Title

Multidisciplinary Simulation Acceleration using MultipleShared-Memory Graphical Processing Units

Authors

Kemal, Jonathan Yashar
Davis, Roger L
Owens, John D

Publication Date

2016-04-01

Peer reviewed

Multidisciplinary Simulation Acceleration using Multiple Shared-Memory Graphical Processing Units

Jonathan Y. Kemal[I],

Roger L. Davis[II], and John D. Owens[III]

University of California, Davis, California, 95616

In this paper, we describe the strategies and programming techniques used in porting a multidisciplinary fluid/thermal interaction procedure to graphical processing units (GPUs). We discuss the strategies for selecting which disciplines or routines are chosen for use on GPUs rather than CPUs. In addition, we describe the programming techniques including use of Compute Unified Device Architecture (CUDA), mixed language (Fortran/C/CUDA) usage, Fortran/C memory mapping of arrays, and GPU optimization. We solve all equations using the multi-block, structured grid, finite-volume numerical technique, with the dual time-step scheme used for unsteady simulations. Our numerical solver code targets CUDA-capable Graphical Processing Units (GPUs) produced by NVIDIA. We use NVIDIA Tesla C2050/C2070 GPUs based on the Fermi architecture, and compare our resulting performance against Intel Xeon X5690 CPUs. Individual solver routines converted to CUDA typically run about 10 times faster on a GPU for sufficiently dense computational grids. We used a conjugate cylinder computational grid and ran a turbulent steady flow simulation using 4 increasingly dense computational grids. Our densest computational grid is divided into 13 blocks each containing 1033x1033 grid points, for a total of 13.87 million grid points or 1.07 million grid points per domain block. Comparing the performance of 8 GPUs to that of 8 CPUs, we obtain an overall speedup of about 6.0 when using our densest computational

[I] Graduate student, Mechanical and Aerospace Department.
[II] Professor, Mechanical and Aerospace Engineering Department.
[III] Professor, Electrical and Computer Engineering.

grid. This amounts to an 8-GPU simulation running about 39.5 times faster than running than a single-CPU simulation.

Nomenclature

c_p = specific heat at constant pressure

dt = time step

h = flow enthalpy

I = flow rothalpy

k = turbulent kinetic energy or thermal conductivity

Pr = Prandtl number

R = radius

t = time

T = temperature

u = flow velocity component

x = Cartesian coordinate components

ω = turbulent dissipation frequency

Ω = rotation vector

ρ = density

μ = coefficient of viscosity

τ = shear stress

subscripts

i = streamwise grid index

j = tangential grid index

s = solid

I. Introduction

THE design and analysis of jet engine components has matured to the point where multidisciplinary numerical simulations are necessary for performance, durability, and reliability improvements. Additionally, simulations that analyze and predict performance with sufficient accuracy to distinguish between component configurations require high computing capability. To accomplish this, the development of a high-fidelity multidisciplinary fluid/thermal interaction analysis tool, known as MBFLO[1], has been an ongoing project. MBFLO is a multi-block solver that makes use of MPI in order to take advantage of multiple processors (or CPU cores) as well as multiple network nodes. The code has been developed to solve multidisciplinary systems of partial differential equations on both two and three dimensional geometries, and primarily employs the finite volume computational scheme[2].

In order to help satisfy the immense computational requirements of large scale CFD simulations on dense computational grids, we have modified the two-dimensional solver to be able to use CUDA enabled Graphical Processing Units (GPUs). Many applications in science and engineering[3] have now been demonstrated using fully programmable GPUs. This functionality includes the ability to solve systems of both ordinary and partial differential equations for a wide variety of engineering applications with most, if not all, focused on a single discipline. Here, we address issues, strategies, and programming techniques that deal with multiple disciplines.

NVIDIA produces graphics cards specifically for general purpose computing, such as the Tesla series processors we use, which are programmable using an API set called CUDA. GPU architecture is intrinsically parallel in nature, as the card is composed of many arithmetic logic units (ALUs), which are essentially tiny processors. The Tesla C2050/C2070 cards we use, for example each contain 448 ALUs. Thus the device is designed to maximize overall throughput, as opposed to minimizing the latency between individual operations. This makes the GPUs ideal for parallelizable problems, including the numerical integration of systems of differential equations. Our work involves solving the compressible fluid dynamics equations in conjunction with the thermal conduction equation across a computational

domain including both “fluid” and “solid” regions, which is necessary for optimizing turbomachinery design. As both types of equations can be solved with similar numerical techniques, GPUs are an ideal tool for this multidisciplinary work. We can therefore use some GPUs to perform the thermal calculations while other GPUs are simultaneously focused on the flow solver. We can even consider using CPUs for some parts of the multidisciplinary simulation that are not compute or data intensive and GPUs for those parts that are. Techniques can even be considered for automated optimization in the use of CPUs and GPUs to minimize overall computational time[4].

Research has shown that GPUs can reliably produce an order of magnitude speedup, when measured against comparable high performance CPUs, for a wide range of scientific applications[5-15]. Bolz et al. showed that NVIDIA's GeForce FX cards were well suited for a conjugate gradient solver as well as a multi-grid solver, which was promising in the context of fluid solvers[5]. Brandvik et al. demonstrated that both 2D and 3D Euler solvers were accelerated by GPUs, implementing both NVIDIA and ATI GPUs. Using a combination of BrookGPU and CUDA, they achieved speedups of 29 for the 2D case and 16 for the 3D solver[6]. Goodnight et al. also showed that GPUs were well-suited for multi-grid applications, exploring broader applications of solving complex boundary value problems[7.] Hagen et al. explored the application of shallow-water wave simulations, using cases that involved dry-bed zones and non-trivial topographies. Using the Lax-Friedrichs and 2nd-order central-upwind numerical schemes to solve their hyperbolic PDEs, they were able to achieve speedups as high as 30 for sufficiently dense (uniform) computational grids[8]. Using the NVIDIA GeForce FX Ultra GPUs, and the Cg shading language, Harris et al. were able to show that GPUs are well suited for cloud dynamic models. These PDEs related to cloud dynamics describe thermodynamics, fluid motion, water phase transitions, and other physical behaviors[9]. Kruger and Westermann described how various GPUs are well suited for solving the 2D wave equation and the incompressible Navier-Stokes equations[10]. Li et al. used the Lattice Boltzmann method for their flow simulations with complex boundaries. They also used OpenGL and the Cg shading language, with NVIDIA GPUs, and were able to attain speedups of about a factor of 9[11]. Scheidegger et al. were able to show that NVIDIA GPUs based on the NV35 and NV40 architectures were well suited for CFD

simulations using the SMAC (Simplified Marker and Cell) method. With sufficiently dense grids, they were able to achieve speedup factors as high as 21[12]. Similarly, Hagen et al. achieved speedups as high as 20 using NVIDIA GeForce GPUs. They implemented a finite-volume scheme for their Euler solver[13]. Goddeke et al. determined that a heterogeneous computing cluster using GPUs and CPUs scales well with the number of nodes in the cluster, and ultimately provides a cost/performance advantage over clusters that only rely on CPUs[15]. Overall, in all of the above applications with the variety of numerical methods used, the researchers were able to attain substantial speedups over CPU-based simulations – usually of at least a factor of 10.

This paper presents the process of modifying MBFLO routines to run on GPUs, as well as optimization processes and limitations. The code for dividing the computational grid into contiguous blocks, such that the domain problem can be solved in parallel on multiple processors, has already been written. This includes the implementation of MPI to handle message passing of boundary information between processes. However, modifying the code to use GPU presents additional parallelization challenges. Since every GPU itself consists of many processors performing computations in parallel, we could not merely have the GPUs and CPUs run essentially identical solver code. In order to avoid "race conditions" and other issues arising from parallelism, it became necessary to modify some of our algorithms. These modifications are discussed in more detail in Section VI.

Converting previous versions of MBFLO flow routines to run on a GPU has been explored[16]. The encouraging results lead us to believe that converting the latest multidisciplinary version of MBFLO, which is more computationally intensive, would result in substantial speedups. This also involved converting the thermal conjugate solver to run on GPUs. In this case, the fluid and thermal solvers run on different portions of the domain. This provides the opportunity to have certain MPI processes responsible for only the fluid or thermal portions of the domain, thus increasing the advantage of parallelism. However, as the fluid solver is much more involved, the thermal solver runs significantly faster for an equivalent number of grid points. As the solver requires all code to complete before the next iteration can begin, a load balancing analysis specific to each computational grid is necessary to ensure all MPI

processes complete iterations in approximately the same amount of time. Various other improvements were made to the GPU-enabled version of MBFLO, including memory management enhancements and re-converting updated versions of the primary flow solver routines. The optimizations and memory management will be discussed in detail in Sections IV and V.

II. Numerical Solver Approach

With advances in computational performance, parallelization, automation, and numerical techniques a direct-coupled approach to multidisciplinary design and analysis is now possible. Our approach will be based on the general solution procedure called MBFLO, in which the coupled fluid and thermal fields are computed simultaneously. This procedure exists for axisymmetric, two-dimensional, and three-dimensional configurations. The scope of this work, however, is limited to the two-dimensional solver. All equations are solved using the multi-block, finite-volume numerical scheme. The governing system of partial differential equations for the flow, thermal, and structural fields is described below.

A. Fluid Dynamics Equations

The unsteady, Favre-averaged governing flow-field equations for an ideal compressible gas in the right-handed Cartesian coordinate system, using relative-frame primary variables, are written as:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \left(\rho u_i\right)}{\partial x_i} = 0 \tag{1}$$

$$\frac{\partial \left(\rho u_i\right)}{\partial t} + \frac{\partial \left(\rho u_j u_i\right)}{\partial x_j} + \frac{\partial p}{\partial x_i} = \frac{\partial \tau_{ji}}{\partial x_j} - \overline{Sm}_i \tag{2}$$

$$\frac{\partial E}{\partial t} + \frac{\partial \left(\rho u_j I\right)}{\partial x_j} = \frac{\partial}{\partial x_j}\left[u_i \tau_{ij} + \left(\frac{\mu}{\mathrm{Pr}} + \frac{\mu_t}{\mathrm{Pr}_t}\right)\frac{\partial h}{\partial x_j}\right] \tag{3}$$

The above equations refer to conservation of mass, momentum, and energy, respectively. The body-force vector in the momentum equation, $\overline{Sm_i}$, represents body forces per unit volume such as those due to rotation (Coriolis and centripetal forces).

Two additional governing equations are solved for the transport of turbulent kinetic energy and the turbulence dissipation rate in order to obtain the turbulent viscosity, μ_t, according to the Wilcox[17] turbulence model.

B. Thermal Dynamics Equations

The physics governing the flow of heat through a solid body is given bythe transient heat conduction equation, which represents conservation of energy, as given in Eq. (4)where T_s is the temperature of the solid, ρ_s is the solid density, c_{ps} is the solid specific heat, and k_s is the solid thermal conductivity.

$$\rho_s \cdot c_{p_s} \frac{\partial T_s}{\partial t} = k_s \cdot \left[\frac{\partial^2 T_s}{\partial x_i^{\,2}} \right] \tag{4}$$

The flow field conservation equations given in Eqs. (1)-(3) and the turbulence equations are solved using a Lax-Wendroff control-volume time-marching scheme as developed by Ni[18], Dannenhoffer[19], and Davis[2]. Numerical solutions of unsteady flows are obtained by implementing a dual time-step procedure[20]. These techniques are second-order accurate in both time and space. The thermal energy conservation equation for the solid is also solved using a control-volume, time-marching scheme that is very similar to what is used for the flow-field equations[21]. By ensuring conservation of energy between the fluid and solid, the solution to Eq. (4) for the solid can be directly related to that of Eq. (3) for the fluid. This requires enforcing a unified heat flux at the fluid/solid boundary as well as ensuring the temperature at this boundary is the same for both the fluid and solid media.

A multi-block, structured-grid strategy is used throughout the domain for both fluid and solids. Point-matched blocks are used as much as possible and overset grids are used where flexibility in geometry creation and placement is required, such as for film-cooling holes or leakage slots. Blocks are constructed from multiple faces. Each face of a block is allowed to have an arbitrary number of sub-faces to enable virtually any connectivity of blocks and any combination of boundary conditions along a face of a block. Block connectivity and boundary condition information is defined by a global connectivity file which is

automatically produced during the grid generation process. With the computational domain split into blocks, along with a complete connectivity file, blocks are ready to be distributed among multiple processors for parallel computation.

III. Parallelization Approach

In this section, we will discuss an overview of the parallelization and programming strategies, memory management on the GPU, GPU kernel launch schemes, and optimization of GPU code.

A. Parallelization Strategy Overview

Our two-dimensional multidisciplinary solver has previously been programmed to run a multi-block simulation on many CPU processes. As long as the number of processors requested does not exceed the number of domain blocks, the code by default splits up the blocks evenly (or as evenly as possible) among the available CPUs (or cores). The open-source Message Passing Interface (Open MPI) is used to launch a number of processes, or threads, across the available compute nodes. Each MPI process/thread runs the solver routines on its own CPU/core, and uses MPI libraries to communicate necessary block information (such as boundary/ghost-node information) to other MPI processes. Such communication occurs at multiple points throughout the solver's main iteration loop. MPI determines, behind the scenes, whether information needs to be sent to another process on the same computer, or to a different network compute node.

Our strategy for using Graphical Processing Units (GPUs) is for the user to decide whether the solver is to use only CPUs, only GPUs, or a combination of the two for primary solver computations. That is, with an appropriate flag set, the user will decide whether an MPI process will launch the primary solver routines on a GPU or a CPU. Each MPI process primarily uses one computing resource – one CPU or GPU – for the bulk of the solver computation. However, when a GPU is selected, some operations are still performed on a CPU core. Figure 1 illustrates the MBFLO procedure and distinguishes between operations performed on the GPU and on the CPU. Most of these operations are related to setup or post processing operations, such as setting up computational grid arrays or writing out solution files. These

operations only happen once, not throughout the iteration loop. However, other tasks including various boundary condition operations, message passing operations, and multi-grid routines do occur within the iteration loop and are handled by a CPU core. These operations are generally low-volume, meaning they do not involve operating on every grid point contained in a domain block. Since GPUs require a high computational load to reach optimal performance, low volume operations do not provide much (if any) performance advantage. The user must keep in mind that even MPI processes selected for GPU implementation do use some CPU resources. Thus, when a GPU is selected for use, a corresponding CPU core is also isolated for that MPI process. For example, running an 8-block simulation across 8 GPUs will also make limited use of 8 CPU cores.

The names of the subroutines responsible for primary solver computations are included in Fig. 1. If the user elects to use a GPU for primary computations, the routines highlighted in yellow are performed on the GPU. These routines are the most computationally intensive and time consuming portions of the iteration loop. Although all primary computations highlighted in Fig. 1 are executed on the GPU in this investigation, further optimization of the combined use of CPUs and GPUs could be performed on a per-routine basis in the future. After the primary thermal and flow solver routines are completed, the low volume multi-grid routines are called. As discussed below, these operations are performed quickly relative to computationally intense flow solver routines, and are not handled by a GPU. Inter-block communication (using MPI for message passing), boundary condition applications, and multi-grid related operations are left to the CPU associated with that particular MPI process. When running a serial simulation, one process is responsible for the entire simulation. In this case, the thermal equations are solved in the beginning of the main iteration loop (the “conjugatep” routine). As described below, this routine only operates on solid domain blocks, so if the simulation only has fluid blocks, then the thermal solver is skipped. When running a simulation in parallel, any MPI process assigned both solid and fluid blocks will also run the thermal solver (on just the solid blocks) at the beginning of the iteration loop. However, it is also possible (and possibly preferable) to assign only solid domain blocks to a particular

MPI process. In this case, the iteration loop for that process consists of only the thermal solver, updating the temperature variables, and inter-block MPI communication.

In most modern compute cluster setups, each CPU has multiple cores that are treated as multiple processors and, in our code, are associated with multiple MPI threads. Each cluster compute node, however, can have multiple multi-core CPUs installed. Two of our compute nodes, for example, each have 2 processors that each contains 6 CPU cores, for a combined total of 12 available CPU cores per node. However, since our Intel Xeon CPUs also support hyper-threading, this appears to the operating system as a total of 24 available CPU cores. Furthermore, each compute node has its own RAM. That is, within each compute node, all processes launched share the system's main memory (RAM). However, as each compute node has its own RAM, a simulation can also be launched with multiple compute nodes such that every process is on a different compute node and thus has a need for un-shared memory. Thus, our cluster design includes a combination of distributed and shared memory among CPU processes. A GPU can be viewed as an additional processor that sits adjacent to the CPU with its own memory system. Each GPU has its own RAM and thus the GPU's operations can be viewed as a distributed memory system. However, when a GPU is elected for use, the solver code must transfer data from the host (a CPU process accessing main RAM) to the GPU's private RAM. While some memory can be freed on the host side after transfer to the GPU is complete, this does imply that a certain amount of host memory must remain allocated (used) throughout the solver's operation even in cases where a GPU is being used as the primary computational resource. Thus, the user must remain aware that MPI processes associated with GPUs do use some of the host's memory resources. The user must also of course keep in mind the total available RAM of every compute node, and keep track of total memory allocations to ensure that resources are not exceeded. While operating systems can make use of hard disk swap space for allocations in excess of available system RAM, this can dramatically slow down simulations and should be avoided. If a user attempts to allocate more memory on a GPU than its private RAM can accommodate, the code will not execute.

