Brute Force Hash Code Cracking: Performance of GPU compared to CPU

Bradley Mayo
Bryce Browner

Abstract

This paper is an analysis of the performance of a GPU versus a CPU for brute force hash cracking. The results of this paper are not meant as the be-all and end-all of results for performance, rather it is simply an example of a use case of GPU over the CPU and a demonstration of the speed-up that can result. We also want to take into consideration the amount of effort that it takes to make each implementation. This is important in the real world when deciding the types of implementations you are going to write. In our specific tests of brute forcing MD5, we used both OpenMP and CUDA against an arbitrarily chosen password. To crack this password, we hashed every string possible, after which we compared the resultant hashes to the hash of the actual password. We stopped hashing and testing strings once we found the correct one.

1 Introduction

GPUs were initially created to speed up or allow for 3D game rendering. Since their initial implementation, GPUs have progressed far past their primary purpose. While a GPU is still necessary for high performance gaming, their unique optimizations allow them to be useful for things other than just graphics rendering. “GPUs are optimized for taking huge batches of data and performing the same operation over and over very quickly, unlike PC microprocessors, which tend to skip all over the place” [1]. Generally, CPUs have a few cores that have a relatively high amount of cache memory that can work with a couple of software threads at one time. GPUs, in contrast, have hundreds, if not thousands of cores that can simultaneously run orders of magnitude more threads than CPUs. While CPUs are still the primary tool for many computational tasks, many problems are more quickly solved in parallel which makes GPUs superior for them. Common examples of this are transcoding video between different formats, image recognition, and virus pattern matching [1].

The point of our research was to see how much of this speed boost was theoretical and how much was actually attainable for an easily threadable problem. The problem we decided to use for this purpose was a simple brute force hash crack. Because each string hash is independent of all the others and the hashing process is the same for each string, having a large number of threads to hash each individual string concurrently should see a large improvement in performance over the lower number of threads of the CPU.

This paper is broken into several sections. The related work section covers other similar tests which we found detailing the performance differences between GPUs and CPUs. The system model describes the system environment on which our test code is being run. This is important because different system structures can give vastly different results. The problem statement is an exact description of the brute force hashing problem. The

solution section describes the steps our code takes to solve the brute force hash cracking problem while summarizing our specific code. Analysis covers the results of our experiment while simulation and experimentation covers the specific outputs we got. Appendix A contains the actual code we used to do the testing.

2 Related Works

Obviously, when there are claims of several orders of magnitude of speedup for computation, there is also a lot of argument over whether or not that is true. In this section, I will cover the results of other people in attempting to quantify the benefits of using a GPU over a CPU for certain applications.

In a self-analysis of a paper written by a Berkeley student, they describe the problems they found when trying to compare CPU and GPU performance. While the paper found that GPU implementations of the RSA algorithm showed a factor of 22.6 to 31.7 improvement over the CPU implementation, they also cover several caveats to this speed-up factor [2].

Primarily, they describe three deadly sins. One, not all parts of algorithms are completely offloadable to the GPU. Thus, when analyzing performance of GPU algorithms, you must also account for the added time of GPU computations and any computation that is still done on the CPU. Essentially, when running a GPU-centric algorithm, it is important to record the whole runtime and not just the GPU kernel runtime.

Second, it is important to not assume that the input data is already in memory on the GPU. Data transfers are a large part of the GPU performance time and thus must be accounted for when doing benchmarks. In some cases, it could be possible that the memory transfer times are larger than the time spent by the GPU doing computation.

Finally, it is often assumed that you will have large enough data for the parallelism to make full use of the GPU. In many real-life scenarios, this is not the case. As such, the speedup can be significantly less as the overhead of using the GPU may be not worth the small performance boost of a non-fully utilized GPU.

Luckily, for our testing purposes, none of these problems are difficult to solve. When doing a brute force hash crack, you will always have full utilization as you can simply parallelize your hash attempts based on your number of cores. Additionally, it is easy to consider the transfer and CPU processing times by simply taking the benchmark of the full execution of both the CPU and GPU implementations. This creates a more “fair” comparison between the two different methods.

In an article discussing the engineering applications of GPUs, they come to their own conclusion about the necessity of CPUs and GPUs. They mention the possibility of using only GPUs for engineering applications [3]. The problems they have with this are primarily the time it takes to develop specifically for GPUs and the misleading

nature of some of the speedup claims [3].

While a large speedup seems worth it, often there is a difficulty in writing code that can be run on the GPU. Engineering groups would also have to take time and money transitioning their hardware from CPUs to GPUs, their software from all CPU to all GPU, and their skillset in that they would have to be writing for a completely different architecture [3]. The cost of this transfer is not trivial and thus the speedup may not even be worth the cost in some cases.

The other problem with using only GPUs for engineering groups is with the misleading nature of the speedup claims. While GPUs can achieve significant performance boosts in some parallel areas, their performance in normal serial code can often be worse than that of a CPU [3]. Thus, while a GPU might perform spectacularly at some part of an application, if that portion of the application is not significant, it is possible to see an overall performance decrease by using a GPU over a CPU in some cases.

While the other sources are hesitant to claim overall superiority of GPUs, NVidia is highly confident in the ability of their cards. In a blog post written on NVidia's website, they cite the fact that GPUs being "only" 14 times faster than CPUs is misleading [4]. They then go on to say that benchmarks aren't necessary to see the benefits of using GPUs over CPUs.

