GPU Symmetric Key Encryption and Decryption

CSE 4059: Applied Parallel Programming using GPUs

Group Members

- Member 1: Preston Meek
- Member 2: Zichu Pan

1 Abstract

Encryption and decryption are key elements in modern cybersecurity. Networking protocols, Bluetooth devices, banking software, and more rely on modern encryption algorithms. Since data security and privacy are extremely important to the end user, much research has already been done to find optimal encryption algorithms. Two facets of an 'optimal' algorithm are low latency and high throughput. Low latency is vital for real-time data, while high throughput is desired for large data blocks.

This project aims to port an existing algorithm, Advanced Encryption Standard in Counter Mode (AES-CTR), to the CUDA architecture. Parallelism will allow the algorithm to run much faster than traditional CPU implementations by leveraging the massive computational power of modern GPUs. Specifically, this implementation will take advantage of CTR mode's inherent parallelism, where each thread processes a 16-byte input block independently. Optimizations such as shared memory usage, coalesced memory accesses, and thread coarsening will further enhance performance.

The primary objective of this project is to develop a CUDA-C implementation of AES-CTR that significantly outperforms CPU-based algorithms for large datasets. By maximizing throughput and occupancy, this project will demonstrate the practical benefits of GPU-accelerated encryption. The final results illustrate the efficiency gains achieved through utilizing parallel architecture and common optimization techniques.

2 Introduction

Advanced Encryption Standard (AES) is a widely adopted symmetric encryption algorithm that secures data by transforming readable information (plaintext) into an unintelligible format (ciphertext). As a block cipher, AES processes fixed-size blocks of data—specifically, 128 bits per block. When the input data exceeds this size, it is partitioned into multiple blocks for encryption.

2.1 Encryption Overview

Encryption involves three fundamental components:

- **Plaintext and Ciphertext**: Plaintext refers to the original human-readable payload that requires protection. Ciphertext is the encrypted version of the plaintext, rendered unintelligible to unauthorized individuals.
- **Encryption and Decryption Algorithms**: The encryption algorithm is a mathematical process that converts plaintext into ciphertext. A cryptographic key is used to introduce uniqueness to the algorithm. Prominent encryption algorithms include AES, RSA, and DES. Decryption is an inverse process that transforms ciphertext back into plaintext with the appropriate decryption key.
- **Cryptographic Key**: This is a string of bits utilized by the algorithm for both encryption and decryption. AES employs symmetric key encryption, meaning the same key is used for both processes.

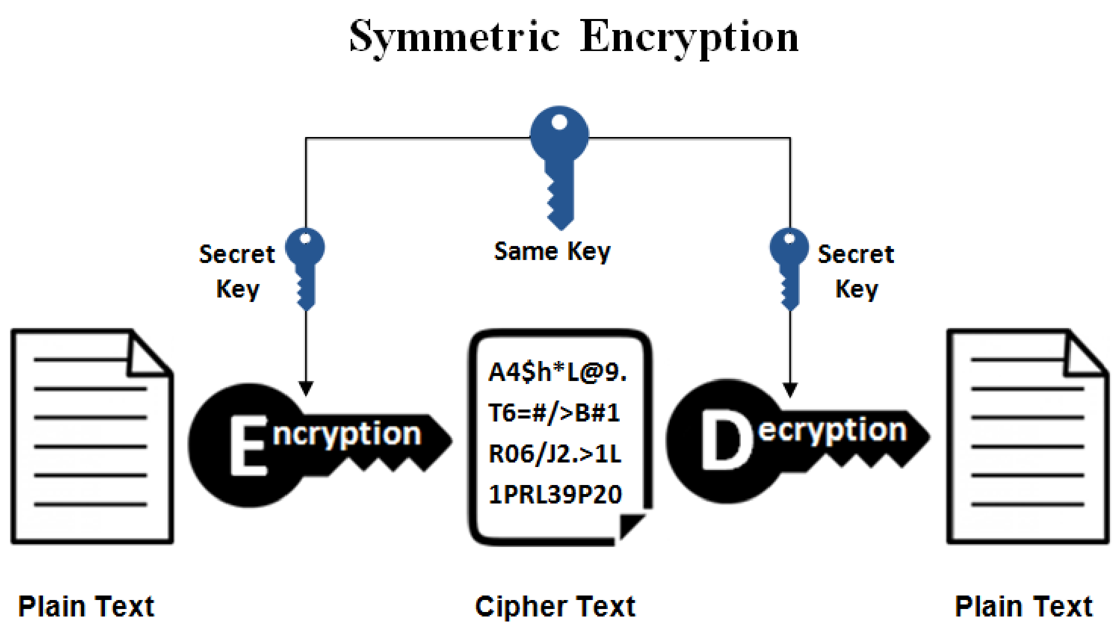

Figure 1: The symmetric key encryption and decryption process.

2.2 Advanced Encryption Standard (AES)

AES operates on 128-bit (16-byte) data blocks and supports 128-, 192-, and 256-bit keys. For example, AES-192 uses a 192-bit key. Note that regardless of key length, the block size is always 128-bit.

A longer key improves security but requires more rounds of mathematical operations. For example, AES-256 involves 14 rounds, while AES-128 only involves 10. Each encryption round comprises substitution, shifting, mixing, and the addition of a round key to thoroughly scramble the data. Decryption computes these operations in the opposite order to restore the original plaintext.

