Uzochi Dimkpa
Final project – Hash Collision Calculation – Architecture 1
ITCS 5182-001
High Performance Computing
Prof. Erik Saule
November 21, 2023

To (attempt to) further improve on my first architecture's optimizations, I began porting the code to GPU kernels. At this stage, the most important task at hand was to move the functions over to the GPU and get them to function properly. In order to do so, I had to replace the strings and vector types with plain arrays of unsigned characters and integers (8 or 32 bits long).

One of the optimizations problems I had to decide early on was how the guess string would work. Generating variable length strings on the GPU was not very feasible, so I settled on doing fixed-length guesses in order to cut down on code complexity. This also allowed me to be able to more consistently check

I first tried some rather niche optimizations, where I turned the most common calculations into macros, figuring that the function overhead would be eliminated, and the results would compound and add up as more and more calculations were made. The macros did not actually make much of an improvement at all, but I decided to leave them in there to improve code appearance.

I then decided to use threads to increase the number of concurrent calculations being made. I settled on using 128 blocks with 256 threads per blocks, as my machine was able to handle it. I also figured that the thread initialization and resolving overhead would be minimal compared to the amount of work each thread would be doing. I figured the same would apply when adding up the number of hashes each thread individually calculated, because for the thousands (up to potentially millions) of guesses each thread would make, it would only need to lock and unlock a mutex one time and make a single addition to properly count the total number of hashes calculated across the threads.

I do not have any pictures of GPU utilization because the windows Task Manager does not give me an option to view CUDA usage.

The calculations per second I get as a result of these optimizations went to over 110% that of my first architecture usage, but then they would drop to around 55 – 60% that of the first architecture optimizations.

General improvements have been made and implemented. The GPU architecture submission is officially the highest-throughput submission for this problem!

I decided to print out the guess counts of all of the threads to see how many guesses each thread would make before a solution was found. I very quickly discovered that not all of the threads were making the same amount of guesses, and the difference was by a very large margin. For example, I would push a 1-character length message file to the solution and then count the guesses per thread, only to realize that some threads were making only 1 guess while others were making around 29 to 30 or even up to 50.

I deduced that these threads with low guess counts were newer in that they weren't running at the same time as some of the 'older' threads; these older threads were being resolved before the new ones could begin. I then decided to cut down the number of total threads being allocated to 4096

(16 blocks, 128 threads per block), and that improved my throughput of hashes calculated per second to over 110% the performance of my first architecture optimization submission, which equates to about 930000 hashes/sec.

However, for message files that are two characters or longer, the performance visibly decreased back to about 70% or so of the first architecture optimization. So, I decided to move the message digest array to shared memory, because that array would be accessed by every single thread, so storing it locally should theoretically improve access speeds, thus improving hash calculation speeds overall.

As it turns out, moving the message digest array to shared memory drastically improves performance, pushing it back to that original peak of 110% that of architecture 1's optimization (930000 hashes/sec).

Increasing the character length of the message brings performance back down, so I decided to do some more shared memory optimizations by moving some of the random state generators to shared memory. This took up too much shared memory, however, so I eventually removed those optimizations.

From the experiments that I am running, the length of the input message seems to have a drastic impact on performance in hashes calculated per second. At this point, moving the performance goal post makes no sense, so I will have to leave it as it is for now and figure out why the length of the message messes with performance. It seems to be that there is a sweet spot between message length and optimal performance between 2-character messages and having all of the shared memory optimizations I could use. I am currently only seeing the first 4096 threads actually doing work before the others are (seemingly) spooled up, so I will stop here for now.