Multi-level Parallelism with MPI and OpenACC for CFD Applications

Andrew J. McCall

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Aerospace Engineering

Christopher J. Roy, Chair
Eric de Sturler
Eric Paterson

February 24, 2017
Blacksburg, Virginia

Keywords: Multilevel parallelism, GPU, OpenACC, MPI, CFD, parallel programming

Multi-level Parallelism with MPI and OpenACC for CFD Applications

Andrew J. McCall

ABSTRACT

High-level parallel programming approaches, such as OpenACC, have recently become popular in complex fluid dynamics research since they are cross-platform and easy to implement. OpenACC is a directive-based programming model that, unlike low-level programming models, abstracts the details of implementation on the GPU. Although OpenACC generally limits the performance of the GPU, this model significantly reduces the work required to port an existing code to any accelerator platform, including GPUs. The purpose of this research is twofold: to investigate the effectiveness of OpenACC in developing a portable and maintainable GPU-accelerated code, and to determine the capability of OpenACC to accelerate large, complex programs on the GPU. In both of these studies, the OpenACC implementation is optimized and extended to a multi-GPU implementation while maintaining a unified code base. OpenACC is shown as a viable option for GPU computing with CFD problems.

In the first study, a CFD code that solves incompressible cavity flows is accelerated using OpenACC. Overlapping communication with computation improves performance for the multi-GPU implementation by up to 21%, achieving up to 400 times faster performance than a single CPU and 99% weak scalability efficiency with 32 GPUs.

The second study ports the execution of a more complex CFD research code to the GPU using OpenACC. Challenges using OpenACC with modern Fortran are discussed. Three test cases are used to evaluate performance and scalability. The multi-GPU performance using 27 GPUs is up to 100 times faster than a single CPU and maintains a weak scalability efficiency of 95%.

Multi-level Parallelism with MPI and OpenACC for CFD Applications

Andrew J. McCall

GENERAL AUDIENCE ABSTRACT

The research and analysis performed in scientific computing today produces an ever-increasing demand for faster and more energy efficient performance. Parallel computing with supercomputers that use many central processing units (CPUs) is the current standard for satisfying these demands. The use of graphics processing units (GPUs) for scientific computing applications is an emerging technology that has gained a lot of popularity in the past decade. A single GPU can distribute the computations required by a program over thousands of processing units.

This research investigates the effectiveness of a relatively new standard, called OpenACC, for offloading execution of a program to the GPU. The most widely used standards today are highly complex and require low-level, detailed knowledge of the GPU's architecture. These issues significantly reduce the maintainability and portability of a program. OpenACC does not require rewriting a program for the GPU. Instead, the developer annotates regions of code to run on the GPU and only has to denote high-level information about how to parallelize the code.

The results of this research found that even for a complex program that models air flows, using OpenACC to run the program on 27 GPUs increases performance by a factor of 100 over a single CPU and by a factor of 4 over 27 CPUs. Although higher performance is expected with other GPU programming standards, these results were accomplished with minimal change to the original program. Therefore, these results demonstrate the ability of OpenACC to improve performance while keeping the program maintainable and portable.

Dedication

To him from whom, through whom, and to whom are all things.

Acknowledgments

The amount of support I received toward the completion of this thesis is overwhelming. I am grateful for the financial support of the Air Force Office of Scientific Research (AFOSR) through the Basic Research Initiative grant, as well as from the Virginia Tech Synergistic Environments for Experimental Computing (SEEC) Center. The use of the HokieSpeed computer at Virginia Tech through the NSF grant CNS-0960081 was of immeasurable value in the completion of this thesis.

Without the guidance and support from my academic advisor, Dr. Christopher Roy, I would certainly be far from the completion of my thesis and much less satisfied with the direction of my research.

The patience, dedication, and care of my fiancée, Christina, has helped me maintain my sanity and health. She spent many late nights helping me stay focused and using her technical editing expertise to redeem my compilations of words into readable documents.

My mother, father, brothers, and family have all played a key role in helping motivate me and support me in this effort. I give particular thanks to the companionship and encouragement of my brothers and roommates as we lived together throughout my graduate career.

I thank my research lab-mates for the many late nights and long days we spent together. Without the intelligence, resourcefulness, and helpfulness of my lab-mates I would still be finding out how to open a text editor in Linux.

Without the help of the Ribbens family, I would still be laboring on my thesis. I thank them for providing me housing many days during my final semester, as I was commuting back and forth between home and Blacksburg for research meetings and to conduct research that could not be completed at home. I also thank my friends who housed me for the other trips I made to Virginia Tech.

Finally, a thank you to the countless and dear friends that made my time at Virginia Tech as a graduate student more than just survivable, but memorable and enjoyable.

Contents

List of Figures

List of Tables

Attribution

For the first manuscript, the first author (Andrew McCall) provided the primary contribution to the manuscript's research and content. The multi-CPU and multi-GPU implementations were developed by the first author and all performance and scalability results derived from these implementations were collected and analyzed by the first author. The second author (Behzad Baghapour) researched and analyzed the single GPU optimizations for the two-dimensional and three-dimensional solvers. Furthermore, visualizations of the problem solution were obtained by the second author. The final author (Christopher J. Roy) provided the guidance and feedback necessary for research development and composition of this manuscript.

For the second manuscript, the first author (Andrew McCall) provided the primary contribution to the manuscript's research and content. All research was conducted by the first author and all results were collected and analyzed by the first author. The second author (Christopher J. Roy) provided the guidance and feedback necessary for research development and composition of this manuscript.

Chapter 1

Introduction

The scientific computing community has found interest recently in the acceleration of computing performance through the use of GPUs designed for general purpose computations. The GPU provides multiple benefits over the traditional CPU for scientific computing applications, oftentimes achieving a higher performance to cost ratio and a more energy-efficient design that reduces power consumption. The design of a GPU is optimized to handle massively parallel computations using a Single Instruction, Multiple Threads (SIMT) architecture, which proves effective for accelerating the performance of many computational models that iteratively solve a set of discrete equations over discretized domains with many elements or cells. More details on the GPU's parallelism and architecture are provided in Appendix A.

The release of the OpenACC[1] standard in 2011 opened the door for high-level programming models in the GPU computing arena. Similar to the design of OpenMP, OpenACC is a directive-based programming model where the instructions for parallelizing the code are defined by annotations to the serial code base. The directive statements are effectively comments in Fortran code and pragmas in C or C++ code. This directives-based design minimizes the changes required and allows the program to easily compile for the GPU or the CPU, using the same code base.

These directive statements have two primary purposes: to dictate the loops or sections of the code to accelerate on the GPU, and to dictate the transfer of data between the GPU and the CPU. Clauses are appended to these directive statements to define the details of the parallelism and the details of the data transfer. The OpenACC programming model uses three levels of parallelism: gang, worker, and vector. These levels are synonymous to the CUDA[2] terminology of blocks, warps, and threads, respectively.

OpenACC provides the advantage of portability as the serial and GPU-accelerated versions of the code remain within a unified code base. Furthermore, the annotations are abstracted enough from the low-level implementation details to offload acceleration to any accelerator architecture with only adding directive statements to the code. OpenACC also provides the advantage of maintainability as the code does not have to be rewritten with updates in the GPU architecture or the emergence of new platforms; this work will ideally be handled by the compiler. In addition, minimal modification from the CPU version of the code enhances maintainability as the code structure is more easily understood and less complex. In contrast, low-level programming models such as CUDA[2] or OpenCL[3] suffer from poor portability and maintainability, as complex, hardware-specific implementation details limit these qualities of the code. Furthermore, offloading computation to the GPU often proves intractable for complex and large programs, since using these models would require rewriting significant portions of the code.

The drawback to using OpenACC is the limitations on performance. Due to the very abstraction that enhances the programmability, portability, and maintainability of this model, the compiler is unable to fully take advantage of the underlying hardware architecture to accelerate performance. Therefore, an analysis of the performance obtained using OpenACC is necessary to determine the viability of this model for the problem of interest.

Computational fluid dynamics (CFD) is a branch of scientific computing that has found benefit in the use of GPU computing to accelerate compute performance. Research in OpenACC acceleration for CFD includes the work of Markidis et al. [4] in spectral methods for the sim-

ulation of 3D incompressible flows and the work of Xia et al. [5] in higher order discontinuous Galerkin methods for 3D compressible flows on hybrid unstructured grids. Others that have investigated the use of OpenACC with CFD include Chrust[6], Luo and co-workers [7, 8], Hoshino et al.[9], Corrigan and co-workers [10, 11], and van Werkhoven and Hijma[12]. This thesis investigates the viability of using OpenACC for the solution of multiple CFD solvers.

This thesis contains two primary sections. First, a solver for incompressible lid-driven and lid-driven cavity flows is accelerated using OpenACC. This solver uses an artificial compressibility method and a uniform two-dimensional or three-dimensional grid. The implementation is extended to a multi-CPU and multi-GPU code with the use of MPI and a one-dimensional domain decomposition.

Second, a large and complex CFD solver named SENSEI is accelerated with OpenACC. This code is a second order, finite volume CFD solver for the Euler and Navier-Stokes equations on structured, curvilinear, multi-block grids. MPI is also used to implement a multi-CPU and multi-GPU version of SENSEI and the best domain decomposition is automatically determined at runtime to make use of the available processes. More details of the MPI parallel implementation of SENSEI and the domain decomposition algorithm are discussed in Appendix B.

The performance and scalability for both of these solvers are presented and discussed.

References

[1] OpenACC-standard. *OpenACC Home*. `http://www.openacc.org`.

[2] NVIDIA. *CUDA*. `http://www.nvidia.com/object/cuda_home_new.html`.

[3] The Khronos Group. *The open standard for parallel programming of heterogeneous systems*. `https://www.khronos.org/opencl/`.

[4] S. Markidis et al. "OpenACC acceleration of the Nek5000 spectral element code". In: *International Journal of High Performance Computing Applications* 29.3 (Mar.

2015), pp. 311–319. ISSN: 1741-2846. DOI: 10.1177/1094342015576846. URL: http://dx.doi.org/10.1177/1094342015576846.

[5] Y. Xia et al. "OpenACC acceleration of an unstructured CFD solver based on a reconstructed discontinuous Galerkin method for compressible flows". In: *Int. J. Numer. Meth. Fluids* 78.3 (Feb. 2015), pp. 123–139. ISSN: 0271-2091. DOI: 10.1002/fld.4009. URL: http://dx.doi.org/10.1002/fld.4009.

[6] M. Chrust, E. Laurendeau, and L. Ostiguy. "Accelerating low-fidelity aerodynamic codes on multi- and many-core architectures". In: *The Journal of Supercomputing* 71.9 (May 2015), pp. 3456–3481. ISSN: 1573-0484. DOI: 10.1007/s11227-015-1444-6. URL: http://dx.doi.org/10.1007/s11227-015-1444-6.

[7] L. Luo et al. "GPU Port of A Parallel Incompressible Navier-Stokes Solver based on OpenACC and MVAPICH2". In: *7th AIAA Theoretical Fluid Mechanics Conference*. American Institute of Aeronautics and Astronautics (AIAA), June 2014. DOI: 10.2514/6.2014-3083. URL: http://dx.doi.org/10.2514/6.2014-3083.

[8] L. Luo, J. R. Edwards, and H. Luo. "Performance Assessment of Multi-block LES Simulations using Directive-based GPU Computation in a Cluster Environment". In: *52nd Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics (AIAA), Jan. 2014. DOI: 10.2514/6.2014-1130. URL: http://dx.doi.org/10.2514/6.2014-1130.

[9] T. Hoshino et al. "CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application". In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. Institute of Electrical and Electronics Engineers (IEEE), May 2013. DOI: 10.1109/ccgrid.2013.12. URL: http://dx.doi.org/10.1109/CCGrid.2013.12.

[10] A. Corrigan et al. "Semi-automatic porting of a large-scale Fortran CFD code to GPUs". In: *International Journal for Numerical Methods in Fluids* 69.2 (May 2011), pp. 314–331. DOI: 10.1002/fld.2560. URL: http://dx.doi.org/10.1002/fld.2560.

[11] A. Corrigan and R. Lohner. "Semi-automatic porting of a large-scale CFD code to multi-graphics processing unit clusters". In: *International Journal for Numerical Methods in Fluids* 69.11 (Aug. 2011), pp. 1786–1796. DOI: 10.1002/fld.2664. URL: http://dx.doi.org/10.1002/fld.2664.

[12] B. van Werkhoven and P. Hijma. "An Integrated Approach to Porting Large Scientific Applications to GPUs". In: *2015 IEEE 11th International Conference on e-Science*. Institute of Electrical and Electronics Engineers (IEEE), Aug. 2015. DOI: 10.1109/escience.2015.23. URL: http://dx.doi.org/10.1109/eScience.2015.23.

Chapter 2

An Efficient Directive-Based Multi-GPU Implementation of Computational Fluid Dynamics on Heterogeneous Platforms

Andrew McCall, Behzad Baghapour, and Christopher J. Roy
Department of Aerospace and Ocean Engineering, Virginia Tech, Blacksburg, VA 24061, USA

Attribution

The first author (Andrew McCall) provided the primary contribution to the manuscript's research and content. The multi-CPU and multi-GPU implementations were developed by the first author and all performance and scalability results derived from these implementations were collected and analyzed by the first author. The second author (Behzad Baghapour) researched and analyzed the single GPU optimizations for the two-dimensional and three-dimensional solvers. Furthermore, visualizations of the problem solution were obtained by the second author. The final author (Christopher J. Roy) provided the guidance and feedback necessary for research development and composition of this manuscript.

Abstract

OpenACC is a high-level directive-based parallel library for offloading program execution onto a graphics processing unit (GPU). The more portable directive-based model stands in contrast with the detailed implementation model of CUDA and OpenCL that suffers from poor portability and maintainability with changes in the accelerator hardware. In this paper, we explore the capability of OpenACC to efficiently accelerate the base serial CPU version of a CFD code, achieving a portable code with good computational performance on the GPU. An Artificial Compressibility Method (ACM) is used for studying steady-state incompressible 2D and 3D heat and fluid flows. We study multiple optimizations to improve GPU performance, accounting for the limited on-chip memory. The code is extended to a multi-GPU implementation, evaluating performance, scalability, and the effect that overlapping the communication between GPUs with computation has on these metrics. Memory management and loop scheduling considerations result in approximately 20% speedup in computing performance from the baseline GPU implementation. A single-GPU performance analysis demonstrates that a single NVIDIA Tesla C2075 GPU exhibits comparable speed to 8 cores on a dual-socket Xeon E5-2687W processor workstation. A multi-GPU performance analysis is performed using NVIDIA Tesla M2050 GPUs. Performance and scalability improvements are achieved by overlapping communication between GPUs with computation. The multi-GPU implementation with 32 GPUs attains a performance up to 400 times faster than a single core of a Xeon E5645 processor, 17 times faster than a single GPU, and 12 times faster than 32 CPUs using an MPI multi-CPU implementation. A weak scalability analysis demonstrates up to 99% efficiency with 32 GPUs.

2.1 Introduction

In recent years, the scientific computing community has seen a growing paradigm shift toward the use of general-purpose graphics processing units (GPGPUs) for accelerating high performance computation. This computing paradigm achieves a high throughput dependent upon a much larger number of computing threads with lower processing rates to accelerate the computation through very fine-grained parallelism. Conversely, the standard CPU computing paradigm parallelizes the execution of a program on a more limited number of computing cores with higher clock rates. The use of GPUs for scientific computing applications proves to be a highly energy- and cost-efficient solution compared to the standard CPU computing paradigm [1].

The release of CUDA [2] in 2006 as a programming model for interfacing with NVIDIA GPUs marked the beginning of widespread extension of GPUs to more general scientific computing purposes. OpenCL [3] followed shortly after in 2009, providing a much more portable interface for GPU computing by no longer restricting the GPGPU programming model to NVIDIA GPUs. Fundamentally, the designs of these models center around offloading a kernel code to run on the target GPU and transferring data from the host CPU to the target GPU's memory. Both of these programming models have found success in improving the performance of software in high performance computation, including CFD applications [4, 5, 6, 7, 8, 9, 10], and today both models still provide a basis for interfacing with GPGPUs.

Due to the ever-growing need for computing resources, interest quickly grew in multi-GPU software implementations that offloaded computation to multiple GPUs simultaneously. A practical solution that works for a wide range of computer hardware is the use of hybrid parallelism for handling the communication of multiple host CPUs with their target GPUs. Within a single node, where memory is shared between the CPUs, multi-GPU implementations have been accomplished through multi-threading using the OpenMP [11] standard [4, 6]. This approach requires a careful asynchronous implementation to avoid data race conditions within the shared memory of the CPUs. The use of MPI [12] for multi-GPU implementations also has found popularity [4, 5, 6] in recent years. MPI allows software to take advantage of the computer cluster architecture by using GPUs across multiple nodes. However, this requires additional explicit communication of data between the host CPUs for problems that are not embarrassingly parallel.

For all that CUDA and OpenCL provide in capability for GPU computing, their low-level interfaces require a detailed understanding of the specific GPU's architecture. These low-level interfaces limit portability of a specific implementation to different hardware and severely restrict the maintainability of these implementations. Furthermore, these low-level interfaces require complex implementations of algorithms to utilize the GPU's architecture, making the readability of these programs esoteric even within the scientific computing community.

These issues have driven the development of high-level programming models that provide a degree of separation between the GPU architecture and the software implementation. One such model is OpenACC[13], released in 2011, which imitates OpenMP's use of directive statements to delineate sections of code to parallelize. Each directive statement has accompanying clauses to specify the details of the parallel implementation. This abstracted parallelization model provides the benefit of minimal modification to the original code base. Furthermore, by leaving parallelization details for the compiler to handle, the program's modularity, ease of implementation, maintainability, and clarity are greatly enhanced. As with OpenMP, this enhancement in modularity allows for incremental parallelization of a serial program. For the reader's benefit, another directive-based high-level interface that has received less attention in scientific computing is HMPP [14]; however, the reader is left to explore this programming model independently of this paper.

OpenACC has found applications in scientific computing for areas such as Monte Carlo simulations [15], image processing [16], electromagnetics [17], and CFD [18, 19, 20, 21, 22, 23, 24]. Research in OpenACC acceleration for CFD includes the work of Markidis et al. [18] in spectral methods for the simulation of 3D incompressible flows and the work of Xia et al. [24] in higher order discontinuous Galerkin methods for 3D compressible flows on hybrid unstructured grids. This research even extends to the GPU acceleration of vortex-lattice methods [20]. The CFD solver for incompressible flows analyzed in this paper is based on the same implementation used by Pickering et al. [23] to study optimization and performance tuning of an OpenACC accelerated case study.

Although OpenACC provides great improvements to the modularity, ease of implementation, maintainability, and clarity of software that uses GPU acceleration, these benefits come at the cost of performance. The performance results are expected to vary depending upon the problem's computational numerics and the computer's architecture; however, results demonstrate a general degradation in performance from analogous implementations using either CUDA [25] or OpenCL [26]. This degradation is due to potential optimizations that the compiler is unable to identify independently. Therefore, the reader must consider these trade-

offs to evaluate whether a high- or low-level programming model would be more appropriate to implement for a given problem.

Just as with CUDA and OpenCL, hybrid parallel implementations are used for multi-GPU computing with OpenACC [27, 18, 17, 19, 22]. In the work of Hart et al. [27], the communication between CPUs is abstracted through the use of Fortran coarrays to store the decomposed data on the CPU memory. These coarrays greatly simplify the implementation required for communication of data between processors. The results of Hart's paper demonstrate an increase in the performance of the OpenACC multi-GPU implementation over a hybrid MPI+OpenMP multi-CPU implementation. Hart et al. [27] also explores the use of asynchronous communication and computation with moderate success, only achieving 5% to 10% increase in performance. The GPUDirect capability is used in Otten et al. [17] to bypass some of the overhead due to frequent communication between CPUs and GPUs, allowing MPI calls to perform communications directly between GPUs. Although not available for all systems, the use of GPUDirect was found to reduce the overhead of using CPUs as the intermediary for data transfer, improving speedup performance by as much as 12%. Furthermore, some proposals have been made for extensions to the OpenACC library that would allow for more efficient multi-GPU implementations due to direct GPU - GPU communication without relying on hybrid parallelism. [19, 22]

In the context of this survey of GPU computing, this paper intends to demonstrate the ease of implementation when using OpenACC in transitioning from a serial CPU code to a GPU accelerated code. This analysis is performed for an incompressible flow solver. First, we evaluate multiple implementation considerations for improving the performance of OpenACC accelerated computations for a single-GPU implementation. These considerations include thread mapping, register spilling, loop fission, loop collapsing, and asynchronous computation. Additionally, a hybrid MPI+OpenACC implementation is used to demonstrate the scalability of a multi-GPU implementation for finite difference CFD methods on a computer cluster. The effect of overlapping communication between GPUs and computation on the solver's performance and scalability is investigated.

The remainder of this paper is outlined as follows. Section 2.2 covers the background information on the OpenACC programming model and the CFD solvers used for this analysis. Section 2.3 discusses the single-GPU, multi-CPU, and multi-GPU implementations developed for this analysis. Results of this analysis are presented and discussed in Section 2.4 with conclusions stated in Section 2.5 and future work discussed in Section 2.6.

2.2 Background

2.2.1 OpenACC

The offloading of regions of a code to a GPU is performed by using parallelism directives in OpenACC. Data directives are defined to transfer data from the host (CPU) to the device (GPU). Together, these directives abstract the parallel implementation ported to the GPU, leaving low-level, hardware-dependent implementation details for the compiler to handle. Finer-grained management of data transfer and parallelism is possible through use of clauses defined for each data directive. To control the details of parallelism, such as the level of granularity, variable sharing and privatization, and variable reduction, different clauses are defined for each parallelism directive. The OpenACC programming model uses three levels of parallelism: gang, worker, and vector. These levels are synonymous to the CUDA terminology of blocks, warps, and threads, respectively.

These directive clauses allow the programmer to control the detail of the parallel implementation defined within the code with minimal additional effort. Leaving the hardware-dependent implementation details for the compiler to handle provides an easily portable code base since the same software may be compiled for many different accelerator architectures without modification. Of equal concern, the directive-based model provides better maintainability of the code base with a shorter development cycle and a longer lifespan. In contrast, CUDA and OpenCL require the programmer to define all low-level acceleration implementation details

up-front. This detailed implementation model requires that a code accelerated using CUDA or OpenCL must undergo significant modification with change in the accelerator hardware.

The recently added capabilities of OpenACC 2.0 have resolved some previous limitations in the first release of the standard. [28] It is worth mentioning, at this point, some of the important improvements in the new edition. Calling a device function inside the accelerator region was only possible for inline functions in OpenACC 1.0 [29], which can be handled in the new edition by the "routine" clause. Unstructured data regions help the programmer to efficiently handle frequent use of the PCI-e terminal to transfer data across multiple accelerator regions of the code. For the use of object-oriented programming practices, these regions are helpful in transferring user-defined structures. The OpenACC 2.0 standard enhances data management at runtime through the `acc_map_data` function and simplifies the means for direct access to the host or device copy of an OpenACC variable. Moreover, additional routines have been introduced for interoperating with other GPU compilers such as CUDA and OpenCL. Atomic directives for preventing data race conditions and the tile clause for addition optimizations in parallel directives are some other new features.

2.2.2 Solution Methodology

The steady-state solution of the incompressible heat and fluid flow equations for a lid-driven cavity (LDC) and for a buoyancy-driven cavity (BDC) problem is considered using an artificial compressibility method proposed by Chorin [30]. The governing equations for incompressible, viscous flow with heat transfer can be considered as follows:

$$\begin{aligned} \nabla \cdot \vec{V} &= 0 \\ \frac{\partial \vec{V}}{\partial t} + \vec{V} \cdot \nabla \vec{V} &= -\frac{1}{\rho} P + \nu \nabla^2 \vec{V} + \frac{1}{\rho} \vec{F}_v \\ \frac{\partial T}{\partial t} + \vec{V} \cdot \nabla T &= \alpha \nabla^2 T \end{aligned} \tag{2.1}$$

In Equation (2.1), $\nu = \mu/\rho$ is the kinematic viscosity of the fluid and $\alpha = k/\rho c$ is the thermal dissipation rate where k and c are the thermal conductivity and capacity (at constant pressure) of the fluid, respectively. The term $\vec{F}_v$ represents the volumetric force. Considering buoyancy-driven flow, the volumetric force is $\vec{F}_v = \mathrm{g}(\rho - \rho_\infty)\vec{e}_\mathrm{g}$, where g is the local gravitational acceleration and $\vec{e}_\mathrm{g}$ points in the opposite direction of gravity. When applying the Boussinesq approximation to the volumetric force $\rho - \rho_\infty = \rho_\infty \sigma(T - T_\infty)$ for small temperature differences, the momentum equation couples with the energy equation and the overall system of equations can be solved simultaneously through time. The subscript ∞ refers to a reference value for a primitive variable. In the Boussinesq approximation, σ is the thermal expansion of the fluid [31]. The lid-driven cavity problem uses the same governing equations, with omission of the buoyancy term and the energy equation.

Adding a pseudo-time derivative pressure term in the divergence-free continuity equation transfers the set of above equations to a hyperbolic system solvable with an explicit time discretization method. In this study, a second-order, central, finite difference scheme is used for spatial discretization and, since the steady-state solution is desired, a simple first-order Euler scheme is considered for time advancement. To increase the stability of ACM, a fourth-order numerical damping term is added to the continuity equation and is discretized with a fourth-order, central, finite difference scheme. The modified governing equations are presented as follows:

$$\begin{aligned} \frac{1}{\beta^2}\frac{\partial P}{\partial t} + \rho \nabla \cdot \vec{V} &= \epsilon_j \frac{\partial^4 P}{\partial x_j^4} \\ \frac{\partial \vec{V}}{\partial t} + \vec{V} \cdot \nabla \vec{V} &= -\frac{1}{\rho} P + \nu \nabla^2 \vec{V} + \mathrm{g}\sigma(T - T_\infty)\vec{e}_\mathrm{g} \\ \frac{\partial T}{\partial t} + \vec{V} \cdot \nabla T &= \alpha \nabla^2 T \end{aligned} \tag{2.2}$$

In the above equations, ϵ is the numerical dissipation coefficient, calculated as $\epsilon_j = \lambda_j \Delta x_j C_j$ for the j-th direction, and C_j is the tuning parameter to adjust the artificial viscosity applied to each spatial dimension (typically ~0.01). In addition, β is an artificial compressibility

(AC) parameter calculated using the local velocity magnitude u_{loc} along with a user-defined parameter u_{ref} as $\beta^2 = \max(u_{\text{loc}}^2, r_k u_{\text{ref}}^2)$. [23]

Moreover, this discretization uses a uniform grid spacing of Δx_j in the j-direction. In addition, λ_j is the maximum eigenvalue of the system of Equation (2.1) in the j-direction, computed as follows:

$$\lambda_j = \frac{1}{2}\left(|u_j| + \sqrt{u_j^2 + 4\beta^2}\right) \tag{2.3}$$

where u_j is the j-th component of the velocity field. Both solvers implement a Jacobi-like iteration scheme. Although a Gauss-Seidel iteration scheme provides faster convergence than a Jacobi iteration scheme, the Jacobi method lends itself to more effective parallelism.

