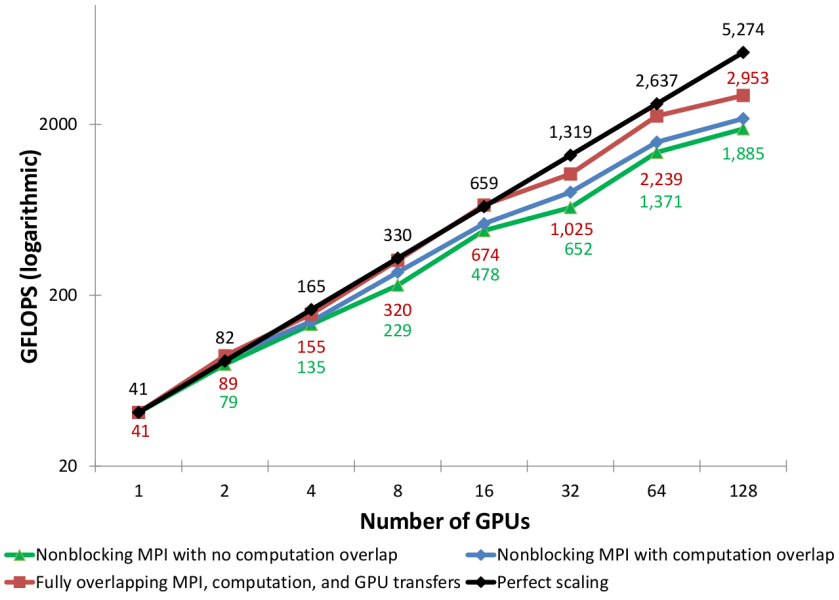
Figure 4.11: Efficiency of the three MPI-CUDA implementations with increasing number of GPUs (weak scalability presentation). Growth is in one dimension. The size of the computational grid is varied from $512 \times 512 \times 256$ to $512 \times 512 \times 65536$ with increasing number of GPUs. (a) NCSA Lincoln Tesla cluster, (b) TACC Longhorn cluster.

Figure 4.10 shows the sustained GFLOPS performance on a logarithmic scale. On the NCSA Lincoln Tesla cluster, 128 GPUs were utilized on 64 compute-nodes to sustain a 4.1 TeraFLOPS performance with the fully overlapped implementation. On the TACC Longhorn cluster, 256 GPUs were utilized on 128 compute-nodes to sustain an 8.4 TeraFLOPS performance. Figure 4.11 indicates how scalability with the fully overlapped implementation is good in this one-dimensional scaling case, dropping from 94% with 4 GPUs to only 90% with 128 GPUs. Note that four GPUs is the first case where the network is utilized. The results from the TACC Longhorn cluster show more consistent behavior, with only a 1% drop in efficiency from 4 GPUs to 256 GPUs.

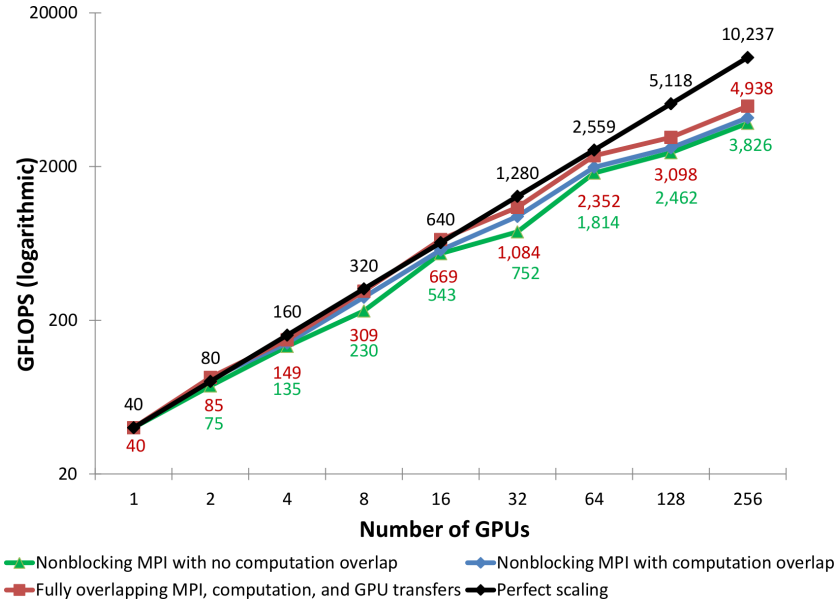
Weak Scaling in Two Dimensions

Again, all three MPI-CUDA implementations were also run with increasing problem sizes such that the memory used per GPU was approximately equal, so weak scalability is examined. In this case, as the number of GPUs is increased, the problem size grows in two dimensions. This is a very common scenario seen in such examples as many lid-driven cavity and buoyancy-driven cases, as well as flow in complex terrain, where covering a larger physical area (e.g. more square blocks in an urban simulation) involves growth in the horizontal dimensions, while the number of cells used for height remains constant.

Figure 4.12 shows the sustained GFLOPS performance on a logarithmic scale. On the NCSA Lincoln Tesla cluster, 128 GPUs were utilized on 64 compute-nodes to sustain a 2.9 TeraFLOPS performance with the fully overlapped implementation. On the TACC Longhorn cluster, 256 GPUs were utilized on 128 compute-nodes to sustain an 4.9 TeraFLOPS performance. With approximately 400GB of memory used

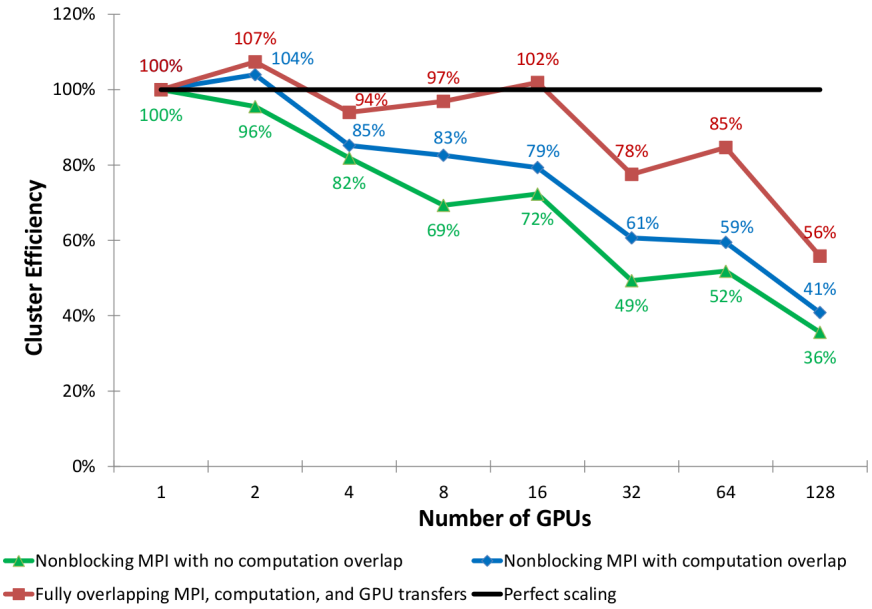


(a) 2D Growth (Lincoln)

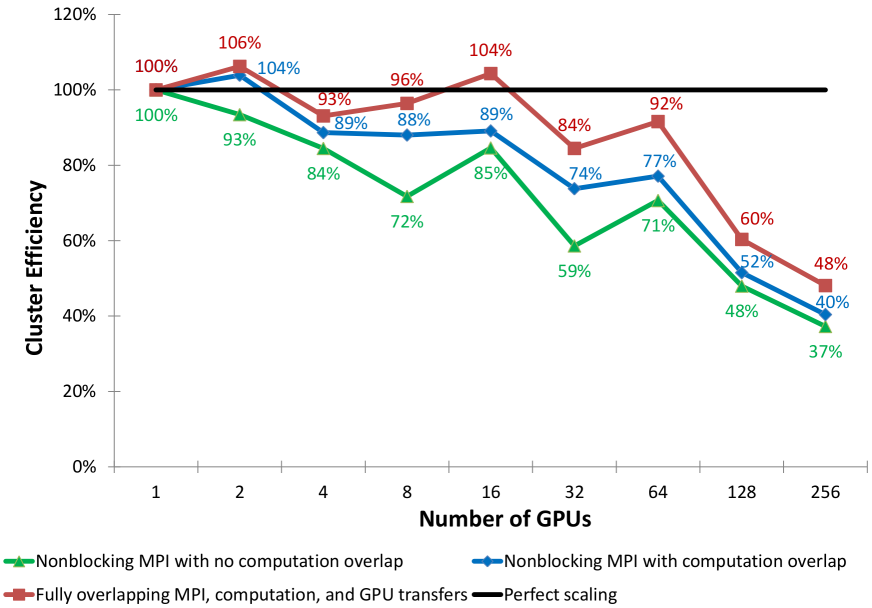


(b) 2D Growth (Longhorn)

Figure 4.12: Sustained cluster performance of the three MPI-CUDA implementations measured in GFLOPS. Growth is in two dimensions, with the Y dimension fixed. The size of the computational grid is varied from $1024 \times 64 \times 1024$ to $16384 \times 64 \times 16384$ with increasing number of GPUs. (a) 2.9 TeraFLOPS with 128 GPUs on the NCSA Lincoln Tesla cluster, (b) 4.9 TeraFLOPS with 256 GPUs on the TACC Longhorn cluster.



(a) 2D Growth (Lincoln)



(b) 2D Growth (Longhorn)

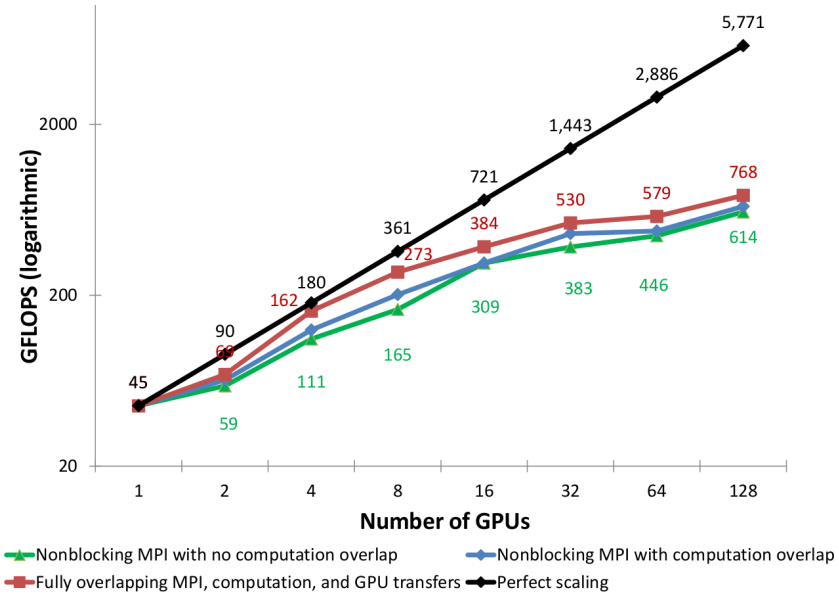
Figure 4.13: Efficiency of the three MPI-CUDA implementations with increasing number of GPUs (weak scalability presentation). Growth is in two dimensions, with the Y size fixed. The size of the computational grid is varied from $1024 \times 64 \times 1024$ to $16384 \times 64 \times 16384$ with increasing number of GPUs. See text for a discussion of the efficiency plot. (a) NCSA Lincoln Tesla cluster, (b) TACC Longhorn cluster.

during the computation on 128 GPUs, it is not possible to directly compare this to a single node CPU implementation on traditional machines. Figure 4.13 shows much improved scaling compared to the fixed problem size case (strong scalability) and the advantage of overlapping computation and communication is also clear. Parallel efficiency in the two-dimensional growth problem with full overlapping is excellent through 16 GPUs, and little additional efficiency is lost with 32 and 64 GPUs.

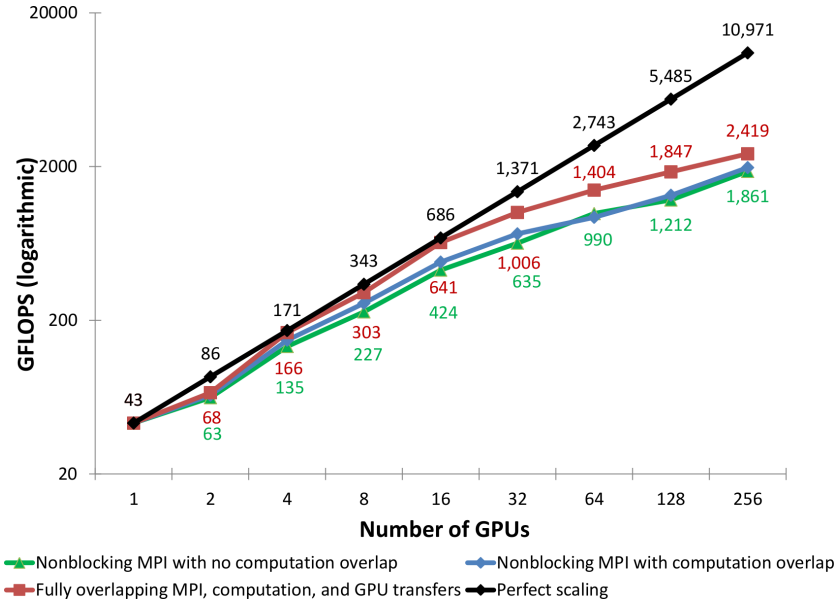
One obvious feature of Figure 4.13 is that efficiency does not fall in a smooth fashion with increasing GPUs, but steps up and down with an overall decreasing trend. This is related to an interaction between the two-dimensional problem size growth and the structure of the CUDA kernels. The mechanism of having each thread loop over all the Z planes is very efficient; however, the CUDA kernel throughput strongly changes as the numbers of threads (the X and Y dimensions) relative to the number of Z planes per GPU is varied. Earlier implementations of the kernels, as seen in Jacobsen et al. [62], show much less variability, but overall performance is lower — for a similar problem the single GPU performance is 33 GFLOPS vs. 41 for the current code, and 2.4 TFLOPS vs. 2.9 TFLOPS with 128 GPUs.

Weak Scaling in Three Dimensions

Figure 4.14 shows the sustained GFLOPS performance on a logarithmic scale. With 128 GPUs, only 768 GFLOPS was sustained with the fully overlapped implementation. On the NCSA Lincoln Tesla cluster, only 768 GFLOPS was sustained with the fully overlapped implementation using 128 GPUs. On the TACC Longhorn cluster, 2.4 TeraFLOPS was sustained using 256 GPUs. Figure 4.15(a) indicates how scalability with the fully overlapped implementation trails off sharply at 16 GPUs, and the gap between the overlapping implementations and non-overlapping narrows.

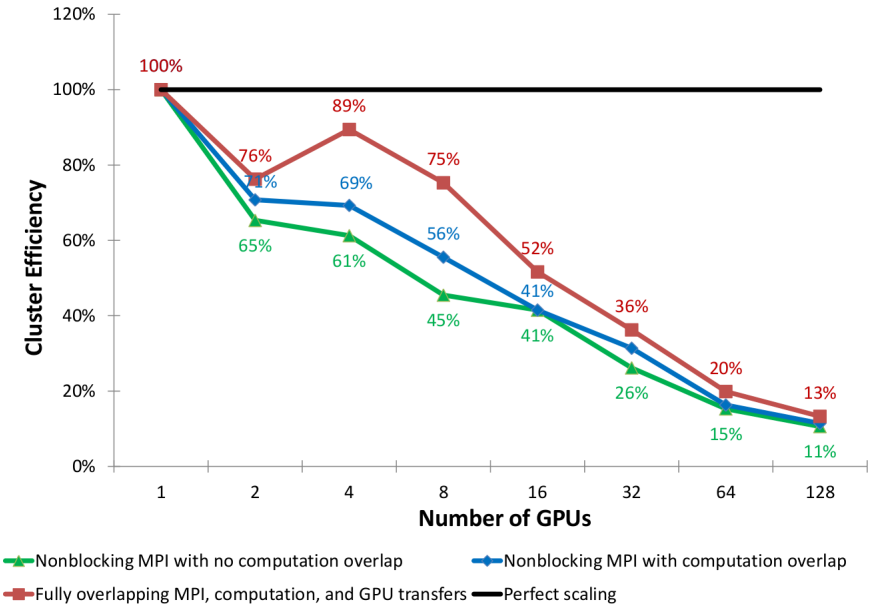


(a) 3D Growth (Lincoln)

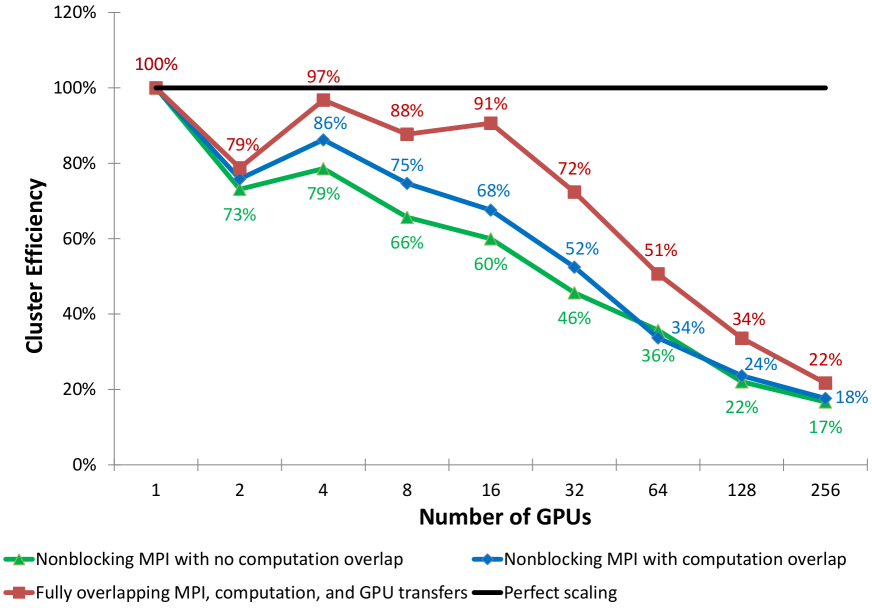


(b) 3D Growth (Longhorn)

Figure 4.14: Sustained cluster performance of the three MPI-CUDA implementations measured in GFLOPS. Growth is in three dimensions. The size of the computational grid is varied from $416 \times 416 \times 416$ to $2688 \times 2688 \times 2560$ with increasing number of GPUs. (a) 0.77 TeraFLOPS with 128 GPUs on the NCSA Lincoln Tesla cluster, (b) 2.4 TeraFLOPS with 256 GPUs on the TACC Longhorn cluster.



