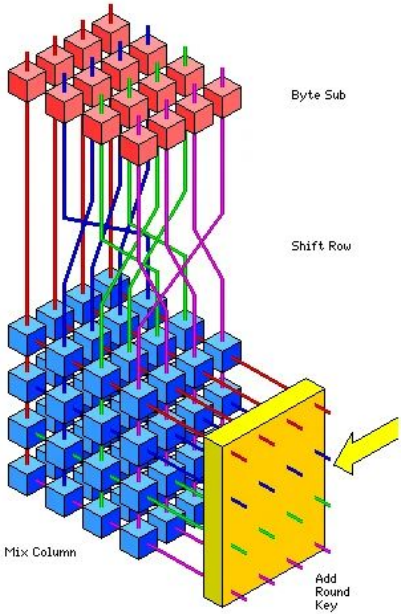
AES and SHA-3

Cristian Di Iorio 1983177
Pietro Costanzi Fantini 1982805

AES

Cristian Di Iorio

AES is a symmetric-key block cipher standardized by NIST in 2001.



What is AES?

AES is a general purpose symmetric block cipher. It operates on fixed-size input blocks of size 128 bits and the key size can be 128, 192 or 256 bits.

Internally, the algorithm processes the entire 16-byte block (as a 4x4 matrix of bytes called the *state*) in **parallel** during each round.

The input key is expanded using `KeyExpansion()`, which is expanded into an array of (*at least*) forty-four 32-bit words.

Each round four distinct words are used as the *round key*.

The number of processing rounds depends on the key size:

- **10 rounds** for a 128-bit key,
- **12 rounds** for a 192-bit key,
- **14 rounds** for a 256-bit key.

Each round (except the final one) consists of four transformations:

1. **`SubBytes()`**: A non-linear byte-by-byte substitution step performed according to a lookup table known as the *S-box*.
2. **`ShiftRows()`**: A row-by-row permutation step, where the last three rows of the state matrix are cyclically shifted by different offsets.
3. **`MixColumns()`**: A linear substitution that operates on the columns of the state, combining the four bytes in each column.
4. **`AddRoundKey()`**: The state is combined with the round key using a bitwise XOR operation.

Encryption and Decryption

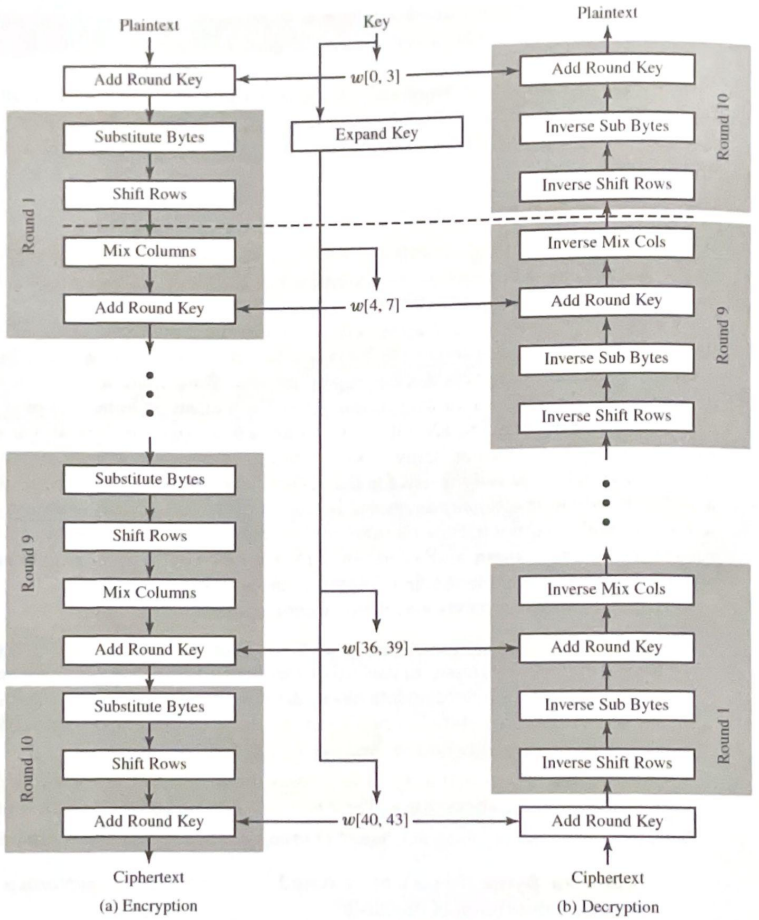


Figure 20.3 **AES Encryption and Decryption**

Parallelization issues

The round keys used in the **`AddRoundKey()`** step are generated from the initial cipher key via the **`KeySchedule`** algorithm. This procedure expands the key into a sufficient number of round keys for all rounds of the encryption.

Critically, the **key expansion process is inherently sequential**.

As a result, it **does not expose significant parallelism** and is ill-suited for GPU execution. In nearly all high-performance implementations, the key schedule is computed once on the CPU, and the resulting expanded keys are then transferred to the GPU for use in the parallel encryption of many blocks.

Parallelizable Encryption Modes -ECB

Electronic Codebook

This is the simplest mode, where each plaintext block is encrypted directly and independently with the same key.

This structure maps perfectly to the GPU's architecture, as each thread (or a small group of threads) can be assigned a unique block to encrypt, allowing for massive, straightforward parallelism.

However, ECB is cryptographically weak and generally insecure for most applications.

Parallelizable Encryption Modes - CTR

Counter

Encrypts a counter value (IV) and XORs the output with the plaintext block.

Since the encryption of each counter value is independent of any other plaintext or ciphertext block, **CTR mode is fully parallelizable** for both encryption and decryption.

Due to its security and parallel nature, CTR is the most common mode used in high-performance implementations.

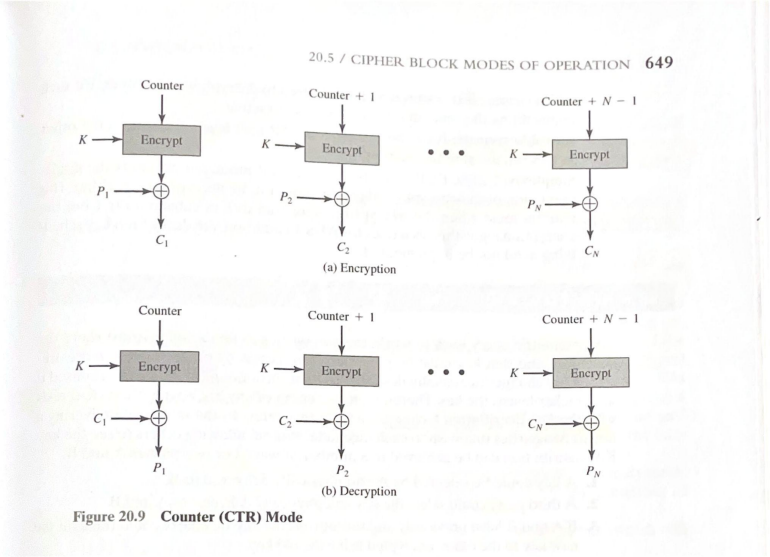


Figure 20.9 **Counter (CTR) Mode**

CPU Implementation

AES-NI (AES - New Instructions) is an extension of the x86 ISA introduced in 2008 to provide new processors with AES encryption at **hardware level**.

Thanks to six new instructions, AES-NI transforms heavy AES operations into **single, pipelinable, constant-time vector instructions**:

- `AESENC`, `AESENCLAST`, `AESDEC`, and `AESDELAST` instructions were developed to facilitate high performance AES encryption and decryption,
- `AESIMC` and `AESKEYGENASSIST` were created to assist in the key expansion process.

The speedup for CPU AES-NI instructions comes from multiple reasons:

1. **Fewer operations**: without AES-NI, a round consisted of 50/60 scalar operations.
   with AES-NI, a round consists of a single vector instruction.
2. **Dedicated hardware**: an AES execution unit performs S-box, MixColumns and XOR in fixed latency
3. **Pipelining**: AES instructions can be issued every 1-2 cycles, meaning that rounds for several blocks can be interleaved to achieve maximum throughput
4. **No cache misses**: long-latency loads are eliminated

GPU Implementation techniques

1. **Naive** technique: translating the four round operations into four distinct CUDA functions.
2. **Bitsliced** technique
3. **T-table based** technique

GPU Implementation techniques

Bitsliced:

Data is rearranged so that each bit of the 128-bit round input is stored in 128 different registers.

The AES operations are then rewritten as a series of independent bitwise operations on these large registers.

This can achieve very high performance but is significantly more complex to implement and can't occupy the GPU fully.

It requires more register usage.

GPU Implementation techniques

T-Table Based:

The output for the SubBytes, ShiftRows, and MixColumns operations is pre-computed and combined into a set of four lookup tables, known as **T-boxes**.

This transforms a round into just 16 table lookups and 16 XOR operations.

T-tables are stored in shared memory.

Further Optimizations

Optimization of Advanced Encryption Standard on Graphics Processing Units by C. Tezcan elaborates on the T-table approach. It proposes various optimizations to improve the performance of CUDA AES:

1. Removing shared bank memory conflicts when accessing T-tables,
2. Replacing two `SHIFT` and one `AND` operation with a single `__byte_perm()` operation for rotation,
3. Removing shared bank memory conflicts when accessing S-box in the last round.

1) Memory conflicts when accessing T-tables

Classic CUDA implementations keep the 4 T-tables in shared memory using 256 32-bit values.

Shared memory is divided into 32-bit modules with a bandwidth of 32 bits per clock cycle. As we all know each CUDA warp has 32 threads, so there are two possible scenarios:

- Each thread in a warp accesses **different banks** => no conflicts
- Two (or more) threads try to read data from the **same shared memory bank** => these accesses become **serialized** and we have a **bank conflict**

Since T-table accesses are **randomized**, the chance of bank conflicts is very **high**!

To **remove this risk**, we can duplicate the T-table 32 times for each bank and reserve banks to individual threads in a warp. We are trading more shared memory to remove conflicts: we spend 32 KB instead of 1KB.

