Accelerated Encryption Techniques Utilizing CUDA

Julian Singkham, Trenton Bowser, Joseph Weller,
Chip Hennig, Nathan DuPont
{singkhamj,bowsert,wellerj,hennigc,dupontn}@msoe.edu

November 9, 2021

Abstract

This report will cover the process of applying parallelization to two widely used encryption algorithms, RSA and AES. The algorithms will initially be implemented serially using the C programming language. Subsequently, CUDA will be used to parallelize them. Both serial and parallel implementations of the algorithms will then be profiled to calculate the speedup which parallelization achieves.

1 Introduction

In modern day cryptography, there are few encryption techniques that have been robust enough to stay in use over time. This report will contain the examination and acceleration of two encryption algorithms which have truly stood the test of the time. The algorithms which were chosen showcase contrasting areas of cryptography as one uses asymmetric keys while the other uses a symmetric block cipher. The results of constructing serial and parallel implementations of both of these techniques will be discussed throughout this report. The run-times of these encryption and decryption algorithm implementations for both techniques will also be discussed.

1.1 RSA Encryption

The Rivest–Shamir–Adleman (RSA) algorithm is an asymmetric cryptography algorithm, meaning that it uses a public and a private key for encryption and decryption [1]. RSA users create and publish a public key based on two large prime numbers, which are kept secret. Messages can be encrypted by anyone, via the public key, but can only be decrypted by someone who knows the original prime numbers or the private key. RSA relies on the difficulty of factoring the product of two large prime numbers, known as the factoring problem, in order to secure encrypted messages.

1.2 AES Encryption

The Advanced Encryption Standard (AES), algorithm is a symmetric cryptographic algorithm, meaning that is uses the same key for both encrypting and decrypting the data. AES utilizes a block cipher and runs multiple iterations of calculations to replace, shift, and mix different bits in order to produce an encrypted output. The operations are reversed for decrypting the data.

1.2.1 Encryption Process

A 128-bit key uses 10 rounds of encryption and 4 words (32-bits) per block. AES is column major, meaning that data is organized by columns. AES can be broken down into 4 key components: SubBytes, Shift Rows, Mix Columns, and Add Round Key. The initial round of AES only uses Add Round Key. All but the last round of AES encryption follows the aforementioned 4-step process, while the final round omits Mix Columns.[2]

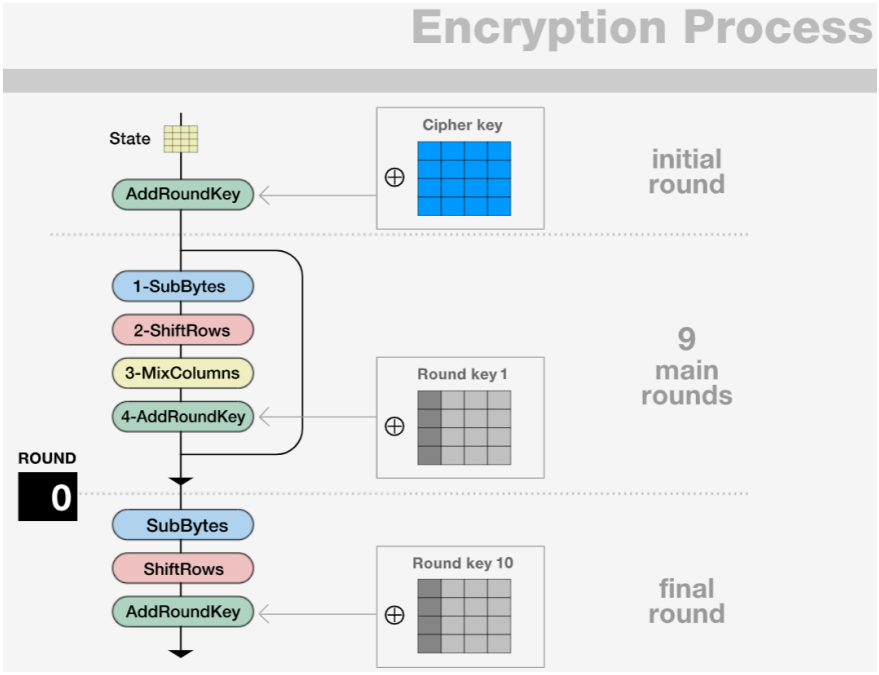

Figure 1: 128-Bit AES Encryption Process

1.2.2 SubBytes

In SubBytes, each byte is replaced by a byte from the S-box lookup table where the left hex value is the x-coordinate and the right hex value is the y-coordinate.