B. General Programming Strategy

There are multiple programming methods available that allow a wide variety of Graphical Processing Units to be used for general-purpose computation. NVIDIA has created a general-purpose parallel computing platform and programming model known as Compute Unified Device Architecture (CUDA). CUDA only works on some NVIDIA GPUs, such as the Tesla series devices used in our computing cluster, and provides a straightforward Application Programming Interface (API). CUDA can be used with either C/C++ or Fortran. However, CUDA-Fortran was developed by "The Portland Group" and requires dependence on their costly compilers. NVIDIA's CUDA-enabled C++ compiler, on the other hand, is free of charge. Thus, it was decided for our purposes to write all of the GPU invoking code in C++.

The MBFLO2 two-dimensional multidisciplinary flow solver, has previously been written exclusively in Fortran 90. Since all of the GPU code is written in C++, mixed language programming and compiling is necessary. While compiling and linking dual-language code is fairly straightforward, some issues do arise that the programmer must be aware of. When Fortran calls and passes variables to a routine, or a C function, it always passes by reference. That is, information is never passed by value, even when the variable passed only contains a single number. Thus, a C routine being called by Fortran must use pointers to receive all variable information. Throughout the solver, variables containing a memory address on the GPU (corresponding to where a data array is located) must be passed back and forth between Fortran and C routines. This is accomplished with the use of 8-bit integer variables stored in a Fortran module. That is, when it is necessary to pass a GPU address to a C function, the 8-bit integer is passed (by reference) to the C function. The C function receives the information as a double pointer, or pointer-to-pointer, of the appropriate data type (typically double precision floats). The value of this variable, the address, can then be passed to CUDA API calls as necessary.

The most notable issue that arises from this mixed-language programming is that multi-dimensional data arrays created in Fortran are stored in memory in column-major order. For example, if a three-dimensional array x(i,j,n) were created in Fortran, the data in the first ('i') column would be ordered

sequentially in system memory. That is, the data stored at x(2,1,1) would be directly adjacent in memory to the data stored at x(1,1,1). C, on the other hand, is designed to store multi-dimensional arrays in row-major order. Thus, using C's syntax, the data at x[1][1][1] would be adjacent to the data stored at x[1][1][0], not x[0][1][1]. When a data array is passed from Fortran to C/C++ code, the multi-dimensional structure of the array is not automatically retained. Nor is the total allocated size of the array. The C function is simply passed an address in memory, and it is up to the programmer to handle things from there. One option is to treat the data as a 1-dimensional array and carefully use the correct strides, within CUDA kernels, to ensure that the appropriate data can be accessed. Another approach, which we have implemented, is to use pointers allocated on the GPU to recreate (map) the multidimensional array structure. For example, the data obtained from x(i,j,n) could then be accessed in C as x[n][j][i]. The indices are reversed because of the column-major storage in Fortran; the "i" array elements are next to each other in linear memory. This plays an important role with coalescing memory accesses to the GPU, which is discussed in section IV. Another method that can be used to deal with this discrepancy is to transpose multi-dimensional arrays in Fortran immediately before passing the array to C code. In this case, x(i,j,n) in the original Fortran array would be accessed as x[i][j][n] in C. However, we decided against this added convenience simply to avoid the extra computations. Lastly, it is possible to allocate arrays in Fortran such that data is stored at negative indices. For example, x(-3,-2,-1) can be legal in Fortran. However, no such flexibility exists in C. A data array in C, multidimensional or not, always begins at the 0^{th} index. It is necessary to be keenly aware of how the two languages store array information.

C. GPU Memory and Kernel Launch Schemes.

In order to perform solver calculations on the GPU, relevant data arrays must be allocated on the device's RAM (or global memory). As our strategy is to only invoke the GPUs for procedures that occur within the solver's iteration loop, most arrays that the GPUs use have some initial values that have already been calculated and set by the host (CPU). Thus, it is necessary to copy this information over to

the device. These actions are accomplished by calling a C routine from Fortran which then uses CUDA API calls to allocate arrays or to copy information. Memory copies between the host and device are expensive (time consuming), and thus the appropriate strategy is to minimize them as much as possible during the solvers iteration loop. CUDA is also capable of copying memory between the GPU and host asynchronously with respect to other ongoing operations, which will be discussed in more detail in the GPU Optimizations Section.

The number of threads that we launch to a GPU kernel is directly related to the number of points that make up our computational grid. A strategy that is often employed is to fix the number of threads launched to a kernel and write code such that each thread does enough operations to modify data associated with multiple grid points. This can be referred to as "persistent-threads" programming. However, our strategy is that each thread modifies data associated with a single grid point and thus the total number of threads is approximately the total number of points. This strategy allows our kernels, which are complex due to the nature of the flow solver algorithms, to be somewhat simpler and thus easier to read and modify.

CUDA allows threads to be launched in a multi-dimensional structure, corresponding to multi-dimensional computational grids. Thus, a thread can access an array in X[j][i] syntax simply by figuring out the 'j' and 'i' values from the automatically generated thread ID variables. Specifically, this information can be obtained using the following code.

$$\text{int } i = blockDim.x * blockIdx.x + threadIdx.x \tag{5}$$

$$\text{int } j = blockDim.y * blockIdx.y + threadIdx.y \tag{6}$$

CUDA launches threads in groups of thread "blocks", not to be confused with the domain blocks that we have already divided our total computational grid into. Thread blocks, for our purposes, are two-dimensional. Each block can consist of a maximum of 1024 threads. However, for reasons that are described in the GPU Optimizations section, we use block sizes of 128 threads. Typically, we use thread blocks that are 4 x 32 threads. This also corresponds to 4 "warps," where each warp is defined as 32

threads, with 4 warps being stacked atop each other. The programmer also must specify the total number of blocks, in each direction, to launch. Every block launched to a particular kernel must have the same thread structure.

Given our constant block size of 128 threads, the number of blocks launched to the kernel depends on the dimensions of our two-dimensional computational grid. Since the total number of grid points is not likely to be exactly divisible by 128, a small fraction of threads will likely not reside within the computational grid. These threads are called “empty threads” and are, using IF logic, assigned to do nothing. While this is not ideal, it is realistically not entirely avoidable, and it is up to the programmer to design computational grids to minimize the number of empty threads being launched.

MBFLO2 is a multi-block solver, meaning the computational grid is previously broken up into segments (blocks) by other codes. The manner in which the domain is broken up and distributed among processors can significantly affect the computational speedup that can be achieved compared to solving the entire domain on a single processor. Chiefly, since the current iteration must complete on all blocks before any processes can continue, it is crucial that each processor have approximately the same computational load. This is referred to as load balancing. If all processors have identical hardware resources and run the same routines, distributing blocks equally among processors should balance the computational load well. MBFLO2, however, can run different (thermal) computations on solid domain blocks that generally complete (during a given iteration) much faster than all of the fluid routines.

Additionally, since GPUs can generally complete computations much faster than CPUs, ideal load balancing becomes trickier still when running a single simulation on both CPUs and GPUs. Due to these complications, it is generally the case that a single process can be assigned an arbitrary number of domain blocks that each has arbitrary dimensions. Since each block can have arbitrary dimensions, the number of CUDA threads necessary to cover the entire block is also arbitrary. Thus, our strategy for handling multiple domain blocks assigned to a single GPU is to launch every kernel once per block. Of course, each domain block is then broken up into thread-blocks by CUDA, as described above. As many thousands of threads are necessary to maximize the efficiency of a CUDA kernel, the important

information is not how many grid points are assigned to a GPU, but how many grid points are assigned to each domain block. While this does not matter for sufficiently dense computationally grids, it is generally the case a lower computational speedup will be achieved on smaller blocks, which can also further complicate the process of achieving ideal load balancing.

IV. GPU Optimizations

As described in the previous section, the key (most computationally intensive) MBFLO routines have been converted to run on CUDA-enabled GPUs. However, merely getting the GPU to run the code (and produce the correct result) is not sufficient to ensure that GPU performance is maximized. If CUDA code is not optimized properly, the achieved speedup can suffer tremendously. In some cases, non-optimized GPU code might actually run slower than the CPU equivalent. As discussed in the previous section, GPUs require high computational loads in order for performance to be maximized. However, beyond making sure that the GPU is assigned enough work to do, there are also various technical optimizations necessary to ensure that the GPU performance is optimal. These techniques are described in this section.

A. GPU Memory Coalescing

Once a CUDA kernel is launched, each thread needs to obtain some data from the device's global memory (RAM). Threads then perform some calculations and ultimately return results to global memory, such that they can be accessed by other kernels or ultimately sent back to the host. Understanding and taking advantage of how the GPU performs global memory reads and stores is a key component to achieving computational speedups. CUDA capable GPUs organize groups of 32 threads into "warps". Fermi architecture GPUs will schedule global memory read and store operations for a warp of threads at a time. This is due to the fact that groups of 32 threads are actually running simultaneously, which is discussed in more detail in the Memory Hierarchy section. Older CUDA capable GPUs schedule such operations on a half-warp basis.

When a CUDA thread requests data from global memory, the GPU accesses a sequential segment of DRAM that is 32, 64, or 128 bytes long[IV]. This occurs even if only a small portion of the information will end up being used by the CUDA kernel. However, if multiple threads of a warp request data that is in the same DRAM segment, only one operation is required to transfer it. Thus, it is ideal for each thread in a warp to access memory in the same 128-byte segment. If each thread in a kernel accesses some element of array X, it is ideal that the threads in every scheduled warp access sequential elements in the array. For example, if the first warp accessed X[0] through X[31], all of the data would be obtained by accessing a single 128-byte DRAM segment. This is referred to as memory coalescing.

If the threads that make up a particular warp access multiple segments of RAM for a single operation, more memory bandwidth is used by each warp and thus it takes longer for the GPU to complete every thread launched to the kernel. Fermi architecture devices access 32, 64, or 128-byte regions of RAM to help combat the effects of non-coalesced memory operations. Older GPUs do not have such flexibility and fetch 128-byte regions only. In order to ensure coalesced memory transactions with multi-dimensional arrays, programmers must be aware of how arrays are stored in memory. Our arrays are all originally created using Fortran 90, which stores multi-dimensional arrays in column-major order. That is, an array originally created in X(i,j) format in Fortran will be accessed as X[j][i] in C++. This means that each warp of threads would ideally access 32 array elements with the same 'j' index and consecutive 'i' indices.

In order to ensure memory coalescing, it is also crucial to be aware of how warps are scheduled with a multi-dimensional CUDA kernel launch scheme. As described in the preceding section, array indices can be determined from built-in thread identification variables. If a kernel is launched on a two-dimensional thread grid, warps are scheduled in the "x" direction first. That is, if the "i" and "j" indices are declared as in the sample above, the first scheduled warp would correspond to threads with j=0 and "i" values ranging from 0 to 31. Thus, if the kernel accesses an array as X[j][i] the GPU will end up fetching data that is coalesced in memory.

[IV] http://developer.nvidia.com/cuda.

Memory coalescing, or a lack there of, has a dramatic effect on the computational speedup of a CUDA kernel. To demonstrate this, the CPU execution time of a fairly simple MBFLO2 routine called “lamvis” is compared with the GPU execution times for a coalesced and non-coalesced launch scheme in Fig. 2. The routine calculates the laminar coefficient of viscosity using Sutherland’s formula. In each case, the time is measured over many executions using four (increasingly dense) computational grids. For the GPU cases, the time is obtained using NVIDIA’s CUDA profiler, while MPI timing routines perform the measurement for the CPU case.

As shown in Fig. 2, computational speedups for this routine are significantly lowered when global memory accesses are not well coalesced. The coalesced kernel is launched with thread-block dimensions of 32x4, giving a total of 128 threads (4 warps). As described in the previous section, this results in every thread in a given warp requesting array data that is aligned. The non-coalesced kernels launch on a thread-block configuration of 1x128 threads. Thus, for an array of dimensions X[j][i], each warp is requesting memory from as many ‘j’ locations as possible, but all at the same ‘i’ location, which results in a completely non-coalesced memory operation. The plot also shows, as expected, that speedup generally increases with the number of computational grid points, or number of CUDA threads per kernel launch.

B. Shared Memory Reduction Kernels

At multiple points during all MBFLO2 iterations, it becomes necessary to search a data array for the maximum or minimum value. Primarily this is necessary in order to check for convergence, where the maximum value from an update array is compared with a pre-set convergence parameter. Furthermore, the time step calculations also require finding the minimum value of the time-step ("dt") array. On a CPU, this calculation is trivial and performed simply by looking through the array and checking each element. If the element is the new maximum/minimum, a variable storing the result is updated. However, performing such operations on a GPU isn't so trivial. If, for example, every CUDA thread in a kernel checked one element, multiple threads might try to update the same result variable. This would result in a "race condition" (described in section D) and the result would not be reliable.

Fortunately, by taking advantage of shared memory, a "reduction tree" algorithm can be employed for these tasks that actually works significantly faster than the CPU code. In addition to finding the maximum or minimum value, this technique can be used for other purposes, such as finding the sum or product of an array. The general concept is that each CUDA thread can perform an operation and load the result into a shared memory block. Then, since all allocated shared memory is accessible by every thread in a thread-block, each thread can perform the remaining necessary operations by comparing its elements with the results of other threads' calculations. Since the number of threads per thread-block is user-defined, the number of remaining operations (after the initial load) can be hard-coded into the kernel with "IF" logic. Since the initial loads into shared memory are the result of a desired operation, it is not necessary to launch one thread per array element. In fact, since each thread can perform an arbitrary number of initial loads, the programmer can decide how many threads to launch. A presentation released by Harris at NVIDIA[V], which goes into more technical detail, was followed precisely for our codes. Since the arrays being reduced for this effort are not typically large enough to potentially warrant otherwise, our kernels only perform one initial load per thread. Thus, the number of threads launched is about half of the number of elements in the target array. For our purposes, all arrays are treated as one-dimensional linear blocks of memory (which is what they really are), since the goal is merely to find a maximum or minimum value.

If a reduction kernel were not used, the alternative would be to copy the entire array back to the host DRAM and allow the CPU to search for the desired value. The associated memory transfer, which would occur during all iterations, would be very time consuming relative to the reduction kernel. Figure 3 compares computational speedups using a reduction kernel as opposed to transferring the data back to the host over a variety of computational grids, varying from roughly 10,000 to 250,000 grid points. The GPU solver being tested is a simplistic heat conduction solver that is nearly identical to the conjugate solver routine in MBFLO2. The plot shows, as expected, that in either case, computational speedup increases with increasing grid density. More importantly, the figure shows that using the reduction kernel makes the

[V] http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

overall solver roughly two times faster than a solver that copies the data arrays back to the host. For the densest grid tested, this implies an overall speedup of about 14 compared to an overall speedup of 7.

C. Block Boundaries and Memory Copies

As previously described, MBFLO2 is a multi-block solver, meaning the computational domain is divided into blocks that are distributed such that an individual processor can be responsible for running the numerical solvers on a small portion of the domain. The fluid and thermal equations are generally solved using the finite volume method. This method involves, for a particular grid point, using information associated with neighboring grid points to estimate partial derivatives and other information. Near block boundaries, the solver code will require information from grid points that are not allocated to the domain block it is currently being worked on. That is, the points needed are just on the other side of the block boundary. This means that, except when a boundary is physical as opposed to inter-block, boundary information must be sent to processes that require them. Since certain flow solver routines depend on the results generated by previous routines, boundary information for various variables must be sent at multiple times during the iteration loop.

