Exploring Parallelism in Cryptographic Algorithms: AES, RSA, ECC, and SHA256

Team3: Chris Huang, Shengqiao Zhao, Xiangyu Liu, Zihao Gong
{chrim.huang, shengqiao.zhao, swift.liu, zihao.gong}@mail.utoronto.ca

December 3, 2024

1 Abstract

Cryptography plays a crucial role in safeguarding data security and privacy in modern digital communication, protecting sensitive information from unauthorized access. Widely used in secure transactions, authentication, and blockchain, cryptographic algorithms must perform efficiently to handle real-time processing and the ever-growing scale of data. This report explores parallelism in four key cryptographic algorithms—Advanced Encryption Standard (AES), Rivest-Shamir-Adleman (RSA), Elliptic Curve Cryptography (ECC), and SHA-256—providing an introduction to each, analyzing their parallelization potential, and detailing key optimizations on CPU and GPU architectures. We compare performance improvements, discuss challenges from failed parallelization attempts, and highlight future opportunities to enhance efficiency and address current limitations through parallel computing techniques.

2 AES

2.1 The AES Algorithm

The Advanced Encryption Standard (AES) is a widely adopted symmetric encryption algorithm established by the U.S. National Institute of Standards and Technology (NIST) in 2001.AES operates on blocks of fixed size (128 bits) and supports key lengths of 128, 192, or 256 bits, offering varying levels of security. The algorithm is based on the substitution-permutation network (SPN) structure, which involves a series of transformations applied in multiple rounds. These include substitution (via S-boxes), permutation (shift rows and mix columns), and the addition of round keys generated from the original encryption key. Figure 1 illustrates the general workflow for AES encryption and decryption. For this evaluation, a key length of 256 bits is selected, providing the highest level of security among the supported key lengths.

2.2 Algorithm Steps Overview

1. **State:** AES processes data in a 4x4 column-major byte array, known as the state.
2. **Key Expansion:** Round keys are derived from the cipher key using the AES key schedule. For a 256-bit encryption key, 14 round keys are generated, with each round key being 128 bits (16 bytes) in length.
3. **Add Round Key:** This step performs an XOR operation between the current state and the round key derived from the key expansion process.
4. **Sub Bytes:** A non-linear substitution step replaces each byte in the state using a fixed Rijndael S-box [1]. The top 4 bits and bottom 4 bits of each byte serve as coordinates for the S-box lookup table.
5. **Shift Rows:** In this transposition step, rows of the state array are shifted to ensure diffusion. For encryption, rows are shifted to the right by 0, 1, 2, and 3 positions, respectively. For decryption, rows are shifted to the left by the same values (0, 1, 2, and 3).
6. **Mix Columns:** This linear mixing operation combines bytes within each column to obscure input-output relationships. Each column of the state matrix is multiplied by a predefined Maximum Distance Separable (MDS) matrix in the finite field $GF(2^8)$. Figure 2 illustrates the operations performed during this step.

2.3 Exploration of Parallelization

The simplest and most straightforward parallelization strategy involves processing input

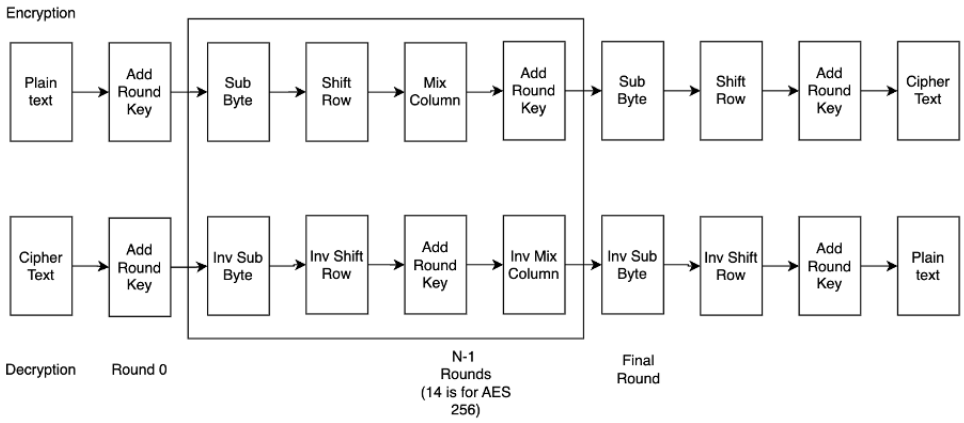

Figure 1: AES work flow

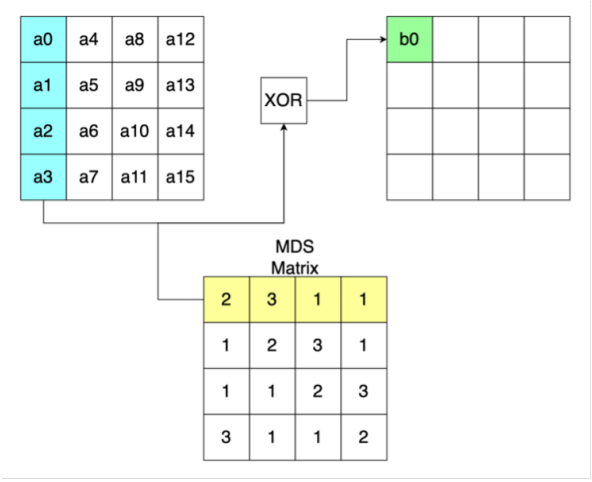

Figure 2: Mix columns in AES.

blocks independently. Since AES encrypts plaintext in 128-bit blocks, and each block is processed independently, parallelizing at the block level is an effective approach. Each block can be assigned to an individual thread or process, enabling simultaneous encryption of multiple blocks.

The Mix Columns step is one of the most computationally intensive operations in AES. As shown in Figure 2, each Mix Columns operation involves 16 independent calculations for every state. This independence makes the operation an ideal candidate for parallelization, and potential optimizations can be applied to enhance performance.

The nature of AES's state representation aligns well with the Single Instruction, Multiple Threads (SIMT) execution model used in CUDA programming. In CUDA, each warp consists of 32 threads, and since AES operates on a 16-byte state, at least two states can be processed concurrently within a single CUDA block. This allows warp-level optimizations to be effectively utilized.