(a) 3D Growth (Lincoln)



(b) 3D Growth (Longhorn)

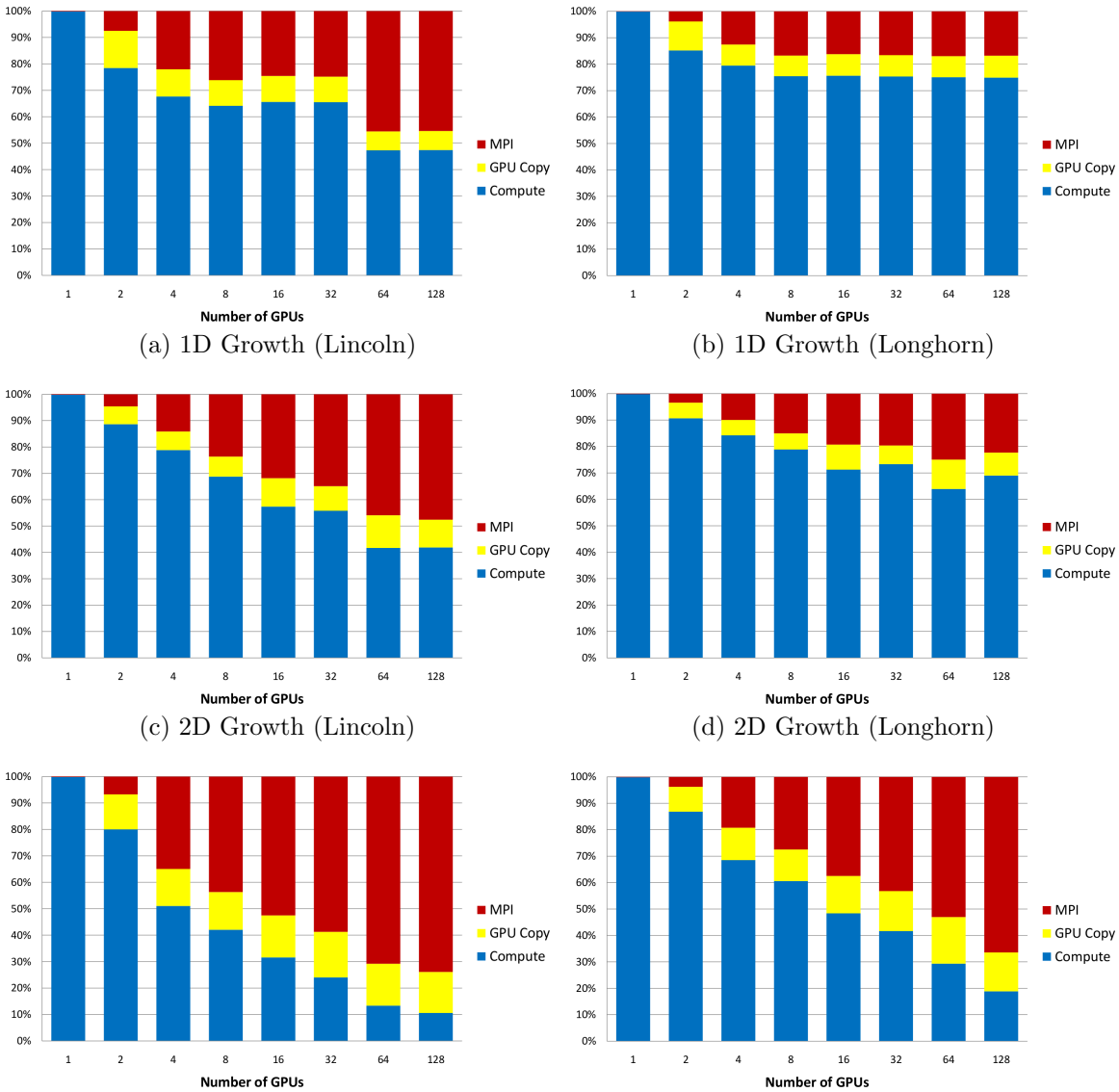
Figure 4.15: Efficiency of the three MPI-CUDA implementations with increasing number of GPUs (weak scalability presentation). Growth is in three dimensions. The size of the computational grid is varied from $416 \times 416 \times 416$ to $2688 \times 2688 \times 2560$ with increasing number of GPUs. (a) NCSA Lincoln Tesla cluster, (b) TACC Longhorn cluster.

The reasons for this behavior are examined in the next section. While the improved communication bandwidths of the TACC Longhorn cluster greatly help scalability (at 128 GPUs, Lincoln is at 13% while Longhorn achieves 34%), the overall behavior is similar.

Further Remarks on Scalability

A design issue with the NCSA Lincoln cluster is noted that has an impact on the results. The connection between the compute-nodes and the Tesla GPUs are through PCI-Express Gen 2 $\times 8$ connections rather than $\times 16$. Measured bandwidth for pinned memory is approximately 1.6 GB/s, which is significantly slower than the 5.6 GB/s measured on a local workstation with PCIe Gen 2 $\times 16$ connections to Tesla C1060s. On a multi-GPU cluster with a faster PCIe bus, performance for all three MPI-CUDA implementations are expected to improve. The fully overlapping method shown in Figure 4.7 would likely see the least benefit as it overlaps all device memory copies while computing. An additional factor that would result in improved efficiency numbers is using a faster network connection, such as Infiniband QDR, rather than the SDR used on the NCSA Lincoln cluster.

Measurements were also performed on the TACC Longhorn cluster which uses GPUs with similar performance (Quadroplex 2200 S4 on Longhorn, Tesla S1070 on Lincoln). However, measured device/host memory transfers are over $2\times$ faster on Longhorn, and its Infiniband QDR shows a $4\times$ increase in interconnect bandwidth with simple benchmarks. It should also be pointed out that as the CUDA kernels are optimized and run faster, less time becomes available for overlapping communications. In the current study, this resulted in changes that improved overall performance, often leading to a loss in parallel efficiency.



(a) 1D Growth (Lincoln)

(b) 1D Growth (Longhorn)

(c) 2D Growth (Lincoln)

(d) 2D Growth (Longhorn)

(e) 3D Growth (Lincoln)

(f) 3D Growth (Longhorn)

Figure 4.16: Percent of pressure Poisson solver (30 Jacobi iterations) time spent in computation, host/GPU memory transfer, and MPI calls. No overlapping is used. The problem size grows such that the number of cells per GPU is approximately constant. (a)-(b) 1D growth, (c)-(d) 2D growth, (e)-(f) 3D growth.

To further examine the reasons behind the scalability results seen, CUDA event timers were used to get high resolution profiles of time spent in the iterative pressure solver. The timers calculated the time spent doing computation in the pressure and boundary condition kernels, the amount of time spent copying data between the GPU and the host, and the time spent in network communications. This data was collected using no overlapping to show the potential benefits of overlapping as well as shed light into the earlier scalability graphs.

For the 1D growth case shown in Figure 4.16a, measured compute and GPU copy time was essentially constant for all runs. This is expected, as the per-GPU dimensions of the pressure domain are identical at each size, and the amount of data to be transferred is constant. The amount of data exchanged by each host also remains constant as the number of GPUs increases, yet the time spent in MPI calls on the Lincoln cluster increases with more GPUs. While performing the solver iterations, each process only synchronizes with its immediate neighbors – no global operations are used. On the Longhorn cluster, the percent of time spent in copy and MPI is essentially constant once the network is utilized at 4 GPUs, which is what is expected.

With the 2D growth case shown in Figure 4.16c, the amount of data to be transferred grows by a factor of $\sqrt{N}$ as the number of GPUs (N) increases. In the 4 GPU case, each transferred layer consists of 2048×64 cells, while with 16 GPUs (a $4\times$ increase) each layer has 4096×64 cells — a $2\times$ increase. With 32 or fewer GPUs, it is possible to completely overlap network traffic and GPU copies with computation. However, the particular size used in this simulation for 32 and 128 GPUs leads to slower computation than other cases, as remarked upon earlier. With 64 and 128 GPUs, complete overlapping of copy, MPI, and computation needs to be done to keep scalability. The data on Longhorn shows a similar pattern, yet scales

better as the communication paths are faster.

The 3D growth case is shown in Figure 4.16e. The amount of data to be transferred grows by a factor of $N^{2/3}$ with the number of GPUs. In the 1 GPU case, each transferred layer consists of 416 × 416 cells, while with 64 GPUs each layer has 1664 × 1664 cells — a 16× increase. Both the GPU copy and MPI communication time increase rapidly, with the GPU copy alone taking more time on 64 GPUs than the entire computation time. The picture on the Longhorn cluster is similar, with the faster data copies just moving the saturation point to more GPUs. While large linear transfers are done to achieve maximum copy efficiency, the amount of data is too large in these cases. Calculation are shown for the 64 GPU case on the Lincoln cluster:

$$\begin{aligned} Copy\ Bandwidth &= (layer\ size \cdot 4 \cdot iterations \cdot timesteps)\,/time \\ &= ((1664 \times 1664 \times 4 \text{ bytes}) \cdot 4 \cdot 30 \cdot 200)\,/139.62 \text{ seconds} \\ &= 1816 \text{ MB/s} \end{aligned} \tag{4.1}$$

$$\begin{aligned} MPI\ Bandwidth &= (layer\ size \cdot 4 \cdot iterations \cdot timesteps)\,/time \\ &= ((1664 \times 1664 \times 4 \text{ bytes}) \cdot 4 \cdot 30 \cdot 200)\ /\ 624.5 \text{ seconds} \\ &= 405.9 \text{ MB/s} \end{aligned} \tag{4.2}$$

For each GPU, the two edge layers must be copied from the GPU and then again to the GPU, hence the factor of 4. This simple calculation ignores the effect of the edge nodes. The effective GPU copy bandwidth is similar to that reported with memory

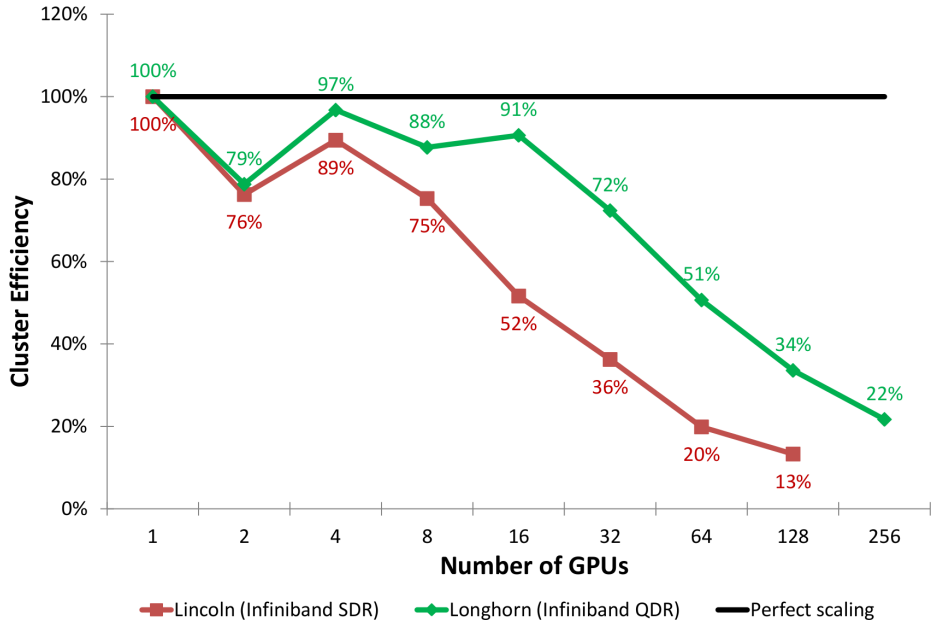


Figure 4.17: A comparison of weak scaling with the fully overlapped MPI-CUDA implementation on two platforms, with growth in three dimensions. Longhorn has higher bandwidth for both GPU/host and network data transfer.

benchmarks on this platform, which is 2 to 3 times less than newer hardware. The effective MPI bandwidth is lower than the bidirectional bandwidth measured with MPI benchmarks, suggesting this as a possible point to investigate.

A 2D decomposition would greatly reduce the amount of data transferred with these large 2D and 3D simulations. Assuming a domain partition in the growth dimensions, the 2D and 3D simulations would see a 4× reduction in the number of bytes transferred. The ramifications to CUDA are discussed in Section 4.2.3. It seems likely that for the 3D problems on many GPUs, the extra CUDA work will be worth the per-GPU cost.

To further point out the effect of data transfer, identical simulations were performed on the Longhorn GPU cluster at the Texas Advanced Computer Center (TACC) at the University of Texas at Austin. This cluster consists of 240 compute-

nodes, each with Intel Nehalem cores and two NVIDIA Quadro FX 5800 GPUs. Infiniband QDR is used to connect the nodes. The performance of single-node CUDA programs is relatively similar between this cluster and the NCSA Lincoln Tesla cluster used in the previous results. However, copy bandwidth was measured using standard benchmarks at over twice that of the NCSA Lincoln cluster, while network exchange rates were almost four times higher. Figure 4.17 directly compares the weak scaling efficiency with growth in three dimensions using a fully overlapped version of the model.

4.2.5 Tri-Level MPI-OpenMP-CUDA Implementation

To investigate whether additional efficiency can be gained from removing redundant message passing when processes are on the same host, a threading model is added. The effectiveness of this solution depends on a number of factors, with some barriers to effectiveness being:

- Density of nodes. With more GPUs per node, the potential effectiveness can be increased. Only clusters with two GPUs per node were available for this study.

- MPI implementation efficiency. The OpenMPI 1.3.2 software on the NCSA Lincoln Tesla cluster seems reasonably well optimized. Goglin [42] discusses optimizations of MPI implementations to improve intra-node efficiency. A number of optimizations have been performed on MPI implementations since the early hybrid model papers were written, including a reduction in the number of copies involved, as well as the extensive optimizations performed in Open-MX. Since the application being studied only using OpenMP and MPI for coarse-grain

parallelism, any benefits in latency for small transactions will not have an impact.

- A large number of nodes. Many of the hybrid model papers note benefits occurring only as the number of nodes grows [21, 49, 76]. While the 64-node 128-GPU implementation used in this study is larger than many published cluster results, it may still be too small to see an appreciable benefit.

- A good match between the hardware, the threading models, and the domain decomposition. A number of hybrid model papers show application / hardware combinations that show reduced performance with the hybrid model [21, 28, 53, 72].

- Interactions between OpenMPI, OpenMP, and CUDA can exist. For instance, the default OpenMPI software on the NCSA Lincoln Tesla cluster is compiled without threading support.

There are two popular threading models in use today: POSIX Threads (Pthreads) and OpenMP. OpenMP has become the dominant method used in the HPC community, and it was decided this was the model to be used for this study. It is not believed that this choice had a noticeable performance impact, and OpenMP is clearer to read. The thread-level parallelism is on a coarse-grain level, since CUDA is handling the fine-grain parallelism.

MPI defines four levels of thread safety: `SINGLE`, where only one thread is allowed. `FUNNELED` is the next level, where only a single master thread on each process may make MPI calls. The third level, `SERIALIZED`, allows any thread to make MPI calls, but only one at a time is using MPI. Finally, `MULTIPLE` allows complete multi-threaded operation, where multiple threads can simultaneously call MPI functions.

With many clusters having pre-installed versions of MPI libraries, sometimes with custom network infrastructure, it is not always possible to have access to the highest (`MULTIPLE`) threading level. Additionally, this level of threading support typically comes with some performance loss, so lower levels are preferred if they do not otherwise hinder parallelism [45]. Three implementations were created, using the `SERIALIZED`, `FUNNELED`, and `SINGLE` levels. The first implementation used one thread per GPU, with each thread responsible for any possible MPI communications with neighboring nodes. The second used $N+1$ threads for N GPUs, where a single thread per node handles all MPI communications and the other threads manage the GPU work. This can help alleviate resource contention between MPI and GPU copies, since each activity is on its own thread. Additionally, this lets one use the `FUNNELED` level, which increases portability and possibly can increase performance. Lastly, the third version uses OpenMP directives to only perform MPI calls inside single-threaded sections.

