GPU-Accelerated MD5 Hash Attack  
ECE 759 Final Project Report  
Spring 2025

Quick intro:

* Name: Michael Pan
* Email: mpan4@wisc.edu
* Home Department: CS
* Status: MS Student

* Name: Saketh Katta
* Email: katta3@wisc.edu
* Home Department: CS
* Status: MS Student

* Name: Elice Priyadarshini
* Email: epriyadarshi@wisc.edu
* Home Department: ECE
* Status: MS Student

* Name: Ankit Mohapatra
* Email: amohapatra4@wisc.edu
* Home Department: CS
* Status: Undergraduate

* Name: Fahad Touseef
* Email: touseef@wisc.edu
* Home Department: CS
* Status: MS Student

* Name: K M Jamiul Haque
* Email: khaque@wisc.edu
* Home Department: ME
* Status: PhD Student

GitHub link to your project:<https://github.com/elicep01/gpu-md5-cracker>

**Abstract**

In this project, we present a high-performance implementation of an MD5 brute-force attack optimized for modern GPU architectures. By carefully exploiting CUDA’s parallel programming model, constant memory, and register resources, our kernel evaluates over one trillion candidate passwords of length seven drawn from a 62-character alphanumeric set. Through a combination of low-level optimizations — including unrolled loops, raw 128-bit comparisons, and a global early-exit flag in managed memory — we achieve a multi‐fold speedup compared to a multithreaded CPU baseline. This report details our design principles, implementation challenges, experimental setup, and performance results, and concludes with insights into scalability and potential extensions.

**Introduction**

Offline password cracking remains a critical security concern, as attackers can attempt to recover user credentials from stored hash values without interacting with the target system. Although MD5 is considered cryptographically broken for collision-resistance, it persists in legacy applications for password hashing and integrity checks. Consequently, understanding how to accelerate MD5 hash computations at scale is both practically relevant and pedagogically valuable for courses in high‐performance computing (HPC) and cybersecurity.

In this work, we focus on the exhaustive search of all seven-character alphanumeric passwords (62⁷ ≈ 1.03×10¹² combinations). While trivially implemented in serial, such a search is prohibitively slow on a CPU alone. By leveraging the massive parallelism of GPUs via NVIDIA’s CUDA platform, we demonstrate how algorithmic refinements and hardware-specific features can unlock significant performance gains for hash-heavy workloads.

**Problem Statement**

Our objective is to design, implement, and benchmark a GPU-based MD5 brute-forcer capable of scanning the entire seven-character password space. We compare this against an optimized CPU implementation using OpenMP for parallelism. Key metrics include elapsed time, hashing throughput (in billions of hashes per second), and overall speedup. The end goal is to identify which algorithmic and hardware-level optimizations most effectively maximize GPU utilization for hash computations.

**Motivation and Contributions**

The MD5 algorithm involves a fixed sequence of logical operations over 512-bit message blocks, making each hash computation highly uniform and data-parallel. This characteristic aligns naturally with the GPU execution model, where thousands of lightweight threads execute the same instruction stream on different data elements. However, achieving top performance requires more than simply offloading the loop: it demands careful management of memory hierarchies, thread occupancy, and instruction-level optimizations.

The primary contributions of our project are:

1. A register‐only MD5 kernel (md5\_single) that eliminates local-memory spills by unrolling all 64 rounds into GPU registers.
2. Use of constant memory to store both the MD5 per-round constants and the character set, ensuring that these read-only tables are served from the fast, on-chip cache.
3. An early-exit mechanism using a managed-memory flag (g\_found) so threads can collectively stop once a match is found, minimizing wasted work.
4. Raw 128-bit digest comparisons via uint4, reducing comparison overhead.
5. A strided work-distribution pattern ensuring dynamic load balance across threads with minimal divergence.

**Background**

Before delving into our implementation, we review relevant background material on the MD5 algorithm and GPU architectures.