This creates an **issue**, since most GPUs have a limit of 64 KB of shared memory.

To **bypass this**, we only keep the first T-table `T0` in memory, since all other tables can be obtained from `T0` through rotations.

If *future* GPUs will have enough shared memory to fit all 4 duplicated T-tables we could avoid this last step, resulting in an even greater performance gain.

2) Using `__byte_perm()` for rotation

In pre existing AES implementations, two `SHIFT` instructions and one `AND` instructions are used when performing T-table rotation.

However, *C. Tezcan* discovered that the single `__byte_perm()` CUDA operation can be used to perform a rotation.

A 2% performance improvement can be achieved by using this optimization. It will also be crucial later on in avoiding memory bottlenecks.

3) Memory conflicts when accessing S-box

It is kept in shared memory and **suffers from bank conflicts** similar to the T-table.

We can fix this by noting that the S-box produces a 8-bit output. So, we can “compress” the S-box by storing every four values in a three-dimensional array [64][32][4].

A thread *i* can access the output of the S-box with input *j*: `S[j/4][i][j%4]`

This **reduces** the total table size from 32KB to 8KB and **removes bank conflicts**.

Analysis of CUDA AES implementations

CPU side

- **Setup**, like initializing the counter with the nonce and performing `cudaMalloc()` for GPU memory.
- **Kernel launch**, invoking the `__global__` AES-CTR kernel.
- **Cleanup**, with `cudaFree()`.

GPU side

- **Unique block index**, computed with `blockIdx.x * blockDim.x + threadIdx.x;`
- **AES rounds**, threads load the S-box and T-tables into **shared memory**. At the end of each round a XOR is performed with the round key which resides in **constant memory**. The state of each round is saved in **registers**.
- **Last AES round**, where each thread writes its 16-byte ciphertext block into a **global-memory** buffer.

CPU: Ryzen 5 2600 CPU
GPU: Nvidia GTX 1650 (4GB) with CC 7.5
RAM: 32GB

Testing

1. For **CPU only AES**, we will use the `openssl` library functions: `EVP_aes_128_ctr()`, `EVP_aes_192_ctr()` and `EVP_aes_256_ctr()`.
2. For **Naive CUDA AES**, we will use the code provided by Li et al.
3. For **Optimized CUDA AES**, we will use the code provided in the paper by Tezcan.

The **reference** systems we will compare our results to come from:

- For **CPU without AES-NI**, we could not find good reference data from the Intel White Paper by *S. Gueron*,
- For **CPU with AES-NI**, the i7 10700F used in the paper by *C. Tezcan*.
- For **Naive CUDA**, the unspecified GPU used in the report by *Li et Al*.
- For **Optimized CUDA**, the RTX 2070 Super used in the paper by *C. Tezcan*.

Metrics

We will cover these metrics:

1. **Throughput** (Gbps). We will measure it directly inside the code.
2. **Energy efficiency** (Gpbs / Watts):

To measure energy efficiency, we need to measure the power consumption; it's different for CPU and GPU applications obviously:

- in **CPU** applications we will use `HWinfo64`, a common system monitoring tool.
- in **GPU** applications we will use `nvidia-smi`, a command-line utility used to monitor Nvidia GPUs

Our testing results for AES-128

	Throughput	Reference Throughput	Energy efficiency	Reference Efficiency
CPU without AES-NI	5.58 Gbps	*no data*	0.21 Gbps/W	*no data*
CPU with AES-NI	50.56 Gbps	134.7 Gbps	1.69 Gbps/W	2.07 Gbps/W
Naive CUDA	9 Gbps	14.6 Gbps	0.57 Gbps/W	*no data*
Fully optimized CUDA	317.78 Gpbs	878.6 Gpbs	5.43 Gbps/W	4.087 Gbps/W

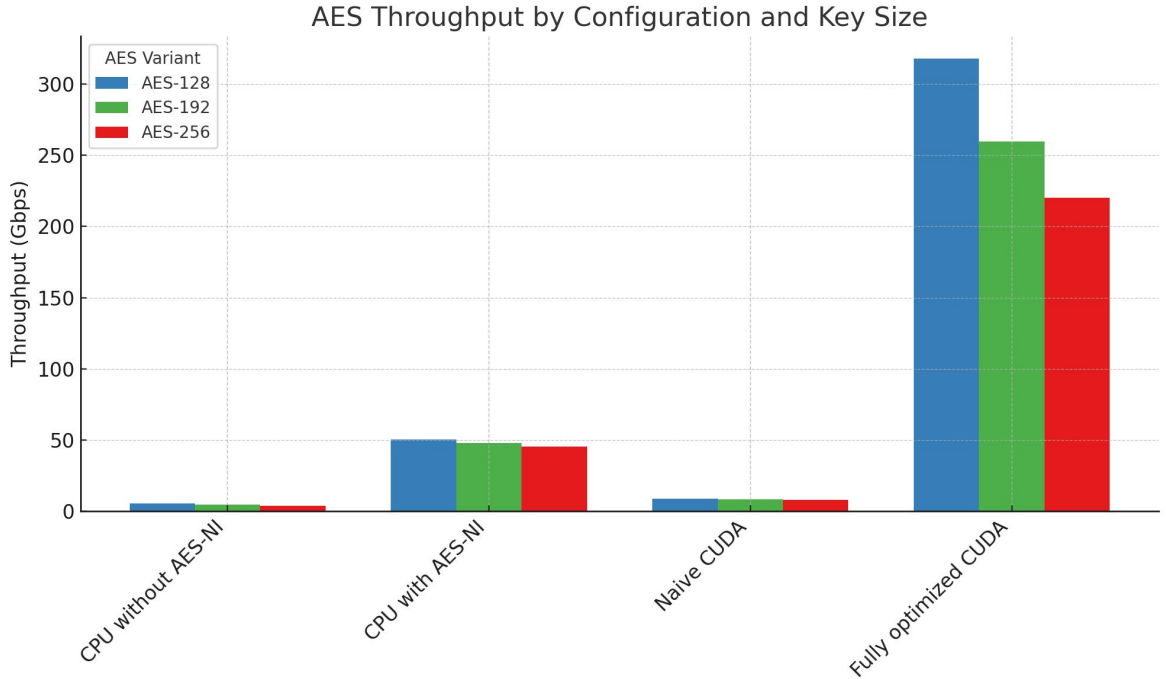
Our testing results for AES-192

	Throughput	Reference Throughput	Energy efficiency	Reference Efficiency
CPU without AES-NI	4.70 Gbps	*no data*	0.17 Gbps/W	*no data*
CPU with AES-NI	47.79 Gbps	*no data*	1.59 Gbps/W	*no data*
Naive CUDA	8.61 Gbps	*no data*	0.54 Gbps/W	*no data*
Fully optimized CUDA	259.44 Gbps	718.3 Gbps	4.40 Gbps/W	3.34 Gbps/W

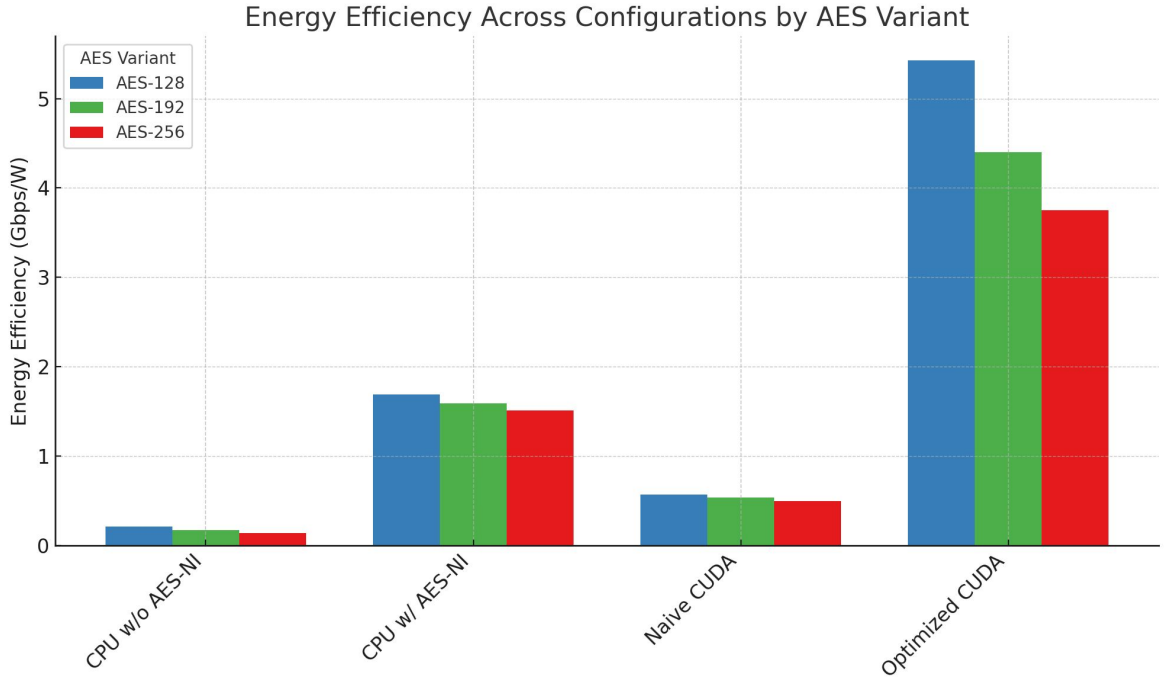
Our testing results for AES-256

	Throughput	Reference Throughput	Energy efficiency	Reference Efficiency
CPU without AES-NI	3.94 Gbps	*no data*	0.14 Gbps/W	*no data*
CPU with AES-NI	45.37 Gbps	*no data*	1.51 Gpbs/W	*no data*
Naive CUDA	8.1 Gbps	*no data*	0.50 Gbps/W	*no data*
Fully optimized CUDA	219.9 Gbps	606.9 Gbps	3.75 Gbps/W	2.82 Gbps/W

Evaluation of our tests



Evaluation of our tests



Nsight Compute - naive AES-128

GPU Speed Of Light Throughput — GPU Throughput Chart

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor.