Similar to the dual-level MPI-CUDA testing, simulation runs were performed on the NCSA Lincoln Tesla cluster. At the time this study was performed, the MPICH2 implementation had interactions with the CUDA pinned-memory support, making it very slow for the CUDA streams overlapping cases. OpenMPI was used instead. Unfortunately, the OpenMPI versions available do not support any threading level other than `SINGLE`, and optimal network performance was not obtainable with custom compiled versions by the author. Hence, only the last implementation was used. An example implementation is shown in Figure 4.18, where simple computational overlapping is performed. CUDA computations are performed on threads $1 - N$, while MPI calls are performed on the single thread 0. With a `FUNNELED` hybrid implementation, the `omp master` pragma would be used instead, with care taken

```
// COMPUTE EDGES
if (threadid > 0)
  pressure <<<grid_edge,block>>> (edge_flags, div,p,pnew);

#pragma omp single
{
  MPI_Irecv(new ghost layer from north)
}
if (threadid > 0)
  cudaMemcpy(south edge layer from device to host)
// Ensure all threads have completed copies
#pragma omp barrier
#pragma omp single
{
  MPI_Isend(south edge layer to south)
  MPI_Irecv(new ghost layer from south)
}
if (threadid > 0)
  cudaMemcpy(north edge layer from device to host)
// Ensure all threads have completed copies
#pragma omp barrier
#pragma omp single
{
  MPI_Isend(north edge layer to north)
}

// COMPUTE MIDDLE
if (threadid > 0)
  pressure <<<grid_middle,block>>> (middle_flag, div,p,pnew);

#pragma omp single
{
  MPI_Wait(new ghost layer from north)
  MPI_Wait(new ghost layer from south)
}
// Ensure all threads wait for MPI communication
#pragma omp barrier
if (threadid > 0) {
  cudaMemcpy(new north ghost layer from host to device)
  cudaMemcpy(new south ghost layer from host to device)
}
// Ensure all threads have completed copies
#pragma omp barrier
#pragma omp single
{
  MPI_Waitall(south and north sends, allowing buffers to be reused)
}

if (threadid > 0)
  pressure_bc <<<grid,block>>> (pnew);

ROTATE_POINTERS(p,pnew);
```

Figure 4.18: An example Jacobi pressure loop using tri-level MPI-OpenMP-CUDA and simple computational overlapping. This uses the `SINGLE` threading level.

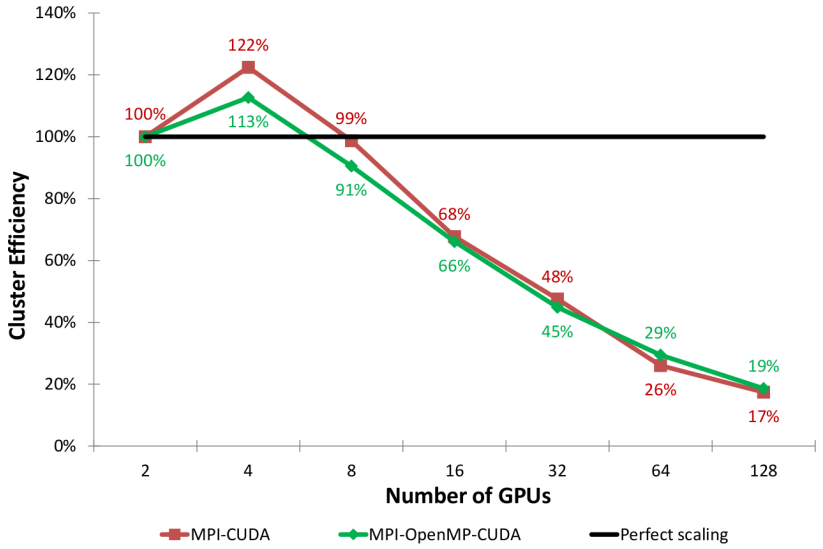


Figure 4.19: A comparison of weak scaling with the fully overlapped MPI-CUDA and single-threaded MPI-OpenMP-CUDA implementations, with growth in three dimensions. Since the hybrid implementations use all the GPUs of a single node, the base value for parallel scaling is set to a single node of the NCSA Lincoln Tesla cluster containing two GPUs.

since it has no implied barrier as `omp single` does.

4.2.6 Tri-Level MPI-OpenMP-CUDA Performance Results with NCSA Lincoln Tesla Cluster

Similar to the dual-level performance results, a lid-driven cavity problem at a Reynolds number of 1000 was chosen for performance measurements on the NCSA Lincoln Tesla cluster. As mentioned earlier, software issues on the NCSA Lincoln cluster precluded effective testing of anything but the tri-level implementation using single threading. Strong scaling and weak scaling measurements were performed, with little difference seen in most results. The weak scaling results with growth in three dimensions is the worst case for this application, and shows the most difference between the parallel methods. Figure 4.19 shows the scaling efficiency of the fully overlapped dual-level

MPI-CUDA and the tri-level MPI-OpenMP-CUDA implementations in the 3D growth weak scaling scenario. The MPI-CUDA data matches the fully overlapped data from Figure 4.15, though 100% is set with two GPUs (a single node) rather than one.

With fewer than 4 nodes (8 GPUs), the dual-level MPI-CUDA implementation performs better. This may be due to the more inefficient synchronization methods used in the tri-level method with single-threaded MPI. With 32 and 64 nodes (64 and 128 GPUs), there is a small benefit with the MPI-OpenMP-CUDA implementation. At this point, the amount of data being transferred may bring any efficiencies of the shared-memory model to the forefront, outweighing single-node synchronization. These results are not inconsistent with the hybrid performance results shown by Nakajima [76], where MPI-vector outperformed his hybrid MPI-OpenMP-vector model at 64 and fewer nodes, and started showing an increasing benefit at 96 nodes and beyond.

4.3 Conclusions

Three methods for exploiting the coarse-grain parallelism in a multiple-GPU system were described, and performance on a multi-GPU cluster was measured. For reasonably large domains, a similar dual-level MPI-CUDA implementation was shown to perform well compared to a dual-level Pthreads-CUDA implementation. This indicates a small penalty for large message transfers within a single node when using contemporary MPI implementations. This result is supported by comparisons of the dual-level MPI-CUDA implementations with tri-level MPI-OpenMP-CUDA implementations. Little benefit was seen in adopting the hybrid model with the hardware and software available.

Overlapping and domain decomposition are the two most important factors for

cluster scalability, while the CUDA kernel implementation is also important for overall performance. Implementations and scaling results have been detailed for three overlapping methods: no computation overlapping; computation with MPI; and fully overlapping computation, GPU data transfers, and MPI. Overlapping computation has been shown to be important for good scalability, and fully overlapping using CUDA streams is critical to get the best scalability with many GPUs. The domain decomposition method of 1D between GPUs, and 2D in the orthogonal dimensions within the GPUs typically maximizes the GPU efficiency and is simple to implement. However, with medium size clusters (16 or more GPUs) and domain sizes that increase in all three dimensions with larger problems, the data transfer size overtakes computation on the NCSA Lincoln cluster. In this case, the large drop in data transferred would likely outweigh any additional complexity of a 2D decomposition.

Because of the focus on domain decomposition, its effects on data transfer sizes, and therefore parallel scalability, the present study takes care to examine the cases of growth in 1, 2, and 3 dimensions when investigating weak scaling. For the results shown here, one of the growth dimensions is always in the 1D coarse-grain decomposition direction. The results show proper weak scaling in each case, where the amount of data per GPU is constant as more GPUs are added, but the results are quite different. In the 1D growth case, the amount of data transferred between partitions remains constant, and also the exact dimensions of the per-GPU partition remain identical. Given more resources and no growth in communications, it is possible to achieve perfect scaling if the interconnect does not saturate with exchanges and overheads can be well controlled. For the 2D and 3D growth cases where a 1D decomposition is used, there will always be growth in the amount of data to be exchanged as the number of GPUs increases. Since the work per-GPU remains constant and the amount

of data to transfer is growing, there will be some point where scaling starts to fall. It is important in any weak scaling study to note how the problem is growing and how it interacts with the domain decomposition method chosen.

CHAPTER 5

GEOMETRIC MULTIGRID FOR GPU CLUSTERS

5.1 Introduction

Multigrid methods are a class of techniques to solve boundary value problems such as the Poisson equation. They are described briefly in Press et al. [90] and in detail in Briggs et al. [18] and in Trottenberg et al. [107]. The fundamental idea is to apply multi-scale techniques, where the problem is solved at multiple resolution levels. This leads to not only a very efficient method with excellent convergence, but one where the convergence rate can be independent of the grid size. For the large problem domains expected to be run on HPC clusters, this is a very important feature.

Parallel multigrid solvers have been studied for some time. McBryan et al. [74] survey parallel multigrid methods, and Chow et al. [25] gives a more recent survey of important techniques. Particularly relevant to this work is the discussion on domain partitioning and the discussion of methods for coarse-grid solving. McBryan et al. show parallel efficiency results for numerous architectures, indicating very poor weak scaling results with most implementations. In contrast, the very recent results of Göddeke [38] show excellent scaling on a GPU cluster.

Previous studies using multigrid for GPUs include Goodnight et al. [43], who describe a multigrid solver on a single GPU for boundary value problems. This early work was done using early GPU hardware that were limited in both hardware

and software compared to current state of the art GPU computing. Cohen and Molemaker [26] describe the single GPU implementation and validation of an incompressible Navier-Stokes solver with Boussinesq approximation, using a multigrid solver. Göddeke et al. [41] describe integrating parallel multigrid solvers into an existing finite element solver using mixed precision. This is done in a framework of multiple CPU and GPU solvers, with choices that are made dynamically at each step.

The 2010 dissertation by Dominik Göddeke [38] discusses many aspects of multigrid on GPUs and GPU clusters. Parallel multigrid is investigated as part of an unstructured finite element solver that runs on CPUs or GPUs. Of particular relevance to this work is his discussion of smoothers, coarse-grid solvers, multigrid cycle type, and general comments on GPU and GPU cluster performance. A number of the issues are similar to those independently investigated here, with a different approach taken to the coarse-grid solver.

5.2 Geometric Multigrid Method

Direct solving of large systems of partial differential equations is computationally overwhelming, both in time and space. Iterative solvers such as the Jacobi and Gauss-Seidel methods offer a practical alternative; however, their convergence rate can be quite slow for large domains – a fact noted by Seidel in 1874 [96]. The conjugate gradient method (CG) discovered independently in 1952 by Hestenes and Stiefel [54], with extensions by Lanczos [69], provides a much faster technique, though still proportional to the domain size. The works by Fedorenko [32] and Bakhvalov [3] in the 1960s were the first to investigate multigrid techniques, showing their asymptotic optimality. The extensive work by Brandt in the 1970s [15, 16] demonstrated

the numerical efficiency of the method in operation, as well as developing many of the ideas used in current techniques.

Given the system $Au = f$ where A is an $N \times N$ matrix, the solver starts with an initial guess v. As iterative solvers proceed, the current solution v converges to the real solution u. From this, the error

$$e = u - v, \tag{5.1}$$

and the residual

$$r = f - Av, \tag{5.2}$$

can be defined. Manipulation of these give rise to the residual equation $Ae = r$ and the residual correction $u = v + e$. Together these formulate the residual correction algorithm:

begin

$r = f - Av$	calculate residual
$e = \mathsf{Solve}(A, r)$	solve for the error
$u = v + e$	residual correction

end

which shows how solving for the error can solve the equation. The multigrid method will make use of this idea.

When running an iterative solver such as Jacobi or Gauss-Seidel, an observation that can be made is that the error is *smoothed*. High frequency terms in the error rapidly diminish, while low frequency terms are removed much more slowly. This is a natural outcome of the narrow support of the discrete operation, where values can only propagate one grid cell per iteration. The multigrid method makes use of this

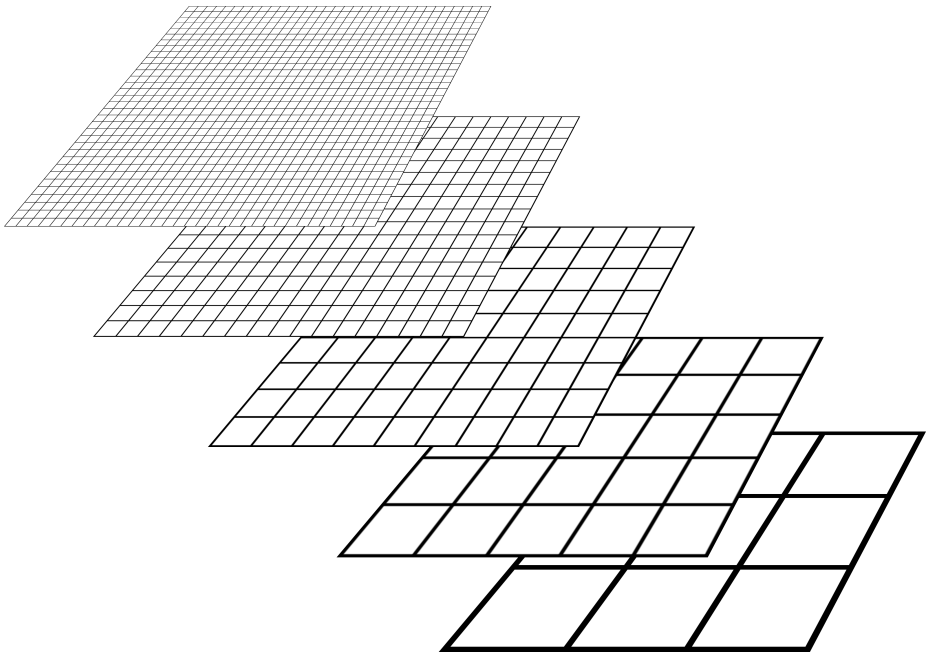
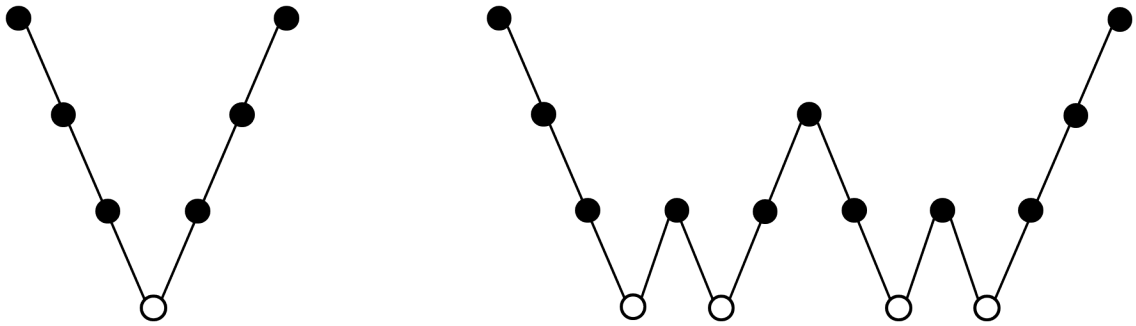


Figure 5.1: A view of a grid hierarchy with a structured rectilinear grid, going from 33^3 at the fine level to 3^3 at the coarsest level.

behavior, and these are called *smoothers* when run inside multigrid.

Given a smoothed error matrix, looking at the error on a coarser grid would raise the frequencies — low frequency errors on the fine grid become high frequency errors on the coarse grid. This leads to multigrid: errors are smoothed at the current grid level, the residuals are transferred to a coarser grid by a process called *restriction*, the problem is further solved at this coarse level, and then the result is interpolated back to the current grid (known as *prolongation*). This process creates a hierarchy of grids (see example in Figure 5.1), with the coarsest grid being solved by another method, such as a direct solver or a stationary iterative technique, such as Jacobi or Gauss-Seidel. The multigrid cycle may be repeated, and it does not have to follow a strict progression of fine to coarse. Two example cycle types, the common V-cycle and W-cycle, are represented in Figure 5.2.



(a) V-cycle (b) W-cycle

Figure 5.2: Examples of multigrid cycle types. Closed circles represent smoothing and open circles represent a coarsest grid solve (which may be approximate). (a) V-cycle, (b) W-cycle.

```
proc mgcycle(γ, u_k, f, s_pre, s_post)
  for i := 1 to s_pre do : Smooth                    pre-smoothing
  r_k := f − L u_k                                   compute residual using Laplacian
  r_{k−1} := R r_k                                   restrict residual
  if k − 1 = coarsest level
    then
        CoarseGridSolve(u, f, r_{k−1})               Apply coarse-grid solver
     else
        for i := 1 to γ do :
            mgcycle(γ, 0, r_{k−1}, s_pre, s_post)    multigrid on coarser levels
  fi
  u_k := u_k + P v_{k−1}                             prolong coarse result to this level
  for i := 1 to s_post do : Smooth                   post-smoothing
```

Figure 5.3: Geometric Multigrid Algorithm

A distinction can be made between geometric multigrid methods, which define coarse meshes directly from the fine mesh, and algebraic methods, which operate directly on the matrix of equations. The algebraic method is usually preferable for complex or unstructured grids. Since this research uses a structured Cartesian mesh, the geometric multigrid method is used. The complete algorithm for a multigrid cycle is shown in Figure 5.3. Setting $\gamma = 1$ will give a V-cycle, while $\gamma = 2$ gives a W-cycle.

5.3 GPU Implementation

Figure 5.4 shows the host code for a multigrid cycle. The `mgdata_t` structure array is initialized only once, and contains parameters for each level (grid sizes and spacing, pointers to pressure and residual at each level, etc.). Setting $gamma = 1$ leads to a V-cycle, while $gamma = 2$ leads to a W-cycle. Four routines are called from this host code: restriction, prolongation, a smoother, and a method for the coarsest grid solve. The `MGzeroMesh` function is a wrapper around `cudaMemset` to zero the appropriate device memory.