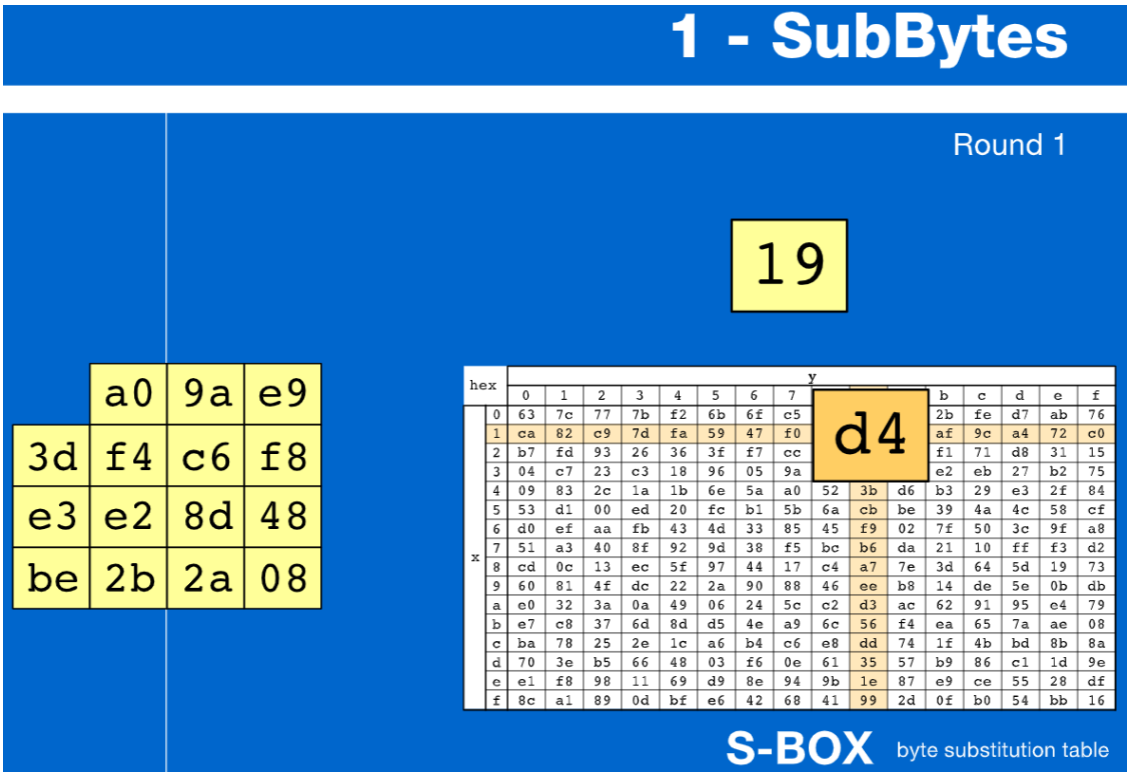

Figure 2: SubBytes

1.2.3 Shift Rows

In Shift Rows, each row is shifted to the left by its row number, where the first row is labeled as 0.