The key expansion step, however, does not benefit significantly from parallelization, as the round keys are shared across all encryption blocks and can be precomputed. For large datasets, the computational cost of generating round keys is negligible compared to the encryption itself.

2.4 Implementation

The baseline implementation of AES-256 is a sequential version developed from scratch. This implementation serves as a performance benchmark and provides in-depth insights into the core AES algorithm. It also ensures the correctness of other parallelized implementations.

To exploit basic parallelism, a simple threaded parallelization strategy was employed. The input plaintext was divided into blocks, which were then processed independently by threads. Each thread handled a range of data blocks independently, improving throughput.

Further optimization was applied to the Mix Columns step using OpenMP. A thread pool was constructed using OpenMP to manage the parallel execution of Mix Columns operations. For each state of encryption, 16 tasks were spawned to handle the independent operations in Mix Columns, significantly reducing processing time.

In the SIMT implementation using CUDA,

Input Size	Serial	Simple Threaded (16 Threads)	OpenMP (16 Threads)	CUDA
16 characters	1,242 ns	478,653 ns	16,330 ns	37,678 ns
1,600 characters	122,228 ns	361,041 ns	1,694,506 ns	43,025 ns
16,000 characters	1,555,453 ns	479,739 ns	18,302,014 ns	42,407 ns
160,000 characters	12,263,766 ns	2,042,547 ns	183,257,649 ns	105,028 ns

Table 1: 160000 bytes input with different parameters.

	16 threads per block	32 threads per block	64 threads per block	128 threads per block
Coalesce 1	183,162 ns 10000 blocks 16 bytes shared memory	103,005 ns 5000 blocks 32 bytes shared memory	95,171 ns 2500 blocks 64 bytes shared memory	95,168 ns 1250 blocks 128 bytes shared memory
Coalesce 4	190,739 ns 2500 blocks 64 bytes shared memory	109,040 ns 1250 blocks 128 bytes shared memory	104,579 ns 625 blocks 256 bytes shared memory	105,925 ns 313 blocks 512 bytes shared memory
Coalesce 8	198,284 ns 1250 blocks 128 bytes shared memory	125,475 ns 625 blocks 256 bytes shared memory	126,971 ns 313 blocks 512 bytes shared memory	130,872 ns 157 blocks 1024 bytes shared memory
Coalesce 16	235,209 ns 625 blocks 256 bytes shared memory	165,877 ns 313 blocks 512 bytes shared memory	168,962 ns 157 blocks 1024 bytes shared memory	168,841 ns 79 blocks 2048 bytes shared memory

Table 2: 160000 bytes input with different parameters.

each thread processed one byte of the input data and applied the initial XOR operation with the round key. During the AES rounds, steps such as SubBytes and ShiftRows were executed independently without synchronization, with results being written directly to shared memory. The Mix Columns step required synchronization across all threads to ensure consistent access to the shared state data. AddRoundKey operations were performed independently by each thread.

The final rounds in the CUDA implementation followed a similar approach to earlier rounds but excluded the Mix Columns operation. Several optimizations were employed to improve performance, including limiting synchronization to critical stages such as the Mix Columns operation and output writing. Coalesced memory access was used to optimize global memory bandwidth utilization, while pre-computed lookup tables and shared memory usage reduced temporary variable creation and improved memory efficiency.

2.5 Performance Evaluation

The experiments were conducted on a University of Toronto Engineering Computing Facility machine equipped with an 11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz CPU and an NVIDIA GeForce RTX 3070.

Table 1 shows the time taken to complete encryption for various input sizes across different implementations. As observed, the CUDA implementation outperforms all CPU-based approaches after 1,600 characters in this experiment, which will be further discussed in the next section. It's performing worse than other operations, on 16 bytes of input, as it consume more time to spin up the kernel.

Although the Mix Columns step is an ideal candidate for parallelization, it incurs significant overhead. Each Mix Columns call takes approximately 120 ns on average, while the overhead of spawning and synchronizing threads in OpenMP can take up to 500 ns per task, leading to a total time exceeding 5,000 ns. This makes parallelizing short-running tasks less efficient than expected.

The CUDA implementation demonstrates superior performance compared to CPU-based implementations, as the AES encryption logic fits the SIMT model effectively.

Additional tests for coalesced input and varying numbers of threads were conducted. Table 2 shows the results. As observed, increasing the number of threads could improve performance, but it saturates around 128 threads per block. Coalescing factors can lead to performance bottlenecks caused by shared memory allocation and bank conflicts.

3 RSA

RSA is one of the most widely used public-key cryptographic algorithms, playing a crucial role in secure data transmission. The current parallelization method for RSA involves applying the exponentiation by squaring method, as detailed in Section 3.2 to each character independently across multiple threads. While effective for small-scale messages, this approach becomes inefficient for larger messages. To address this limitation, this section introduces an advanced technique: packing multiple characters into a single big integer, as described in Section 3.3. This method significantly reduces the number of modular exponentiation operations while maintaining the same message size. Additionally, the section explores experimentation results, alternative parallelization attempts, and their analysis.

3.1 The RSA Algorithm

The RSA algorithm, named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman [2]. It is based on the mathematical principles of modular arithmetic and the difficulty of factorizing large integers. RSA employs two keys: a public key used for encryption, and a private key used for decryption. The security of RSA relies on the fact that it is computationally infeasible to factorize a sufficiently large integer, which is the product of two prime numbers.

To generate keys for RSA, the user selects two large random prime numbers p and q, and computes their product $n = p \cdot q$, which serves as the RSA modulus. The Euler totient function $\phi(n)$ is then calculated as $(p-1)(q-1)$. An integer e, known as the encryption exponent, is chosen such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$. The decryption exponent d is computed as the modular multiplicative inverse of e modulo $\phi(n)$, satisfying $d \cdot e \equiv 1 \pmod{\phi(n)}$. The public key is represented as the pair (n, e), and the private key as (d, n).

