In the Applied Aerosciences and CFD branch at Johnson Space Center, computational simulations are run that face many challenges. Two of which are the ability to customize software for specialized needs and the need to run simulations as fast as possible. There are many different tools that are used for running these simulations and each one has its own pros and cons. Once these simulations are run, there needs to be software capable of visualizing the results in an appealing manner. Some of this software is called open source, meaning that anyone can edit the source code to make modifications and distribute it to all other users in a future release. This is very useful, especially in this branch where many different tools are being used. File readers can be written to load any file format into a program, to ease the bridging from one tool to another. Programming such a reader requires knowledge of the file format that is being read as well as the equations necessary to obtain the derived values after loading. When running these CFD simulations, extremely large files are being loaded and having values being calculated. These simulations usually take a few hours to complete, even on the fastest machines. Graphics processing units (GPUs) are usually used to load the graphics for computers; however, in recent years, GPUs are being used for more generic applications because of the speed of these processors. Applications run on GPUs have been known to run up to forty times faster than they would on normal central processing units (CPUs). If these CFD programs are extended to run on GPUs, the amount of time they would require to complete would be much less. This would allow more simulations to be run in the same amount of time and possibly perform more complex computations.

Designing and Implementing an OVERFLOW Reader for ParaView and Comparing Performance between Central Processing Units and Graphical Processing Units

David M. Chawner[1] and Ray J. Gomez[2]
NASA Johnson Space Center, Houston, Texas, 77058

In the Applied Aerosciences and CFD branch at Johnson Space Center, computational simulations are run that face many challenges. Two of which are the ability to customize software for specialized needs and the need to run simulations as fast as possible. There are many different tools that are used for running these simulations and each one has its own pros and cons. Once these simulations are run, there needs to be software capable of visualizing the results in an appealing manner. Some of this software is called open source, meaning that anyone can edit the source code to make modifications and distribute it to all other users in a future release. This is very useful, especially in this branch where many different tools are being used. File readers can be written to load any file format into a program, to ease the bridging from one tool to another. Programming such a reader requires knowledge of the file format that is being read as well as the equations necessary to obtain the derived values after loading. When running these CFD simulations, extremely large files are being loaded and having values being calculated. These simulations usually take a few hours to complete, even on the fastest machines. Graphics processing units (GPUs) are usually used to load the graphics for computers; however, in recent years, GPUs are being used for more generic applications because of the speed of these processors. Applications run on GPUs have been known to run up to forty times faster than they would on normal central processing units (CPUs). If these CFD programs are extended to run on GPUs, the amount of time they would require to complete would be much less. This would allow more simulations to be run in the same amount of time and possibly perform more complex computations.

Nomenclature

ALU	=	Arithmetic Logic Unit
API	=	Application Programming Interface
CEV	=	Crew Exploration Vehicle
CFD	=	Computational Fluid Dynamics
CPU	=	Central Processing Unit
CUDA	=	Compute Unified Device Architecture
GPU	=	Graphics Processing Unit
JSC	=	Johnson Space Center
LIC	=	Line Integral Convolution
NASA	=	National Aeronautics and Space Administration
OVERFLOW	=	Overset Grid Flow Solver
PC	=	Personal Computer
SDK	=	Software Development Kit
USRP	=	Undergraduate Student Research Program

I. Introduction

In order to fully comprehend what forces are acting on a vehicle at any point in flight at any possible condition, it is common practice to compile data from 3 sources: flight tests, wind tunnel tests, and CFD (Computational Fluid Dynamics.) Each source has strengths and weaknesses which make it a necessary source of information, so each

[1] USRP Intern, Applied Aerosciences and CFD Branch, NASA Johnson Space Center, Texas A&M University.
[2] Aerospace Engineer, Applied Aerosciences and CFD Branch, NASA Johnson Space Center, Mail Stop – EG3.

needs to be used appropriately. Flight tests provide the most accurate data, but are the most expensive. Wind tunnel testing is about one order of magnitude less expensive, but has scale and interference issues. This leaves CFD as the cheapest alternative for data collection. Once the data is collected it is more useful to visualize it in a graphical form. One of the CFD solvers used at Johnson Space Center (JSC) is called the Overset Grid Flow Solver (OVERFLOW.)[1] OVERFLOW solves the Navier-Stokes equations at each grid point in order to determine the forces and moments at each point on the grid. OVERFLOW produces output files that contain this data in an easy-to-read format for a visualization tool.

There are many commercial visualization tools available that can read the CFD solutions, one very powerful visualization tool is an open source tool called ParaView. When a piece of software is classified as open source it means that the source code that the software was written in is openly available for public use. A person is able to modify the source code in any way that they want and is able to have their new code included in newer releases of the product. Since ParaView is open source, a tool can be written to read the OVERFLOW output file and display the data appropriately.

CFD solvers take a significant amount of time to provide the desired data. Many of these solvers are only utilizing the Central Processing Units (CPUs) which are common for programs. However, a new trend in programming is to write code that utilizes the computing power of the Graphical Processing Units (GPUs.) GPUs are known to be able to perform mathematical computations at a fraction of the time of the CPU. The GPU and the CPU are still required to communicate with each other because of the memory restrictions in the GPU. If CFD solvers and other codes were capable of being executed on GPUs and CPUs, the time it takes to receive output from the solver would be dramatically decreased, thus allowing for more complex computations to occur and for more cases to be run in a shorter amount of time.