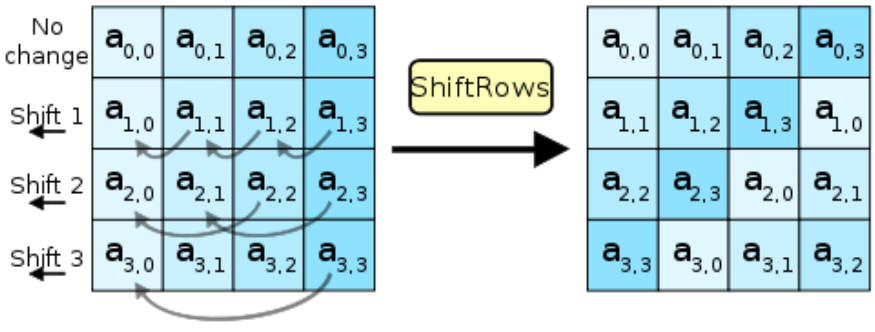

Figure 3: Shift Rows

1.2.4 Mix Columns

In Mix Columns, each column is modulo multiplied in Rijndael's Galois Field. Shift Rows along with Mix Columns are the primary source of diffusion in AES.

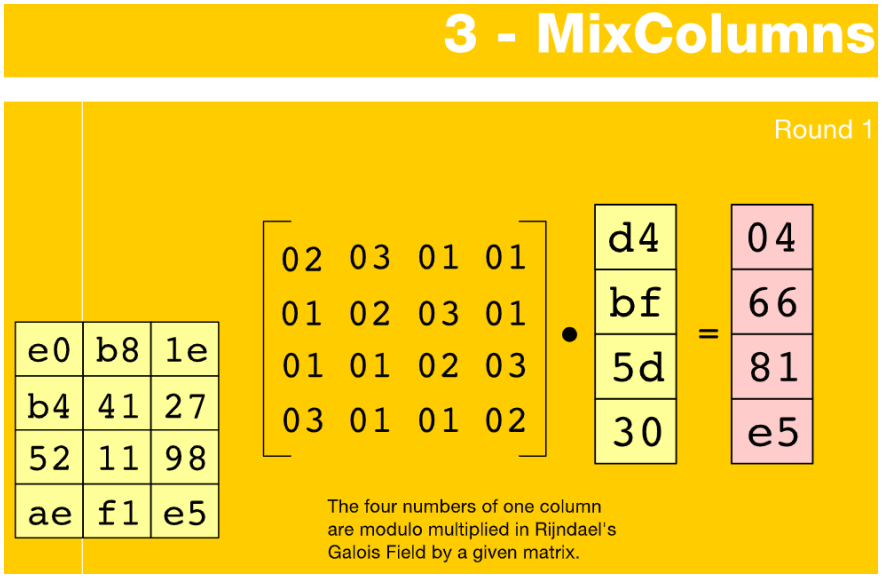

Figure 4: Mix Columns

1.2.5 Add Round Key

In Add Round Key, each column is XOR'd with the corresponding round key column.

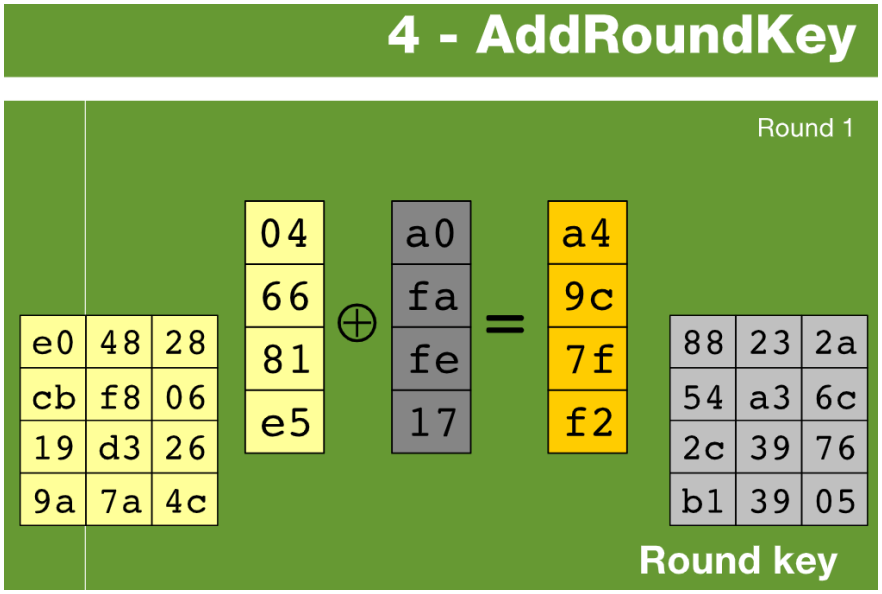

Figure 5: Sub-Bytes

1.2.6 Key Schedule

Since AES is a round based cipher, the given key needs to be expanded so that each round is given a unique key break. The first part to creating the expanded key is to take the previous column and RotWord it, meaning the column is shifted up by 1 byte.