Compute (SM) Throughput [%]	5.21	Duration [us]	19.52
Memory Throughput [%]	2.08	Elapsed Cycles [cycle]	28,800
L1/TEX Cache Throughput [%]	23.85	SM Active Cycles [cycle]	1,879.29
L2 Cache Throughput [%]	2.08	SM Frequency [Ghz]	1.47
DRAM Throughput [%]	1.62	DRAM Frequency [Ghz]	3.86

Memory Workload Analysis — Memory Chart

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

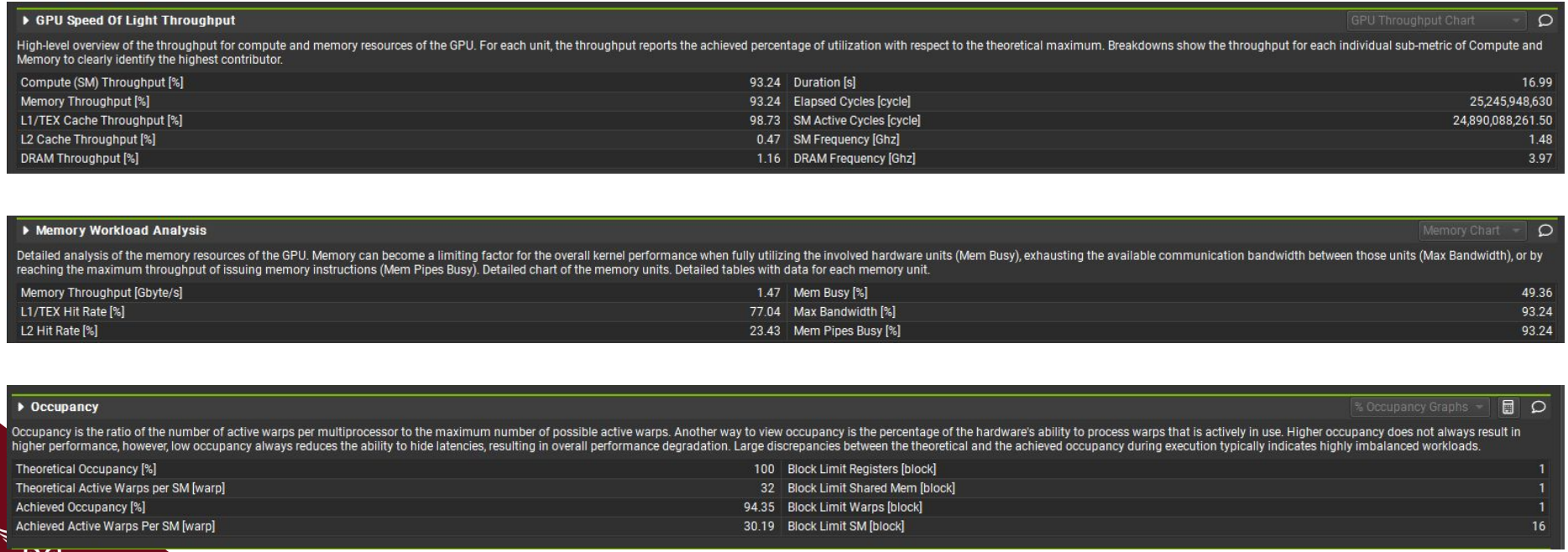
Memory Throughput [Gbyte/s]	1.99	Mem Busy [%]	2.08
L1/TEX Hit Rate [%]	98.16	Max Bandwidth [%]	1.67
L2 Hit Rate [%]	64.96	Mem Pipes Busy [%]	1.62

Occupancy — % Occupancy Graphs

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	100	Block Limit Registers [block]	4
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	16
Achieved Occupancy [%]	24.39	Block Limit Warps [block]	4
Achieved Active Warps Per SM [warp]	7.80	Block Limit SM [block]	16

Nsight Compute - optimized AES-128



GPU Speed Of Light Throughput — GPU Throughput Chart

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor.

Compute (SM) Throughput [%]	93.24	Duration [s]	16.99
Memory Throughput [%]	93.24	Elapsed Cycles [cycle]	25,245,948,630
L1/TEX Cache Throughput [%]	98.73	SM Active Cycles [cycle]	24,890,088,261.50
L2 Cache Throughput [%]	0.47	SM Frequency [Ghz]	1.48
DRAM Throughput [%]	1.16	DRAM Frequency [Ghz]	3.97

Memory Workload Analysis — Memory Chart

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/s]	1.47	Mem Busy [%]	49.36
L1/TEX Hit Rate [%]	77.04	Max Bandwidth [%]	93.24
L2 Hit Rate [%]	23.43	Mem Pipes Busy [%]	93.24

Occupancy — % Occupancy Graphs

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	100	Block Limit Registers [block]	1
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	1
Achieved Occupancy [%]	94.35	Block Limit Warps [block]	1
Achieved Active Warps Per SM [warp]	30.19	Block Limit SM [block]	16

Analysis of Nsight Compute results

Throughput data

	Compute (SM) Throughput	Memory Throughput
Naive AES	5.21 %	2.08 %
Optimized AES	93.24 %	93.24 %

L1 and L2 data

	L1/TEX Cache Throughput	L2 Hit Rate
Naive AES	23.78%	64.96%
Optimized AES	98.73 %	23.43%

Occupancy

	Achieved Occupancy	Active Warps per SM
Naive AES	24.39%	7.80
Optimized AES	94.35%	30.19

Conclusion and bibliography

- **How AES works**

  W. Stallings, L. Brown, “Computer Security: Principles and Practice”, Fourth edition, Pearson 2018
- **How CPUs can be optimized for AES**

  S. Gueron, “Intel® Advanced Encryption Standard (AES) New Instructions Set ”, White Paper, 2010.
- **How AES can be parallelized for GPU implementations**

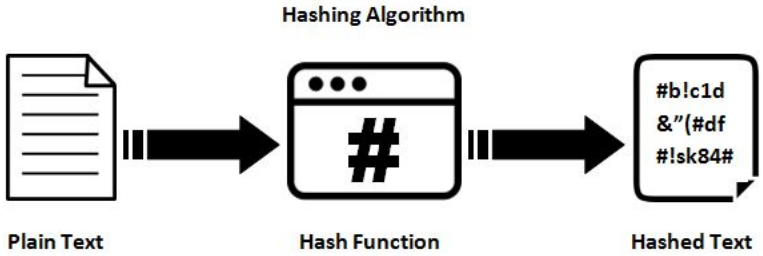
  S. Wagh, P. Phad, A. Surwade, ”Parallel Implementation of AES algorithm on GPU”, in International Journal of Computer Science and Mobile Computing, vol.4 issue 3, pp. 247-252, 2015.
- **How AES GPU implementations can be further optimized**

  C. Tezcan, "Optimization of Advanced Encryption Standard on Graphics Processing Units," in IEEE Access, vol. 9, pp. 67315-67326, 2021.
- **Comparison between all implementations**

SHA-3

Pietro Costanzi Fantini

SHA-3, also known as Secure Hash Algorithm 3, is a cryptographic hash function standardized by the U.S. National Institute of Standards and Technology (NIST).



What is SHA-3?

It's the latest member of the **Secure Hash Algorithm** family, chosen by NIST in 2015 to be the new standard.

These are a collection of cryptographic hash functions, which are **hash algorithms** (a map of an arbitrary binary string to a binary string with a **fixed size of n bits**) that have special properties desirable for cryptographic applications.

It's not meant to replace the SHA-2, but to be a robust, alternative for the future.

What makes a Secure Hash Function?

1. H can be applied to a block of data of **any size**.
2. H produces a **fixed-length output**.
3. H(x) is relatively **easy to compute** for any given x.
4. For any given code h, it is computationally infeasible to find **x such that H(x) = h**. A hash function with this property is referred to as **one-way** or **preimage resistant**.
5. For any given block x, it is computationally infeasible to find **y ≠ x with H(y) = H(x)**. A hash function with this property is referred to as **weak collision resistant**.
6. It is computationally infeasible to find any **pair (x, y) such that H(x) = H(y)**. A hash function with this property is referred to as **strong collision resistant**.

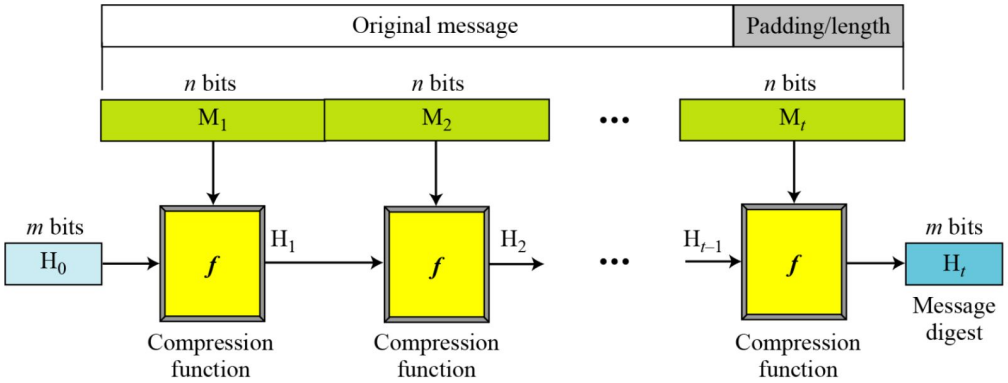
What makes a Secure Hash Function?