II. Designing an OVERFLOW Reader for ParaView

OVERFLOW output files follow a format that simplifies the process of creating a reader, but there are many different variations of the format depending on the characteristics of the grid file that was used. Grid files contain every point that is to be evaluated. Some grids are grouped together, referred to as a single zone, while other grids have multiple zones. Grids can also be two or three dimensional, which changes the layout of the output file. The output files may contain different numbers of turbulence and chemistry variables as well. In order to get the simplest possible version of the code these different formats must be generalized to create a layout that all variations can apply and can be coded in a readable fashion.

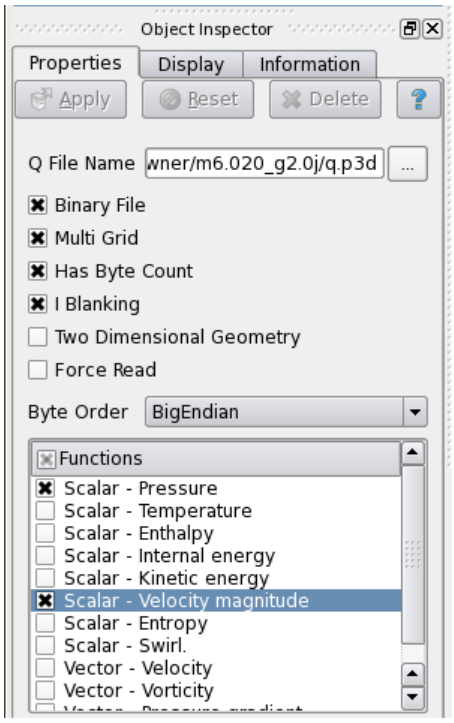


Figure 1. PLOT3D Reader in ParaView. *Shows the user interface for the PLOT3D reader in ParaView.*

A. Visually Determining File Characteristics

There are many signs in the OVERFLOW output file that can be used to determine whether a grid was single zoned or was a three dimensional grid. The files contain size information, or record markers, that are written by the machine itself, not necessarily the program. These record markers show how many bytes of data are contained in each of the sections in the file. For example, the output files will either start with how many zones are contained in each file, or with the dimensions of the only zone in the file. This information is either one or three integers, four or twelve bytes. If the size marker is four bytes, then it can be determined that the grid is a single zone; twelve bytes indicate a multiple zoned grid. After the number of zones has been determined there is a header of information that contains information pertaining to how many chemistry quantities there are for each point in each zone. If a file contains a single zone, the data at each point in that zone can be read, otherwise, each zone will have to be read one after another until all values have been read. The record markers are key pieces of information that reveal a lot of information about the file itself.

B. Programming the Reading Capabilities

There are two files that need to be read into ParaView in order for the visualization to begin, the grid file and the output file or solution file. The grid file reading capabilities were borrowed from another similar reader in ParaView that reads PLOT3D files and this portion of

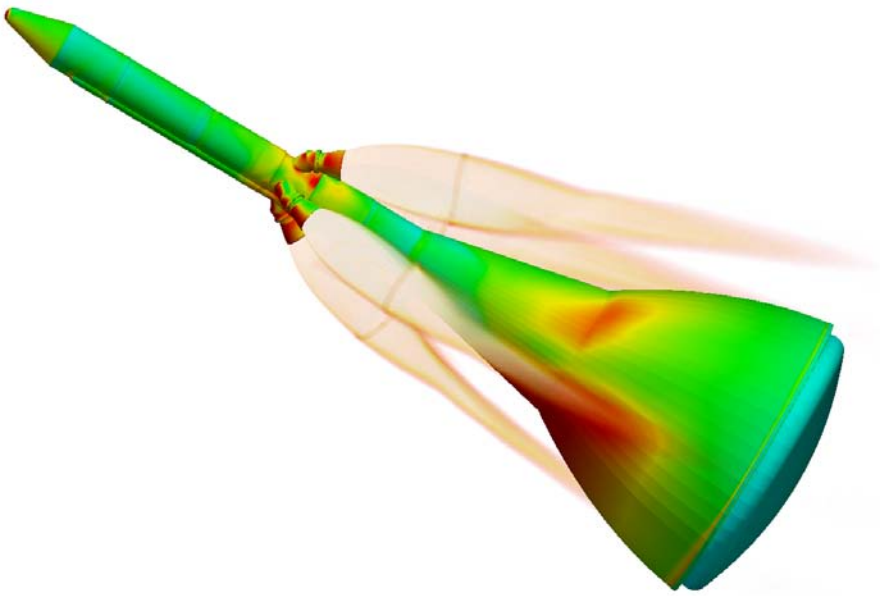


Figure 2. Volume Rendering of Plumes. *Shows an example of ParaView's volume rendering of plumes on the Orion CEV.*

the reader only required minor changes. Figure 1 shows the interface for the PLOT3D reader in ParaView. Note that the user must manually enter in whether the file is a multi grid file or uses iblanking. One of the goals for the new OVERFLOW reader is to automatically detect these qualities so that the user is not required to manually enter them. Since ParaView is open source as previously mentioned, borrowing the grid reading components from this reader is possible. However, there exist some major differences between PLOT3D and OVERFLOW solution files making it not as simple as copying an existing reader. For instance, additional chemistry and physics quantities are available in the OVERFLOW file. Once the solution file has been input by the user using ParaView's interface, the new reader can begin to read the files.

First, the reader determines whether the file is a grid file or a solution file and calls the appropriate C++ functions. If the file is a grid file, the endianness, or byte order, of the file is determined by reading in the first integer and seeing if it makes sense if the bytes were swapped. Since this first integer is either going to be a four or a twelve, it is easy to see if it makes sense "backwards." This first integer can also be used to determine if the file is a single zoned or a multiple zoned grid.