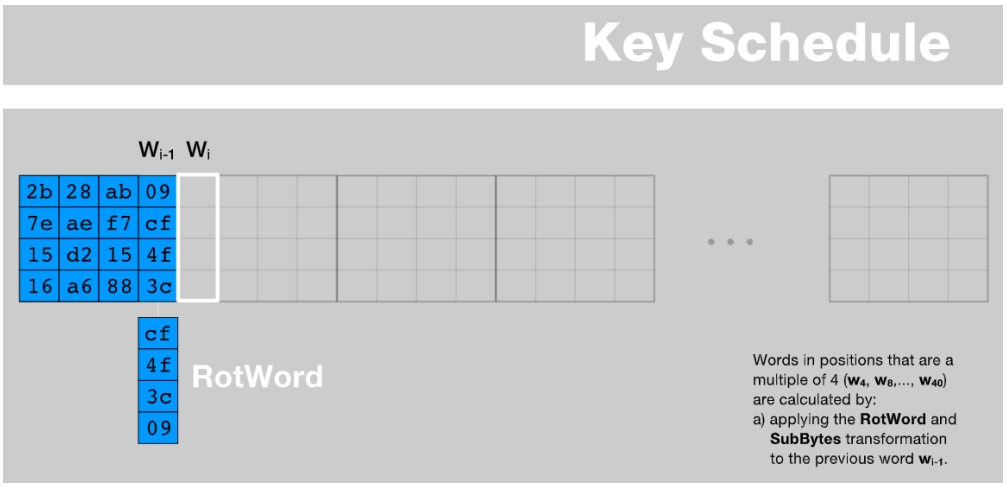

Figure 6: Key Schedule Part 1

The column then goes through SubBytes.

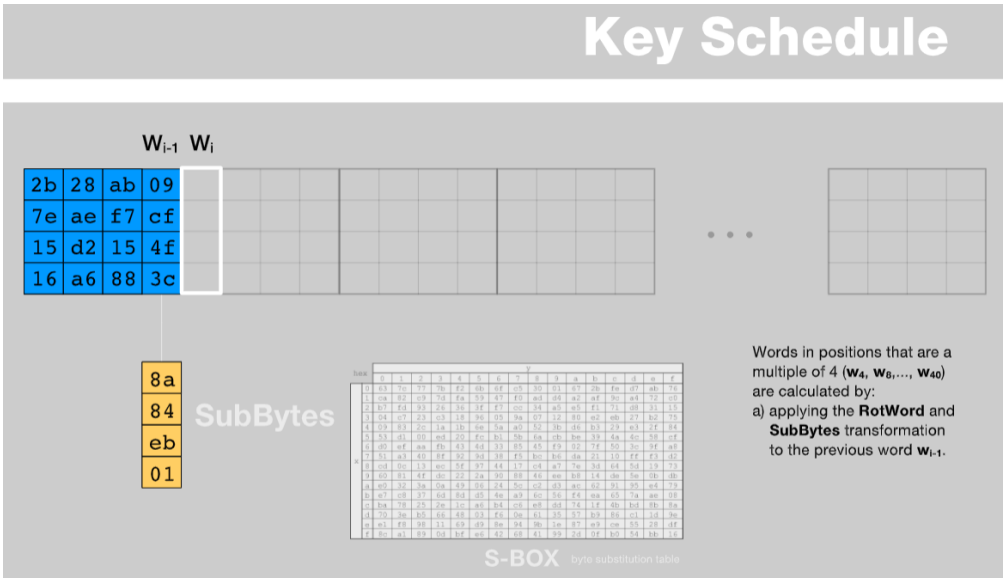

Figure 7: Key Schedule Part 2

The column is then XOR'd by the corresponding previous round column and XOR'd by the corresponding Rcon column.

