HPSC Lab 13

Luna McBride

December 21 2023

1 Introduction

GPUs are becoming a strong contender in many industries for speedup potential. This is not just true for high-performance scientific computing, but for machine learning and data science as well. Pushing content to the GPU can increase speeds of larger jobs significantly when done correctly, but it is an art that is quite difficult to master. This final lab will provide a sandbox to assist in wrangling this topic by introducing the concept of GPU usage with OpenACC and speeding up the ovenWalls code from last lab. A second version of the code has also been added for testing, specifically doing its calculations inline to show off GPU capabilities with different styles of code.

Both versions of the code were run first without changes to establish the baseline behavior of the code. The graphs are expected to look the same as the previous lab. These initial graphs are included below to both show both the expected results as well as the correctness of the baseline models:

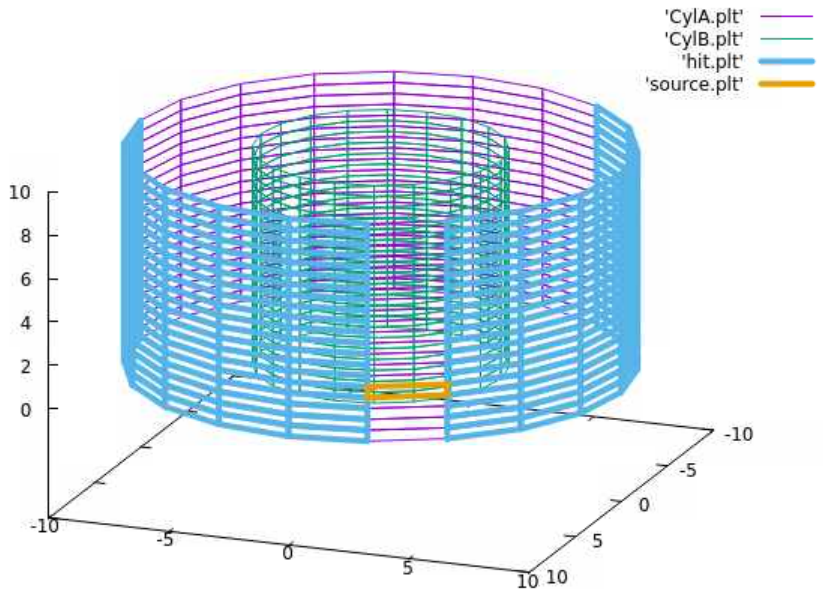

Figure 1: The baseline behavior of the inline model

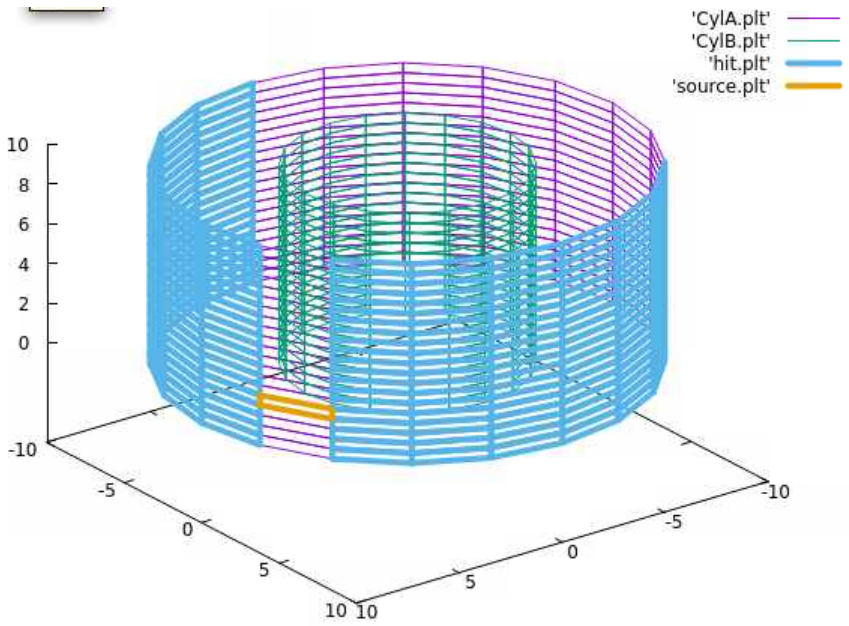

Figure 2: The baseline behavior of the functional model

The view of the images were altered to show that these were indeed different graphs, but the concept is the same. They both are outputting the expected results before any changes have been made. Any GPU change should still result in a graph that looks like this. The code is expected to behave the same throughout the whole process with the one exception of GPU usage, which is to show only in the times provided and not the graph's output. There is a section later on to verify this factor, so expect to see these images again. They will just be from slightly different angles to prove image uniqueness.