2.3 Porting to the GPU

The overall computational procedure for an explicit forward-time-central-space (FTSC) ACM can be itemized as follows:

1. Extrapolate pressure on the domain boundaries
2. Calculate the artificial compressibility and numerical damping terms
3. Calculate the residuals at grid points
4. Update the solution field (primitive variables)
5. Rescale the pressure solution to a set value at the cavity center

The overall procedure that a programmer uses to offload computation in the base CFD code from the CPU to the GPU is shown in Figure 1. As seen in this figure, little effort is required to develop a functional, parallelized version of the CFD solver.

To maintain fourth-order accuracy for the numerical damping term, a 9-point stencil for the 2D problem and a 13-point stencil for the 3D problem is required for the pressure solution.

All other terms require a 5-point stencil for 2D solutions and a 7-point stencil for 3D solutions. Table 2.1 shows the memory transfer and computational operations per grid node for the 2D and 3D problems.

```
----------------------------------------
OpenACC annotation of a typical CFD code
----------------------------------------
1) Allocate memory on CPU
2) Initialize the solution
3) Transfer memory from CPU to GPU
!$acc data copy(A,B,...) [clauses]
4) Begin the iteration loop (Accelerator region)
t = 0
!$acc kernels present(A,B,...) [clauses]
do while (t < t_end)
  ...
  !$acc loop [clauses]
  do j = 1,n
    do i = 1,n
      ...
    end do
  end do
  t = t + dt
end do
5) End of iteration loop
!$acc end kernels
6) Transfer data from GPU to CPU
!$acc end data
7) Post-process data on CPU
```

Figure 2.1: Pseudo-code for accelerating the base CFD code with a GPU using OpenACC.

Data caching for the reused memory between adjacent nodes is useful for reducing global memory loads. However, transferring data from global to local memory outperforms the global memory based access when a large number of mathematical operations are considered for the cached data. OpenACC has an ability to check if such automatic caching is beneficial for the calculations local to the loop and seeks to perform an implicit caching to increase performance. Explicit data caching is also available in OpenACC by adding cache clauses to the accelerator regions.

Table 2.1: Detailed memory transfer and mathematical operations per grid node for different problems.

	2D LDC	**3D LDC**	**3D BDC**
Data-solution loads	19	34	41
Single-precision memory transfer (bytes)	76	136	164
Double-precision memory transfer (bytes)	152	272	328
Number of floating point operations	130	217	257

2.3.1 OpenACC

For optimizing the computations performed on the GPU using OpenACC, the following items are considered:

- **Thread mapping:** To achieve an optimal thread mapping for parallelizing the loops with OpenACC, different arrangements of gang-vector sizes are examined for the maximum number of grid points in each direction. For the 2D problem, the best thread mapping arrangement obtained is 16×8 (or 32×4 in some cases) which implies a 2D vector of parallel threads works on a chunk of the domain with length 16 in the first Cartesian direction and length 8 in the second direction. It is worth mentioning that the new "tile" mapping feature in OpenACC 2.0 is also examined; however, even the best tiling configuration results in lower performance than the manually achieved 16×8 pattern for thread mapping.

- **Register pressure:** Another issue that often arises with complex CFD calculations is the shortage of on-chip memory. Although loop fusion helps the compiler at runtime to execute the code more efficiently, unifying the loops without paying careful attention to the register resources available for each block of threads in the GPU may lead to high register pressure or even register spilling. In the worst case, when the provided amount of registers is not sufficient to handle the operations, the excess required memory will be transferred from the register to local memory with a very low clock-rate. In this case, the performance of the computation decreases significantly no

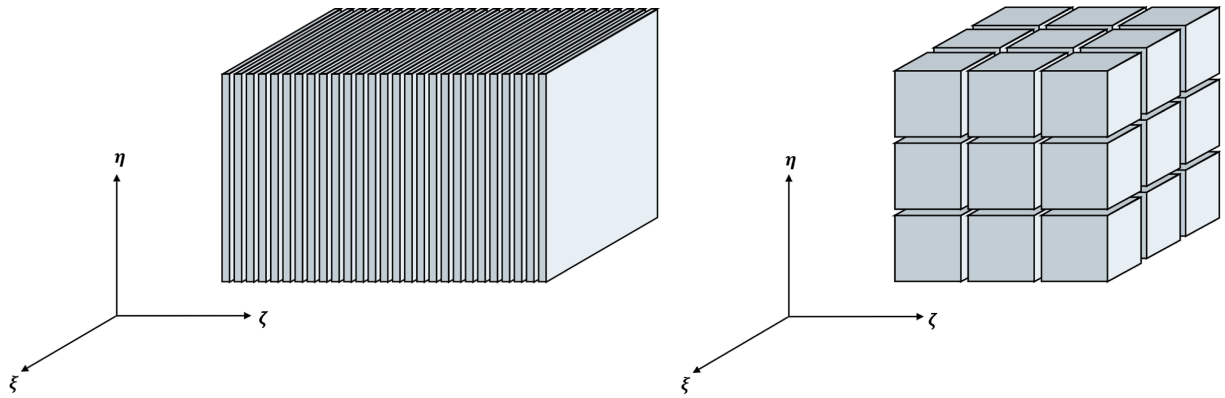
matter the grid size of the problem. Based on the experience of the authors with the ACM CFD algorithm for Cartesian meshes on the GPU, fusing the loops from the base code step-by-step could increase the overall performance. However, fusing the AC part (regarding the calculation of artificial compressibility coefficient β and the numerical damping term) with the residual calculations (including first and second derivatives of the field variables) results in register spilling. Splitting these two parts in the 2D problem avoids register spilling to the local memory and no stack frame is required for the operations. For the 3D problem, splitting the AC and residual computations helps the compiler to alleviate the register pressure, but some parts of the computation still demonstrate small spills in the register. Therefore, further investigation is necessary for optimizing the computation for 3D problems.

- **Loop collapsing:** The Cartesian-based CFD solver includes nested loops performing the operations in multiple directional sweeps. Collapsing the consecutive "do loops" on the domain by adding the collapse clause gives an opportunity for the OpenACC compiler to further optimize parallel performance. Implementing loop collapsing results in up to a 40% increase in clock-rate from the base OpenACC implementation in this research.

- **Asynchronous execution:** OpenACC supports asynchronous data movement and computations inside the accelerator region. Two asynchronous activities with different "async" argument values have the possibility to overlap. In this case the overall time to finish these two activities will be reduced, resulting in higher computation rates. Since a fully explicit time integration scheme is used in this research, there is much potential for parts of the algorithm to overlap with each other and decrease the overall time of code execution on the GPU. Results show that considering the `async` clause leads to improvement in the performance up to 50% over the performance of the code with loop collapsing optimizations.

2.3.2 MPI

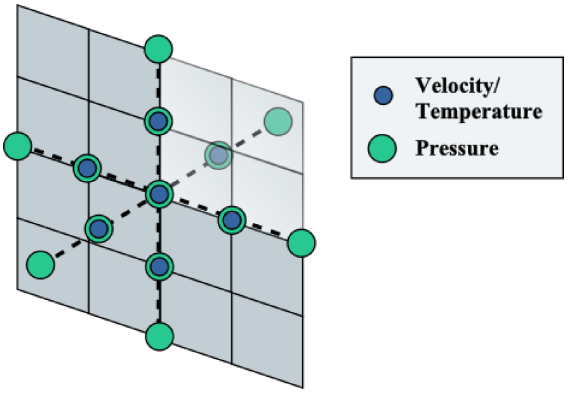
A multi-GPU implementation of the 3D BDC solver is accomplished through the use of MPI with OpenACC to take advantage of the computer cluster architecture of supercomputing resources at Virginia Tech, as discussed in Section 2.3.3. This implementation decomposes the domain across processes with separate memory spaces and communicates between these processes to exchange inter-block boundary solution data. Since the BDC solver starts with a single-block domain, the domain is evenly subdivided across the processes. Using a 1D decomposition allows for transfers of contiguous boundary data between sub-domains; however, we note that a 3D decomposition does provide advantages over a 1D decomposition. For example, a 3D decomposition may minimize total data transfer and provide a more scalable decomposition, as illustrated in Figure 2.2. However, 3D decompositions perform best in bandwidth-bound communication, not latency-bound communication. Since communication between the CPU and GPU becomes a source of high latency with highly non-contiguous data transfer, a 1D decomposition is preferred for this application. Since Fortran uses column-major ordering, the computational domain is decomposed in the ζ dimension of the computational domain as shown in Figure 2.2. This dimension corresponds to the z direction in physical space.



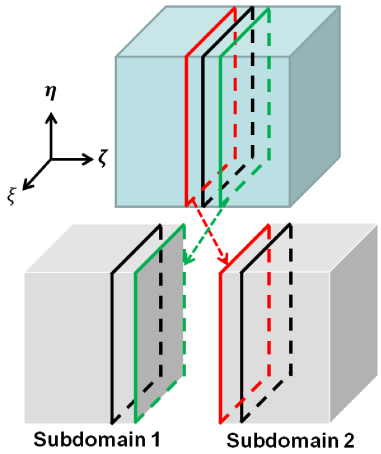
(a) 1D domain decomposition (b) 3D domain decomposition

Figure 2.2: Different domain decompositions with the same number of sub-domains in computational space. This illustrates the limited scalability of a 1D decomposition.

As discussed in Section 2.2, the 4th order numerical damping term requires a 13-point stencil using the pressure solution data, and the remaining residual calculations require a 7-point stencil for all solution variables. As shown in Figure 2.3a, two layers of ghost nodes are required for computing the numerical damping term at the inter-block boundaries. The remaining residual calculations only require transferring a single layer of ghost nodes. This inter-block boundary data exchange is illustrated in Figure 2.3b.



(a) Illustration of the pressure and other solution variable stencils.



(b) Boundary data is transferred between sub-domains to update ghost nodes.

Figure 2.3: Inter-block boundary data is exchanged for computation of the residuals near the boundary.

These values are exchanged through MPI library calls between CPUs with each iteration. Multiple implementation choices allow for improvement in the efficiency of this exchange, including non-blocking communication, 1-sided communication in the MPI-2.0 standard, GPUDirect GPU-GPU communication, and the shared memory implementation defined in the MPI-3.0 standard. This study uses non-blocking communication with `MPI_ISEND` and `MPI_IRECV` to achieve asynchronous communication, allowing data transfers to occur simultaneously. The transformation to a parallel implementation is illustrated with a pseudo-code outline in Figure 2.4a. Before running the solver, the processes must be assigned to separate GPU devices. The necessary solution and residual computation data for each sub-domain is copied to the associated GPU. To transfer boundary data between processes, the data must

first be passed back to the host CPU before processes can communicate with each other using MPI. Finally, the loops used to compute the AC terms, numerical damping terms, and residuals of the sub-domains are offloaded as kernels to the GPUs to accelerate performance.

```
! Set GPU device
call acc_set_device_num(num, acc_device_nvidia)

! Decompose domain in zeta-direction

!$acc data copy(solution,...)
do while (error > tol .and. iter < max_iter)

  ! Transfer inter-block boundary data
  !$acc update host(solution(start:end)) async(1)
  call MPI_IRECV(solution(start),...)
  !$acc wait(1)
  call MPI_ISEND(solution(start),...)

  call MPI_WAITALL(...)



  !$acc update device(solution(start:end))
  call MPI_BARRIER(...)

  ! Compute all AC, numerical damping terms


  !$acc wait

  ! Compute residuals

  ! Rescale Pressure
  call MPI_BCAST(center_pressure,...)

  ! Compute global residual norm
  call MPI_REDUCE(..., MPI_SUM, ...)
end do
!$acc end data
```

(a) Pseudo-code for the baseline parallel implementation with OpenACC.

```
! Set GPU device
call acc_set_device_num(num, acc_device_nvidia)

! Decompose domain in zeta-direction

!$acc data copy(solution,...)
do while (error > tol .and. iter < max_iter)

  ! Transfer inter-block boundary data
  !$acc update host(solution(start:end)) async(1)
  call MPI_IRECV(solution(start),...)
     !$acc wait(1)
  call MPI_ISEND(solution(start),...)

  ! Compute AC, numerical damping terms
  ! (interior nodes)

  call MPI_WAITALL(...)
  !$acc update device(solution(start:end))
  call MPI_BARRIER(...)

  ! Compute AC, numerical damping terms
  ! (remaining nodes)

  !$acc wait

  ! Compute residuals

  ! Rescale Pressure
  call MPI_BCAST(center_pressure,...)

  ! Compute global residual norm
  call MPI_REDUCE(..., MPI_SUM, ...)
end do
!$acc end data
```

(b) Pseudo-code for the parallel implementation with overlapping communication and computation.

Figure 2.4: Pseudo-code of the baseline multi-GPU implementation and the multi-GPU implementation with overlapping communication and computation. Modifications to overlap communication and computation are shown in red.

The baseline multi-CPU and multi-GPU implementations only make use of asynchronous communication; however, non-blocking communication also allows for overlap of communication with computation. The use of asynchronous execution, as discussed in Section 2.3.1,

can improve GPU performance by masking memory access latencies within the GPU. Similarly, if the inter-GPU data transfer is executed asynchronously with computations on the GPU, the parallel communication overhead may be reduced. Overlap of communication and computation is achieved through the use of non-blocking MPI communication calls between CPUs and asynchronous data transfers between the GPU and host CPU. For a multi-GPU implementation, this data transfer process consists of three steps, listed as follows:

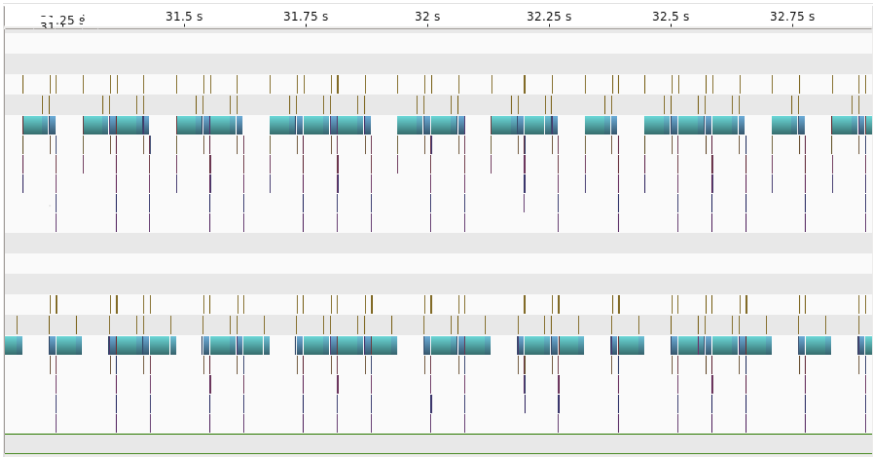
1. Transfer boundary data from the GPU to the host CPU.
2. Transfer data to the host CPU of the neighboring sub-domain.
3. Transfer the ghost node data from the host CPU to the GPU.

To study the effect of overlapping communication and computation on performance and scalability of the parallel implementation, we must identify what computations should overlap with the data transfer communication. Due to the different stencils used to calculate the artificial compressibility and numerical damping terms for nodes near domain boundaries (not inter-block boundaries), these calculations are splintered into multiple loop structures to handle interior nodes, boundary face nodes, edge nodes, and corner nodes. Conversely, since the artificial compressibility and numerical damping terms are computed separately from the residuals to reduce register pressure, the residuals of the entire domain are computed in a single nested loop structure. Therefore, a synchronization point for the GPU kernels must occur before reaching the residual calculation loop to ensure all artificial compressibility and numerical damping terms have been computed.

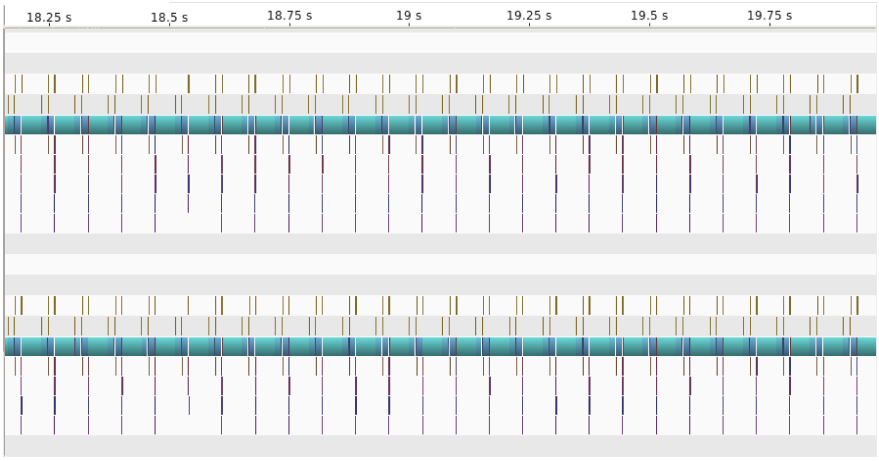
With this structure in mind, only artificial compressibility and numerical damping term computations can overlap with the inter-GPU data communication. Otherwise, the residual loop, which performs a reduction operation to calculate the residual norms, would also need to be split apart. The computational overhead and reduced maintainability of the code, induced by breaking apart the residual computation loop and separately computing the residual norms, negates the benefit of this modification. Furthermore, computation cannot overlap

with communication for nodes that are located near inter-block boundaries, on the ζ_{min} and ζ_{max} faces, and include ghost nodes being updated by the inter-GPU data communication within their stencil. Therefore, the loops for computing artificial compressibility and numerical damping terms are split to separate the computation that can and cannot overlap with communication. Although this modification introduces additional computational overhead, this modification is necessary to avoid a race condition. Figures 2.4a and 2.4b illustrate the changes made in the program structure to overlap communication with computation, where the modifications are highlighted in red.

It is important to note in Figure 2.4a the `MPI_BARRIER` call added after the inter-block ghost node data transfer has completed. Although performance is generally degraded by synchronization, careful usage of synchronization may improve performance when the processes must communicate with each other. Figure 2.5 illustrates the execution profile of GPU kernels using two GPUs over time with and without the `MPI_BARRIER` call. This execution timeline was produced using the NVIDIA Visual Profiler tool[32]. Each colored bar within the timeline represents the execution of a given GPU kernel. The white spaces between these colored bars in Figure 2.5a indicate that nothing is executing on the GPU during that timespan. This figure shows that execution of the two GPUs can become desynchronized, such that one GPU performs the residual computations for the next iteration before fully transferring ghost node data to its neighboring sub-domain. This subsequently requires the sub-domain to wait for the neighboring sub-domain to catch up in the residual computation before continuing. Figure 2.5a illustrates that this desyncronization happens every one to three iterations. However, adding the synchronization point forces both GPUs to execute close enough to synchronously to avoid this problem, as shown in Figure 2.5b, and improve performance by up to 28%. Figure 2.5b also illustrates the effect of asynchronous execution on the GPU, as up to six GPUs kernels execute simultaneously, indicated by the kernels with overlapping timespans being stacked vertically.



(a) Execution profile of two GPUs without barrier synchronization.



(b) Execution profile of two GPUs with barrier synchronization.

Figure 2.5: Synchronization calls can improve performance by preventing complete desynchronization of processes that must communicate with each other. Each colored bar represents the execution of a GPU kernel, so the white spaces between bars indicates time wasted by the GPU.

2.3.3 Computing Resources

Table 2.2 summarizes the machines used and their specifications. The single-GPU performance results for the LDC and BDC solvers are obtained using a Dell T7600 Precision workstation with dual-socket 8-core Intel Xeon E5-2687W processors, for a total of 16 cores. The workstation also contains two NVIDIA Tesla C2075 GPUs and 67 GB memory. Scala-

bility and performance analyses of the multi-GPU implementation are conducted using the HokieSpeed [33] CPU-GPU computer cluster at Virginia Tech. HokieSpeed has 204 compute nodes connected by a QDR Infiniband interconnect, each containing dual-socket 6-core Intel Xeon E5645 processors, for a total of 12 cores within each node. Every node also contains two NVIDIA Tesla M2050 GPUs and 24 GB of memory.

Table 2.2: Specifications for the hardware in the different machines used in this study. Note that two processors and two GPUs exist on each node for all machines.

Machine Name	**Workstation**	**HokieSpeed**	**NewRiver**
Memory Specifications			
Size (GB)	67	24	256
Bandwidth (GB/s)	51.2	32.0	51.2
Processor Specifications			
Model	Intel Xeon E5-2687W[34]	Intel Xeon E5645[35]	Intel Xeon E5-2680[36]
Physical Cores	8	6	12
Base Clock Rate (GHz)	3.10	2.40	2.70
Shared L3 Cache (MB)	20	12	20
GPU Specifications			
Model	NVIDIA Tesla C2075[37]	NVIDIA Tesla M2050[38]	NVIDIA Tesla K80[39]
Architecture	Fermi	Fermi	Kepler
Chip	GF110	GF100	2 × GK210
Memory Size (MB)	6144	3072	2 × 12288
Bandwidth (GB/s)	144.0	148.4	2 × 240.6
Cores (Total)	448	448	2 × 2496
Core Clock Rate (GHz)	1.150	1.150	0.560 – 0.875

The comparison of performance between the Fermi and Kepler architectures for NVIDIA Tesla GPUs makes use of the Virginia Tech NewRiver [40] computer cluster. NewRiver has 134 nodes, with 8 nodes dedicated to GPU computing. These nodes use dual-socket 12-core Intel Xeon E5-2680 processors, for a total of 24 cores within each node. Furthermore, each node contains two NVIDIA Tesla K80 GPUs and 256 GB of memory. Due to the limited availability of nodes, only the performance of a single GPU is analyzed on NewRiver to compare with the single GPU performance on the workstation and on HokieSpeed. All

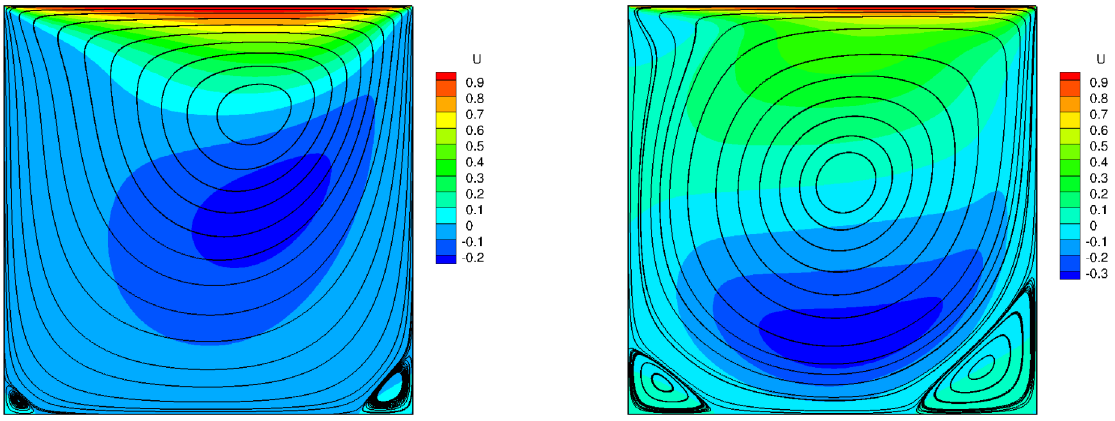
analyses use the PGI Fortran 15.7 compiler and the multi-GPU analysis on HokieSpeed makes use of the Open MPI 1.8.5 library.

2.4 Results and Discussion

This section presents the computational performance achieved by offloading computation in the CFD base code to the GPU with OpenACC. The computation grid for the single-GPU analysis varies from 128^2 to 4096^2 in 2D and from 16^3 to 256^3 in 3D simulations. This covers the range of grid sizes that can be used, limited by the maximum available amount of memory for a single NVIDIA Tesla C2075. For the multi-GPU analysis with the 3D BDC code, the computational grid varies from 32^3 to 512^3 between the performance and strong scalability analyses. Due to memory limitations, the maximum grid size computable on a single GPU on HokieSpeed is a 306^3-node domain. The weak scalability analysis maintains a fixed computational grid size for each process. The weak scalability analysis is run for multiple local grid sizes ranging from $256 \times 256 \times 64$ to $256 \times 256 \times 256$ nodes.

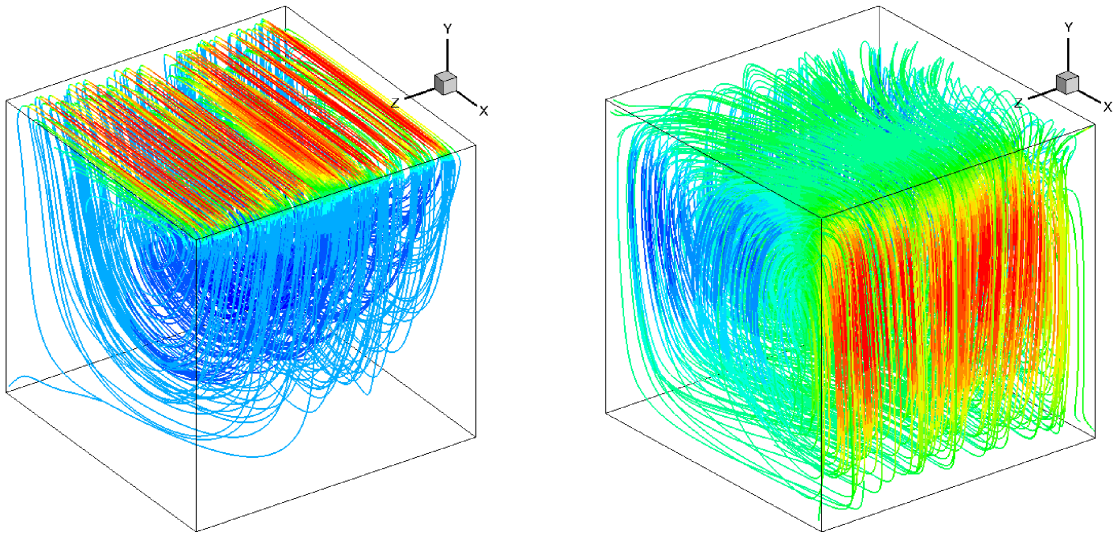
2.4.1 Problem Solution

Figures 2.6 and 2.7 show the flow fields inside the lid-driven cavity and the buoyancy-driven cavity for different Reynolds (Re) and Rayleigh (Ra) numbers. Moreover, Figure 2.8 shows the fluid velocity and temperature distribution in different slices of the 3D LDC and BDC problems. As shown in Figure 2.7, a constant velocity $u_{\text{lid}} = 1.0$ is considered at the top face of the 3D cavity normal to xz-plane. For the BDC problem, a temperature difference of $\Delta T_{\text{lid}} = 10.0$ is considered at front face of the cavity normal to the yz-plane.