AES block ciphers support various modes of operation, including Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR) mode, each suited to different use cases. This paper focuses on the CTR mode of operation.

2.2.1 Counter Mode (CTR) Operation

In AES-CTR, encryption does not directly transform the plaintext. Instead, it generates a keystream by encrypting a unique counter value for each data block. The counter typically increments sequentially. The process involves:

- Encrypting the counter value with AES-CTR to produce a keystream block.
- XORing the keystream block with the plaintext to generate the ciphertext.

For AES-CTR specifically, the decryption process is identical to encryption: the same counter values are encrypted to reproduce the keystream. XORing this keystream with the ciphertext retrieves the original plaintext. This process relies on the following property:

$$A \oplus B \oplus B = A$$

where $\oplus$ denotes the XOR operation. Applying this to encryption and decryption results in the following:

$$Plaintext \oplus Keystream = Ciphertext$$

$$Ciphertext \oplus Keystream = Plaintext$$

AES-CTR operates as a stream cipher, meaning each block is processed independently. In contrast, modes like CFB introduce inter-block dependencies, where each block depends on the output of the previous one. The independence of blocks in CTR mode enables both encryption and decryption to be performed in parallel, making AES-CTR particularly well-suited for parallel-processing architectures such as GPUs.

3 Algorithms

The AES algorithm consists of multiple rounds of transformations—SubBytes, ShiftRows, MixColumns, and AddRoundKey—which are applied to the plaintext block in a well-defined sequence based on the key size.

CTR mode modifies the typical AES process by encrypting a counter value rather than the plaintext directly. Each 128-bit counter value is formed by concatenating a unique number with a block index. This counter is then encrypted using AES to produce a keystream block. The keystream block is XORed with the corresponding plaintext block to yield the ciphertext. Decryption uses the same keystream, reversing the operation via XOR.

In the CUDA implementation, each thread will be assigned a 16-byte block of plaintext. Plaintext blocks will be loaded into shared memory to reduce global memory access latency. After encrypting the counter and producing the keystream, each thread will perform an XOR operation with the corresponding plaintext to generate the ciphertext. An additional optimization step will efficiently combine the encrypted blocks into the final output buffer.

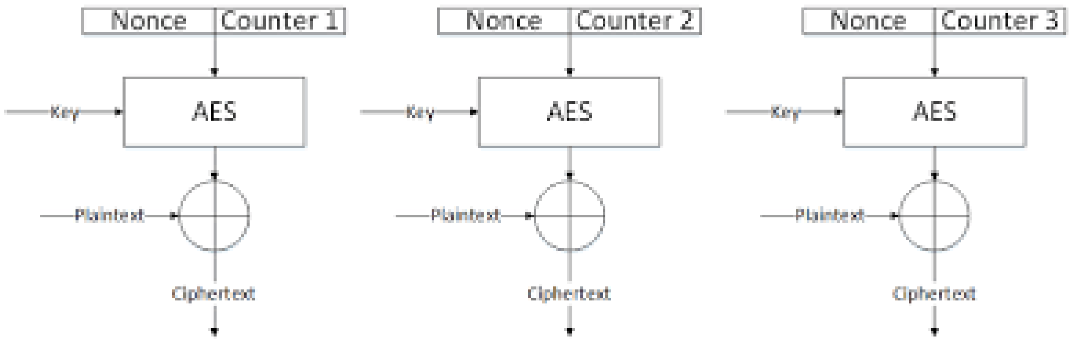

Figure 2: Block Diagram of AES-CTR Encryption

The diagram above illustrates how the counter values are generated, encrypted, and XORed with the plaintext blocks to form ciphertext. Each step is fully parallelizable across GPU threads.

Pseudocode for AES-CTR Encryption:

```
for each block index in total block:
    counter = unique_num + i
    keystream = AES_encrypt(counter, key)
    ciphertext[i] = plaintext[i] XOR keystream
```

The function `AES_encrypt(counter, key)` performs the core AES block cipher encryption on the 128-bit counter value using the key. The steps involved in AES-128 encryption include:

1. **Key Expansion**: Expand the original key into a key schedule consisting of 15 round keys (14 rounds + initial key) [slicing the key in to smaller keys].
2. **Initial AddRoundKey**: XOR the input block (counter) with the first round key.
3. **Rounds (1 to 13)**:
    - **SubBytes**: Replace each byte using a substitution table
    - **ShiftRows**: Circularly shift the rows of the state matrix.
    - **MixColumns**: Transform each column using a linear mixing operation.
    - **AddRoundKey**: XOR the result with the corresponding round key.
4. **Final Round (Round 14)**:
    - **SubBytes**
    - **ShiftRows**
    - **AddRoundKey**

5. **Output**: The resulting 128-bit block is returned as the keystream block.

These operations are applied to a 4x4 matrix representation of the input block. The use of a fixed number of well-defined rounds and deterministic key scheduling makes AES efficient and secure for our GPU implementation.

