




PARALLEL CRYPTOCURRENCY HASHING USING CUDA

Mohammed Armughanuddin

TEAM NAME - HASHING SPEEDSTERS

SID -862395044

marmu001@ucr.edu



Project Idea / Overview

Presentation Link- <https://youtu.be/IYKwZ6MNO3Y>

This project was conceived with the aim to increase the efficiency and speed of cryptocurrency hashing functions through the implementation of parallel computing. Specifically, we utilized the NVIDIA CUDA (Compute Unified Device Architecture) platform, a software layer that provides direct access to the GPU's virtual instruction set and parallel computational elements, to expedite the execution of the SHA256 hashing algorithm.

The SHA256, a member of the SHA-2 cryptographic hash functions designed by the NSA, is prominently used in the process of mining Bitcoin, hence, optimizing this process could lead to significant improvements in mining efficiency. The core of our project revolves around the understanding that mining is fundamentally a race to solve complex mathematical problems, which inherently involves repeated hashing. By boosting the speed of these hashing functions, we could drastically augment the performance of the cryptocurrency mining process.

In systems like Bitcoin that employ Proof of Work (PoW) as their consensus algorithm, the mining operation, which involves solving intricate mathematical problems through repeated hashing to add new blocks to the blockchain, is of paramount importance. This process is often computationally intense and time-consuming when executed sequentially. However, by parallelizing these hashing functions, we can significantly reduce the time taken to find the solution, thereby increasing the efficiency of the PoW mechanism.

To demonstrate the efficiency and speed of our parallel computing approach, we developed a password cracking application utilizing the parallelized SHA256 hashing function. This was not built with malicious intent but was designed to provide a tangible comparison of the improvements achieved by parallel processing over traditional sequential methods.

As a crucial part of this application, we designed a robust password generator that produced a multitude of varying and complex passwords. These passwords were then hashed using the SHA256 algorithm to create a hash database. By tasking the password cracking application to match the hashes with their corresponding plaintext passwords, we were able to effectively benchmark the performance and efficiency of our parallelized hashing algorithm. Additionally, I designed a sequential version of the program to demonstrate the difference in speed between the two.

How Is The GPU Used To Accelerate This Program?

The CUDA program provided harnesses the computational prowess of the General Processing Unit (GPU) to accelerate the brute force password cracking process. This task involves going through every possible password combination until the correct one is found. The GPU's ability to execute operations in parallel significantly boosts this process. The central component of this parallelism is the efficient adaptation and parallelization of the SHA-256 hash algorithm, which includes the use of '#pragma unroll' to further optimize the efficiency. Here's the detailed program breakdown:

1. **Parallel Algorithm Details:** The program employs a parallel algorithm where each thread on the GPU independently generates and tests potential passwords. Each thread calculates the SHA-256 hash of a possible password using an adapted CUDA version of the SHA-256 algorithm. The algorithm employs the CUDA **device** function qualifier to denote functions that run on the GPU. Additionally, the '#pragma unroll' directive is used in the SHA-256 algorithm to fully unroll loops, thereby reducing the branching overhead and optimizing the execution on the GPU. CUDA's **device** and **constant** qualifiers are utilized alongside GPU-specific functions such as ROTRIGHT and ROTLEFT for bitwise operations. When a hash matches the original password's hash, it indicates that the password has been cracked. The parallelization and loop unrolling in the SHA-256 algorithm are central to the speed and efficiency of the process.
2. **Problem Space Partition:** The problem space, encompassing all potential passwords, is divided based on the number of blocks and threads per block, a critical aspect of ensuring efficient parallel execution of the SHA-256 algorithm. This partitioning is evident in the lines `unsigned long long thread_index = blockIdx.x * blockDim.x + threadIdx.x;` and `if (thread_index < password_count)`. Every thread has a unique thread_index, used to generate a distinct potential password.
3. **Software Pipeline Parallelization:** The stages of the software pipeline that are parallelized include the generation of potential passwords, the calculation of their SHA-256 hashes, and the comparison of these hashes to the hash of the original password. This parallelization is implemented within the password_cracker kernel, executed on the GPU. While the parallel execution and loop unrolling in the SHA-256 algorithm significantly boost computational speed, some tasks still run on the CPU, such as variable initialization, operation timing, progress reporting, and password match verification. The process overall demonstrates a balance between CPU and GPU tasks, with a strong emphasis on the GPU parallelization of the SHA-256 hash algorithm to optimize efficiency and speed.