The reader then begins back at the beginning of the grid file, reads in the number of zones, calculates how many points there would be if the file was in three dimensions, determines the size of this information and then compares that to the actual record marker of the next block, which is how many points there actually are. If the predicted number and the actual size are equal, the file is in three dimensions; otherwise the file is in two dimensions.

After the correct number of dimensions is known, the grid file is then checked to see what the precision of the data is and whether or not the grid contains iblanking, or point visibility, information. The file can either be written with single or double precision,

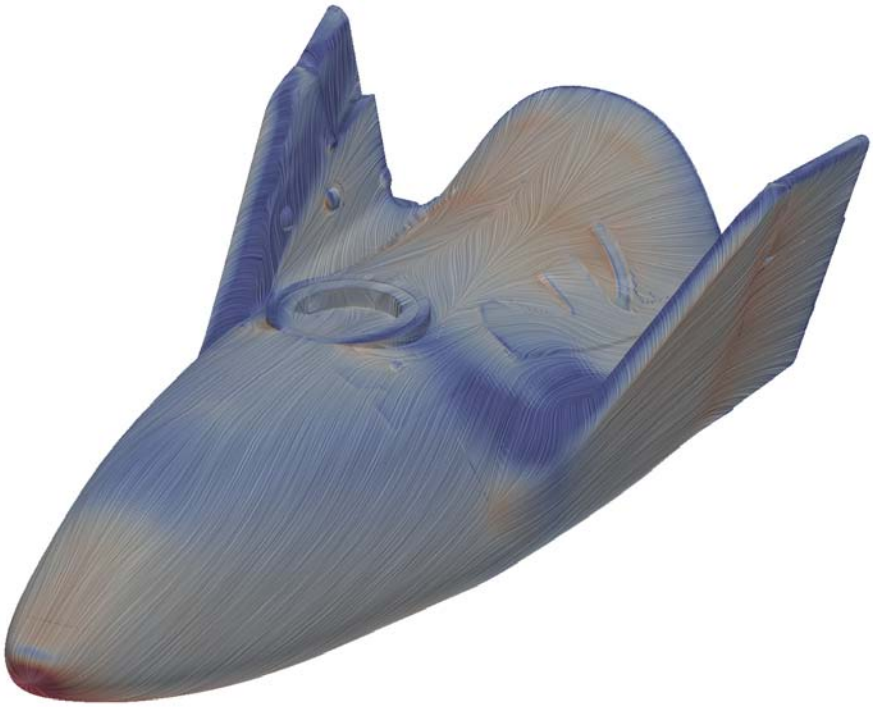


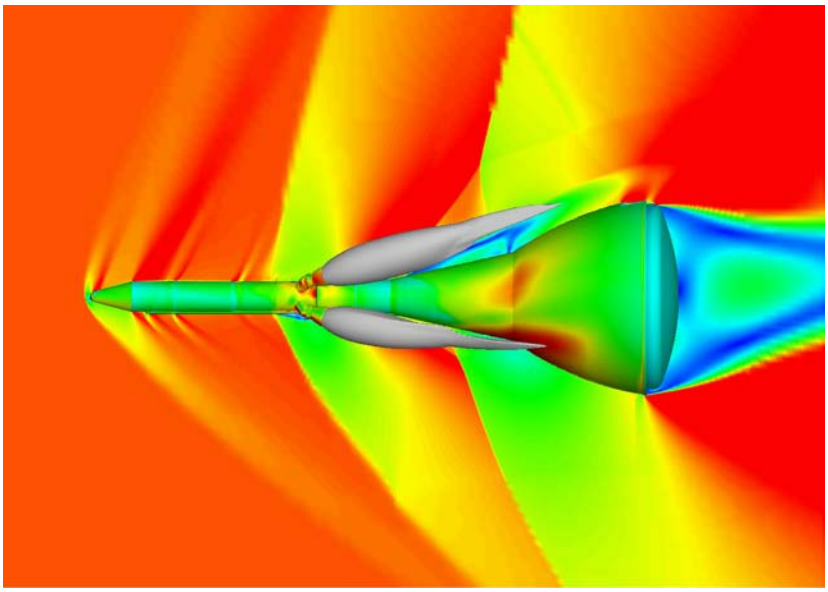
Figure 3. Surface LIC of X-38. *Shows an example of ParaView's Surface LIC on the body of an X-38. This LIC shows local flow near the surface as read by the new reader.*

which refers to the size of the numbers in the file; many problems require double precision. In order to determine the precision of the file, the reader starts at the beginning of the grid file determines how many zones there are and uses the dimension figured out previously to calculate the total number of points and then reads in the next record marker, similar to what is done in previous steps. There are four possibilities for the size of this next record marker, single precision with iblanking, double precision with iblanking, single precision with no iblanking and double precision with no iblanking. Single precision is calculated as the total number of points multiplied by the size of a float, four bytes while double precision is calculated as the total number of points multiplied by the size of a double, eight bytes. Iblanking is calculated by taking the total number of points multiplied by the size of an integer, also four bytes. This record marker is compared to all four combinations of these values to determine the precision of the file and if the file contains iblanking information.