RSA encryption transforms a plaintext message m (where $0 \leq m < n$) into a ciphertext c using the formula [2]:

$$c = m^e \mod n \tag{1}$$

Decryption is performed using the private key with the formula:

$$m = c^d \mod n \tag{2}$$

The correctness of RSA is based on modular arithmetic properties, ensuring that:

$$((m^e)^d) \mod n = m \tag{3}$$

Algorithm 1 Mathematical Operations in the RSA Algorithm [3]

Input: Two large prime numbers p, q
Output: n, e, d

1. Choose two large prime numbers, p and q.
2. Compute $n = p \cdot q$.
3. Compute the Euler totient function $\phi(n) = (p-1) \cdot (q-1)$.
4. Choose an integer e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$.
5. Compute $d = e^{-1} \mod \phi(n)$ (modular multiplicative inverse of e modulo $\phi(n)$).

3.2 Modular Exponentiation in RSA

The most computationally intensive aspect of the RSA algorithm is modular exponentiation, which is required during both encryption and decryption processes, as outlined in equations (1) and (2). Here, n denotes the RSA modulus, e is the public exponent, and d is the private exponent. Typically, the size of the RSA modulus n is 1024 or 2048 bits, making the computation of large powers mod n a non-trivial task. Performing the modular exponentiation naively—by computing $a^b \pmod{n}$ through sequentially multiplying the base a by itself $b-1$ times and reducing modulo n after each multiplication—would lead to excessive computational overhead. To mitigate this, a fast exponentiation technique, *exponentiation by squaring*, is employed [2]. This can be explained with an example: $2^{1234} \pmod{789}$, as shown below:

$$2^2 \equiv 4$$

$$2^4 \equiv 4^2 \equiv 16$$

$$2^8 \equiv 16^2 \equiv 256$$

$$2^{16} \equiv 256^2 \equiv 49$$

$$2^{32} \equiv 49^2 \equiv 34$$

$$2^{64} \equiv 34^2 \equiv 367$$

$$2^{128} \equiv 367^2 \equiv 559$$

$$2^{256} \equiv 559^2 \equiv 37$$

$$2^{512} \equiv 37^2 \equiv 580$$

$$2^{1024} \equiv 580^2 \equiv 286$$

Since $1234 = 1024 + 128 + 64 + 16 + 2$, we have:

$$2^{1234} \equiv 2^{1024} \cdot 2^{128} \cdot 2^{64} \cdot 2^{16} \cdot 2^{2} \pmod{789}$$

$$\equiv 286 \cdot 559 \cdot 367 \cdot 49 \cdot 4 \equiv 481 \pmod{789}$$

With this method, any modular exponentiation $a^b \pmod{n}$ can be computed efficiently in $\lfloor \log_2 b \rfloor$ squarings modulo n and at most $\lfloor \log_2 b \rfloor$ additional multiplications. Notably, the algorithm ensures that the computation never involves numbers larger than n^2, which significantly reduces both time complexity and memory requirements compared to naive methods.

3.3 Efficient Packing of Characters into a Big Integer

Traditionally, RSA applies modular exponentiation to each character of a message individually. This process involves converting each character into its ASCII representation and then performing modular exponentiation on each value. While straightforward, this approach is highly inefficient, as the number of modular exponentiation operations is directly proportional to the number of characters in the plaintext message. Given that modular exponentiation is the most computationally expensive step in RSA, this method results in significant overhead.

To address this inefficiency, our implementation leverages a more advanced technique: packing multiple characters into a single big integer. In this way, we can perform modular exponentiation on multiple characters simultaneously. This approach drastically reduces the number of modular exponentiation operations required, further accelerating both encryption and decryption processes.

3.3.1 Determining the Number of Characters per Big Integer

One might wonder, how many characters can be packed into a big integer? While big integers theoretically have no size limit and are only constrained by hardware memory, the RSA algorithm imposes a practical limitation. Specifically, given the modulus n, the plaintext space is defined as $\{0, 1, \ldots, n-1\}$. This means the value of the packed big integer must not exceed n.

Each character in ASCII representation requires at most 7 bits (representing values from 0 to 127). To ensure that the packed big integer does not exceed n, the bit length of the packed integer must always be slightly shorter than the bit length of n. This is achieved by subtracting one bit and applying a floor operation during the calculation. Theoretically, the maximum number of characters that can be safely packed into a single integer is determined by the following formula:

$$\begin{aligned} \text{Max\_Chars} &= \left\lfloor \frac{\text{bit\_length(n)} - 1}{\text{bit\_length(char)}} \right\rfloor \\ &= \left\lfloor \frac{\log_2 n - 1}{7} \right\rfloor \end{aligned} \tag{4}$$

Packing is achieved by iteratively shifting the existing packed value left by 7 bits (to accommodate the next character) and adding the ASCII value of the character being packed. Conversely, unpacking involves shifting the packed value right by 7 bits and extracting the ASCII value of each character.

For example, assume the modulus n in RSA is 1024 bits. The maximum number of characters that can be packed into a single big integer is $\left\lfloor \frac{1024-1}{7} \right\rfloor = 146$, meaning that a single modular exponentiation can encrypt or decrypt up to 146 characters simultaneously. For a modulus n of 2048 bits, this technique can handle up to 292 characters at once, further improving efficiency.

It is important to note that performing modular exponentiation with a large big integer, such as one packed with 146 or 292 characters, takes the same amount of time as performing modular exponentiation on a small number. This is because the time complexity of modular exponentiation depends primarily on the size of the exponent, rather than the size of the base n, as we explained in Section 3.2.

3.4 Parallel Implementation and Experimentation Using OpenMP

Table 3 illustrates the parameters applied in our experiment. Our input is a plaintext message with approximately 2500 characters. The implementation first divides this message into multiple chunks, where the chunk size is calculated using the formula (4). The number of chunks is then equally assigned to all the spawned threads using OpenMP.

p
14299623962416399520070177382898895550795403345466153217470516082934737582776038882967213386204600674145392845853859217990626450972452084065728686565928113
q
76309791959704047218912018477920021255354012927791239372074475745966927885136471792353355293072513505707284073737055647088717620330170968099103152128841 01
n (derived from $n = p \times q$)
10912013296739942927886096050899554152823750290279812912346875793726629149257644633073969600111060390723088861007265581882535850342905759282762943641310856602909362821263595383668656267584972062078627943109021801768106152175505671082387647644426055814717970711967428398241915211810375907603061668397856663141 3
$\phi(n)$ (derived from $\phi(n) = (p-1) \times (q-1)$)
10912013296739942927886096050899554152823750290279812912346875793726629149257644633073969600111060390723088861007265581882535850342905759282762943641310854409849046982583171187530733198495204428998164118581306333971740399032468542113781427389534504629515499099842105641763645261989075360684974104497678781920 0
e (public exponent)
65537
d (private exponent, derived from $d = e^{-1} \mod \phi(n)$)
46730330223584118622160180015036832148732986808519344675210555262940258739805766860224610646919605860206328024326703361630109888417839241959507572247284807035235569619173792292786907845791904955103601652822519121908367187885509270025388641700821735345222087940578381210879116823013776808975766851829020659073

Table 3: RSA Parameters Used in the experiment

Each thread processes its assigned chunk by performing modular exponentaition independently. After a thread finishes encrypting a chunk, it stores the resulting ciphertext into a shared buffer at the position corresponding to the index of the chunk. Once all chunks are processed, the main thread performs a serial write operation to save the ciphertext to a file.