The times are the main focus of the assignment, so it is just as important to establish those baselines. The time it took each model to execute on both CPU and GPU are displayed here:

Version	**CPU**	**GPU**
Inline	2.075 ms	334.454 ms
Functional	1.235 ms	342.934 ms

Table 1: CPU vs GPU Run-times for Provided Code

The baseline behavior shows heavy slowdown from the GPU processes to start. There is a certain amount of memory passing and setup that is necessary to utilize the GPU, which the code is clearly not accounting for. The CPU and GPU do not necessarily share memory, more often than not requiring a bridge between two separate parts of memory. This code clearly does not account for this, given the large amount of time leak that is not accounted for elsewhere.

It is also interesting to note in the baseline behavior that the inline code is faster on GPU already, but slower on CPU. This shows the basic behavior of the inline code is more suited for the GPU even before optimization. Such a difference being this clear before any change proves promising for overall speedup. This is likely why GPU usage is being associated more with machine learning and

Python. Inline functions may be the player to beat in this match-up.

2 GPU Porting Process Summary

The success criterion for this lab will be to simply get the code running faster on GPU than its CPU counterpart. This may be a broad goal, but being able to do so on a smaller dataset will prove to be a challenge in itself.

The code that was given was a good place to start, as it already included a line of OpenACC code, being:

```
#pragma acc enter data copyin( coordB[ 0:nFieldB+1 ][0:3], faceB[ 0:nFacesB+1 ][0:5],
normalB[ 0:nFacesB+1 ][0:3], blocked [ 0:nFacesB+1 ] , Ray[ 0:3], Ray0 [ 0:3] )
#pragma acc parallel loop private(  Q0[0:3], Q1[0:3], Q2[0:3], Q3[0:3],
xyzInt[0:3], normal[0:3], dotProduct, dotProduct2 ) \
present( coordB [ 0:nFieldB+1 ][0:3], faceB  [ 0:nFacesB+1][0:5],
normalB[ 0:nFacesB+1][0:3], blocked[0:nFacesB+1], Ray[0:3], Ray0[0:3] )
```

This code in itself was a good clue on the general structure, but that was not the interesting factor about it. The interesting part was actually its placement, creating the parallel loop zone before the blocker loop rather than being before the main loop. Another parallel loop before the main loop would not work due to OpenACC not allowing nested parallelism. Simply removing the parallel from this loop would not work because the present statement is an aspect of the parallel clause, not the loop clause. The only way that could work to allow parallelism around the whole main loop without removing the present clause was to bring the whole thing above the main loop. The entire clause could be moved out besides blocked and Ray due to the changes made before this loop, so the copyin for these two would remain the same. Dot product variables were also moved to accommodate. This was change number 1.

The second change comes down to the original intention for the first change, being to add the #pragma acc loop to each of the loops. Now that the parallel is on the outside, marking the loops simply makes it easier to parallelize. This was change number 2. Altering the sumBlocked loop to break when a block was found was also attempted around this time, but it proved to be slower in this case.

The third change was interesting. When playing around with the present that was now at the top, it became clear that much faster times occurred by removing the data in. This only worked if the present still existed at the top and the data in for blocked and Ray that was created from the first step remained intact. Commenting out that other data in would have it throw errors for data not being present. This actually was the step that drove the decision to not change anything before the timer start, as it really brought down the time from both of these aspects together. This change alone, however, brought down the time significantly. Memory operations are some of the most expensive, so it makes sense that removing them would cause speedup. This was also the point where these initial changes were added to the inline code, so consider that a step 3.5.