5.3.1 Restriction

The restriction step calculates the residual for the fine grid and downscales (restricts) it to a coarser grid. The host routine pseudo-code is shown in Figure 5.5. Creating the residual is a Laplacian operation, which is performed on the fine grid. The restriction operation takes a weighted average of neighboring cells on the fine grid. Common methods include 1-point injection, 7-point half-weighting, and 27-point full-weighting. All three methods are selectable in the current implementation by a compile-time decision, though it could easily be made a run-time decision.

```
static void mgcycle (mgdata_t* mgd,
                     int m, int mgend,
                     int gamma,
                     REAL* dphibuf, mpi_exchange* mex)
{
   int g;
   assert(gamma > 0);
   assert(m < mgend);

   // Smooth the result
   //   : smooth E[m] using initial value and R[m]
   MG_SMOOTHER(mgd, m, HCONSTANT_SMG_ITERV1, dphibuf, mex);

   // Make a coarser mesh of residuals.
   //   : E[m] and R[m] create R[m+1]
   restriction(mgd, m, dphibuf, mex);

   // Clear out the initial corrections for the coarser level
   //   : E[m+1] = 0
   MGzeroMesh(mgd, m+1);

   if (m+1 >= mgend) {
      mg_coarse_grid_solve(mgd, m+1, dphibuf, mex);
   } else {
      for (g = 0; g < gamma; g++) {
         mgcycle(mgd, m+1, mgend, gamma, dphibuf, mex);
      }
   }

   // Create the finer mesh
   //   : E[m] += interpolated E[m+1]
   prolongation(mgd, m+1, mex);

   // Smooth the result
   //   : smooth E[m] using initial value and R[m]
   MG_SMOOTHER(mgd, m, HCONSTANT_SMG_ITERV2, dphibuf, mex);
}
```

Figure 5.4: The host code for a multigrid cycle.

```
static void restriction(mgdata_t* mgd, int mglevel, REAL* dphibuf, mpi_exchange* mex)
{
   Zero edges of fine grid
   call LaplacianGPU kernel to create fine grid residual values
   exchange the results between GPUs

   zero edges of coarse grid
   call RestrictionGPU kernel to create coarse grid values from fine grid
   exchange the results between GPUs
}
```

Figure 5.5: Host pseudo-code for the restriction operation. The implementation also allows overlapping of computation and communication by computing the edges first, then starting asynchronous communication while computing the middle section.

The CUDA kernel implementations of the Laplacian and Restriction operations are shown in Appendix E.1 (`LaplacianGPU`) and Appendix E.2 (`RestrictionGPU`). Both versions shown use global memory rather than shared memory. The Laplacian kernel runs on the fine grid, while the restriction kernel has been implemented to run with one thread per coarse-grid cell, which greatly helps performance as each thread does a uniform amount of work.

On isotropic simulations, where $\Delta X = \Delta Y = \Delta Z$, half-weighting performs quite well. The convergence rate drops significantly as the simulation becomes anisotropic – where one dimension is significantly different from another. Full-weighting maintains the best convergence rates in these situations. The increase in global convergence rate far outweighs the small amount of extra time taken for weighting, recalling that the restriction kernel runs on the coarse grid, and hence operates with 8 times fewer threads than a fine-grid kernel. The 27-point full-weighting is used for all results in this thesis.

5.3.2 Prolongation

The prolongation operation for multigrid ought to be the inverse operator of the restriction operation. With full-weighting used for restriction, the 3D adjoint is trilinear interpolation. The value in the new fine grid will be a distance-weighted average of the values of the surrounding coarse-grid cells. Care must be taken with the operation ordering to prevent undesired floating-point rounding differences.

The prolongation host code calls the prolongationGPU kernel, shown in Appendix E.3 followed by setting the pressure boundary conditions and an exchange between GPUs. The operation occurs on the coarse grid, and no MPI overlapping is implemented for this function. A shared memory implementation of this kernel was created but showed no advantage over this global memory version on the tested platforms, so it is not used.

5.3.3 Smoother

Two smoothers have been implemented: weighted Jacobi and Red-Black Gauss-Seidel, including a relaxation parameter to allow overrelaxation. The host code for weighted Jacobi performs a number of iterations of the sequence: smoother kernel, exchange, and set boundary conditions. Additionally, an exchange is done at the end to ensure all ghost cells are consistent. The Jacobi smoother allows computation / communication overlap. The host code for weighted Red-Black Gauss-Seidel performs a number of iterations of the sequence: smoother kernel (red), exchange, smoother kernel (black), set boundary conditions, and exchange. The global memory CUDA kernel code for these kernels is shown in Appendix E.4 and E.5. A shared memory version of the weighted Jacobi solver has also been implemented.

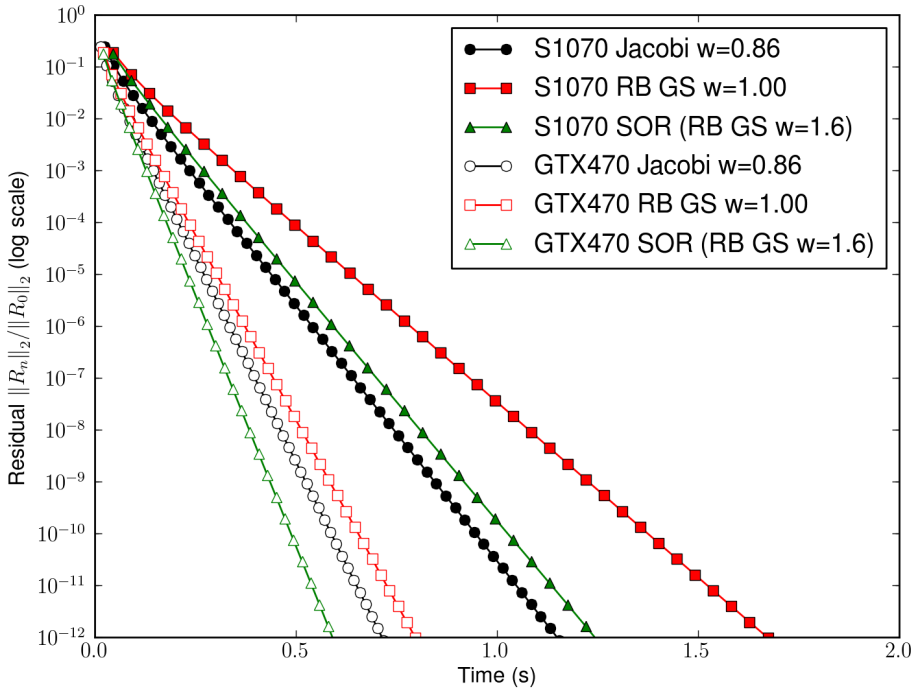


Figure 5.6: Comparison of smoothers used inside a multigrid V-cycle on the NVIDIA Tesla S1070 and GTX 470 with the Fermi architecture. Time is plotted against the residual level for a 129^3 problem on a single GPU using double-precision calculations. 4 pre-smoothing and 4 post-smoothing iterations at each grid level. No multigrid truncation was applied. The weighted Jacobi solver uses shared memory.

Figure 5.6 shows the performance of the smoothers on two platforms when used inside a multigrid V-cycle. The time taken counts all multigrid activity, of which the smoother was measured to be between 67% and 82% of the total. As expected, the spacing between cycles indicates the Gauss-Seidel method converges faster than weighted Jacobi ($\omega = 0.86$) and the SOR weighting ($\omega = 1.60$) converges faster yet. On both platforms, using weighted Jacobi finishes faster than Red-Black Gauss-Seidel, which indicates an implementation difference. Using SOR with a near-optimal weight for this problem, the performance improves substantially, but does not outweigh the implementation performance difference on the S1070. It does lead to the fastest solution on the GTX 470. Comparing the two architectures, the previous generation CC 1.3 S1070 and the current CC 2.0 (Fermi) GTX 470, not only is the Fermi system faster overall, but all the solutions are closer.

5.3.4 Coarse-Grid Solver

Classic multigrid uses a full-depth cycle along with an exact solution for the coarsest grid. When data is distributed among multiple nodes of a parallel system, it is not possible to reduce the depth to the lowest level without repartitioning the data. Additionally, if the grid size is not identical in all dimensions, some form of semi-coarsening (restriction in only some of the dimensions) is needed to continue reduction below the point where the smallest dimension has only one interior cell. Finally, even when this reduction is done, the varying boundary conditions allowed make an exact solution not as simple as for the case of all-Dirichlet boundaries.

The first method applied is the approximate solution method, where a fixed number of iterations of the smoother are applied at the coarsest level. Figure 5.7 shows the effect of this method as the multigrid cycle is truncated earlier. While

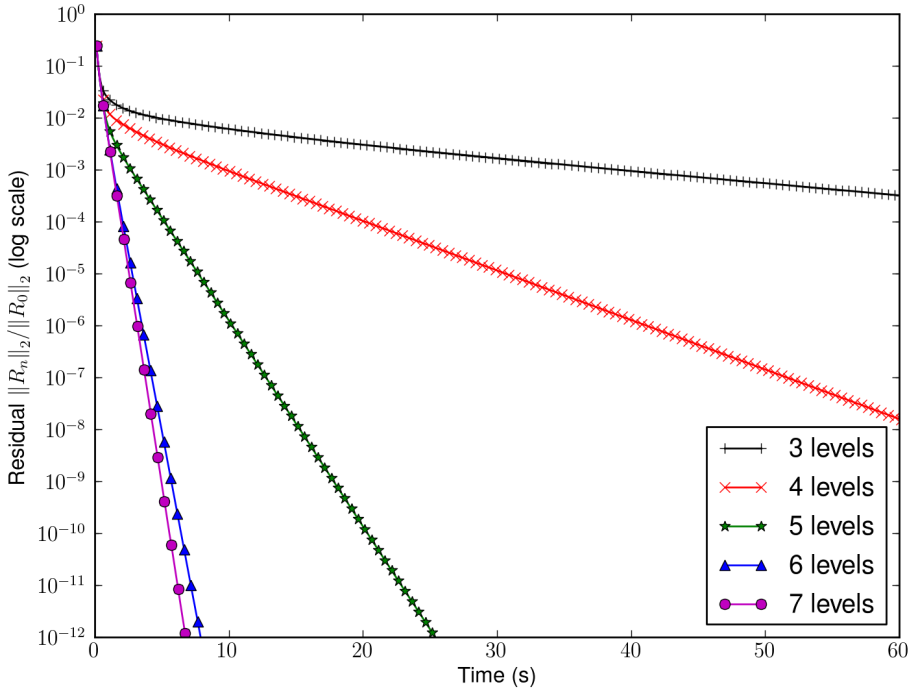


Figure 5.7: Effect of early truncation level using V-cycles. Time is plotted against the residual level for a 257^3 problem on a single Tesla S1070 GPU using double-precision calculations. The smoother is a weighted Jacobi with $w = 0.86$ and 4 pre-smoothing and 4 post-smoothing iterations at each grid level. The coarse-grid solver is 16 iterations of the weighted Jacobi smoother. A marker is shown for each 4 loops of the multigrid cycle. The coarsest grid in this example for 3 levels is 65^3, and for 7 levels is 5^3.

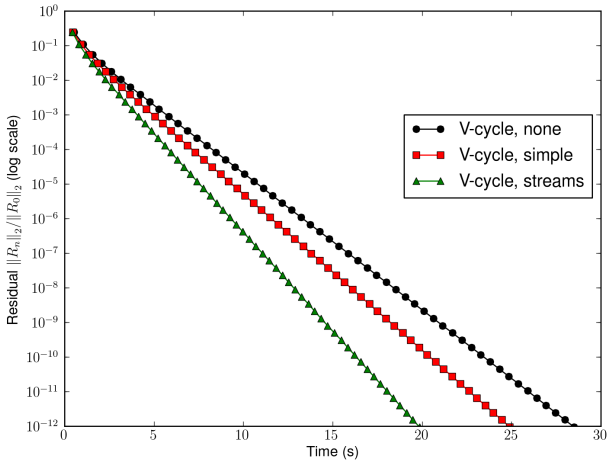
excellent convergence and computational performance is obtained with deep cycles, with earlier truncation the performance begins to take on the characteristics of the iterative solver.

Another method tried is a dynamic number of iterations of the smoother, where the number of iterations is either based on the coarsest grid size or is run until the residual is reduced to a desired accuracy level. This mitigates the effect of cycle truncation on convergence, but the number of iterations required grows quite rapidly. Over 1,000 iterations of the Jacobi solver are needed to obtain an accurate solution on a 65^3 grid. This is even less acceptable on a cluster, where communication must be performed between each iteration. One solution to this dilemma is to use a well-tuned parallel solver, such as the parallel conjugate gradient method. This is the solution used by Göddeke [38], where a conjugate gradient coarse-grid solver is used and set to reduce the initial residuals by two decimal digits.

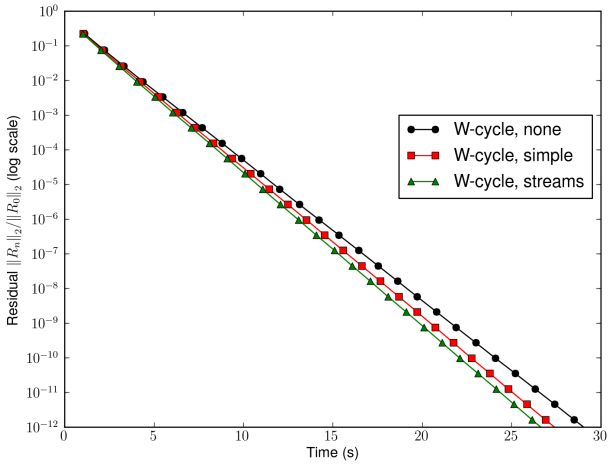
The solution we propose is to use multigrid as the coarse-grid solver. This embedded multigrid solver could use different parameters (smoother type, number of iterations, cycle type) than the outer multigrid solver and could be iterated more than once. While on a single GPU this method has little impact (if the parameters are identical, it is identical to performing no truncation), it has some strong advantages on a cluster. These will be explored in Section 5.4.

5.3.5 Computational Overlapping

The three overlapping strategies described in Section 4.2.3 are also used in the smoother and restriction operations during the multigrid process. The prolongation operation does not overlap, but it uses less than 10% of the multigrid time on these tests. Figure 5.8 compares the three strategies in the multigrid solver for a calculation



(a)



(b)

Figure 5.8: Comparison of overlapping strategies. Time is plotted against the residual level for a double-precision 513^3 problem using 8 GPUs. The smoother is a weighted Jacobi with $w = 0.86$ and 4 pre-smoothing and 4 post-smoothing iterations at each grid level. Truncation occurs at 6 levels (17^3), where the coarsest grid is amalgamated to a single GPU and four V-cycles are performed on this grid. A marker is shown for each loop of the multigrid cycle. (a) V-cycle, (b) W-cycle

using eight GPUs (four nodes of the NCSA Lincoln Tesla cluster). With V-cycles, the performance gain from overlapping is significant for both simple overlapping and again when using CUDA streams. Both show an improvement over the W-cycle results for this problem, and benefits for overlapping are minimal for the W-cycles.

5.4 Amalgamated Multigrid for GPU Clusters

In a parallel multigrid implementation, the decomposition of each coarsest grid is a critical factor. In this implementation, the dimensions (nx_c, ny_c, nz_c) of the coarse grid are $nx_c = (nx_f - 1)/2 + 1$, $ny_c = (ny_f - 1)/2 + 1$, and $nz_c = (nz_f - 1)/2 + 1$. No semi-coarsening is applied, meaning each dimension is reduced. Each GPU is responsible for the coarse-grid cells deriving from the restriction operator applied to its current domain, meaning the partitioning is static on each grid. One implication of this partitioning is that the number of GPUs used remains constant while the work decreases by a factor of $2^3 = 8$ at each step. More importantly, when $nz_c = \#gpus$, no more coarsening can be applied, as this would result in some GPUs having no pixels. Even before this time, the amount of work on each GPU is very small.

To alleviate the issue of rapidly shrinking work and the effect it has on computation / communication ratios, it is not uncommon for the multigrid cycle (e.g. V-cycle) to be truncated, and the coarse-grid solver be set to an algorithm such as parallel conjugate gradient as used in Göddeke [38]. A related idea discussed by Gropp [47] and expanded on by Chow et al. [25] is to amalgamate the grids at the coarse level, meaning a gather operation combines the coarse grids at all levels, which is then solved on a single node, and the results are then scattered back. Since the coarse levels are extremely small compared to the fine grid, the amount of data distributed

across the network is relatively small, and the resulting combined coarse grid level may be more effectively solved.

Gropp further notes that by using an allgather operation and redundant calculations, each node can calculate the coarse grid, which means there is no need for a scatter operation. The allgather method was implemented and compared with the regular gather/solve/scatter solution. Measured times were effectively equal at most truncation levels, but were longer with shallow truncation levels (where the coarse grid was large). While in theory the performance of `MPI_Allgather` can approach that of `MPI_Gather`, tests on the NCSA Lincoln Tesla cluster show that `MPI_Allgather` is slower, and the difference widens as more GPUs are added. With 8 nodes, it is 1.5 to 2 times slower, which confirms the results. An additional consideration for large clusters is power consumption, which may mean large clusters will want to reduce redundancy when it offers no clear benefit. Future directions for GPU scheduling may also allow GPUs to be dynamically used by other users, which indicate that the amalgamation technique of idling GPUs at small problem sizes may present a benefit to the total cluster throughput.

Altogether, the issue of coarse-grid solving is one of solving the fine grids with high performance, where domains are large and performance is dominated by computation and bandwidth; and also effectively solving coarse grids, where domains are small and performance is dominated by latency. There are numerous ways to approach the solution, and a number of enhancements can be used for each. What this study proposes is an *embedded multigrid with amalgamation* strategy.

For the coarse-grid solver, it is clear from Figure 5.7 that a process of truncation with Jacobi iterations is not acceptable, as many GPUs will force early truncation. Amalgamating the coarse grids of GPUs allows deeper levels to be taken. In the

limit, this means amalgamation to a single GPU where a full depth multigrid cycle can be performed. Hence, multigrid is embedded as the coarse-grid solver in the parallel multigrid implementation. Convergence rates are now equal to those seen with textbook full-depth multigrid cycles, and for large coarse grids (e.g. $129 \times 129 \times 129$ seen on 128 GPU tests), performance is high relative to other methods. Since network communication costs are zero within the amalgamated solver, it can be advantageous to perform multiple multigrid cycles at the coarse level, leading to an improvement in the overall convergence rate for very little cost.

Amalgamation can often be performed with little to no extra memory use, as the rapid reduction in size per coarsening quickly reduces the memory needed. The current implementation can amalgamate to a single GPU at 3 levels for most problems without using any additional memory, as the second pressure buffer used by the Jacobi solver can be split into three parts holding the coarse grid pressure values, the coarse grid residuals, and a pressure buffer sized for the coarse grid.