In the CPU code, previously written block boundary routines are responsible for copying relevant portions of data arrays to buffers and, in conjunction with information obtained from block connectivity files, using embedded MPI API calls to send the data to the appropriate processors. The routines also receive buffer data from other processors and place them in the appropriate arrays. However, if a process is using a GPU for computation, the relevant data must be transferred back to the host's data arrays before these routines can work. The simplest way to do this is to copy back entire data arrays. However, this results in unneeded data being passed (not on the boundaries) and is very time consuming. Thus, routines were written to copy only the appropriate boundary information for a given array back to a buffer on the host, and then copy the information to the appropriate data locations. The ordinary block boundary message passing MPI routines are then called and work properly. Figure 4 shows the speedup obtained by

copying just the appropriate boundary information compared to an entire data array, for 4 increasingly dense computational grids.

As Fig. 4 shows, the advantage of this technique increases with increasing grid density. This is due to the fact that, with a two-dimensional grid, data volume (i.e. the number of overall grid points or array elements) increases quadratically with the grid dimensions. For example, a 1000x1000 grid has 1,000,000 points, which is 100 times as many points as a 100x100 grid. However, when only copying the four boundaries, a 1000x1000 grid requires copying approximately ~ 4000 points, which is only 10 times as many as the 400 required by the 100x100 computational grid. Since copying just boundary data involves using buffers, the speedup isn't quite as high as one would expect merely comparing the number of points being copied. Nonetheless, for our 1033x1033 computational grid, copying boundaries is about 28 times faster than copying a full data array.

D. Asynchronous Memory Copies

Copying memory between the host and device (GPU) is very time consuming and thus should be avoided as much as possible. Fortunately, CUDA enabled GPUs are capable of operations that are asynchronous with respect to the main CPU code. By default, a GPU kernel is asynchronous, or non-blocking, meaning the GPU will perform the kernel operations while the CPU code continues further operations. However, if a second GPU kernel is launched, the code by default waits (or blocks) until the first kernel's execution is completed. This is also usually true of memory copy requests, which wait until the previous kernel has completed before commencing. Memory copies are also themselves, by default, blocking, meaning the code will not continue with other operations until the memory copy has completed.

However, using CUDA streams, kernel launches as well as memory copies can be launched asynchronously with respect to both the host and the GPU. It is possible for two (or more) "streams" of events to occur simultaneously. If two kernels are launched on different streams, the GPU will attempt both of them simultaneously as long as resources are available to do so. Furthermore, and more importantly for our purposes, streams make it possible to have memory copies between the GPU and host

occur asynchronously with respect to both the GPU and CPU code. This means that the main code can continue performing operations on the CPU while results from the GPU are being transferred. Furthermore, it is also possible to have a kernel running while memory copies are occurring. It is up to the programmer to make sure that no memory conflicts arise from the asynchronous behavior, such as the host code referencing an array element that has not yet been copied back to the host. To aid with these concerns, CUDA streams have the functionality to block, or wait until the stream has completed, at the programmer's request.

In order for asynchronous memory copies to occur, the relevant locations of CPU memory must be page-locked, which NVIDIA/CUDA refers to as "pinned memory." This means that the operating system ensures that a block of memory (for an array) is allocated directly on DRAM, and not in any swap disk space. By default, when an array is allocated (in either C or Fortran), it is not page-locked. This means the data may or may not be stored physically in RAM. On the GPU side, DRAM is the only resource where data can be stored after a kernel execution is complete, thus memory on a GPU is always pinned. The device is capable of transferring memory from the GPU's DRAM directly to the host's DRAM. Therefore, if memory on the host is not pinned, copying memory may involve first using the host's DRAM as a buffer, before finally moving the data to the appropriate location. This is why host memory must be pinned in order for asynchronous memory copies to work. Furthermore, this implies that merely pinning relevant host memory generally provides some speedup, whether or not memory copies are asynchronous. There are multiple ways to have an operating system request that an array be page-locked, including the "mlock" system command on Linux systems. However, asynchronous memory copies will not work unless the memory is pinned using the appropriate CUDA API call. Otherwise, the CUDA driver will be unaware that the memory is pinned and behave as though it is not. Due to the fact that relevant data arrays are originally allocated in Fortran, and then later need to be pinned, the appropriate API call to use is “cudaHostRegister.”

D. Atomic Operations and Race Conditions

One issue that arises when dealing with parallel algorithms is how to deal with "race conditions." A race condition, in the context of GPUs, refers to multiple threads trying to access the same data in memory simultaneously. The problem is easily illustrated by the task of summing the values in an array. A simple way to do this would be for each CUDA thread to add the value of its corresponding array element to a variable storing the sum. Each thread must read the sum variable, add some value to it, and write the new value back in the sum variable. However, if a thread writes its result to the variable in between the time another thread reads and writes to that memory, its result will be lost. The effect will be that the sum will have an incorrect answer; some elements will not have been added. This behavior is inherently unpredictable and there is no way of knowing how many (if any) conflicts will arise.

In MBFLO2, this issue does not arise in the context of summing an array. If it did, that would be handled with a shared-memory reduction tree as described in the above section. However, as MBFLO2 employs a finite volume numerical scheme, some algorithms require that every thread add a contribution to the 4 corners of the computational "cell" it is assigned to. This is particularly necessary when computing flux values ("flux" and "fluxtrb" routines). In the CPU solver, this is accomplished trivially with nested loops. However, in a parallel algorithm, this means that 4 threads may be attempting to add data to the same location in an array simultaneously, which results in a race condition.

One way of resolving a race condition is to take advantage of atomic operations. An atomic operation ensures that no other threads can access the applicable memory until the operation has completed. For example, an atomic addition blocks a variable's memory from being accessed by other threads until the entire addition operation, from reading the initial value to writing the modified one, is complete. There are also atomic operations for other arithmetic operations as well as maximum and minimum operations. Atomic operations are specified within the kernel, and the GPU does the rest. In the example code below, multiple threads may be attempting to access the same array element in the “du” array. For example, one thread’s [j+1][i+2] element will refer to the same location as [j+1][i+1] of another thread, so a race condition occurs. As shown, the atomicAdd operator ensures that only one thread will access a particular memory location at a time.

While atomic operations do solve the problem, they are unfortunately fairly time consuming. We have found that a superior method for addressing race conditions of this type is to make use of temporary memory buffers that are allocated on the GPU. Each thread will begin by adding only one of its contributions to the result array ("du" in the above example). For example, if every thread only adds the [j+1][i+2] contribution, then no race condition occurs. Each thread then stores the remaining 3 nodal contributions to a multidimensional temporary array in a location that no other thread will attempt to access. Finally, this kernel completes, and three consecutive kernels are executed with the sole task of adding one of the nodal contributions. For example, one kernel will add only the [j+1][i+1] contributions, avoiding race conditions. Thus, all contributions have eventually been added, and no race conditions have been encountered. Of course, using a temporary staging buffer is time consuming, but we have found that it is a better option than atomic operations. Figure 5 compares the computational speedup of the turbulent flux routine, which encounters this issue toward the end of the procedure, using both atomic operations and temporary buffers. The speedups are obtained by comparing execution time of 1 CPU and 1 GPU, using 4 increasingly dense computational grids As Fig. 5 shows, the speedups obtained using the temporary buffers technique are significantly higher.

V. Results

Achieving overall speedups with our multidisciplinary solver is the ultimate goal of this research. The multidisciplinary test case we used is a two-dimensional conjugate cylinder. This domain, cylindrical in shape, consists of a "solid" region in the center of the cylinder, which is surrounded by the fluid region. The thermal solver (the conjugate routine) is run only on the solid region, while the remainder of the MBFLO flow solver operates on the fluid portion of the domain. We tested this geometry using 4 increasingly dense computational grids. Each domain was divided into 13 blocks of an equal number of grid points. Eight of these blocks represent fluid regions of the domain, while the remaining 5 represent solid regions. The least dense computational grid consists of blocks that are 129 x 129 grid points, or roughly 16.6 thousand grid points per block. The densest grid is divided into blocks that are 1033x1033

grid points, or about 1.07 million grid points per block (over 13 million grid points in total). Figures 6 and 7 show the entire computational grid, divided into blocks, using the grid with a block size of 129x129 points.

Figures 6 and 7 show that the computational grid is non-uniform. Even though each block has the same number of grid points in both directions (in this case, 129x129), and each block has the same total number of grid points, the blocks are not identical in shape and do not comprise equal portions of the physical domain. Using 4 increasingly dense grids of this type, we ran a steady flow turbulent test case with a Reynolds number of 200,000. Figure 8 shows the color Mach number contours of the converged flow field. Figure 9 shows the converged temperature field along with the outline of the domain blocks . Note that the flow and temperature contours do not exhibit unsteadiness for this case due to the steady assumption and the use of local time-steps.

A. Speedups for Primary Solver Routines

The most computationally intensive solver routines, within the main simulation iteration loop, have been converted to run on CUDA-enabled GPUs. As previously mentioned and shown in Fig. 1, certain routines that are low-volume and/or highly dependent on divergent "if" logic, such as boundary condition routines or multi-grid routines, are left to run on the CPU. Initial geometry setup, as well as all other operations performed prior to the start of the iteration loop, remain tasked to a CPU. Due to the memory transfers required, and due to the fact that not all computations occur on the GPUs, the typical speedup achieved by an individual GPU kernel is higher than the speedup achieved by the overall simulation. A summary of the maximum speedup we have been able to achieve for key MBFLO2 routines is provided in Figs. 10 and 11. These results, for both the fluid and thermal (conjugate) routines, are obtained using a computational grid broken into blocks of 1033x1033 grid points, or roughly 1.07 million grid points per block. In other words, this number of threads actually launched to a CUDA kernel is about 1.07 million.

As the figures show, we have obtained a speedup of at least 10 for most of the key solver routines when using a sufficiently dense computational grid. The speedups were calculated by measuring the time

taken to complete each routine on a single CPU and a single GPU. As previously discussed, high computational volume is necessary to maximize the efficiency of a GPU. Thus, smaller (less dense) computational grids produce a lower (if any) computational speedup. Figures 12-20 show, for each of the above routines, the computational speedup we achieved over 4 increasingly dense computational grids. The speedup is plotted against the number of CUDA threads launched to the CUDA kernels, which is the number of grid points in a particular domain block. The densest computational grid is the same described above, in which there are about 1.07 million grid points per block. These results were obtained again by using only one CPU or GPU.

As shown by Figs. 12-20, launching many CUDA threads to the GPU is necessary to maximize GPU performance and achievable speedup. While the above results were obtained using only one GPU, multiple GPUs would be used in practice for a large simulation. Figures 21-28 compare, for 8 primary flow solver routines, the speedup obtained using between 4 and 8 GPUs against the speedup obtained using the same number of CPUs. These points are labeled with blue diamonds, as indicated by the provided legends. The plots also provide, again using between 4 and 8 GPUs, speedup obtained when comparing against a single CPU. These points are indicated by red squares. The densest computational grid (as described above) was used to produce these results. The reason these plots do not contain data between 1 and 3 GPUs is that a single GPU, either the Tesla C2050 or Tesla C2070, does not have enough available RAM to allocate the arrays necessary for all routines using the densest computational grid. For this grid consisting of more than 8 million total points, a minimum of 4 GPUs is required to allocate the necessary memory for the entire code. The Tesla C2070's 6GB of RAM can accommodate the data for 2 of these domain blocks, but not 3. The Tesla C2050, which only has 3GB of RAM, can only accommodate all of the data for 1 of these domain blocks. The single GPU results, using this same computational grid, in Figs. 12-20 were obtained by allocating only the arrays necessary for the particular routine being studied to be run on a GPU. This allowed us to measure how much faster a single GPU was executing the routine without running out of memory on the GPU. The following results in Figs. 21-28,

however, were obtained by running the full MBFLO2 code across multiple GPUs (and CPUs) and measuring the execution time of each routine.

As one might expect, the speedup we achieved using multiple GPUs remained fairly constant when using between 4-8 GPUs. Since the performance of the GPU is related to the number of threads launched to a CUDA kernel, and our kernels are launched once per domain block, we did not expect GPU assigned multiple blocks to out-perform a GPU assigned a single block. These results also confirm that using multiple CPU cores does not diminish performance on the CPU side. That is, each of the CPU cores seemed to produce similar performance. The algorithm used by MBFLO2 for assigning blocks to MPI processes simply divides the number of domain blocks by the total number of MPI processes. If the number of blocks is not evenly divisible by the number of processes, the last MPI process is assigned the remaining blocks. For example, if 4 GPUs were selected with our 13 block scheme, 3 GPUs would be assigned 3 blocks and the 4^{th} GPU would be assigned 4 blocks. Specifically, since the first 8 blocks are fluid blocks and the last 5 blocks are solid blocks, this would result in the first 2 GPUs being assigned 3 fluid blocks each, the third GPU being assigned 2 fluid blocks and 1 solid block, and the fourth GPU being assigned 4 solid blocks. With 8 GPUs selected for use, the first 7 GPUs are assigned 1 fluid block each while the 8^{th} GPU is assigned one fluid block along with all 5 solid blocks. Since, with our computational grid, the thermal solver runs so much faster per-block than the flow solver, using 8 GPUs turns out to be a fairly well load-balanced setup. This is discussed further in the following section. Looking at the performance compared to a single CPU, we see approximately the same results for using between 4 and 7 GPUs. This is due to the way that the equally-sized domain blocks were being assigned to the MPI processes. When using less than 8 GPUs or CPUs, at least one process (and associated GPU/CPU) must be assigned at least 2 fluid blocks. Since the iteration must complete on all MPI processes before the simulation can continue, the process assigned more work creates a bottleneck. However, when 8 GPUs or CPUs are used, each MPI process is assigned only one fluid block, and the speedup compared to one CPU increases. This is to be expected, since here we are comparing only the execution times of the routines themselves, and not memory transfers between GPUs and the host

machines or MPI related communication overhead. Lastly, there is no plot for the conjugate routine due to the fact that the thermal solver operates on only the 5 solid domain blocks that cannot be spread across as many as 8 GPUs.

B. Overall Speedups

As previously mentioned, the computational domain is divided into 8 fluid region blocks and 5 solid region blocks. Since the conjugate routine only operates on solid blocks, and the rest of the primary solver routines only operate on fluid blocks, it is possible for a particular MPI process (and ultimately a particular CPU or GPU) to be assigned only fluid or only solid blocks. Thus, in practice, a load balancing analysis is required to determine the optimal distribution of solid and fluid blocks among processors. Specifically, this involves timing how long the iteration takes for each type of block and distributing them in such a way that all MPI processes take approximately the same amount of time to complete the iteration. Since every process must complete the iteration before the next one can begin, the slowest process is ultimately the bottleneck. As our achieved GPU speedups for each routine do vary, the ratio of conjugate to fluid execution times will be slightly different if measured using the GPU enabled solver. However, since our achieved speedup using the conjugate routine with dense grids is roughly 10, as most other routines, this difference is slight and the ideal block distribution is likely to be the same. For our particular test case, with 13 blocks of equal size, we determined that the time it takes to run all 5 solid blocks through the conjugate routine is insignificant in comparison to the time it takes to run one fluid block through the flow solver routines. Figure 29 shows the execution time required to complete a single iteration for each processor when using a total of 8 CPUs or 8 GPUs with our densest computational grid. In both cases shown in Fig. 29, using either 8 CPUs or 8 GPUs, 7 of the MPI processes are assigned 1 fluid block while the remaining process is assigned 1 fluid block along with all 5 solid blocks. Looking at the plot, it is not obvious which process is carrying the extra load. In both cases, it is the 8^{th} MPI process that is assigned the solid blocks. Since the execution times in this example are so similar, the load is fairly

well balanced between MPI processes. Using this configuration, we obtained speedups using the same 4 computational grids, which are shown in Fig. 30.

Figure 30 shows speedups comparing 8 GPUs both against 8 CPUs and single CPU performance. Note that the RAM restrictions of a single Tesla C2050/C2070 that prevent us from running the entire simulation on one GPU do not apply to running the simulation only on a single CPU core, which makes use of the machine's main RAM. Comparing against 8 CPUs, we have obtained an overall speedup of just over 6 when using our densest computational grid. Furthermore, our simulation running on 8 GPUs runs about 40 times faster than it does on a single CPU.

VI. Conclusions

Implementing Graphical Processing Units into computing clusters can greatly increase the overall speed of compressible fluid simulations, including a conjugate thermal diffusion solver. Moreover, we have found that CUDA-enabled GPUs are well suited for the finite-volume numerical scheme used here. We have found that ensuring global memory accesses are coalesced is the most influential of optimization processes, as non-coalesced reads and writes can decrease the speed of a CUDA kernel by more than a factor of 2. Furthermore, minimizing memory copies between host RAM and GPU RAM, within the primary iteration loop, is crucial to maximizing overall simulation speedup. We have also found that when such memory transfers are necessary, taking advantage of CUDA streams and asynchronous memory copies provides a significant added advantage. Finally, while our results are encouraging, we believe higher speedups can be obtained with three-dimensional simulations, newer hardware, and possibly some alternative techniques.