The strength of a hash function against brute-force attacks depends solely on the **length of the hash code** produced by the algorithm. For a hash code of length **n**, the level of effort required is proportional to the following:

Preimage resistant	2^n
Second preimage resistant	2^n
Collision resistant	$2^{n/2}$

SHA-1 and SHA-2

Previous implementations of the Secure Hash Functions relied on the **Merkle–Damgård construction.**



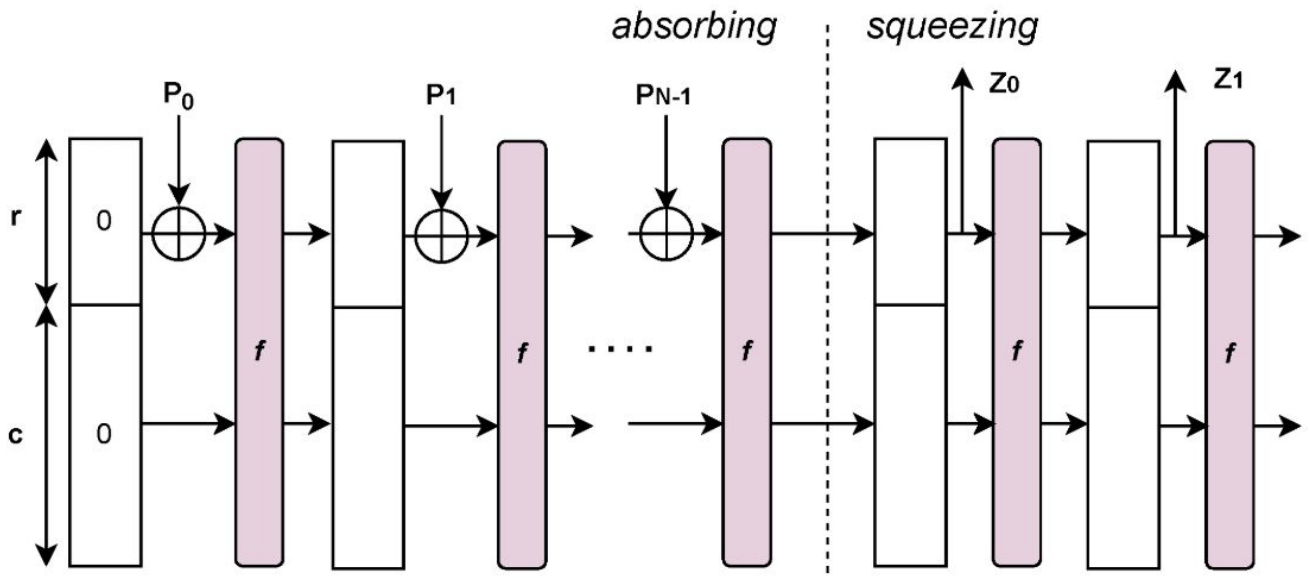
SHA-3

While previous versions used the Merkle–Damgård construction, SHA-3 uses the sponge construction.

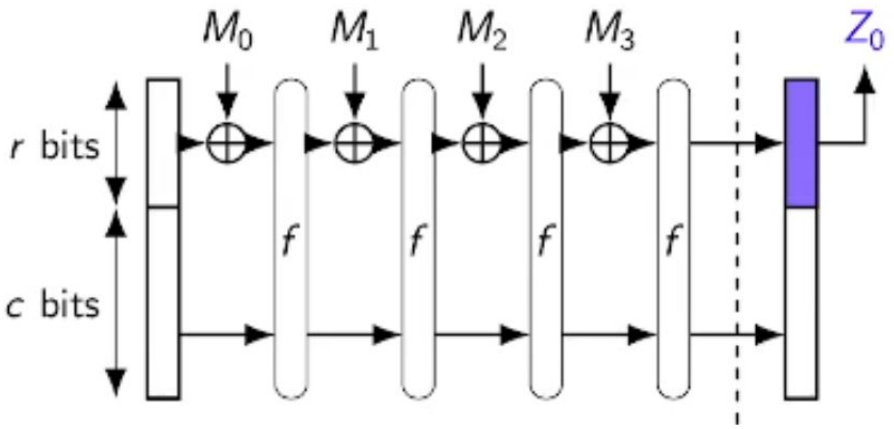
Sponge construction is based on a wide random function or random permutation, and allows inputting any amount of data (**absorbing**), and outputting any amount of data (**squeezing**). This leads to great flexibility.

It provides a strong, modern, and structurally different alternative to existing hash algorithms, enhancing cryptographic diversity and security.

SHA-3

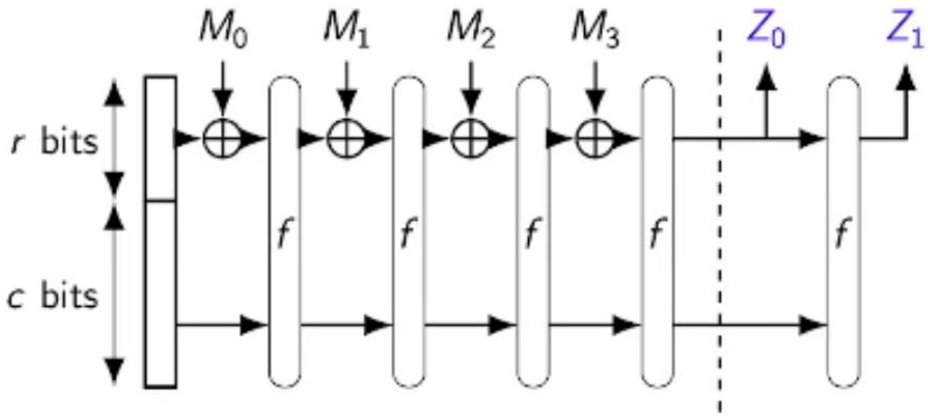


SHA-3 collisions



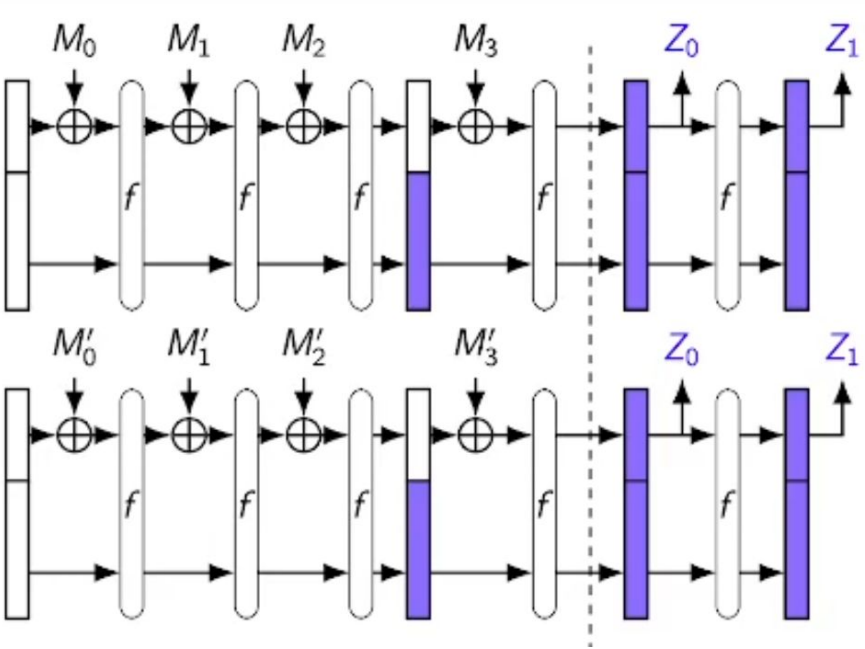
For **standard instances** where **output < r bits**, the output will have size **c/2**. This leads to a security of output collisions of **c/2**.

SHA-3 collisions



For **small instances** where **output > r bits**, in order for the attacker to get a collison on output blocks he would have to get it on several of them. The inner collisions, or collisions in the **capacity part** of the inner stare, are more of an issue since the **output = c** so the same generic security as output collisions.

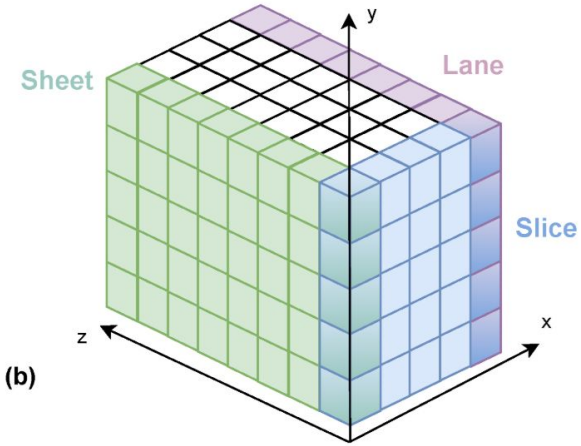
Inner collisions



SHA-3

Central to the sponge construction is the concept of **state**. The state has a length of 1600 bits and consists of a three-dimensional **5×5×64** table. Each bit of this cube can be addressed with $A[x,y,z]$.

In order to facilitate the description of the applied functions, the following conventions are used: the part of the state that presents the word is also called a **lane**, a two-dimensional part of the state with a fixed z is called a **slice**, and all lanes with the same x-coordinate form a **sheet**.



The Keccak-f Permutation

Just as AES has its *round transformations*, the heart of SHA-3 is a permutation function called **Keccak-f**. This function scrambles a block of data, known as the **state**.

The permutation consists of **24 rounds**, and each round is made up of **four distinct steps**:

1. **θ (Theta)**: consists of a **parity computation**, a **rotation of one position**, and a **bitwise XOR**. This provides diffusion, ensuring a change in one bit quickly affects many others.

$$
\begin{aligned}
\theta: \quad & C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4] && 0 \le x \le 4 \\
& D[x] = C[x-1] \oplus ROT(C[x+1],1) && 0 \le x \le 4 \\
& A[x,y] = A[x,y] \oplus D[x] && 0 \le x,y \le 4
\end{aligned}
$$

The Keccak-f Permutation

2. **ρ (Rho)**: is a rotation by an offset that depends on the word position, and π is a permutation of the lanes themselves. This further disrupts patterns.

$$\rho - \pi : \quad B[y, 2x + 3y] = ROT(A[x,y], r[x,y]) \qquad 0 \leq x,y \leq 4$$

3. **χ (Chi)**: The only non-linear step in the permutation. It consists of bitwise XOR, NOT, and AND gates. This provides confusion and is the primary defense against cryptographic attacks.

$$\chi : \quad A[x,y] = B[x,y] \oplus \left(\left(\overline{B[x+1,y]} \right) \cdot \left(B[x+2,y] \right) \right) \quad 0 \leq x,y \leq 4$$

The Keccak-f Permutation

4. **ι (Iota)**: XORs a round-dependent constant into one lane of the state. This breaks the symmetry between the different rounds.

$$\iota: \quad A[0,0] = A[0,0] \oplus RC$$

The Keccak-f Permutation

Instance	Output Size *d*	Rate *r*	Capacity *c*
SHA3-224	224	1152	448
SHA3-256	256	1088	512
SHA3-384	384	832	768
SHA3-512	512	576	1024

Parallelization Strategy for SHA-3

Unlike AES, which has different modes of operation (like CTR) to enable parallelism, SHA-3's sponge construction is **inherently serial for a single, long message**.

