CSE 6163 - Project 3: Parallelizing Fluid Simulations with CUDA

Jacob Kutch

April 27, 2023

Introduction

This project involves the parallelization of a serial computational fluid dynamics solver used to solve the Navier-Stokes equations using numerical methods for turbulent flows. In particular, it solves a benchmark problem that simulates the breakup and dissipation of a large eddy structure. The benchmark, the Taylor-Green vortex problem, is commonly used to evaluate the suitability of fluid solvers in the simulation of the dynamic details of turbulent flows.

The code involves computation of total kinetic energy of fluid in all cells in the mesh by Runge-Kutta numerical integration at various time steps. These time steps must be minimized over each cell considering inviscid and viscous terms of the problem to find a stable time step for each iteration of the simulation. This is the starting point of the time integration. Due to the nature of considering the mesh in a grid form, while actually a continuous surface, careful attention is given to enforce periodic boundary conditions on the mesh to correct for "ghost grid values." Error is approximated by the sum of momentum fluxes of each cell's faces. These residuals are intended to be minimized by minimizing total flux through cells and their immediate neighbors. A weighted sum procedure is used to update solutions by using previous solutions and the computed residuals. This iteration repeats until the entire time interval has been traversed.

Parallelization attempts leverage the Compute-Unified Device Architecture (CUDA) programming model developed by Nvidia, wherein the model is updated to use the hardware structure of the general purpose graphical processing unit (GPGPU) architecture to achieve massive parallelism. The control flow is organized in a heterogeneous coprocessor model with the CPU acting as the Host, executing the serial portions of the code and launching CUDA kernel functions on the GPGPU, which acts as the Device where the majority of intensive computation occurs. GPU threads are grouped into thread blocks where threads in different thread blocks generally run without synchronization, while threads within the same thread block generally run with synchronization since they are assigned to the same streaming multiprocessor. Ultimately, the goal of this project is to utilize this scheme to achieve large-scale parallelism on a program that would otherwise scale incredibly poorly on traditional serial processors.

Methods

The original incarnation of the C++ project was provided by Dr. Edward Luke of Mississippi State University. All utilities were provided save for a parallelized driver file. The original driver file, ***fluid.cu***, was mostly implemented sequentially but written so as to be easily parallelized with CUDA. Two functions, `setInitialConditions()` and `integrateKineticEnergy()` were already parallelized with CUDA to be used as an example. Additionally, the latter function had an associated Device kernel function, `sumKernel()` created to emulate a sum reduction operation. The code was tested on Mississippi State University's High Performance Computing Center's Scout cluster of the Titan supercomputer, using CUDA v11.2.1 and slurm batch scheduling. All results are estimated from running on the cluster.

fluid.cu was updated incrementally, so that each function was implemented as a new kernel function and tested alongside the serial function implementations, as advised by Dr. Luke. This initially necessitated copying data between the CPU and GPU with some provided wrapper functions for `cudaMemcpy()`. Any data

located on the CPU and handled by the serial versions of functions had to be copied onto the GPU, accessed or modified with the updated kernel functions, and copied back to the CPU to be used by other serial functions.

The first function encountered in the solver was `setInitialConditions()`, which was the first to be updated as a kernel function. It initializes each variable for each cell using provided parameters of the canonical Taylor-Green vortex problem. This contains a doubly-nested canonical *for* loop with no dependencies in loop conditions that needed to be amended. This kernel is launched only once with n_i thread blocks and n_k threads per block, outside the main time-stepping loop of the simulator. Because of the lack of loop dependencies, it was easily parallelized by replacing loop variable i with n_i thread blocks, each with n_k threads indexed by the replaced loop variable k. With the remaining loop variable j, it could then be logically viewed as a process distributed across several thread blocks each executing a loop in terms of j, where n_k threads within each thread block operated on adjacent memory locations for each iteration j.