For decryption, the same approach is applied: the ciphertext is read from the file, the workload is distributed evenly among all threads, and after all chunks are decrypted, the main thread writes the results to the decrypted file. Finally, by comparing the original plaintext with the decrypted text, we validate the correctness of the implementation.

3.5 Performance Evaluation and Analysis

In the RSA algorithm, the decryption process is significantly slower than encryption due to the larger size of the decryption key compared to the encryption key. To ensure fair experimentation, we measured the total time by summing the encryption and decryption times.

As shown in Table 4, we began by testing a serial implementation without the packing technique, which resulted in a total time of 1.61246 seconds. Next, we tested a serial implementation with the packing technique, which reduced the total time to 0.020374 seconds. This demonstrates a speedup of approximately $\frac{1.61246}{0.020374} \approx 79.14$, highlighting the substantial benefits of the packing method.

We further tested parallel implementations with packing, utilizing up to the hardware limit of 22 threads. The best performance was achieved with 8 threads, where the total running time was reduced to 0.003111 seconds, yielding a speedup of $\frac{1.61246}{0.003111} \approx 518.39$. This total speedup can be decomposed into the speedup from packing, which is 79.14, and the additional speedup from parallelism $\frac{0.020374}{0.003111} \approx 6.55$.

The experiments were conducted on a MacBook equipped with an Apple Silicon processor. The hardware features a single CPU package with 11 cores organized into two clusters. Cores 0–5 are efficiency cores, each with a 64KB L1 data cache and a 128KB L1 instruction cache, sharing a 4096KB L2 cache. Cores 6–10 are performance cores, each with a 128KB L1 data cache and a 192KB L1 instruction cache, sharing a 16MB L2 cache.

3.6 Other Parallelism Attempt and Analysis

An attempt was made to utilize multiple threads to perform modular exponentiation for a single value. For instance, calculating $2^{64} \mod 78$ was divided among 4 threads. Each thread independently computed $2^{16} \mod 78$ using the exponentiation by squaring method. Afterward, the results were combined either serially or progressively in an upside-down tree shape, where pairs of results were multiplied and reduced modulo 78 until the final value was obtained.

Despite these efforts, the results showed no speedup and, in some cases, were worse than a single-threaded implementation using the exponentiation by squaring method. This is likely due to the significant synchronization overhead and the inherently serial nature of the exponentiation by squaring algorithm, which makes it difficult to parallelize effectively.

Threads	Total Time (s)	Speedup
1 (no packing)	1.61246	—
1 (with packing)	0.020374	1.00
2	0.01258	1.62
3	0.009111	2.24
4	0.005135	3.97
5	0.006115	3.33
6	0.003914	5.20
7	0.003964	5.14
8	0.003111	6.55
9	0.003469	5.87
10	0.003285	6.20
11	0.003263	6.24
12	0.003717	5.48
13	0.00385	5.29
14	0.004083	4.99
15	0.004377	4.66
16	0.005158	3.95
17	0.004595	4.43
18	0.004186	4.87
19	0.003969	5.13
20	0.004348	4.69
21	0.003947	5.16
22	0.004241	4.80

Table 4: Performance comparison of RSA implementations, where the total time is the sum of encryption and decryption times. The first row corresponds to the serial implementation without packing. The remaining rows with thread counts 1–22 represent implementations with packing. The 1-thread implementation with packing is used as the baseline for speedup calculations.

Another attempt was made to implement RSA on a GPU using CUDA. However, CUDA lacks support for big integers, and there is no existing library capable of handling 1024-bit or 2048-bit keys required for RSA. To address this, we developed a custom big integer library from scratch. Unfortunately, the output showed a performance degradation, likely due to the inefficiencies introduced by the custom library implementation. We then considered reducing the RSA key size to 64 bits to fit within the GPU's constraints. However, this significantly undermines the security of the RSA algorithm, as its security relies on the large size of the key. A 64-bit modulus n can be trivially factorized into two prime numbers using brute force, rendering the encryption effectively meaningless. Furthermore, the data collected using a 64-bit key on the GPU cannot be directly compared to the results from the CPU implementations with larger key sizes, as the drastically different key lengths lead to an invalid comparison.

4 ECC

4.1 The ECC Algorithm

Elliptic curve cryptography is a public key encryption algorithm based on elliptic curve mathematical theory. Compared with other methods, ECC can provide the same security with a shorter key length, which has significant advantages in resource-constrained scenarios. The security of ECC is based on the computational difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP) that, given a point A on the elliptic curve and its multiple point Q, it is difficult to find k such that $Q = kA$ (scalar multiplication). From the graph, as k increases, it is harder to use endpoint Q and start point A to calculate step k [4].

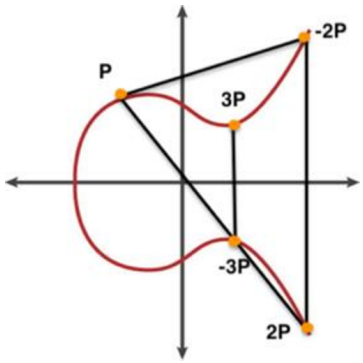

Figure 3: Elliptic curve

On finite fields, an elliptic curve equation is defined as:

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

with $4a^3 + 27b^2 \mod p \neq 0$.