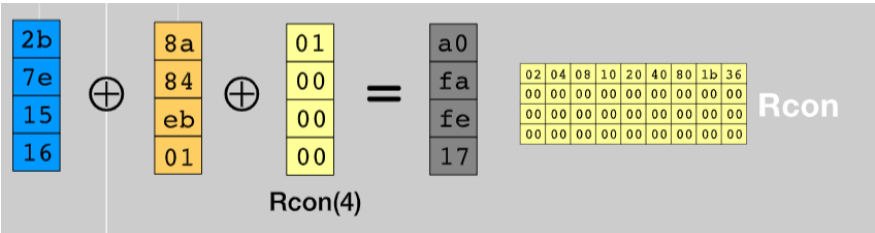

Figure 8: Key Schedule Part 3

The rest of the columns in the round are then created by XORing the corresponding column in the previous round to the previous column in the current round.

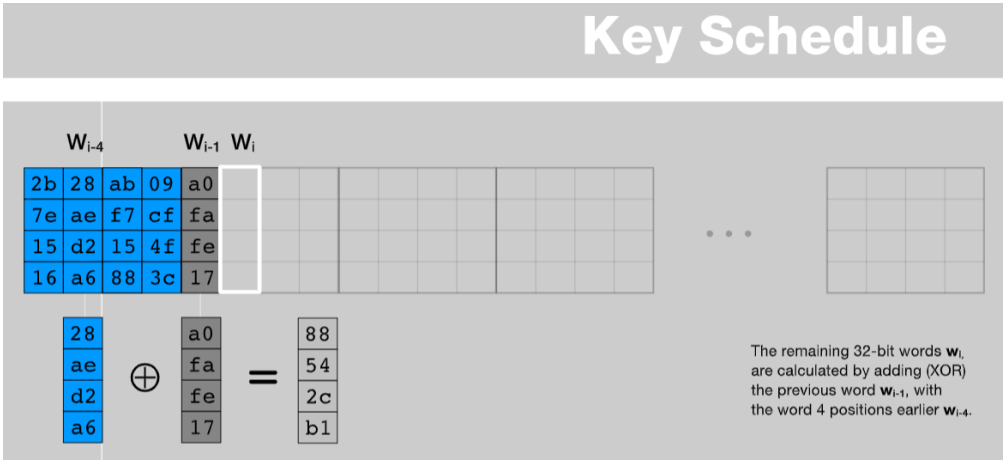

Figure 9: Key Schedule Part 4

2 CPU Implementations

2.1 RSA Implementation

This algorithm was originally implemented from scratch in C based on a lecture from CMU's CS15-251 course [3]. Two relatively large prime numbers were generated using an online tool [4]. These prime numbers and their product, the modulus, were then used to calculate the private and public exponents which were used in combination with the modulus to create the asymmetric public and private keys. Next, an encryption method was written which serially iterated through the inputted message in blocks. For each of these blocks, its equivalent encrypted output was calculated by taking the block value to the public exponent and subsequently apply the modulus. After each block was encrypted it was stored in a temporary array. A decryption method was then written which serially iterated through blocks of the temporary encrypted array, using the same block size as the encryption method. Each of these blocks were decrypted by taking their value to the private exponent and once again subsequently apply the modulus. The result of this decryption method was then saved to a text file to verify the validity of the encryption and decryption process.

2.2 AES Implementation

Our implementation provided support for 128-bit AES, which involved use of a 128-bit symmetric key, iteration for 10 rounds of data transformations, and operation on 4x4 byte matrix. Functions were implemented for key expansion, bit-wise matrix operations, lookup tables, all individual steps of AES, and the inverse functions of the three main AES steps (mix columns, shift rows, and sub bytes). The code was written using an online implementation in Go as a reference [5]. Since encryption and decryption were supported to operate on a 4x4 byte matrix, this meant 128-bits of data could be processed for a single thread iteration. Like RSA, this sequential block cipher is what was parallelized.