This same task distribution scheme was used in the updated versions of the functions `zeroResidual()`, `computeStableTimestep()`, `integrateKineticEnergy()`, and `weightedSum3()`. This exact strategy worked in the updated in `weightedSum3()`, which computes the weighted sum over each variable to be used in the next step of Runge-Kutta time integration, with no other changes required, since the loop iterations were exactly the same with no data dependencies. For `zeroResidual()`, which resets residual vectors back to 0 for the next integration step, some care had to be taken to zero out the boundary cells at indices $(-1, j, -1)$, $(-1, j, n_k)$, $(n_i, j, -1)$, and (n_i, j, n_k) for $j = -1, 0, \cdots, n_j$. These indices were handled in a separate loop in a serial manner. The remaining part was handled in parallel using the aforementioned task distribution scheme, with loop variable $j = -1, 0, \cdots, n_j$.

`copyPeriodic()`, which copies periodic boundary conditions at the boundary of the faces given by the yz-trace, xy-trace, and xz-trace, was handled with a similar task distribution. Its kernel function was also launched with n_i thread blocks and n_k threads per block, wherein i was used as the index for each thread block and k was used as the index for each thread within a block. However, this used the assumption that $n_i = n_j = n_k$, since this is what the code is initialized with in every test case. The nested loop for copying the i-periodic faces that previously looped over j and k (in that order) was updated to run in parallel without looping, but simply replaced all usage of j with the new block index i since $n_i = n_j$ in all test cases, meaning the iterations were identical. The nested loop for copying the j-periodic faces that previously looped over i and k (in that order) was updated to run in parallel with minimal changes beyond eliminating the loop. The nested loop for copying the k-periodic faces that previously looped over i and j (in that order) was updated to run in parallel without looping, but replaced the inner loop's usages of j with thread index k, which relied on the assumption that $n_j = n_k$ similar to the assumption with the i-periodic faces. To eliminate the dependence on this assumption, `copyPeriodic()` could be split into 3 separate kernel functions, with the function for the i-periodic faces being launched with n_j blocks and n_k threads, the j-periodic faces being launched with n_i blocks and n_k threads, and the k-periodic faces being launched with n_i blocks and n_j threads. This may be implemented in the future.

Very similar to the technique used in `integrateKineticEnergy()`, the function `computeStableTimestep()` was also implemented using a strategy common for parallel reductions in CUDA: a new temporary array is passed to the function, where the minimum values found throughout the execution of the loop in each thread block i and thread k are saved to this temporary array, indexed as $i * n_k + k$. Upon returning, the scratch array is filled with local minimums within each block and use of a new kernel function `minKernel()` is used to compute the minimum within each thread block in parallel, leaving the final global minimization over each thread block to be completed in serial on the CPU.

As was suggested as a possible implementation within `copyPeriodic()`, one of the most computationally expensive functions named `computeResidual()` was broken into 3 separate kernel functions, each covering a loop that iterated over different physical dimensions of the mesh. The original functions computed the residue of the computed rate of change in each direction for the pressure variable p and the three components of the velocity vector denoted (u, v, w). This proved to be the most difficult to parallelize because of non-uniform looping conditions depending on the associated direction in each loop. Each nested loop block contained one loop variable that experienced an extra iteration determined by that loop block's associated direction. This was not easily

handled by launching the kernel with only a constant number of thread blocks and threads per block. Instead, `computeResidual_x_kernel()` was created to iterate over the x dimension, `computeResidual_y_kernel()` was created to iterate over the y dimension, and `computeResidual_z_kernel()` was created to iterate over the z dimension.

