MCA Lab 5: GPU and CUDA 2
Lab Report

Haopeng Han	Nawabul Haque	Xu Cao	Sanjay Santhosh Kumar
367944	368140	367965	367909

February 12, 2015

1 Introduction

In this lab, We have transformed a CPU version of a password cracker to a GPU version. The purpose was to exploit the GPU capabilities to execute the program in parallel in order to achieve the results faster. GPU provides a lot of computational resources which can be used for computation in parallel. However the key to exploit these resources is to find the parallelism in the application: The blocks of the program which can be run in parallel will be executed on different threads of the GPU in parallel. Additionally, to not have memory accesses as the barrier for the performance improvement, it has to be made sure that memory accesses to DRAM have to be avoided as much as possible.

2 Simulation Results

The 4 letter passwords found are as follows:

fire, blue, tong, cool, dark, qian, temp, qwer, test, pass

We also found the password for 6 letters running it on the GPU cluster:

https://www.pdc.kth.se/resources/computers/zorn/hardware

The 6 letter passwords found are as follows:

joshua, energy, qwerty, rkqian

Table 1 provides the performance statistics of the two version of the programs we developed for GPU in two stages. The unoptimized version was developed in the first stage and provided only parallel execution of the code block. In the second stage, the program was optimized with various techniques like loop unrolling, function coalescing and reduced memory accesses etc., to provide a better execution performance. As evident from the table, with the unoptimized version, the simulation cycles required to get the result is more than 5 times the cycles required for the optimized version. We observed that the simulation for our program on the TUB Ubuntu server takes on average 12 to 15 minutes to complete and provide the result.

Table 1: Performance statistics of the program

Program	gpu_sim_cycle	gpu_sim_insn	gpu_ipc
Unoptimized(1st Phase)	1968537	1465979033	744.7048
Optimized(2nd Phase)	367995	228510891	620.9620

Table 2 provides the cache statistics of the two version of the program, unoptimized and optimized versions, as explained above. This table clearly illustrates the difference between the cache characteristics of the two versions. At the same time, it helps to figure out one prominent reason behind the difference in performance characteristics of the two versions as shown in the Table 1. It is very evident that the

unoptimized version has a lot of L1 data cache accesses and almost 94% of these accesses result in misses which degrades the performance. In case of optimized version, the number of accesses to L1 data cache is lower and the number of misses are just 56. Although the number L1 instruction misses are higher in case of optimized version, it is still low and close to 10%.

Table 2: Cache statistics of the program

Program	(L1I cache stats) (accesses, misses)	(L1D cache stats) (accesses, misses)	(L1C cache stats) (accesses, misses)	(L2 cache stats) (accesses, misses)
Unoptimized(1st Phase)	24496394,1819	1506844,1417102	2067238,1091	1417558,458570
Optimized(2nd phase)	4222138,432141	14933,56	129682,1920	56919,63

Table 3 provides the performance comparison between CPU and GPU in terms of execution time. We can clearly see the benefits achieved by the GPU version. The performance boost in terms of execution time speedup achieved is above 700.

Table 3: Performance comparison

Program	Execution time (μs)
CPU version	374186
GPU version	525

3 Explanations and observations

There were two phases in which we transformed CPU version to GPU version. In the first phase, we converted the C program to create blocks which can be run parallel to each other. This new version could be used with the thread features of GPU and a multiple threads could run in parallel to execute this block to achieve the required computations.

However, the results were not satisfactory and it required a lot of simulation cycles(1968537) to finish the computation. We figured out that the *for* loop used in MD5 hash calculation was the bottleneck for the performance. This loop uses a lot of variables for the calculation, and therefore, a lot of memory accesses are required. Since there are several threads running in parallel at a given point of time, the number of memory accesses will be very huge. We figured out that this version had a high number of accesses (1506844) to L1 data cache and most of them (1417102) resulted in a cache miss as reported in the result of the simulation of this version. Also the loop creates branching which degrades the performance as jump instructions are very expensive. And therefore, we need to unroll this loop in MD5 calculation and another loop while comparing the calculated hash with the given hash for 4 letter passwords.

Taking the cue from the above mentioned issues, we proceeded to the second phase of the transformation of the program. First we unrolled the for loop being used in MD5 calculation to avoid various memory accesses and branching. We also unrolled the loop for comparison of the calculated hashes with the provided hashes. This optimization led to a very minimal number (50) of accesses to L1 data cache. Although 46 out of these 50 accesses to L1 data cache resulted in miss, it did not hamper the performance as they were very few. With these changes we managed to reduce the simulation cycles required from 1968537 to 508219.

We also looked for other venues where we could have achieved performance benefits and tried to optimize in those areas as well. First we realized that the call of a function from another function leads to various stack related operations and management. This definitely hampers the performance and therefore, we merged the functions being called from the kernel function into the kernel function itself. We observed that this provided close to 1% improvement in the overall simulation cycles. We also did some optimization for the code we had written and made it use less resources. One example is the copying of a found password into a memory location exclusively without any contention from other thread. Earlier we were using a loop to identify the free location in the password memory structure using atomicCAS operation. However, we realized that we can achieve it by having a global variable to identify the free location and thus we reduced the number of atomic operations.

Another venue we found where we could have achieved some performance benefits was to stop the execution of all the threads to be started as soon as we found all the 10 passwords. We tried to use *trap* instruction to stop all the threads after all the passwords were found, however, this instruction did not work with the simulator and threw an error at runtime due to an assertion failure. Then we achieved this through setting a flag after we found all the result and checking it before letting a new thread proceed with the execution at the beginning of the kernel. This provided us a great improvement of 21% and the number of simulation cycles reduced to 367995 from 467081.

After we were done with the 4 letter password, we proceeded to crack the 6 letter passwords on a real GPU hardware. We had access to a Supercomputer at KTH PDC center for high performance computing. We modified our optimized version of 4 letter password program. For every single anticipated password, we created a new thread to run in parallel. As reported in the results section, the program reported the four 6 letter passwords.