Acknowledgments

This effort was collaboratively funded by the Wright-Patterson Air Force Research Laboratory under contract 11-S590-0020-18-C1 R4 with Dr. John Clark as technical monitor and the University of California, Davis. The authors would like to thank the managers of both facilities for their support. The

authors would also like to thank the managers of Intelligent Light for the academic grant to the FIELDVIEW scientific visualization software.

References

[1]Davis, R. L. and Dannenhoffer III, J. F., A Detached-Eddy Simulation Procedure Targeted for Design," Vol. 24, No. 6, Nov./Dec. 2008.

[2]Davis, R. L., Ni, R. H., and Carter, J. E., "Cascade Viscous Flow Analysis Using The Navier-Stokes Equations," *AIAA Journal of Propulsion and Power*, Vol. 3, No. 5, September-October 1987.

[3]Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A. E., and Purcell, T., A Survey of General-Purpose Computation on Graphics Hardware," Computer Graphics Forum, Vol. 26, No. 1, 2007, pp. 80-113.

[4]Stone, C. and Davis, R. L., "High-Performance Multi-Disciplinary Fluid/Thermal Prediction using Combined Multi-Core/Multi-GPGPU Computer Systems," PETTT PP-CFD-KY05-002-P3 Final Report to the Air Force Research Laboratory, August 22, 2014.[5]Bolz, J., Farmer, I., Grinspun, E., and Schroder, P., Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," ACM Transactions on Graphics, Vol. 22, No. 3, July 2003, pp. 917-924.

[6]Brandvik, T. and Pullan, G., Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware," Proceedings of the 48th AIAA Aerospace Sciences Meeting and Exhibit, No. AIAA 2008-607, Jan. 2008.

[7]Goodnight, N., Woolley, C., Lewin, G., Luebke, D., and Humphreys, G., A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware," Graphics Hardware 2003, July 2003, pp. 102-111.

[8]Hagen, T. R., Hjelmervik, J. M., Lie, K.-A., Natvig, J. R., and Henriksen, M. O., Visual simulation of shallow-water waves," Simulation Modelling Practice and Theory, Vol. 13, No. 8, Nov. 2005, pp. 716-726.

[9]Harris, M. J., Baxter III, W., Scheuermann, T., and Lastra, A., Simulation of Cloud Dynamics on Graphics Hardware," Graphics Hardware 2003, July 2003, pp. 92-101.

[10]Kruger, J. and Westermann, R., Linear Algebra Operators for GPU Implementation of Numerical Algorithms," ACM Transactions on Graphics, Vol. 22, No. 3, July 2003, pp. 908-916.

[11]Li, W., Fan, Z., Wei, X., and Kaufman, A., GPU-Based Flow Simulation with Complex Boundaries," GPU Gems 2, edited by M. Pharr, chap. 47, Addison Wesley, March 2005, pp. 747-764.

[12]Scheidegger, C. E., Comba, J. L. D., and da Cunha, R. D., Practical CFD Simulations on Programmable Graphics Hardware using SMAC," Computer Graphics Forum, Vol. 24, No. 4, 2005, pp. 715-728.

[13]Hagen, T. R., Lie, K.-A., and Natvig, J. R., Solving the Euler Equations on Graphics Processing Units," Proceedings of the 6th International Conference on Computational Science, Vol. 3994 of Lecture Notes in Computer Science, Springer, May 2006, pp. 220-227.

[14]Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S., “GPU Cluster for High Performance Computing," SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, Washington, DC, USA, 2004

[15]Goddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S. H., Grajewski, M., and Turek, S., Exploring weak scalability for FEM calculations on a GPU-enhanced cluster," Parallel Computing, Vol. 33, No. 10-11, 2007, pp. 685-699.

[16]Phillips, E. H., Zhang, Y., Davis, R. L., and Owens, J. D., “Rapid Aerodynamic Performance Prediction on a Cluster of Graphical Processing Units,”*AIAA Journal of Aerospace Computing, Information, and Communication,* Vol. 8, August 2011, pp 237-249.

[17]Wilcox, D. C., Turbulence Modeling for CFD, DCW Industries, Inc., La Cannada, CA, 1998.

[18]Ni, R. H., “A Multiple Grid Scheme for Solving the Euler Equations,” *AIAA Journal*, Vol. 20, 1982, pp. 1565- 1571.

[19]Dannenhoffer, J. F., “Grid Adaptation for Complex Two-Dimensional Transonic Flows”, CFDL-TR-87-10, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, August 1987.

[20]Jameson, A., “Time Dependent Calculations Using Multi-grid, with Applications to Unsteady Flows Past Airfoils and Wings,” AIAA 91-1596, June 1991

[21]“A Conjugate Heat Transfer RANS/DES Simulation Procedure,” Fife, M. and Davis, R. L., AIAA 2009-913, presented at the AIAA 47th Aerospace Sciences Meeting, Orlando Florida, January 2009.

[22]Phillips, Everett, Davis, R. L., and Owens, J. D., “Unsteady Turbulent Simulations on a Cluster of Graphics Processors”, AIAA 2010-5036, June 2010.

List of Figures

Figures

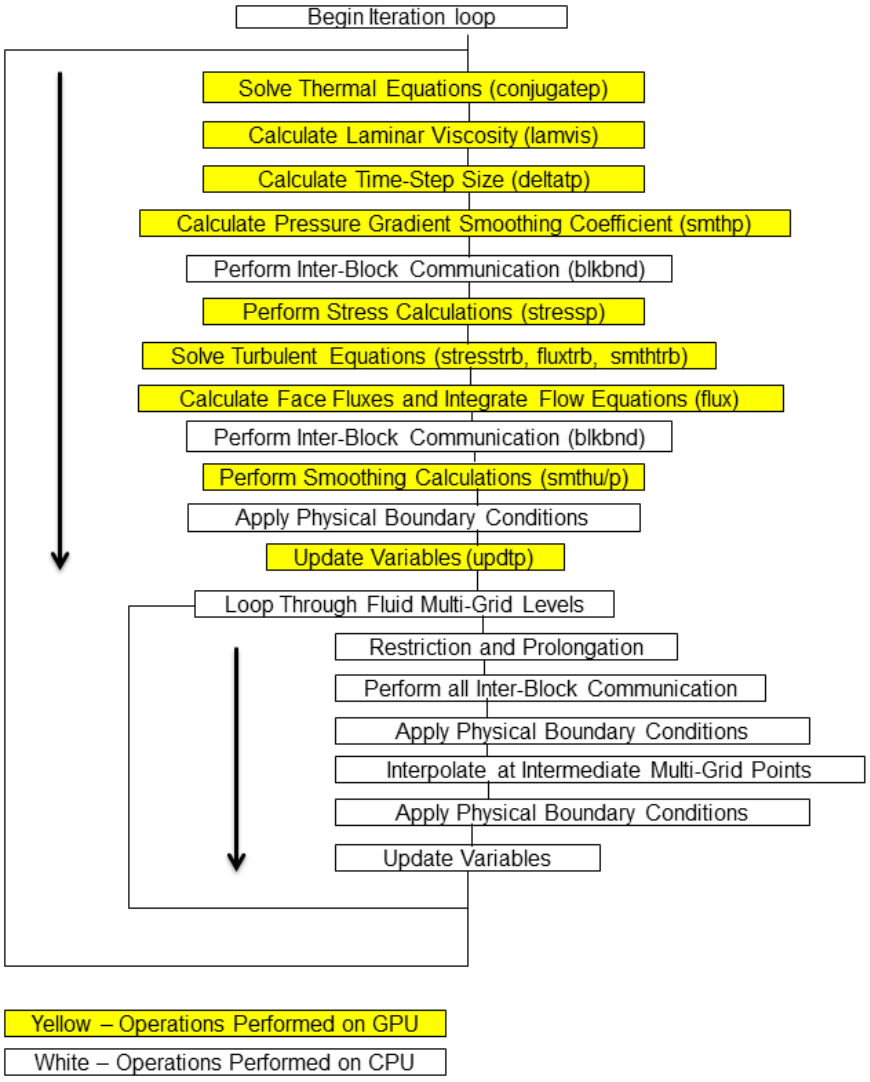

Figure 2: Diagram of MBFLO primary iteration loop.

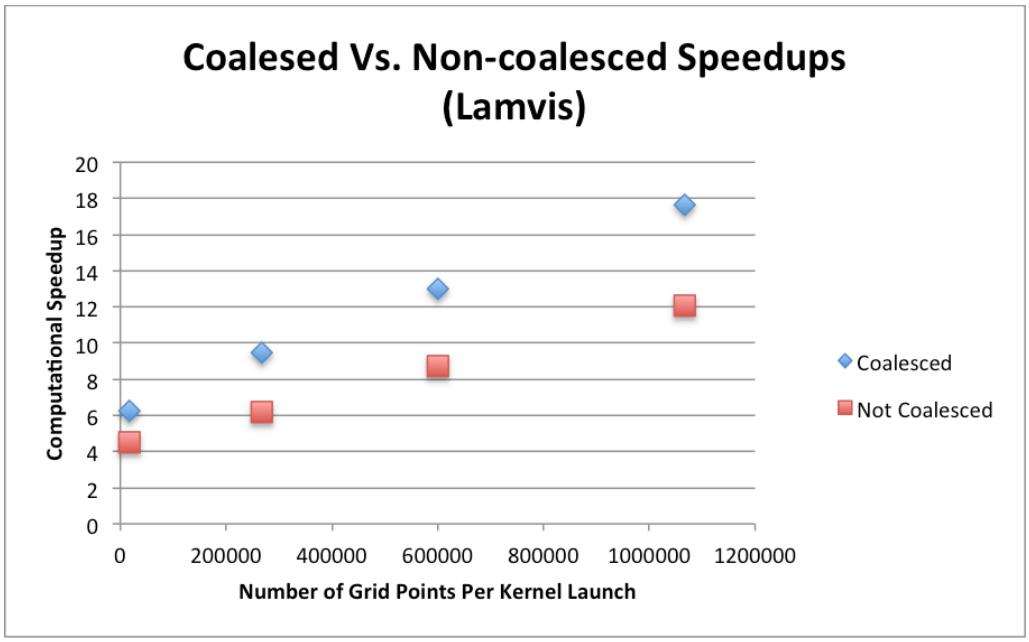

Figure 2: Computational speedups for coalesced and non-coalesced CUDA kernel launch schemes.

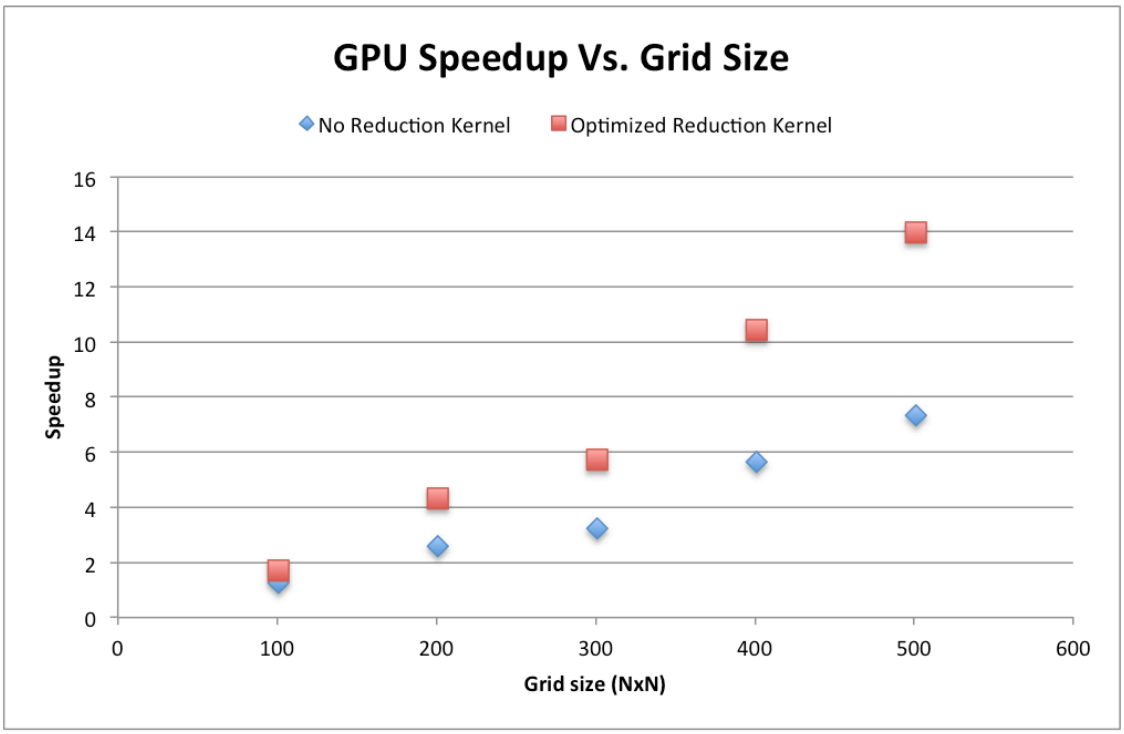

Figure 3: Speedup vs. grid density for optimized reduction kernel and for no reduction kernel.

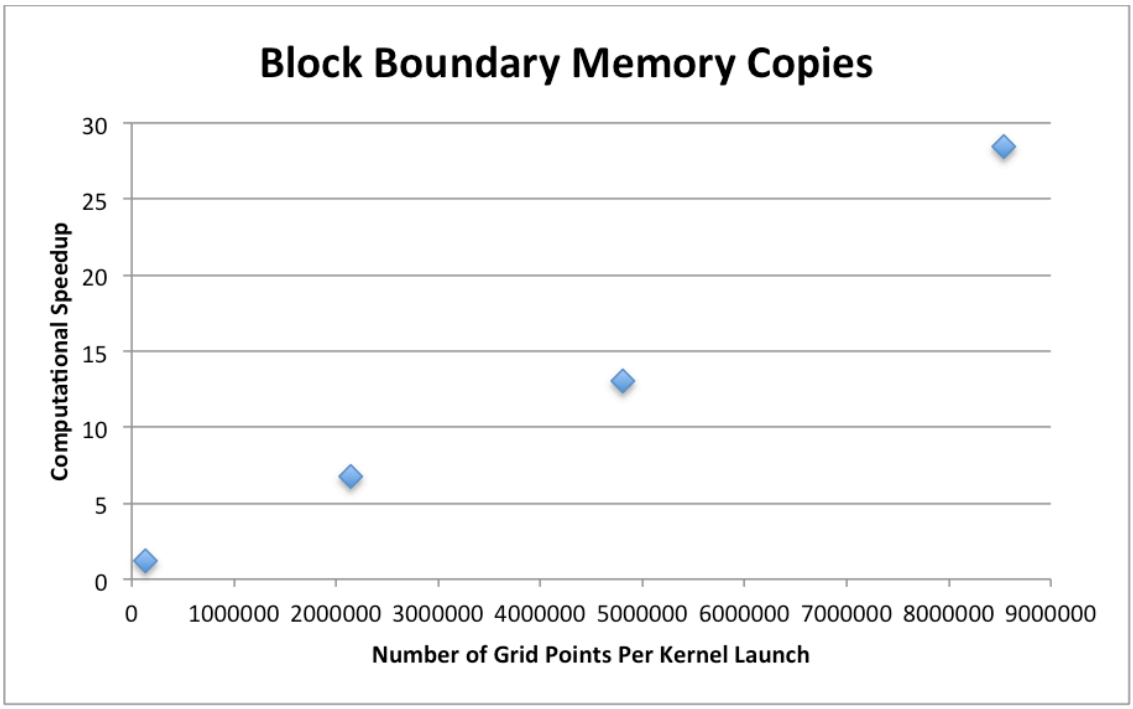

Figure 4: Speedup achieved by copying only block boundaries compared to entire data arrays.