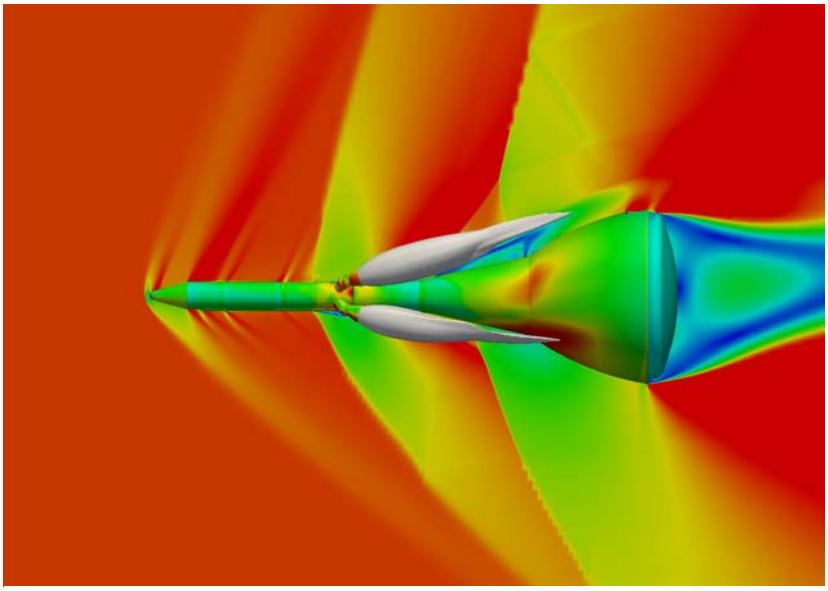
After gathering this information from the grid file, it is then used to read in the physics and chemistry calculations from the solution file. To handle the fact that each file can have a different number of physics and chemistry quantities, C++ objects are created only when that quantity needs to be stored. These objects hold all of the quantities at each point in the grid. After all of the quantities have been collected, other quantities such as Mach number, velocity and pressure coefficient are calculated at each point if the user desires them.

C. Using ParaView with the New Reader

To use the new reader a user would simply load the grid file and corresponding solution file and press the apply button. After the file is loaded a user could create contours, slices and many other unique visualization tools. A visualization of the body is created by doing a contour of velocity at a very low number. Color scales can be changed to show the various calculations that were done while reading the file. Also, bodies and slices can be colored by other quantities. A unique tool in ParaView that is enabled as a plugin is the Surface Line Integral Convolution (LIC.) It is difficult to graphically display vector quantities effectively. Drawing arrow heads on a body creates excess clutter.[2] This feature provides a unique way to visualize vector fields. As illustrated in Figure 3, the vector quantities are shown by lines along the surface of the body. This removes the need for arrow heads or other distracting information. In order to check the integrity and validity of the new reader an image that was created



(a)



(b)

Figure 4. Comparison between FieldView and ParaView. (a) *Shows the image created by loading the Orion CEV in FieldView.* (b) *Shows the image created by loading the Orion CEV in ParaView.*

using another visualization tool called FieldView was compared to the image created by ParaView.

D. Comparing Different Readers

A grid file and the corresponding solution file of the Orion Crew Exploration Vehicle (CEV) were loaded into ParaView. The shape of the vehicle was created by creating a contour at velocity magnitude equal to 0.0001. This contour is then colored by pressure coefficient and the range of the contour is changed to -1 to 1. Next, a slice is taken that is normal to the y-axis. This slice is colored by Mach number on a range from 0 to 1.4. Finally, the plumes are created by creating another contour of the second species density at a value of 0.95. The plumes are colored by the standard gray solid color. Figure 4 shows both images created using FieldView and ParaView. Notice that the shape of the blue region behind the CEV is smoother in the ParaView image than it is in the FieldView image. However, notice that the body is smoother in the FieldView image. ParaView also created an image that appears darker, but that can be changed by modifying some display settings in the application. Overall, it appears that the new OVERFLOW reader for ParaView worked very well on this case; the images compare very well and there are no major discrepancies on the ParaView rendered image.

E. Assuring Reader Correctness

To fully test that the reader works for a variety of cases and files, many more files were tested. Doing this ensures that the reader is able to correctly load any type of file, with any of the key characteristics that were described earlier. The files were loaded into ParaView and put through the same rigorous testing of matching previously created images. Other test files that were loaded include a complete Shuttle launch vehicle, an entry Orbiter, a parachute and an X-38 grid. These files were also loaded with their respective solution files to ensure proper calculation of physics and chemistry quantities. Every time a change in the reader code was made, all previous cases were run in order to ensure that the new changes didn't affect any previous fixes. This practice is commonly referred to as regression testing. In the end, all files that were loaded were successful, and therefore, the new reader can be considered a success.

III. Comparing Performance between GPUs and CPUs

Most programs today do not fully take advantage of the processing power that is available. Programs do not need to be executed on high powered machines in order to experience large increases in performance. The same resource that a user's personal computer (PC) uses to display the images on the screen can be used to see improved performance in execution times. There exist many useful tools and packages that a programmer can download and install on their PC to link standard programming languages such as C, Python and Java to libraries of functions that ease the transition from executing solely on the Central Processing Unit (CPU) to executing on both the CPU and Graphics Processing Unit (GPU.)