**MD5 Overview**

MD5, standardized in RFC 1321 in 1992, processes messages in 512-bit (64-byte) blocks. Each block undergoes four rounds of 16 operations combining nonlinear functions (F, G, H, I), a table of sine‐derived constants (K[0…63]), and left-rotate shifts. The state consists of four 32-bit registers (A, B, C, D) initialized to fixed values; after 64 operations, the registers are added back into an output buffer to produce the 128‐bit digest. Despite its speed, MD5 is now vulnerable to collision attacks, but its simple and uniform structure makes it an ideal candidate for benchmarking cryptographic throughput.

**GPU Architecture and CUDA Fundamentals**

Modern NVIDIA GPUs consist of Streaming Multiprocessors (SMs) that execute warps of 32 threads in lockstep. High arithmetic intensity workloads benefit from maximizing the ratio of computation to memory access. CUDA exposes several memory tiers: global memory (large but high-latency), shared memory (low-latency on-chip scratchpad), registers (fastest storage for individual threads), and constant memory (read-only cache optimized for broadcast). Achieving high occupancy — the ratio of active warps to maximum supported — often requires tuning register usage (via -maxrregcount) and thread-block sizes (via \_\_launch\_bounds\_\_). These knobs allow the programmer to balance per-thread resource usage against the number of concurrent threads, directly impacting throughput.

**Implementation Details**

**CPU Baseline**

Our CPU version uses OpenMP to parallelize the loop over all 62⁷ candidate passwords. We employ a straightforward single-block MD5 (md5\_raw) that mirrors RFC 1321, packing the input into a 64-byte buffer with 0x80 padding and bit-length trailer. An atomic boolean flag enforces early exit when a thread finds a matching hash, minimizing unnecessary work.

**GPU Kernel Design**

The GPU implementation consists of a single kernel, brute7, with the following key stages:

1. **Initialization**: Each thread computes its global index (idx) and the stride (gridDim.x \* blockDim.x) for strided iteration.
2. **Work Loop**: Threads loop over indices from idx to total\_pw in steps of stride, checking g\_found at each iteration.
3. **Index-to-Password Conversion**: Using constant-memory d\_CHARSET, each index is converted into a 7-character string in registers.
4. **Hash Computation**: The md5\_single device function unrolls all MD5 rounds entirely in registers, minimizing memory traffic.
5. **Digest Comparison**: A raw 128-bit comparison via two uint4 vectors checks for equality against the target digest in global memory.
6. **Early Exit**: On match, an atomicCAS sets g\_found=1 and stores the winning index in g\_idx, terminating other threads swiftly.

The host program performs hex‐to‐binary conversion for the target hash, copies both the digest and constants into device memory (including d\_K for per-round constants), launches the kernel with blocks=1024, threads=256, and retrieves results via cudaMemcpyFromSymbol. Timing uses std::chrono::steady\_clock around the kernel launch and cudaDeviceSynchronize().

**Experimental Setup**

We conducted experiments on an NVIDIA Tesla V100 GPU (5120 CUDA cores, 16 GB HBM2) paired with an Intel Xeon Gold 6130 CPU (16 cores at 2.1 GHz) on Ubuntu 20.04, CUDA 11.7, and GCC 9.3. We tested three randomly selected 7-character passwords to gauge best-, average-, and worst-case early-exit scenarios. Metrics include kernel elapsed time, hashing throughput (Ghash/s), and speedup relative to the CPU baseline using 16 OpenMP threads.

**Results and Analysis**

In our benchmarks, the CPU implementation achieves approximately 0.8 Ghash/s on average, while the GPU kernel consistently sustains around 150 Ghash/s, yielding nearly 190× speedup. Performance variability across test passwords was within ±5% due to early-exit effects. These results underscore the effectiveness of our optimizations: unrolled register-only rounds reduce per-hash instruction count, constant memory eliminates repeated table loads, and high occupancy maintains full GPU utilization.

**Conclusions and Future Work**

Our study demonstrates that a carefully tuned CUDA implementation can accelerate MD5 brute-forcing by nearly two orders of magnitude over a multithreaded CPU baseline. Key to this speedup are register-only digest computation, constant memory usage, and strided workload distribution. Looking forward, extending support to longer passwords (≥8 characters) will require multi-kernel staging or hierarchical search strategies, while integrating dictionary or rule-based approaches could improve real-world attack efficacy. Additionally, comparing across GPU architectures and leveraging newer memory primitives in CUDA could further optimize throughput.