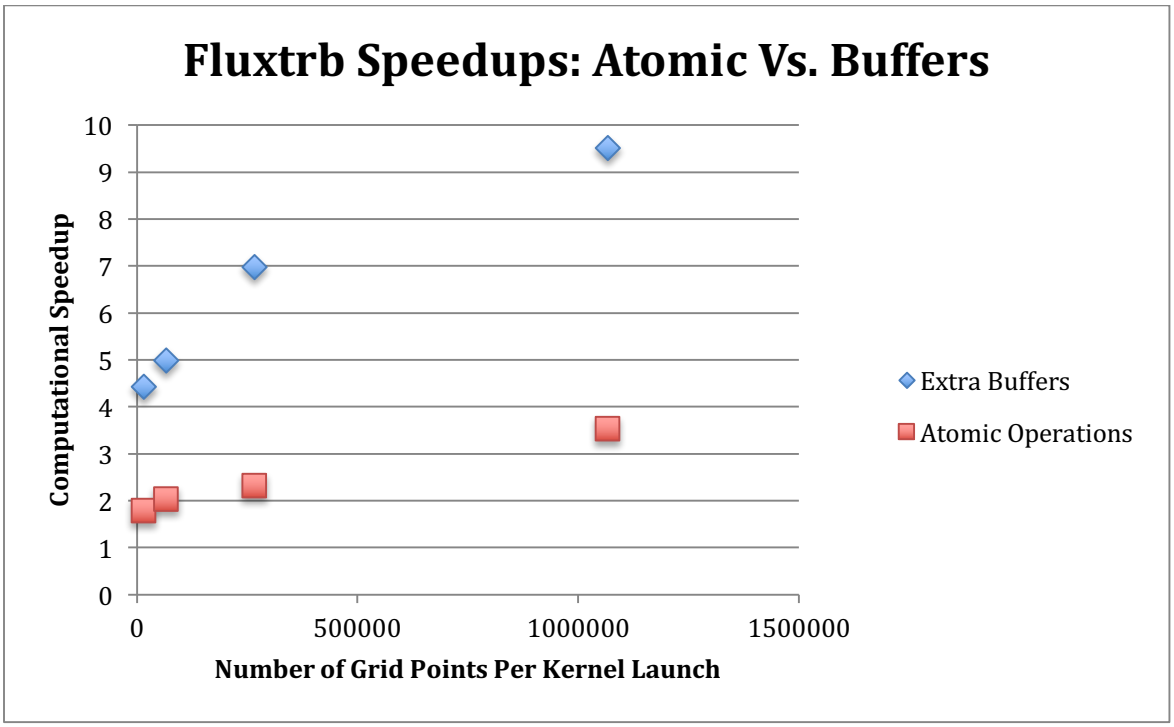

Figure 5: Speedups Obtained Using Atomic Operations and Temporary Buffers.

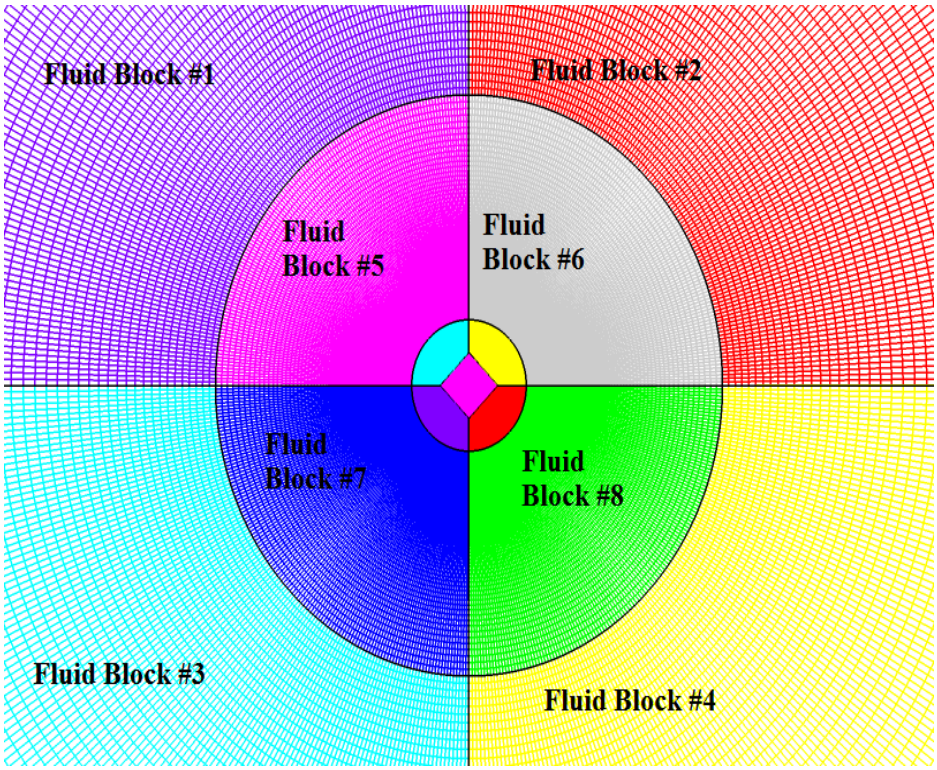

Figure 6: All Fluid Blocks of the Conjugate Cylinder.

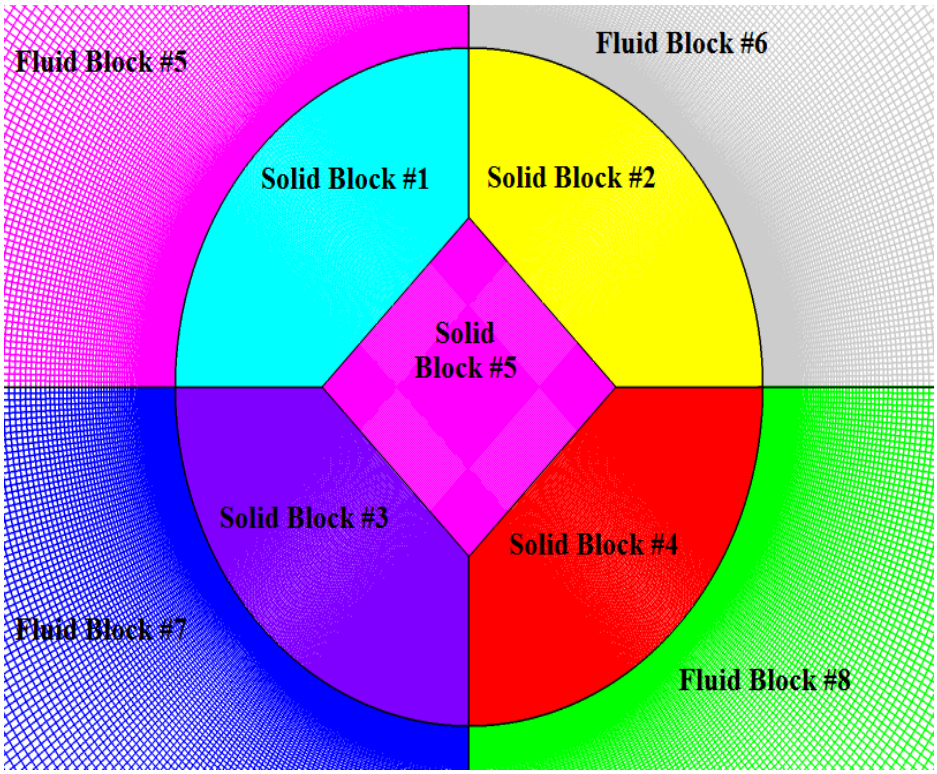

Figure 7: Solid Blocks of the Conjugate Cylinder grid.

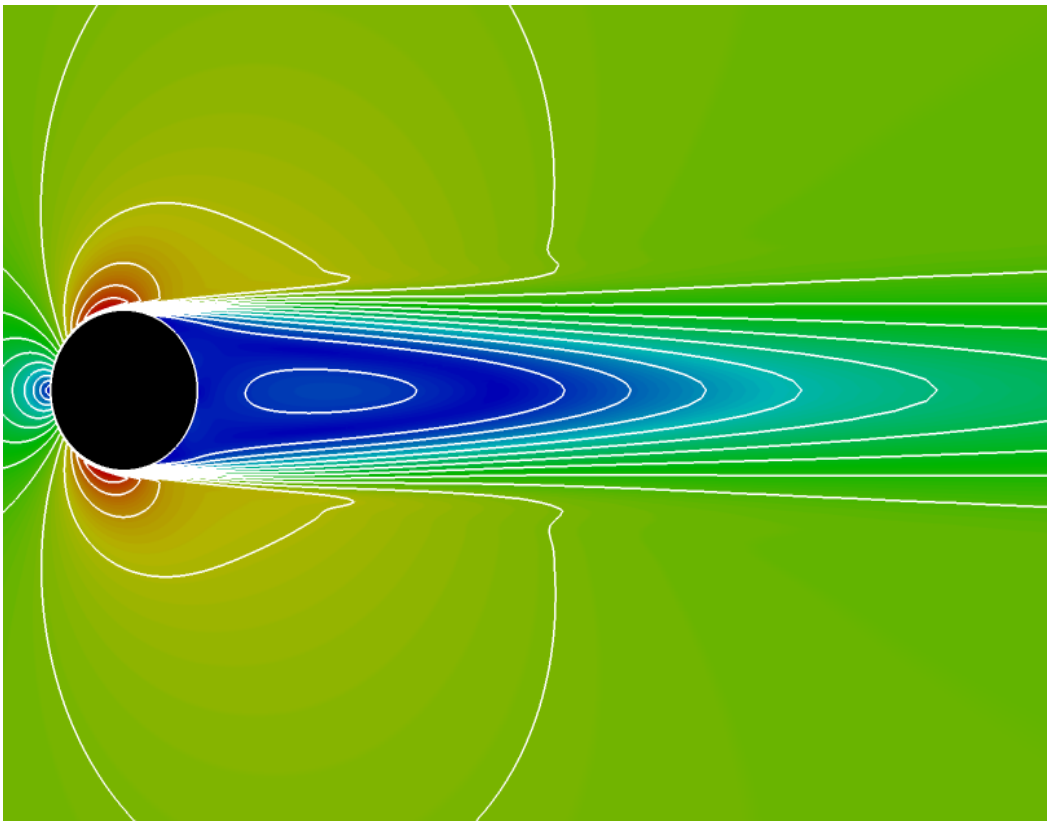

Figure 8: Flow Solution Mach Number Contours.

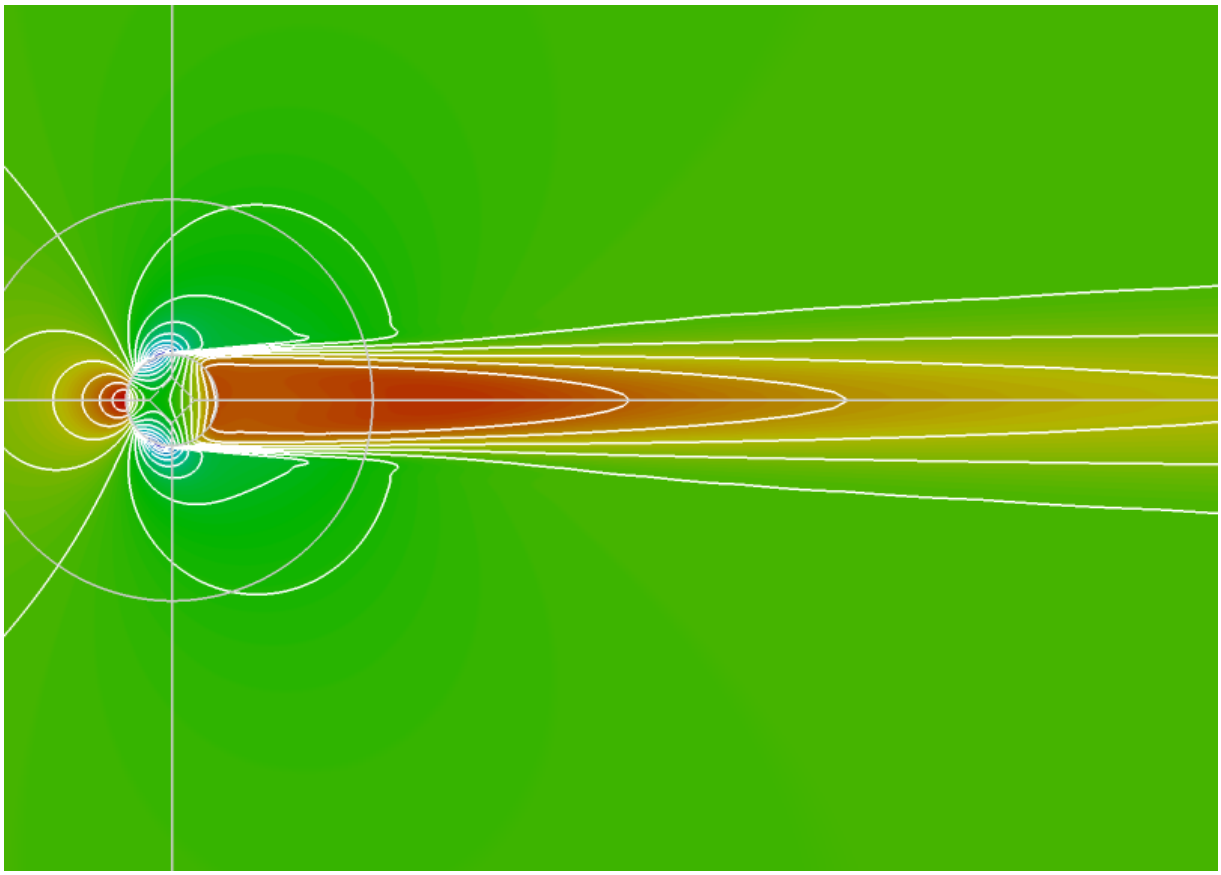

Figure 9: Temperature Solution Contours.

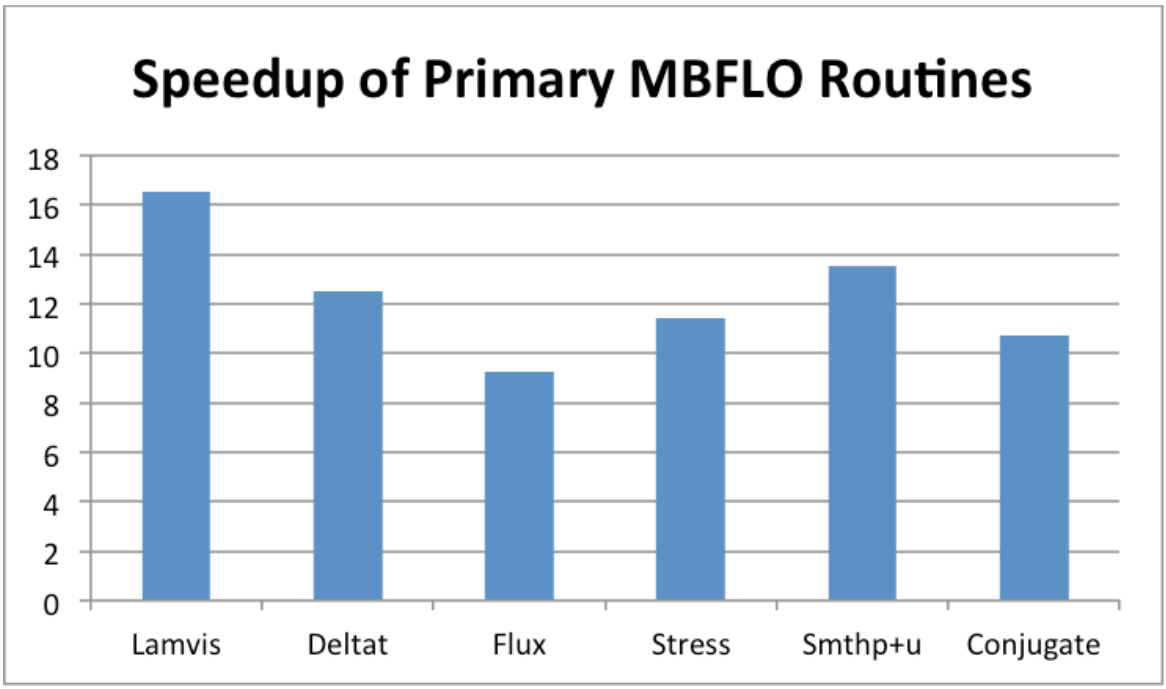

Figure 10: Speedup of Primary MBFLO Routines.