3 GPU Implementation

3.1 RSA Implementation

Since the encryption and decryption with RSA is applying the same operations in the same order, a single GPU kernel was created based on the CPU implementation with the ability to handle both encryption and decryption. This was done through the use of shared memory. The implementation contains two shared long values that are used by the GPU kernel for the encryption and decryption process. The first shared long is for the exponent value of the encryption or decryption key. This exponent value must be set prior to the kernels execution since it determines whether the data being passed to the kernel is being encrypted or decrypted. The second shared long is for the modulus value of the encryption and decryption key. This modulus value is the same number for both encryption and

decryption so it only needs to be set a single time in the code.

The GPU kernel itself takes in a pointer to an array of integers representing each character in the same input.txt file created for the CPU version. The GPU kernel also takes in another pointer to an array of integers the same length as the character array pointer. This second pointer is used to store the encrypted or decrypted output from the kernel and moved to the host after kernel execution to retrieve the kernel output. The third input the GPU kernel has is a long representing the total length of characters being encrypted or decrypted. This size variable is used to verify that the threads are staying within the bounds of the array of characters and not performing operations on extra spaces in memory. Using these inputs each thread on the GPU is assigned a message block of data to encrypt or decrypt allowing the message to be encrypted or decrypted in parallel which causes a performance speed up from the CPU version. Testing was accomplished by passing the kernel the contents of the randomly generated input.txt file to the kernel with the encryption key and then passing the first kernel output to the kernel again with the decryption key. This second kernel output was then saved to a new text file to verify the validity of the encryption and decryption process.

3.2 AES Implementation

The parallelized CUDA version of AES we developed uses identical kernels for both encryption and decryption. A kernel accepts the entirety of data to be transferred, and indexes the particular 128-bit chunk to process individually for each thread. The kernel checks explicitly that thread aligns with a chunk to process as there will be excess threads created. Because each chunk is independent, the input and output data is allocated in global memory rather than shared. Additionally, the lookup tables used in AES operations are easily loaded into the device's constant memory as they are statically allocated arrays.

4 Results

With the two encryption algorithms implemented in C, we moved to test both of the implementations that we had created in order to determine the ideal optimizations we could make to both programs. We conducted extensive profiling across both of our implementations, in order to determine where bottlenecks were located, and to help us determine which components should be implemented on the GPU for larger performance gains. For each of our implementations, we conducted time profiling on the encryption and decryption algorithms, testing various sizes of inputs. In addition, we profiled the total time to load in data, perform encryption, perform decryption, and move data to the host (as applicable).

4.1 RSA Profiling

To determine the overall performance of our RSA implementation, time profiling was conducted on the implementation in order to determine where speedups could most effectively be obtained, as well as to provide benchmarks between our CPU and GPU implementations.

Our code was broken into three main phases: file I/O, encryption, and decryption. In this case, file I/O needed to remain on the host due to restrictions on accessing I/O hardware from the GPU. In order to obtain a speedup from our CPU implementation, we decided to parallelize the encryption and decryption process. Being a block cipher, we are able to parallelize blocks of these operations instead of performing them all sequentially as done in the CPU implementation.

To determine the CPU performance, we time profiled the CPU implementation to determine the average wall time of the encryption and decryption methods, with results located in Table 1. For these tests, the public and private exponents were defined statically and not re-defined between trials.

File Size (bytes)	Encryption Time (ms)	Decryption Time (ms)
100	72.9140	66.3125
1000	633.5960	662.4540
2000	1283.0883	1341.1290
5000	3253.4970	3408.9306
10000	6272.3776	6747.9717
50000	31439.8594	33516.4922
100000	62714.8503	66684.9141

Table 1: Performance results from RSA CPU implementation

Next, we implemented a GPU version to focus on providing a speedup to the encryption and decryption code. This focused on parallelizing the block algorithm, with timing results present in Table 2.