The fourth change applies specifically to how the inline code handles the inside angles. The code uses a summation system to check if the ray is in the block. It very well could have used continues to break the loops if the ray is not in that face. This change was made to both the CPU

and GPU code, but it was made for the sake of the GPU code due to unnecessary loops. This change puts the overall code in line with the functional code, putting them at almost equal time values for both GPU and CPU.

This is actually where the lab came to an abrupt halt. There was a large chunk of time spent to attempting to move the normalA, centerA, normalB, faceB, and coordB arrays into the GPU fully. The timer had been moved to account for the loops that were originally made to fill those arrays as well, as a time save of around ten times means nothing when it is on the back of an around 1000 times increase before the timer. These numbers are not fake; the GPU came to around 0.124 milliseconds from 1.234 milliseconds in the established section where the initial loops before the timer came to 300.533 milliseconds from 0.032 milliseconds. There were many experiments done to try and bring that number even further down to no avail. This cycle would have continued for a long time if it had not been noted that the time in that section looked eerily similar to the time shaved off by removing the data copyin.

That is when the realization hit that the arrays may have been put into the GPU, but they are filled by accessing the cylinder information, which was not on the GPU. The same data transfer slowdown was happening since the GPU arrays needed to access the cylinder CPU memory. This means the cylinder would need to be in GPU as well. Even if there was still plenty of time to do this, memory problems were already occurring from the combination of these arrays being in memory and the required copyin for the internal loop. It is not clear whether or not the cylinder in itself could fit into GPU memory, plus the added complexity of trying to parallelize a class would prove to be beyond the scope of this assignment.

As for some additional experimentation attempts, reduction was also toyed with to attempt speedup. The majority of loops being small kLOOPs, however, proved to only slow the program down rather than speed it up. In fact, any attempt to speedup the small kLOOPs only caused slowdown from the extra time the compiler took to process the keyword. Gang and worker tags were also originally added to the loops explicitly, but the compiler was more optimal when distributing them implicitly. Adding nohost to the functions was also considered to reduce the amount living on the CPU, but this would not work due to the CPU section also using those functions. Independent and async were also added to the bigger loops for a point, but they did not provide much speedup, even for larger meshes.

The mesh structure given did not provide much room for speedup besides data transfer components and basic loop parallelization. There was always some detail that lowered the possibility for speedup just by the nature of how the code works. The GPU could not see speedup past that of the CPU for this reason, but it did get quite close. They kept pretty even at the end for larger meshes, coming to a CPU-GPU ratio of 3:4. This is still a massive improvement over the initial 1:118 for the inline code or 1:342 for the functional code. One change specifically to the inline code even put the two codes at almost the exact same speed for both CPU and GPU components when the inline started at about half the speed. The goal of complete speedup beyond the CPU may not have been met, but given the issues caused by the nature of the code, the significant speedup seen is, in itself, a miracle.

3 Correctness Verification

3.1 Correctness at Each Step

Note: given the structure of the code, all changes made to factors shared between the two codes could easily be implemented in the other with minimal difficulty. As such, one graph will be shown for each change to show correctness while experimenting. Any change that can only exist due to a unique aspect of one of the files will be noted as such.

Change Number 1: Moving the parallel loop outside (functional).