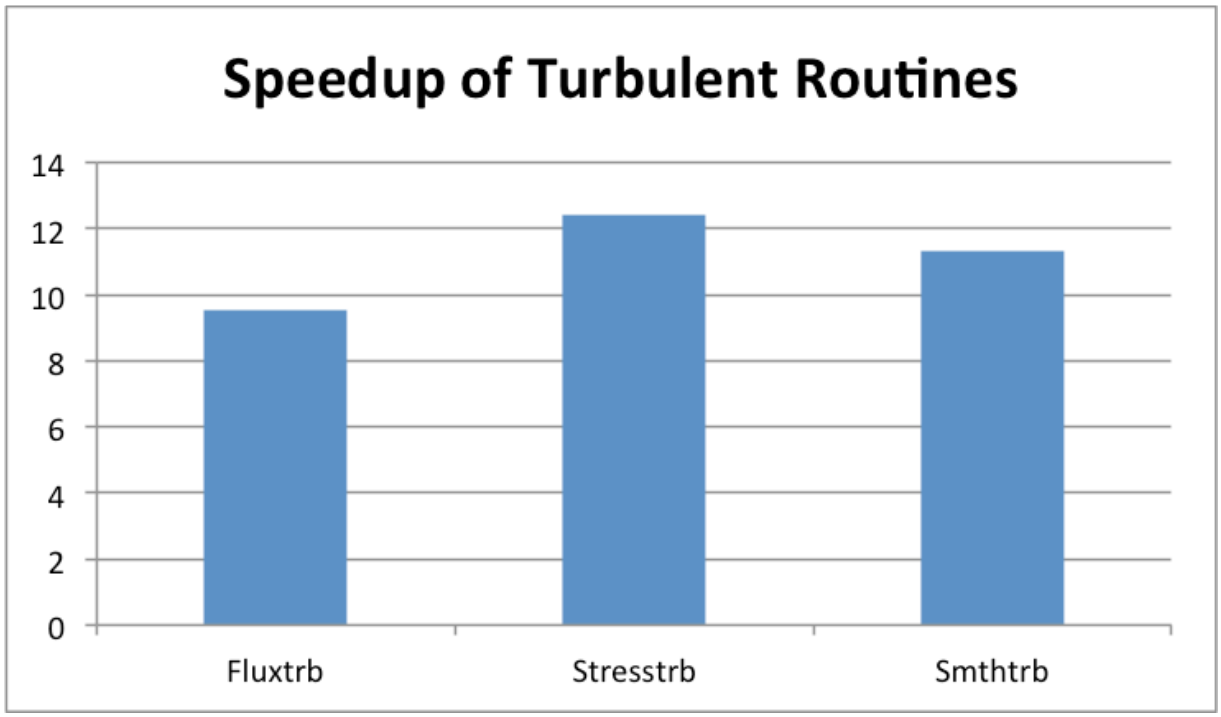

Figure 11: Speedup of Turbulent MBFLO Routines.

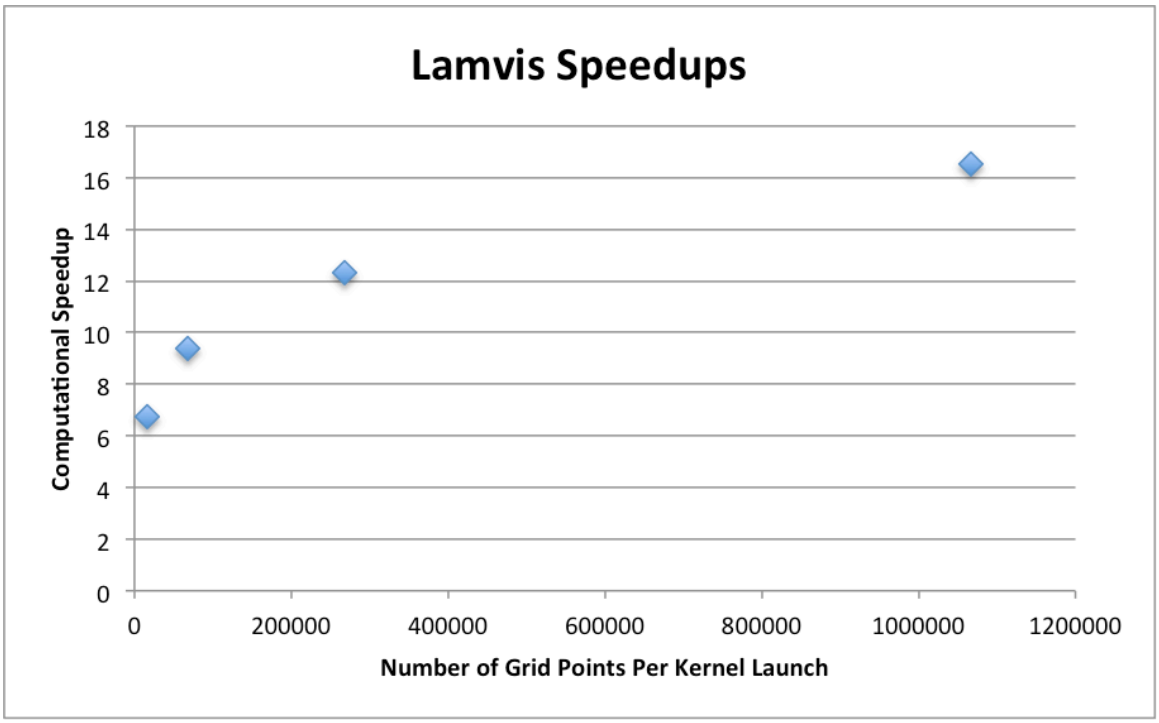

Figure 12: Speedups for Lamvis Routine.

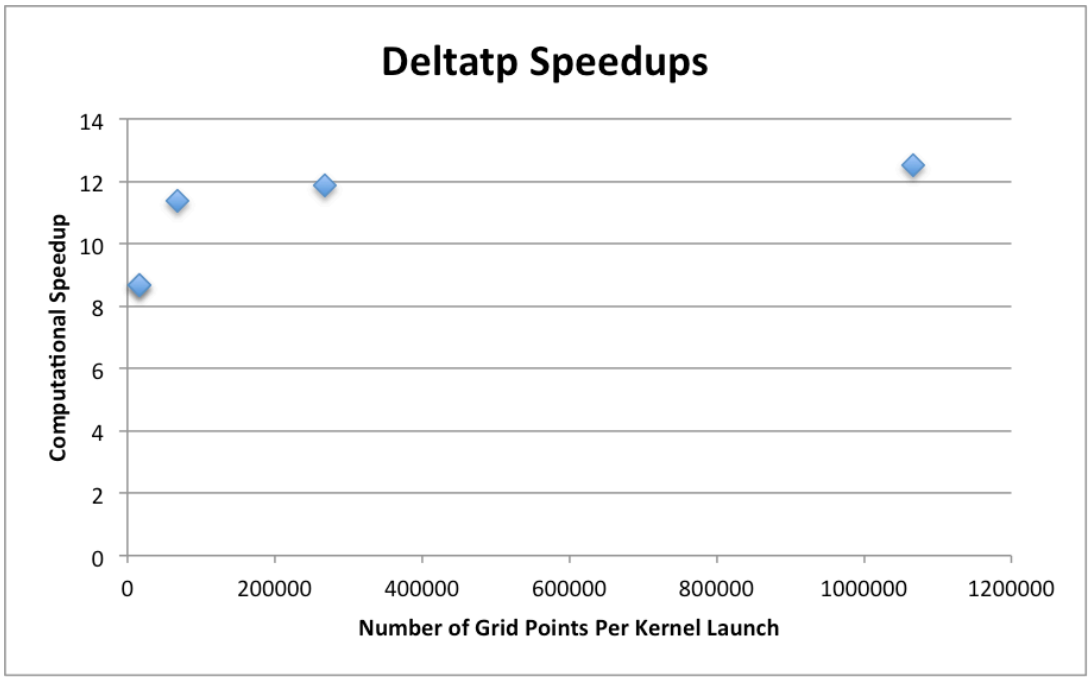

Figure 13: Speedups for Deltatp Routine.

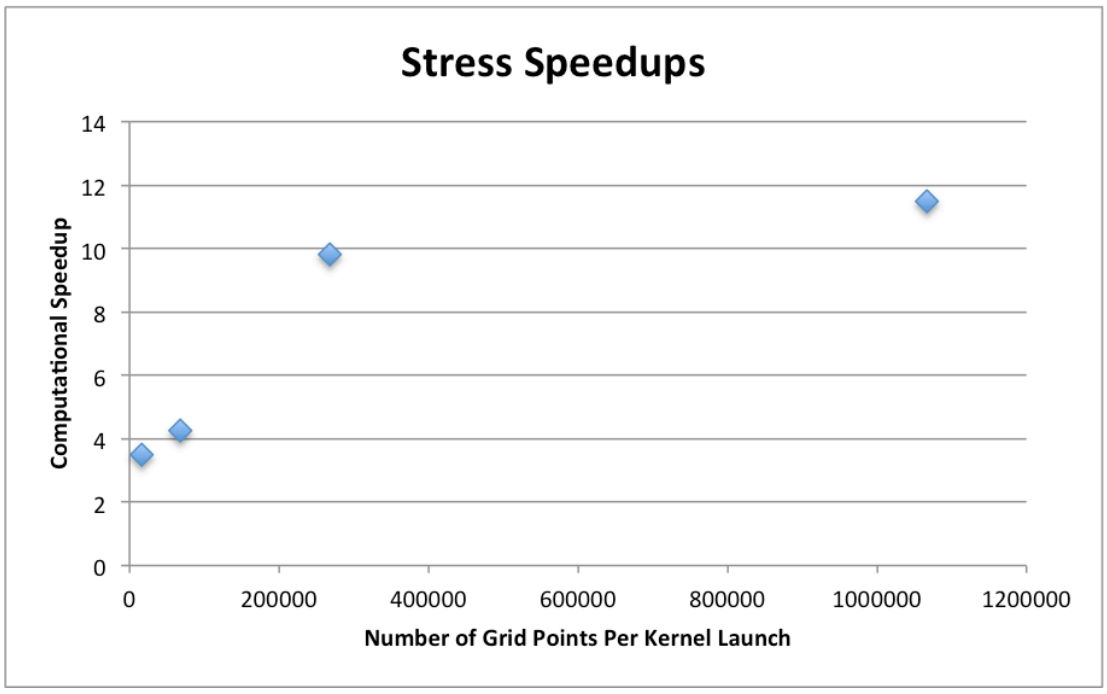

Figure 14: Speedups for Stress Routine.

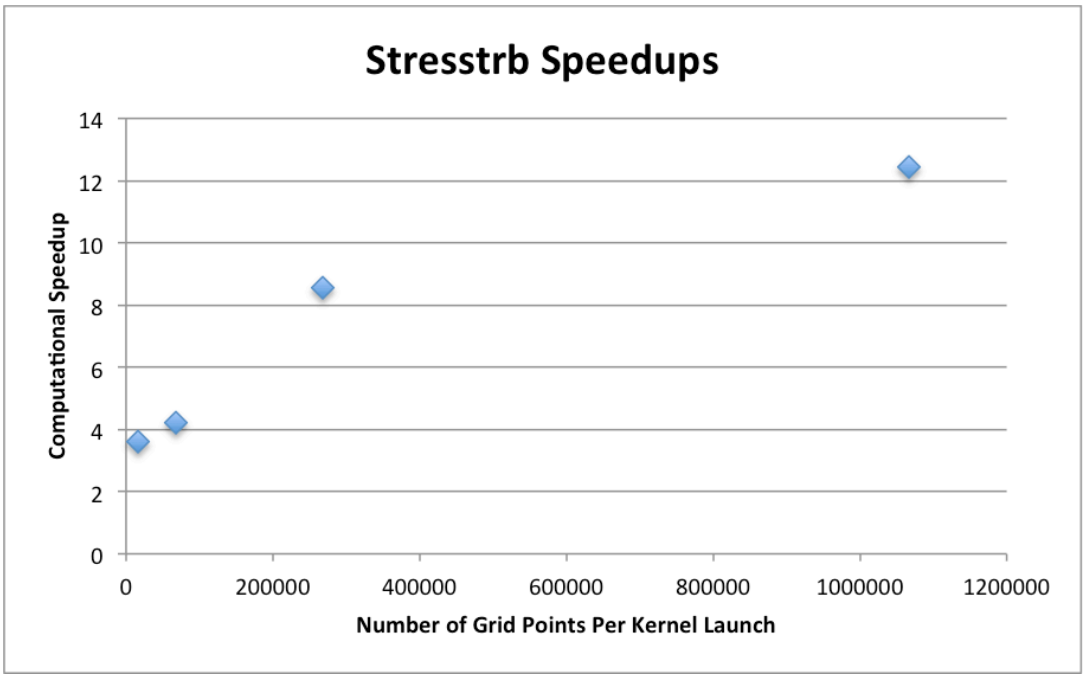

Figure 15: Speedups for Stresstrb Routine.

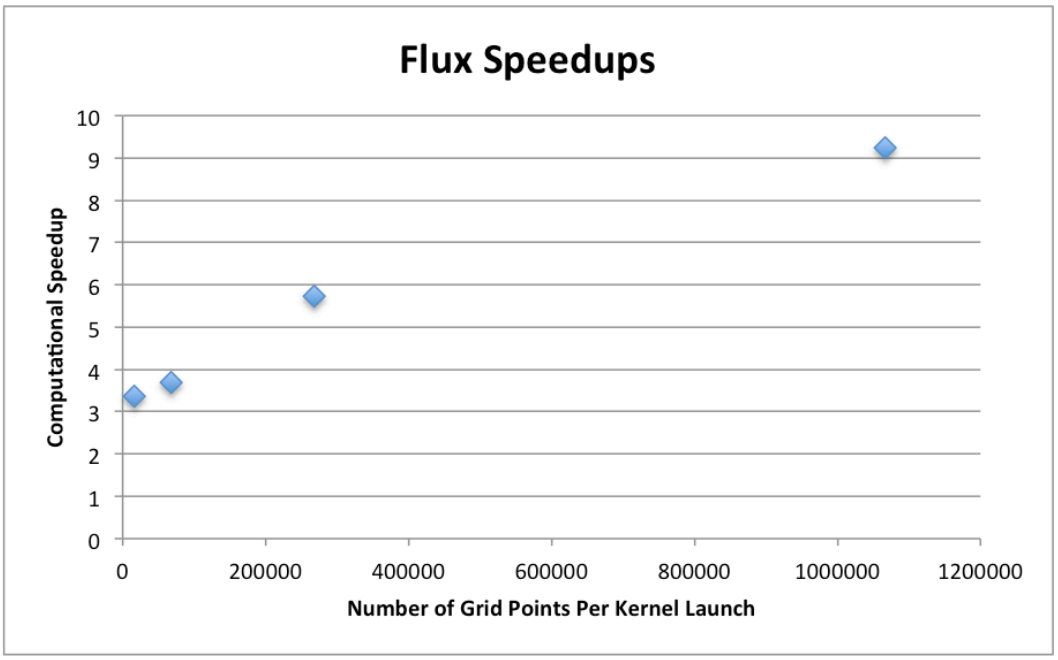

Figure 16: Speedups for Flux Routine.

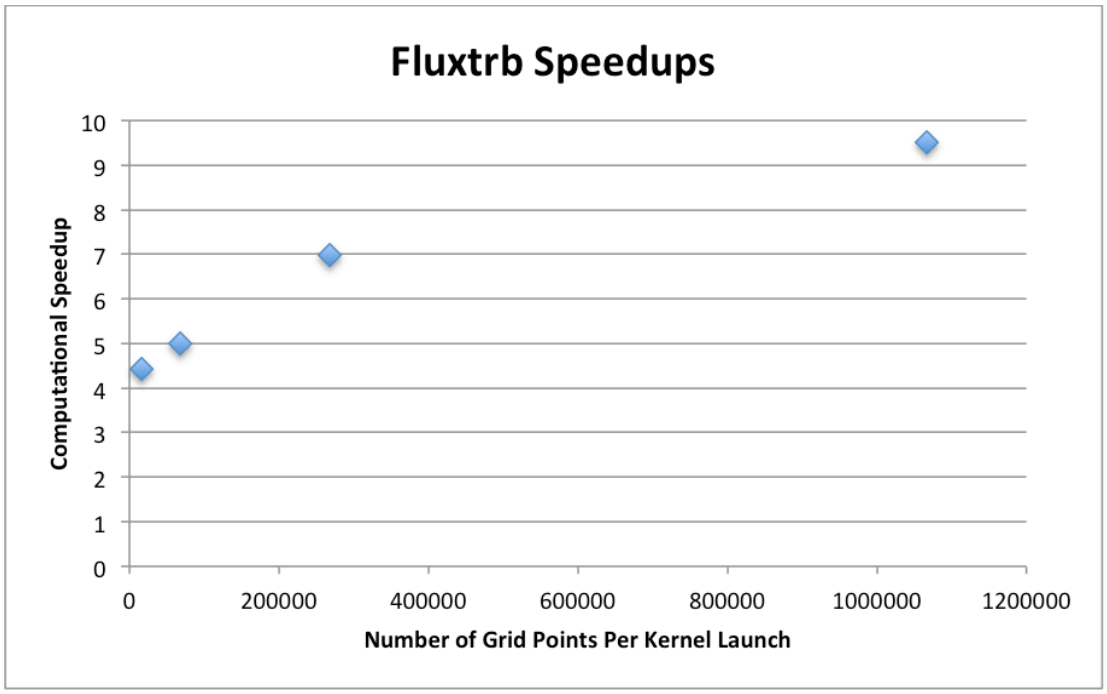

Figure 17: Speedups for Fluxtrb Routine.