File Size (bytes)	Encryption Time (ms)	Decryption Time (ms)
100	15.3449	16.3297
1000	27.2004	28.9561
2000	27.2072	28.9547
5000	27.2121	28.9657
10000	27.2160	28.9664
50000	54.3959	57.7565
100000	81.5753	84.8336

Table 2: Performance results from RSA GPU implementation

File Size (bytes)	Encryption Speedup	Decryption Speedup
100	4.7517	4.0609
1000	23.2937	22.8779
2000	47.1599	46.3182
5000	119.5608	117.6887
10000	230.4663	232.9584
50000	577.9817	580.3070
100000	768.7967	786.0675

Table 3: Speedups obtained between RSA CPU and GPU implementations

We found that we were able to get a significant speedup in the GPU implementation when compared to the CPU implementation. Being able to perform the algorithms in parallel instead of sequentially resulted in massive improvements, making the algorithms much more viable for larger files when using the GPU implementation. In addition, we found that the GPU wall time increased somewhat logarithmically (instead of linearly for CPU), making it much more scalable to larger files.

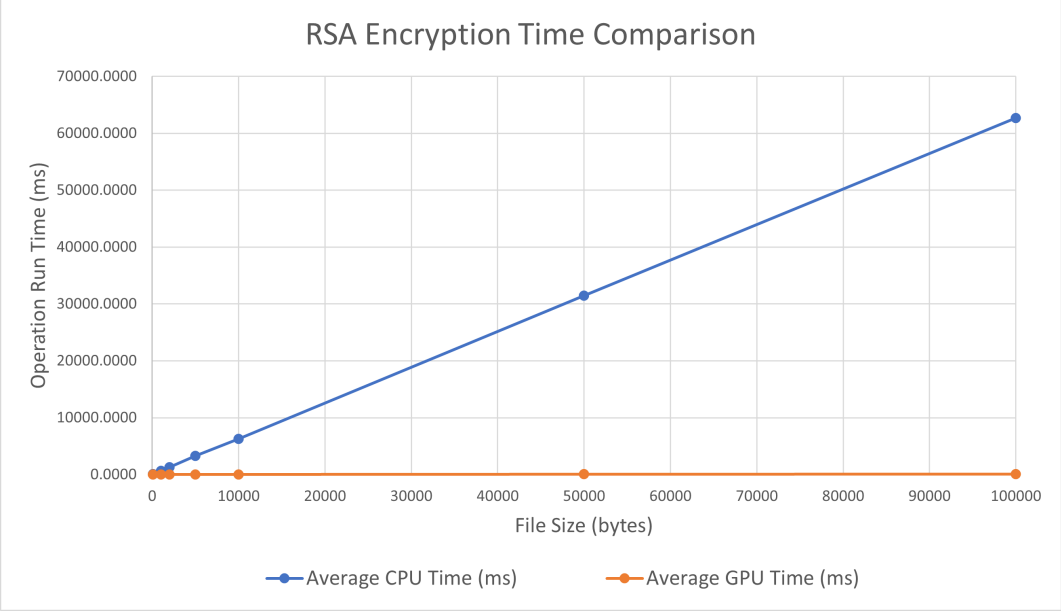

Figure 10: RSA encryption time on CPU vs GPU

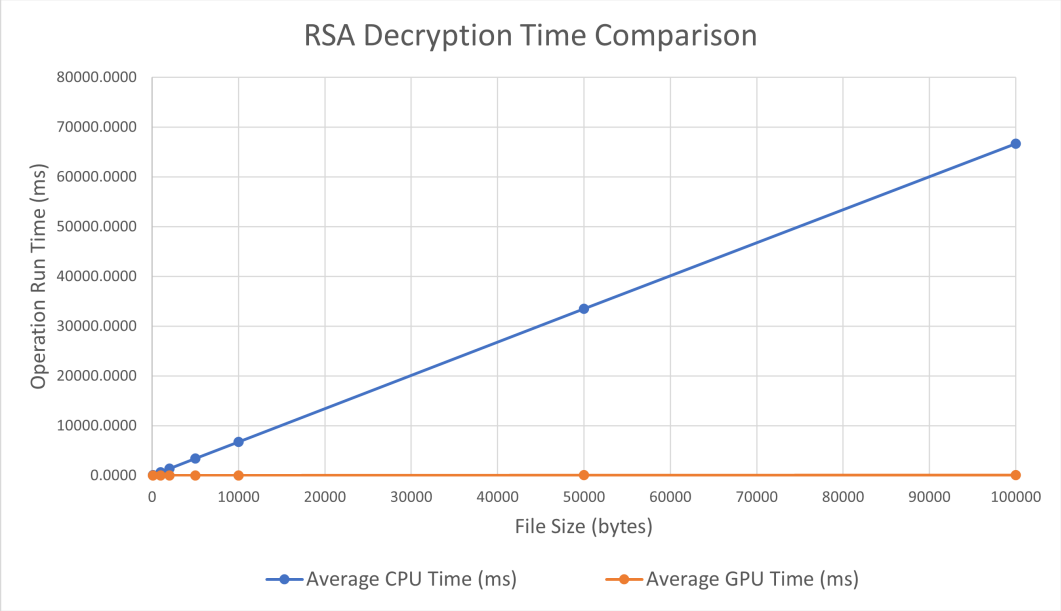

Figure 11: RSA decryption time on CPU vs GPU

Overall, we were able to gain large speedups in our GPU implementation exceeding 750 times, as shown in Table 3. We expect these values to be larger with larger input files, showcasing the power of GPU parallelization.

4.2 AES Profiling

Similar to our work with RSA, time profiling was conducted on our AES CPU implementation to determine overall performance and bottlenecks. To determine initial performance metrics for our implementation, we conducted initial timing tests on the encryption and decryption functions, as well as on the entire CPU implementation including data loading and reading. For each, we ran a total of 10 runs for each of the input byte lengths. One set of bytes of the pre-defined length was created, which was used as the input for all trials.