To use ECC for encryption:

$$Q = kG$$

where G is a point on the curve, and k is a private key. The sender uses the public key Q and a random number r to encrypt the plaintext point M into a point pair (C_1, C_2), calculated as:

$$C_1 = rG$$

$$C_2 = M + rQ$$

The receiver recovers the original message M using the private key k with the following equation:

$$M = C_2 - kC_1$$

$$M + rQ - k(rG) = M + r(kG) - k(rG) = M$$

Compared with other same-type encryption algorithms, ECC encryption takes up less space. For the same security strength, the key length required by ECC is much smaller than RSA and other traditional algorithms. For example, the security provided by ECC using a 256-bit key is equivalent to the 3072-bit key of RSA. ECC is widely used in key exchange (ECDHE), digital signature (ECDSA), HTTPS protocol, mobile payment (Apple Pay), and smart cards (credit cards, passports).

ECC is a popular algorithm for encrypting small data, but few papers have studied ECC encryption of large data. After ECC encryption, one point M will generate two points C_1 and C_2, which increases the cost of storage. Because of large integers, the efficiency of directly encrypting plaintext may not be as good as other algorithms. However, ECC has great potential to encrypt large files in the future. We can split files into multiple data blocks to avoid extremely large integer calculations and use parallel programming to calculate data blocks. ECC-encrypted files are more secure than traditional large file encryption algorithms such as AES because their security relies on mathematical problems. Under the threat of quantum computing, symmetric encryption algorithms may need to increase the key length to maintain security, but ECC can provide quantum resistance by expanding the range of finite fields.

We modify the current ECC algorithm to be able to encrypt and decrypt different types and sizes of files. After reading the file, we transform it into binary type and split it into data blocks, where each data block contains m plaintext. For each data block that contains m plaintext, we use m as the x of the point M and check if there exists a y that satisfies the elliptic curve equation. If not, we add n to x until the condition is satisfied. After forming the point M, we store n and the point M. After encryption, we store the last data block length and each data block C_1, C_2, n into a binary file. After decryption, we add n to $M.x$ to reconstruct the plaintext in binary and finally transform the file to the correct type.

Comparison with Existing Work: A common practice is to use symmetric encryption algorithms like AES to encrypt large files and then use ECC to secure the AES keys. Our focus is on exploring and implementing a method to use ECC directly for encrypting any type of large file. We leverage GPUs and CPUs to evaluate the parallelization potential of ECC encryption and decryption processes for large-scale data.

4.2 Parameters for ECC Implementation

The ECC implementation in this study is based on 64-bit integers (`int64`), which reduces security compared to using 128-bit integers (`int128`). However, this decision allows a better comparison of the efficiency of ECC for large files on both CPU and GPU platforms. This is because CUDA lacks a suitable large-integer library to support ECC's mathematical operations effectively. For instance, GMP is more mature in CPU optimization than in GPU optimization. Defining a custom large-integer library introduces performance bottlenecks, as ECC operations (such as multiplication and modulus) are heavily impacted by basic integer operations. These limitations may obscure the real performance improvements achieved through ECC parallelization.

CUDA and C++ both support `int64`, which means operations are more closely aligned with the underlying hardware performance. This makes comparisons of ECC parallelization efficiency between CPUs and GPUs more reliable. Therefore, this study focuses on speedup achieved through parallelization rather than absolute security enhancements.

For the elliptic curve equation $y^2 = x^3 + ax + b$, the parameters are defined as:

$$G = (1, 360265885966316755)$$

$$P = 1157920892373161953$$

$$A = 3, \quad B = 7$$

The finite field value P determines the complexity of the operation and security level. The chosen P enables encryption of a single data block of 7 bytes without causing overflow for `int64` data.

The GPU device used in this implementation is an NVIDIA GeForce RTX 4080. Testing with thread configurations of 256, 512, and 128 threads per block showed similar performance, with encryption of a 292591KB file taking approximately 26836–26825ms. Thus, the configuration of 256 threads per block was selected.

For the CPU implementation, an Intel Core i9-14900KF capable of running 32 threads simultaneously was used. Testing showed that the

operation efficiency peaked at 40 CPU threads, delivering the fastest performance. This efficiency may relate to the proportion of I/O operations versus computational operations during encryption and decryption.

4.3 Useful Optimization

4.3.1 Reading Data and Generating Random Numbers in Parallel

From the introduction section of ECC, in step 3, each data block generates its own random r. This random number generation can be moved to the data reading stage to optimize performance. For large data files, which require significant time to load, multiple threads can be utilized to calculate random numbers simultaneously, equivalent to the number of data blocks. This allows pre-generated random numbers to be directly loaded during step 3.

Table 5: Optimization results for different file sizes

Data Size (KB)	Speed Up Ratio	Before Opt (us)	After Opt (us)
17	44.1%	51774	28922
426	53.1%	1340079	628390
8028	44.7%	21925434	12119562
292591	42.5%	855275098	491564361

From Table 5, the speed-up is effective. This is because before optimization, each data block calls the function for generating random numbers multiple times, which adds significant overhead.

The acceleration efficiency exceeded expectations. It is highly likely that thread contention caused by multiple threads concurrently invoking the same function slowed down the execution time. Moving the random number generation to a single thread resolved this issue.

4.3.2 C_1, C_2 Multi-Threaded Computing

From step 3 of the encryption algorithm in the introduction section, we can see that there is no correlation between the calculation of C_1 and C_2, which provides the potential for parallelization. In the original algorithm, we made two threads calculate C_1 and C_2 separately, but this optimization has a negative effect, which increases the running time of the 292591KB file by about 20 percent because the thread overhead is greater than the time reduced by the parallel calculation of C_1 and C_2.

Because we use ECC for large files, we can try to increase the workload for additional threads to solve this problem. We allocated the file range that each thread needs to encrypt first, then when each thread completes step 3 of a data block, it will store the pre-data needed to calculate C_1 and C_2, and directly start encrypting the next data block until the number of data blocks assigned to the thread is completed. Finally, each of the threads will be divided into two threads to calculate a group of C_1 and a group of C_2 separately. In this way, we increase the data that the new thread needs to process to try to offset the overhead of the new thread and provide faster running efficiency.