4 Problem Statement

Modern cryptographic workloads require processing large volumes of data with minimal latency. Current CPU-based implementations of AES-CTR face performance bottlenecks when encrypting and decrypting large datasets. For certain applications, such as live video encryption and large-scale cloud storage encryption, parallel processing offers significant speedup by enabling simultaneous block operations. This project addresses these limitations by leveraging the massive parallelism available in modern GPUs to accelerate AES-CTR encryption and decryption operations.

5 Approach and Implementation

Before implementing a parallelized, GPU-compatible AES-CTR algorithm, a sequential, CPU-based program was developed. This was to ensure that the base functionality of the algorithm was in-place before implementing complex parallel optimizations.

For this project, AES-128 CTR in particular was implemented. While adding support for other key sizes would not be particularly difficult, the focus of the project is on optimizations through parallelism. Therefore, the key size should be consistent across implementations.

5.1 Sequential CPU Implementation

There are a large number of existing open-source AES-128 CTR implementations. Additionally, especially for sequential programs, there would be extremely minimal (if any) latency reduction from writing a new implementation: CTR mode is quite simple, and there is not much room for optimization within the algorithm itself. Rather, the optimization will come from how the algorithm is used. Therefore, the TinyAES library was used as a basis for the sequential CPU implementation [1].

Instead of solely using the high-level API provided by TinyAES, internal library functions were exposed to allow for extra control over the functionality and execution of the algorithm. The sequential CPU program was designed to allow for different implementations (i.e., parallel) to be easily added while minimally changing the program structure.

5.2 Parallel CPU Implementation

Before implementing a CUDA version of AES-128 CTR, a parallelized CPU version was created. This is primarily for benchmarking purposes as both the parallel CPU and GPU implementations are based on the sequential CPU version. Originally, this version of the algorithm was implemented by using the `fork()` syscall to create child processes.

Each child process performed a given amount of work determined by a pre-defined coarsening factor. However, child processes are expensive as shared memory must be used (via `mmap()`) since each process has their own copy-on-write address space. Therefore, CPU threads were used instead via the `pthread_create()` library function. Threads are lightweight since they use the same address space as the parent process. Note that regardless of implementation, a rather large coarsening factor has to be used since there is a much smaller number of available CPU threads than there are GPU threads, and `pthread_create()` will fail if too many threads have already been created.

5.3 Parallel GPU Implementation

The parallel GPU implementation is designed to run on WashU Linux Lab Academic Desktop GPUs, which have the following specifications that influenced the optimization of our implementation:

- Shared Memory per Block: 49,152 bytes
- Registers per Block: 65,536
- Warp Size: 32
- Max Threads per Block: 1024
- Max Grid Size: [2147483647, 65535, 65535]
- Memory Bus Width: 160 bits

5.3.1 Base Implementation

The base CUDA kernel (`aes_ctr_kernel`) launches one thread per input block. Each thread:

1. Generates a unique counter value using the nonce and block index (`get_counter()`).
2. Encrypts the counter using the AES round key schedule via the `Cipher_device()` routine.
3. XORs the encrypted counter with the input plaintext block to produce the ciphertext.

This approach avoids inter-thread dependencies entirely, maximizing potential parallelism.

The AES key expansion is done on the host using `KeyExpansion()`, and the expanded round key is copied to the device. Memory transfers are batched for input, output, key, and nonce using standard `cudaMemcpy` calls.

5.3.2 Determine Thread Count per Block

To determine the optimal configuration, experiments were conducted using varying thread counts (32, 64, 128, 256, 512, and 1024) across multiple file sizes ranging from 1MB to 2GB. The performance was evaluated based on encryption time and throughput in $\frac{MB}{s}$.

Figure 3 shows that all lower thread counts except the maximum thread-per-block configuration exhibit logarithmic throughput growth as file size increases. While performance improves with input size, the rate of improvement slows down, indicating underutilization of the GPU's parallel resources.

In contrast, using the maximum of 1024 threads per block shows near-linear or even superlinear scaling of throughput. With 1024 threads per block, the throughput scales directly with file size—reaching 100,000 $\frac{MB}{s}$ for a 2GB file. This linear growth suggests that the GPU has achieved full occupancy and operating at maximum efficiency, taking advantage of all available multiprocessors and effectively reducing memory latency.

It is important to recognize that using too many threads per block is not always beneficial. Excessively large thread blocks can increase register pressure, potentially reducing the number of blocks that can run concurrently on an SM. In our AES-CTR implementation, the kernel is relatively lightweight in terms of register usage. As a result, using the maximum thread count of 1024 per block does not exhaust available resources, allowing for maximum occupancy and full utilization of the GPU.

5.3.3 Coarsening

Thread coarsening is an optimization technique where each GPU thread is assigned multiple units of work rather than just one. In our implementation, each thread processes multiple 16-byte blocks, thereby reducing the overhead of the kernel launch and thread scheduling. It also helps to better amortize the cost of AES key expansion and counter setup over a larger amount of work per thread.

In the implemented coarsened kernel, each thread is assigned `blocks_per_thread` AES blocks to encrypt. This allows a reduction in the total number of threads required to process large inputs, which can lead to improved instruction-level parallelism and better cache utilization.