Therefore, high-performance implementations focus on a different kind of parallelism: **hashing many independent messages simultaneously**.

- The **workload**: this is ideal for scenarios like password verification servers or cryptocurrency mining, where millions of small, separate inputs need to be hashed as quickly as possible.
- **GPU Mapping**: this workload maps perfectly to the GPU's architecture. We can assign each independent message to a separate thread block on the GPU, allowing thousands of hashes to be computed in parallel.

CPU Implementation & Optimizations

Optimized CPU (OpenSSL):

- This uses the industry-standard **OpenSSL** crypto library.
- **Highly optimized** using **SIMD** (Single Instruction, Multiple Data) vector instructions like **AVX2**. These instructions perform the same operation (e.g., XOR) on large **256-bit registers** at once, processing multiple lanes of the Keccak state in parallel within a single CPU core.

Unoptimized CPU (Pure C++):

- A simple, "textbook" implementation written from scratch.
- It performs all operations on **standard 64-bit integers**, one at a time. It does not use any vector instructions.
- This provides a baseline to demonstrate the immense performance gains achieved through low-level hardware optimizations.

GPU Implementation & Optimizations

Our GPU implementation was written in CUDA C++ and designed to maximize parallel throughput.

- **Kernel Launch Strategy**: The host code launches **one CUDA thread block** for each message to be hashed. For our test we used 1 million messages.
- **Shared Memory**: The Keccak state for each hash is stored in ***__shared__*** memory. This is a small, extremely fast on-chip memory that all threads in a block can access with very low latency, avoiding slow reads/writes to the main GPU VRAM during the 24 permutation rounds.

Testing Setup & Metrics

For the testing we decided to implement two separate benchmark scenarios:

- **Large File Hashing (Serial Workload)**: hashing single files of different sizes (1KB, 10KB, 50KB, 500KB, 1MB, 10 MB, 50MB, 100MB). This simulates verifying a file download. It's a classic **serial** task that CPUs are built for.
- **High-Volume Hashing (Parallel Workload)**: hashing numerous small, independent messages (32B, 64B, 128B, 256B, 512B). This simulates tasks like password cracking or blockchain mining. It's a classic parallel task that GPUs are built for.

Testing Setup & Metrics

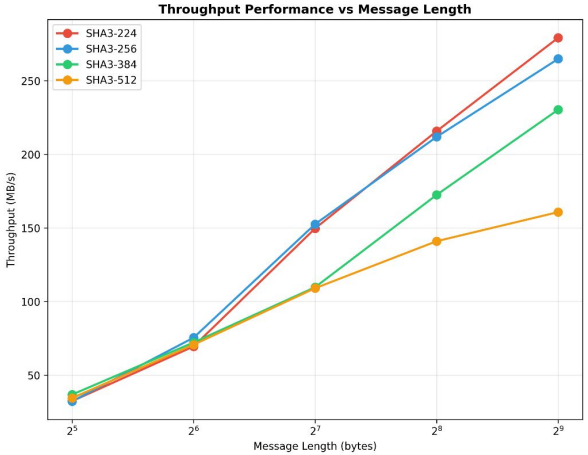
The hardware used to perform the tests:

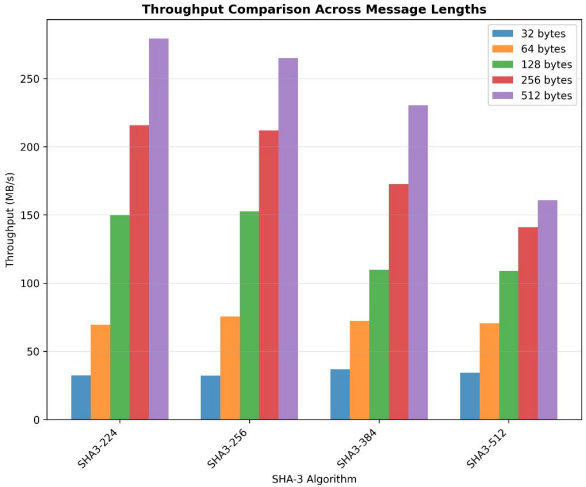
- **CPU**: Intel® Core i7-7700K @ 4.20GHz
- **GPU**: NVIDIA GeForce GTX 1070

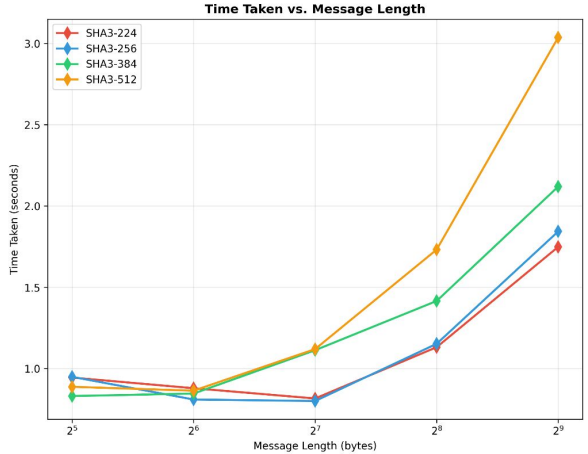
Primary metrics used:

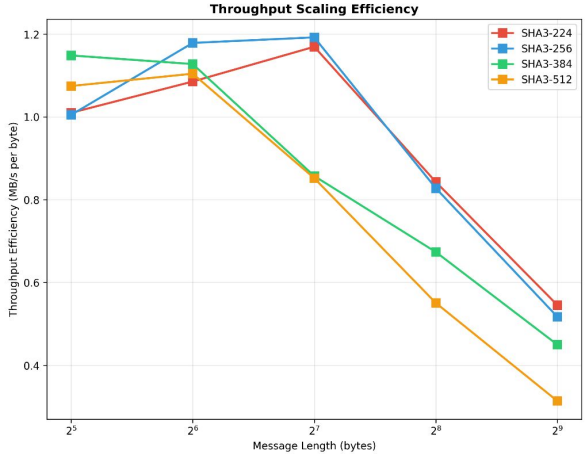
- **Throughput** (MB/s): The total amount of message data processed per second.
- **Time taken** (seconds): The raw execution time for the benchmark.
- **Speedup Factor** (x): How many times faster the GPU is compared to the CPU implementations.

CPU SHA-3 High-Volume Benchmark (Optimized)

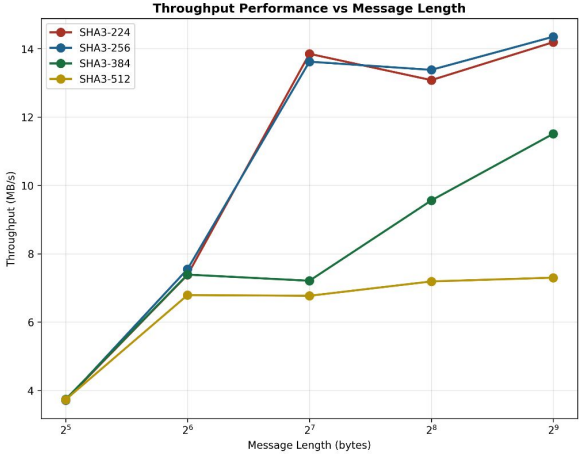


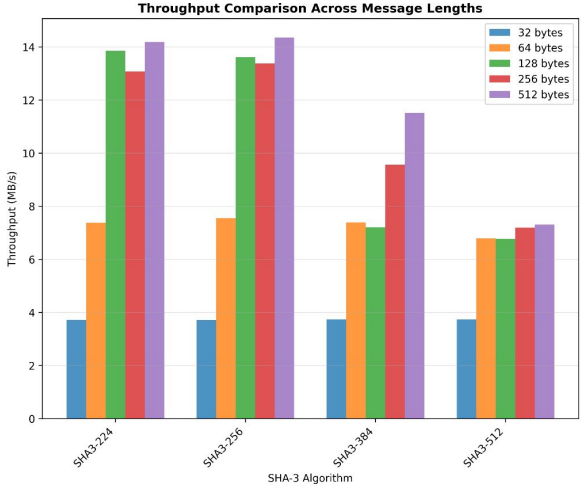


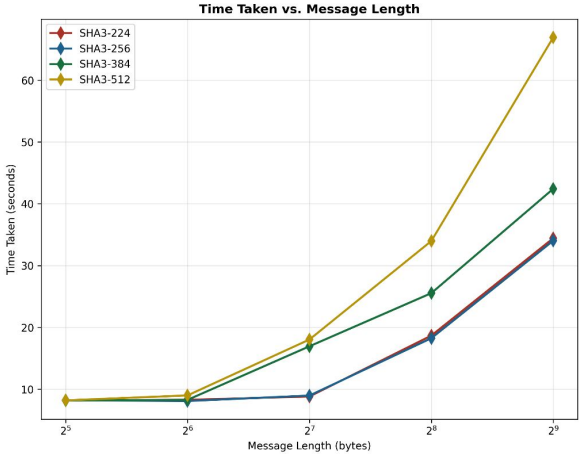


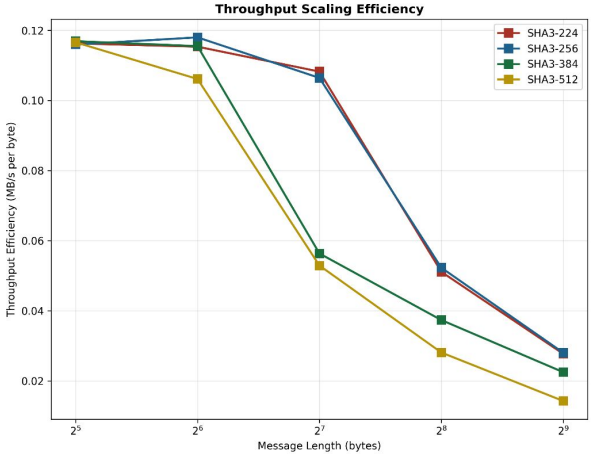


CPU SHA-3 High-Volume Benchmark (Unoptimized)