One limitation of the current implementation is that a single depth is chosen, at which point all the results are amalgamated to a single GPU. While this drives communication costs to zero during the coarse solve, it also removes all parallelism from the other $N - 1$ GPUs. It is likely that a solution fanning in, for instance from 64 GPUs, to 8, then to 1, would better control the computation / communication ratio. Another limitation is that even on a single GPU, there is likely to be a point earlier than the coarsest $3 \times 3 \times 3$ grid, where a different solver, such as conjugate gradient, will be faster.

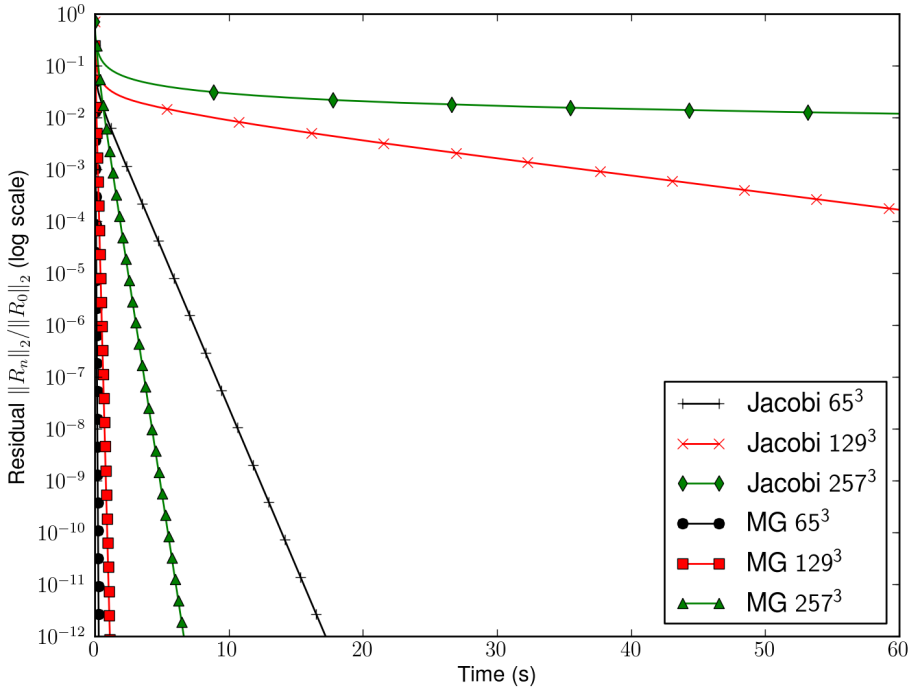


Figure 5.9: Performance of full-depth V-cycle multigrid compared to iterative Jacobi. Time is plotted against the residual level for a 65^3, 129^3, and 257^3 problem on a single Tesla S1070 GPU using double-precision calculations. The smoother is a weighted Jacobi with $w = 0.86$ and 4 pre-smoothing and 4 post-smoothing iterations at each grid level. For clarity, a marker is shown for each 2 loops of the multigrid cycle, every 1000 Jacobi iterations at 257^3, and every 4000 Jacobi iterations at 129^3 and 65^3.

5.5 Performance Results with NCSA Lincoln Tesla and TACC Longhorn Clusters

A comparison of the computational performance between multigrid and unweighted Jacobi is shown in Figure 5.9. A 3D lid-driven cavity problem was started at three different grid sizes, and the time taken by the pressure solver is plotted against the residual level for the initial time step.

Figure 5.10 shows the performance of the multigrid algorithm on a GPU cluster

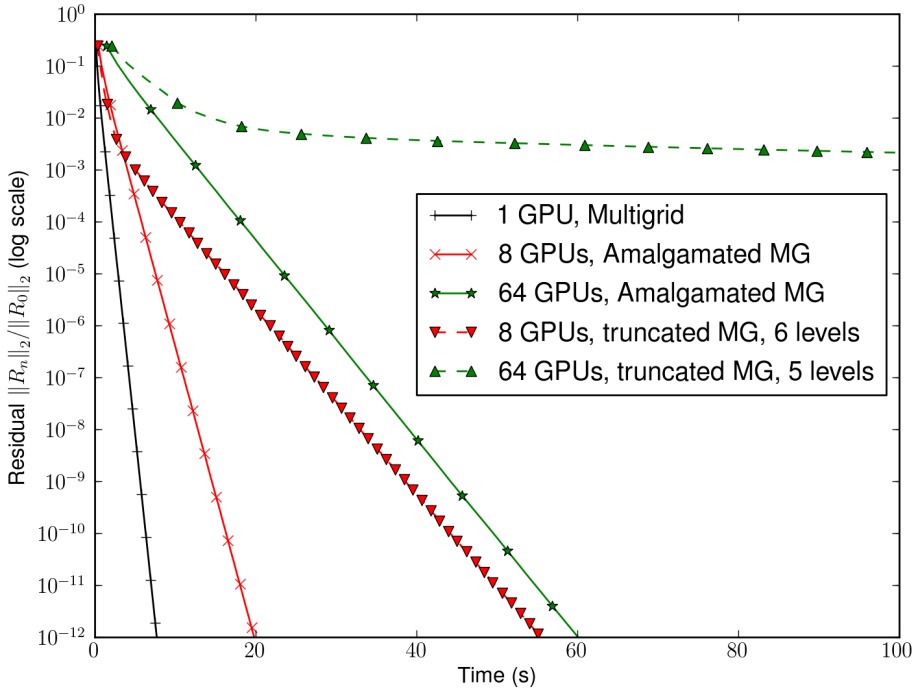


Figure 5.10: Convergence and parallel efficiency of truncated and amalgamated multigrid on 1, 8, and 64 GPUs where the problem size scales with the number of GPUs. Time is plotted against the residual level for a double-precision problem using 257^3 on 1 GPU, 513^3 using 8 GPUs, and 1025^3 using 64 GPUs on the NCSA Lincoln Tesla cluster. The smoother is a weighted Jacobi with $w = 0.86$ and 4 pre-smoothing and 4 post-smoothing iterations at each grid level. A marker is shown for each 4 loops of the multigrid cycle. V-cycles and CUDA streams overlapping were used for each.

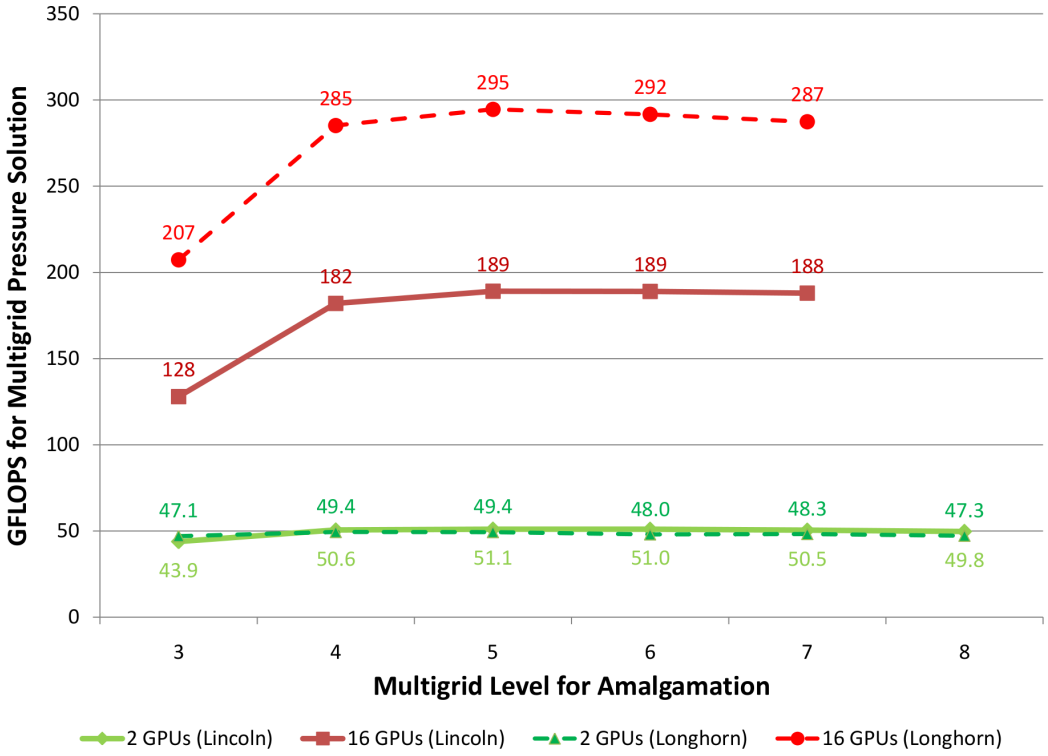


Figure 5.11: Performance of the multigrid solver with different truncation levels selected, using single precision. Problem size scales with the number of GPUs, with 513^3 on 2 GPUs and 1025^3 using 16 GPUs on the NCSA Lincoln Tesla and TACC Longhorn clusters. The smoother is a weighted Jacobi with $w = 0.86$ and 4 pre-smoothing and 4 post-smoothing iterations at each grid level. V-cycles and CUDA streams overlapping were used for each.

for relatively large problems (16M, 128M, and 1024M cells). In particular, the results of the amalgamation with embedded multigrid are compared to the fixed iteration approximate coarse-grid solver. With 8 GPUs, the coarsest grid is 17^3, while with 64 GPUs with coarsest grid is 65^3. On a single GPU, a full-depth V-cycle was performed, and hence no truncation was performed.

Figure 5.11 shows the GPU cluster performance at different amalgamation levels. Amalgamating at a shallow level leads to a very large coarse grid; for example, in the 1025^3 problem using 3 multigrid levels, the coarsest grid is 257^3. This leads to poor

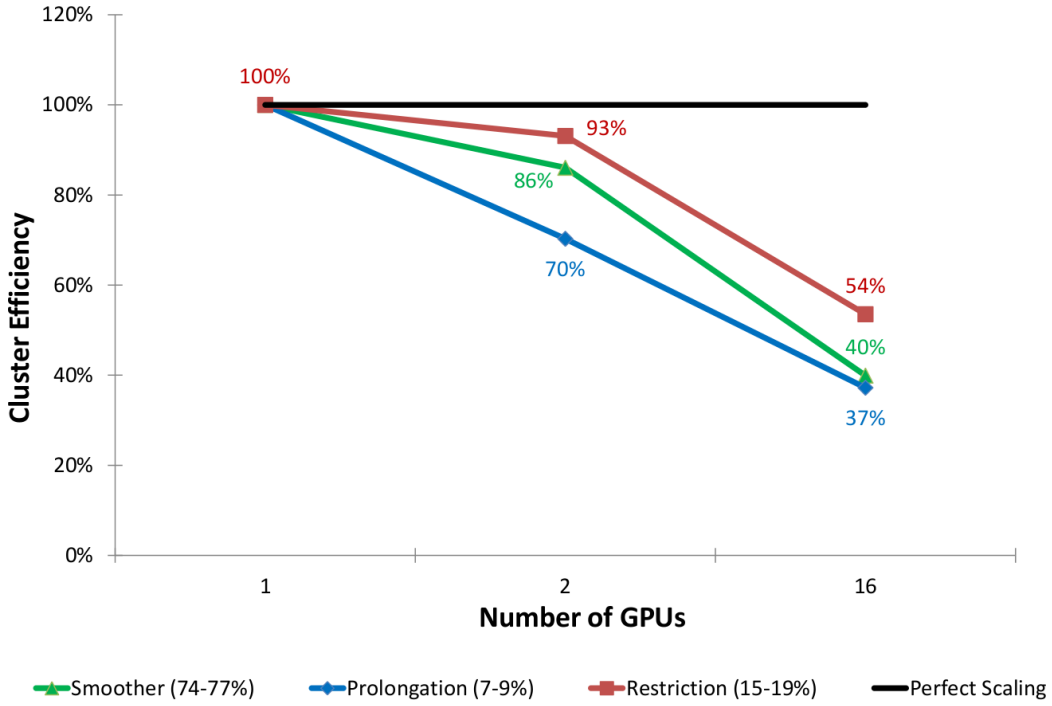


Figure 5.12: Weak scaling performance of the multigrid solver components, with their overall portion of the solver time shown. Amalgamated multigrid at 5 levels was used with parallel implementations, and the fully overlapped versions were used. Problem size scales with the number of GPUs, with 513^3 on 2 GPUs and 1025^3 using 16 GPUs on the NCSA Lincoln Tesla cluster. The smoother is a weighted Jacobi with $w = 0.86$ and 4 pre-smoothing and 4 post-smoothing iterations at each grid level. V-cycles and CUDA streams overlapping were used for each, and all computations were in single precision. The chart legend indicates the percentage of the multigrid solver time taken by that component.

performance both because of the large size to be communicated and for the loss of parallel computation at a level where it is still helpful. It can seen from the figure that there is a small benefit to amalgamating earlier than is required by the parallel implementation. Comparing the two clusters, the 2 GPU solution has little difference as it benefits only from the increased host/device memory bandwidth, while the 16 GPU solution is over 50% faster on the TACC Longhorn cluster, where the increased Infiniband bandwidth leads to faster performance.

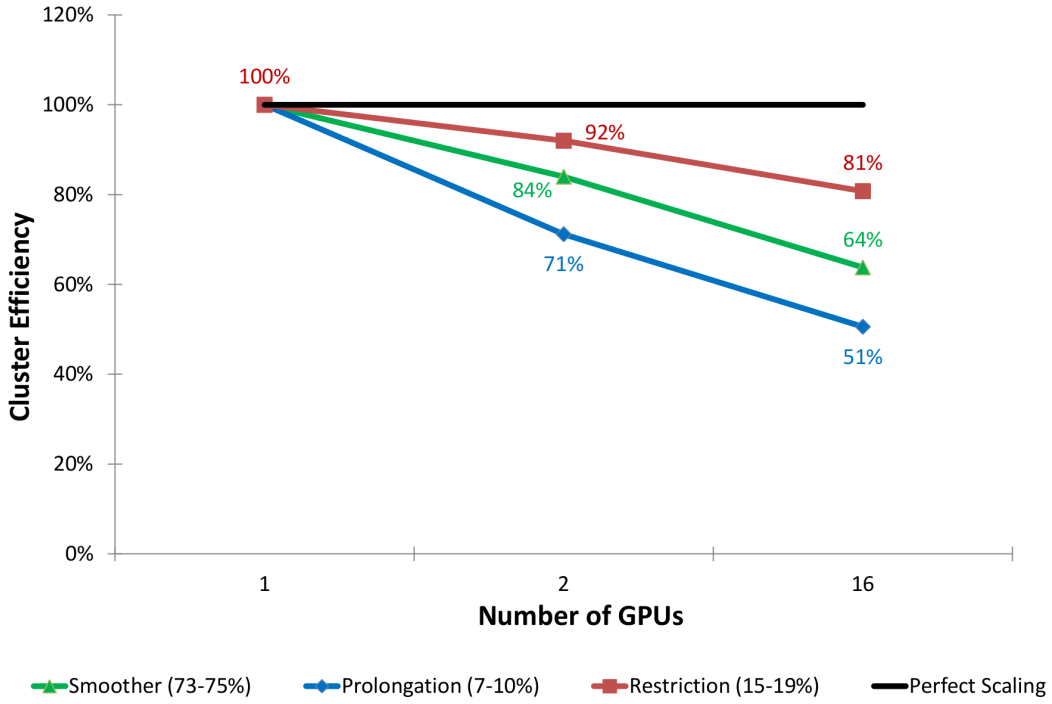


Figure 5.13: Weak scaling performance of the multigrid solver components using the TACC Longhorn system. Better scalability is seen with the faster memory transfer and network speeds.

Figures 5.12 and 5.13 break out the weak scalability by multigrid component. The prolongation component has the worst scaling, but also takes the least amount of time. Restriction scales better than the other components, but still shows poor scaling behavior at 16 GPUs. The smoother (weighted Jacobi for this case) takes the majority of the time, and shows scalability between the other components. While scaling to two GPUs is not bad, scaling to 16 GPUs is disappointing. Comparing the scaling of the two measured systems shows that the scaling to 16 GPUs is improved on the Longhorn system but still trends downward. Note that this is a three-dimensional scaling case, and comparing to the weak scaling results for Jacobi, shown in Figure 4.15, indicates that some of the responsibility may lie in excessive data movement at the fine-grid level.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

This thesis presents multiple approaches to parallelizing an incompressible Navier-Stokes flow solver on multi-GPU clusters. NVIDIA's CUDA is used for fine-grain parallelism, while combinations of Pthreads, MPI, and hybrid MPI-OpenMP are used for coarse-grain parallelism. Comparisons between contemporary CPUs and GPUs using a CFD flow solver show significant advantages for GPUs in computational performance, energy efficiency, and cost effectiveness. Reviewing the dual-level Pthreads-CUDA and MPI-CUDA shows that the performance difference on even relatively small problems is minimal even with no computational overlapping, indicating that modern MPI implementations can provide excellent throughput with large messages. With this minimal performance loss on a single node, a dual-level MPI-CUDA solution can provide the flexibility to run seamlessly on a single GPU, a multi-GPU workstation, or a multi-GPU cluster.