Table 6: Results of Multi-Threaded Computing for C_1, C_2

Data Size (KB)	Speed Up Ratio	Before Opt (us)	After Opt (us)
17	-40%	28922	48239
426	-20%	628390	788774
8028	-13.4%	12119562	14005367

From Table 6, we can see that when the amount of data is too small, it has a huge negative effect. Even if the workload of the new thread is increased, due to the small amount of computation influence of the file, it is difficult to cover the overhead of the additional thread. When processing large files, the reduced speedup ratio gradually decreases. From this trend, we can observe that as increasing computation time becomes large enough, this optimization may eventually no longer have negative effects and may speed up the code. Especially, in practical applications, we are using `int128` for ECC. As the C_1, C_2 calculation amount increases due to the expansion of the range of the finite field, this optimization method may have beneficial performance, which gives us a new strategy for ECC encryption of large files. But what must be discussed is that the reduction in negative impact can be simply due to the increase in the total workload resulting in a decrease in the proportion of negative impact. Due to the limitations of int64, it is difficult for us to use large integer operations to test whether it is effective. This may require further verification after using large integers in the future.

4.4 Performance and Analysis

4.4.1 File Size Increase

Table 7: File Size Increase after Encryption

Original File Size	Encrypted File Size
17 KB	94 KB
426 KB	2436 KB
8028 KB	45873 KB
292591 KB	1671949 KB

After encryption of different sizes of files, the file size grew 5.6 times on average. This is the disadvantage of ECC encryption of big data at present, as the encrypted files are too large.

4.4.2 Performance on CPU Serial, CPU Parallel, and GPU

From Figure 4 comparing CPU serial, parallel, and GPU, ECC encryption of large files is very suitable for parallelization. The efficiency improvement brought by using CPU or GPU parallel computing is very obvious. This is because ECC encrypts different data blocks completely independently of each other, and the proportion of serial computing is relatively low. As the amount of file size increases, the CPU's parallel running time grows faster than the GPU's. This is because the GPU has more threads than the CPU and can calculate more data in parallel.

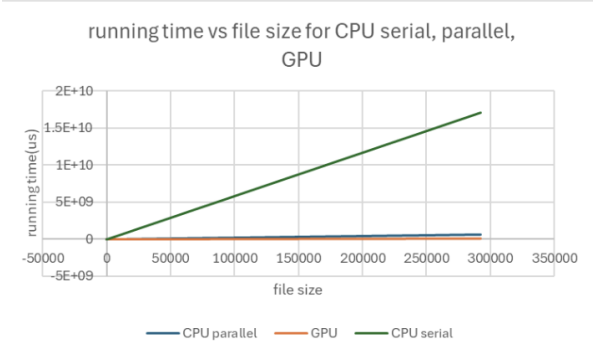

Figure 4: Running Time vs File Size for CPU Serial, Parallel, and GPU

From Figure 5, we can see that compared with the CPU, the actual time used for ECC encryption of small files by the GPU is relatively low. This may be due to the large overhead of data transfer. Small data should be processed more efficiently on the CPU. When the file size is 17 KB, the CPU completes the encryption in half the time of the GPU.

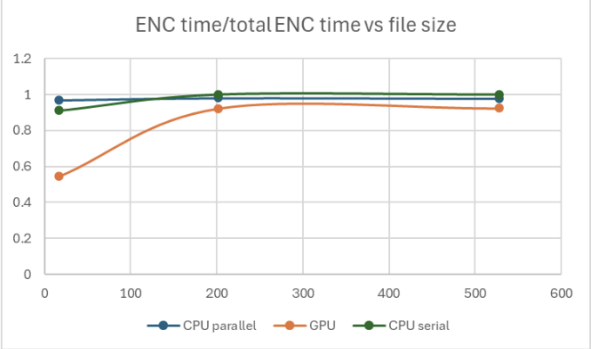

Figure 5: ENC time to total ENC time vs file size.

4.4.3 Speedup on CPU Serial, CPU Parallel, and GPU

Table 8: Speedup Comparison for ECC Encryption

File Size (KB)	CPU Speedup	GPU Speedup
17	19.01159	4.351504
202	23.85288	66.55653
528	26.19468	139.8957
2655	27.44297	303.8002
292591	29.09379	539.0668

In general, when ECC encrypts files, if the file is small, using the CPU can bring faster speedup. As the file size increases, the speedup brought by using the GPU far exceeds the parallelism of the CPU. The advantage of the GPU's large number of threads calculating simultaneously masks the overhead of data transmission and kernel startup.

5 SHA-256

5.1 The SHA-256 Algorithm

SHA-256 stands for Secure Hash Algorithm 256-bit. It is a cryptographic hash function that transforms input data of any size into a fixed-length 256-bit (32-byte) hash value. The algorithm is called SHA-256 because its output is always 256 bits, regardless of the input size. It is widely used in security protocols and applications such as Bitcoin, file integrity verification, and SSL certificates due to its core characteristics: deterministic results, irreversibility, uniqueness, collision resistance, and speed.

For instance, SHA-256 ensures that the same input always produces the same hash value, and the output is irreversible. Even a slight change

in the input results in a completely different hash value, and it is computationally infeasible to find two different inputs that produce the same hash output.

Algorithm 2 SHA-256 Main Process Steps

Input: Message M
Output: 256-bit hash value
Prepare Input:
Pad the input message M to ensure its length is a multiple of 512 bits.
Initialize hash values $h[0]$ to $h[7]$ with predefined constants.

Process Blocks:
foreach *512-bit block B of the padded message* **do**
Expand B into 64 subblocks $W[0 \dots 63]$ (32 bits each).
for $i = 0$ *to* 63 **do**
$T_1 \leftarrow h[7] + \Sigma_1(h[4]) + \text{Ch}(h[4], h[5], h[6]) + K[i] + W[i]$
$T_2 \leftarrow \Sigma_0(h[0]) + \text{Maj}(h[0], h[1], h[2])$
Update working variables:
$h[7] \leftarrow h[6]$, $h[6] \leftarrow h[5]$, $h[5] \leftarrow h[4]$,
$h[4] \leftarrow h[3] + T_1$, $h[3] \leftarrow h[2]$, $h[2] \leftarrow h[1]$,
$h[1] \leftarrow h[0]$, $h[0] \leftarrow T_1 + T_2$
end
end
Update Hash Values:
After processing each block, update $h[0]$ to $h[7]$.

Output Hash:
Concatenate $h[0]$ to $h[7]$ to form the final 256-bit hash value.