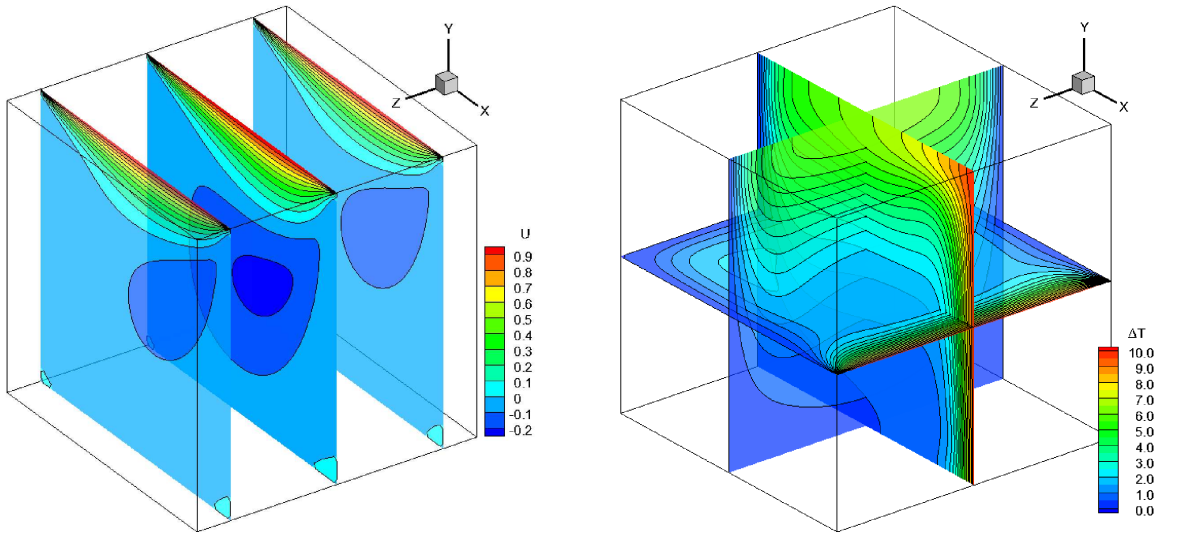
(a) LDC (Re = 100) (b) LDC (Re = 1000)

Figure 2.6: Streamlines and horizontal velocity contours of 2D lid-driven cavity flows.



(a) LDC (Re = 100) (b) BDC (Ra = 10^5)

Figure 2.7: Streamlines for lid-driven and buoyancy-driven flows in a 3D cavity.



(a) LDC (Re = 100) (b) BDC (Ra = 10^5)

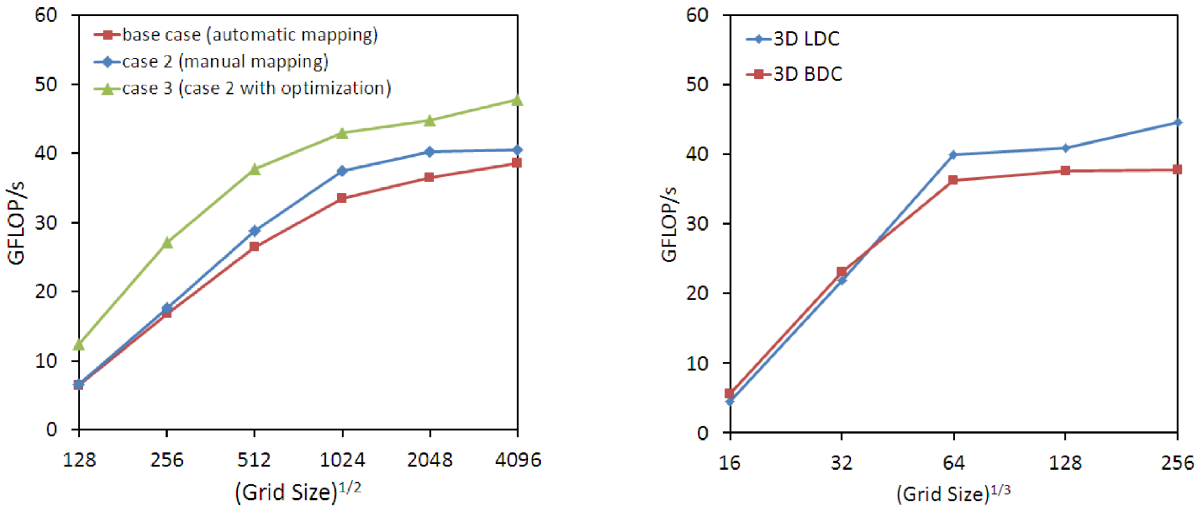
Figure 2.8: First velocity component and temperature distribution inside the 3D LDC and BDC fields.

2.4.2 OpenACC Optimization

Figure 2.9a demonstrates the achieved performance in terms of gigaflops (GFLOP/s) using OpenACC optimization strategies in comparison with the base GPU implementation. As can be seen, using an optimal thread mapping results in better performance, especially in finer grids, with respect to the automatic mapping selected by the compiler. Moreover, applying loop collapsing and asynchronous executions leads to a considerable increase in the total performance. The optimized code in this case provides up to 24% better performance than the base code.

The performance demonstrated in Figure 2.9b for 3D LDC and BDC is attained by pursuing the strategies considered for improving the performance of the 2D LDC solver in the 3D problem. According to this figure, the computational throughput of the isotherm (4 equations) and thermal (5 equations) problems deviate from each other as the grid size increases. The performance achieved with a single GPU at the maximum grid size is 44.6 GFLOP/s for the lid-driven cavity and 37.7 GFLOP/s for the buoyancy-driven cavity. Moreover, the

performance improvement with a single GPU stalls sooner in the thermal problem. This decrease in performance is due to the increased the number of governing equations, consequently increasing the total amount of required on-chip register memory. Furthermore, this issue shows a performance barrier due to register resource limitation in the GPU.



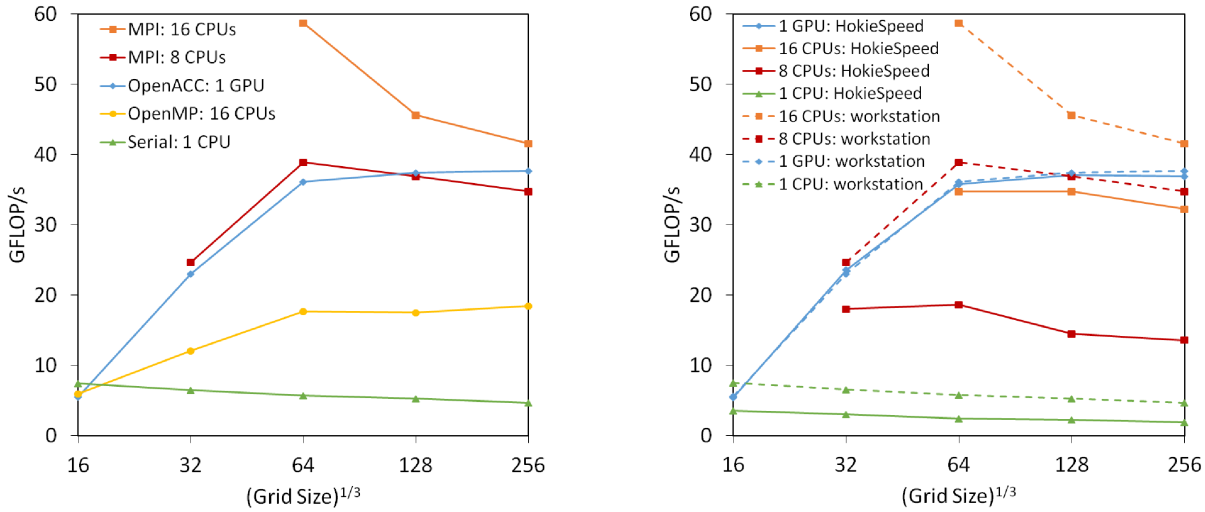
(a) Effect of optimal thread mapping and other optimizations on performance with respect to the base GPU implementation for the 2D LDC problem.

(b) Achieved computational performance for the 3D LDC and BDC problems.

Figure 2.9: Performance optimization results for the single-GPU implementation.

Figure 2.10a compares the achievable increase in performance for a single NVIDIA Tesla C2075 with respect to a single CPU and with respect to MPI and OpenMP multi-CPU implementations for the 3D BDC problem. Note that the MPI implementation is not tested for the full range of grid sizes. This is because the minimum grid dimension is restricted to 4 nodes that are not ghost nodes. This restriction prevents ghost node regions of different neighboring sub-domains from overlapping with each other, which leads to instability in the solver. According to this figure, a single GPU achieves a performance of 37.7 GFLOP/s at the maximum grid size. In comparison, the achievable performance of the multi-CPU implementations at the maximum grid size are 38.9 GFLOP/s for the MPI implementation with 8 cores and 18.4 GFLOP/s for the OpenMP implementation using 16 cores. The

performance of the single-CPU implementation is 4.66 GFLOP/s at the maximum grid size. Therefore, the single-GPU implementation provides approximately a factor of 8 improvement in performance compared to the serial CPU code, a factor of 2 improvement in performance compared to the OpenMP multi-CPU implementation using 16 cores, and an 8.2% increase in performance relative to the MPI multi-CPU version of the code using 8 cores. The multi-CPU code does provide up to an 8.7% increase in performance over the single-GPU implementation for smaller grid sizes; however, for grids larger than 64^3 nodes the multi-CPU performance degrades with increase in grid size.



(a) Comparison of the performance of a single NVIDIA Tesla C2075 GPU with single-CPU and multi-CPU implementations of the CFD code on a Dell 7600 Precision workstation.

(b) Comparison of performance on the HokieSpeed supercomputer (one CPU per socket) with performance on a Dell T7600 Precision workstation.

Figure 2.10: Performance comparisons of the single-GPU implementation for the 3D BDC problem.

As illustrated in Figure 2.10a, the performance of the single-CPU implementation steadily decreases by approximately 12% with each increase in the grid size. This effect is likely due to a greater number of cache misses as, proportionally, less of the grid fits into the cache. This effect is further demonstrated in the MPI implementation run with 8 CPUs for grid sizes ranging from 64^3 nodes to 256^3 nodes. When the MPI implementation is run with 16 CPUs a

much more significant performance degradation occurs with increase in grid size, decreasing in performance by over 22% between the 64^3-node grid and the 128^3-node grid. This more dramatic performance degradation illustrates the effect of running all cores within a socket, causing the CPUs to compete for shared cache, memory, and communication bus resources. The shared-memory parallel architecture that OpenMP uses does not seem to suffer from these same performance degradations; however, the overall performance of the OpenMP multi-CPU implementation is considerably worse than the performance of the MPI multi-CPU implementation. For a 64^3-node grid, use of a baseline OpenMP implementation in place of the optimized MPI implementation leads to a 70% decrease in performance. A more scalable OpenMP implementation is necessary to determine whether or not a shared-memory architecture similarly suffers from performance degradation due to resource contention.

Performance comparisons between HokieSpeed's computing resources and the workstation's computing resources are shown in Figure 2.10b for the 3D BDC problem. As shown, an NVIDIA Tesla M2050 in HokieSpeed produces very similar performance to an NVIDIA Tesla C2075, with less than 3% difference in performance. However, a single core of the Xeon E5645 processor in HokieSpeed exhibits 52% to 58% lower performance than a single core of the Xeon E5-2687W processor. These performance results for the single-CPU implementation lead to a reduction in performance of the MPI multi-CPU implementation on HokieSpeed. Using 8 CPUs for grid sizes ranging from 64^3 nodes to 256^3 nodes, the performance on HokieSpeed is 52% to 61% lower than the performance on the workstation. However, the performance of 16 CPUs on HokieSpeed is only 22% to 41% lower than the performance on the workstation. This discrepancy is explained by a lack of resource contention on HokieSpeed because only one CPU is run on each socket.

The GPUs on the workstation and HokieSpeed are Fermi architecture NVIDIA Tesla GPUs, released in 2011 [37, 38]. Since the release of HokieSpeed, later in 2011[41], the first Kepler architecture was released in 2012[42]. The NewRiver computing cluster at Virginia Tech was released in 2015[43] and contains 16 K80 GPUs, the latest Kepler architecture GPU, released in 2014[39]. Due to limited access to these nodes, only the performance of a single K80 GPU

is compared with the performance of the C2075 and M2050 GPUs to illustrate the potential of GPU computing with more modern GPUs. As Figure 2.11 shows, for grid sizes greater than 64^3, the K80 GPU achieves between 300% and 320% of the performance of the Fermi architecture GPUs. This performance is 14.7 to 17.9 times greater than the performance of a single CPU on NewRiver. Therefore, with access to the latest technology the reader may expect even better performance results relative to the CPU than shown in this paper.

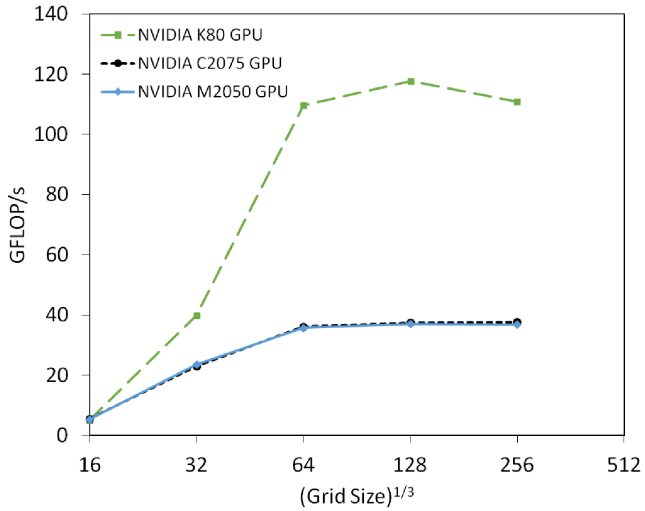


Figure 2.11: Illustration of the performance improvement of the Kepler architecture over the Fermi architecture GPUs using the 3D BDC test case.

2.4.3 Multi-GPU Analysis

This study analyzes the computational performance as well as the strong and weak scalability of the 3D BDC solver to evaluate the parallel implementations developed. The strong scalability analysis observes the speedup and efficiency of the implementation for a constant overall grid size. For a multi-CPU implementation, we measure speedup relative to a single CPU, whereas for a multi-GPU implementation, we measure speedup relative to a single GPU. Efficiency for the strong scalability analysis compares the obtained speedup with the ideal linear speedup. In the weak scalability analysis, the grid size remains constant on each process, so the problem sizes increases as processes are added. The only metric used for this

analysis is efficiency, which compares the execution time per iteration for a given number of processes with the execution time per iteration for a single process.

Baseline Implementation

With the noted improvements of a single GPU over both single and multi-CPU implementations, it is desirable to test the scalability and performance of a multi-GPU implementation in comparison with a multi-CPU implementation on the HokieSpeed cluster. This analysis is limited to the 3D BDC solver due to the similar performances between the LDC and BDC codes. Furthermore, the additional computational expense and memory cost of the BDC solver allow for a more extensive test of the multi-GPU implementation. As Figure 2.9b illustrates, the performance of a single CPU is relatively insensitive to change in the problem size, with a decrease in performance of less than 30% from the 16^3-node case. A single GPU's performance, however, increases by over 500% from the 16^3-node case. Similar results are replicated when the 3D BDC code is run on HokieSpeed, as shown in Figure 2.12. This figure additionally illustrates the overall performance of the MPI multi-CPU and MPI+OpenACC multi-GPU implementations with grid refinement when run on HokieSpeed.

As illustrated in Figure 2.12, the multi-CPU implementation demonstrates a marginal effect of problem size on the performance. The overall performance of the multi-GPU implementation not only continues to increase with grid size, but also experiences a greater sensitivity of performance to grid size with an increased number of GPUs. This performance sensitivity produces a compounding effect of increasing performance with grid size for the multi-GPU implementation. This increase in performance is bounded on a single GPU, leveling off after the 64^3 grid; however, as the number of GPUs increases, this performance bound is raised. As discussed in Section 2.4.2, this performance bound is likely due to a register resource limitation on the GPU. The 128^3 grid, run with 8 GPUs, contains the same amount of memory per GPU as the 64^3 grid on a single GPU, and yet the performance on 8 GPUs continues to increase past the 128^3 grid. This is likely due to the proportional decrease of communication

time between GPUs relative to the amount of work performed within each iteration.

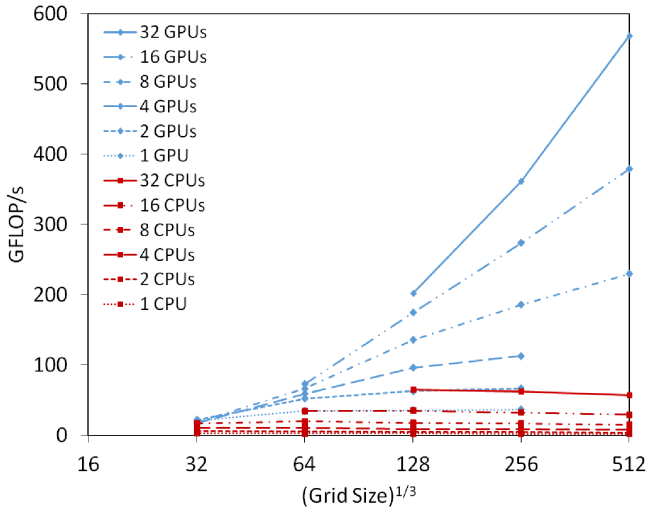
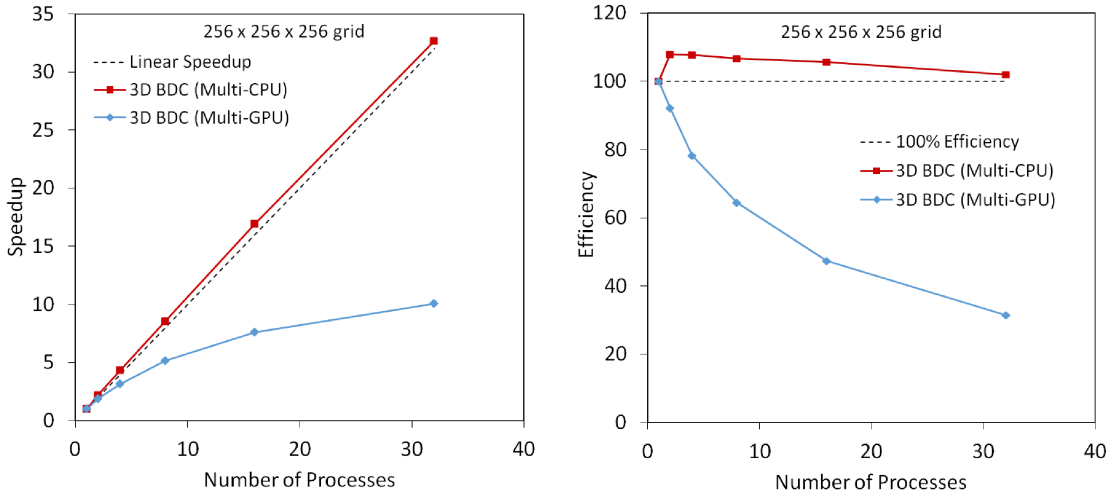


Figure 2.12: Performance of multi-CPU and multi-GPU implementations over a range of grid sizes.

For the single-GPU case, the performance levels off with a factor of 1.7 increase in the performance between the 32^3-node domain and the 256^3-node domain. However, with 8 GPUs the performance continues to increase over the range of grid sizes tested for this analysis, reaching over a factor of 12.6 increase in the performance between the 32^3-node domain and the 512^3-node domain. This behavior demonstrates that the massively parallel GPU architecture produces better performance when computing over high volumes of data, whereas the parallelism exploited in a distributed memory multi-CPU architecture produces relatively constant performance with grid size. The performance of the 512^3-node case with 32 GPUs is over 364 times faster than a single CPU run on HokieSpeed, over 15.7 times faster than a single GPU run on HokieSpeed, and 10 times faster than 32 CPUs bound to different sockets on HokieSpeed.

The speedup and efficiency results for a strong scalability test of these multi-GPU and multi-CPU implementations are illustrated in Figure 2.13. The multi-CPU implementation exhibits a speedup proportional to the number of processes used, producing a slightly superlinear speedup up to 32 processes with an efficiency of 102%. This superlinear speedup

is likely due in part to cache effects as the problem is decomposed into chunks that better fit into the cache local to each socket. For 2 or more CPUs the multi-CPU implementation exhibits an almost linear decrease in efficiency with increase in the number of processes, having a coefficient of determination of 0.991. The slope of this linear correlation indicates a loss of 1.97% in efficiency with an increase of 10 CPUs in the total process count. These results demonstrate that the multi-CPU implementation is highly scalable for lower numbers of processes. With the addition of a few OpenACC directives to accelerate the code on the GPU, the scalability performance rapidly degrades to 31.4% efficiency with 32 processes. This behavior is explained by the results previously discussed in Figure 2.12. With a strong scalability analysis, the total grid size is fixed so that as the number of processes is increased, each GPU computes on less data. Therefore, the performance of the GPU degrades because the amount of data per GPU is not maximized. Consequently, a weak scalability test with fixed grid size per GPU provides a better analysis of the effect of communication overhead incurred by transferring data between the CPU and the GPU on the implementation's scalability.



(a) Strong scalability speedup.

(b) Strong scalability efficiency.

Figure 2.13: Strong scalability results for the multi-CPU and multi-GPU implementations.

Figure 2.14 illustrates the results for the weak scalability analysis of the multi-CPU and

multi-GPU implementations on HokieSpeed. Multiple weak scalability tests were performed for different local grid sizes stored in the memory of each processing unit. Since the decomposition of the domain among processes is one-dimensional, the grid size is increased only in the z-dimension (the dimension along which the domain is decomposed) so that the memory storage pattern for data transfer of a local grid remains constant.

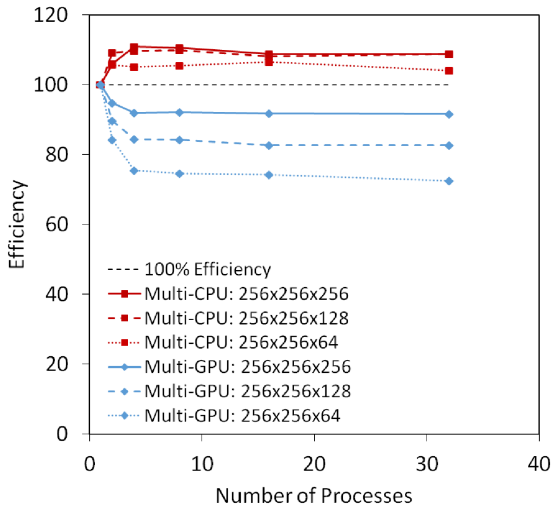


Figure 2.14: Memory-constrained weak scalability results of the multi-CPU and multi-GPU implementations for multiple fixed local memory sizes.

The multi-CPU implementation exhibits close to the same superlinear efficiency for different local grid sizes with less than 5% difference in efficiency. The efficiency of the multi-CPU implementation is also almost constant with increase in the number of processes, exhibiting only 2% difference in the efficiency when using more than 2 processes. These results indicate a scalable implementation. Since the local grid size is kept fixed for different numbers of processes, this superlinear behavior is not explained by cache effects. This superlinear behavior perhaps relates to the additional cost of boundary evaluations, such as pressure extrapolation, that are divided up between different processes when run in parallel.

Unlike the multi-CPU implementation, the efficiency of the multi-GPU implementation changes with the local grid size. For a local grid size of $256 \times 256 \times 64$, the efficiency reaches a lower bound of 74%. The efficiency for a local grid size of $256 \times 256 \times 128$ de-

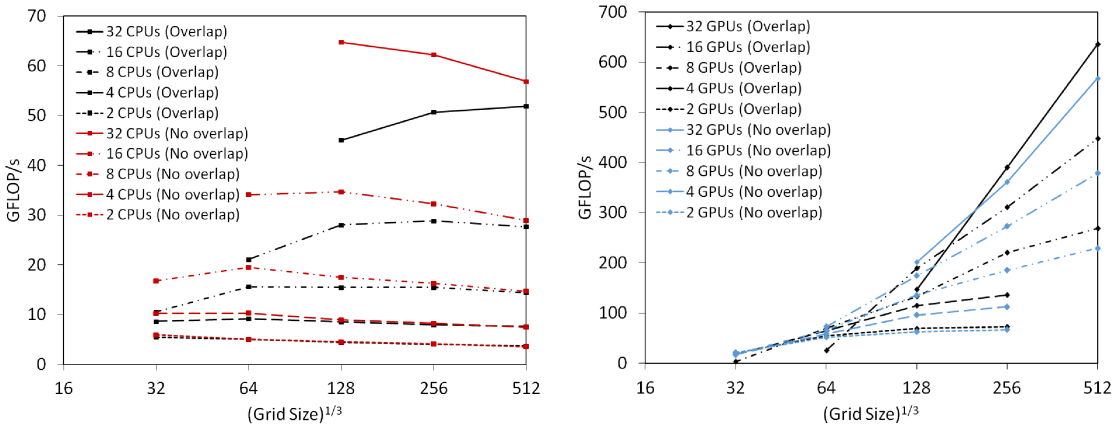
grades to a bound of 83% and for a local grid size of $256 \times 256 \times 256$ degrades to 92%. This behavior demonstrates a dependence of the efficiency on the grid size. Since the grid size is only increased in one dimension, the amount of data that each process transfers remains constant. Therefore, as the size of the local grid becomes larger the communication overhead becomes proportionally smaller relative to amount of the work performed within each iteration. Notably, this overhead has little effect on the scalability efficiency for the multi-CPU implementation, illustrating the significance of the communication overhead incurred by the transfer of data between the host CPU and the GPU. All of these tests still indicate a scalable multi-GPU solution since the efficiency remains bounded with increase in the number of GPUs. The results of Figures 2.13 and 2.14 seem to concur with the results found in Jacobsen [44], where strong scalability results for an MPI+CUDA implementation demonstrated poor scalability. Also in Jacobsen's paper, the weak scalability results for the multi-GPU implementation maintained an efficiency of approximately 80% up to 128 processes.