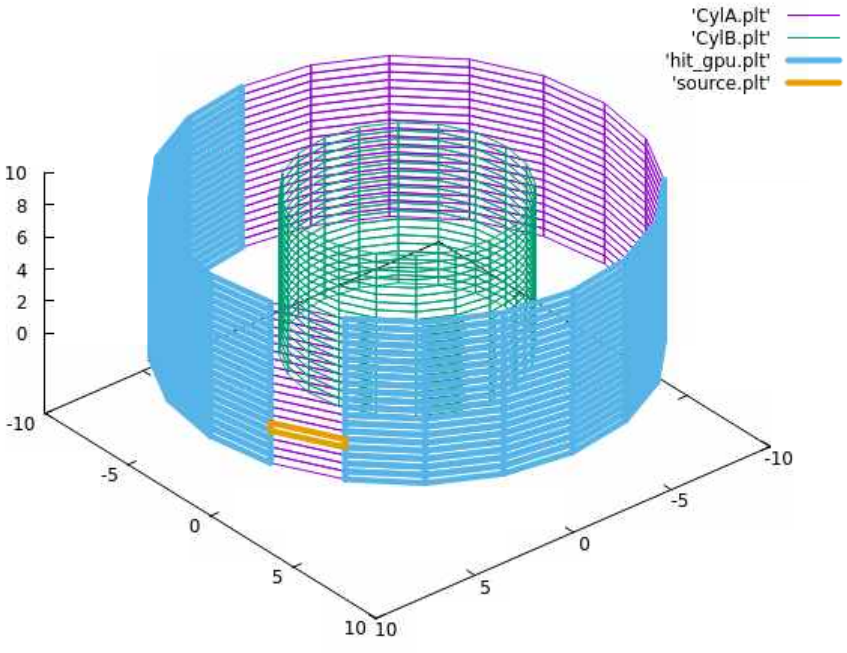

Figure 3: Change 1 Image

Version	CPU	GPU
Before Step 1	1.235 ms	342.934 ms
After Step 1	1.370 ms	306.556 ms

Table 2: Step 1 Before and After

Change Number 2: Adding acc loop to each loop (functional).

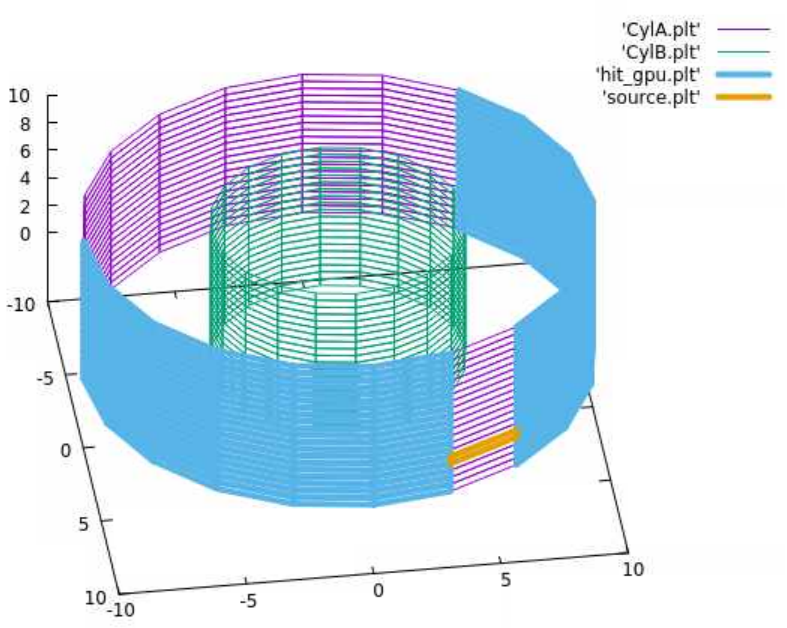

Figure 4: Change 2 Image

Version	**CPU**	**GPU**
Before Step 2	1.370 ms	306.556 ms
After Step 2	2.161 ms	299.572 ms

Table 3: Step 2 Before and After

Change Number 3: removing the first data in (functional):