Implementation Details

The implementation of our CUDA program is rooted in the efficient use of GPU resources to execute a parallelized version of the SHA-256 algorithm for brute-force password cracking. This section provides an overview of the major implementation aspects, emphasizing the parallel computation and optimized execution on the GPU.

1. Kernel Configuration: In a CUDA program, computations on the GPU are performed by functions known as kernels. The kernel configuration involves setting up a grid of blocks, where each block consists of multiple threads. In our program, the kernel configuration is set up in a way that maximizes GPU utilization, with the number of threads per block chosen based on the GPU's capability.

2. Loop Unrolling: To reduce the overhead associated with branching, loop unrolling is performed in the SHA-256 algorithm using the '#pragma unroll' directive. This technique eliminates the control flow instructions required for loop increment and termination check, thereby leading to a more streamlined execution. Loop unrolling in the SHA-256 algorithm enhances efficiency by ensuring concurrent execution of independent operations.

3. Password Generation: The generation of potential passwords is divided among the GPU threads to ensure parallel execution. Each thread is assigned a unique index, which is used to generate a distinct password. This design enables simultaneous testing of multiple password combinations, greatly accelerating the cracking process.

4. Hash Calculation and Comparison: After a potential password is generated, each thread calculates its SHA-256 hash. The SHA-256 implementation is tailored for parallel execution on a GPU, using the CUDA **device** and **constant** qualifiers and GPU-specific bitwise operations. The calculated hash is then compared to the original password's hash. If a match is found, the kernel records the matching password and signals that the password has been cracked.

5. Host-Device Communication: Data is transferred between the host (CPU) and the device (GPU) using CUDA's memory management functions. These functions are used to allocate memory on the GPU, copy the original password's hash to the GPU, retrieve the cracked password from the GPU, and clean up the GPU memory after the computation is finished.

The CUDA implementation of the brute-force password cracking and the parallelized SHA-256 algorithm demonstrate a significant speedup compared to a CPU-based implementation. The extensive use of parallelism, combined with an efficient GPU-specific adaptation of the SHA-256 hash algorithm, makes this program a highly effective solution for password cracking tasks.

How To Run The Code?

1. Clone the Github Repository
2. Change directory to the repo
3. Type `"nvcc -o parallel-sha256 parallel-sha256.cu"` to compile the parallel-sha256 and generate the parallel-sha256.out file.
4. Run the parallel SHA256 algorithm using `"./parallel-sha256"`. The default password is set as "armug" (first five letters of my name), but any password can be tested by passing it as an argument like `"./parallel-sha256 password"`.
5. For compiling the sequential version, we have to use `"gcc -o sequential-sha256 sequential-sha256.c sha256.c -std=c99 -lm"`
6. We can run the sequential version using `"./sequential-sha256"`. The default password is set as "armu" (first four letters of my name), but any password can be tested by passing it as an argument like `"./sequential-sha256 password"`.

Evaluation & Test Results

The efficiency of the parallelized SHA-256 hash function was benchmarked using a password cracker against a sequential counterpart. The following table illustrates the results of this test:

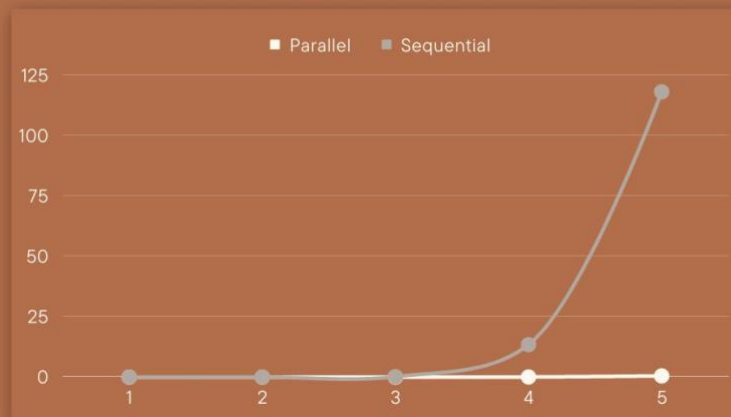
Password Size	Parallel Time	Sequential Time
1 letter (m)	~0 seconds	~0 seconds
2 letter (zz)	~0 seconds	~0.01 seconds
3 letter (zzP)	~0.001 seconds	~0.19 seconds
4 letter (kOmZ)	~0.05 seconds	~13.43 seconds
5 letter (armug)	~0.5 seconds	~118 seconds (easier example chosen for testing, could go up to 12 minutes)