SHA-256 pads the input to ensure its length is a multiple of 512 bits. Padding starts with a single 1 bit, followed by enough 0 bits to make the length 448 bits modulo 512, leaving 64 bits at the end to store the original message length in binary [5]. For example, a 24-bit message is padded with a 1 bit, sufficient 0s to reach 448 bits, and the 64-bit binary representation of 24, ensuring the total length is a multiple of 512 bits. The padded message is then divided into 512-bit blocks for processing.

SHA-256 employs logical operations to achieve its cryptographic properties. These include left rotation ($\text{ROTL}_n(x)$), right shift ($\text{SR}_n(x)$), and functions like "choose" ($Ch(x, y, z)$), which selects bits from y or z based on x, and "majority" ($Maj(x, y, z)$), which selects the majority bit among x, y, z. Additional operations, $\Sigma_0(x)$, $\Sigma_1(x)$, $\sigma_0(x)$, and $\sigma_1(x)$, combine rotations and shifts to enhance diffusion in the hash function.

The algorithm initializes hash values from the square roots of prime numbers for randomness. Each 512-bit block is expanded into 64 subblocks (W_t), where the first 16 are derived directly from the message, and subsequent subblocks use σ_1, σ_0, and prior values for mixing. This prepares the data for the main hash computation.

Each block undergoes 64 rounds of operations, using the expanded subblocks and predefined constants. Eight working variables (a, b, c, d, e, f, g, h) are initialized from previous hash values. For each round, temporary values T_1 and T_2 are computed. T_1 incorporates h, $\Sigma_1(e)$, $Ch(e, f, g)$, a constant, and $W[r]$, while T_2 uses $\Sigma_0(a)$ and $Maj(a, b, c)$. The working variables are updated iteratively. After processing all rounds, the hash values (h_0 to h_7) are updated. Once all blocks are processed, the final 256-bit hash is formed by concatenating h_0 to h_7.

5.2 Parallel Implementation

The parallel implementation of SHA-256 was performed on both the CPU and GPU. The primary focus was on embarrassingly parallel workloads with multiple inputs, as attempts to parallelize the computation for a single input were unsuccessful (detailed in the failure analysis).

5.2.1 Multiple Input Parallelism on CPU with Thread Pool and AVX

On the CPU, parallelism was achieved using a thread-pool [7] and AVX (Advanced Vector Extensions). A thread pool (16 threads) was implemented to manage threads efficiently. When a new input arrived, it was assigned to a free thread from the pool, avoiding the overhead of repeatedly creating and destroying threads. Inputs were processed in parallel, and threads continued to work until the input pool was empty.

To enhance performance, AVX [6] instructions were used to perform SIMD (Single Instruction Multiple Data) operations, enabling multiple values to be processed simultaneously within a single register. For example, AVX1 uses 128-bit registers to handle 4 values at once, while AVX2 extends this to 256-bit registers, processing 8 values simultaneously. The operations utilized included:

- `_mm128_srli_epi32`: Right logical shift (`>>`) applied to each element in a vector.

- `_mm128_set1_epi32`: Sets all elements in a vector to the same value.
- `_mm128_set_epi32`: Sets elements of a vector one by one.
- `_mm128_slli_epi32`: Left logical shift (<<) applied to each element in a vector.
- Logical operations such as AND, XOR, and OR, applied to entire vectors.

Both AVX1 and AVX2 implementations were tested to compare performance differences, with AVX2 demonstrating better performance due to its larger register size and ability to process more data in parallel.

5.2.2 CUDA Implementation on GPU

For the GPU implementation, CUDA was employed to distribute the workload across multiple threads, effectively harnessing the parallel processing power of the GPU. The input data was divided into blocks, with each block containing a specified number of threads, determined by `threads_per_block`. Each thread independently processed one input string through the SHA-256 algorithm, ensuring efficient parallelism.

The input data array was split into segments, and each thread accessed its assigned input string from the global memory. After completing the SHA-256 computation, the thread stored the resulting hash value back into the global memory, where the output data array was maintained. This process allowed for simultaneous handling of a large number of input strings, maximizing computational efficiency.

To facilitate the GPU computations, data transfers were required between the host (CPU) and the device (GPU). Input strings were transferred to the GPU prior to computation, and the resulting hash values were transferred back to the host once the processing was complete. The implementation leveraged CUDA's thread hierarchy and memory management to achieve high throughput and scalability for processing extensive input datasets.

6 Performance Evaluation and Analysis

Using an AMD Ryzen 7 7800X3D CPU and an NVIDIA RTX 4060 Ti GPU for testing, the running time is directly proportional to the input length due to its dependency on processing blocks. After testing different input lengths, this linear relationship was evident (see Figure 6). We observed significant differences between CPU and GPU performance, with the disparity increasing for longer input lengths. This observation highlights why CPUs are typically preferred for large-file SHA-256 authentication tasks. CPU parallel SHA-256 is commonly used for applications such as file and data integrity checks, deduplication, forensics, and malware analysis.

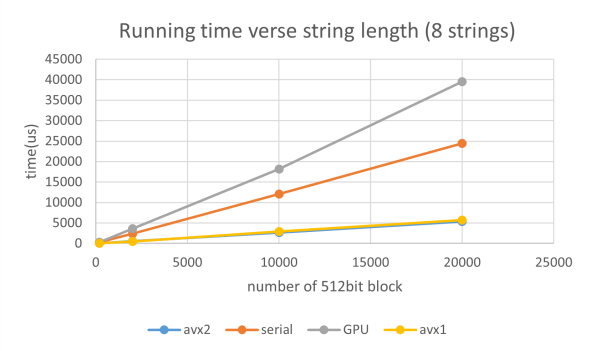

Figure 6: Linear relationship between input length and running time.

The relationship between the number of inputs and running time shows distinct trends for CPU and GPU implementations (see Figures 7 and 8). On the CPU side, the running time increases significantly when the total number of inputs exceeds the number of threads multiplied by the AVX vector size. At this point, additional threads are required to process new inputs, causing idle threads to execute additional calculations and thereby increasing running time. Conversely, GPU performance exhibits a threshold behavior, where running time remains relatively flat until the workload exceeds the GPU's capacity. Once this threshold is crossed, the running time increases drastically due to oversubscription.