To determine the optimal coarsening factor, the encryption throughput was benchmarked over a range of values: `1`, `2`, `4`, `8`, and `16`. Each was tested with a 500MB file and 1024 thread per block. Below are the results:

Blocks per thread	Encryption Time(ms)
1	33.57
2	24.299
4	24.192
8	26.988
16	25.964

From the results, a coarsening factor of **4** achieves the lowest encryption time, closely followed by factor 2. This indicates that a moderate level of coarsening offers the best trade-off: it reduces scheduling overhead while avoiding the diminishing returns and increased register pressure at higher coarsening levels. Based on the benchmark data, a coarsening factor of **4** was chosen for the final implementation.

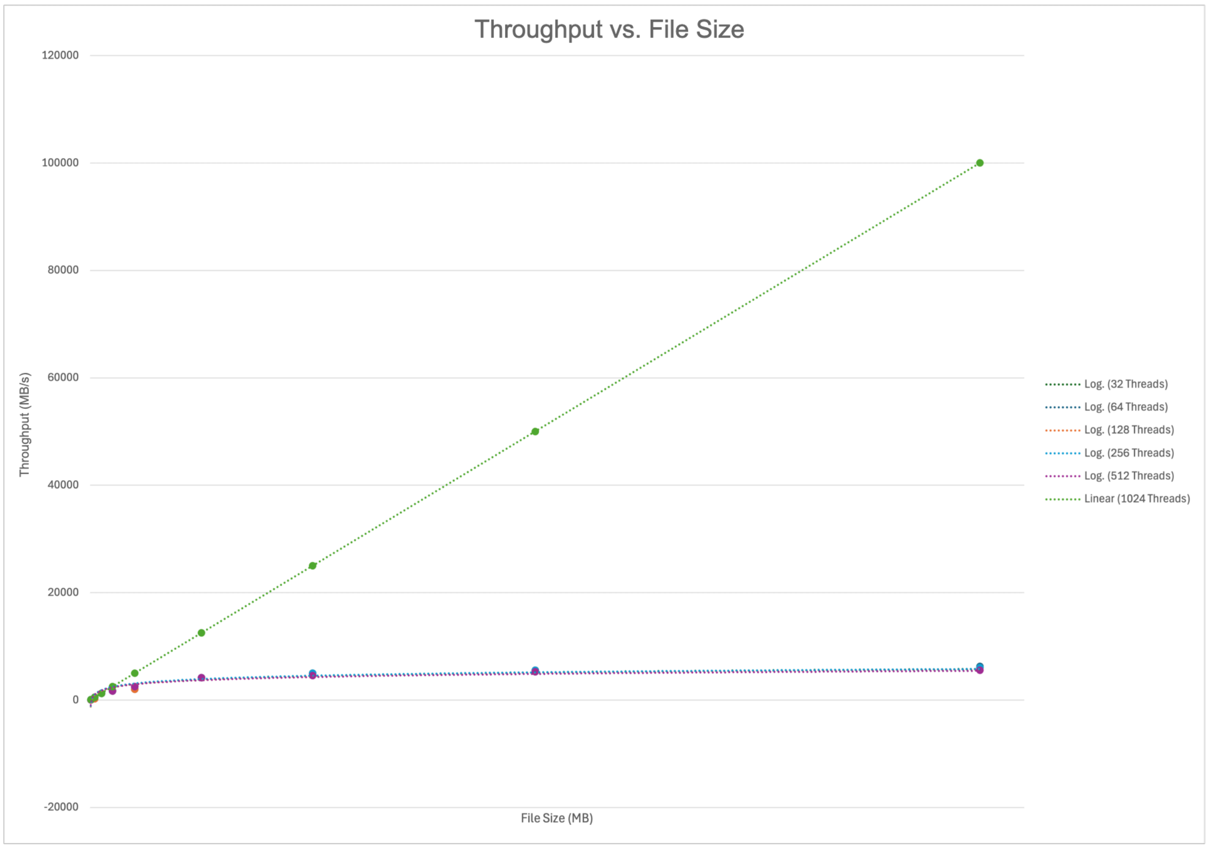

Figure 3: Throughput ($\frac{MB}{s}$) vs. File Size for Varying Thread Counts Per Block. The 1024-thread configuration shows near-linear scaling, indicating optimal GPU utilization, while lower thread counts demonstrate logarithmic performance growth due to underutilization.

5.3.4 Shared Memory

To improve memory access latency and reuse of read-only data, shared memory was introduced into the AES-CTR GPU kernel, specifically for storing the AES expanded round key. In AES encryption, the round key is reused by every thread during the encryption of each block. Without shared memory, each thread independently accesses the round key from global memory, resulting in redundant and slower memory transactions.

To eliminate this inefficiency, the expanded round key was copied once into shared memory per thread block using a cooperative loading pattern. A portion of threads within the block load segments of the key in parallel using strided access.

After this setup, all threads in the block use the shared memory copy of the round key for encryption. The use of `__syncthreads()` ensures that all round key data is fully loaded before any thread proceeds with encryption, preserving correctness.