The loop variable associated with a particular direction remained as a loop variable while the remaining loop variables in the block were made into CUDA block and thread indices. The exception to this strategy was `computeResidual_z_kernel()`, which normally contained an extra iteration in the k loop, but since threads must operate on adjacent memory at the lowest level of a nested loop to achieve speedup from parallelism, k remained the thread index. Instead, i was used as the top level loop variable, j became the block index, and k was used as the thread index, but the kernel was launched with $n_k + 1$ threads. Given that the values of these parameters must be multiples of 32, and given that CUDA will simply allocate the next multiple of 32 for the number of threads in such a scenario, this implementation always resulted in 31 inactive threads, reducing thread block efficiency. An alternative strategy was to launch the kernel with n_k threads and to simply repeat the iteration, using n_k (the (n_k+1)th iteration) as a constant within the code just as the index k was formerly used for the final iteration previously seen in serial. Finally, to ensure thread synchronization between memory writes, the additions and subtractions which always happened at different indices based on operation, were separated with a call to `__syncthreads()` between them to prevent race conditions.

As all functions were parallelized incrementally, more care had to be taken with appropriate updates between data copied to and from the device in the initial stages. As more functions were implemented in parallel, there were much fewer calls to `cudaMemcpy()` and the resulting time per cell per time step steadily went down. Eventually, with all functions implemented as CUDA kernel functions, computation was done entirely on the Device. In the end, the only required copying from the Device back to the Host was done once by `computeStableTimestep()` to determine the appropriate time step length and done each loop by `integrateKineticEnergy()` to update the new kinetic energy value.

Analysis

Using the GPU as in this project allows massive parallelism and high performance per node, so that a single computer using a GPU may outperform small CPU-based clusters for some tasks [2]. An example GPU architecture with its simple core structure allowing for high volume data and control between cores can be seen in Figure 1.

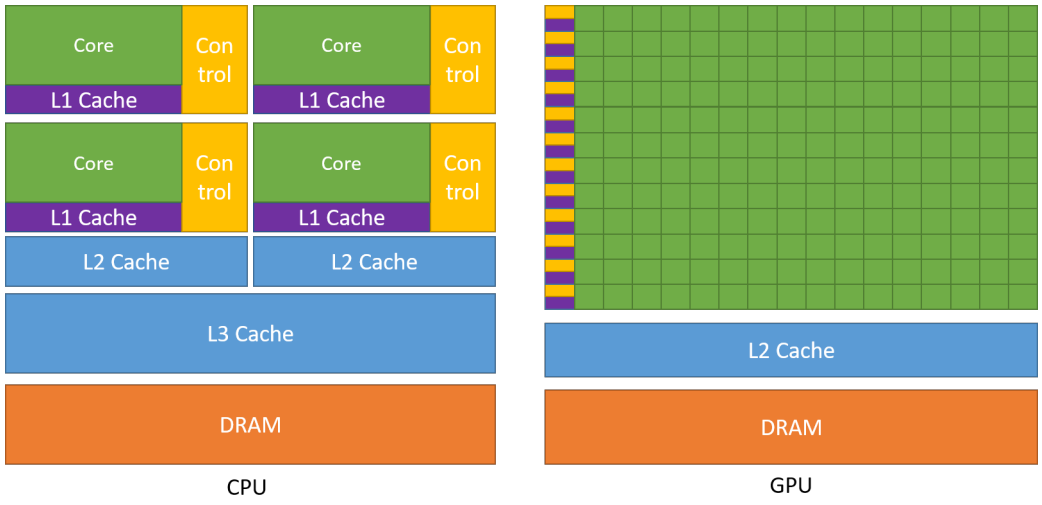

Figure 1: CPU vs GPU Architecture

After an initial endeavor to develop an analysis based on Amdahl's Law under the CUDA programming model, I saw that the analysis was not as simple as when considering the situation on a single device. Such

complications came from the following considerations:

- The GPU has many more cores than the CPU.
- The CPU has low latency and the GPU has high throughput.
- The CPU minimizes latency while GPU hides latency by overlapping communication.
- Comparing speeds is not straightforward since the GPU clock (Mhz) is slower than a CPU (Ghz) clock.
- Bandwidth is not shared or limited between them but by the ratio of GPU cost to CPU cost.
- GPU parallelization is more about huge data parallelism over the same operations (stream computing), like vectorization on the CPU