On the CPU side, running time increases only when the number of inputs exceeds the product of `c`, `threads`, and `vector_size` (where `c` is a int starts from 1). Threads then perform extra rounds of calculations, leading to a sudden increase in running time. In contrast, GPU processing handles inputs in parallel until it reaches the capacity limit. Beyond this threshold, oversubscription results in a substantial increase in running time.

Before reaching the threshold, GPU performance achieves a speedup of approximately 170 times for large input numbers, significantly outperforming the maximum 27 times speedup

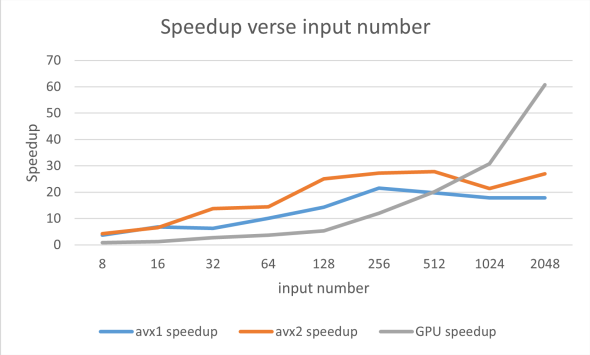

Figure 7: Speedup versus input number for different methods when string length is 200×512.

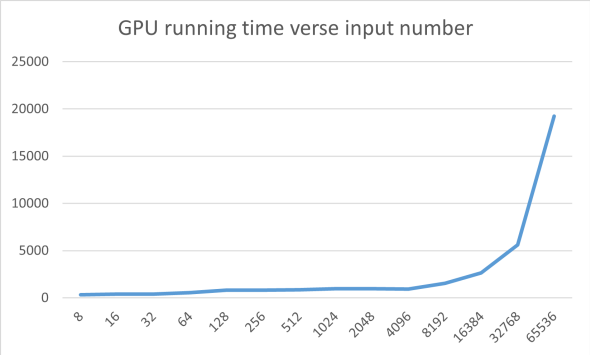

Figure 8: GPU running time versus input number. A significant increase occurs beyond the threshold of 32,768 inputs.

achieved by CPU AVX2 implementations. This makes CUDA SHA-256 well-suited for tasks involving password hashing, blockchain mining, and network security, which typically require handling small input sizes but massive input numbers.

6.1 Other Parallelism Attempt and Analysis

Data expansion is the only part of SHA-256 that can be parallelized for a single input, with each thread handling a portion of the input and producing part of the expanded subblocks. For example, with an input size of 256×512 bits and 16 threads available, 200 blocks need to be expanded into $16,384 \times 32$ subblocks. Each thread processes 16×512 bits of data and outputs 512 subblocks. To balance the workload, a thread pool was used to dynamically allocate tasks among threads.

However, the results were worse than expected. The running time was even longer compared to serial execution. Further tests revealed that the data expansion step takes approximately 2 microseconds for a 256×512 input, which is significantly lower than the thread overhead, rendering the parallel approach inefficient. CUDA was also explored to parallelize the expansion process, but the performance did not improve as anticipated.

7 Future Work and Conclusion

For AES, our future efforts will focus on utilizing the SIMD AVX instruction set to accelerate CPU execution. The AVX supports 128-bit registers and operations, offering potential for significant performance gains. Additionally, we plan to address the bank conflicts that arise from using shared memory. By resolving these conflicts, we aim to improve performance scalability with an increasing number of threads per block and higher coalescing factors.

For RSA, future work will aim to optimize both CPU and GPU implementations. On the CPU side, we will explore more efficient methods for parallelizing modular exponentiation for single values, addressing current synchronization and serial dependency challenges that hinder performance. On the GPU side, our focus will be on refining the custom big integer (bigint) library to eliminate performance degradation. This includes optimizing memory usage and computation to better leverage GPU architecture. These improvements will enhance the overall efficiency of RSA encryption and decryption across platforms.

For ECC, our current implementation is based on 64-bit integers, which can only approximate some trends of 128-bit integer operations in terms of calculation. However, the volume of calculations with int64 cannot compare to those of larger integers, making it challenging to fully verify the effectiveness of optimization strategies in C_1 and C_2 multi-threaded computations. Future work will focus on implementing large integer operations to rigorously evaluate and validate all optimization strategies for ECC encryption.

For SHA-256, future research will concentrate on optimizing both CPU and GPU usage for specific applications such as cryptocurrency mining and file integrity checking. For mining, we aim to enhance GPU kernel efficiency and memory management to handle massive input sizes with minimal latency. For file integrity checking, refinements will target CPU-based SIMD operations and adaptive thread

pooling to efficiently process large files. These optimizations will tailor performance improvements to the distinct requirements of each application.

This study provides a preliminary exploration into the potential of parallelization to improve the performance of cryptographic algorithms. While progress has been made in identifying optimization opportunities and addressing some challenges, much work remains to fully realize the benefits of parallel computing for cryptographic applications. We hope that the insights gained here serve as a small step towards more efficient and secure implementations in the future.

References

[1] Federal Information Processing Standard, *The Advanced Encryption Standard (AES)*, FIPS PUB 197: The official AES standard, 2001-11-26. Retrieved 2010-04-29.

[2] Karl Dilcher, *Cryptography: Course Notes for CSCI/MATH 4116*, Dalhousie University, pp. 70–74.

[3] Md. Ahsan Ayub, Zishan Ahmed Onik, Steven Smith, "Parallelized RSA Algorithm: An Analysis with Performance Evaluation using OpenMP Library in High Performance Computing Environment," in *2019 22nd International Conference of Computer and Information Technology (ICCIT)*, IEEE, 18-20 December, 2019.

[4] S. R. Singh, A. K. Khan, and T. S. Singh, *A critical review on Elliptic Curve Cryptography*, 2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT), Pune, India, 2016, pp. 13–18, doi: https://doi.org/10.1109/ICACDOT.2016.787754310.1109/ICACDOT.2016.7877543.

[5] RFC Editor, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)," *RFC 6234*, 2011. Available at: https://www.rfc-editor.org/rfc/rfc6234

[6] Intel Corporation, "Intel® Intrinsics Guide," *Intel Developer Documentation*, 2021. Available at: https://www.intel.com/content/dam/develop/external/us/en/documents/18072-347603.pdf

[7] progschj, "Thread Pool Implementation in C++11," *GitHub Repository*, 2013. Available at: https://github.com/progschj/ThreadPool