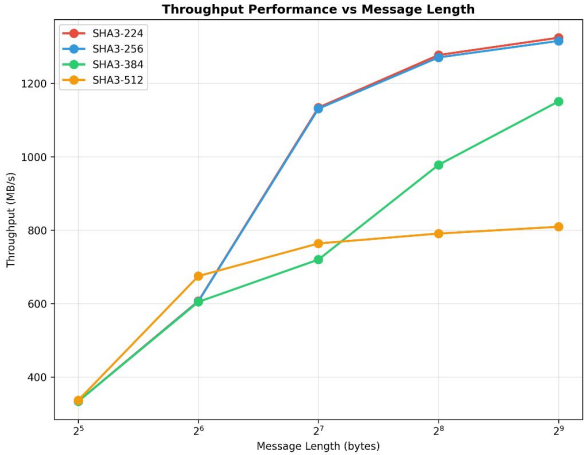


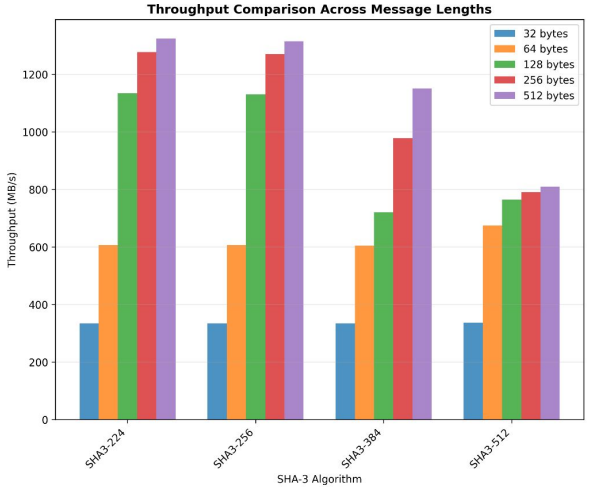


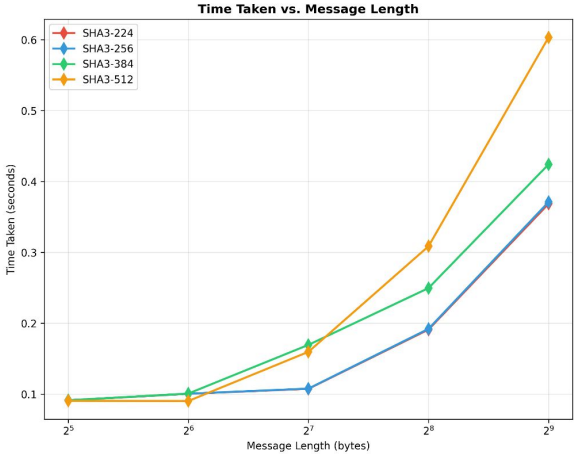


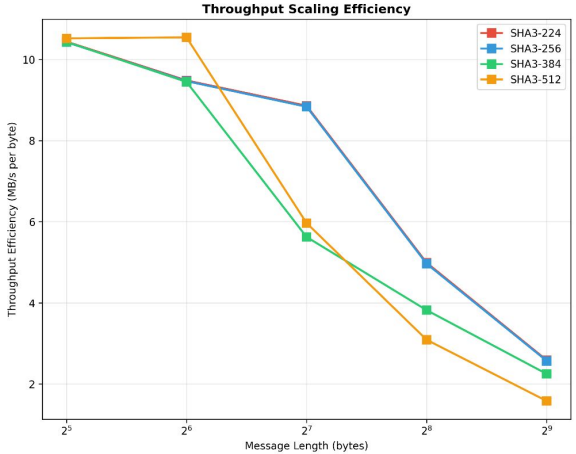


GPU SHA-3 High-Volume Benchmark Results

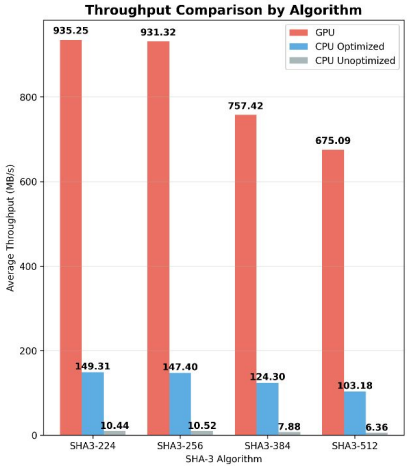


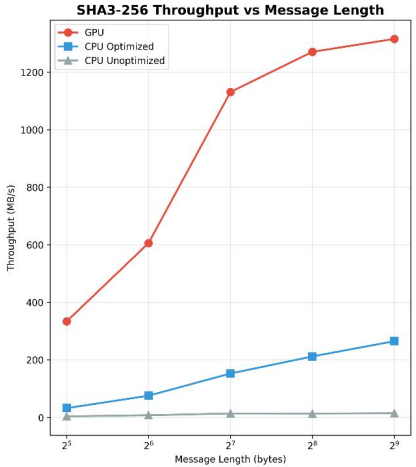


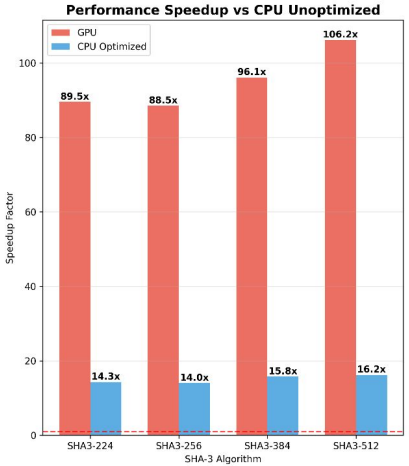


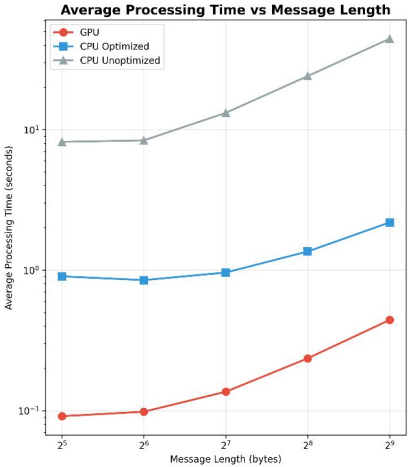


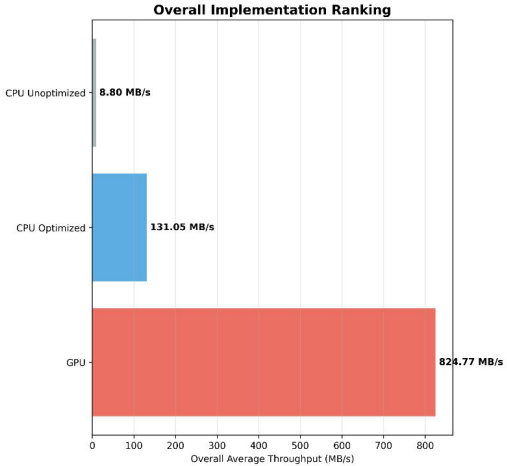
Comprehensive SHA-3 High-Volume Benchmark Comparison