A. What are GPUs and Why Use Them?

GPUs are mainly used in PCs today as the computing resource that renders windows and images on the computer monitor. Rendering and displaying images requires a lot of computing power to quickly calculate what color each pixel on the screen should be in order to correctly display the image as a whole. Not only do these calculations have to be quick, they also must be accurate. All of the calculations on the CPU or GPU occur in Arithmetic Logic Units (ALUs.) If more ALUs are available, the computations will be done in a quicker amount of time. However, the GPU ALUs are slower in performance to the CPU ALUs. This is not seen by the user due to the fact that the GPU is performing many more calculations simultaneously than the CPU is. Figure 5 shows that

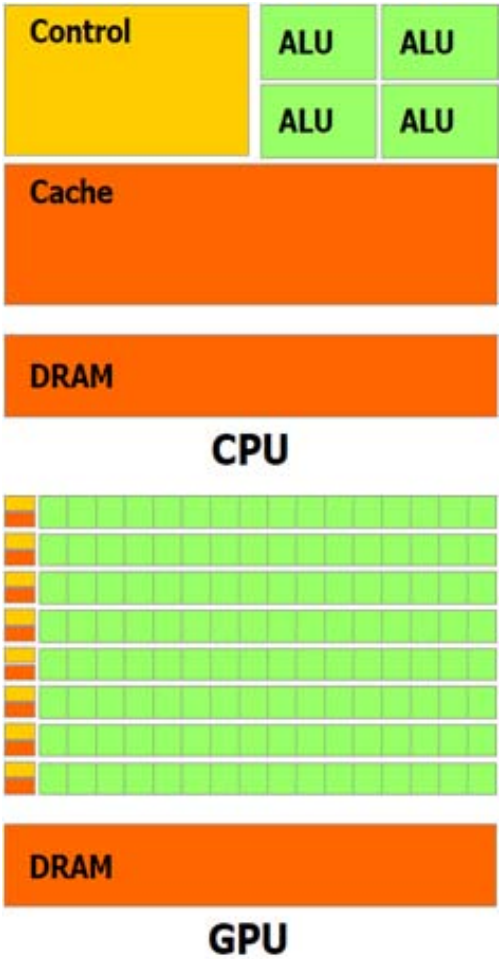


Figure 5. Typical CPU and GPU Chip Configurations. *Shows the ratio of ALUs (green boxes) to memory (dark orange boxes) on the CPU and GPU.*

the GPU has many more ALUs than the CPU, but the CPU has more memory available.[3] Each group of ALUs in the GPU has a small amount of shared cache or memory.

In a CFD solver, the same calculations are solved at every point in the grid hundreds or thousands of times to receive a single solution. As mentioned previously, running a CFD solver takes a long amount of time, even on the fastest machines, due to the large numbers of calculations that need to be solved. When doing repeated calculations such as multiplying matrices, these computations take a long amount of time. This is the type of situation that GPU programming is suited for. GPU programming will show a large increase in performance in cases where the same calculations need to be performed repeatedly. With more ALUs available, more calculations are able to be done simultaneously. If more calculations are done in a shorter amount of time, the program will take less time to run the same amount of steps.

A large workstation with many CPUs capable of running hundreds of processes simultaneously is not required for a program to experience a performance increase, neither is a PC with multiple cores on which the programmer must have knowledge and experience of manually splitting up the program in order to take advantage of these resources. Many companies such as NVIDIA who manufacture graphics cards are making the transition for programmers to utilize the resources available in a typical PC. This study will measure the performance results of two benchmarks running on both CPUs and GPUs.

B. What is CUDA?

NVIDIA has created a tool called the Compute Unified Device Architecture (CUDA) that makes the transition to GPU programming simpler. This tool was selected for the performance comparisons in this study because it is a tool that works only on specific NVIDIA graphics cards, cards that the PC that these tests were performed on used. CUDA is a collection of instructions that a programmer can use to directly talk to the GPU in order to execute tasks on the GPU. CUDA provides programmers with both a high level and low level Application Programming Interface (API) that can be used to communicate with the GPU. An API allows and facilitates interaction between the programmer and the computer. The CUDA high level API allows programmers to stay far away from many of the details of the GPU while still achieving the desired performance increase. However, the low level API allows the programmer to manipulate the GPU in order to receive an additional performance increase.

The high level API was used to program the tests used for the comparison. The high level API provides programmers with functions that look very similar to standard C functions. These functions perform some low level tasks such as figuring out how many processes are capable of being run, scheduling them and actually performing the computations. In order to program the GPU, all that an inexperienced programmer would have to learn is a few basic CUDA functions, how they are used with the standard C programming language. Integrating all of these components provides the user with the desired performance increase. However, not all aspects of the C language are allowed with the CUDA API. For instance, the use of recursion is not allowed and there is no support for function pointers.

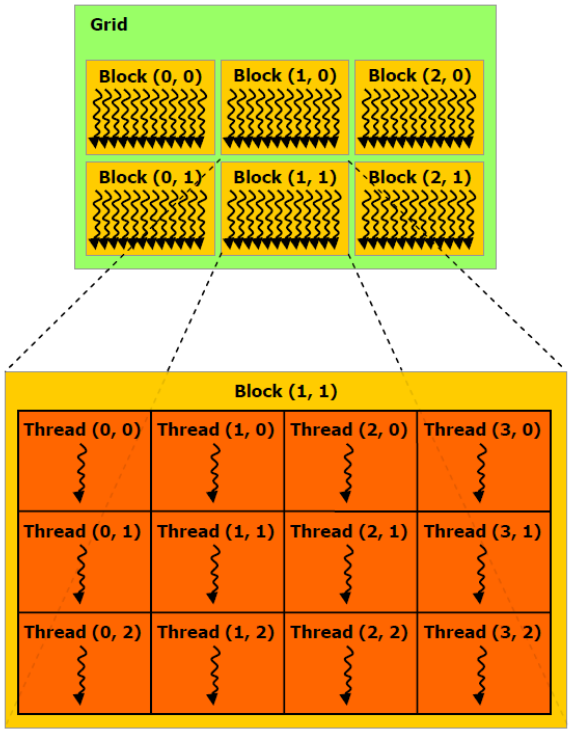


Figure 6. Thread Diagram. *Shows a typical way that a GPU program is divided into blocks and threads.*

A typical CUDA program would begin by loading the data from the CPU to the GPU. As shown in Figure 5 by the dark orange boxes, there is a larger amount of memory on the CPU. The size of the available memory depends on the manufacturer of the CPU and GPU and if there are other processes running that are using some of this memory. After the data is loaded from the CPU to the GPU, the CPU tells the GPU what is to be executed. Once that is done, the GPU begins to execute those instructions in parallel. Once all instructions have been performed, the data is then copied back to the CPU.