Furthermore, an accurate analysis depended much more heavily on the specifications of the Scout cluster. Although many unknowns remained, the following information was given on the Scout cluster:

- debug slurm partition uses 3 nodes (48 processors)
- each node uses 16 cores (2x Xeon E5-2680 2.7 GHz (turbo, 3.5 GHz) 8 core Sandy Bridge Processors)
- 32 GB memory (8x 4GB DDR3-1600 MHz)
- FDR (56 Gb/s) Infiniband network
- 8 nodes with 1 Nvidia K20 GPU

This analysis instead focuses more on the specifications gleaned from the fact that 1 Nvidia K20 GPU was used for every debug job submitted to the batch scheduler. From Nvidia specifications [3], we have

- Number of processor cores: 2496
- Processor core clock: 706 MHz
- Memory clock: 2.6 GHz
- Memory bandwidth: 208 GB/sec
- Total board memory: 5 GB
- 20 pieces of 128M $\times$ 16 GDDR5, SDRAM

From information given above, we do however have enough information to create a roofline plot. This plots arithmetic intensity in flops per byte along the horizontal axis and arithmetic performance in gigaflops per second on the vertical axis, along with device-defined peak bandwidth performance and peak arithmetic performance. This determines whether a program may be bandwidth-bound, compute-bound, or both. From the GPGPU specifications,

$$(2496 \text{ cores})(0.706 \text{ Ghz})(2 \text{ flops/cycle}) = (2496)(\frac{0.706 \cdot 10^9 \ \cancel{\text{clock cycles}}}{\text{second}})(2\frac{\text{flops}}{\cancel{\text{clock cycle}}}) = 3524.352\frac{\text{Gflops}}{\text{s}}$$

From this, we have a peak arithmetic performance bound, and from the given memory bandwidth, we have a peak bandwidth performance given by the line $y = 208x$, where x is input in units of arithmetic intensity. The roofline plot given in Figure 2 was generated by a Python script using simple calls to `matplotlib.pyplot`.

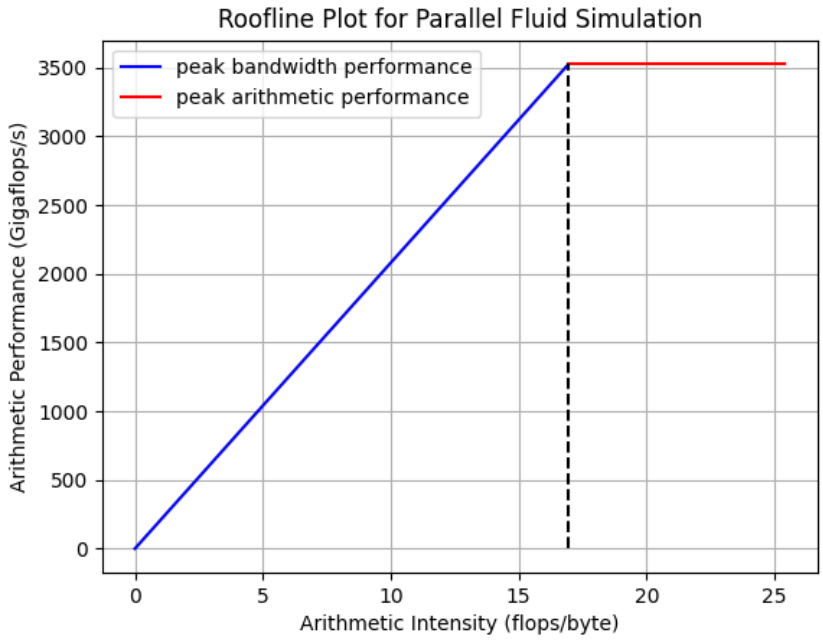

Figure 2: Roofline Plot

This same Python script also calculated the flop counts and memory accesses of each kernel function given in Table 1. The functions used to calculate these values were found manually by counting indivisible floating point operations and each memory read and write operation within the each kernel function. A function for asymptotic complexity, where only the dominant term was kept, determined the values in the table. Within the table, only counts associated with a $32 \times 32 \times 32$ sized mesh were used, since the principal focus here is on the measure of arithmetic intensity, which will have the same ratio value regardless of mesh size.