Overlapping Communication and Computation

Although the baseline parallel implementation has already been shown to be scalable, we seek to further optimize performance by overlapping communication of boundary data between processes with computation of the artificial compressibility and numerical damping terms. As Figure 2.14 illustrates, the overhead of data communication between a GPU and the host CPU introduces a significant penalty to the performance and scalability, so we seek to mask this overhead.

Using the implementation described in Section 2.3, the change in performance relative to the baseline parallel implementation is illustrated in Figure 2.15. Figure 2.15a illustrates that the performance of the multi-CPU implementation is consistently lower when using overlapping communication and computation. This is due to the large computational overhead relative to the communication overhead between CPUs. For large grid sizes, the performance reaches equivalence with the baseline implementation's performance, but does not exceed the baseline

implementation's performance.



(a) Multi-CPU performance. (b) Multi-GPU performance.

Figure 2.15: Multi-GPU performance improves by up to 21% for larger grid sizes with overlapping communication and computation, while multi-CPU performance decreases.

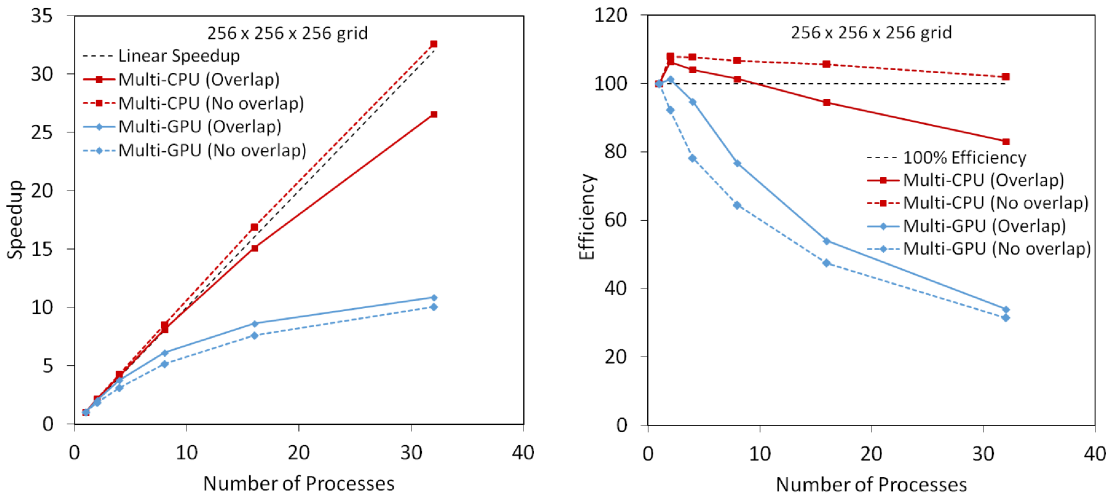
The reason that the performance when overlapping communication and computation never exceeds the baseline implementation performance is that the communication between CPUs does not produce a significant overhead cost, so there is not much overhead to mask. This is apparent from Figure 2.14, where the multi-CPU implementation already has superlinear weak scalability that is relatively unaffected by grid size.

Figure 2.15b shows that performance for the multi-GPU implementation remains unchanged or even decreases for smaller grid sizes by overlapping communication and computation. However, for the larger grid sizes the performance improves by up to 21%. When using 32 GPUs to compute the 512^3 grid solution, the performance increases from 568 GFLOP/s to 636 GFLOP/s. In comparison, this performance is over 400 times greater than the performance of a single CPU, over 17.6 times greater than the performance of a single GPU, and over 11.2 times greater than the performance of 32 CPUs bound to different sockets on HokieSpeed. The baseline multi-CPU implementation is used in this comparison to produce the best performance comparison. The decrease in performance for smaller grid sizes is attributed to a large computational overhead relative to the total computation performed

asynchronously with data communication. As discussed in Section 2.3, the computational overhead is introduced by splitting loops to separate computation that cannot overlap with communication.

When considering using overlapping communication and computation to improve performance, one should first consider the problem size of interest for the program and the weak scalability of the implementation without overlapping communication and computation. If planning to use a smaller grid size or if the weak scalability efficiency of the code is already very high, the effort required to overlap communication and computation may not be of benefit and may even reduce performance.

Now that we have demonstrated improved performance in the multi-GPU implementation for larger grid sizes, we desire to determine the effect of overlapping communication and computation on the implementation's scalability. Figure 2.16 illustrates the effect of overlapping communication with computation on strong scalability. The same grid size is used for this analysis as for the baseline implementation to allow for a direct comparison.



(a) Strong scalability speedup.

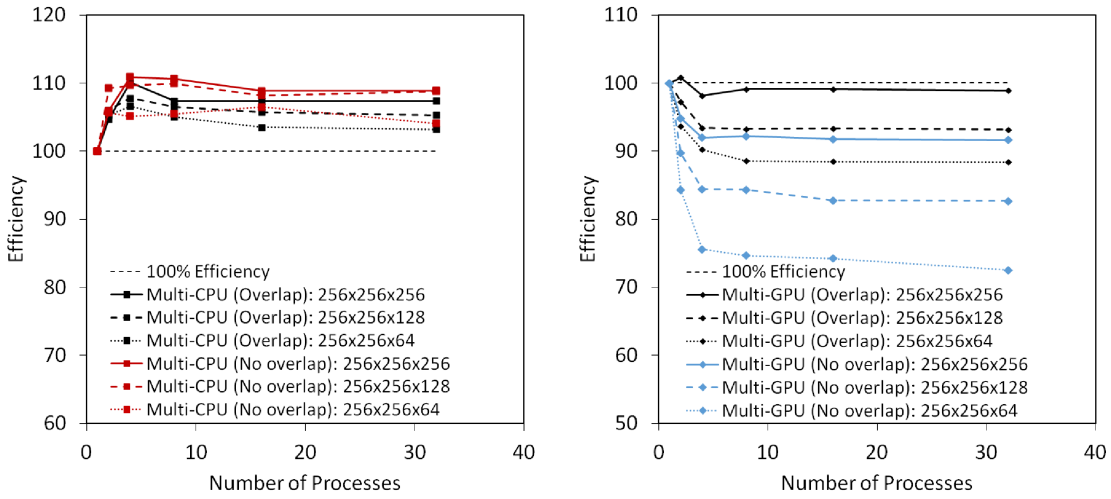
(b) Strong scalability efficiency.

Figure 2.16: Overlapping communication and computation increases strong scalability only if the communication overhead is greater than the computational overhead introduced by overlapping communication and computation.

The multi-CPU implementation consistently demonstrates a decrease in efficiency when overlapping communication with computation relative to the baseline multi-CPU implementation, dropping to 83% efficiency with 32 CPUs instead of increasing to 102% efficiency. For 2 or more CPUs, the multi-CPU implementation still maintains a linear correlation between efficiency and process count when overlapping communication and computation. The coefficient of determination for this correlation is 0.998 and indicates a loss of 7.69% in efficiency for every increase by 10 in the number of the processes. This slope is 3.9 times greater than the slope of this correlation for the baseline multi-CPU implementation, indicating that overlapping communication and computation significantly decreases the scalability of the multi-CPU implementation. The explanation for this decrease in scalability is the same as the explanation for the decrease in performance: the multi-CPU implementation does not have a high communication overhead. Therefore, overlapping communication and computation does not gain performance by masking the data communication cost, leaving the multi-CPU implementation to only suffer from the computational overhead of the code structure for overlapping communication and computation.

Conversely, the multi-GPU implementation consistently demonstrates an improvement in efficiency when overlapping communication with computation, with a difference as much as 16.5% using 4 GPUs. This benefit diminishes as the number of processes increases. With 32 GPUs, the multi-GPU implementation with overlapping communication and computation achieves an efficiency of 34%, compared to 31.4% for the baseline multi-GPU implementation. The decrease in efficiency improvement with increase in the number of processes is attributed the decreasing grid size local to each GPU that becomes too small outweigh the computational overhead of overlapping communication with computation.

Finally, the effect of overlapping communication and computation on weak scalability is analyzed for the multi-CPU and multi-GPU implementations. The results of this analysis are shown in Figure 2.17. To further emphasize the effect of overlapping communication and computation on the weak scalability the plots are zoomed in, not including zero efficiency on the y-axis. However, the same scaling is applied to both plots to accurately compare the



(a) Multi-CPU weak scalability efficiency. (b) Multi-GPU weak scalability efficiency.

Figure 2.17: Overlapping communication and computation masks the overhead of inter-GPU data communication for the multi-GPU implementation, increasing the asymptotic efficiency for weak scalability.

changes in efficiency between the multi-CPU and multi-GPU implementations. Furthermore, the same grid sizes are used to analyze weak scalability as for the baseline implementation to allow for a direct comparison.

The weak scalability of the multi-CPU implementation is not significantly affected by overlapping communication and computation, as shown in Figure 2.17a. Although the efficiency for all grid sizes does decrease, the largest decrease is 3.54% efficiency and all grid sizes maintain superlinear weak scalability. Similar to the baseline implementation, the efficiency varies by less than 3.4% for greater than two processes, indicating a scalable implementation. The computational overhead of overlapping communication and computation does not appear to affect the weak scalability as much as it affected the strong scalability, likely because the amount of computation performed on each process remains constant and large enough to mask the computational overhead of overlapping communication and computation.

In contrast, Figure 2.17b illustrates that overlapping communication and computation significantly improves the weak scalability efficiency for the multi-GPU implementation. The

asymptotic efficiency shifts by 14% efficiency for the smallest grid size, from 74% to 88% efficiency, by 10% for the medium grid size, from 83% to 93%, and by 7% for the largest grid size, from 92% to 99% efficiency. With close to linear efficiency now for the largest grid size, these results demonstrate the effectiveness of overlapping communication and computation in masking the inter-GPU data transfer latencies.

2.5 Conclusions

In this paper, we investigated the use of OpenACC to transform a serial CPU code into a single-GPU implementation. This transformation was shown to require minimal modification to the code base with the use of directive statements. In addition, these directive statements provide an abstraction from low-level implementation details that makes the code highly modular in contrast with other GPU-acceleration models such as CUDA and OpenCL. We analyzed the potential bottlenecks of this implementation that prevent efficient performance of the GPU-accelerated code. It was determined that a 24% increase in speedup is attainable by addressing these implementation bottlenecks. The optimized OpenACC implementation produces a factor of 8 speedup over the serial CPU execution using an NVIDIA Tesla C2075 GPU and comparable performance to an MPI multi-CPU implementation using 8 cores of a dual-socket Xeon CPU workstation.

Scalability and performance analyses of a hybrid MPI+OpenACC multi-GPU implementation of the 3D BDC solver demonstrates excellent weak scalability and performance on the HokieSpeed supercomputer with up to 92% efficiency using 32 GPUs. Overlapping communication and computation for the multi-GPU implementation is found to improve performance by up to 21% for large grid sizes and improve weak scalability to up to 99% effficiency with 32 GPUs. The performance decreases for smaller grid sizes since too little data computation is performed to outweigh the overhead of overlapping communication with computation.

Conversely, overlapping communication and computation always decreases the performance

and scalability of the baseline multi-CPU implementation. For large grid sizes, the performance with overlapping communication and computation attains the performance of the baseline implementation, yet never exceeds the baseline performance. This lack of performance improvement is due to the already small communication overhead of the multi-CPU implementation. Therefore, masking the communication overhead does not significantly improve performance, whereas the computational overhead of overlapping communication and computation reduces performance.

The most efficient version of the multi-GPU implementation overlaps communication and computation; however, the most efficient version of the multi-CPU implementation does not because of the performance and scalability degradation that overlapping communication with computation produces. It is important to note that overlapping communication with computation for this solver reduces portability, as the multi-CPU and multi-GPU implementations no longer use the same code base. For similar situations, the reader must use discretion to weigh the importance of portability against performance and scalability for the available computational platforms.

Using the HokieSpeed supercomputer, the performance of the final multi-GPU implementation for the 512^3-node case with 32 GPUs is over 400 times faster than a single CPU, 17.6 times faster performance than a single GPU, and 11.2 times faster than 32 CPUs bound to different sockets using the final multi-CPU implementation. These results demonstrate the capability of OpenACC to develop programmable, portable, and efficient codes from a base serial or multi-CPU implementation.

2.6 Future Work

Further analysis is required to alleviate the remaining performance degradation for the 3D LDC and BDC codes due to register spilling. Analysis in development of a 3D decomposition that minimizes the overhead incurred by non-contiguous data transfer from the GPU

to the CPU would allow for an implementation that performs well with larger numbers of GPUs and modest computational grid sizes. Moreover, a performance and scalability analysis with GPUDirect 2.0 communication is recommended to quantify the overhead of GPU-CPU communication and improve performance of the multi-GPU BDC solver. The current configuration of HokieSpeed only supports GPUDirect 1.0 communication and is not configured properly to use GPUDirect with Open MPI.

Acknowledgments

The authors acknowledge the support the Air Force Office of Scientific Research (AFOSR) Basic Research Initiative program provided for this work with Drs. Fariba Fahroo and Jean-Luc Cambier as the program managers. Support for this work was provided in part by the VT Synergistic Environments for Experimental Computing (SEEC) Center. Finally, this work was supported in part by NSF grant CNS-0960081 and the HokieSpeed supercomputer at Virginia Tech.

References

[1] T. Dong et al. "A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU." In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. IEEE, 2014.*

[2] NVIDIA. *CUDA*. http://www.nvidia.com/object/cuda_home_new.html.

[3] The Khronos Group. *The open standard for parallel programming of heterogeneous systems*. https://www.khronos.org/opencl/.

[4] D. Jacobsen and I. Senocak. "Scalability of Incompressible Flow Computations on Multi-GPU Clusters Using Dual-Level and Tri-Level Parallelism". In: *49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition* (Jan. 2011). DOI: 10.2514/6.2011-947. URL: http://dx.doi.org/10.2514/6.2011-947.

[5] D. Mu, P. Chen, and L. Wang. "Accelerating the discontinuous Galerkin method for seismic wave propagation simulations using multiple GPUs with CUDA and MPI". In: *Earthq Sci* 26.6 (Dec. 2013), pp. 377–393. ISSN: 1867-8777. DOI: 10.1007/s11589-013-0047-7. URL: http://dx.doi.org/10.1007/s11589-013-0047-7.

[6] C. Xu et al. "Collaborating CPU and GPU for large-scale high-order CFD simulations with complex grids on the TianHe-1A supercomputer". In: *Journal of Computational Physics* 278 (Dec. 2014), pp. 275–297. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2014.08.024. URL: http://dx.doi.org/10.1016/j.jcp.2014.08.024.

[7] T. Brandvik and G. Pullan. "Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware". In: *46th AIAA Aerospace Sciences Meeting and Exhibit* (Jan. 2008). DOI: 10.2514/6.2008-607. URL: http://dx.doi.org/10.2514/6.2008-607.

[8] J. Cohen and M. J. Molemaker. "A fast double precision CFD code using CUDA". In: *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions* (2009), pp. 414–429.

[9] H.-Y. Schive, Y.-C. Tsai, and T. Chiueh. "GAMER: A GRAPHIC PROCESSING UNIT ACCELERATED ADAPTIVE-MESH-REFINEMENT CODE FOR ASTROPHYSICS". In: *The Astrophysical Journal Supplement Series* 186.2 (Feb. 2010), pp. 457–484. ISSN: 1538-4365. DOI: 10.1088/0067-0049/186/2/457. URL: http://dx.doi.org/10.1088/0067-0049/186/2/457.

[10] J. Thibault and I. Senocak. "CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows". In: *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition* (Jan. 2009). DOI: 10.2514/6.2009-758. URL: http://dx.doi.org/10.2514/6.2009-758.

[11] *The OpenMP API specification for parallel programming*. http://openmp.org/wp/.

[12] *Message Passing Interface Forum*. http://www.mpi-forum.org/.

[13] OpenACC-standard. *OpenACC Home*. http://www.openacc.org.

[14] CAPS. *HMPP Directives Reference Manual*. URL: https://www.olcf.ornl.gov/wp-content/uploads/2012/02/HMPPWorkbench-3.0_HMPP_Directives_ReferenceManual.pdf.

[15] Y. Komura. "OpenACC programs of the Swendsen–Wang multi-cluster spin flip algorithm". In: *Computer Physics Communications* (Aug. 2015). ISSN: 0010-4655. DOI: 10.1016/j.cpc.2015.08.022. URL: http://dx.doi.org/10.1016/j.cpc.2015.08.022.

[16] M. Misic, D. Dasic, and M. Tomasevic. "An analysis of OpenACC programming model: Image processing algorithms as a case study". In: *Telfor Journal* 6.1 (2014), pp. 53–58. ISSN: 1821-3251. DOI: 10.5937/telfor1401053m. URL: http://dx.doi.org/10.5937/telfor1401053M.

[17] M. Otten et al. "An MPI/OpenACC Implementation of a High Order Electromagnetics Solver with GPUDirect Communication". In: *The International Journal of High Performance Computing Applications* 30.3 (2016), pp. 320–334. DOI: 10.1177/1094342015626584.

[18] S. Markidis et al. "OpenACC acceleration of the Nek5000 spectral element code". In: *International Journal of High Performance Computing Applications* 29.3 (Mar. 2015), pp. 311–319. ISSN: 1741-2846. DOI: 10.1177/1094342015576846. URL: http://dx.doi.org/10.1177/1094342015576846.

[19] R. Xu et al. "Multi-GPU Support on Single Node Using Directive-Based Programming Model". In: *Scientific Programming* 2015 (2015), pp. 1–15. ISSN: 1875-919X. DOI: 10.1155/2015/621730. URL: http://dx.doi.org/10.1155/2015/621730.

[20] M. Chrust, E. Laurendeau, and L. Ostiguy. "Accelerating low-fidelity aerodynamic codes on multi- and many-core architectures". In: *The Journal of Supercomputing* 71.9 (May 2015), pp. 3456–3481. ISSN: 1573-0484. DOI: 10.1007/s11227-015-1444-6. URL: http://dx.doi.org/10.1007/s11227-015-1444-6.

[21] C. Couder-Castaneda et al. "Performance of a Code Migration for the Simulation of Supersonic Ejector Flow to SMP, MIC, and GPU Using OpenMP, OpenMP+LEO, and OpenACC Directives". In: *Scientific Programming* 2015 (2015), pp. 1–20. ISSN: 1875-919X. DOI: 10.1155/2015/739107. URL: http://dx.doi.org/10.1155/2015/739107.

[22] T. Komoda et al. "Integrating Multi-GPU Execution in an OpenACC Compiler". In: *Parallel Processing (ICPP), 2013 42nd International Conference on*. 2013, pp. 260–269. DOI: 10.1109/ICPP.2013.35. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6687359.

[23] B. P. Pickering et al. "Directive-based GPU programming for computational fluid dynamics". In: *Computers & Fluids* 114 (July 2015), pp. 242–253. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2015.03.008. URL: http://dx.doi.org/10.1016/j.compfluid.2015.03.008.

[24] Y. Xia et al. "OpenACC acceleration of an unstructured CFD solver based on a reconstructed discontinuous Galerkin method for compressible flows". In: *Int. J. Numer. Meth. Fluids* 78.3 (Feb. 2015), pp. 123–139. ISSN: 0271-2091. DOI: 10.1002/fld.4009. URL: http://dx.doi.org/10.1002/fld.4009.

[25] M. Norman et al. "A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel". In: *Journal of Computational Science* 9 (July 2015), pp. 1–6. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2015.04.022. URL: http://dx.doi.org/10.1016/j.jocs.2015.04.022.

[26] H. Matsufuru et al. "OpenCL vs OpenACC: Lessons from Development of Lattice QCD Simulation Code". In: *Procedia Computer Science* 51 (2015), pp. 1313–1322. ISSN: 1877-0509. DOI: 10.1016/j.procs.2015.05.316. URL: http://dx.doi.org/10.1016/j.procs.2015.05.316.

[27] A. Hart, R. Ansaloni, and A. Gray. "Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers". In: *Eur. Phys. J. Spec. Top.* 210.1 (Aug. 2012), pp. 5–16. ISSN: 1951-6401. DOI: 10.1140/epjst/e2012-01634-y. URL: http://dx.doi.org/10.1140/epjst/e2012-01634-y.

[28] *The OpenACC API: Version 2.0*. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf.

[29] *The OpenACC API: Version 1.0*. http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf.

[30] A. J. Chorin. "A numerical method for solving incompressible viscous flow problems". In: *Journal of Computational Physics* 2.1 (Aug. 1967), pp. 12–26. ISSN: 0021-9991. DOI: 10.1016/0021-9991(67)90037-x. URL: http://dx.doi.org/10.1016/0021-9991(67)90037-X.

[31] W. Kays, M. Crawford, and B. Weigand. *Convective Heat and Mass Transfer*. 4th. McGraw-Hill, 2004.

[32] N. Corporation. *NVIDIA Visual Profiler*. https://developer.nvidia.com/nvidia-visual-profiler.

[33] Advanced Research Computing at Virginia Tech. *HokieSpeed (CPU/GPU)*. https://secure.hosting.vt.edu/www.arc.vt.edu/hokiespeed-cpugpu/.

[34] Intel. *Intel Xeon Processor E5-2687W*. http://ark.intel.com/products/64582/Intel-Xeon-Processor-E5-2687W-20M-Cache-3_10-GHz-8_00-GTs-Intel-QPI.

[35] Intel. *Intel Xeon Processor E5645*. http://ark.intel.com/products/48768/Intel-Xeon-Processor-E5645-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI.

[36] Intel. *Intel Xeon Processor E5-2680*. http://ark.intel.com/products/64583/Intel-Xeon-Processor-E5-2680-20M-Cache-2_70-GHz-8_00-GTs-Intel-QPI.

[37] www.techpowerup.com. *NVIDIA Tesla C2075*. https://www.techpowerup.com/gpudb/563/tesla-c2075.

[38] www.techpowerup.com. *NVIDIA Tesla M2050*. https://www.techpowerup.com/gpudb/1534/tesla-m2050.

[39] www.techpowerup.com. *NVIDIA Tesla K80m*. https://www.techpowerup.com/gpudb/2616/tesla-k80m.

[40] Advanced Research Computing at Virginia Tech. *NewRiver*. https://secure.hosting.vt.edu/www.arc.vt.edu/computing/newriver/.

[41] Virginia Tech Department of Computer Science. *HokieSpeed*. https://www.cs.vt.edu/facilities/hokiespeed.

[42] Lawrence Latif. *Nvidia announces its Kepler-based Tesla K10 GPGPU board*. http://www.theinquirer.net/inquirer/news/2174756/nvidia-announces-kepler-tesla-m10-gpgpu-board.

[43] Advanced Research Computing at Virginia Tech. *NewRiver Supercomputer Released to VT Researchers*. https://secure.hosting.vt.edu/www.arc.vt.edu/2015/08/arc-releases-supercomputer-newriver-to-vt-researchers/.

[44] D. A. Jacobsen and I. Senocak. "Multi-level parallelism for incompressible flow computations on GPU clusters". In: *Parallel Computing* 39.1 (Jan. 2013), pp. 1–20. DOI: 10.1016/j.parco.2012.10.002. URL: http://dx.doi.org/10.1016/j.parco.2012.10.002.

Chapter 3

Multilevel Parallelism with MPI and OpenACC for Complex CFD Codes

Andrew McCall and Christopher J. Roy
Department of Aerospace and Ocean Engineering, Virginia Tech, Blacksburg, VA 24061, USA

Attribution

The first author (Andrew McCall) provided the primary contribution to the manuscript's research and content. All research was conducted by the first author and all results were collected and analyzed by the first author. The second author (Christopher J. Roy) provided the guidance and feedback necessary for research development and composition of this manuscript.

Abstract

One of the challenges in GPU computing today is the extensive work required to transform an existing program that runs on the CPU to one that runs on the GPU. This research studies the ability of OpenACC to develop a portable GPU-accelerated CFD code while maintaining the same code base as an existing code that runs on the CPU. The code used is a second-order accurate, structured, finite volume CFD solver for the Euler and Navier-Stokes equations. The challenges of using OpenACC with a complex program written in modern Fortran are detailed in this paper, as well as a number of optimization considerations when using OpenACC. MPI is used to communicate between GPUs for a multi-GPU implementation of the code base. Test cases of a NACA 0012 airfoil, lid-driven cavity, and M6 Onera wing are used to test the performance and scalability of the multi-CPU and multi-GPU implementations compiled from the same code base. The performance results demonstrate that a single NVIDIA Tesla M2050 GPU attains a 3.7 speedup for the airfoil, 4.25 speedup for the lid-driven cavity, and 3.2 speedup for the wing over a single Intel Xeon E5645 processor core. The performance of 27 GPUs is up to 100 times faster than a single core and 25 times faster than 27 cores and the multi-GPU implementation maintains an efficiency of 95% up to 27 GPUs.

3.1 Introduction

With the expansion of compute capability in graphics processing units (GPUs) to include general purpose computation, many areas of scientific computing have found significant performance improvements with the use of the GPU over the CPU. These areas include Monte Carlo simulations [1], image processing [2], electromagnetics [3], and computational fluid dynamics (CFD) [4, 5, 6, 7, 8, 9, 10]. Multiple programming models, including CUDA [11], OpenCL [12], and OpenACC[13], have been designed to allow developers to interface with the GPU. The central design of these models is to transfer data between the host CPU's memory and the target GPU's memory and offload kernel codes to run on the target GPU. The GPU subsequently operates on the transferred data using the kernel code.

One significant challenge in exploiting the performance benefits of the GPU is that many complex and lengthy codes have already been developed, and rewriting significant portions of these codes to run on the GPU presents both a time-consuming and an expensive process.

For commercial codes on the order of 1 million lines of code, many would consider the time, money, and effort required to rewrite significant portions of the code to outweigh the benefit of the potential performance gains that GPUs offer. The standard programming models for the GPU, CUDA and OpenCL, require explicit definition of low-level implementation details dependent upon the GPU's architecture, meaning that any kernel code run on the GPU must be rewritten to account for these architecture differences. Moreover, since the architecture of GPUs rapidly changes, the code would likely need to be modified again by the time of completion of the initial GPU implementation in order to work well with the latest architecture. Furthermore, the complexity of these low-level programming models adds to the intractability of their use for large codes that already exhibit significant complexity.