C. Performance Limitations

While the performance of executing a program on a GPU should be improved from running it on a CPU, there are some limitations to the amount of improvement. First,

since the data is being transferred from CPU to GPU and vice versa the speed that the data can be transferred is a limiting factor on performance. The data is transferred through a subsystem called a bus. The maximum speed of the data travelling through this bus depends on the manufacturer. Also, if there is other data travelling or waiting to be sent through the bus, this could cause slowdowns in performance.

Another limiting factor to performance is that not all of the code should be executed on the GPU. As mentioned previously, the GPU is very efficient at performing similar computations simultaneously. The GPU is not very efficient at doing other types of computations. If the GPU is performing a task that it is not designed to do, it will execute those calculations at an unimpressive speed. This will create a limitation in performance. However, if the task was performed on the CPU, this will also create a slowdown, but it will not seem as dramatic. It is good practice to only use the GPU to perform tasks that it will excel at, because performing other tasks will cause significant slowdowns.

Another limiting factor to performance is properly splitting the original program into blocks and threads. When executing tasks on the GPU, the large task is split into smaller tasks, called blocks, through either API. This allows for the maximum usage of the GPU core and leaves no wasted space. Each thread would then execute the appropriate calculations, get grouped back together into blocks and then the blocks will get grouped back together back to the original grid. This technique is depicted in Figure 6. A grid is divided into blocks and each block is divided into threads. Each thread executes the instructions on a different collection of data.

Also, if the program contains a control structure such as an if statement or for loop, this could decrease performance. The uncertainty provided by these statements cause threads to be manipulated in ways that they were not supposed to be, creating a potential for performance decreases.

Finally, it is important for the programmer to remember that some GPUs do not have ALUs that are capable of performing double precision calculations. Precision will be lost if values that are too large are being computed. Depending on the make and model of the GPU, this might not be the case, and it might support double precision calculations. If the application a programmer is running requires double precision capabilities, a newer, more expensive GPU might be required.

When programming using the CUDA architecture on a GPU, a programmer must be very careful to try and not violate any of the practices, otherwise performance will be compromised.

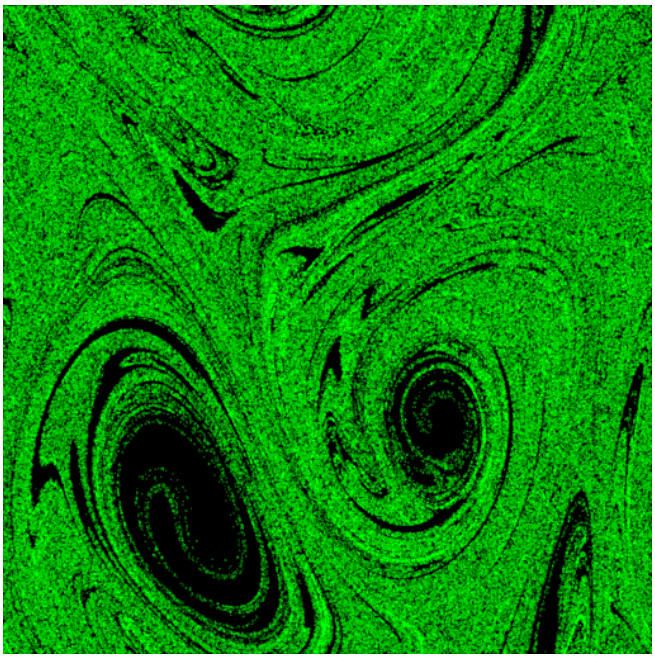


Figure 7. CUDA Fluids. *A screenshot of the fluids simulation from the CUDA SDK.*

D. CUDA SDK and Additional Materials

When downloading the NVIDIA CUDA toolkit it is also recommended to also download the CUDA Software Development Kit (SDK.) The SDK provides many examples of the various functions in the API. These examples vary from getting GPU information to image rendering and post processing. Figure 7 shows a screenshot of a fluids example from the CUDA SDK. There are also additional reference materials on the NVIDIA CUDA website such as Getting Started Guides, Best Practices Guides, and API references. These are all valuable tools in order for beginners to learn the languages and restrictions as well as see detailed examples of how the code should be written.

A recent study performed by Intel claims that the GPU will not perform 100 times better than the CPU on many applications.[4] In this article, Intel states that after optimizing the code, the average performance gap between GPU and CPU is only 2.5 times faster on average. These cases were run on CPUs and GPUs that many people have already installed in their PCs, however, it is unclear whether the optimizations were properly done. The study that is described below is another attempt at gathering performance information between the CPU and GPU.