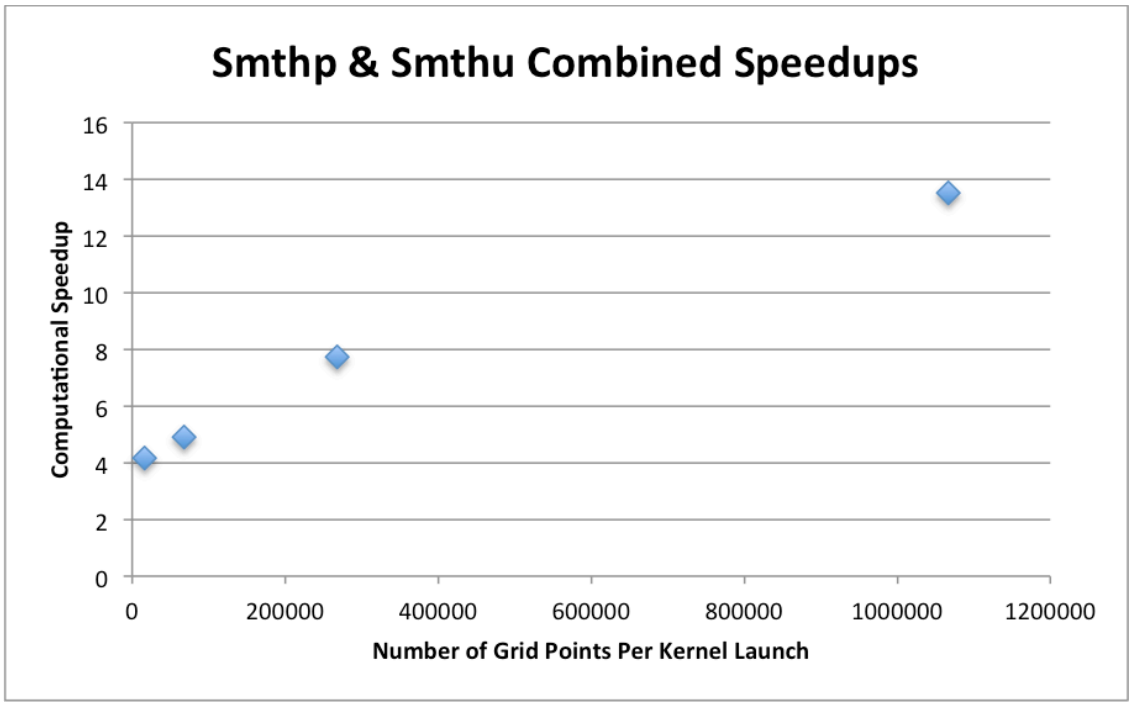

Figure 18: Speedups for Smoothing Routines.

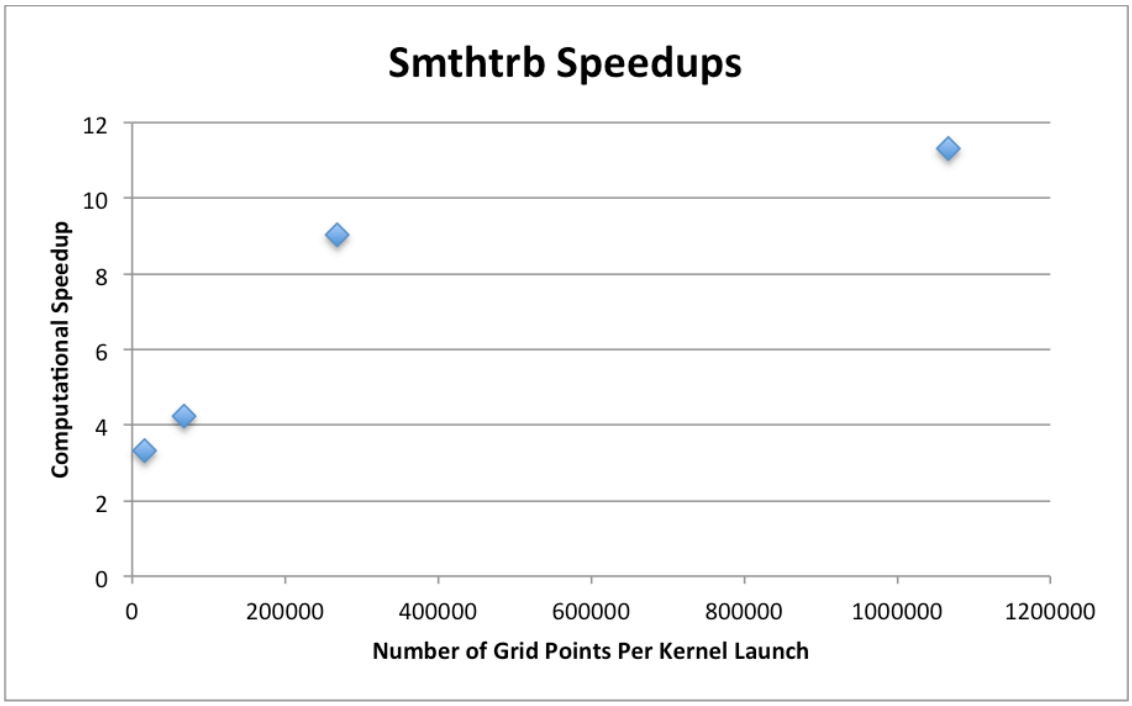

Figure 19: Speedups for Smthtrb Routine.

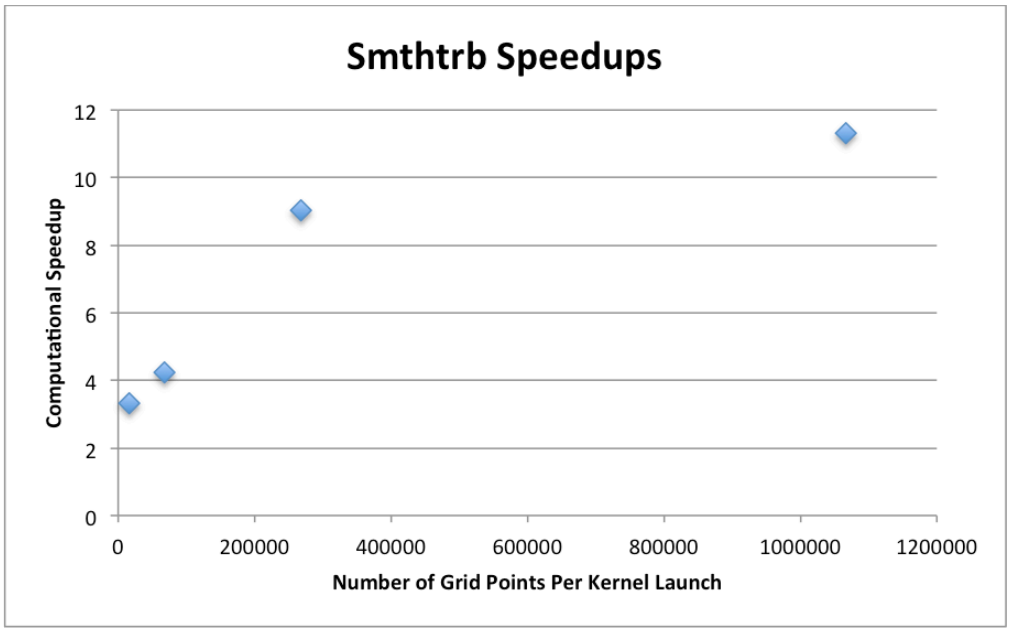

Figure 20: Speedups for the Thermal (Conjugatep) Solver.

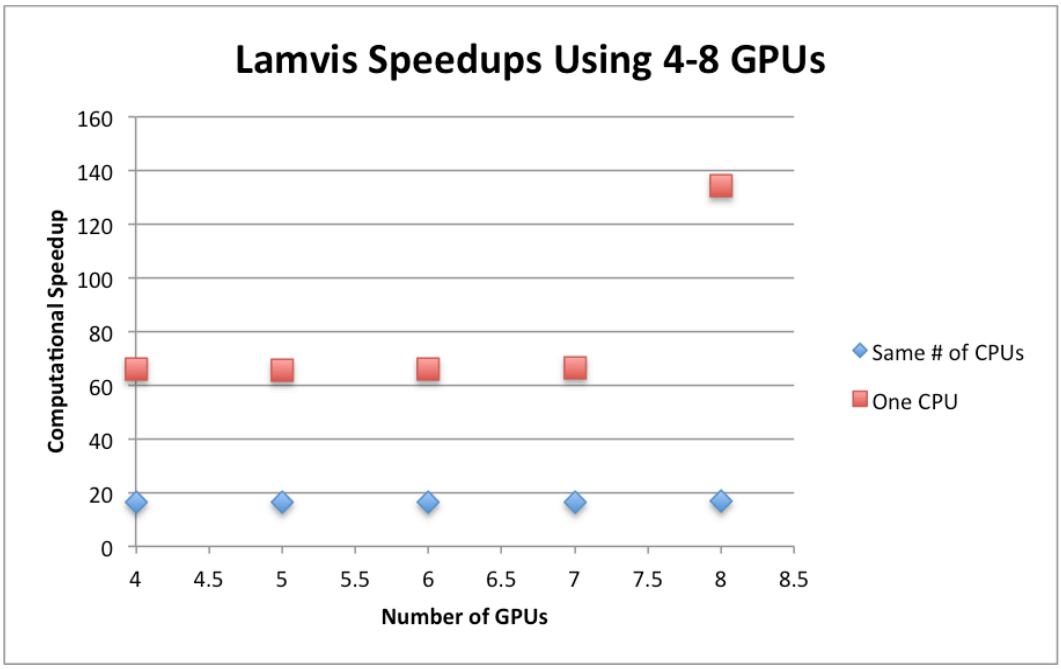

Figure 21: Multi-GPU Speedups for Lamvis.

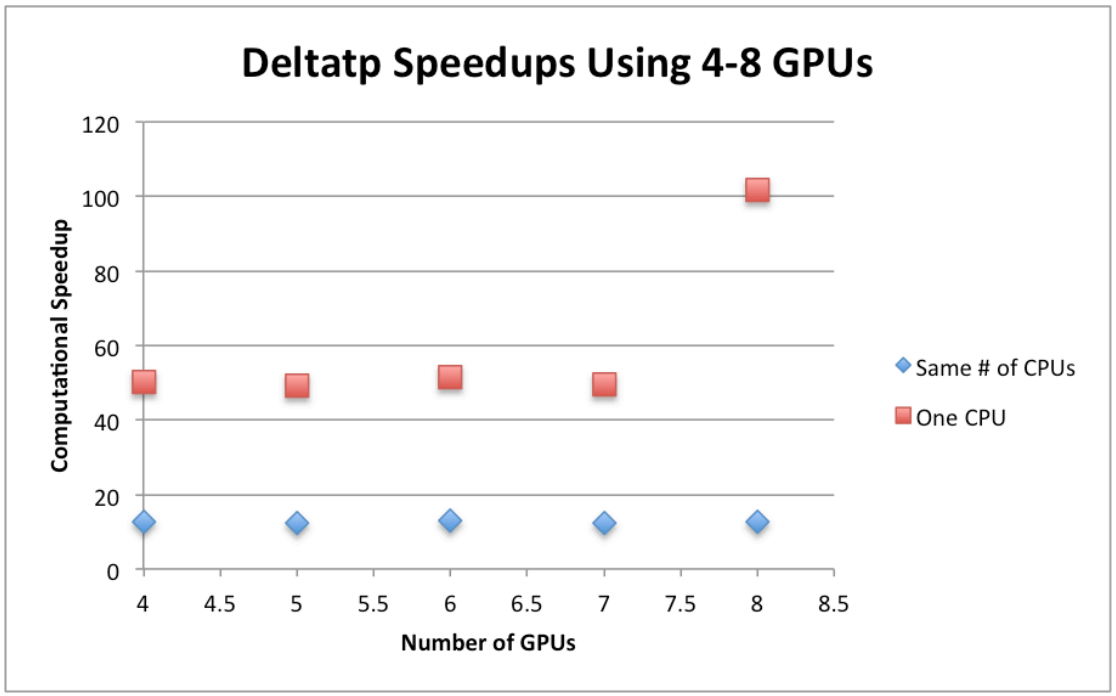

Figure 22: Multi-GPU Speedups for Deltatp.

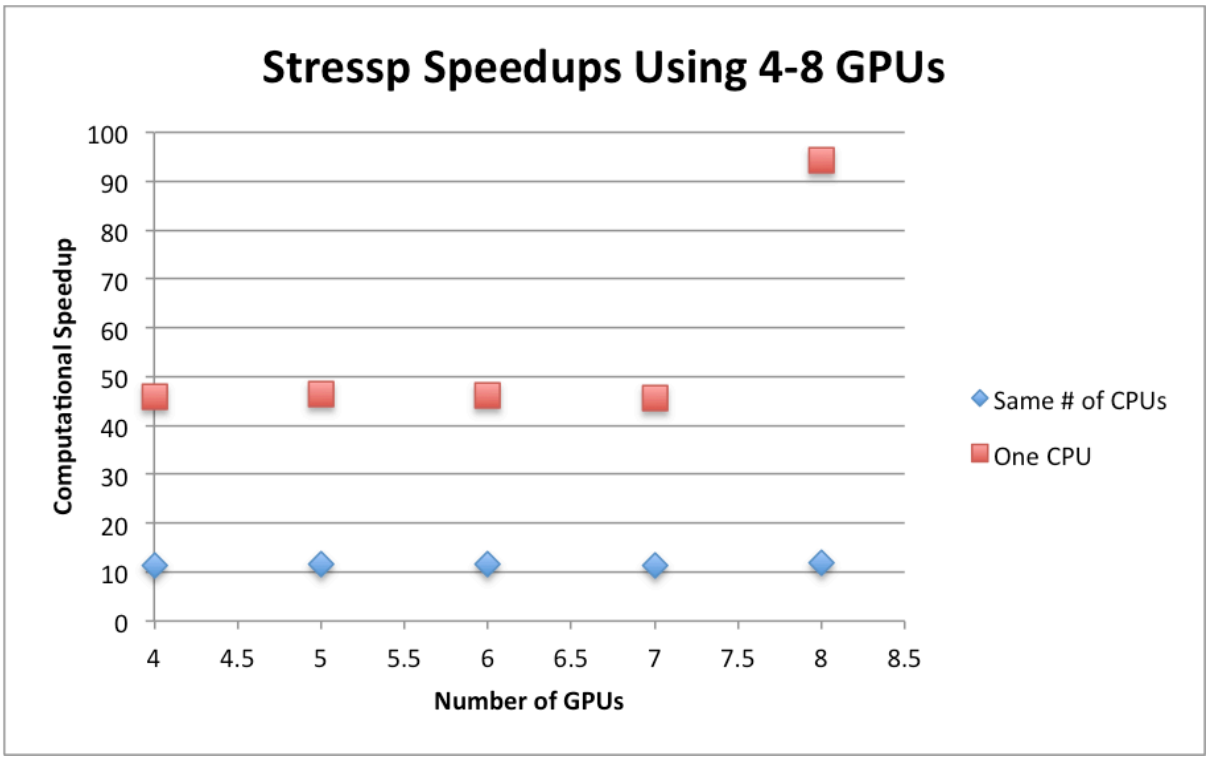

Figure 23: Multi-GPU Speedups for Stressp.