Three methods for computational overlapping in the MPI-CUDA method have been implemented, with performance measurements collected on 64 nodes (128 GPUs) of the the NCSA Lincoln Tesla Cluster. As expected, computational overlapping on systems with many GPUs is quite important. The method using CUDA streams to fully overlap computation, GPU data transfers, and MPI network communica-

tions is the best solution on all cases shown, with the largest benefits over simple overlapping typically seen between 4 and 16 GPUs. The most advantageous parallel implementation (fixed size communication) using 128 GPUs obtained a speedup of 1332 over the eight cores of a single CPU node. With 2D growth, a speedup of 943 was obtained, and with 3D growth a speedup of 245 was seen. These results use a 1D decomposition across the GPUs, with an orthogonal 2D stencil decomposition used within each GPU. Timing measurements of weak scaling with 1D, 2D, and 3D growth show this to be a bottleneck for 3D growth beyond 16 GPUs on the NCSA Lincoln cluster. Results of the same fully overlapped implementation run on the Longhorn cluster at the Texas Advanced Computing Center (TACC) indicated the faster memory transfers and networking resulted in twice the parallel efficiency with 32 GPUs.

Methods for tri-level MPI-OpenMP-CUDA parallelism were explored, with mixed results. A number of structural issues came into play: the multi-GPU clusters available had no more than two GPUs per node, limiting the possible intra-node efficiency gain; the number of nodes required to see the best benefit may be more than the 64 nodes available for this research on the largest cluster used; the MPI implementation used has seen intra-node message passing optimization since the early papers in this area, limiting the possible gain; the network-optimized MPI implementations available on the clusters used have the minimum threading level support, making the hybrid implementation tested more inefficient than it could otherwise be.

A numerically efficient Poisson solver is crucial in CFD applications for conservation of physical quantities. For simulations that need high accuracy or time-accurate solutions, the addition of a parallel multigrid pressure Poisson solver allows rapid

convergence of the pressure at each timestep even with very large models. Multigrid solvers introduce software complexity and require attention to detail to achieve the best convergence rates. As part of this research, an MPI-CUDA multigrid solver was developed. Each part of the GPU-enabled multigrid solver is examined, including convergence and performance of the Jacobi, Red-Black Gauss-Seidel, and Successive Overrelaxation smoothers on both the newest Fermi architecture and the previous generation. Early multigrid truncation with an approximate coarse-grid solver was examined, and while some truncation is possible with little difference, more than a few of the coarsest levels skipped leads to slow convergence. As with the Jacobi solver, overlapping of the CUDA kernels with GPU data transfers and MPI network communication is investigated. With V-cycles, there is a distinct performance benefit to each level of overlapping, with the fully overlapped implementation almost 40% faster on 8 GPUs.

Coarse-grid solvers are a important feature of parallel multigrid solvers, and numerous methods have been proposed for their solution. This thesis presents a new method: embedded multigrid with amalgamation. In this technique, the coarsest grid is assembled on a single GPU where it is solved with a single-GPU multigrid implementation, with identical logic to the outer multigrid solution. This combines minimization of network communication on small grids, amalgamation to provide the GPU with enough work that it operates efficiently, and the excellent convergence rate of multigrid for the coarse grid.

6.2 Future Work

While the numerical computations are sufficient for a wide variety of CFD work, many improvements could be made, and a number of them are noted in Section 3.2.1. The projection model as implemented is first-order in time which could be improved to second-order. The Runge-Kutta time-stepping methods have been implemented, but validation remains to be performed. The simple Smagorinsky turbulence model is implemented but remains to be validated, and other turbulence models can be considered.

The current implementation uses a uniform mesh to represent the domain. Structured mesh methods to adapt to differing regions would greatly improve the model, such as body-fitted coordinates, hybrid meshes, or adaptive mesh refinement (AMR). The AMR method of Berger and colleagues [6, 7, 8] looks particularly well suited to this method. AMR uses a set of nested grids at different resolutions, which maintains the simplicity of a structured rectilinear grid while allowing the model to adapt the mesh resolution to resolve detail in areas where it is needed.

Alternate domain decompositions should be compared. While there are compelling reasons to use the 1D inter-GPU decomposition for small clusters, this method becomes inefficient for 32 or more nodes. The zero copy feature of CUDA 2.2 and later can be used to effectively overlap the domain edge memory transfer along with the gather / scatter kernels. Analysis of data transfers indicate that a 2D decomposition should have a significant impact with large data sets on many nodes.

The CUDA kernels are implemented in fairly straightforward ways, and do not use excessive optimizations. It should be possible to improve their performance if needed. One feature often used in similar code is the use of texture memory to

take advantage of its cache as well as special features such as built-in tri-linear interpolation. While some CUDA optimizations were pursued, the focus of this thesis is on the interaction of the fine-grain CUDA parallelization with the coarse-grain MPI and OpenMP methods, and hence many opportunities for faster CUDA kernels were not followed through.

The amalgamation method with embedded multigrid solver has been shown to be an effective solution. It has a sharp transition from using all available GPUs to only one, which could be improved. One possibility is step merging where every other GPU merges with its neighbor, another is to choose discrete merge points such as $N \Rightarrow 16 \Rightarrow 4 \Rightarrow 1$ GPU. Another useful line of investigation would be looking at alternate solvers such as the conjugate gradient method for the coarse-grid solver.

REFERENCES

[1] Akio Arakawa and Vivian R. Lamb. Computational design of the basic dynamical processes of the UCLA general circulation model. *Methods in Computational Physics*, 17:173–265, 1977.

[2] Jong-Jin Baik, Jae-Jin Kim, and Harindra J. S. Fernando. A CFD model for simulating urban flow and dispersion. *Journal of Applied Meteorology*, 42(11):1636–1648, 2003.

[3] Nikolai Sergeevich Bakhvalov. On the convergence of a relaxation method with natural constraints on the elliptic operator. *USSR Computational Mathematics and Mathematical Physics*, 6(5):101–135, 1966.

[4] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing L. Lusk, Rajeev Thakur, and Jasper Larsson Träff. MPI on a million processors. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 20–30, 2009.

[5] Aaron Becker, Isaac Dooley, and Laxmikant V. Kalé. Flexible hardware mapping for finite element simulations on hybrid CPU/GPU clusters. In *SAAHPC: Symposium on Application Accelerators in HPC*, July 2009.

[6] John Bell, Marsha Berger, Jeff Saltzman, and Mike Welcome. Three-dimensional adaptive mesh refinement for hyperbolic conservation laws. *SIAM Journal on Scientific Computing*, 15(1):127–138, January 1994.

[7] Marsha Berger and Phillip Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, May 1989.

[8] Marsha Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, March 1984.

[9] Marsha J. Berger, Michael J. Aftosmis, D. D. Marshall, and Scott M. Murman. Performance of a new CFD flow solver using a hybrid programming paradigm. *J. Parallel Distrib. Comput.*, 65(4):414–423, 2005.

[10] Konstantin Berlin, Jun Huan, Mary Jacob, Garima Kochhar, Jan Prins, William Pugh, P. Sadayappan, Jaime Spacco, and Chau-Wen Tseng. Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In *LCPC*, pages 194–208, 2003.

[11] Jiri Blazek. *Computational Fluid Dynamics: Principles and Applications*. Elsevier, 2001.

[12] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *ACM SIGGRAPH 2003 Papers*, pages 917–924, San Diego, California, 2003. ACM.

[13] Roland Bouffanais. *Simulation of shear-driven flows: transition with a free surface and confined turbulence*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2007.

[14] Steve W. Bova, Clay P. Breshears, Christine E. Cuicchi, Zeki Demirbilek, and Henry A. Gabb. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. *Int. J. High Perform. Comput. Appl.*, 14(1):49–64, 2000.

[15] Achi Brandt. Multi-level adaptive technique (MLAT) for fast numerical solutions to boundary value problems. In H. Cabannes and R. Temam, editors, *Lecture Notes in Physics 18*, pages 82–89, Berlin, 1973. Proc. 3rd Int. Conf. Numerical Methods in Fluid Mechanics, Springer.

[16] Achi Brandt. Multi-level adaptive solutions to boundary-value problems. *Math. Comput.*, 31(138):333–390, April 1977.

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

[18] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial (2nd ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[19] David L. Brown, Ricardo Cortez, and Michael L. Minion. Accurate projection methods for the incompressible navier-stokes equations. *Journal of Computational Physics*, 168(2):464–499, 2001.

[20] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[21] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 12, Dallas, Texas, United States, 2000. IEEE Computer Society.

[22] Franck Cappello, Olivier Richard, and Daniel Etiemble. Performance of the NAS benchmarks on a cluster of SMP PCs using a parallelization of the MPI programs with OpenMP. In *PaCT*, pages 339–350, 1999.

[23] Franck Cappello, Olivier Richard, and Daniel Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. In *HPCA*, pages 349–359, 2000.

[24] Alexandre Joel Chorin. Numerical solution of the Navier–Stokes equations. *Math. Comput.*, 22:745–762, 1968.

[25] Edmond Chow, Robert D. Falgout, Jonathan J. Hu, Raymond S. Tuminaro, and Ulrike Meier Yang. *A Survey of Parallelization Techniques for Multigrid Solvers*, chapter 10. SIAM Series on Software, Environments, and Tools. SIAM, 2006.

[26] Jonathan M. Cohen and M. Jeroen Molemaker. A fast double precision CFD code using CUDA. In *Proceedings of Parallel CFD*, 2009.

[27] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[28] Suchuan Dong and George E. Karniadakis. Dual-level parallelism for deterministic and stochastic CFD problems. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

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

[30] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, pages 47+, Washington, DC, USA, 2004. IEEE Computer Society.

[31] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Memory - sequoia: programming the memory hierarchy. In *SC*, page 83, 2006.

[32] Radii Petrovich Fedorenko. A relaxation method for solving elliptic difference equations. *USSR Computational Mathematics and Mathematical Physics*, 1(4):1092–1096, 1964.

[33] Harindra J. S. Fernando, Dragan Zajic, Silvana Di Sabatino, Reneta Dimitrova, Brent Hedquist, and Ann Dallman. Flow, turbulence, and pollutant dispersion in urban atmospheres. *Physics of Fluids*, 22(051301), 2010.

[34] Joel H. Ferziger and Milovan Perić. *Computational Methods for Fluid Dynamics*. Springer, 3rd edition, 2002.

[35] Julia E. Flaherty, David Stock, and Brian Lamb. Computational fluid dynamic simulations of plume dispersion in urban Oklahoma City. *Journal of Applied Meteorology and Climatology*, 46(12):2110–2126, December 2007.

[36] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. *SIGPLAN Notices*, 45(3):347–358, 2010.

[37] Urmila Ghia, Kirti N. Ghia, and C.T. Shin. High-Re solutions for incompressible flow using the Navier–Stokes equations and a multigrid method. *Journal of Computational Physics*, 48:387–411, 1982.

[38] Dominik Göddeke. *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters*. PhD thesis, Technischen Universität Dortmund, Dortmond, Germany, February 2010.

[39] Dominik Göddeke, Sven H.M. Buijssen, Hilmar Wobker, and Stefan Turek. GPU acceleration of an unmodified parallel finite element Navier–Stokes solver. In *International Conference on High Performance Computing & Simulation*, pages 12–21, 2009.

[40] Dominik Göddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Sven H.M. Buijssen, Matthias Grajewski, and Stefan Turek. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing, Special issue: High-performance computing using accelerators*, 33(10–11):685–699, November 2007.

[41] Dominik Göddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Hilmar Wobker, Christian Becker, and Stefan Turek. Using GPUs to improve multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering (IJCSE)*, 4(1):36–55, 2008.

[42] Brice Goglin. High throughput intra-node MPI communication with Open-MX. In *PDP*, pages 173–180, 2009.

[43] Nolan Goodnight, Gregory Lewin, David Luebke, and Kevin Skadron. A multigrid solver for boundary-value problems using programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 102–111, 2003.

[44] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. *Numerical Simulation in Fluid Dynamics: A Practical Introduction*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[45] William Gropp and Rajeev Thakur. Thread-safety in an MPI implementation: Requirements and analysis. *Parallel Computing*, 33(9):595–604, 2007.

[46] William Gropp, Rajeev Thakur, and Ewing Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, 1999.

[47] William D. Gropp. Parallel computing and domain decomposition. In Tony F. Chan, David E. Keyes, Gérard A. Meurant, Jeffrey S. Scroggs, and Robert G. Voigt, editors, *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, Philadelphia, PA, USA, 1992. SIAM.

[48] Jean Luc L. Guermond, Peter Minev, and Jie Shen. An overview of projection methods for incompressible flows. *Comput. Methods Appl. Mech. Engrg*, 196:6011–6045, 2006.

[49] Georg Hager, Gabriele Jost, and Rolf Rabenseifner. Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Proceedings of the Cray Users Group Conference*, May 2009.

[50] Steven R. Hanna, Michael J. Brown, Fernando E. Camelli, Stevens T. Chan, William J. Coirier, Olav R. Hansen, Alan H. Huber, Sura Kim, and R. Michael Reynolds. Detailed simulations of atmospheric flow and dispersion in urban downtown areas by computational fluid dynamics (CFD) models – an application of five CFD models to Manhattan. *Bulletin of the American Meteorological Society*, 87(12):1713–1726, 2006.

[51] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 92–101, 2003.

[52] Rolf Hempel. The MPI standard for message passing. In Wolfgang Gentzsch and Uwe Harms, editors, *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1994, Munich, Germany,*

April 18-20, 1994, Proceedings, Volume II: Networking and Tools, volume 797 of *Lecture Notes in Computer Science*, pages 247–252. Springer, 1994.

[53] David S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 10, Dallas, Texas, United States, 2000. IEEE Computer Society.

[54] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, December 1952.

[55] Jay P. Hoeflinger. Extending OpenMP to clusters. Intel Corporation White Paper, 2006.

[56] HPCwire. Colfax unveils first eight Tesla GPU server. `http://www.hpcwire.com/offthewire/Colfax-Server-Features-Eight-NVIDIA-Tesla-GPUs-64090602.html`, October 2009.

[57] HPCwire. Microway 9U compact GPU cluster with octoputer. `http://www.microway.com/tesla/clusters.html`, November 2009.

[58] HPCwire. China's new Nebulae supercomputer is no. 2, right on the tail of ORNL's Jaguar. `http://www.hpcwire.com/offthewire/Chinas-New-Nebulae-Supercomputer-Is-No-2-Right-on-the-Tail-of-ORNLs-Jaguar-95258669.html`, May 2010.

[59] HPCwire. NVIDIA Tesla GPUs power world's fastest supercomputer. `http://www.hpcwire.com/offthewire/NVIDIA-Tesla-GPUs-Power-Worlds-Fastest-Supercomputer-105983244.html`, October 2010.

[60] HPCwire. PGI to develop compiler based on NVIDIA CUDA C architecture for x86 platforms. `http://www.hpcwire.com/offthewire/PGI-to-Develop-Compiler-Based-on-NVIDIA-CUDA-C-Architecture-for-x86-Platforms-103471359.html`, September 2010.

[61] Intel. Intel microprocessor export compliance metrics. `http://www.intel.com/support/processors/sb/cs-023143.htm`, May 2009.

[62] Dana A. Jacobsen, Julien C. Thibault, and Inanc Senocak. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *48th AIAA Aerospace Science Meeting*, January 2010.

[63] Khronos OpenCL Working Group. *The OpenCL Specification: Version 1.1*. 2010.

[64] John Kim and Parviz Moin. Application of a fractional-step method to incompressible navier-stokes equations. *Journal of Computational Physics*, 59(2):308–323, 1985.

[65] Volodymyr Kindratenko, Jeremy J. Enos, Guochun Shi, Michael T. Showerman, Galen W. Arnold, John E. Stone, James C. Phillips, and Wen mei Hwu. GPU clusters for high-performance computing. In *Proceedings of the IEEE Workshop on Parallel Programming on Accelerator Clusters*, August 2009.

[66] Géraud Krawezik. Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 118–127, San Diego, California, USA, 2003. ACM.

[67] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM SIGGRAPH 2003 Papers*, pages 908–916. ACM, 2003.

[68] Pijush K. Kundu and Ira M. Cohen. *Fluid Mechanics*. Academic Press, 4th edition, 2007.

[69] Cornelius Lanczos. Solution of systems of linear equations by minimized iterations. *Journal of Research of the National Bureau of Standards*, 49:33–53, July 1952.

[70] Youquan Liu, Xuehui Liu, and Enhua Wu. Real-time 3D fluid simulation on GPU with complex obstacles. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, pages 247–256. IEEE Computer Society, 2004.

[71] Phu Luong, Clay P. Breshears, and Le N. Ly. Coastal ocean modeling of the U.S. west coast with multiblock grid and dual-level parallelism. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 9–9, New York, NY, USA, 2001. ACM.

[72] Ewing Lusk and Anthony Chan. Early experiments with the OpenMP/MPI hybrid programming model. *Lecture Notes in Computer Science*, 5004:36, 2008.

[73] Satoshi Matsuoka, Takayuki Aoki, Toshio Endo, Akira Nukada, Toshihiro Kato, and Atushi Hasegawa. Gpu accelerated computingfrom hype to mainstream, the rebirth of vector computing. *Journal of Physics: Conference Series*, 180(1):012043, 2009.

[74] Oliver A. McBryan, Paul O. Frederickson, Johannes Linden, Anton Schüller, Karl Solchenbach, Klaus Stüben, Clemens-August Thole, and Ulrich Trottenberg. Multigrid methods on parallel computers—a survey of recent developments. *IMPACT Comput. Sci. Eng.*, 3(1):1–75, 1991.

[75] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, 2009.

[76] Kengo Nakajima. Three-level hybrid vs. flat MPI on the Earth Simulator: Parallel iterative solvers for finite-element method. *Applied Numerical Mathematics*, 54(2):237–255, 2005.

[77] Kengo Nakajima and Hiroshi Okuda. Parallel iterative solvers for unstructured grids using an OpenMP/MPI hybrid programming model for the GeoFEM platform on SMP cluster architectures. In *Proceedings of the 4th International Symposium on High Performance Computing*, pages 437–448, Kansai Science City, Japan, 2002. Springer-Verlag.

[78] Paul A. Navratil. Introduction to Longhorn and GPU-based computing. In *4th Iberian Grid Infrastructure Conference*, May 2010.