E. Comparing GPU and CPU Execution Times

The following comparisons were made between GPU and CPU versions of similar codes executing on the same PC. It should be noted that the CPU version of the code only executed on only 1 core, while the GPU version had the potential of executing using 64 cores, 8 processors with 8 cores each. Steps were taken to ensure that the codes were identical, except for the necessary CPU versus GPU distinctions such as data transfer and variable loading. The GPU versions of the code came from the CUDA SDK and were slightly modified to perform the necessary actions as discussed below. When timing the functions, special care was taken to make sure that only the pertinent sections of code were being timed. CPU versions of the code were written in standard C++.

1. Matrix Multiplication

This benchmark performed matrix multiplication of specific sizes containing random numbers. The computations that were performed yielded results to the equation A x B = C, where A is of scale 1 x 2, B is of scale 1 x 1 and C is of scale 1 x 2. To keep consistent with the practices mentioned above, the tests were run at multiples of 80, meaning that the actual size of the matrices is the scale factor multiplied by 80, i.e. at size factor 1, matrix A was of size 80 x 160, B was of size 80 x 80, and C was of size 80 x 160.

The GPU version of the matrix multiplication code came directly from the SDK and required very few modifications. The modifications were to change the output formatting to allow for a script to run the test cases automatically and gather the output. The CPU version of this matrix multiplication code performed the same variable initializations and computations as the GPU code. The test was run a total of five times at each size factor and the results were averaged together. The only portion of the code in each version that was being timed was the multiplication portion. Therefore, the results are only showing the ratio of computing the multiplication between CPU and GPU.

To create a ratio plot as shown in

Figure 8. Matrix Multiplication Benchmark Results. *Shows the ratio of execution times by dividing CPU execution time by GPU execution time.*

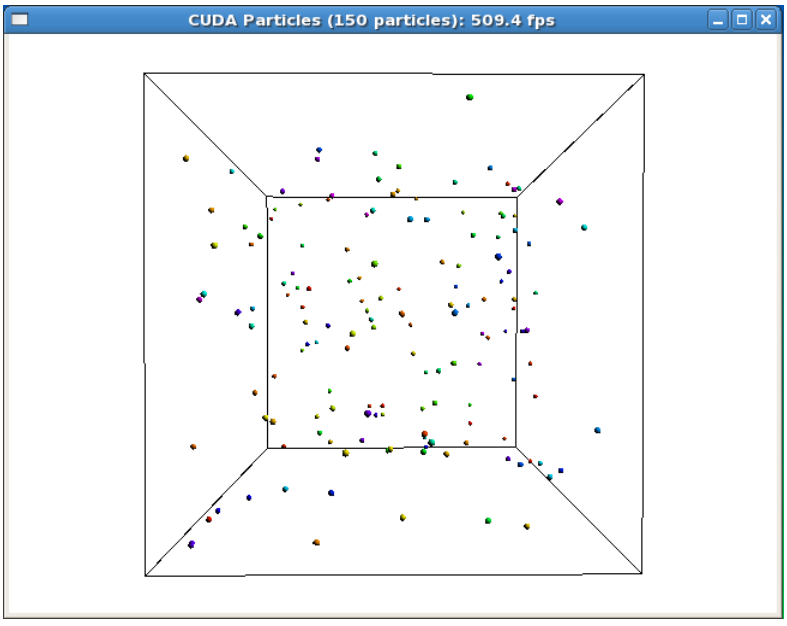


Figure 9. Particles in a Box Benchmark. *Shows a screen capture of the particles in a box simulation executing on the GPU.*

Figure 8, the average execution time for the CPU case was divided by the average time for the GPU case, so that the results were a number larger than 1. This ratio plot shows how many times faster the GPU version of the code is to the CPU code.

The ratio appears to be showing a linear trend; as the size of the matrices increase, the ratio of CPU to GPU execution times also increases. While this is a rather simple example to be comparing, it shows that the potential for performance increases are possible even in trivial tests such as matrix multiplication.

2. Particles in a Box

This code simulates particles acting freely in a box, able to collide with other particles and the box walls as shown in Figure 9. The code works by first creating a three dimensional grid of a specified size in the code. Next, initial positions and velocities are calculated for each particle. For each iteration specified by the user, new positions and velocities are calculated based on gravity and other forces existing in this box. After all particles have these updated quantities, the program figures out which cell the particle is in and creates a table of particles and the cell they are located based on the center of the particle. This table is then sorted according to cell number. For every particle in a given cell number, a collision algorithm is used to see if these particles are colliding and it applies the appropriate calculations. Since particles can be in more than one cell at a time, the collision algorithm is applied to all of the neighboring cells as well. All of these steps are applied each iteration.

The GPU version of this code was taken from the CUDA SDK and was modified in order to remove gravity and allow for perfectly inelastic collisions. The original code also allowed a graphical representation of the box to be displayed; this was removed to make the CPU and GPU versions more similar. Also removed from the GPU version were some printing statements so that the code can be easily executed from a script and output gathered in a single location. The CPU version performs the same steps as the GPU code, namely no graphical display, no fancy output and same computations in the same order.

This benchmark ran several tests, each with a different number of particles at 18 different numbers of iterations. Each combination of number of particles and iterations was run three times, and an average was taken. That average execution time was divided by the number of iterations to calculate a time per iteration value. This was done on both the CPU and GPU versions. A ratio was also calculated by dividing the average execution time for the CPU and dividing it by the average execution time for the GPU at each combination. The results were then grouped by number of particles and plotted. The results of this benchmark are shown in Figure 10.