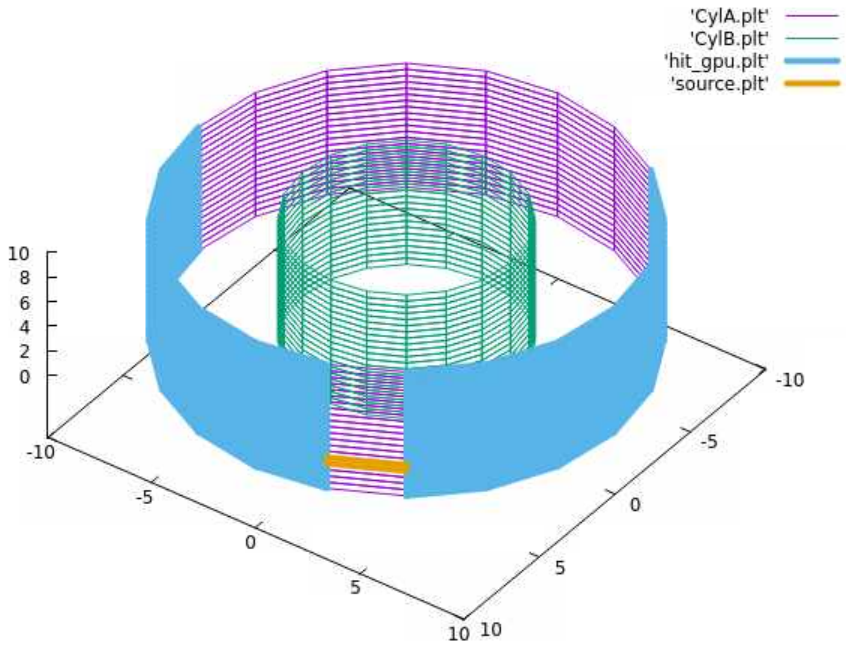

Figure 5: Change 3 Image

Version	CPU	GPU
Before Step 3	2.161 ms	299.572 ms
After Step 3	1.231 ms	1.637 ms

Table 4: Step 3 Before and After

Change Number 3.5: Updating the inline with these major changes (inline):

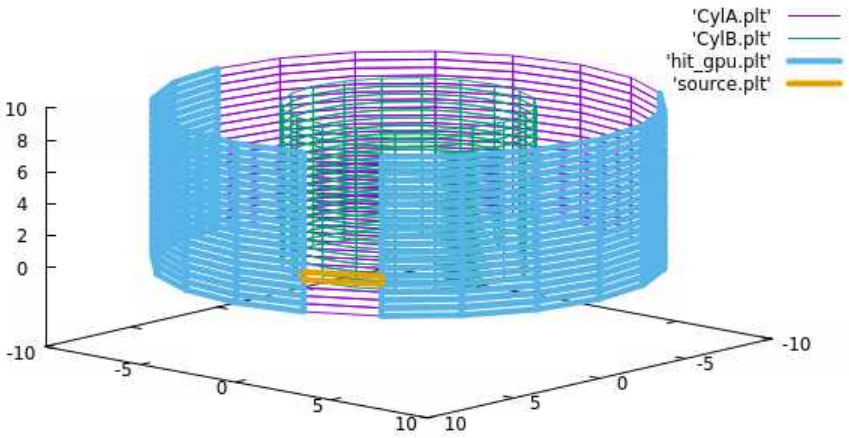

Figure 6: Change 3.5 Image

Version	CPU	GPU
Before Step 3.5	2.075 ms	334.454 ms
After Step 3.5	1.882 ms	2.384 ms

Table 5: Step 3.5 Before and After

Change Number 4: Updating the inside with continues (Inline):

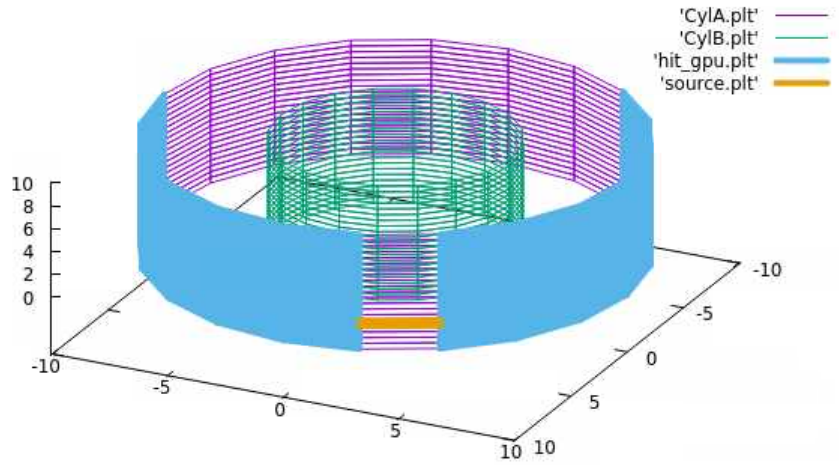

Figure 7: Change 4 Image

Version	CPU	GPU
Before Step 4	1.882 ms	2.384 ms
After Step 4	1.234 ms	1.617 ms

Table 6: Step 4 Before and After

End Speeds:

Version	CPU	GPU
Functional	1.231 ms	1.637 ms
Inline	1.234 ms	1.617 ms

Table 7: End Speeds