File Size (MB)	Time (No SM)	Time (SM)
1	41.186 ms	36.476 ms
20	30.422 ms	32.760 ms
25	285.706 ms	31.784 ms
50	32.717 ms	24.648 ms
100	27.551 ms	111.802 ms
250	35.280 ms	29.126 ms
500	30.800 ms	34.340 ms
1000	28.082 ms	35.279 ms
2000	35.700 ms	40.696 ms

For small and medium file sizes (especially 25MB and 50MB), using shared memory results in significant performance gains—up to **9.3**× improvement. This is because the AES round key, which is reused across all blocks, benefits from faster access in shared memory compared to global memory. However, for large input sizes (1000MB and 2000MB), shared memory introduces minor overhead or even performance regression, likely due to shared memory saturation or insufficient benefit compared to memory coalescing already happening in global memory.

5.3.5 Coalescing

In the original AES-CTR GPU kernel, each thread processed an AES block (16 bytes) using a loop to copy and XOR individual bytes:

```
for (int j = 0; j < 16; ++j) {
    output[block_idx * 16 + j] = input[block_idx * 16 + j] ^ counter[j];
}
```

This approach results in 16 separate 1-byte memory accesses per thread, which degrades performance due to uncoalesced global memory transactions. To improve memory throughput, the kernel was rewritten to use vectorized 128-bit loads and stores via the `uint4` type:

```
const uint4* input_block = (const uint4*)&input[block_idx * 16];
uint4* output_block = (uint4*)&output[block_idx * 16];
const uint4* counter_block = (const uint4*)counter;

output_block[0].x = input_block[0].x ^ counter_block[0].x;
output_block[0].y = input_block[0].y ^ counter_block[0].y;
output_block[0].z = input_block[0].z ^ counter_block[0].z;
output_block[0].w = input_block[0].w ^ counter_block[0].w;
```

This ensures that each thread performs a single 16-byte aligned global memory read and write, which allows the GPU to fully coalesce memory transactions across threads in a warp. Since AES blocks are 16 bytes and memory is allocated with `cudaMalloc()`, alignment constraints are naturally satisfied.

Two benchmark sets were collected to evaluate the effect of coalesced memory access: one with the original (uncoalesced) kernel and one using the optimized coalesced version. All tests used 1024 threads per block and a coarsening factor of 4.

File Size (MB)	Uncoalesced	Coalesced	Improvement
1	25.00	9.09	↓ 64%
20	666.66	666.66	No change
25	89.28	1250.00	↑ 13.9×
50	1666.66	2500.00	↑ 50%
100	5000.00	5000.00	No change
250	8333.33	12500.00	↑ 50%
500	16666.66	16666.66	No change
1000	50000.00	33333.33	↓ 33%
2000	66666.66	66666.66	No change

For smaller file sizes (25–250MB), coalesced access provides significant performance improvement—up to a **13× speedup at 25MB**. For mid-sized inputs (50–250MB), throughput improved by 50% as memory coalescing reduced redundant memory transactions. For very large files (500MB+), the effect is less pronounced. Throughput plateaus or even slightly drops due to already saturated memory bandwidth or other bottlenecks (e.g., instruction dispatch, kernel overhead). At very small sizes (1MB), the coalesced version may perform worse due to kernel launch overhead outweighing memory efficiency gains.

5.4 Verification of Correctness

As the previous section mentioned in CTR mode, encryption and decryption are symmetric operations. That is, applying the encryption function twice with the same key and nonce should yield the original plaintext. This makes the verification for correctness as simple as `diff original.txt double_encrypted.txt`.

6 Results

For this project, throughput is the primary performance benchmark. It ties execution time directly to the input file size, providing a fair measurement of the encryption algorithm's speed.

A custom shell script was used to generate input files with random content via the following command:

```
dd if=/dev/urandom of=$FILE bs=1M count=$SIZE status=progress
```

where `$FILE` is the path to the output file and `$SIZE` is the size of the file in megabytes. The following file sizes were used for testing: { 1MB, 10MB, 25MB, 50MB, 100MB, 250MB, 500MB, 1000MB, 2000MB }.

A custom shell script was used to run each implementation of the algorithm on each input file and benchmark the performance. Note that all CPU tests were ran on Linux Lab eight-core sessions for consistency and all GPU test were ran on Linux LAB GPU. The results are summarized below:

Table 1: Throughput metrics for CPU and GPU implementations.

Type	Average Throughput (MB/s)	Minimum Throughput (MB/s)	Maximum Throughput (MB/s)	Coefficient of Variation (MB/s)
Sequential CPU	5.384	1.565	6.206	0.279
Parallel CPU (coarse=4096)	14.979	1.953	18.709	0.369
Parallel CPU (coarse=8192)	12.339	1.828	15.043	0.345
Parallel CPU (coarse=16384)	14.830	1.812	18.595	0.373
GPU (no optimizations)	2671.588	33.330	5714.280	0.817
GPU (thread=1024)	14305.356	42.208	53166.038	1.362
GPU (thread=1024 coarse=4)	14925.488	28.673	60177.524	1.413
GPU (thread=1024 coarse=4 shared)	14757.622	34.026	64162.202	1.455
GPU (thread=1024 coarse=4 shared coal)	15388.352	8.236	62908.908	1.417