The release of OpenACC in 2011[13] provided a solution to this issue. By compromising absolute control of the low-level implementation details, the directive-based programming model, similar in design to OpenMP [14], allows the programmer to identify parallelism in the code with minimal modification to the code base. The OpenMP 4.0 standard even supports offloading computation to an accelerator; however, compiler implementations have not yet leveraged this standard for GPU acceleration. This model reduces code modification to a level that makes the porting of large codes to the GPU feasible. Furthermore, this model allows for the potential of a single code base for the CPU and the GPU, an accomplishment not possible with CUDA and OpenCL. With a single code base, the programmer no longer has to abandon the CPU version of the code or maintain two separate versions of the code, greatly improving the portability and maintainability of the code.

Although these advantages are alluring, the reader must be mindful of the cost of using the OpenACC programming model. The portability, maintainability, and simplicity of the OpenACC programming model comes at the price of performance degradation. As demonstrated by Luo[15] and Hoshino et al.[16], an OpenACC implementation of a given algorithm can exhibit a factor of two to four in performance reduction relative to CUDA. Therefore, some programs may find little to no performance increase with the use of OpenACC due to a numerical algorithm that already limits the GPU performance. In these cases, the

meager performance gains using OpenACC negate the benefit of effort spent to utilize GPU acceleration.

For CFD applications, acceleration of these codes using OpenACC has been shown to be beneficial to performance in some cases. Research in OpenACC acceleration for CFD includes the work of Markidis et al. [4] in spectral methods for the simulation of 3D incompressible flows and the work of Xia et al. [10] in higher order discontinuous Galerkin methods for 3D compressible flows on hybrid unstructured grids. The use of OpenACC for CFD codes even extends to the GPU acceleration of vortex-lattice methods [6]. The work of Luo et al. [17] demonstrated a speedup of 3.5 over the CPU version using OpenACC for a multi-block incompressible Navier-Stokes solver. Even in the case of large, complex, commercial CFD codes, OpenACC has proven itself effective in accelerating performance [16]. An alternative solution to the GPU parallelization of complex CFD codes that has found popularity is the development and use of tools that semi-automatically port the code to the GPU, as researched by Corrigan and co-workers[18, 19] and van Werkhoven and Hijma[20].

This work intends to demonstrate the capability of OpenACC to simplify the process of porting a large and complex CFD code to the GPU using a research CFD code named SENSEI[21]. SENSEI is implemented with modern Fortran and as such requires special attention be given to the interaction between OpenACC and Fortran to design a program that works on the CPU and easily ports to the GPU using OpenACC. Furthermore, this work intends to show that the ported code achieves scalable peformance that is notably faster than the serial version.

The remainder of this paper is outlined as follows. Section 3.2 covers the background information and the theory behind the CFD solver used for this analysis. Section 3.3 discusses the process of porting SENSEI to the GPU while maintaining the same code base. This discussion includes the compatibility issues found between OpenACC and Fortran with resolutions to these issues, as well as discussion of the use of MPI for the multi-GPU implementation, and the overall implementation structure. The performance and scalability results using

three test geometries are presented and discussed in Section 3.4 with conclusions stated in Section 3.5 and future work discussed in Section 3.6.

3.2 Background and Theory

As already mentioned, the research code SENSEI[21] is used to test the capability of OpenACC to port a complex CFD code to the GPU. SENSEI is a second order, structured grid, finite volume solver for both the Euler and Navier-Stokes equations, developed to solve compressible flow problems. SENSEI can use both explicit and implicit time integration; however, for this study only the explicit solver, a Runge-Kutta method, will be ported to the GPU. Additional features of SENSEI include capability for the method of manufactured solutions (MMS) for code verification, axisymmetric problems, truncation error estimation, RANS turbulence models, and grid adaptation. In total, SENSEI contains over 123,000 lines of code. Although many commercial codes have significantly more lines of code, SENSEI nevertheless provides a good test case for porting large and complex codes to the GPU.

The three-dimensional Navier-Stokes equations may be written in weak, conservation form for a fixed, infinitesimal control volume as follows[22]:

$$\frac{\partial}{\partial t}\int_{\Omega}\vec{U}d\Omega + \oint_{S}\vec{F}d\vec{S} - \oint_{S}\vec{F}_{v}d\vec{S} = \int_{\Omega}\vec{Q}d\Omega \tag{3.1}$$

where the conserved variables $\vec{U}$ and the inviscid and viscous flux vector tensors $\vec{F}$ and $\vec{F}_v$ are functions of the primitive variables density ρ, the velocity components v_i, and pressure p:

$$\vec{U} = \begin{bmatrix} \rho \\ \rho\vec{v} \\ \rho e_t \end{bmatrix}, \vec{F} = \begin{bmatrix} \rho\vec{v} \\ \rho\vec{v}\otimes\vec{v} + p\bar{\bar{I}} \\ \rho h_t\vec{v} \end{bmatrix}, \vec{F}_v = \begin{bmatrix} 0 \\ \bar{\bar{\tau}} \\ \bar{\bar{\tau}}\cdot\vec{v} + k\vec{\nabla}T \end{bmatrix}$$

The variables k and T are the coefficient of thermal conductivity and temperature, respectively. The total energy e_t and total enthalpy h_t are computed using the assumption of a perfect gas. Under the assumption of a Newtonian fluid that follows the Stokes relation, the shear stresses are computed as

$$\tau_{ij} = \mu(\delta_i v_j + \delta_j v_i) - \frac{2}{3}\mu(\vec{\nabla} \cdot \vec{v})\delta_{ij} \tag{3.2}$$

where μ is the dynamic viscosity. With no body forces and when not using the method of manufactured solutions, the source term $\vec{Q}$ is neglected. Upwind flux schemes with MUSCL extrapolation and limiters are used to compute a second order accurate solution. Local time stepping is computed based on the wave stability condition and global time stepping uses the minimum local time step. SENSEI's design is further detailed by Derlaga[21].

3.3 Porting to the GPU

3.3.1 Initial Considerations

A number of considerations must be made in the process of accelerating computation with a GPU. The first step in offloading computation to the GPU is to identify the most computationally dense portions of the serial code. Figure 3.1 is a summary of the profile results for the explicit, serial version of SENSEI. These results show that 88.9% of the code execution occurs in the residual calculation; therefore, the primary focus for acceleration of the code lies in offloading computation of the residuals with OpenACC.

Due to the high latency of communication between the GPU and CPU, transferring solution data at each time step between the GPU and CPU is not feasible while still attaining reasonable performance. Therefore, since the residual computation requires storing the full solution data on the GPU, other operations on the solution data need to be performed on the GPU. These operations including calculation of the time step, calculation of the residual

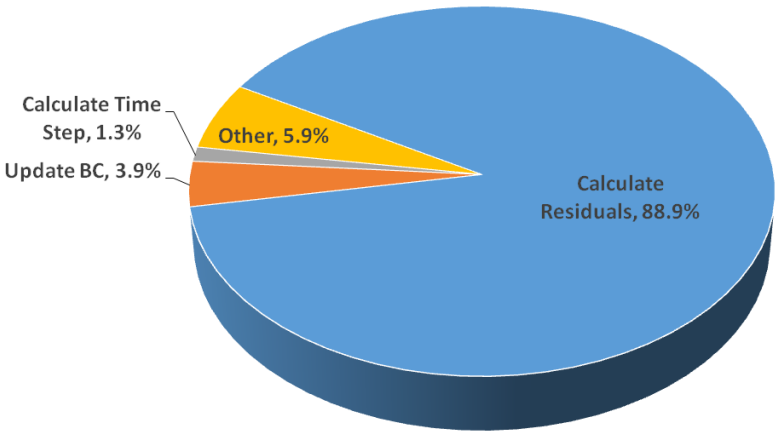


Figure 3.1: Summary profile of SENSEI's serial execution on the CPU.

norms, and updating the solution. Enforcement of the boundary conditions is also an operation on the solution data; however, since this operation only requires solution data along the boundaries, further consideration must be made before determining whether or not to offload the computations for boundary condition enforcement. As Figure 3.1 illustrates, the boundary condition enforcement and time step calculations are the most expensive computations outside of the residual calculation, consuming 3.9% and 1.3% of the execution time, respectively.

3.3.2 Using OpenACC with Modern Fortran

Restrictions

Aside from the standard modifications necessary to port a code to the GPU using OpenACC, there exist multiple restrictions on the use of The Portland Group, Inc. (PGI) OpenACC implementation[23] with modern Fortran that may require additional modification of the original code base. Some of these restrictions are not as clearly defined in the OpenACC standard or in the PGI OpenACC implementation resources as others; thus, we outline a

number of important restrictions in using PGI OpenACC with Fortran. These restrictions relate to the following functionality.

1. Procedure optional arguments.
2. Array-valued functions.
3. Multi-dimensional array assignments.
4. Temporary arrays.
5. Reductions with derived type members.
6. Procedure pointers.
7. Derived types with allocatable members.

Although optional arguments for procedure calls are allowed, the Fortran `present` function will return `.true.` for all optional arguments because OpenACC implicitly passes a dummy argument value. This issue relates not only to PGI OpenACC, but to the OpenACC standard, and has been proposed to be resolved in the OpenACC 2.6 standard[24].

Similarly, calls to array-valued functions (functions with an array return type) are permissible within an OpenACC kernel, yet PGI OpenACC handles these functions in a non-intuitive manner. The result from the function call is stored in a temporary array and subsequently assigned to the array specified on the left-hand side of the statement. PGI OpenACC automatically generates this temporary array, not as a private, local variable within the kernel loop but as data to be copied between the host and the accelerator device. This issue not only introduces a potential loop dependency but also reduces the overlap of asynchronously executing kernels as the transfer of data between the host and the device delays kernel execution. Furthermore, Fortran intrinsic functions that have array-valued results are not supported for use within OpenACC kernels. This includes functions such as the intrinsic matrix multiplication function `matmul`. Therefore, all array-valued functions used within an

OpenACC kernel should be transformed to subroutines with a single `intent(out)` parameter within the call statement. This modification avoids issues with temporary array generation and unsupported intrinsic functions.

In Fortran, array operations may be stated in vector form, preventing the need for an explicit loop to assign values to an array. In the case of multi-dimensional array assignment operations, the developer is not required to specify the indices or dimensions of the array if all indices in the array are being set to the same value. When using PGI OpenACC, the developer must be careful to specify the number of dimensions of the array in these situations. Otherwise, a PGI function `pgf90_mzero8` is called to perform the assignment, which is not supported in PGI OpenACC. For example, consider the initialization of a three-dimensional array, `A`, to zero. Rather than writing this statement as follows,

```
A = 0
```

one must write the initialization in the following manner:

```
A(:,:,:) = 0
```

Another feature of Fortran that is not supported with PGI OpenACC is the generation of temporary arrays as parameters to a procedure call. These temporary arrays are generated when passing a non-contiguous sub-array as a parameter or when performing some operation upon the array within the procedure call. For example, since Fortran uses column-major ordering of arrays, the following statement requires generation of a temporary array due to non-contiguous memory access:

```
call foo(...,rho(i_low:i_high,:,:),...)
```

Furthermore, the following statement requires a temporary array due to operations performed upon the array within the procedure call:

```
call foo(...,-eta_normal(:),...)
```

To resolve this issue, the developer must manually create and pass these temporary arrays as parameters to the procedure call.

The following restrictions apply to modern Fortran and the use of derived types, available since the Fortran 90 standard. First, PGI OpenACC does not support reduction operations on derived type members. This issue is resolved by declaring an additional variable within the current scope of program execution for the reduction operation and subsequently assigning the reduced value to the derived type member.

Second, PGI OpenACC does not support the use of procedure pointers within OpenACC kernels. This issue is resolved by calling a procedure that uses a `select case` structure to determine which procedure to call. Although tests show this implementation method impacts the overall performance by less than 1%, even for frequently called procedures, the developer may desire to preserve procedure pointers when compiling the code base for the CPU. To maintain a single code base, preprocessing directives allow the programmer to use the same name for the OpenACC procedure with `select case` statements as the name of the procedure pointer used for the CPU version.

Finally, although the latest versions of PGI OpenACC support arrays of derived types and derived types with allocatable members, PGI OpenACC does not yet support arrays of derived types with allocatable members as of PGI version 16.9. To make use of derived types with allocatable members, the developer must explicitly reference each allocatable member that will be used on the accelerator within a data directive[25].

Recommendations

A number of issues did not consistently present themselves in relation to a specific Fortran feature; however, the authors present some recommendations to either avoid these issues or to provide insight toward finding a resolution for these issues.

Although the OpenACC 2.0 standard allows for procedure calls within kernels, this is still not always the best approach to port an existing code to the GPU. In complex CFD codes, these procedure calls will sometimes cause subtle errors in the program execution buried within the underlying OpenACC kernel implementation. In these cases, the most straightforward correction of the error may be to use the PGI compiler option `-Minline` to force the compiler to inline specific procedures. Less than 1% performance differences were noted when inlining these procedures in SENSEI. As an example, the limiter calculation procedure had to be inlined to prevent errors in the residual calculations.

Another issue that may appear in more complex codes is the overwriting of data on the device that is not supposed to be modified. In SENSEI, an allocatable array within a derived type contains the normal vectors for all ξ (corresponding to the i index) cell faces. Since the grid geometry is fixed, these values should remain unmodified throughout program execution; however, during execution of the GPU-accelerated version of the code, some values in this array were modified. We determined that this issue was caused by performing array operations on private arrays within a parallel loop kernel, which would overwrite the data within the normal vector array. Since these array operations were in vector form, a directive could not be used to specify the level of parallelism for these inherent loop structures. By replacing the array operations with an explicit loop and applying a loop directive to express the level of parallelism for the loop, these issues were circumvented. This resolution made less than 1% of an impact on performance.

These, and other complex errors make the development and debugging of complex OpenACC codes considerably more challenging than for more basic codes. The authors did not experience these issues when using OpenACC for a more basic research code[26] that did not make use of derived types or much of the other functionality that modern Fortran provides. However, the authors did manage to find resolutions for all of these issues without making significant sacrifices to performance. Furthermore, as the PGI OpenACC implementation becomes more mature, the authors anticipate these issues to be resolved. Support for most modern Fortran functionality has only begun since the release of the OpenACC 2.0 standard

in 2014[13].

3.3.3 MPI

As will be discussed in Section 3.3.5, MPI is used to develop the multi-GPU solver. Although the use of MPI to communicate data between blocks is largely transparent to the OpenACC implementation, as it occurs on the CPU, OpenACC does still have to transfer boundary data to the CPU in order to exchange data between GPUs. As such, the decomposition method utilized by MPI is important to take into account when designing the GPU implementation.

Multiple considerations must be made in the decomposition of the domain. The domain could be sub-divided along a single dimension or along multiple dimensions. The desirable characteristics that serve as metrics to influence the choice of decomposition type are as follows.

1. Maximize the utility of the decomposition method for any general case and any number of available processes.
2. Minimize the total amount of data transferred between sub-domains.
3. Minimize the difference in decomposed domain sizes to balance the computational load.
4. Minimize the number of communication calls made between sub-domains.
5. Maximize the amount of contiguous memory transferred between sub-domains.

SENSEI automatically decomposes the domain by focusing on satisfying the first three metrics. This solution is determined by seeking a decomposition that minimizes the surface area to volume ratio and minimizes the load imbalance between processes. However, when using the GPU, which suffers from latency-bound communication with the CPU, the final metric for contiguous memory transfers becomes much more important. Therefore, since the decomposition method allows for inter-block i_{min} and i_{max} faces, which contain completely

non-contiguous data, a mitigation for the performance degradation caused by non-contiguous data transfer is necessary. Section 3.4.2 addresses this issue among other optimization considerations.

3.3.4 Implementation Structure

Due to the latency-bound communication between the CPU and the GPU, data transfer between the host and device should be minimized. This requires that the solution data remain on the GPU across iterations and the majority of the loop iteration must be executed on the GPU. Therefore, kernels are generated to compute the local time step for each cell and to perform a reduction operation to determine the global time step. As discussed in Section 3.4.2, the boundary conditions are updated on the CPU for performance reasons. Therefore, solution data must be transferred across inter-block boundaries and periodic boundaries. Without using GPUDirect, this requires that the solution data on all boundaries be passed back to the CPU. Finally, the residual calculation and reductions to compute the residual L2 norms are computed on the GPU. The solution is updated on the GPU using the residual calculations and checked for convergence by passing the computed L2 norms to the host CPU.

A number of modifications to SENSEI's residual calculation structure were necessary to remove loop dependencies. Originally, when computing the residual of the solution, the ξ-, η-, and ζ-fluxes (corresponding to the i, j, and k indices of the array) were computed in separate loops and directly added to the residual array. Each loop iteration computed the flux at a given face, requiring assignments to two different indices of the residual array within each loop iteration. This structure generated loop dependencies that prevent parallelization. Furthermore, since the ξ-, η-, and ζ-fluxes all operated directly on the residual array, dependencies existed between the different loop structures, preventing asynchronous execution of the kernels. To remove these dependencies, arrays were allocated for the ξ-, η-, and ζ-fluxes to directly store them, with a final loop added to compute the complete

residuals using the computed fluxes. Although adding these three arrays increases the memory usage on the GPU by 32.2% when using the Euler equations and by 14.6% when using the Navier-Stokes equations, this method avoids atomic operations within the kernels, which would cause significant performance degradation. Maintaining this structure for running on the CPU marginally increases the memory usage by less than 10% and causes less than a 3% difference in performance; therefore, the code base was modified uniformly for both execution on the CPU and the GPU to maintain the same code base.

Furthermore, the serial version of the code reuses face limiter calculations between iterations to reduce the total computation. Since this structure improves performance for the CPU version, preprocessing directives are used to keep this algorithm structure for the CPU, yet avoid these loop dependencies on the GPU.

3.3.5 Computing Resources

Table 3.1 summarizes the machines used for this study and their specifications. The single-GPU performance optimization results for SENSEI are obtained using a Dell T7600 Precision workstation with dual-socket 8-core Intel Xeon E5-2687W processors, for a total of 16 cores. The workstation also contains two NVIDIA Tesla C2075 GPUs and 67 GB memory. Scalability and performance analyses of the multi-GPU implementation are conducted using the HokieSpeed [27] CPU-GPU computer cluster at Virginia Tech. HokieSpeed has 204 compute nodes connected by a QDR Infiniband interconnect, each containing dual-socket 6-core Intel Xeon E5645 processors, for a total of 12 cores within each node. Every node also contains two NVIDIA Tesla M2050 GPUs and 24 GB of memory.

Table 3.1: Specifications for the hardware in the different machines used in this study. Note that two processors and two GPUs exist on each node for all machines.

Machine Name	**Workstation**	**HokieSpeed**	**NewRiver**
Memory Specifications			
Size (GB)	67	24	256
Bandwidth (GB/s)	51.2	32.0	51.2
Processor Specifications			
Model	Intel Xeon E5-2687W[28]	Intel Xeon E5645[29]	Intel Xeon E5-2680[30]
Physical Cores	8	6	12
Base Clock Rate (GHz)	3.10	2.40	2.70
Shared L3 Cache (MB)	20	12	20
GPU Specifications			
Model	NVIDIA Tesla C2075[31]	NVIDIA Tesla M2050[32]	NVIDIA Tesla K80[33]
Architecture	Fermi	Fermi	Kepler
Chip	GF110	GF100	$2 \times$ GK210
Memory Size (MB)	6144	3072	2×12288
Bandwidth (GB/s)	144.0	148.4	2×240.6
Cores (Total)	448	448	2×2496
Core Clock Rate (GHz)	1.150	1.150	0.560 – 0.875

The comparison of performance between the Fermi and Kepler architectures for NVIDIA Tesla GPUs makes use of the Virginia Tech NewRiver [34] computer cluster. NewRiver has 134 nodes, with 8 nodes dedicated to GPU computing. These nodes use dual-socket 12-core Intel Xeon E5-2680 processors, for a total of 24 cores within each node. Furthermore, each node contains two NVIDIA Tesla K80 GPUs and 256 GB of memory. Due to the limited availability of nodes, only the performance of a single GPU is analyzed on NewRiver to compare with the single GPU performance on the workstation and on HokieSpeed. All analyses use the PGI Fortran 15.7 compiler and the multi-GPU analysis on HokieSpeed makes use of the Open MPI 1.8.5 library.

3.4 Results and Discussion

This section presents the computational performance achieved by importing SENSEI's base code into the GPU with OpenACC. A description of the test cases used to analyze performance and scalability as well as a discussion of the significant optimizations discovered in the development of the OpenACC version of SENSEI preface the presentation and discussion of the performance results.

3.4.1 Test Case Descriptions

NACA 0012 Airfoil

Three test cases are used to test the performance of SENSEI when accelerated using OpenACC. The first test case is a two-dimensional NACA 0012 airfoil geometry with the boundaries 500 chord lengths away from the airfoil. The grids used for this study are listed in Table 3.2. This test case is used for performance optimization and to demonstrate the performance and scalability of the GPU-accelerated version of SENSEI for two-dimensional problems.

Table 3.2: Grid sizes used for the NACA 0012 airfoil test case.

h_1	1792×512
h_2	896×256
h_4	448×128

This two-dimensional geometry is solved using the Euler equations for the steady-state, inviscid flow solution. Global time-stepping is used with the 4-step Runge-Kutta time integration scheme and the second-order residuals are computed using the Van Leer[35] upwind flux scheme with MUSCL extrapolation[35] and a limiter function developed by Michalak and Ollivier-Gooch[36]. The farfield conditions of this test case are listed in Table 3.3 and the steady-state flow solution is illustrated in Figure 3.2.

Table 3.3: Farfield conditions for the NACA 0012 airfoil test case.

$\alpha\ (deg)$	5
$Mach$	0.25
$p_\infty\ (Pa)$	84,307
$T_\infty\ (K)$	278

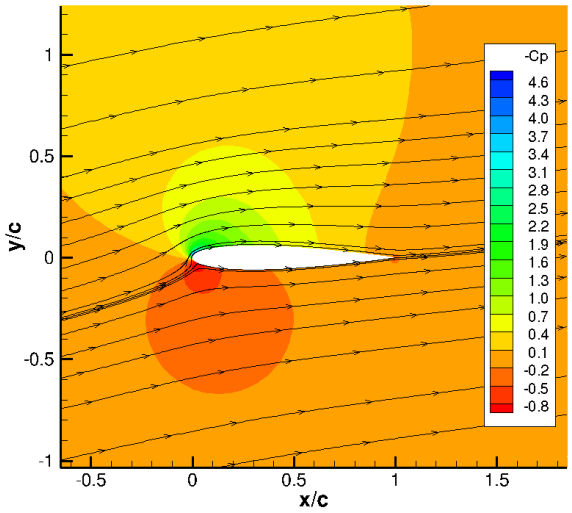


Figure 3.2: NACA 0012 airfoil steady-state solution.

Lid-driven Cavity (LDC)

The second test case is a 0.001 m cubic, lid-driven cavity (LDC) at a Reynolds number of 4,660. The grids used are systematically refined from a $32 \times 32 \times 32$-cell uniform grid. Performance optimizations are conducted using this test case as well as a demonstration of the performance and scalability of the GPU-accelerated version of SENSEI for three-dimensional problems. This test case solves the laminar Navier-Stokes equations for a steady-state solution of the compressible air flow within the cavity. As with the previous test case, global time-stepping is used with the 4-step Runge-Kutta time integration scheme and the fluxes are computed using the Van Leer flux scheme with MUSCL extrapolation. For this test case, the van Albada[37] limiter function is used. The lid flow conditions are fixed at

the values listed in Table 3.4 and the steady-state flow solution is illustrated in Figure 3.3.

Table 3.4: Lid conditions for the LDC test case.

Re	4,660
$u\ (m/s)$	68.0
$v, w\ (m/s)$	0.0
$Mach$	0.2
$p_\infty\ (kPa)$	101.325
$T_\infty\ (K)$	288.15

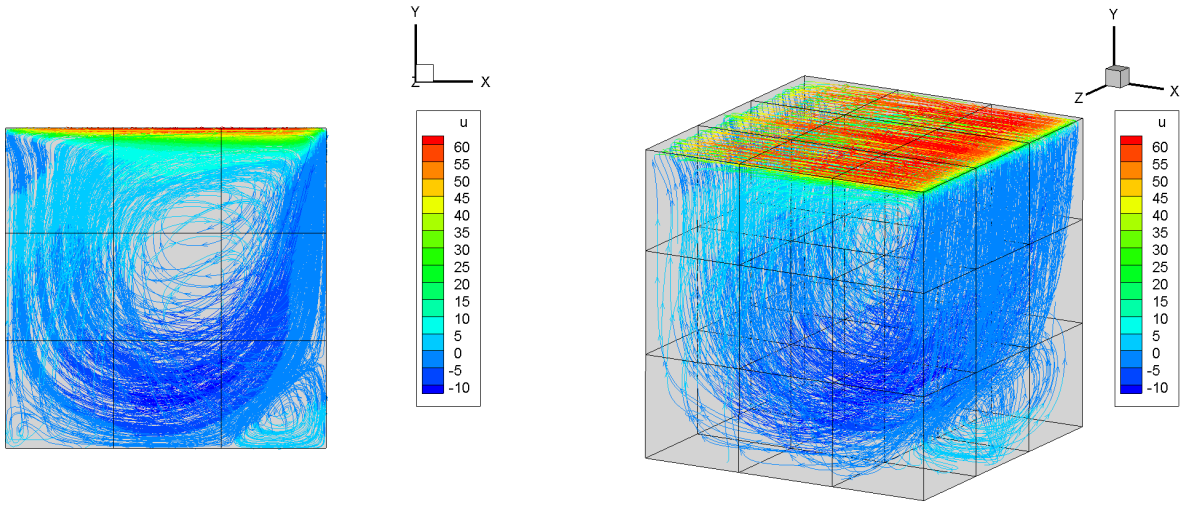


Figure 3.3: Streamlines for the solution of compressible airflow in a three-dimensional lid-driven cavity at a Reynolds number of 4,660.

M6 Onera Wing

The final test case is for the classic M6 Onera wing geometry, tested with viscous flow at transonic speeds. The laminar Navier-Stokes equations are used to determine a steady-state solution. The multi-block, structured grid that is used comes from NASA[38] and has four blocks: two $24 \times 48 \times 32$-cell blocks in the wake aft of the wing, and two $72 \times 48 \times 32$-cell blocks to cover the wing surface and upstream domain. The total cell count is approximately 295,000 cells. This test case demonstrates the capability of SENSEI, with GPU acceleration,

to solve a multi-block, three-dimensional geometry with a complex, transonic flow. With a multi-block grid, load-balancing becomes an issue as the provided processes must be divided among the four blocks in a way that minimizes imbalance, while attempting to satisfy the other metrics of a satisfactory decomposition, as described in Section 3.3. The Reynolds number of the flow for this test case is 11.72 million, using the mean aerodynamic chord of 0.64607 m as the reference length. The farfield conditions are listed in Table 3.5 and the steady-state solution is illustrated in Figure 3.4.