When it gets right down to it, benchmarks aren't real life. So, to truly see the effect of GPUs, we can look at the speed up gained by people who have implemented GPU-centric systems. In their article, NVidia cites 10 different real-life developers that have achieved speedups of at least 100x in their real-world use cases. These developers range from Cambridge University (100x improvement) to Massachusetts General Hospital (300x improvement) [4].

NVidia also went on to debate the development difficulties of implementing things on a multi-core CPU versus a GPU. They claim that students learning about parallel programming could achieve 35x more performance from a GPU over a CPU with the same investment of coding time [4].

While these other articles discuss the tradeoffs between using only CPUs or only GPUs or the benefits of large-scale implementations of GPUs, we hope to narrow down the benefits that an individual user can experience by optimizing some parts of their code using GPUs. By using CUDA, we can use both the CPU and GPU when appropriate and measure the difference of using GPU code on a single home system. By measuring the overall performance of a single application using the two different approaches on a normal user device, we hope to demonstrate the capabilities of GPUs to benefit everyday users.

3 System Model

The system we did our testing on was a home gaming system. The video card was an NVidia GTX 970, and the CPU was an Intel i7 2700K. The system had 32GB of RAM. We feel this was representative of a real developer's system and could provide an accurate estimate for what an individual user could see. The operating system used was Ubuntu 17.10.

4 Problem Statement

We are measuring the performance boost that can be achieved by using a GPU over CPU multithreading for a single user using a single practical real-world application. This is also a measure of development effort for both types and a benchmark for a real world individual user's system. By comparing an everyday computer's CPU against its GPU, we should be able to practically show the normal benefit that an end user would experience when using a GPU through CUDA.

5 Solution

To demonstrate the speedup possible for an individual user we decided to write a program that an individual user could decide to run that was also able to be made to run concurrently. Thus, a brute force hash cracker was decided upon.

In our specific implementation, we used a modified open source string generator to create the strings which the GPU would then hash [5]. This implementation uses a linked list approach which allows for us to generate and test arbitrarily long strings. It only generates strings with ASCII values from 32 to 126 which are all "printable" characters and the space character.

These generated strings are then hashed. This is done in three different ways to benchmark the differences. The first implementation is a simple synchronous solution using no parallelization. In the second implementation, we use OpenMP to use CPU parallelization to achieve a speedup for the brute force algorithm. In the third implementation, we use CUDA to harness GPU parallelization. After these implementations complete their hashes, the results are compared to the "actual" password hash we are testing against. These computation times are then compared and we can view the speedup of each implementation.

The GPU implementation specifically uses a batch of input strings with the size of the batch being equal to the number of CUDA cores on the system we were using. The number of threads we used for the OpenMP version was the total number of CPU threads available on the system with the NVidia card we were testing.

Our timing measurement for both implementations is from full start to finish. That means that all memory allocation, memory movement, processing, and memory freeing are included in the run time. This allows our results to demonstrate the speedup to the user as opposed to just the speedup of the GPU code over the CPU code specifically.

6 Analysis

The string we decided to test against for consistency was "!!!". This was because "!" is the first printable ASCII character so that is a relatively quick key to find without being too small.

The simple implementation with no concurrency could find the key by generating each string, hashing it, and comparing the hash against the input hash in .170284 seconds.

The OpenMP implementation managed to do the same thing using CPU parallelization in .499808 seconds. This time is much longer than the basic implementation and is probably due to the low benefit of doing the hashing concurrently compared to the overhead of using the concurrency. With only 8 threads, we are only able to hash 8 inputs at a time. But to create these 8 threads, we must do much more memory copying, function calls to get the thread IDs, and even waiting for the slowest thread after every string batch. Overall, the OpenMP version could probably be improved greatly, but with a similar amount of effort to the non-concurrent version (note that we are testing the speedup based on effort) we saw it take more than 3x longer on average.

While we attempted to implement the hash crack using CUDA, we reached the limit of our time constraint. This is important to note because both the simple implementation and the openMP CPU concurrency implementation were written in less time than it took to not be able to implement the CUDA version. While this obviously does not show the actual speedup possible with CUDA, from a developer's standpoint, it could be stated here that the performance for effort of the CUDA implementation was infinitely less than both other implementations as they were completed within a reasonable amount of time. CUDA may have been worth the additional effort if we were attempting to create an industry-level cracker or to bypass someone's password, but in this case, it doesn't seem to be worth the additional time.

7 Conclusions

While the theoretical speedups achievable with both CPU and GPU concurrency are great, we have found that for a single use case, it can often be easier and faster to simply use the most basic implementation.

8 Bibliography

References

[1] https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

[2] http://people.eecs.berkeley.edu/~sangjin/2013/02/12/CPU-GPU-comparison.html

[3] http://www.digitaleng.news/de/gpu-vs-cpu-computing/

[4] https://blogs.nvidia.com/blog/2010/06/23/gpus-are-only-up-to-14-times-faster-than-cpus-says-intel/

[5] https://gist.github.com/pazdera/1123306

[6] https://github.com/weidai11/cryptopp

[7] https://github.com/xpn/CUDA-MD5-Crack

Appendix A

Our code can be found at https://github.com/Brycey92/CUDAHashSlingingSlasher