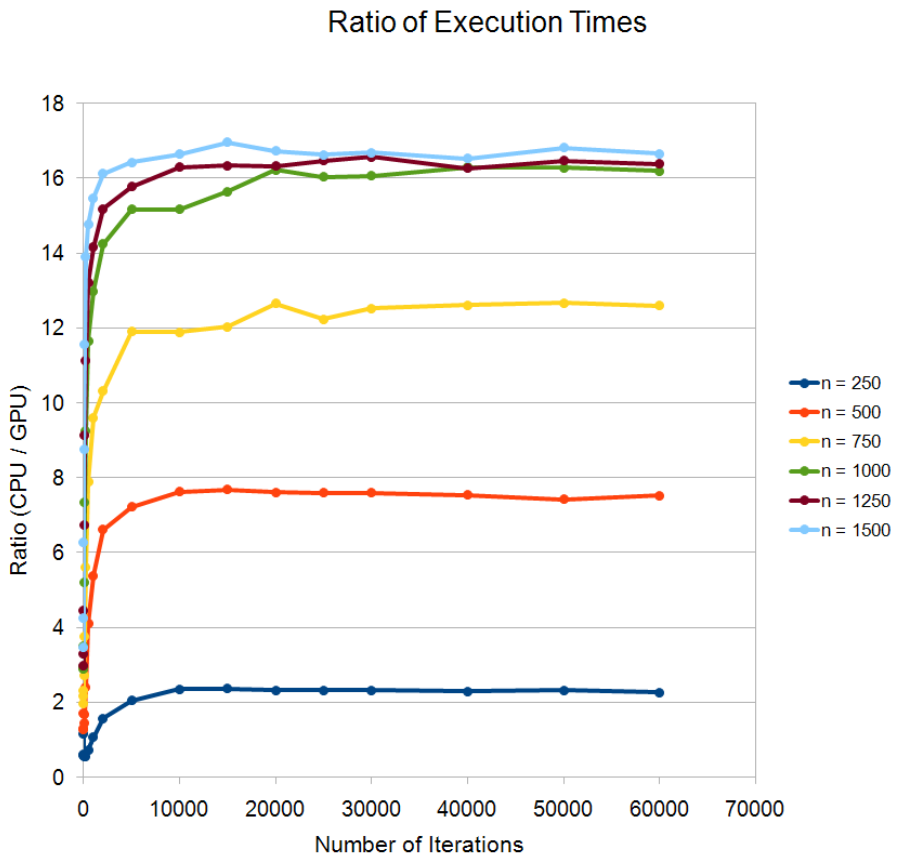


Figure 10. Particles in a Box Benchmark Results. *Shows the ratio of execution times by dividing CPU execution time by GPU execution time. Each line corresponds to a different number of particles that were run in this benchmark.*

As the number of particles is increasing, the ratio of CPU execution time to GPU execution time is increasing. However, it appears that around 1000 particles, the ratio seems to be hitting a maximum value of approximately 16. This means that the GPU code is 16 times as fast as the CPU code. It is unclear whether this value is the absolute maximum value without doing more testing.

However, as the number of particles increases, the execution time also increases, and it becomes more time consuming to run these simulations. But it can be implied that as the number of particles continues to increase, the ratio of execution times should be no less than 16. This speedup may only occur to a fraction of the entire code, meaning that the entire program will not be 16 times faster on GPU than on CPU.

IV. Conclusion

Running a CFD simulation provides the user with a file of numerical data. This data must be read into a visualization tool in order to graphically understand what the data is. Adding a reader to an open source tool such as ParaView can improve the visualization of that numerical data. Open source software simplifies adding new capabilities to existing software. When the output file that is to be read follows a specific format it makes it easier to create such a reader. Verifying that the reader works correctly requires many different test cases and a lot of patience. Any new change to the reader during testing requires that all previously tested cases be retested. While this procedure is time consuming it is essential to assuring that the reader will work for multiple cases. It is also a good idea to see if an image that was created using another visualization tool can be duplicated using the new tool with the new reader. If CFD simulations were able to run using both CPUs and GPUs they would require less time to be completed. As shown in this study, for matrix multiplication codes, the GPU code can run at least 150 times faster than the CPU code. This is quite a tremendous improvement that increases as the size of the matrices increase. The particles in a box benchmark that was run represented a more realistic application of what could be achieved using GPU processing power. As the number of particles increased, the ratio of CPU execution time to GPU execution time also increased. The ratios that were gathered from this study indicate that the potential for improvements in code execution time exist. Remembering that the CPU code could be optimized to enable multiple processors on a PC, this code only utilized one of eight available processors, could decrease the ratio. There are GPU workstations available that contain multiple powerful graphics cards, and the CUDA toolkit provides programmers with functions to link multiple GPUs together. These benchmarks could be run on higher quality, faster, more powerful machines and a new set of ratios could be received. This study showed what a typical programmer might see, using the PC readily available to them.

Acknowledgments

We would like to thank Jay LeBeau for his help with organizing and assisting in these tasks by providing computing resources.

References

[1]Nichols, R.H., Buning, P.G., “Users Manual for OVERFLOW 2.1,” University of Alabama and NASA Langley Research Center, 2008.

[2]Forssell, L., “Visualizing Flow Over Parametric Surfaces Using Line Integral Convolution,” *Proc. IEEE Visualization 1994*, Washington, DC, 1994.

[3]NVIDIA CUDA Programming Guide version 3.1.1 (7/21/2010) http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf [cited 3 Aug 2010].

[4]Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., et al. “Debunking the 100X GPU vs. GPU myth: an evaluation of throughput computing on CPU and GPU,” *ACM SIGARCH Computer Architecture News*, Vol. 38, No. 3, June 2010, pp. 451-460.