	flop count (flops)	memory R/W (bytes)	arithmetic intensity (flops/byte)
setInitialConditions	1867776	7077888	0.2638888888888889
copyPeriodic	193536	1396736	0.13856304985337242
zeroResidual	163840	4980736	0.03289473684210526
computeResidual	3932160	23068672	0.17045454545454544
computeStableTimestep	1015808	6160384	0.16489361702127658
minKernel	5120	40960	0.125
integrateKineticEnergy	425984	3276800	0.13
sumKernel	5120	40960	0.125
weightedSum	327680	2621440	0.125
mean of all kernels			0.1417438708955765

Table 1: Arithmetic Intensity on $32 \times 32 \times 32$ mesh

The most important finding here is that every single kernel function is bandwidth bound, as its arithmetic intensity never exceeds that of the intersection of the two performance bounds, given by the dotted vertical line. From this, we can say that further analysis would heavily depend on the read and write frequency of the particular kernel function's algorithm and the memory specifications of the Device and that of the cluster.

Results

Below, the results of running different mesh sizes, which also represented both the number of blocks and threads in all test cases, is given by Table 2.

Blocks/Threads	1	32	64	128	256
Iterations	1775	1775	3554	7109	3555
fluid execution time (ms)	16568	6365.9	20874	218900	686500
time per cell per timestep (ns)	284.85	109.45	22.405	14.683	11.511

Table 2: Runtimes on $n_i \times n_j \times n_k$ mesh with CUDA

We observe an appreciable speedup between the (nearly) serial implementation in the first column and the fully-parallelized program given in the second column. Unfortunately, this is the only simple comparison that may be made at face value since the two programs both use the same mesh size, while the remaining columns are different. No serial runtime was able to be recorded from the majority of the other mesh sizes, since the Scout cluster limited debug runtimes to 30 minutes. Given the total parallelized execution time of the larger mesh sizes, however, we can infer that the runtime speed was drastically improved over mesh sizes under the original implementation. A fluid execution time of 218900 ms = 218.9 s in the case of 128 cells in each direction is obviously better that whatever time was likely to result from the program execution that would take over 30 minutes.

Conclusion

After incrementally implementing each formerly serial function as a kernel function on the Device, all intensive computation is performed on the GPU, with vectors allocated on the GPU being the only ones in play. Thus we observe an appreciable speedup after parallelizing this project with CUDA. The very uniform kernel launches each time step contributed to easy data parallelism, performing repetitive yet complex tasks to a large amount of data concurrently, so that massive data parallelism was achieved. We can expect a significant speedup and the parallelized simulator allows for almost arbitrarily fine meshes, which in some cases will produce much more accurate kinetic energy values. While future work demands more complex utilization of CUDA's architecture and considering of the bandwidth bounded nature of the new kernel functions, this project demonstrated the huge potential of on-Device data parallelism when applied to scientific programming applications.

References

[1] VICTOR EIJKHOUT, Parallel Programming in MPI and OPENMP, vol. 2.

[2] "Introduction to GPU," Introduction to GPU - OpenACC/CUDA for beginners. [Online]. Available: https://enccs.github.io/OpenACC-CUDA-beginners/1.01_gpu-introduction/. [Accessed: 27-Apr-2023].

[3] "Tesla K20 GPU active accelerator - nvidia." [Online]. Available: https://www.nvidia.com/content/PDF/kepler/tesla-k20-active-bd-06499-001-v03.pdf. [Accessed: 01-May-2023].