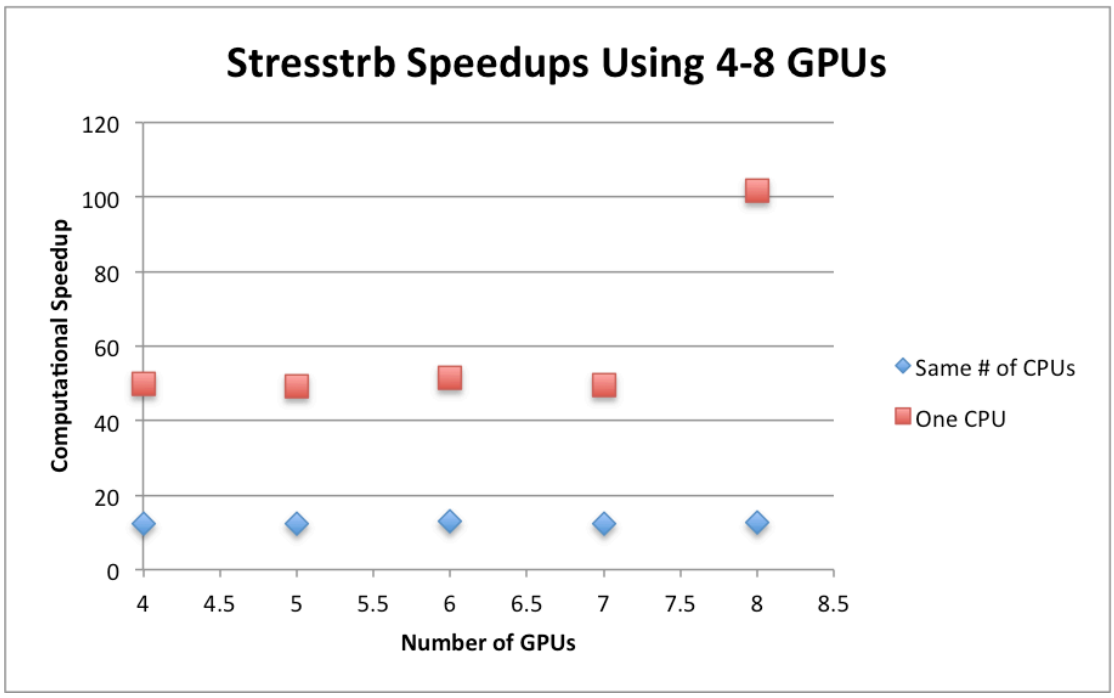

Figure 24: Multi-GPU Speedups for Stresstrb.

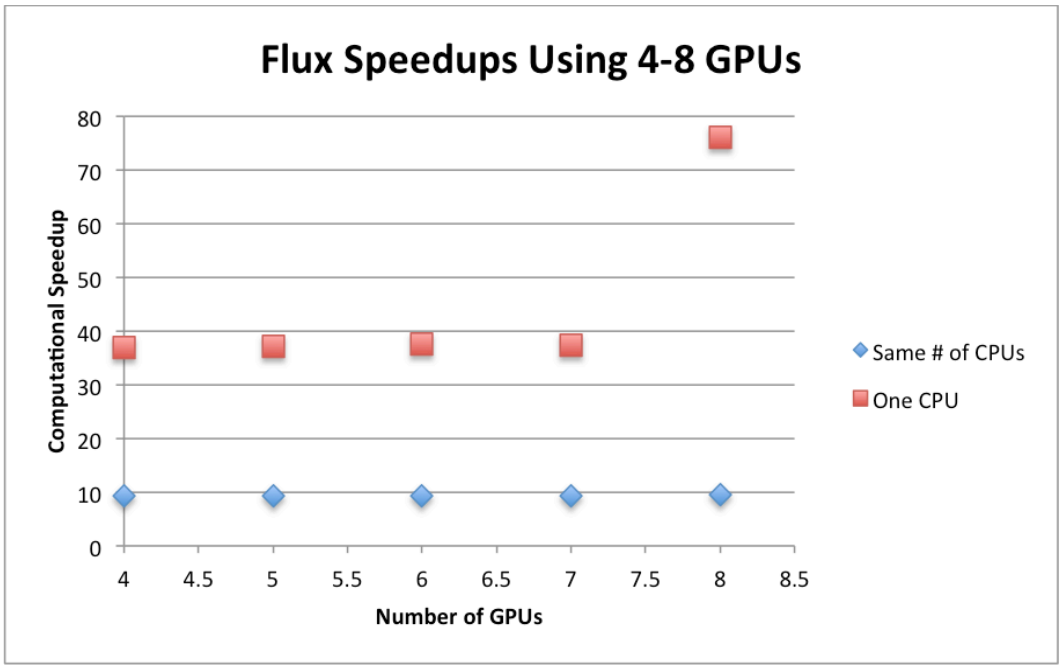

Figure 25: Multi-GPU Speedups for Flux.

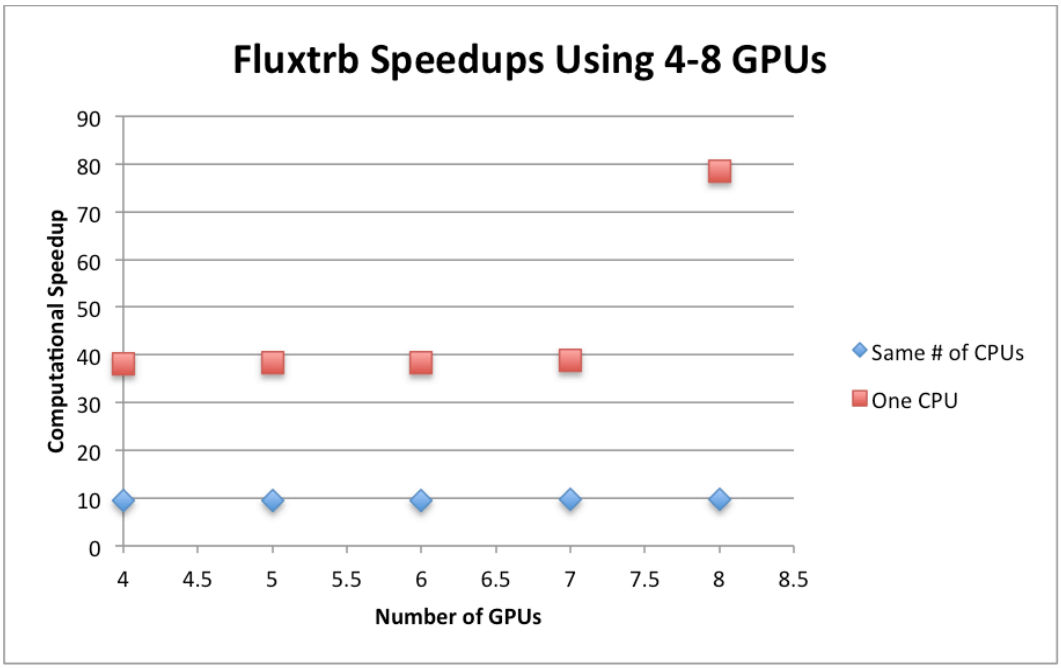

Figure 26: Multi-GPU Speedups for Fluxtrb.

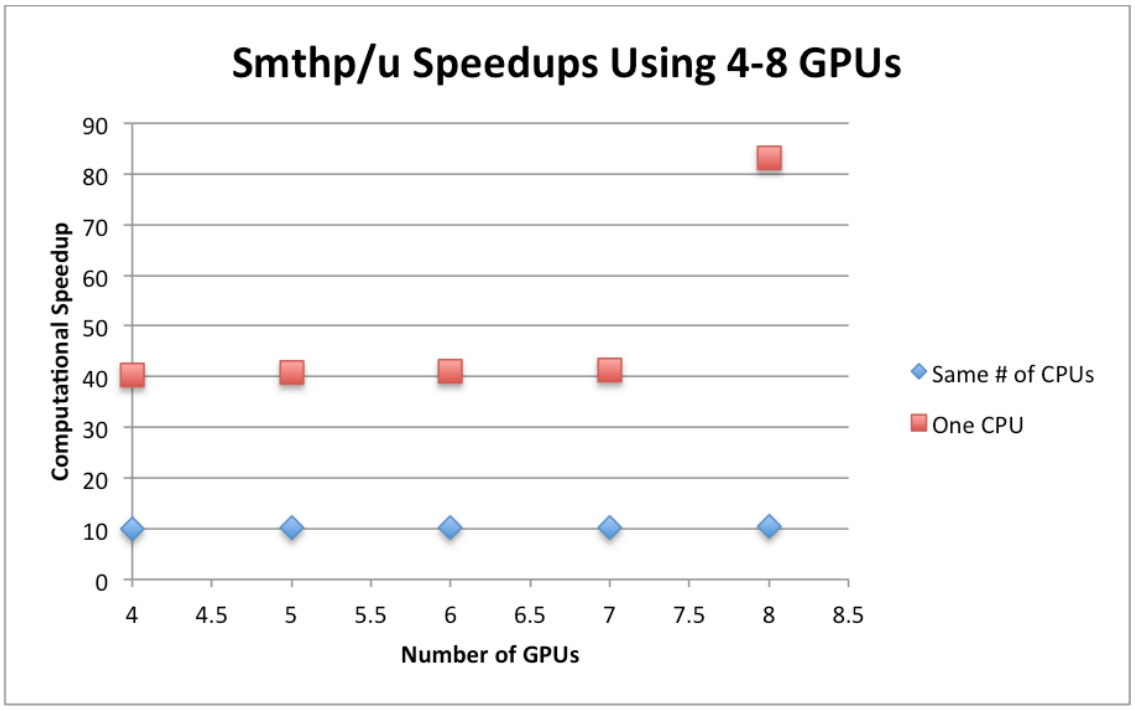

Figure 27: Multi-GPU Speedups for Smthp/u.

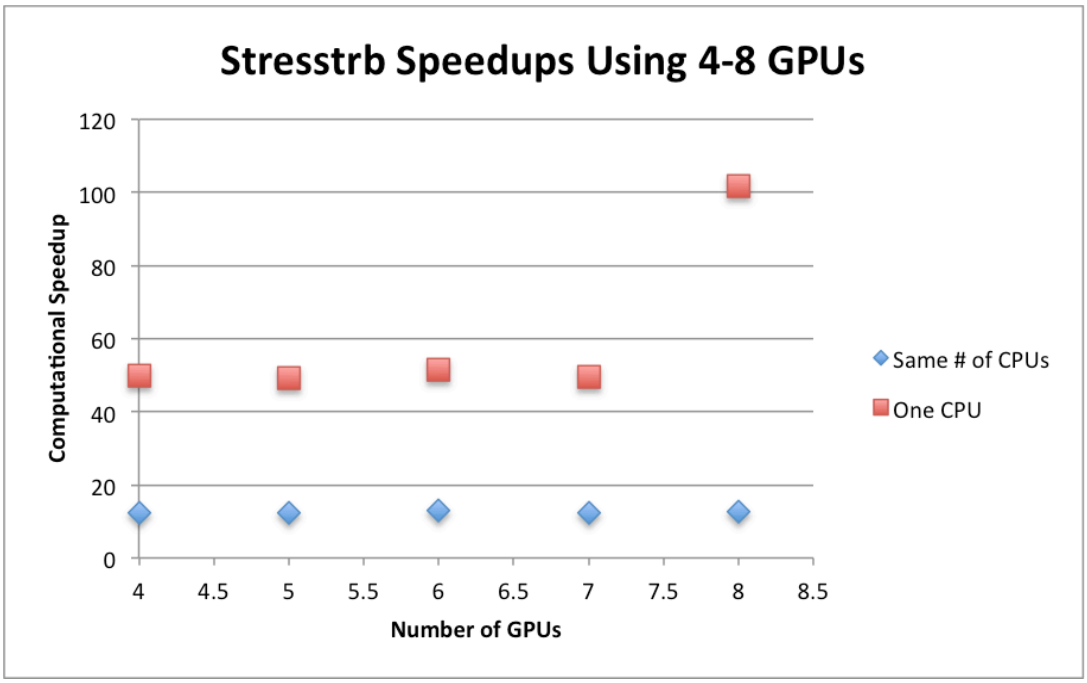

Figure 28: Multi-GPU Speedups for Smthtrb.

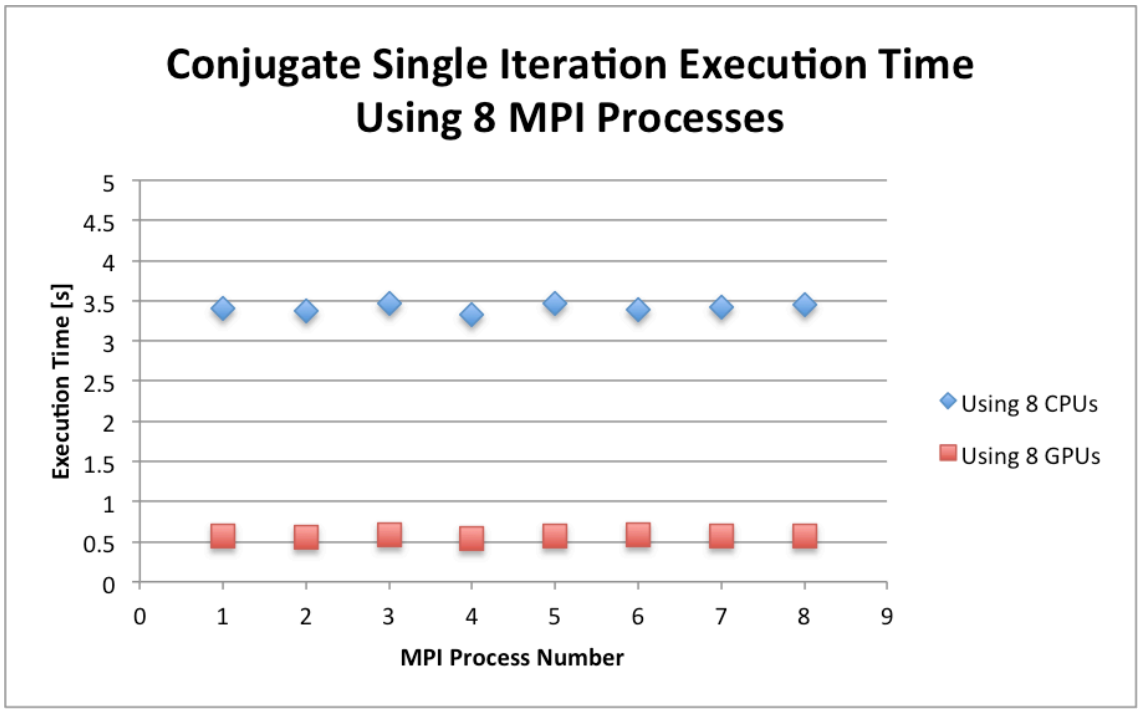

Figure 29: Execution Times for a Single Iteration of the Conjugate Cylinder Test Case.

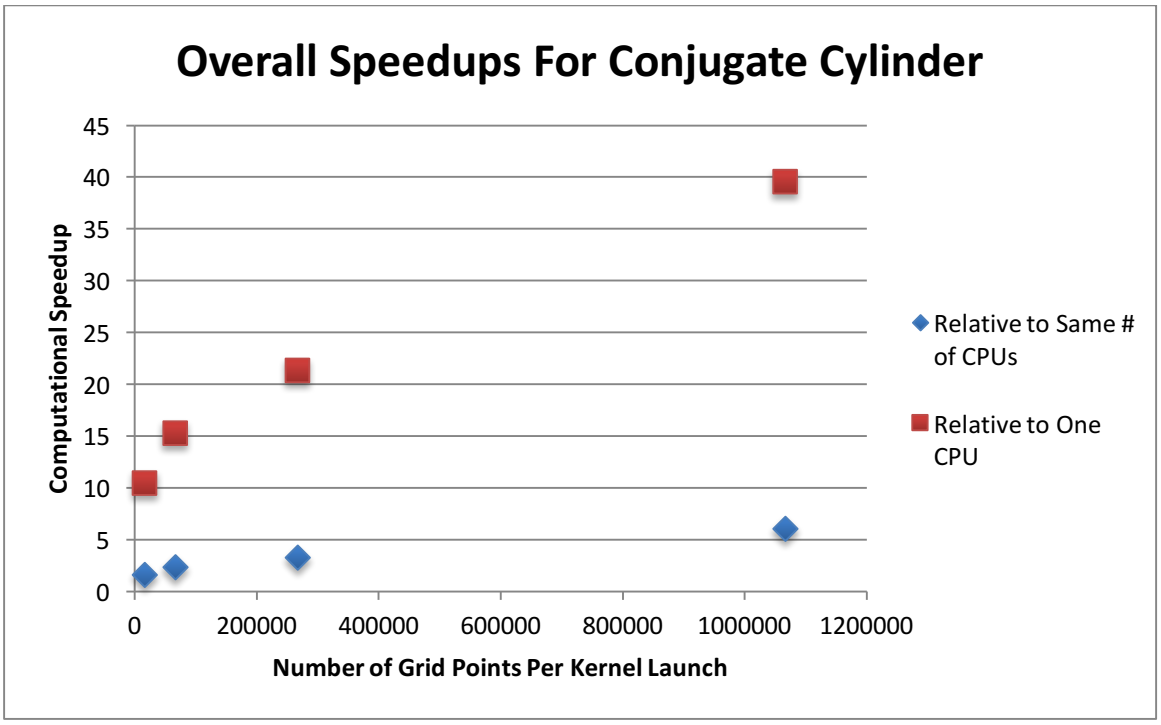

Figure 30: Overall MBFLO Speedups for Conjugate Cylinder Test Case.