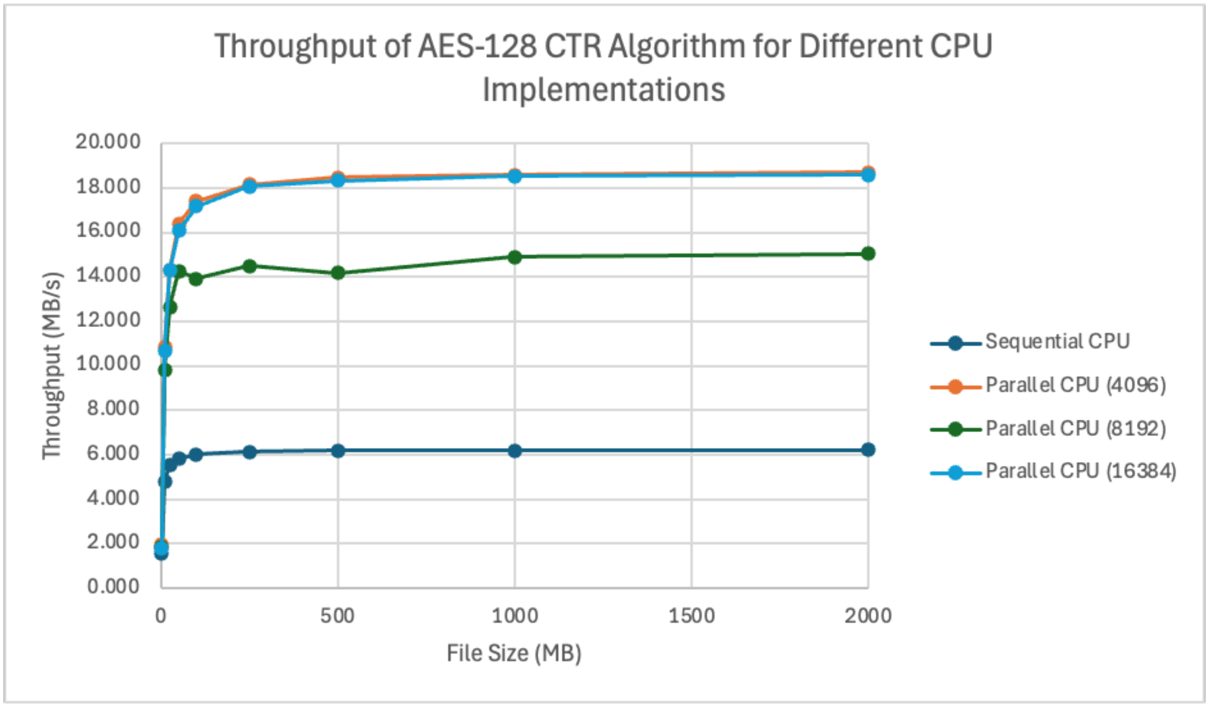

Figure 4: Linearly-scaled graph of AES-128 CTR throughput for different CPU implementations. Note that the values in parentheses in the legend represent coarsening factors. For example, 4096 means each thread processed 4096 128-bit blocks (65536 bytes of data per thread).

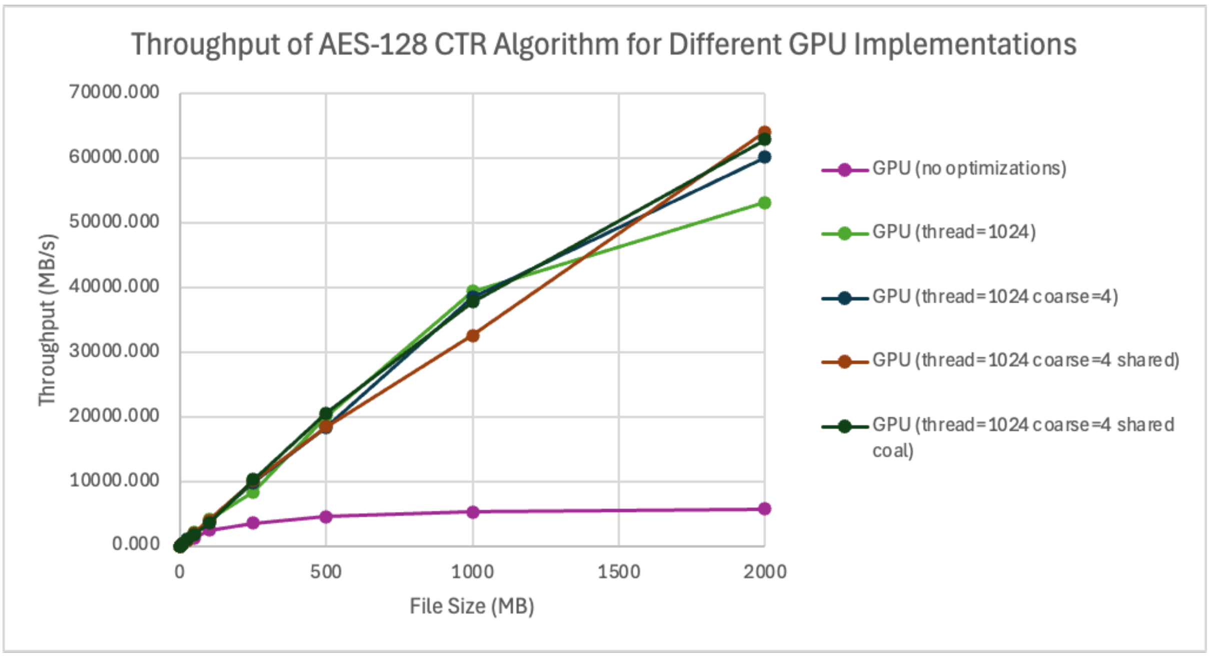

Figure 5: Linearly-scaled graph of AES-128 CTR throughput for different GPU implementations.