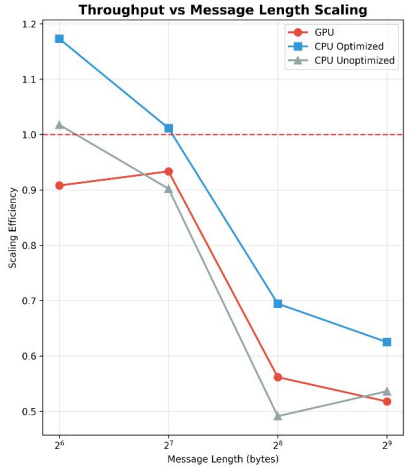




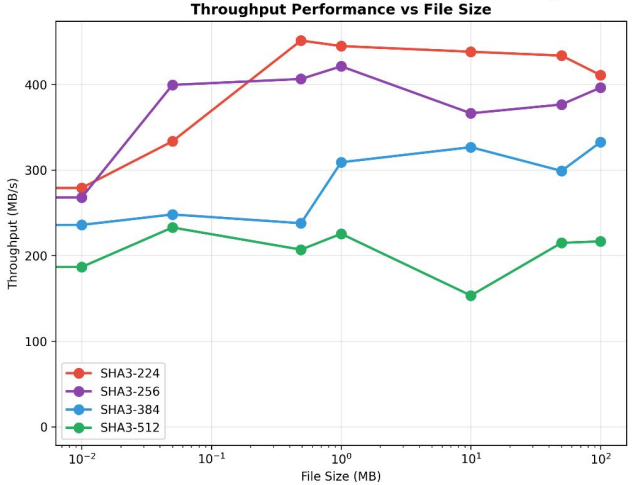


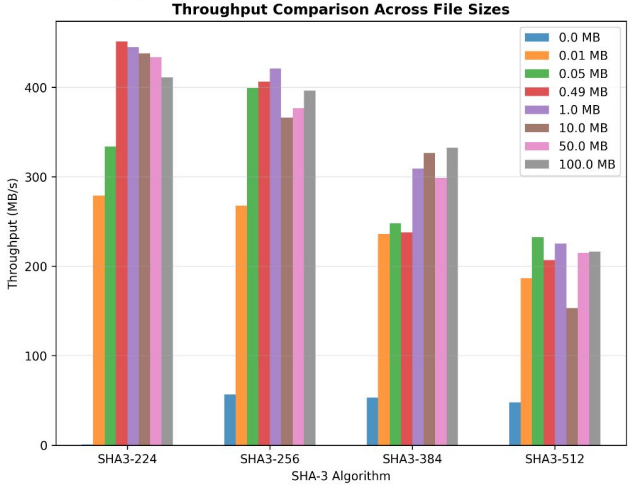


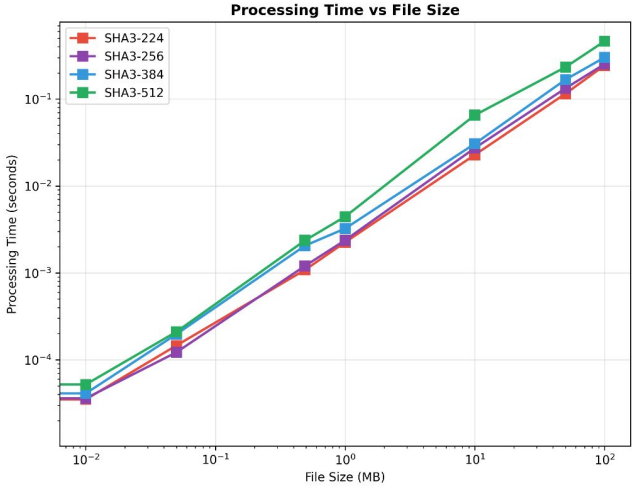


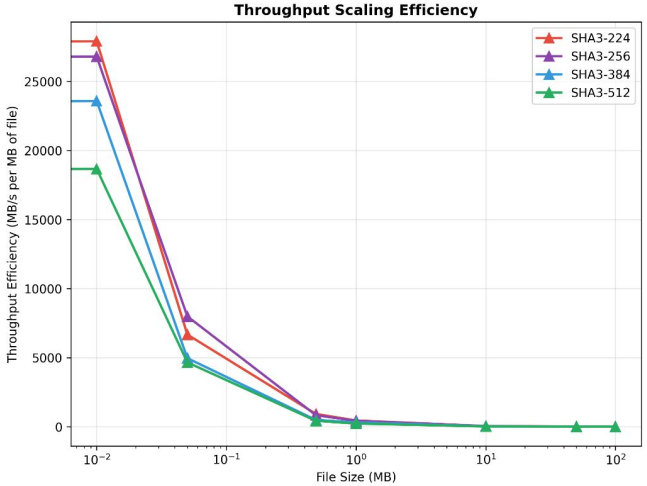


CPU SHA-3 Large File Benchmark (Optimized)

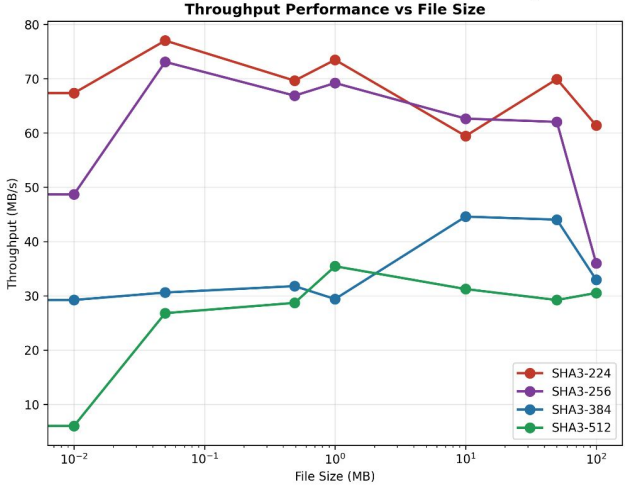


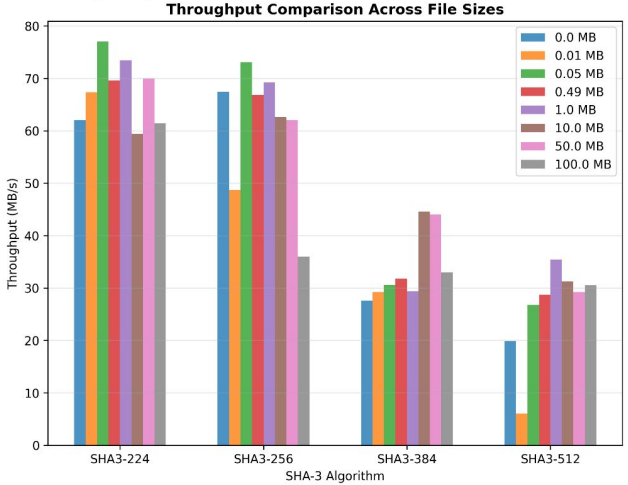


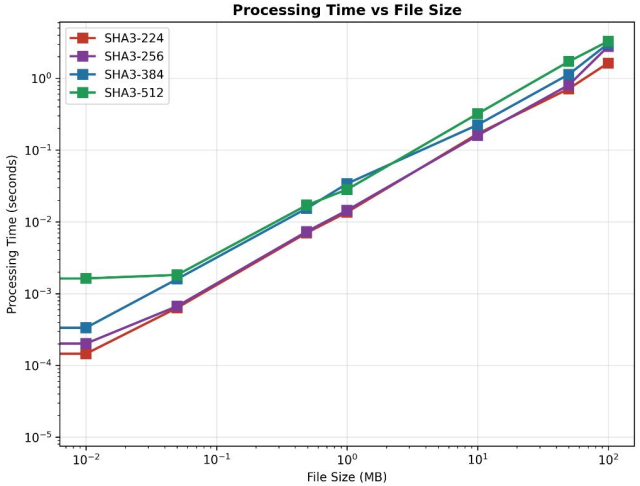


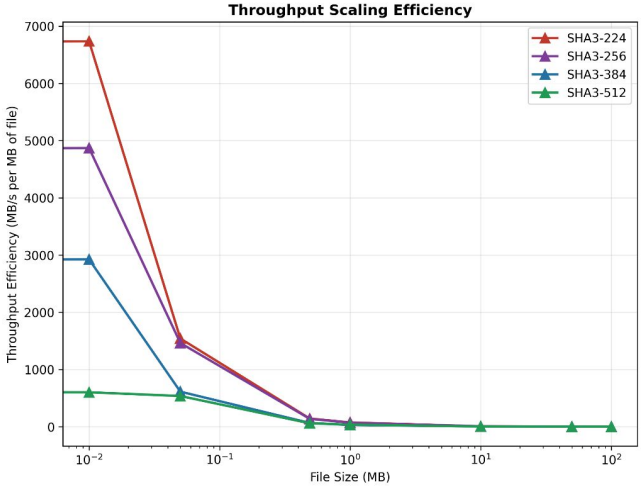


CPU SHA-3 Large File Benchmark (Unoptimized)