Table 3.5: Farfield conditions for the M6 Onera wing test case.

Re	11.72 million
$Mach$	0.8395
α (deg)	3.06
β (deg)	0.0
p_∞ (kPa)	315.980
T_∞ (K)	255.56

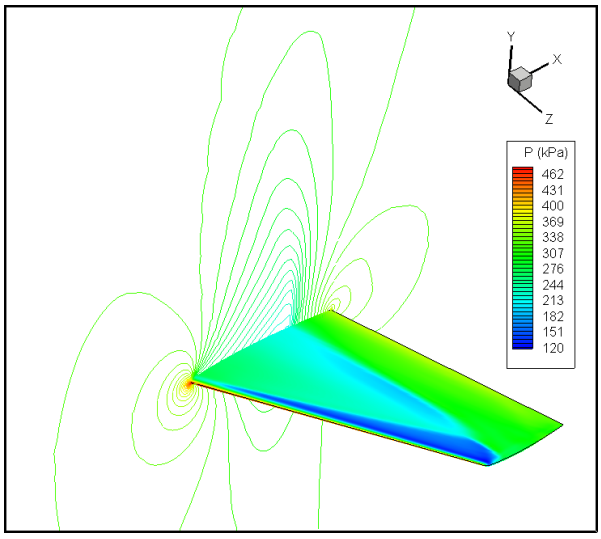


Figure 3.4: M6 Onera wing steady-state solution.

3.4.2 OpenACC Optimization

The main focus of this study is to minimize the changes made to the code base to demonstrate OpenACC's capability to feasibly port large and complex codes to the GPU; however, some optimization is necessary to obtain a reasonable speedup over the CPU's performance. Therefore, a few optimization considerations that require minimal alteration of the code base are evaluated in this study. These optimizations are based on the findings from optimizing SENSEI's OpenACC implementation and later were individually tested to verify their impact on the performance of the optimized version of the code. The results presented in the initial discussion are for the NACA 0012 airfoil test case. In the following discussion, these results are compared with the LDC test case results. The NVIDIA profiling tool[39] is used to plot the timeline execution of individual kernels for some of the discussed optimizations.

Transfer Contiguous Data

The first optimization under consideration relates to the transfer of boundary data between sub-domains. The high latency of data transferred to and from the GPU mandates that the number of transfers made be minimized to achieve good performance. Furthermore, large transfers amortize the latency cost. However, the underlying implementation set by the compiler only transfers data in contiguous chunks. Therefore, the transfer of data along a highly non-contiguous set of boundary data induces a significant performance penalty since separate data transfers are scheduled for each contiguous chunk of memory.

Using a representative code that performs just the residual calculations, boundary condition updates, and the transfer of data along inter-block boundaries between GPUs, it was determined that transferring contiguous data is more important than minimizing the transfer of data between the CPU and the GPU. Figure 3.5 compares the data transfer performance between the full boundary, a contiguous data set, and a non-contiguous portion of the boundary, arbitrarily chosen to be 10% of the size of the full boundary. Despite transferring 10

times more data, the full boundary data transfer executes 3.6 times faster than the partial boundary data transfer. This knowledge implies that rather than just passing the solution data along an inter-block or periodic boundary, the boundary data along the entire face should be passed between the CPU and the GPU to maintain a contiguous data transfer.

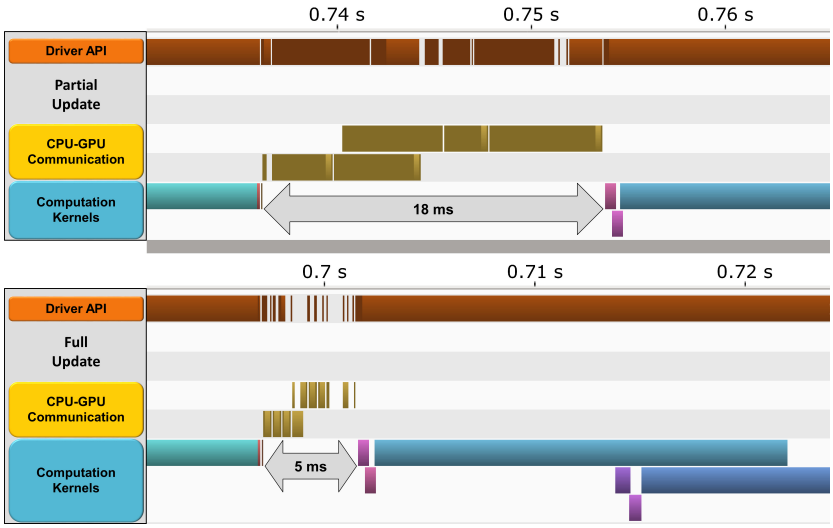


Figure 3.5: A CPU-GPU transfer of the entire boundary data is 3.6 times faster than the transfer of a non-contiguous portion of the boundary that is 10% of the full boundary size, due to the overhead of non-contiguous memory transfer.

The exception to this rule is for the i_{min} and i_{max} faces, when using Fortran, where all data along the face is non-contiguous; in this case, passing the minimum amount of data is the most efficient method. Figure 3.6 illustrates the difference in data transfer performance between the different domain faces. As shown, the j and k faces both transfer data close to a rate of 6.2 GB/s, whereas the i faces transfer data at a rate of 300 to 600 MB/s. This factor of ten difference in data transfer rate is attributed to non-contiguous data transfer. Although the j faces are not completely contiguous, as with the k faces, the amount of data transferred within each contiguous chunk is still sufficiently large to mask the latency cost of data transfer. To resolve this issue of low data transfer rates for the i faces, a buffer is generated to store the boundary data contiguously before transferring between the CPU and the GPU.

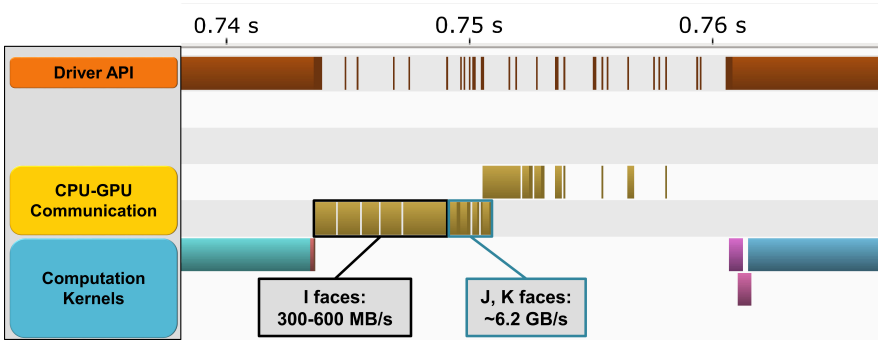


Figure 3.6: Data transfer rates for the i_min and i_max faces are over ten times slower than the data transfer rates for other faces, due to a highly non-contiguous memory storage pattern.

Enforce Boundary Conditions on the CPU

Although the boundary conditions were initially enforced on the GPU to avoid transferring more boundary data to the CPU, computing the boundary enforcement on the CPU instead of the GPU produced a 32% increase in performance. This demonstrates that the structure and the numerical algorithms of the boundary enforcement are not well-suited for parallel execution on the GPU. When enforcing the boundary conditions entirely on the CPU, all boundary data, including on the i_{min} and i_{max} faces, must be passed between the CPU and the GPU, further emphasizing the importance of a mitigation for the highly non-contiguous data transfer on these faces.

Reduce the Level of Loop Collapsing

Multiple kernels make use of loop collapsing to optimize performance. However, we found that for both two-dimensional and three-dimensional problems, better performance is generally obtained by only collapsing the outer loop(s) and applying a loop directive on the innermost loop. This structure permits the programmer to enforce finer-grained parallelism for the innermost loop with the use of vector or worker parallelism. Using the finer-grained parallelism of the GPU better exploits the performance capabilities of the GPU and masks the GPU latencies more effectively. Applying this optimization to the flux calculation ker-

nels, within the residual calculation procedure, produces a 620% increase in performance. Furthermore, this optimization produces an 90% increase in performance when applied to the solution update kernel. To understand why this optimization provides such significant performance improvements, we must look at the time calculation kernel.

In contrast, applying this optimization to the time step calculation kernel, a similar kernel structure, provides almost no benefit to the solver's performance. However, an array operation within the time step calculation kernel was already transformed to an explicit loop structure with an explicit `$acc loop seq` directive to prevent vector parallelism from being applied to this small loop. Otherwise, an error would occur in the time step calculation. Therefore, the cause of performance degradation in the other kernels is attributed to the implicit use of vector parallelism by OpenACC on small vector computations within the kernel rather than using vector parallelism for the primary kernel loops. This means that all array operations within the flux calculation kernels and the solution update kernel could similarly be transformed to explicit loops to achieve the same performance gains; however, reducing the levels of loop collapsing is a less intrusive method of achieving better performance, allowing the programmer to still use Fortran's shorthand notation of array operations.

Fuse Array Operations into Nested Loop Structure

Similarly, the residual calculation kernel exhibited better performance by eliminating array operations within the innermost loop. This modification is accomplished by adding an additional inner loop to perform computation of the residual for each equation individually. Figure 3.7 illustrates this optimization. A performance improvement of 35% was obtained by collapsing all four loops for this optimized kernel.

```
!$acc loop collapse(3)
do k = 1, k_cells
 do j = 1, j_cells
  do i = 1, i_cells

   residual(:,i,j,k) =                  &
    i_flux(:,i+1,j,k)*i_area(i+1,j,k) - &
    i_flux(:,i  ,j,k)*i_area(i  ,j,k) + &
    ...

  end do
 end do
end do
```

(a) Residual calculation loop before optimization.

```
!$acc loop collapse(4)
do k = 1, k_cells
 do j = 1, j_cells
  do i = 1, i_cells
   do n = 1, n_eq
    residual(n,i,j,k) =                  &
     i_flux(n,i+1,j,k)*i_area(i+1,j,k) - &
     i_flux(n,i  ,j,k)*i_area(i  ,j,k) + &
     ...
   end do
  end do
 end do
end do
```

(b) Residual calculation loop after optimization.

Figure 3.7: The residual calculation loop is optimized by fusing array operations into the nested loop structure.

Inline Subroutines

As mentioned in Section 3.3.2, function calls within OpenACC kernels limit asynchronous execution of the kernels, reducing performance. However, even numerous calls to subroutines within a kernel can limit performance, especially if multiple arrays are created within the scope of the subroutine with minimal computation performed to outweigh the cost of array allocation. A subroutine for computing the flux at the domain boundaries demonstrated this issue and had to be manually inlined, as shown in Figure 3.8. With the boundary flux calculation subroutine inlined, the contained arrays are declared within the scope of the residual calculation subroutine and these arrays are declared as private for each kernel thread. This optimization increases performance by 20% on the GPU, yet decreases performance on the CPU by approximately 5%. Therefore, preprocessing directives are used to manually inline the function only when accelerating on the GPU.

```
  double precision :: rho_ghost (5)
  double precision :: vel_ghost (3,5)
  double precision :: p_ghost   (5)
  double precision :: temp_ghost(5)
  ...

  ! Calculate i-min boundary fluxes
  !$acc loop
  do k = k_min, k_max
  !$acc loop worker private(rho_bound, vel_bound, p_bound, temp_bound)
  do j = j_min, j_max
    ...
#ifdef _OPENACC
    !$acc loop vector
    do n = 1, length
      rho_bound (n)   = rho (i_low+n-1,j,k)
      vel_bound (:,n) = vel (:,i_low+n-1,j,k)
      p_bound   (n)   = p   (i_low+n-1,j,k)
      temp_bound(n)   = temp(i_low+n-1,j,k)
    end do

    call set_bcflux(rho_bound, vel_bound, p_bound, temp_bound, i_flux(:,i,j,k))
#else
    call bc_flux_imin(j, k, i_flux(:,i,j,k))
#endif
    ...
  end do
```

Figure 3.8: The boundary flux calculation loop is optimized on the GPU by manually inlining the subroutine.

Eliminate Private Arrays

With the addition of private arrays for the manually inlined subroutine, it is desirable to reduce the total number of private variables within kernels. The computed flux was originally stored in a private variable before being assigned to the flux array. Directly assigning the flux to the flux array removed the need for this private variable. Meager performance gains of only 2% were observed for this optimization on the GPU, indicating a lack of sensitivity to the number of private array variables, at least for some kernels.

Change Loop Ordering

Finally, performance optimization was attempted by switching the order of the flux calculation loops. Instead of computing the fluxes in the order of ξ-fluxes, η-fluxes, and ζ-fluxes, the authors' experience has demonstrated that computing fluxes in the reverse order can improve the performance of the program execution. The reason for this performance improvement is that the more non-contiguous memory accesses in the ζ- and η-flux calculation loops causes these loops to execute more slowly than the ξ-flux calculation loop. Oftentimes, better overlap between asynchronously executing kernels is achieved by starting the most work-intensive kernels first. However, for this program only a performance improvement of 1% was observed.

A summary of the optimizations considered in this study are listed in Table 3.6. Figure 3.9 compares the effects of these optimizations on the performance of SENSEI with GPU acceleration. The top chart in the figure shows the entire results, while the bottom chart focuses on the smaller optimizations. These results were obtained on the T7600 workstation for the NACA 0012 test case on the 896×256-cell grid and the LDC test case on the 64^3-cell grid. Note that optimizations 1 and 5, marked out with cross marks, were not able to be tested for the LDC grid because the lack of these optimizations generates errors for three-dimensional grids. This figure illustrates that the most effective optimizations were

collapsing only two loop levels for the flux calculation kernels and the solution update kernel. Furthermore, the results for the LDC test case indicate that optimizations 3 and 4 were less effective for three-dimensional problems by approximately 30%. However, optimization 2 was more effective for the LDC test case by approximately 3%.

Table 3.6: Listing of optimizations for Figure 3.9.

1	Update boundary solution on the CPU
2	Only collapse two loop levels for residual calculation kernels
3	Only collapse two loop levels for solution update kernel
4	Add fourth inner loop to source term/residual norm calculation kernels
5	Manually inline boundary flux calculation subroutine
6	Eliminate flux private variable
7	Switch loop ordering

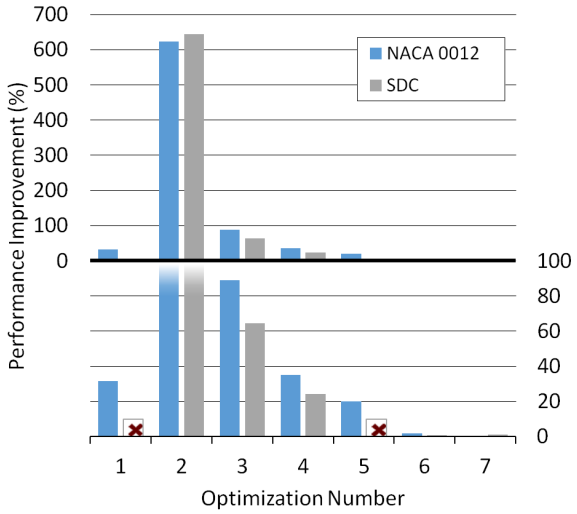


Figure 3.9: Effect of optimizations on OpenACC performance. The top chart illustrates the full results, whereas the lower chart focuses on the smaller optimizations.

3.4.3 Multi-GPU Analysis

Using the MPI implementation discussed in Section 3.3, the single-GPU implementation of SENSEI is extended to a multi-GPU implementation. The following section analyzes the

scalability and performance of this optimized implementation. The strong scalability analysis maintains a fixed total problem size with an increasing number of processes. Conversely, the weak scalability analysis maintains a fixed problem size local to each process, so the total problem size is directly proportional to the number of processes. All performance results, both for the GPU and the CPU, are normalized relative to the slowest CPU performance for the same test case, which for all analyses in this study is the single CPU performance on the smallest grid size.

Since all performance optimizations were analyzed on the T7600 workstation, Figure 3.10 compares the performance of the workstation with the performance of the HokieSpeed and NewRiver supercomputers at Virginia Tech. These results are based on the normalized performance of the multi-CPU and multi-GPU implementations of SENSEI for the LDC test case.

Figure 3.10 shows that the CPU performance increases by 30% to 33% with increase in grid size from the 32^3-cell grid to the 128^3-cell grid. Furthermore, the performance of the workstation CPU is very similar to the performance of a NewRiver CPU, with less than 6% difference. A CPU on HokieSpeed is 38% to 43% slower than the CPU performance of the workstation. On the other hand, the GPU performance increases by 32% to 41% between the 32^3-cell grid and the 64^3-cell grid, maintaining close to the same performance for the 128^3-cell grid. Although a previous study has shown the performance of the NVIDIA C2075 GPU on the workstation and an NVIDIA M2050 GPU on HokieSpeed to be the same within 3% difference[26], the performance of the workstation in this study is approximately 8% to 15% greater than the performance of HokieSpeed for the single GPU case. This discrepancy is due to performing the boundary enforcement on the CPU, meaning that the CPU performance has a greater effect on the overall performance of the GPU-accelerated version of SENSEI. The NVIDIA K80 GPU on NewRiver demonstrates 62% to 68% better performance than the workstation, illustrating the improvement in performance that the new Kepler architecture provides over the Fermi architecture of the workstation's and HokieSpeed's GPUs for a complex, CFD code.

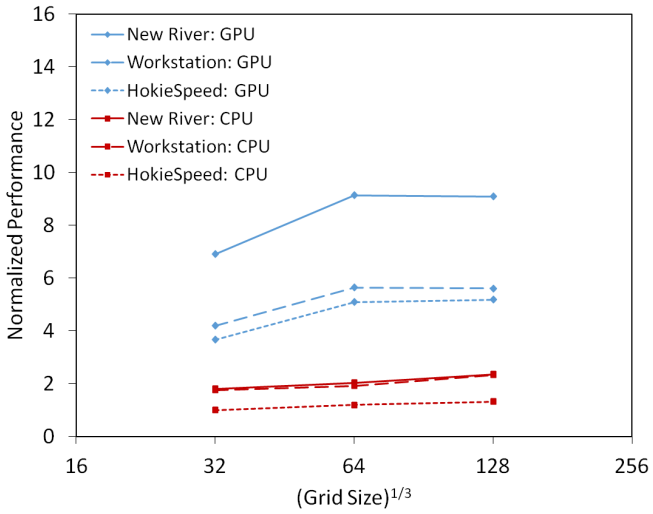


Figure 3.10: Comparison of SENSEI's single CPU and single GPU performance on multiple system architectures.

NACA 0012 Airfoil Test Case

The grids used for the NACA 0012 airfoil test case are single-block, but non-uniform to provide better resolution near the airfoil surface. There are 3.5 times more cells in the i direction; therefore, SENSEI chooses two-dimensional decompositions with more subdivisions in the i direction. The decompositions used for this analysis are listed in Table 3.7. Decompositions are identified by the number of blocks in the i, j, and k directions, respectively. A decomposition of 30 processes is used instead of 32 processes because a decomposition of 30 processes better satisfies the first two decomposition metrics, as described in Section 3.3.3, by decreasing the surface area to volume ratio.

Figure 3.11 illustrates the performance of the NACA 0012 test case over the range of grid sizes specified in Section 3.4.1. The multi-CPU performance increases by 8% to 15% as the grid size is increased. Relative to the multi-GPU performance, this performance increase is minor. The multi-GPU performance increases by 100% for a single GPU and by 250% for four GPUs. The h_4 grid is too small to be divided into 8 or 16 sub-domains, so only the

Table 3.7: Domain decompositions used for the NACA 0012 airfoil test case.

Number of Processes	Decomposition ($i \times j \times k$)
2	$2 \times 1 \times 1$
4	$4 \times 1 \times 1$
8	$4 \times 2 \times 1$
16	$8 \times 2 \times 1$
30	$10 \times 3 \times 1$

h_1 and h_2 grids are analyzed for these results. With 16 GPUs, the performance is up to 36 times greater than the serial code and 2.5 times greater than 16 CPUs, run on separate sockets.

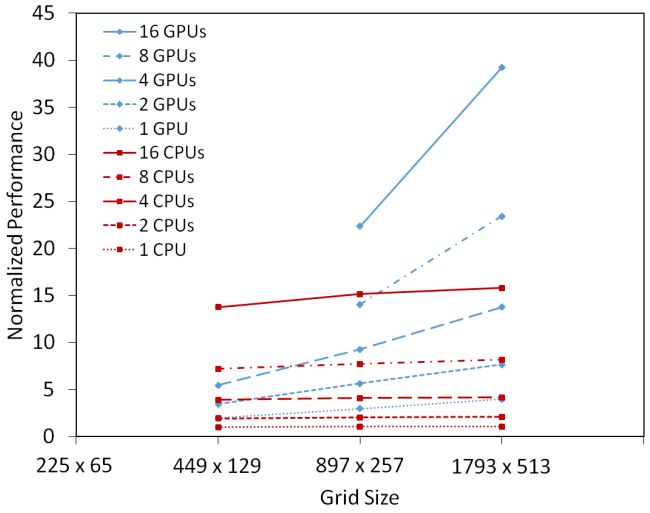
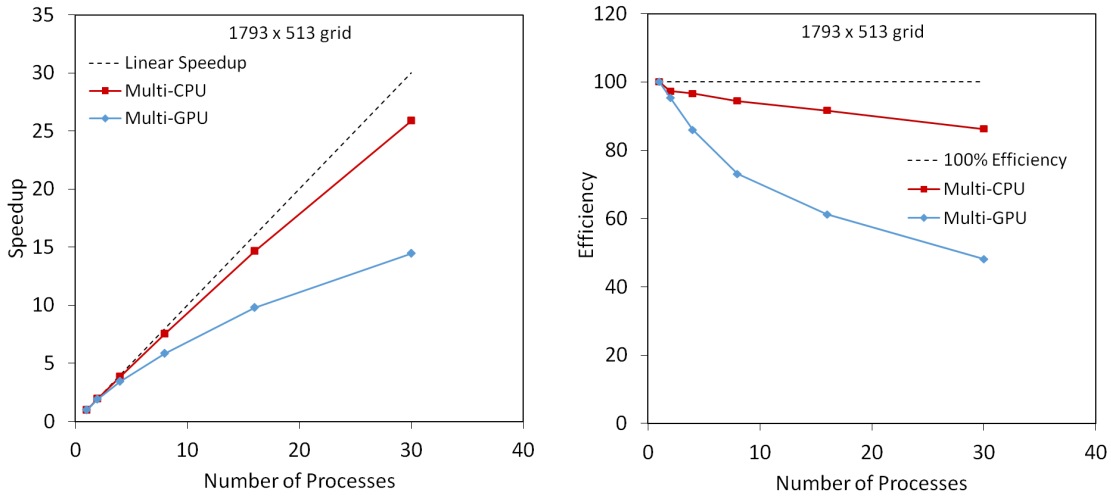


Figure 3.11: Performance of SENSEI for the NACA 0012 test case over a range of grid sizes.

Figure 3.12 illustrates the strong scalability results of multi-CPU and multi-GPU SENSEI. The h_1 grid is used as the fixed domain grid size for these results. The multi-CPU implementation demonstrates an efficiency of 86% with 30 CPUs, while the multi-GPU implementation demonstrates only an efficiency of 48% with 30 GPUs. However, GPUs inherently are not designed for strong scalability, as the performance results in Figure 3.11 illustrate that the performance drastically decreases as the problem size, local to a given GPU, decreases.



(a) Strong scalability speedup.

(b) Strong scalability efficiency.

Figure 3.12: Strong scalability results for the NACA 0012 test case.

Lid-driven Cavity (LDC) Test Case

The LDC test case uses single-block, uniform, three-dimensional grids. This simple test case allows for a more detailed analysis of scalability and performance than more complex multi-block grids, such as for the M6 Onera wing, since load-balancing is an issue for multi-block grids. The decompositions used for this analysis are listed in Table 3.8. Again, the decompositions are selected in an effort to best satisfy the decomposition metrics, as discussed in Section 3.3.3. Since the grid is the same size in all dimensions, a 3D decomposition with the same number of sub-divisions in every dimension provides the most scalable decomposition with large numbers of processes, as it is less limited than 1D or 2D decompositions by the number of cells in each dimension. This decomposition also minimizes the surface area to volume ratio of the computational sub-domains, reducing the total data communicated between processes. Table 3.8 shows that this decomposition is fully achieved for 8 processes and 27 processes.

For this test case the multi-CPU performance, as shown in Figure 3.13, increases by 32%

Table 3.8: Domain decompositions used for the LDC test case.

Number of Processes	Decomposition ($i \times j \times k$)
2	$2 \times 1 \times 1$
4	$2 \times 2 \times 1$
8	$2 \times 2 \times 2$
16	$4 \times 2 \times 2$
27	$3 \times 3 \times 3$

for a single CPU and by 53% for 16 CPUs with increase in grid size. The multi-GPU implementation increases by 41% for a single GPU and by 250% for four GPUs. Although the multi-CPU performance increase is still smaller than the GPU performance increase with increase in grid size, it is no longer negligible. Similar to the NACA 0012 test case, the smallest grid size is too small to run with more than four sub-domains. Furthermore, the 64^3-cell grid is too small to run with 27 processes. On the other hand, the largest grid size that a single GPU can hold in memory with SENSEI is the 128^3-cell grid, so the 256^3 grid is unable to fit in the GPU's memory until is is split into at least 8 sub-domains. The performance of 27 GPUs is up to 100 times greater than the serial version of SENSEI and 4.1 times greater than the performance of 27 CPUs.