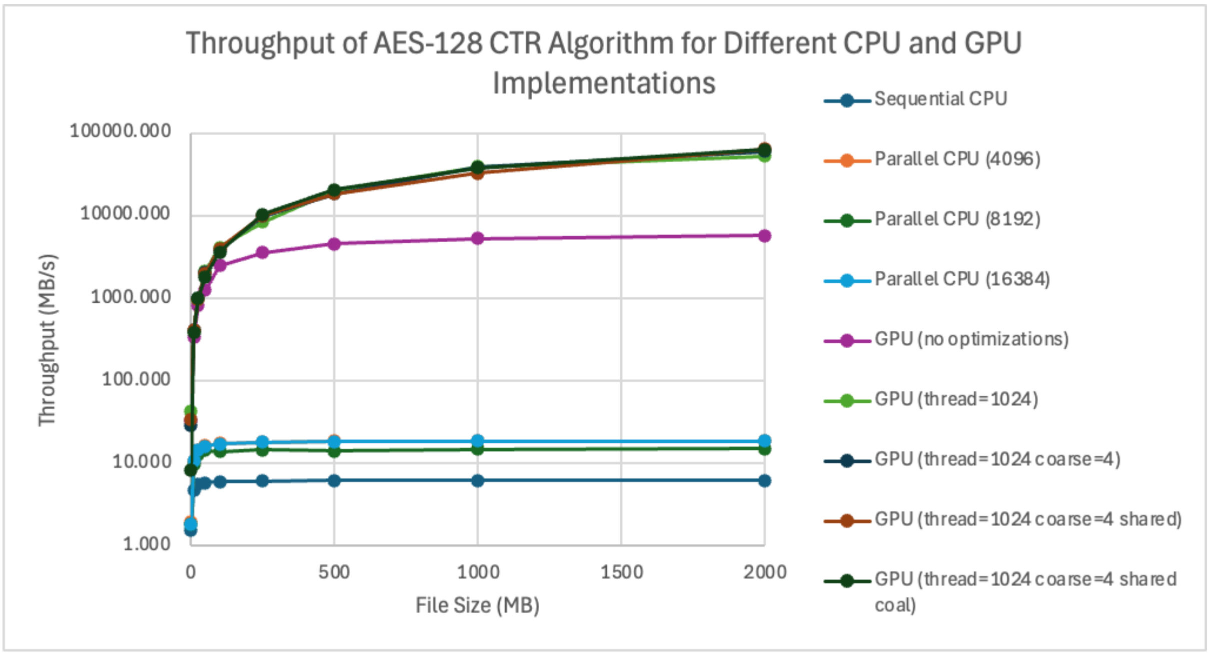

Figure 6: Logarithmically-scaled graph of AES-128 CTR throughput for different CPU and GPU implementations.

7 Discussion

There were significant performance benefits from implementing the AES-128 CTR algorithm on GPU architecture compared to the CPU. Even without any GPU-specific optimizations, there were over 300× gains in throughput for large input files. This clearly demonstrates the incredible potential of GPU-based encryption with stream ciphers.

The performance was further enhanced by applying several GPU-based optimizations, including:

- **Thread Coarsening**: Each thread was assigned multiple AES blocks to process, reducing scheduling overhead and maximizing instruction-level parallelism.
- **Memory Coalescing**: Global memory reads and writes were aligned using 128-bit vector types (`uint4`), allowing threads in a warp to access contiguous memory regions, minimizing memory transactions.
- **Shared Memory**: The AES round key was cached in shared memory, improving reuse across threads and reducing global memory load.

With these optimizations, throughput increased substantially — reaching up to **66,666 MB/s** on the GPU. These improvements show that proper alignment with GPU hardware paradigms can significantly elevate the performance of cryptographic workloads.

These results suggest that systems requiring high-speed encryption—such as secure cloud storage or real-time data transmission—could see substantial improvements by offloading cryptographic workloads to GPUs.

7.1 Benchmarking

CPUs with AES-specific instructions (AES-NI) take around 1.3 cycles per byte [3]. To convert this to throughput on a 3.0 GHz CPU, with 4 cores dedicated to the encryption algorithm:

$$\frac{1\ B}{1.3\ cycles} * \frac{1\ MB}{1024 * 1024\ B} * \frac{3.0 * 10^9\ cycles}{s} * 4\ cores \approx 8800\ \frac{MB}{s}$$

However, this is based purely on theoretical calculations. An open-source AES-NI implementation achieved a maximum throughput of around 730 $\frac{MB}{s}$, which is significantly less than the GPU implementations' throughputs, even without optimizations [2].

When comparing this to the optimized GPU implementation, the contrast becomes even more significant. With optimizations such as thread coarsening, memory coalescing, and in some cases shared memory usage, the GPU implementation achieved throughput as high as **66,666 MB/s** on large files. This is nearly **90× faster** than the real-world AES-NI performance of 730 $\frac{MB}{s}$ on CPUs, and even substantially exceeds the theoretical maximum of 8800 $\frac{MB}{s}$ calculated for a 4-core, 3.0 GHz CPU with full AES-NI utilization.