These results indicate a substantial improvement in speed due to the parallelization of the SHA-256 function. While the time difference for cracking smaller passwords is minimal, the advantages of the parallel algorithm become remarkably evident as the password length increases.

For instance, for a 5-letter password, the sequential algorithm took roughly two minutes (for an easier example chosen for testing, it could extend up to 12 minutes) whereas the parallel algorithm managed to crack the same password in approximately half a second. This results in a speed-up of roughly 300x on average. For 6-letter passwords, the sequential approach becomes nearly infeasible to run due to the high computational time. However, the parallel algorithm continues to demonstrate its efficiency by producing results in about 40 seconds.

Following the tabulated results, we've also included a graph for a better visualization of the time scaling of both algorithms. This graphical representation gives a clear picture of the drastic difference in time taken by the parallel and sequential algorithms with increasing password length. The diverging trend lines highlight the pronounced efficiency of the parallelized SHA-256 hash function as the password size increases.

SHA-256 Password Cracking Comparison



Hence, the data clearly validates the efficacy of our parallel implementation in reducing computational time and improving performance.

Problems Faced

The development of our CUDA program for brute-force password cracking presented several challenging issues, which primarily revolved around parallelizing the SHA-256 algorithm and designing effective test cases.

1. Parallelizing the SHA-256 Algorithm: The SHA-256 hash algorithm was initially not designed for parallel execution. As a result, one of the major challenges was the adaption of this algorithm for parallel execution on a GPU. The algorithm involves several sequential operations and dependencies that had to be carefully managed to allow for concurrent execution. This required a deep understanding of both the algorithm itself and GPU architecture. It also involved careful tuning and optimization, such as loop unrolling, to ensure that the parallel version was not only functional, but also efficient.

2. Designing Test Cases: It was a challenge to design specific test cases that accurately reflected the program's efficiency. The brute-force approach inherently involves a significant amount of computation, especially when dealing with longer or more complex passwords. Designing a test case that was complex enough to demonstrate the program's capability, while not being so complex that the computation would take an unreasonable amount of time, required a delicate balance.

3. Ensuring Thread Safety: Given the concurrent nature of the program, thread safety was a key concern. It was crucial to ensure that race conditions, in which multiple threads try to update the same variable simultaneously, did not occur. Handling this problem required us to implement atomic operations that ensure only one thread can update a variable at a time.

4. Memory Management: Effective memory management was another hurdle. The GPU has various types of memory (like global, shared, constant, etc.) each with different characteristics in terms of size and speed. Deciding what data should reside in which type of memory, to maximize performance, was a complex task. Furthermore, data transfers between the host (CPU) and the device (GPU) could be costly, so we had to carefully manage these to minimize the negative impacts on performance.

5. Debugging: Debugging parallel programs can be significantly more challenging than debugging sequential programs, due to the potential for timing-related bugs and the difficulty of monitoring the behaviour of individual threads. Additionally, traditional debugging tools are often designed for CPU-based programs and might not provide complete support for GPU-based applications.

Despite these challenges, we were able to develop an effective and efficient solution for brute-force password cracking using CUDA. Through careful design, rigorous testing, and continuous optimization, we managed to overcome the difficulties and create a parallel program that significantly accelerates the password cracking process.

Task Breakdown

Task	Breakdown
Implementation	Mohammed- Armughanuddin - 100%
Report	Mohammed- Armughanuddin - 100%

References

1. SHA-256 implementation - <https://github.com/B-Con/crypto-algorithms/blob/master/sha256.c>
2. SHA-256 Parallelization using CUDA - <https://www.youtube.com/watch?v=6AXl0sGjxg0>