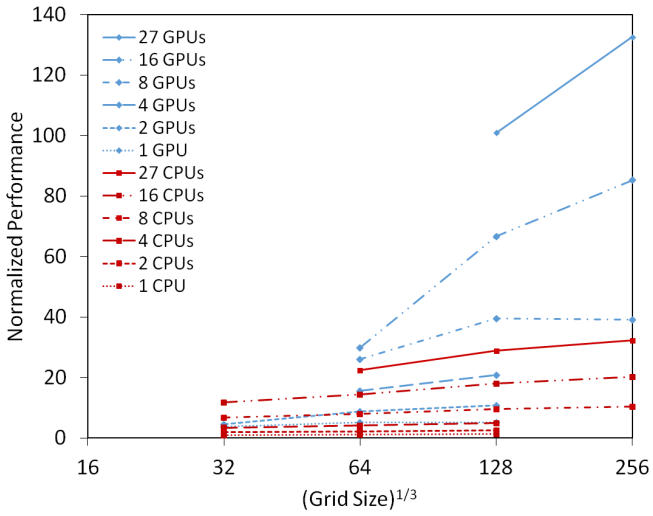
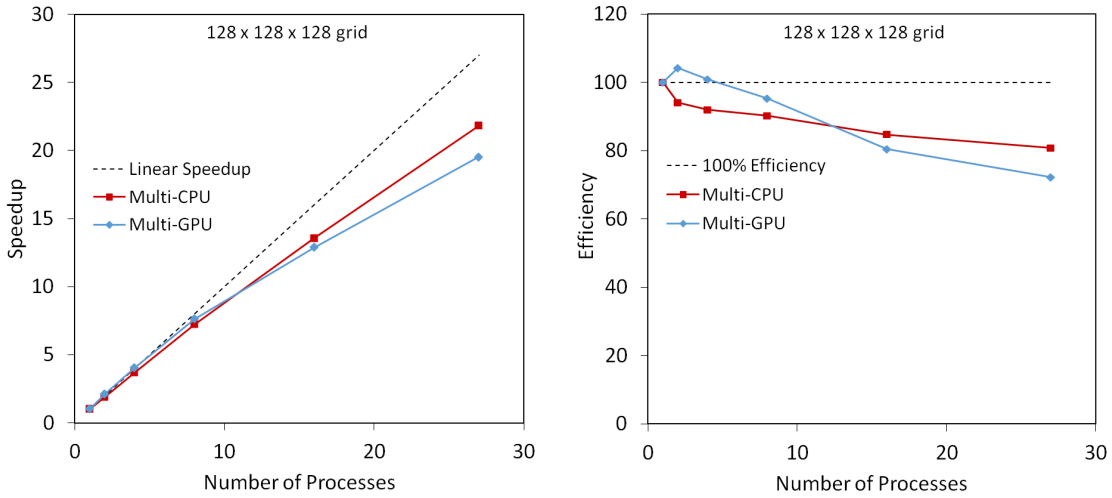


Figure 3.13: Performance of SENSEI for the LDC test case over a range of grid sizes.

The strong scalability results for the LDC test case, shown in Figure 3.14, demonstrate that the multi-GPU implementation actually exhibits fairly good scalability. The multi-GPU implementation runs with greater efficiency than the multi-CPU implementation up to eight processes and demonstrates 104% efficiency with two GPUs. The multi-GPU implementation maintains 72% efficiency with 27 GPUs, whereas the multi-CPU implementation achieves 81% efficiency with 27 CPUs. Although the efficiency of the multi-GPU implementation does continue to drop with increase in process count, the high efficiency results relative to the NACA 0012 test case are attributed to using the maximum amount of memory possible on the GPU. This means that even with decomposing the domain into 27 sub-domains, the sub-domains are large enough to prevent a significant loss of efficiency, unlike the NACA 0012 test case.



(a) Strong scalability speedup.

(b) Strong scalability efficiency.

Figure 3.14: Strong scalability results for the LDC test case.

A weak scalability analysis is run for the LDC test case, where the grid local to each GPU is maintained at a constant size. These results are illustrated in Figure 3.15. For a fixed local grid size of 32^3 cells, the multi-GPU implementation maintains 81% efficiency, whereas the multi-CPU implementation maintains an efficiency of 93%. The efficiency drops when using 27 sub-domains, likely because that decomposition is the only one tested with a sub-domain

that must communicate with neighbors on all of its faces. This seems to affect the multi-CPU performance as much as the multi-GPU performance, as the efficiency drops from 99% to 93% for the multi-CPU implementation and from 89% to 81% efficiency for the multi-GPU implementation. This issue seems to affect the multi-CPU performance even more drastically for the fixed local grid size of 64^3 cells, as the efficiency drops from 99.9% to 88% between the use of 8 CPUs and 27 CPUs. The multi-GPU implementation, however, seems to be less affected for the larger grid size, maintaining an efficiency of 95% with 27 GPUs.

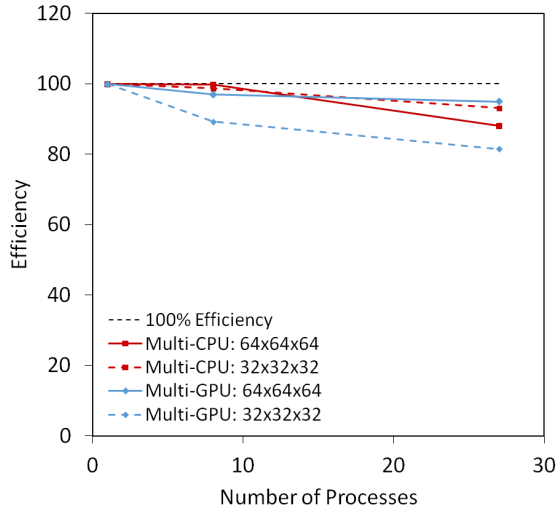


Figure 3.15: Weak scalability efficiency results for the LDC test case.

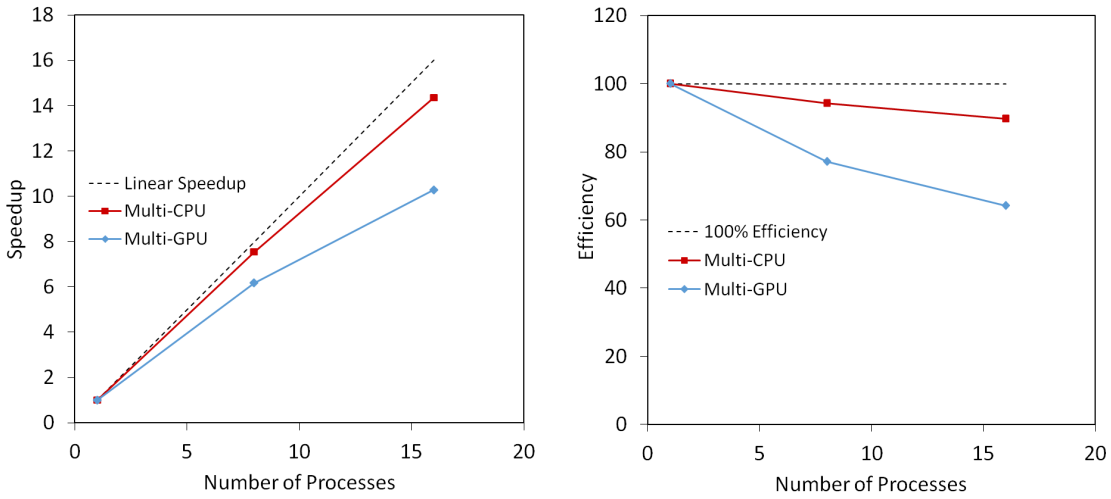
M6 Onera Wing Test Case

The final test case, the M6 Onera wing, uses a multi-block, structured grid. The block numbers, associated with the block dimensions, are listed in Table 3.9. Only a single grid size is used, so only a strong scalability analysis is performed for this test case. Due to the grid complexity, the decomposition with a given number of processes is no longer guaranteed to be load-balanced. Table 3.10 lists the decompositions used for this analysis, as well as the load imbalance of these decompositions. The load imbalance metric defines the percent increase in cells of the largest sub-domain over the average sub-domain size.

Table 3.9: Grid blocks for the M6 Onera test case.

Block Number	Block Dimensions
1	$24 \times 48 \times 32$
2	$72 \times 48 \times 32$
3	$72 \times 48 \times 32$
4	$24 \times 48 \times 32$

Figure 3.16 illustrates the strong scalability results for the M6 Onera test case, ignoring load-imbalanced decompositions. This figure demonstrates a smooth curve for the decrease in efficiency of the multi-CPU and multi-GPU implementations. The efficiency using 16 CPUs is 90%, whereas the efficiency using 16 GPUs is 64%. Again, these results reflect the expectation of good strong scalability performance for the multi-CPU implementation, yet poor strong scalability performance for the multi-GPU implementation.



(a) Strong scalability speedup.

(b) Strong scalability efficiency.

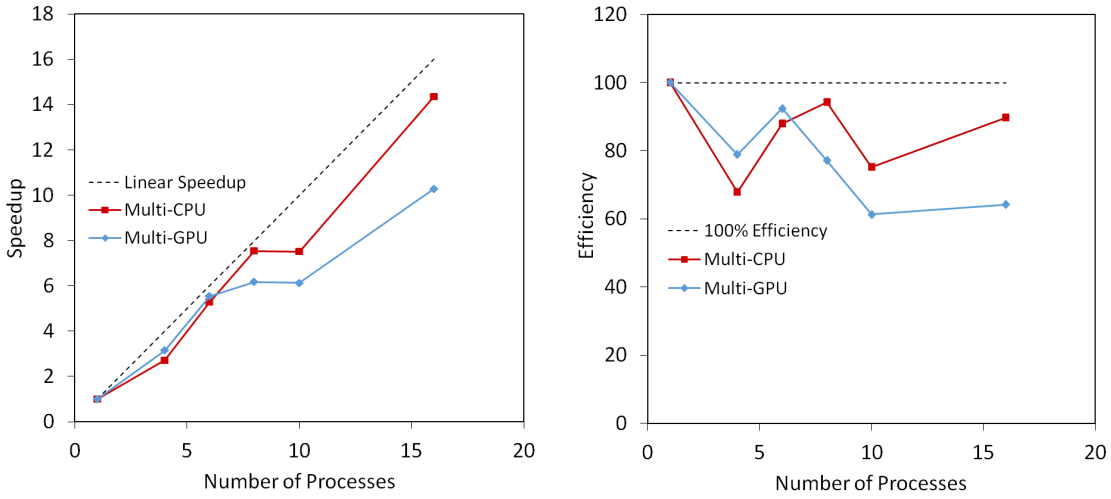
Figure 3.16: Strong scalability results for the M6 Onera wing, excluding results with imbalanced loads.

Including the load-imbalanced decompositions produces the results shown in Figure 3.17. The speedup and efficiency curves no longer indicate smooth or consistent trends with increase in the number of processes. Considering the smoothness of the curves in Figure 3.16,

Table 3.10: Domain decompositions used for each of the four blocks in the M6 Onera test case.

Number of Processes	Decomposition ($i \times j \times k$)	Load Imbalance (%)
4	$1 \times 1 \times 1$ $1 \times 1 \times 1$ $1 \times 1 \times 1$ $1 \times 1 \times 1$	50.0
6	$1 \times 1 \times 1$ $2 \times 1 \times 1$ $2 \times 1 \times 1$ $1 \times 1 \times 1$	12.5
8	$1 \times 1 \times 1$ $3 \times 1 \times 1$ $3 \times 1 \times 1$ $1 \times 1 \times 1$	0.0
10	$1 \times 1 \times 1$ $2 \times 2 \times 1$ $2 \times 2 \times 1$ $1 \times 1 \times 1$	25.0
16	$1 \times 2 \times 1$ $3 \times 2 \times 1$ $3 \times 2 \times 1$ $1 \times 2 \times 1$	0.0

this decrease in consistency is caused by the load imbalance of these decompositions degrading the performance. Interestly, six GPUs actually demonstrate a higher efficiency than expected from Figure 3.16.



(a) Strong scalability speedup.

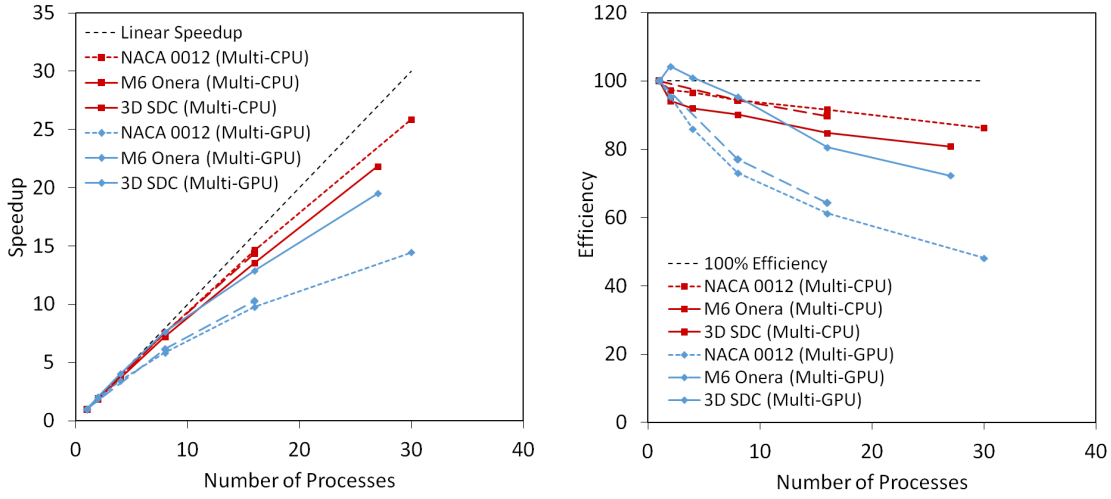
(b) Strong scalability efficiency.

Figure 3.17: Strong scalability results for the M6 Onera wing, including results with imbalanced loads.

Comparison of Results

Finally, we desire to compare the scalability results between the different test cases. Figure 3.18 illustrates the differences in strong scalability for all test cases. Interestingly, the LDC test case has the worst multi-CPU scalability results, yet the best multi-GPU scalability results. The better scalability on the GPU is attributed to the larger grid size for the LDC test case, compared with the grids used for the strong scalability analyses of the other test cases. The grid size used for the LDC test case is 128^3 cells, the maximum possible grid size that fits in a single GPU's memory. The NACA 0012 grid used for strong scalability analysis is the h_1 grid, which is approximately 44% of the LDC grid size. Furthermore, only the Euler equations are solved for the NACA 0012 grid, so all of the viscous subgrid data is not needed, further reducing memory usage by 55%. In addition, the M6 Onera grid is approximately 7.1 times smaller than the LDC grid. Therefore, it should be anticipated that the strong scalability efficiency is lower for these test cases, since the GPU's performance is significantly affected by the grid size. Therefore, to achieve better GPU performance

the memory size should be increased to accommodate even larger grid sizes, as this would improve strong scalability as well as performance. The design of new GPU architectures such as the NVIDIA Tesla K80 on NewRiver supports this statement, as a single K80 GPU holds 24 GB of memory[33], rather than just 3 GB with the NVIDIA Tesla M2050[32].



(a) Strong scalability speedup.

(b) Strong scalability efficiency.

Figure 3.18: Comparison of the strong scalability results for the different test cases.

3.5 Conclusions

In this paper, we investigated the use of OpenACC to port the SENSEI CFD code from a CPU implementation to a multi-GPU implementation. We discussed the challenges in porting the code to the GPU and the optimizations exploited to improve GPU performance, while maintaining a single code base for the CPU and GPU implementations. Multiple optimizations were found to improve the performance of the OpenACC code. The most effective optimization discussed was the reduction of levels of loop collapsing for some kernels to prevent vector parallelism being only applied to small inner loops, providing in total over 700% increase in performance. Other useful optimizations for SENSEI included enforcing the boundary conditions on the CPU and adding an inner loop to some kernels to

combine multiple array operations, increasing performance by 32% and 35%, respectively. The performance results demonstrate that a single NVIDIA Tesla M2050 GPU attains a 3.7 speedup for the airfoil, 4.25 speedup for the lid-driven cavity, and 3.2 speedup for the wing over a single Intel Xeon E5645 processor core. The performance of 27 GPUs is up to 100 times faster than a single core and 4.1 times faster than 27 cores for the lid-driven cavity. The multi-GPU implementation also maintains an efficiency of 95% up to 27 GPUs with the lid-driven cavity flow. These results demonstrate the capability of OpenACC to develop portable, accelerated codes from a base CPU implementation with minimal modifications, even when the base implementation is a large and complex CFD code.

3.6 Future Work

The primary area of interest for further development would be in running these tests for larger-scale GPU clusters. Multiple design considerations, including the domain decomposition are more important for larger scale parallelism. Furthermore, having access to a large GPU cluster with the latest K80 GPUs would allow for scalability and performance testing of one of the latest GPU designs. With the increased memory capacity and compute capability, we believe the results of this study would be even further enhanced.

Acknowledgments

The authors acknowledge the support of the Virginia Tech Synergistic Environments for Experimental Computing (SEEC) Center for this research. This work was supported in part by the HokieSpeed supercomputer at Virginia Tech through NSF grant CNS-0960081.

References

[1] Y. Komura. "OpenACC programs of the Swendsen–Wang multi-cluster spin flip algorithm". In: *Computer Physics Communications* (Aug. 2015). ISSN: 0010-4655. DOI: 10.1016/j.cpc.2015.08.022. URL: http://dx.doi.org/10.1016/j.cpc.2015.08.022.

[2] M. Misic, D. Dasic, and M. Tomasevic. "An analysis of OpenACC programming model: Image processing algorithms as a case study". In: *Telfor Journal* 6.1 (2014), pp. 53–58. ISSN: 1821-3251. DOI: 10.5937/telfor1401053m. URL: http://dx.doi.org/10.5937/telfor1401053M.

[3] M. Otten et al. "An MPI/OpenACC Implementation of a High Order Electromagnetics Solver with GPUDirect Communication". In: *The International Journal of High Performance Computing Applications* 30.3 (2016), pp. 320–334. DOI: 10.1177/1094342015626584.

[4] S. Markidis et al. "OpenACC acceleration of the Nek5000 spectral element code". In: *International Journal of High Performance Computing Applications* 29.3 (Mar. 2015), pp. 311–319. ISSN: 1741-2846. DOI: 10.1177/1094342015576846. URL: http://dx.doi.org/10.1177/1094342015576846.

[5] R. Xu et al. "Multi-GPU Support on Single Node Using Directive-Based Programming Model". In: *Scientific Programming* 2015 (2015), pp. 1–15. ISSN: 1875-919X. DOI: 10.1155/2015/621730. URL: http://dx.doi.org/10.1155/2015/621730.

[6] M. Chrust, E. Laurendeau, and L. Ostiguy. "Accelerating low-fidelity aerodynamic codes on multi- and many-core architectures". In: *The Journal of Supercomputing* 71.9 (May 2015), pp. 3456–3481. ISSN: 1573-0484. DOI: 10.1007/s11227-015-1444-6. URL: http://dx.doi.org/10.1007/s11227-015-1444-6.

[7] C. Couder-Castaneda et al. "Performance of a Code Migration for the Simulation of Supersonic Ejector Flow to SMP, MIC, and GPU Using OpenMP, OpenMP+LEO, and OpenACC Directives". In: *Scientific Programming* 2015 (2015), pp. 1–20. ISSN: 1875-919X. DOI: 10.1155/2015/739107. URL: http://dx.doi.org/10.1155/2015/739107.

[8] T. Komoda et al. "Integrating Multi-GPU Execution in an OpenACC Compiler". In: *Parallel Processing (ICPP), 2013 42nd International Conference on*. 2013, pp. 260–269. DOI: 10.1109/ICPP.2013.35. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6687359.

[9] B. P. Pickering et al. "Directive-based GPU programming for computational fluid dynamics". In: *Computers & Fluids* 114 (July 2015), pp. 242–253. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2015.03.008. URL: http://dx.doi.org/10.1016/j.compfluid.2015.03.008.

[10] Y. Xia et al. "OpenACC acceleration of an unstructured CFD solver based on a reconstructed discontinuous Galerkin method for compressible flows". In: *Int. J. Numer. Meth. Fluids* 78.3 (Feb. 2015), pp. 123–139. ISSN: 0271-2091. DOI: 10.1002/fld.4009. URL: http://dx.doi.org/10.1002/fld.4009.

[11] NVIDIA. *CUDA*. http://www.nvidia.com/object/cuda_home_new.html.

[12] The Khronos Group. *The open standard for parallel programming of heterogeneous systems*. https://www.khronos.org/opencl/.

[13] *The OpenACC API: Version 2.0*. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf.

[14] *The OpenMP API specification for parallel programming*. http://openmp.org/wp/.

[15] L. Luo, J. R. Edwards, and H. Luo. "Performance Assessment of Multi-block LES Simulations using Directive-based GPU Computation in a Cluster Environment". In: *52nd Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics (AIAA), Jan. 2014. DOI: 10.2514/6.2014-1130. URL: http://dx.doi.org/10.2514/6.2014-1130.

[16] T. Hoshino et al. "CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application". In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. Institute of Electrical and Electronics Engineers (IEEE), May 2013. DOI: 10.1109/ccgrid.2013.12. URL: http://dx.doi.org/10.1109/CCGrid.2013.12.

[17] L. Luo et al. "GPU Port of A Parallel Incompressible Navier-Stokes Solver based on OpenACC and MVAPICH2". In: *7th AIAA Theoretical Fluid Mechanics Conference*. American Institute of Aeronautics and Astronautics (AIAA), June 2014. DOI: 10.2514/6.2014-3083. URL: http://dx.doi.org/10.2514/6.2014-3083.

[18] A. Corrigan and R. Lohner. "Semi-automatic porting of a large-scale CFD code to multi-graphics processing unit clusters". In: *International Journal for Numerical Methods in Fluids* 69.11 (Aug. 2011), pp. 1786–1796. DOI: 10.1002/fld.2664. URL: http://dx.doi.org/10.1002/fld.2664.

[19] A. Corrigan et al. "Semi-automatic porting of a large-scale Fortran CFD code to GPUs". In: *International Journal for Numerical Methods in Fluids* 69.2 (May 2011), pp. 314–331. DOI: 10.1002/fld.2560. URL: http://dx.doi.org/10.1002/fld.2560.

[20] B. van Werkhoven and P. Hijma. "An Integrated Approach to Porting Large Scientific Applications to GPUs". In: *2015 IEEE 11th International Conference on e-Science*. Institute of Electrical and Electronics Engineers (IEEE), Aug. 2015. DOI: 10.1109/escience.2015.23. URL: http://dx.doi.org/10.1109/eScience.2015.23.

[21] J. M. Derlaga, T. Phillips, and C. J. Roy. "SENSEI Computational Fluid Dynamics Code: A Case Study in Modern Fortran Software Development". In: *21st AIAA Computational Fluid Dynamics Conference* (June 2013). DOI: 10.2514/6.2013-2450. URL: http://dx.doi.org/10.2514/6.2013-2450.

[22] H. Hirsch. *Numerical computation of internal and external flows: Computational Methods for Inviscid and Viscous Flows*. Vol. 2. John Wiley & Sons, 1990, pp. 536–556.

[23] N. Corporation. *PGI Accelerator Compilers with OpenACC Directives*. https://www.pgroup.com/resources/accel.htm.

[24] OpenACC.org. *OpenACC 2.6 Proposed Features*. http://www.openacc.org/sites/default/files/OpenACC26-proposed_features.pdf.

[25] NVIDIA Corporation. *PGI Accelerator Compilers OpenACC Getting Started Guide*. https://www.pgroup.com/doc/openacc_gs.pdf.

[26] B. Baghapour, A. J. McCall, and C. J. Roy. "Multilevel Parallelism for CFD Codes on Heterogeneous Platforms". In: *46th AIAA Fluid Dynamics Conference*. American Institute of Aeronautics and Astronautics (AIAA), June 2016. DOI: 10.2514/6.2016-3329. URL: http://dx.doi.org/10.2514/6.2016-3329.

[27] Advanced Research Computing at Virginia Tech. *HokieSpeed (CPU/GPU)*. https://secure.hosting.vt.edu/www.arc.vt.edu/hokiespeed-cpugpu/.

[28] Intel. *Intel Xeon Processor E5-2687W*. http://ark.intel.com/products/64582/Intel-Xeon-Processor-E5-2687W-20M-Cache-3_10-GHz-8_00-GTs-Intel-QPI.

[29] Intel. *Intel Xeon Processor E5645*. http://ark.intel.com/products/48768/Intel-Xeon-Processor-E5645-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI.

[30] Intel. *Intel Xeon Processor E5-2680*. http://ark.intel.com/products/64583/Intel-Xeon-Processor-E5-2680-20M-Cache-2_70-GHz-8_00-GTs-Intel-QPI.

[31] www.techpowerup.com. *NVIDIA Tesla C2075*. https://www.techpowerup.com/gpudb/563/tesla-c2075.

[32] www.techpowerup.com. *NVIDIA Tesla M2050*. https://www.techpowerup.com/gpudb/1534/tesla-m2050.

[33] www.techpowerup.com. *NVIDIA Tesla K80m*. https://www.techpowerup.com/gpudb/2616/tesla-k80m.

[34] Advanced Research Computing at Virginia Tech. *NewRiver*. https://secure.hosting.vt.edu/www.arc.vt.edu/computing/newriver/.

[35] B. van Leer. "Towards the Ultimate Conservative Difference Scheme. V. A Second-order Sequel to Godunov's method." In: *Journal of Computational Physics* 32.1 (1979), pp. 101–136. URL: http://dx.doi.org/10.2514/6.2008-607.

[36] C. Michalak and C. Ollivier-Gooch. "Accuracy Preserving Limiter for the High-order Accurate Solution of the Euler Equations." In: *Journal of Computational Physics* 228.23 (2009), pp. 8693–8711. URL: http://dx.doi.org/10.2514/6.2008-607.

[37] G. van Albada, B. van Leer, and W. Roberts. "A comparative study of computational methods in cosmic gas dynamics". In: *Astronomy and Astrophysics* 108.1 (1982), pp. 76–84.

[38] John W. Slater. *ONERA M6 Wing: Study #1*. https://www.grc.nasa.gov/WWW/wind/valid/m6wing/m6wing01/m6wing01.html.

[39] N. Corporation. *NVIDIA Visual Profiler*. https://developer.nvidia.com/nvidia-visual-profiler.

Chapter 4

Discussion and Conclusions

Although the exact performance in GFLOPs is not obtained for SENSEI's results, some comparisons may be made between the scalability and performance results of these two chapters.

The weak scalability efficiency for the cubic, buoyancy-driven cavity in the second chapter remain as high as 99% using 32 GPUs with a grid size that consumes approximately 60% of the available memory on the GPU. In contrast, SENSEI attains a weak scalability efficiency of 95% using only 27 GPUs for a cubic, lid-driven cavity with a grid size that uses almost 100% of the available memory on the GPU. Comparing performance results, the code from the second chapter attained a factor of 400 speedup over the serial implementation using 32 GPUs. SENSEI, however, only manages to attain a factor of 100 speedup over the serial implementation for the lid-driven cavity, using 27 GPUs. Although five fewer GPUs are used when running SENSEI, this difference does not account for the factor of four discrepancy in performance speedup.