This stark performance gap highlights a key distinction: while CPUs rely on specialized AES hardware instructions (AES-NI) to achieve acceptable performance, GPUs deliver massive parallelism through general-purpose cores without any AES-specific circuitry. The flexibility of CUDA programming allows cryptographic algorithms to be fine-tuned to the hardware's strengths, such as warp-level parallelism and memory coalescing.

If GPUs were to include dedicated AES acceleration hardware—analogous to AES-NI on CPUs—the already superior throughput would likely increase even further, potentially by an order of magnitude. This would make GPUs not only a flexible platform but also the fastest option for high-throughput cryptography.

The results demonstrate that even general-purpose GPU hardware, when properly optimized, can surpass CPU-based AES encryption by a wide margin. This makes GPUs an extremely attractive option for scalable, high-speed encryption in both cloud and edge environments.

Additionally, an excerpt from *GPU Gems 3* by NVIDIA details a basic AES-CTR implementation with CUDA [4]. Interestingly, this implementation has higher throughput than this project's implementation for smaller file sizes. However, beyond 1MB inputs, this project has over 10× performance gains, while the paper's implementation plateaus at only 4MB. This illustrates that the optimizations introduced significantly improve throughput for large input sizes.

8 Conclusion

8.1 Challenges

Several technical and practical challenges were encountered throughout the course of this project:

- **Algorithm Portability:** Adapting the CPU-based AES-CTR algorithm to run efficiently on the GPU required significant restructuring, especially since GPU can only call device functions. This resulted in considerable rewrite of the encryption library used.
- **Memory Layout and Coalescing:** The default AES structure accessed memory in a way that could lead to uncoalesced reads and writes, severely limiting performance. Rewriting the kernel to use 128-bit vector types (`uint4`) required strict memory alignment and testing.
- **Thread Coarsening Tradeoffs:** Selecting the right coarsening factor involved balancing register pressure, occupancy, and kernel launch overhead. High coarsening factors increased throughput up to a point but eventually caused diminishing returns due to increased register usage and reduced warp scheduling flexibility.
- **Shared Memory Limitations:** While shared memory provided performance improvements in some cases, its limited size and potential for contention created bottlenecks for larger input sizes.

Despite these challenges, careful optimization and tuning led to an implementation that was robust, secure, and high-performing across a variety of input sizes.

8.2 Future Work

Future work includes exploring practical applications of this approach, particularly in scenarios that require high-throughput, real-time encryption. For example, 4K-and 8K—video resolutions are becoming increasingly common. These resolutions require significantly higher data rates compared to 1080p content. By leveraging the parallelism of GPU-based AES-128 CTR encryption, high-resolution video streams could be encrypted in real time, enabling secure, low-latency, high-quality streaming. A demonstration of this would highlight the algorithm's practical utility and scalability beyond controlled benchmarking environments.

Additionally, future iterations of this project should include a deeper comparison with existing GPU-accelerated encryption libraries such as NVIDIA's cuCrypto, OpenSSL's GPU backends (if available), or third-party implementations like those based on CUDA Thrust or OpenCL. Such comparisons would provide deeper insights into how our custom implementation stacks up in terms of performance, flexibility, and portability. Unfortunately, *GPU Gems 3* was published in 2007, so it is an outdated source and there have been substantial improvements in both hardware and software since its publication [4].

These comparisons were not performed in this paper primarily due to time constraints and the lack of access to standardized and open-source GPU encryption libraries across all test environments. Furthermore, many of these libraries require complex integration or hardware-specific features that were outside the scope of our academic deployment setup. Nonetheless, benchmarking against these alternatives remains a valuable next step for validating the competitiveness and completeness of our approach.

8.3 Summary

The project was an overall success. Several versions of the AES-128 CTR algorithm were implemented and benchmarked to maximize throughput. Our GPU implementations achieve throughput rates that are extremely competitive, approaching or even exceeding those of high-end CPUs with dedicated AES encryption hardware. The system is capable of handling large input files quickly and securely, making it suitable for real-world use cases involving high-volume data. Even without hardware-specific optimizations (i.e., with a generic NVIDIA GPU without dedicated AES PTX instructions), our results show that AES-128 CTR encryption can be performed at scale with impressive speed, highlighting the practicality and power of GPU-accelerated cryptography.

References

[1] kokke. "Kokke/Tiny-AES-c." GitHub, 22 Sept. 2022, github.com/kokke/tiny-AES-c.

[2] tautastic. "GitHub - Tautastic/Aes-Ctr: AES-CTR Implementation Using AES-NI Intrinsics and SIMD Operations." GitHub, 2023, github.com/tautastic/aes-ctr. Accessed 1 May 2025.

[3] Wikipedia Contributors. “Advanced Encryption Standard.” Wikipedia, Wikimedia Foundation, 17 Mar. 2025, en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[4] Yamanouchi, Takeshi. “Chapter 36. AES Encryption and Decryption on the GPU.” NVIDIA Developer, 2007, developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-36-aes-encryption-and-decryption-gpu.