[79] NCSA. Intel 64 Tesla Linux cluster Lincoln webpage. `http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/`, 2008.

[80] Bill Nitzberg and Virginia Mary Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, 1991.

[81] NVIDIA. *NVIDIA CUDA Programming Guide 3.1.1.* 2010.

[82] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[83] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[84] Suhas V. Patankar. *Numerical Heat Transfer and Fluid Flow.* Taylor & Francis, 1980.

[85] Suhas V. Patankar and D. B. Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *Int. J. Heat Mass Transfer*, 15:1787, 1972.

[86] Everett H. Phillips, Yao Zhang, Roger L. Davis, and John D. Owens. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, 2009.

[87] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant V. Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[88] James C. Phillips, John E. Stone, and Klaus Schulten. Adapting a message-driven parallel application to GPU-accelerated clusters. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[89] Achal Prabhakar and Vladimir Getov. Performance evaluation of hybrid parallel programming paradigms. In *Performance analysis and grid computing*, pages 57–76. Kluwer Academic Publishers, 2004.

[90] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 2007.

[91] Feng Qiu, Ye Zhao, Zhe Fan, Xiaoming Wei, Haik Lorenz, Jianning Wang, Suzanne Yoakum-Stover, Arie Kaufman, and Klaus Mueller. Dispersion simulation and visualization for urban security. *Visualization Conference, IEEE*, 0:553–560, 2004.

[92] Rolf Rabenseifner. Communication bandwidth of parallel programming models on hybrid architectures. In *Proceedings of the 4th International Symposium on High Performance Computing*, pages 401–412, Kansai Science City, Japan, 2002. Springer-Verlag.

[93] Rolf Rabenseifner. Hybrid parallel programming on HPC platforms. In *EWOMP '03: Proceedings of the Fifth European Workshop on OpenMP*, pages 185–194, Aachen, Germany, 2003.

[94] Hsi-Yu Schive, Chia-Hung Chien, Shing-Kwong Wong, Yu-Chih Tsai, and Tzihong Chiueh. Graphic-card cluster for astrophysics (GraCCA) – performance tests. *New Astronomy*, 13:418–435, 2008.

[95] Hsi-Yu Schive, Yu-Chih Tsai, and Tzihong Chiueh. GAMER: a GPU-accelerated adaptive mesh refinement code for astrophysics. *Astrophysical Journal Supplement Series*, 186:457–484, 2010.

[96] Ludwig Seidel. Über ein Verfahren, die Gleichungen, auf welche die Methode der kleinsten Quadrate führt, sowie lineare Gleichungen überhaupt, durch successive Annäherung auf-zulösen. In *Abhandfungen der Bayerischen Akademie der Wissenschaften*, volume 11, pages 81–108. Mathematisch-Naturwissenschaftliche Abteilung, 2009.

[97] Inanc Senocak, Julien Thibault, and Matthew Caylor. Rapid-response urban CFD simulations using a GPU computing paradigm on desktop supercomputers. In *Eighth Symposium on the Urban Environment*, Phoenix, Arizona, 10–15 January 2009.

[98] Michael Showerman, Jeremy Enos, Avneesh Pant, Volodymyr Kindratenko, Craig Steffen, Robert Pennington, and Wen-Mei Hwu. QP: A heterogeneous multi-accelerator cluster. In *Proceedings of the 10th LCD International Conference on High-Performance Clustered Computing*, Boulder, Colorado, March 10–12 2009.

[99] Vaclav Simek, Radim Dvorak, Frantisek Zboril, and Jiri Kunovsky. Towards accelerated computation of atmospheric equations using CUDA. In *UKSim 2009: 11th International Conference on Computer Modelling and Simulation*, pages 449–454, 2009.

[100] Horst Simon, Thomas Zacharia, and Rick Stevens. Modeling and simulation at the exascale for energy and the environment. Technical report, DOE ASCR Program, 2008.

[101] Joseph S. Smagorinsky. General circulation experiments with the primitive equations. I: The basic experiment. *Monthly Weather Review*, 91(3):99–164, 1963.

[102] John C. Tannehill, Dale A. Anderson, and Richard H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. Taylor & Francis, 2nd edition, 1997.

[103] The OpenMP ARB. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*. October 1997.

[104] Julien C. Thibault. Implementation of a cartesian grid incompressible Navier–Stokes solver on multi-GPU desktop platforms using CUDA. Master's thesis, Boise State University, Boise, Idaho, May 2009.

[105] Julien C. Thibault and Inanc Senocak. CUDA implementation of a Navier–Stokes solver on multi-GPU platforms for incompressible flows. In *47th AIAA Aerospace Science Meeting*, January 2009.

[106] Jonas Tölke and Manfred Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, August 2008.

[107] Ulrich Trottenberg, Cornelis Oosterlee, and Anton Schüller. *Multigrid*. Elsevier, 2001.

[108] D. C. Wan, B. S. V. Patnaik, and G. W. Wei. A new benchmark quality solution for the buoyancy-driven cavity by discrete singular convolution. *Numerical Heat Transfer, Part B: Fundamentals*, 40(3):199–228, 2001.

[109] Ye Zhao, Feng Qiu, Zhe Fan, and Arie Kaufman. Flow simulation with locally-refined LBM. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, pages 181–188, Seattle, Washington, 2007. ACM.

APPENDIX A

TIME-STEPPING METHODS

Method	Order	Stages	Registers	Description
Euler	1	1	0	Forward-Euler
AB2	2	1	1	2nd-order Adams-Bashforth
SSP22	2	2	1	2nd-order Runge-Kutta method SSP(2,2)
SSP32	2	3	1	2nd-order Runge-Kutta method SSP(3,2)
SSP42	2	4	1	2nd-order Runge-Kutta method SSP(4,2)
AB3	3	1	2	3rd-order Adams-Bashforth
RK3-Williamson	3	3	1	3rd-order Runge-Kutta method of Williamson (1980)
RK3-Wray	3	3	1	3rd-order Runge-Kutta method of Wray (1984)
RK3-Kutta	3	3	2	3rd-order Runge-Kutta method of Kutta
SSP33	3	3	1	3rd-order Runge-Kutta method SSP(3,3)
SSP43	3	3	1	3rd-order Runge-Kutta method SSP(4,3)
SSP53	3	3	1	3rd-order Runge-Kutta method SSP(5,3)
SSP63	3	3	1	3rd-order Runge-Kutta method SSP(6,3)
RK4	4	4	2	4th-order Runge-Kutta method
RK4-SHK	4	4	2	4th-order Runge-Kutta method of Sommiejer et al. (1994)
RK4-NL	4	6	1	4th-order Runge-Kutta method of Berland (2005)

Table A.1: Some of the time-stepping methods implemented and selectable. Register usage shows how many extra per-cell values of momentum, turbulence, and temperature must be kept, which has a fairly large impact on GPU memory use.

A number of methods are implemented and selectable at run-time via the configuration file, and are shown in Table A.1. Initial testing showed that the Runge-Kutta multi-stage methods allow a higher Courant-Friedrichs-Lewy (CFL) condition to be used while maintaining stability, especially the Strong Stability Preserving (SSP) methods which are designed to maximize this behavior.

APPENDIX B

TURBULENCE MODELING

If performed at a fine enough resolution, the incompressible Navier-Stokes equations (Eqs. 3.1-3.2) will capture turbulent flows [11]. This direct simulation of turbulence is called Direct Numerical Simulation. The number of grid points required scales as $Re^{9/4}$, meaning it is generally practical only for relatively small problems at small Reynolds numbers. There are numerous methods for modeling the turbulent energy that is not directly present in the equations as given. One such approach is the Large-Eddy Simulation (LES), where a subgrid-scale model is used to approximate the small-scale effects. The model implemented in this research is a simple zero-equation Smagorinsky LES model [101].

The subgrid velocity, called here the Smagorinsky eddy viscosity, is given by

$$\nu_{tur} = (C_s h)^2 \sqrt{2 S_{ij} S_{ij}}, \tag{B.1}$$

where $h = \sqrt[3]{\Delta x \Delta y \Delta z}$ is the grid scale, C_s is the Smagorinsky coefficient, typically $0.05 < C_s < 0.25$, and S_{ij} is the resolved strain rate given by

$$S_{ij} = \frac{1}{2}\left(\frac{\partial U_i}{\partial x_j} + \frac{\partial U_j}{\partial x_i}\right). \tag{B.2}$$

No near-wall corrections are applied, and a fixed coefficient C_s is used. While this

model has a number of drawbacks compared to more complex models, it is useful for some modeling and shows how LES models can fit into the framework of the multi-GPU flow solver.

If turbulence modeling is requested in the configuration file, a CUDA kernel calculates the turbulent eddy viscosity ν_{tur} for each cell before the momentum kernel runs. The momentum kernel then adds this per-cell ν_{tur} to the global kinematic viscosity ν to determine the actual value used. Care must also be taken in the momentum discretizations to ensure no simplifications are taken assuming a static ν. This approach relies on the eddy viscosity hypothesis of Boussinesq, which assumes the kinematic viscosity ν in Eq. 3.2 is the sum of a laminar and a turbulent component. By using a per-cell ν in the momentum operation, it is easy to accommodate any method which produces a ν_{tur} value without change to the rest of the model.

No validation of the turbulence model was performed. Bouffanais [13] is a recent source for validation using an $Re = 12000$ 3D lid-driven cavity. Since the turbulence model implemented for this study uses a static coefficient, it is highly unlikely to be able to completely match the dynamic Smagorinsky or dynamic mixed model implemented in Bouffanais.

APPENDIX C

RUN-TIME CONFIGURATION

```
#-------------------------------------------------------------------------
#                             Output Setup
#-------------------------------------------------------------------------

# Output formats.  One line per output desired.
# Multiple output formats can be requested (e.g. both Matrix and VTK)
#
#    Matrix  (displays values on the screen)
#    Matlab
#    Plot3D  (A standard, if verbose, format)
#    VTK     (Native format for Paraview)
#
OutputFormat  Matrix
#OutputFormat  Plot3D
OutputFormat  VTK

# If OutputPeriod is set and non-zero, will do a VTK output even N timesteps.
OutputPeriod  10000

# An optional path for the results
OutputPath    results
# Optional prefix (default is 'gin3d_soln')
OutputPrefix  soln

# Which data to output for the matrix:  u, v, w, p, phi
MatrixData  u
# Which 2D plane to output (XY, XZ, YZ) or Mesh for all
MatrixPlane  XZ
# Where in the plane to output as a percent (0.0-1.0)
MatrixSlice 0.50
```

```
#-------------------------------------------------------------------------
#                            Navier-Stokes parameters
#-------------------------------------------------------------------------
# Kinematic viscosity
Nu        0.001
# Finite difference method used
# 0.00 means CDS, 1.00 means FOU, values between are allowed
AdvectionScheme  0.00
# CFL*dz/velmax is the convective dt limit
CFL       0.40
# DTStability*dz*dz/Nu is the viscous dt limit
DTStability  0.4
# The time derivative method
#
#   Euler        (first  order forward Euler)
#   AdamsBash    (second order Adams-Bashforth, aka 'AB2')
#   RK3          (third  order Runge-Kutta, Williamson)
#   RK4          (fourth order Runge-Kutta, classical)
#
TimeMethod  AB2

# Note, if given the values for Rayleigh, Reynolds, and Prandtl, we can
# set the parameters as thus:
#
#    Nu      = (URef*LRef) / Reynolds
#    Gamma   = Nu / Prandtl
#    For Rayleigh, it is proportional to g and beta and ind. to gamma and nu

# Turbulence model: None or Smagorinsky
TurbulenceModel  None

# Cs for Smagorinsky model.  Typically 0.01 - 0.25.
# TurbCS 0.0625  # Typical Channel
TurbCS 0.18    # Typical Cavity

#-------------------------------------------------------------------------
#                            Pressure Solver Setup
#-------------------------------------------------------------------------
# Choose either Iterative or Multigrid
SolverMethod            Multigrid

# Reasonable values:
#            Solver   Jacobi     GS
```

```
#            Loops    10-40      5-30
#            Weight   1.0        1.0-1.8
Iterative_Solver        Jacobi
Iterative_Loops         20
Iterative_Weight        1.0

# Reasonable values:
#   Cycle         V, W, or W1, recommend V
#   Smoother      Jacobi or GS
#   Loops         1-10
#   Levels        2-N, recommend about 2 less than the maximum
#   SmoothIters   1-10.  Other good choices include '2 1' and '4 2'
#   JacobiWeight  0.55 - 0.95, recommend 0.86 (from Trottenberg says 0.857)
#   SORWeight     1.0 - 1.8
MG_Cycle                V
MG_Smoother             Jacobi
MG_Loops                4
MG_Levels               9
MG_Smoother_Iterations  4 4
MG_Jacobi_Weight        0.86
MG_SOR_Weight           1.00


#-------------------------------------------------------------------------
#                           Simple Mesh Definition
#-------------------------------------------------------------------------
# The N* and L* parameters are not used for complex geometry.
#
# Simple mesh size
NX 256
NY  32
NZ 256
# Physical domain size
L_X   1.0
L_Y   1.0
L_Z   1.0



#-------------------------------------------------------------------------
#                            Complex Geometry
#-------------------------------------------------------------------------
```

```
# Complex geometry for obstructions, urban area, etc.
#
#    Cylinder      a cylinder is set up, with parameters in obstacles.c
#    City          obsolete: same as "PGM okc_arena.pgm"
#    PGM <file>    reads in a PGM height map

#Obstruction Cylinder
#Obstruction City
#Obstruction PGM  dems/okc_arena.pgm

# The following are only used for PGM obstructions.

# The number of mesh points used to resolve the jagged geometry cell.
# This must be at least 2, but sometimes more are desired to better
# resolve flow around the blocky cells.  Note that each geometry cell
# will result in this many computational cells in each dimension.  So
# a 10x10x10 geometry file with 4 for the value below will use 40x40x40
# mesh cells.
#
# Can specify one number or three for different <X,Y,Z> resolution
MeshPointsPerGeometryCell 2

# The number of mesh cells to offset the geometry file to allow for boundaries.
# These are specified in map directions, not face directions.
# Note: A single mesh cell is always added to each side to take into account
# the computational boundary.

DomainOffsetEastWest      0   0
DomainOffsetNorthSouth    0   0
DomainOffsetSurfaceSky    0   0

# Typically the input files have more Z resolution than is used via the
# delta z setting.  There are two ways of getting a finer model: go into the
# the PGM file and adjust delta z (e.g. from 10m to 5m).  The second is to
# modify this parameter.  The L* value should remain constant in either case.
# Values less than 1 will coarsen, values more than 1 will make a finer mesh.
MeshHeightScaling 1.0


#-------------------------------------------------------------------------
#                            Boundary Setup
#-------------------------------------------------------------------------
```

```
#    Face type           Boundary setting
#    ------------------  ------------------------------------
#    NoSlip              velocity 0 at boundary
#    FreeSlip            velocity unchanged at boundary
#    Inlet               BC = inlet
#    Outlet              BC = interior velocity
#    ConvectiveOutlet    See Ferziger (2001)
#    Driven              BC = inlet in normal direction
#    Periodic            BC = opposite side
# # Only one each of an inlet, driven, and outlet are allowed.
#
# Looking into the cavity, top/bottom are the ceiling and floor respectively.
# The south/north are normal to the viewer with the south closest.
# The east/west are parallel to the viewer with the west to the left.
#
# Bottom is Z=0, Top   is in the Z+ direction (w component of velocity).
# South  is Y=0, North is in the Y+ direction (v component of velocity).
# West   is X=0, East  is in the X+ direction (u component of velocity).
#
#
# A typical channel setup is to have the west be the inlet and the east
# the outlet, making the flow from left to right.  It would look like:
#
#Face_West   Inlet
#Face_East   ConvectiveOutlet
#Face_South  FreeSlip
#Face_North  FreeSlip
#Face_Bottom NoSlip
#Face_Top    NoSlip

# For standard cases one can use a shortcut:
#   Channel       W Inlet, E ConvectiveOutlet, S/N FreeSlip, B/T NoSlip
#   DrivenCavity  W/E NoSlip, S/N FreeSlip, B NoSlip, T Driven
#   Cavity        W/E NoSlip, S/N FreeSlip, B/T NoSlip
#   Urban         W/E FreeSlip, S NoSlip, N FreeSlip, B Outlet, T Inlet
Boundaries DrivenCavity

# Reference velocity scales
# Note: These are signed vectors, negative is west/-/south
U_Inlet 1.0
V_Inlet 0.0
W_Inlet 0.0
# This can be used to force the vref to something other than inlet magnitude
```

```
#Velocity_Reference 1.0

# Reference Length
# This is used as the characteristic length 'L' in calculations of the
# Reynolds and Rayleigh numbers.  Leave unset if you do not know what it is.
#
# Enter either 'LX', 'LY', 'LZ', or a number.
#ReferenceLength LX

# Notes:
#    Re = (Velocity_Reference * L / Nu)
#    Ra = (g * Beta * (Tmax - Tmin) * L^3) / (Gamma * Nu)
#    Pr = (Nu * RhoInf) / Gamma
#    Gr = Ra / Pr


#-----------------------------------------------------------------------
#                             Temperature
#-----------------------------------------------------------------------

# Whether or not to solve for and output temperature
SolveTemperature  False

# Gravity magnitude
Gravity   9.801
# Thermal expansion coefficient (1/T for ideal gas)
Beta      2.87e-03
# Typically 1.0.
# Setting this to 0 will mean temperature will not drive momentum.
Rho_Infinity  1.0
# Thermal Diffusivity  (Nu / Prandtl number)
Gamma     2.856e-05

# Scalar transport for temperature
# Note that gravity is assumed to act in the negative Z direction.
# The Boussinesq approximation is included in the w-momentum equation.
Temp_West   Dirichlet    373.15
Temp_East   Dirichlet    323.15
Temp_South  Neumann      0
Temp_North  Neumann      0
Temp_Bottom Neumann      0
Temp_Top    Neumann      0
```

```
#-------------------------------------------------------------------------
#                           Reference Points
#-------------------------------------------------------------------------

ReferencePointsOut      output_points.dat
#ReferencePointsOut     WS_Locations.dat
#ReferencePointsIn      input_points.dat

# UTM (zone 11) coordinates of SW corner of DEM file in meters
#ReferencePointsOrigin  634061 4735991 939.5

# Frequency of output, in physical time.  Use 's', 'm', 'h', or 'd'
ReferencePointsOutFreq  0.5s
```