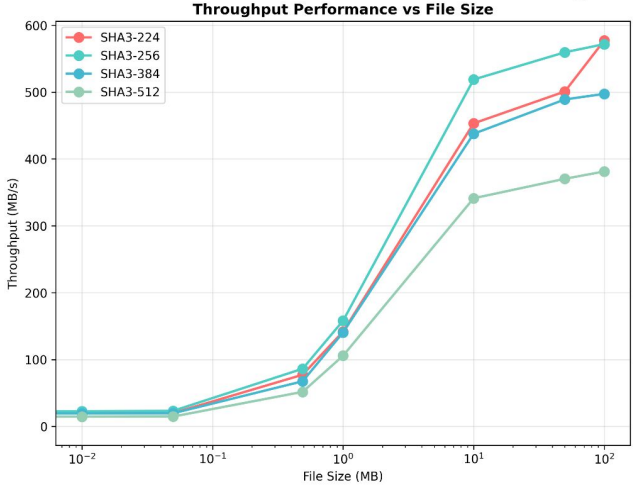


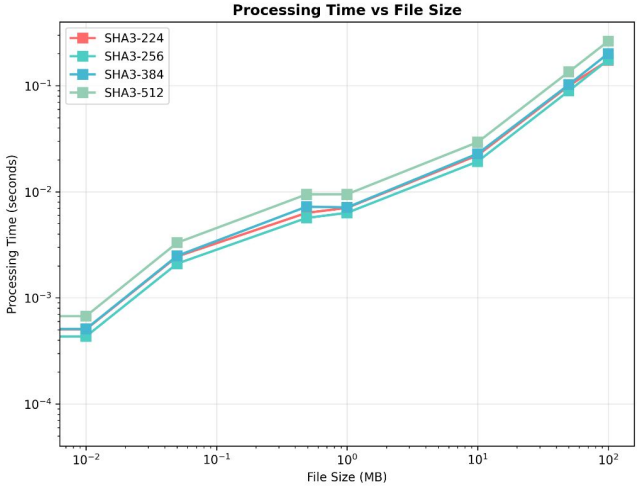


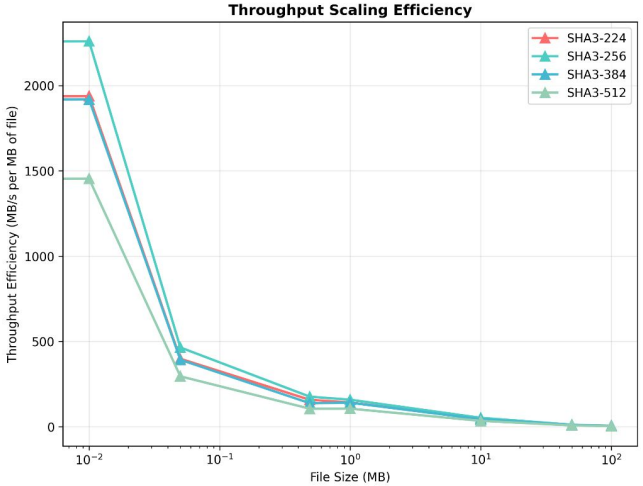




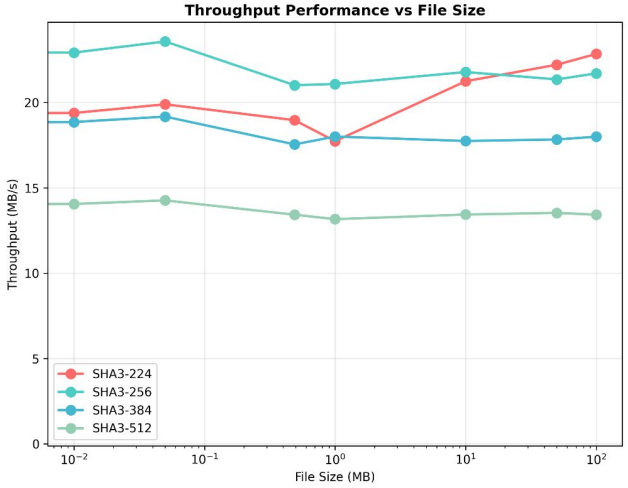
GPU SHA-3 Large File Benchmark Results

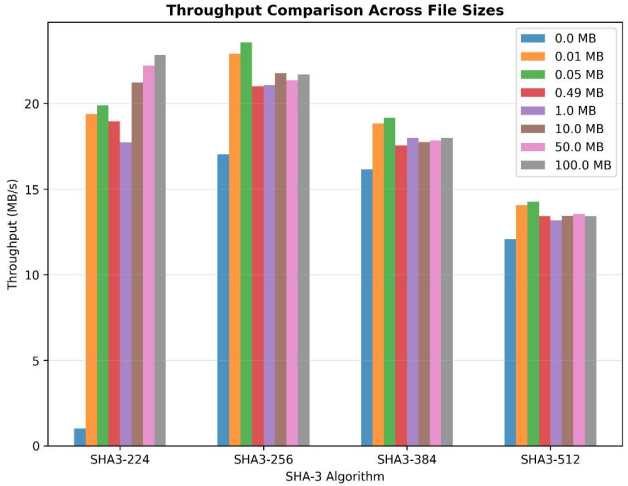


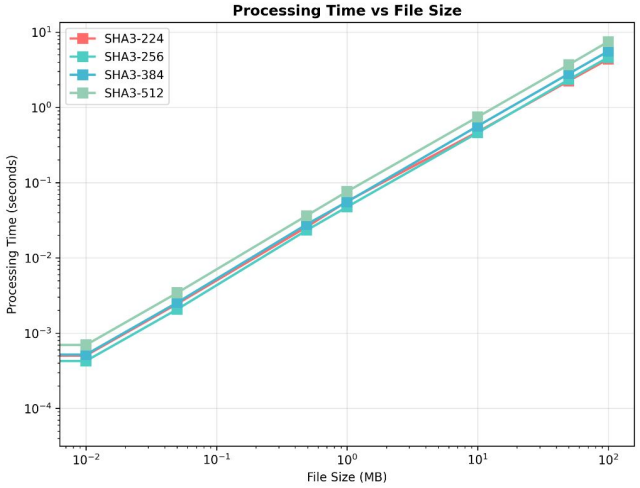


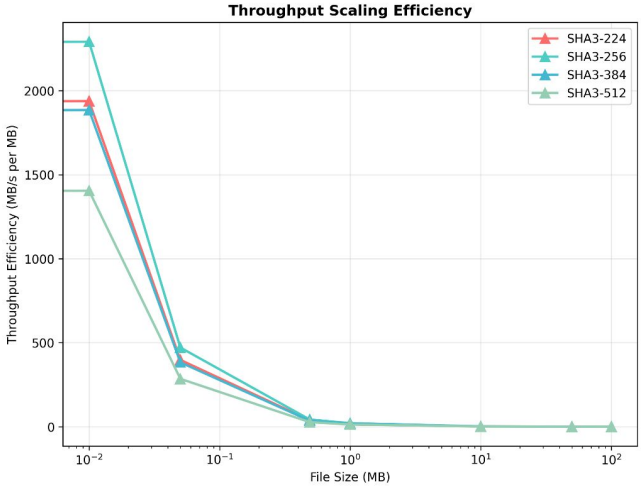


GPU SHA-3 Serial Benchmark Results

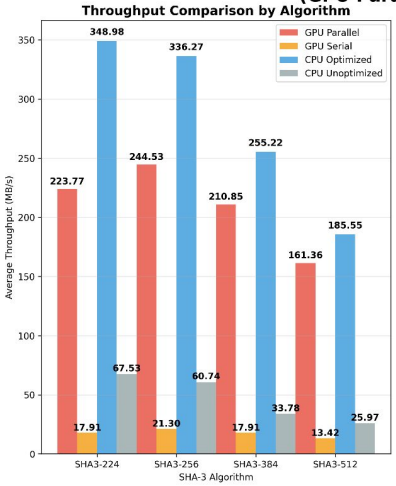


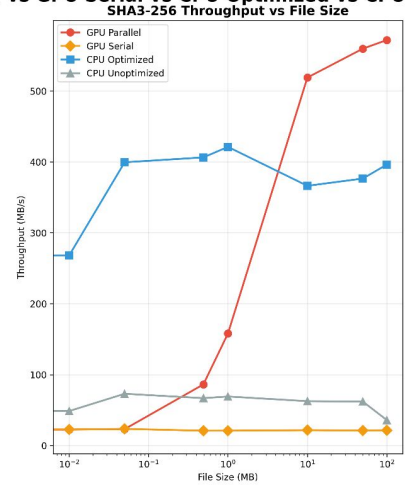


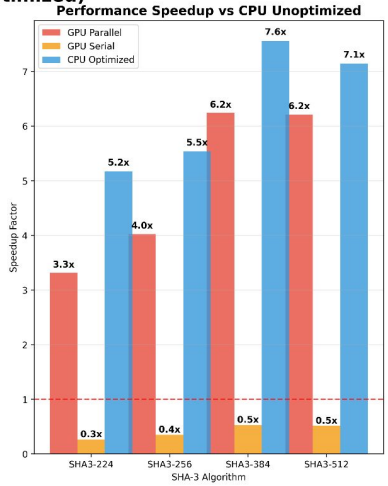


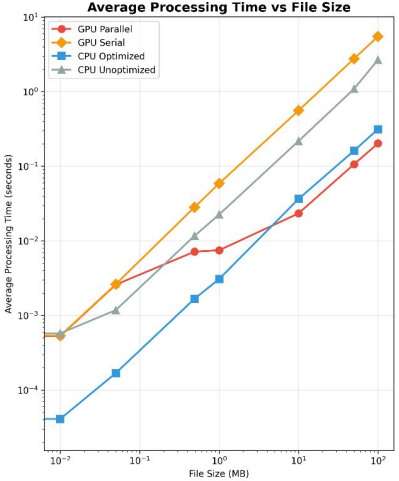


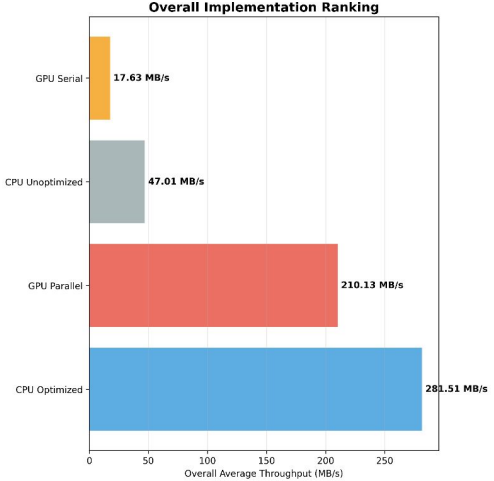
Comprehensive SHA-3 Large File Benchmark Comparison
(GPU Parallel vs GPU Serial vs CPU Optimized vs CPU Unoptimized)

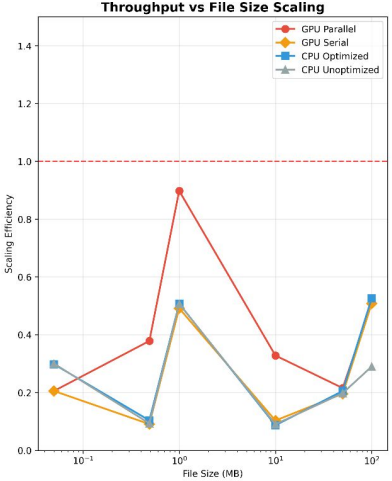












Result analysis

From these results we can say that the optimal hardware for SHA-3 hashing is entirely dependent on the nature of the workload. There is no single "best" architecture, the choice is a trade-off between serial speed and parallel throughput.

Serial Workloads:

The **Optimized CPU** is the clear winner. The OpenSSL implementation, with its high clock speed and use of **AVX** instructions, consistently outperforms both the GPU and the unoptimized CPU.

GPU performance is limited by **overhead**. For smaller files (<10 MB), the GPU was even slower than the unoptimized CPU due to the high latency of memory transfers and kernel launches. This overhead becomes less significant as file size increases.

Result analysis

From these results we can say that the optimal hardware for SHA-3 hashing is entirely dependent on the nature of the workload. There is no single "best" architecture, the choice is a trade-off between serial speed and parallel throughput.

Parallel Workloads:

The **GPU is the clear winner**. In this scenario, the GPU's massively parallel architecture provides a speedup of over 10x compared to the optimized CPU and over 100x compared to the unoptimized version.

This proves the GPU's superiority for tasks like password cracking, blockchain calculations, or handling thousands of simultaneous server requests.

Result analysis

From these results we can say that the optimal hardware for SHA-3 hashing is entirely dependent on the nature of the workload. There is no single "best" architecture, the choice is a trade-off between serial speed and parallel throughput.

Impact of optimization:

The performance gap between the **simple C++** implementation and the **optimized OpenSSL** library is **enormous**, highlighting that using a professionally optimized library is critical for any real-world application.

Bibliography

- C. Wang, X. Chu "**GPU Accelerated Keccak (SHA3) Algorithm**"
- W. Stallings, L. Brown, "**Computer Security: Principles and Practice**", Fourth edition, Pearson 2018
- A. Dolmeta, M. Martina, G.Masera, "**Comparative Study of Keccak SHA-3 Implementations**", Cryptography 2023, 7, 60
- G. Bertoni, J. Daemen, M. Peeters, G. Van Assche and R. Van Keer, "**Keccak implementation overview**", SHA-3 competition (round 3), 2012
- R. Boissier, C. Nous, Y. Rotella, "**Algebraic Collision Attacks on Keccak**", IACR Transactions on Symmetric Cryptology, 2021(1), 239-268


That's all Folks!