The reduced scalability and performance of SENSEI is attributed to its additional numerical and structural complexity. SENSEI is designed to handle a general structured, curvilinear, multi-block grid, as opposed to the code for the second chapter, which specifically solves the

lid-driven or buoyancy driven cavity with a uniform Cartesian grid. Furthermore, SENSEI uses second-order upwind flux schemes with MUSCL extrapolation and limiters, whereas the code for the second chapter is simply a finite-difference discretization with artificial compressibility and pressure rescaling to maintain a fixed pressure at the cavity center. In terms of code structure, SENSEI makes use of many modern Fortran constructs, most importantly including derived types containing allocatable arrays. The difficulty in correctly porting SENSEI's execution to the GPU is much greater than the difficulty in porting the code for the second chapter, which made use of very few modern Fortran constructs. This difficulty not only limited OpenACC's ability to accelerate performance, but also limited the options available for optimizing execution performance of SENSEI relative to the code used in the second chapter. Much more effort was dedicated to performance optimization in the second chapter, as fewer obstacles presented themselves and complicated the optimization process.

In these two chapters, MPI and OpenACC were used to accelerate performance of CFD codes. The remainder of this section outlines the conclusions made for this thesis regarding the viability of the OpenACC programming model.

In the second chapter, the transformation of a serial CPU code into a single-GPU implementation was shown to require minimal modification to the code base with the use of directive statements. In addition, the optimized OpenACC implementation produced a factor of 8 speedup over the serial CPU execution using an NVIDIA Tesla C2075 GPU and comparable performance to an MPI multi-CPU implementation using 8 cores of a dual-socket Xeon CPU workstation. A hybrid MPI+OpenACC multi-GPU implementation of the 3D BDC solver was developed and overlapping communication and computation was found to improve performance by up to 21%. Although using overlapping communication and computation reduced portability of the code, since the multi-CPU implementation attained better performance without this overlap of communication and computation, the performance improvement was substantial enough to offset this issue. Scalability and performance analyses of the multi-GPU implementation on the HokieSpeed supercomputer demonstrated excel-

lent weak scalability on the HokieSpeed supercomputer with up to 99% efficiency using 32 GPUs. Furthermore, the performance of the final multi-GPU implementation for the 512^3-node case with 32 GPUs was over 400 times faster than a single CPU and 11.2 times faster than 32 CPUs. These results demonstrate the capability of OpenACC to develop portable, accelerated codes, while maintaining the same baseline CPU implementation.

In the third chapter, the challenges in porting a large, complex code written in modern Fortran to the GPU were discussed. The performance results demonstrated that a single NVIDIA Tesla M2050 GPU attained a 3.7 speedup for a 2D NACA 0012 airfoil model, 4.25 speedup for a 3D, cubic, lid-driven cavity model, and 3.2 speedup for a 3D M6 Onera wing model over a single Intel Xeon E5645 processor core. SENSEI was extended to a hybrid MPI+OpenACC multi-GPU implementation, which was tested for scalability and performance. The multi-GPU implementation maintained a weak efficiency of 95% up to 27 GPUs with the lid-driven cavity flow and the performance of 27 GPUs was up to 100 times faster than a single core and 4.1 times faster than 27 cores for the lid-driven cavity. These results illustrate OpenACC's ability to port large and complex CFD codes to the GPU, maintaining a unified, portable code base while achieving good scalability and performance.

The results for the second chapter indicate a greater performance increase over the serial and multi-CPU implementations; however, the code for the second chapter is numerically and structurally less complex. As the complexity of the code increases, the performance and scalability of the GPU-accelerated implementation decreases. This behavior is similarly anticipated for low-level programming models; however, the development of a GPU-accelerated version of SENSEI without using OpenACC would be extremely difficult due to the level of refactoring required and the minimal implementation portability. Overall, these results demonstrate OpenACC is a viable and expedient programming model for the acceleration of codes on the GPU.

Appendix A

GPU Parallelism

In the Single Intruction, Multiple Threads (SIMT) architecture of the GPU, as described by NVIDIA[1], groups of threads perform a specific operation on a large data set, leading to a massively parallel execution of data. This massive parallelism masks the high latency of the processing units in accessing data from memory.

The SIMT architecture of the GPU combines aspects of the Single Instruction, Multiple Data (SIMD) vector processing architecture and Intel's Simultaneous Multi-Threading (SMT) hyper-threading architecture. The primary architectural difference is in the hardware design. The SIMD architecture makes use of a vector processing unit that operates on a single thread by executing a single instruction on multiple data values simultaneously. For example, this parallelism may be exploited when performing array operations in Fortran, where an operation is performed element-wise to the array. In contrast, the SIMT architecture executes multiple threads simultaneously to perform the equivalent operation of a vector processor.

On the other hand, the SMT architecture allows a single CPU core to effectively run multiple independent threads simultaneously. The use of multiple threads is similar to the SIMT architecture; however, in current Intel technology only makes use of two hyper-threads. The SIMT architecture makes use of hundreds to thousands of threads. The primary difference in

these architectures is that the SMT architecture is designed for already low-latency processing units, to mask any remaining low-latency overhead. Conversely, the SIMT architecture makes use of many thread groups to mask the high latency of the processing units.

Specifically for the NVIDIA Tesla GPU architecture, execution of a GPU kernel code is parallelized on three levels. Thread blocks are the first level of parallelism and are each mapped to a single streaming multiprocessor (SM) on the GPU. The Fermi architecture for NVIDIA Tesla GPUs contains 16 SMs, as shown in Figure A.1a. Each SM contains many cores that may execute simultaneously. The Fermi architecture contains 32 of these cores within an SM, as shown in Figure A.1b. In the second level of parallelism, multiple warps may execute within each thread block. Warp schedulers within the SM handle switching execution between warps to mask the high latency of the processing units. This warp scheduling is similar to the context switching used with the SMT architecture. Finally, within each warp the threads within the SM are executed simultaneously, similar to the vector processing of the SIMD architecture. These levels of parallelism, thread blocks, warps, and threads, are synonymous to the OpenACC terminology of gangs, workers, and vectors, respectively.

(a) Illustration of full Fermi architecture processing die.

(b) Illustration of a single Fermi architecture SM.

Figure A.1: The Fermi architecture for NVIDIA Tesla GPUs.[2]

Overall, the design of the GPU relies on high occupany (high usage of the threads) to attain good performance and effectively mask the high latency of the GPU architecture. However, this high occupancy also potentially produces a much greater throughput than the CPU, even when using either the SIMD or SMT architecture. Therefore, GPU computing is ideal for accelerating performance of scientific computing applications where a set of operations are performed on a large domain set.

References

[1] NVIDIA Corporation. *NVIDIA"s Next Generation CUDA Compute Architecture: Fermi.* `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[2] NVIDIA Corporation. *White Paper: NVIDIA GF100.* `www.nvidia.com/object/IO_86775.html`.

Appendix B

SENSEI MPI Implementation

B.1 Serial Implementation Theory

The theoretical performance of the serial implementation is outlined in Table B.1. The performance analysis is based on a value n, which is on the same order of the dimensions of the computational domain. In other words, if N is on the order of the total number of cells in the domain, then $O(n) = O(N^{1/3})$. This assumes the problem is three-dimensional. This choice for n is selected because typically when trying to increase the accuracy of a solution, the grid is systematically and consistently refined in all dimensions. Therefore, if one level of grid refinement leads to an increase of both dimensions by a factor of 2, then n also increases by a factor of 2, whereas the total number of cells and total number of degrees of freedom increases by a factor of 8. Therefore, we analyze the theoretical bounds of performance of the problem with increase in grid refinement.

Table B.1: Theoretical performance of serial code.

Explicit		Implicit	
Function	Order	Function	Order
CFL ramping	$O(1)$	CFL Ramping	$O(1)$
Local Time Step	$O(n^3)$	Local Time Step	$O(n^3)$
Global Time Step	$O(n^3)$	Global Time Step	$O(n^3)$
Update BCs	$O(n^3) + O(n^2)$	Update BCs	$O(n^3) + O(n^2)$
Exchange Bound Data	$O(n^3)$	Exchange Bound Data	$O(n^3)$
Compute Residuals	$O(n^3) + O(n^2)$	Compute LHS	$O(n^3)$
Check convergence	$O(n^3)$	Compute Residuals	$O(n^3) + O(n^2)$
Solution Update	$O(n^3)$	Check convergence	$O(n^3)$
		Compute RHS	$O(n^3)$
		Scaling	$O(n^3)$
		Preconditioning	$O(n^3)$
		Iterative Solver	$O(n^3 * k)$
		Solution Update	$O(n^3)$
Overall	$O(n^3)$	Overall	$O(n^3 * k)$

Since SENSEI is capable of solving multi-block grids, note that data still must be exchanged along inter-block boundaries and along periodic boundaries although not using MPI (this is a serial implementation analysis). Furthermore, the orientation of different blocks (or

different periodic boundaries on the same block) is not required to be consistent. This adds the required complication of transforming the memory-storage order of the data before exchanging the data to ensure the data orientation is consistent with the receiving boundary's orientation. The compressed sparse row (CSR) matrix storage format is used by SENSEI to store only nonzero entries in the matrix. For sparse matrices such as the Jacobian matrix used for computing the solution update in the implicit formulation, this reduces the storage size to $O(n^3)$, the order of the number of unknowns. It is also noted that the CSR storage format is what allows the implicit formulation to retain $O(n^3)$ performance for preconditioning and the GMRES linear solve to retain $O(n^3 * k)$ performance. Ideally the number of iterations for convergence of the linear solve, k, is minimally affected by the domain size and is much less than n^3.

B.2 Parallel Implementation Theory

This parallel implementation makes use of the fundamental divide-and-conquer model, where the computational domain is decomposed among the available processes such that each process can independently compute the residuals and update the solution for its local sub-domain. Since SENSEI is capable of solving multi-block problems, the decomposition must be applied to all blocks with the available number of processes. The only communication necessary during the solver iteration is to compute the global time-step, exchange inter-block boundary solution data, and compute the global norm of the residuals to evaluate iterative convergence. The primary development cost for this distributed memory parallelism occurs in the domain decomposition. This is advantageous for the overall performance of the solver because the most potential for parallel overhead occurs in the non-iterative portion of the solver. Typically this portion of the solver consumes less than 1% of the execution time since it is only executed once. Therefore, the solution time will be highly insensitive to any inefficiency in this portion of the implementation.

B.2.1 Domain Decomposition

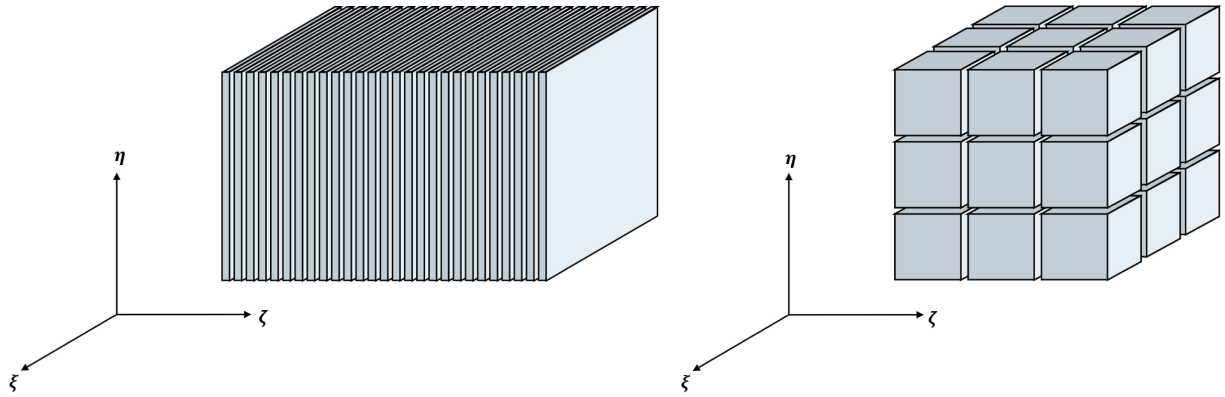
Multiple considerations must be made in the decomposition of the domain. First, we consider the type of decomposition. The domain could be sub-divided along a single dimension or along multiple dimensions. The desirable characteristics that serve as metrics to influence the choice of decomposition type are as follows.

1. Maximize the utility of the decomposition method for any general case and any number of available processes.
2. Minimize the total amount of data transferred between sub-domains.
3. Minimize the difference in decomposed domain sizes to balance the computational load.
4. Minimize the number of communication calls made between sub-domains.
5. Maximize the amount of contiguous memory transferred between sub-domains.

Before discussing these metrics, it is important to discuss the costs of communication. Latency is the time required to communicate data and bandwidth defines the throughput, or rate of data transfer. In latency-bound communication, the latency will remain high even with a high bandwidth because the up-front cost of executing the data transfer, independent of size, will outweigh the high throughput of the data transfer. Conversely, in bandwidth-bound communication the bandwidth for transferring data is low enough that the data transfer cost is primarily attributed to the limitation set by reaching the maximum throughput, or bandwidth.

As a general CFD solver, this first metric is highly important since SENSEI is intended to work for any CFD problem. As a structured grid solver, we may simplify our decomposition analysis by noting that a structured grid may be mapped into a computational domain space that is rectangular (or cubic for 3D problems) with uniform spacing. This allows us to easily visualize the decomposition without loss of generality since the decomposition illustrated in

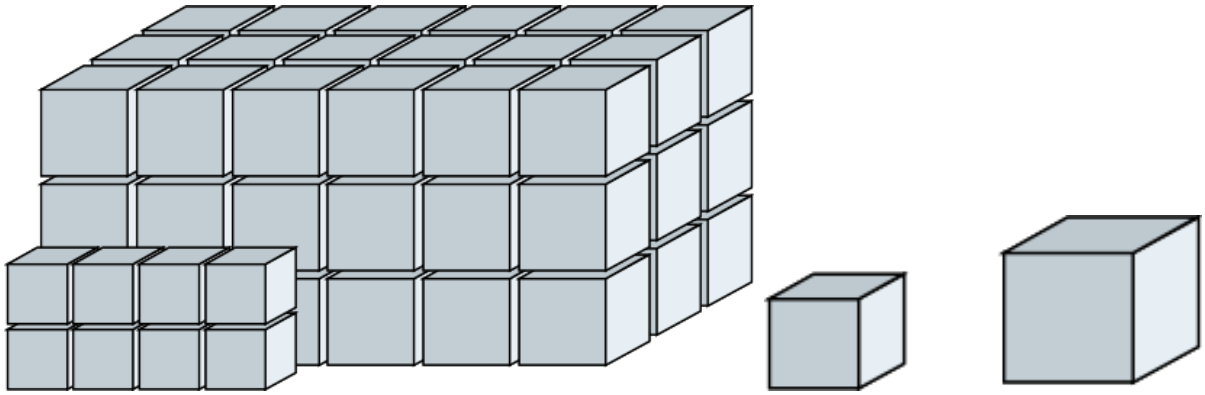
the computational domain could apply with any arbitrary mapping function to any structured, physical domain. Consider Figure B.1, which illustrates a 1D decomposition and a 3D decomposition of the same domain in computation space. This domain has an equal number of cells in each dimension. The same number of subdivisions are used to decompose each domain, yet it is apparent that the number of subdivisions possible in a 1D decomposition are much more limited than for a 3D decomposition. The 1D decomposition is limited by the number of cells in the decomposed dimension of the domain. For example, a $64 \times 64 \times 64$-cell grid used with a second order discretization in SENSEI requires a minimum sub-domain dimension of four cells to prevent adverse interactions between sub-domains. Therefore, the 1D decomposition can only use up to 16 sub-domains, whereas the 3D decomposition can use up to 16^3 $(4,096)$ sub-domains. However, what if the domain is larger in a given dimension and therefore requires more cells? If large enough, the problem becomes highly one-dimensional, and a 1D decomposition might provide the most amount of decompositions possible. For example, if instead of a $64 \times 64 \times 64$-cell grid we used a $1024 \times 16 \times 16$-cell grid, a purely 1D decomposition could have 256 sub-domains, whereas a purely 3D decomposition could only have 4^3 (64) sub-domains. Therefore, the decomposition algorithm must be intelligent enough to determine what mixture of these decompositions to apply.



(a) 1D domain decomposition. (b) 3D domain decomposition.

Figure B.1: Different domain decompositions with the same number of sub-domains in computational space. This illustrates the limited scalability of a 1D decomposition.

This problem relates to and can be solved by addressing the next metric, which is especially important for bandwidth-bound communication. To minimize the total amount of data transferred between sub-domains, we want to minimize the surface area to volume ratio of all sub-domains. Considering that the geometrically minimal surface area solution for any volume is a sphere, we desire to decompose the domain to approximate this shape in the computational domain. Therefore, the minimal surface area solution is a cubic shape in the computational domain. By optimizing this function, we automatically apply more decomposition in the dimensions with more cells. Since this is a discrete problem, we have to determine the discrete decomposition that minimizes the total amount of data transferred for each subdomain.



(a) Blocks can vary drastically in size.

(b) We try to minimize the difference in sub-domain volume.

Figure B.2: Illustration of a hypothetical multi-block grid in computational space.

The next metric is much more consequential for the multi-block case. In a single-block decomposition, we simply divide the total number of grid points as evenly as possible among the sub-domains. However, for the multi-block case, oftentimes the blocks vary drastically in size in the computational domain, as illustrated in Figure B.2. This complicates the decomposition as the appropriate portion of available processes must be assigned to each block so that each block knows how many processes it may use for its decomposition. Again, this would be an elementary operation by simply assigning the number of processes to each block based on its total number of cells relative to the total number of cells in all blocks;

however, this is a discrete problem. This metric usually competes with the previous metric for minimizing the surface area of each subdomain.

Consider the case that the available processes are proportionally distributed to each block, yet the larger block is one process short of a decomposition that would minimize the surface area of each sub-domain. It is quite possible that by giving one more process to the larger block and not minimizing the difference in volumes of the two block's sub-domains, the overall performance of the solver will be increased. Therefore, the implementation must determine a balance between these optimizing functions.

Now consider the fourth metric, to minimize the total number of communication calls made between sub-domains. This metric is very important for latency-bound communication. A 1D decomposition best satisfies this metric, as each sub-domain only communicates with two neighbors. In contrast, a 3D decomposition requires communication with up to 6 neighbors. However, the 1D decomposition cannot guarantee satisfaction of the first two metrics, which are more important for SENSEI to satisfy, so this should be done only in circumstances when the system communication is highly latency-bound.

The final metric provides a similar conclusion to the fourth metric. To maximize the amount of contiguous data transfer, a 1D decomposition in the dimension with the largest stride (distance in memory between consecutive indices) will allow for completely contiguous data transfer between sub-domains. Fortran uses column-major ordering for memory storage, so a decomposition in the third dimension index, corresponding to the z-dimension, would maximize the amount of contiguous data transfer. Again, this conflicts with the first two metrics and should be considered in cases of highly latency-bound communication. In the case of a 3D decomposition, this would also imply that a 2D decomposition in the y and z dimensions would provide better performance than a 2D decomposition in the x and z dimensions or the x and y dimensions.

With all of these considerations, it was determined that a decomposition method based on the first three metrics would provide the best performance and capability desired for SENSEI.

This implementation is accomplished by first proportionally distributing processes among the blocks, then iteratively enforcing as close to a minimum surface area decomposition as possible and reinforcing the proportional distribution of processes until the solver converges to an optimal decomposition or the maximum number of iterations is attained. With each iteration, the remaining processes that are not used because of the attempt to minimize the surface area are distributed among all blocks to determine which block(s) can make the most effective use of the remaining processes. The algorithm is outlined in Figure B.3: This algorithm was found to converge within 2 or 3 iterations for multiple test cases with different domain sizes and numbers of available processes.

B.2.2 Theoretical Analysis

This section provides a basic theoretical analysis for the performance of the parallel implementation of SENSEI for 3D problems. Again, n is on the order of the dimensions of the computational domain. The order of every parallel function within SENSEI is also analyzed accounting for p, the number of processes, t_s, the startup time for setting up parallel communication, and t_w, the per-word parallel communication cost dependent upon the amount of data being transferred. This analysis assumes that all collective calls (broadcasting, scattering, gathering, etc.) are made using a binary tree network topology. This topology dictates that the data is broadcast from one process to two neighbors, which subsequently each broadcast to two other neighbors, communicating the data to all processes in $log(p)$ steps.

Decomposition Parallelism

The steps involved with decomposing the domain and their associated performance are outlined in Table B.2 with the associated order of each function. The domain decomposition requires no parallel communication and the execution time is negligibly affected by the domain grid size, so this operation has a constant order. The order of the block decomposition

```
! Calculate total volume of all blocks

! Determine ideal distribution of nodes among blocks
! (To minimize difference in sub-domain volumes)

! Iterate until converged or reach maximum iteration
for i from 1 to max_iter
  for n from 1 to num_blocks
    ! Enforce as close to minimum sub-domain surface area as possible

    ! Reinforce as close to ideal distribution as possible with
    ! remaining processes available.
  end loop

  ! Calculate total remainder of processes

  if remainder > 0 then
    add remainder to process count for all blocks
  else if remainder == 0 then
    return ! The decomposition is computed
  else if nothing has changed this iteration
    exit loop
  end if

end loop

! Order blocks from largest to smallest

for n from 1 to num_blocks
  ! Now that the decomposition has been optimized, try to reduce the
  ! total remainder as much as possible to make use of all processes.

  if remainder < 0 then
    ! Do whatever it takes to make the remainder positive.
    ! (Makes use of the reordered list of blocks)
  end if
end loop
```

Figure B.3: Algorithm for determining the optimal domain decomposition.

is dictated by multiple broadcast collective calls and blocking send/receive calls to transfer the sub-domain grid geometry. The bound decomposition also makes use of broadcast collective calls.

Table B.2: Theoretical performance of non-iterative portion of the parallel code.

Function	Order
Calculate Decomposition	$O(1)$
Decompose Block	$O(\log p(t_s + t_w)) + O(p(t_s + t_w n^3/p))$
Decompose Bounds	$O(\log p(t_s + t_w))$
Add Interior Bounds	$O(t_s + t_w \delta_1)$
Link Interblock Bounds	$O(\delta_2 \log p(t_s + t_w)) + O(t_s + t_w)$

All parallel communication for adding interior boundaries and linking these interior boundaries between sub-domains is for transfer data sizes of δ. These steps use non-blocking send/receive calls as well as broadcast and gather collective calls. Since δ is much smaller than n, the primary parallel overhead of the decomposition is generated by distributing the physical coordinates of all grid nodes to all other processes from the root process. This function uses blocking communication since all communication is with the root already, so the number of simultaneous communication calls is already limited. Therefore, the parallel execution time and overhead of the non-iterative portion of the code execution is approximately:

$$T_p \approx n^3 + \log p(t_s + t_w) + pt_s + t_w n^2 + n^2/p \tag{B.1}$$

$$T_o \approx p \log p(t_s + t_w) + p^2 t_s + p(t_w + 1)n^2 \tag{B.2}$$

These results lead to an isoefficiency of $O(p^3)$, which is not cost-optimal. This parallel overhead could potentially be reduced by using non-blocking communication; however, since this portion of the solver is executed only once the overhead due to this inefficiency was not found to significantly affect overall execution time.

Iterative Solver Parallelism

With all the work performed to decompose the domain, the only remaining communication for the iterative portion of the solver is to compute the global time step (if used), exchange solution data at inter-block boundaries, and compute the global residual norm to evaluate convergence. These functions and associated order of performance are outlined in Table B.3. The global time stop and computation of a global residual norm both use collective reduction operations to determine the global values. The boundary data interchange uses non-blocking send/receive calls.

Table B.3: Theoretical performance of iterative portion of the parallel code.

Function	Order
Global Time Step	$O(\log p(t_s + t_w))$
Interchange Bound Data	$O(t_s + t_w n^2/\sqrt{p})$
Global Residual Norm	$O(\log p(t_s + t_w))$

The resulting parallel time and overhead are:

$$T_p \approx \log p(t_s + t_w) + t_s + t_w n/\sqrt{p} \tag{B.3}$$

$$T_o \approx p \log p(t_s + t_w) + pt_s + \sqrt{p} t_w n \tag{B.4}$$

Since non-blocking communication is used, the communication that bounds the isoefficiency (assuming a reasonably high bandwidth) is the collective calls for computing the global time step and global residual norm. This leads to an isoefficiency of $O(p \log p)$ which is close enough to linear to be considered cost-optimal. Memory-constrained and time-constrained scalability analyses indicates that the efficiency for these tests is $O(1)$.

Appendix C

Running Parallel SENSEI

Very few modifications are required to run SENSEI in parallel with MPI, OpenACC, or both. The key steps to run SENSEI in parallel are outlined within this section.

C.1 Compiling SENSEI

With the updated use of `cmake` to compile SENSEI, only a couple of flags are necessary to add or modify. The serial `cmake` compile requires execution of the following statements within the build directory:

```
cmake -DCMAKE_BUILD_TYPE=RELEASE -DCMAKE_Fortran_COMPILER=[gfortran, pgfortran]
     [-DMPI=OFF] [-DOPENACC=OFF] <path-to-SENSEI>/SENSEI/

make
```

For the parallel build of SENSEI, an `MPI` option flag and `OpenACC` option flag are added. When using MPI, the compiler should be changed to Open MPI (or some other MPI implementation such as MVAPICH2), that is compiled to use the desired underlying compiler

(e.g. GCC's gfortran or PGI's pgfortran). The network directory for Dr. Roy's research lab contains two compilations for Open MPI 1.10.0: one that uses gfortran and one that uses pgfortran. When using OpenACC, compile with either pgfortran or the Open MPI compiled to run with PGI. The updated compile statements are as follows:

```
cmake -DCMAKE_BUILD_TYPE=RELEASE
      -DCMAKE_Fortran_COMPILER=[gfortran, pgfortran, mpifort]
      -DMPI=[ON,OFF] -DOPENACC=[ON,OFF] <path-to-SENSEI>/SENSEI/

make
```

C.2 Namelist Inputs

No additional inputs are necessary for the input file namelist. However, in the case that the user desires the automatic domain decomposition to give preference to a decomposition that uses more processes than a decomposition that is more optimal in the decomposition metrics, as discussed in Appendix B, the `nonoptimal_decomp` input may be used in the `parallel` namelist section, as shown below. If set to true, a decomposition that uses more processes but is less optimal will be given preference.

```
&PARALLEL
  NONOPTIMAL_DECOMP=[T,F]
/
```

C.3 Running SENSEI

Finally, if running SENSEI with MPI, the `mpirun` executable must be used to start the processes desired for program execution. Details on the use of `mpirun` may be found in the

man page documentation of this executable[1]. A basic example to run SENSEI with 32 processes, mapped and bound to separate sockets, is shown below.

```
mpirun -np 32 --map-by ppr:1:socket --bind-to socket
       <SENSEI-build-dir>/bin/SENSEI
```

References

[1] The Open MPI Project. *mpirun(1) man page (version 1.10.1)*. https://www.open-mpi.org/doc/v1.10/man1/mpirun.1.php.