APPENDIX D

CUDA PRESSURE KERNELS

D.1 Jacobi Pressure Kernel

CUDA kernel for unweighted Jacobi solver. The bitmask `sections` allows the top edge, bottom edge, and middle section to be selected independently. The `d_B` variable allows a per-cell scaling to be given which is used for obstacles.

```
__global__ void pressure_fz(
    int const sections,
    const REAL* d_div, const REAL* d_p, REAL* d_pnew, REAL* d_f, REAL* d_B)
{
   REAL const dti = DCONSTANT_DTI;
   int const nlayers = DCONSTANT_NLAYERS;
   REAL const B_constant = 0.5f / ( dxi2 + dyi2 + dzi2 );

   unsigned int xpos = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned int ypos = blockIdx.y*blockDim.y + threadIdx.y;
   unsigned int I = DCONSTANT_PADNY*DCONSTANT_PADNX + ypos*DCONSTANT_PADNX + xpos;

   if (    (xpos == 0) || (xpos >= (DCONSTANT_NX-1))
        || (ypos == 0) || (ypos >= (DCONSTANT_NY-1))
        || (sections == 0)
      )  return;

   unsigned int kbeg, kend;
   if (sections & SECTION_BOT)  { kbeg = 0;       } else { kbeg = 1;      }
   if (sections & SECTION_TOP)  { kend = nlayers; } else { kend = nlayers-1; }
   bool do_mid = (sections & SECTION_MID);
   unsigned int k = kbeg;
```

```
   while (k < kend) {
      unsigned int base = I + k*NX*NY;

      REAL A =   (d_p[base+1    ] + d_p[base-1    ]) * dxi2
               + (d_p[base+NX   ] + d_p[base-NX   ]) * dyi2
               + (d_p[base+NX*NY] + d_p[base-NX*NY]) * dzi2;

      REAL B = (d_B == 0)  ?  B_constant  : d_B[base];

      d_pnew[base] = B * (A - dti*d_div[base]);

      if ( (!do_mid) && (k == kbeg) ) {
         k = nlayers-1;
      } else {
         k++;
      }
   }
}
```

D.2 Red-Black Gauss-Seidel Pressure Kernel

CUDA kernel for weighted Red-Black Gauss-Seidel solver. This kernel is also used as the smoother in the multigrid method. The caller is responsible for choosing the starting and ending Z layers. Alternate methods for looping through alternate colors were tried, but this simple method proved fastest.

```
__global__ void rbgsGPU(
    int const do_even,
    unsigned int nx, unsigned int ny,
    unsigned int zbeg, unsigned int zend,
    unsigned int pad_nx, unsigned int pad_ny,
    REAL wsor, REAL mdxi2, REAL mdyi2, REAL mdzi2, REAL mw,
    const REAL* percellB, const REAL* rhs, REAL* phi)
{
   unsigned int xpos = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned int ypos = blockIdx.y*blockDim.y + threadIdx.y;
   unsigned int I = (pad_ny*pad_nx)  +  ypos*pad_nx  +  xpos;
```

```
   if (    (xpos == 0) || (xpos >= (nx-1))
        || (ypos == 0) || (ypos >= (ny-1))
      )  return;

   REAL const dti = DCONSTANT_DTI;
   REAL const wB_constant = (wsor * 0.5f) / ( mdxi2 + mdyi2 + mdzi2 );
   unsigned int k;
   for (k = zbeg; k < zend; k++) {
      int eotest = (xpos + ypos + k) % 2;
      if (do_even != eotest) {
         unsigned int const Ik = I + k*pad_ny*pad_nx;

         REAL const phib = phi[Ik-pad_nx*pad_ny];
         REAL const phic = phi[Ik];
         REAL const phit = phi[Ik+pad_nx*pad_ny];
         REAL const phiw = phi[Ik-1     ];
         REAL const phie = phi[Ik+1     ];
         REAL const phin = phi[Ik+pad_nx];
         REAL const phis = phi[Ik-pad_nx];
         REAL const laplacian =   mdxi2 * (phiw + phie)
                                + mdyi2 * (phis + phin)
                                + mdzi2 * (phib + phit);

         REAL const rhsv = (rhs == 0)      ? 0.0         : rhs[Ik];
         REAL const wB   = (percellB == 0) ? wB_constant : wsor * percellB[Ik];

         phi[Ik] = (mw * phic)  +  (wB * (laplacian - dti*rhsv));
      }
   }
}
```

APPENDIX E

MULTIGRID CUDA CODE

E.1 Restriction: Laplacian Lernel

```
__global__ void LaplacianGPU(
    int const sections,
    int nx, int ny, int nz,
    int pad_nx, int pad_ny,
    REAL mdxi2, REAL mdyi2, REAL mdzi2,
    const REAL* rhs, const REAL* phi, REAL* resid)
{
   REAL const dt  = DCONSTANT_DT;
   unsigned int xpos = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned int ypos = blockIdx.y*blockDim.y + threadIdx.y;
   unsigned int I = (pad_ny*pad_nx)  +  ypos*pad_nx  +  xpos;

   if (    (xpos == 0) || (xpos >= (nx-1))
        || (ypos == 0) || (ypos >= (ny-1))
        || (sections == 0)
      )  return;

   unsigned int kbeg = (sections & SECTION_BOT)  ?  0 :    1;
   unsigned int kend = (sections & SECTION_TOP)  ? nz : nz-1;
   bool do_mid = (sections & SECTION_MID);
   unsigned int k = kbeg;

   REAL phic, phit, phib;
   phic = phi[I + (k-1)*pad_ny*pad_nx];
   phit = phi[I + (k  )*pad_ny*pad_nx];

   while (k < kend) {
      unsigned int const Ik = I + k*pad_ny*pad_nx;
```

```
      phib = phic;
      phic = phit;
      phit = phi[Ik+pad_nx*pad_ny];
      REAL const phiw = phi[Ik-1     ];
      REAL const phie = phi[Ik+1     ];
      REAL const phis = phi[Ik-pad_nx];
      REAL const phin = phi[Ik+pad_nx];
      REAL rhsv = (rhs == 0) ? 0.0 : rhs[Ik];

      REAL phiv =   (phiw - 2*phic + phie) * mdxi2
                  + (phis - 2*phic + phin) * mdyi2
                  + (phib - 2*phic + phit) * mdzi2;

      // We're making a residual, so we multiply phiv by dt.
      resid[Ik] = rhsv - dt*phiv;

      if ( (!do_mid) && (k == kbeg) ) {
         k = nz-1;
         phic = phi[I + (k-1)*pad_ny*pad_nx];
         phit = phi[I + (k  )*pad_ny*pad_nx];
      } else {
         k++;
      }
   }
}
```

E.2 Restriction: Restriction Kernel

```
#define stencil_injection(b, I, x, y, z)  b[I]

#define stencil_half(b, I, x, y, z) \
     0.1f * (4.0f*b[I] + b[I-x] + b[I+x] + b[I-y] + b[I+y] + b[I-z] + b[I+z])

#define stencil_full(b, I, x, y, z) \
     0.015625f * ( \
       8.0f * b[I] \
     + 4.0f * ( b[I-x] + b[I+x] + b[I-y] + b[I+y] + b[I-z] + b[I+z] ) \
     + 2.0f * (  b[I+y-x] + b[I+y+x] + b[I-y-x] + b[I-y+x] \
               + b[I-z-x] + b[I-z+x] + b[I-z-y] + b[I-z+y] \
```

```
                + b[I+z-x] + b[I+z+x] + b[I+z-y] + b[I+z+y]) \
     + 1.0f * (  b[I-z-y-x] + b[I-z-y+x] + b[I-z+y-x] + b[I-z+y+x] \
                + b[I+z-y-x] + b[I+z-y+x] + b[I+z+y-x] + b[I+z+y+x]) \
     )

#if   defined(MG_WEIGHTING_INJECTION)
  #define stencil(b, I, x, y, z)   stencil_injection(b, I, x, y, z)
#elif defined(MG_WEIGHTING_HALF)
  #define stencil(b, I, x, y, z)   stencil_half(b, I, x, y, z)
#elif defined(MG_WEIGHTING_FULL)
  #define stencil(b, I, x, y, z)   stencil_full(b, I, x, y, z)
#endif

__global__ void RestrictionGPU(
    int const sections,
    int nx, int ny, int nz,
    int pad_cnx, int pad_cny,
    int pad_fnx, int pad_fny,
    const REAL* residf, REAL* residc)
{
   unsigned int xpos = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned int ypos = blockIdx.y*blockDim.y + threadIdx.y;
   unsigned int Ic = (pad_cny*pad_cnx)  +    ypos*pad_cnx  +    xpos;
   unsigned int If = (pad_fny*pad_fnx)  +  2*ypos*pad_fnx  +  2*xpos;

   if (    (xpos == 0) || (xpos >= (nx-1))
        || (ypos == 0) || (ypos >= (ny-1))
        || (sections == 0)
      )  return;

   unsigned int kbeg = (sections & SECTION_BOT)  ?   0  :     1;
   unsigned int kend = (sections & SECTION_TOP)  ?  nz  :  nz-1;
   bool do_mid = (sections & SECTION_MID);
   unsigned int k = kbeg;

   while (k < kend) {
      unsigned long int Ick = Ic +   k * pad_cny * pad_cnx;
      unsigned long int Ifk = If + 2*k * pad_fny * pad_fnx;

      residc[Ick] = stencil(residf, Ifk, 1, pad_fnx, pad_fny*pad_fnx);;

      if ( (!do_mid) && (k == kbeg) ) {
         k = nz-1;
```

```
      } else {
         k++;
      }
   }
}
```

E.3 Prolongation: Prolongation Kernel

```
__global__ void prolongationGPU(
    int nx, int ny, int nz,
    int pad_cnx, int pad_cny,
    int pad_fnx, int pad_fny,
    const REAL* phi, REAL* res)
{
  // This tries hard to neither read outside the 2D coarse domain, or
  // write outside the 2D fine domain.  It always writes an even number
  // of Z layers.  This means it is essential that the ghost layer is
  // set for the input, and exists to write into for the output.
  //
  // This version is the fastest of the twelve variants I tried, with
  // a shared memory version coming close.

  // Additionally, we compute partial averages such that the Y direction
  // (north / south) is immune to rounding differences.

  unsigned int xpos = blockIdx.x*blockDim.x + threadIdx.x;
  unsigned int ypos = blockIdx.y*blockDim.y + threadIdx.y;
  unsigned int Ic = (pad_cny*pad_cnx)  +    ypos*pad_cnx  +    xpos;
  unsigned int If = (pad_fny*pad_fnx)  +  2*ypos*pad_fnx  +  2*xpos;
  unsigned int k;

  if ((xpos >= nx) || (ypos >= ny))  return;

  bool const sum_e  = (xpos < nx-1);
  bool const sum_n  = (ypos < ny-1);
  bool const sum_ne = sum_e && sum_n;
  REAL phic, phie, phin, phine, phiu, phiue, phiun, phiune;

              phiu    = phi[Ic          ];
  if (sum_e)  phiue   = phi[Ic+1        ];
```

```
if (sum_n)  phiun   = phi[Ic  +pad_cnx];
if (sum_ne) phiune  = phi[Ic+1+pad_cnx];
Ic += pad_cny*pad_cnx;  // Bump so we're reading the upper layer
for (k = 0; k < nz; k++) {
  unsigned int Ick = Ic +   k * pad_cny * pad_cnx;
  unsigned int Ifk = If + 2*k * pad_fny * pad_fnx;
  unsigned int Ifu = Ifk + pad_fny*pad_fnx;

  phic   = phiu;
  phiu   = phi[Ick];
  res[Ifk] += phic;
  res[Ifu] += 0.5f*(phic + phiu);

  if (sum_e) {
    phie   = phiue;
    phiue  = phi[Ick+1];
    REAL ave_ce   = 0.5f*(phic + phie);
    REAL ave_ue   = 0.5f*(phiu + phiue);
    res[Ifk+1] += ave_ce;
    res[Ifu+1] += 0.5f*(ave_ce + ave_ue);
  }

  if (sum_n) {
    phin   = phiun;
    phiun  = phi[Ick+pad_cnx];
    REAL ave_cn   = 0.5f*(phic + phin);
    REAL ave_un   = 0.5f*(phiu + phiun);
    res[Ifk+pad_fnx] += ave_cn;
    res[Ifu+pad_fnx] += 0.5f*(ave_cn + ave_un);
  }
  if (sum_ne) {
    phine  = phiune;
    phiune = phi[Ick+1+pad_cnx];
    REAL ave_ce    = 0.5f*(phic + phie);
    REAL ave_nne   = 0.5f*(phin + phine);
    REAL ave_ne    = 0.5f*(ave_ce + ave_nne);;
    res[Ifk+1+pad_fnx] += ave_ne;
    REAL ave_ucue  = 0.5f*(phiu + phiue);
    REAL ave_unune = 0.5f*(phiun + phiune);
    REAL ave_une   = 0.5f*(ave_ucue + ave_unune);
    res[Ifu+1+pad_fnx] += 0.5f*(ave_ne + ave_une);
  }
}
```

```
}
```

E.4 Smoother: Weighted Jacobi Kernel

```
__global__ void weightedJacobiSolverGPU(
    int const sections,
    unsigned int nx, unsigned int ny, unsigned int nz,
    unsigned int pad_nx, unsigned int pad_ny,
    REAL cc, REAL cx, REAL cy, REAL cz, REAL mw,
    const REAL* rhs, const REAL* phi, REAL* phinew)
{
   unsigned int xpos = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned int ypos = blockIdx.y*blockDim.y + threadIdx.y;
   unsigned int I = (pad_ny*pad_nx)  +  ypos*pad_nx  +  xpos;

   if (    (xpos == 0) || (xpos >= (nx-1))
        || (ypos == 0) || (ypos >= (ny-1))
      )  return;

   unsigned int kbeg = 1;
   unsigned int kend = nz-1;
   if (sections & SECTION_BOT) kbeg = 0;
   if (sections & SECTION_TOP) kend = nz;
   bool do_mid = (sections & SECTION_MID);
   unsigned int k = kbeg;

   REAL phic, phiu, phil;
   phic = phi[I + (k-1)*pad_ny*pad_nx];
   phiu = phi[I + (k  )*pad_ny*pad_nx];

   while (k < kend) {
      unsigned int Ik = I + k*pad_ny*pad_nx;
      phil = phic;
      phic = phiu;
      phiu = phi[Ik+pad_nx*pad_ny];

      REAL const phiw = phi[Ik-1     ];
      REAL const phie = phi[Ik+1     ];
      REAL const phin = phi[Ik+pad_nx];
      REAL const phis = phi[Ik-pad_nx];
```

```
      REAL const rhsv = (rhs == 0) ? 0.0 : rhs[Ik];

      phinew[Ik] =   cx*(phiw + phie)
                   + cy*(phin + phis)
                   + cz*(phiu + phil)
                   - cc*rhsv
                   + mw*phic;

      if ( (!do_mid) && (k == kbeg) ) {
         k = nz-1;
         phic = phi[I + (k-1)*pad_ny*pad_nx];
         phiu = phi[I + (k  )*pad_ny*pad_nx];
      } else {
         k++;
      }
   }
}
```

E.5 Smoother: Red-Black Gauss-Seidel Kernel

```
__global__ void rbgsGPU(
    int const do_even,
    unsigned int nx, unsigned int ny,
    unsigned int zbeg, unsigned int zend,
    unsigned int pad_nx, unsigned int pad_ny,
    REAL wsor, REAL mdxi2, REAL mdyi2, REAL mdzi2, REAL mw,
    const REAL* percellB, const REAL* rhs, REAL* phi)
{
   unsigned int xpos = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned int ypos = blockIdx.y*blockDim.y + threadIdx.y;
   unsigned int I = (pad_ny*pad_nx)  +  ypos*pad_nx  +  xpos;

   if (    (xpos == 0) || (xpos >= (nx-1))
        || (ypos == 0) || (ypos >= (ny-1))
      )  return;

   REAL const dti = DCONSTANT_DTI;
   REAL const wB_constant = (wsor * 0.5f) / ( mdxi2 + mdyi2 + mdzi2 );
```

```
   unsigned int k;
   for (k = zbeg; k < zend; k++) {
      int eotest = (xpos + ypos + k) % 2;
      if (do_even != eotest) {
         unsigned int const Ik = I + k*pad_ny*pad_nx;

         REAL const phib = phi[Ik-pad_nx*pad_ny];
         REAL const phic = phi[Ik];
         REAL const phit = phi[Ik+pad_nx*pad_ny];
         REAL const phiw = phi[Ik-1     ];
         REAL const phie = phi[Ik+1     ];
         REAL const phin = phi[Ik+pad_nx];
         REAL const phis = phi[Ik-pad_nx];
         REAL const laplacian =   mdxi2 * (phiw + phie)
                                + mdyi2 * (phis + phin)
                                + mdzi2 * (phib + phit);

         REAL const rhsv = (rhs == 0)      ? 0.0         : rhs[Ik];
         REAL const wB   = (percellB == 0) ? wB_constant : wsor * percellB[Ik];

         phi[Ik] = (mw * phic)  +  (wB * (laplacian - dti*rhsv));
      }
   }
}
```