Performance of the CPU implementation is described in Table 4. Most of the wall time of the process was incurred during file I/O, as shown by the much larger values for the total time in each trial. However, since performing File I/O requires interfacing with the host, accelerations with these portions of the code cannot be performed and these operations need to remain on the host. As shown by Table 4, the encryption and decryption time roughly have a linear relationship with the file size; which has the potential to introduce large amounts of latency to programs working with files larger than 100 KB. These operations also rely on various matrix operations, which can be slow to execute on the CPU. In order to perform operations to enhance performance, we moved the encryption and decryption logic onto the GPU.

With potential accelerations in mind, we started implementation of two GPU kernels that would be used to perform encryption and decryption on our data. In order to determine the total speedup that these kernels were achieving, we recorded the total time each of the kernels

File Size (bytes)	Encryption Time (ms)	Decryption Time (ms)	Total Time (ms)
100	0.0446	0.0616	22.0610
1000	0.6394	0.7047	159.9384
2000	0.9148	1.3748	323.6053
5000	2.7351	3.4863	823.6949
10000	4.7744	6.1499	1642.0032
50000	22.9175	33.2069	8216.0195
100000	47.0137	67.4437	16426.5273

Table 4: Performance results from AES CPU implementation

spent executing in Table 5. In addition, we recorded the total time for the supporting GPU code to execute, including loading and transferring data to the device, as well as transferring the resultant data back to the host.

File Size (bytes)	Encryption Time (ms)	Decryption Time (ms)	Total Time (ms)
100	0.0425	0.0408	0.5625
1000	0.1366	0.1892	0.8657
2000	0.2066	0.3167	1.0567
5000	0.4637	0.7541	1.7589
10000	0.8938	1.4825	2.9433
50000	1.4498	2.4214	4.4968
100000	1.4551	2.4213	4.6360

Table 5: Performance results from AES GPU implementation

Computing the total speedups obtained for both our encryption and decryption functions in Table 6, we were able to obtain a roughly 30 times speedup when performing AES encryption on files 100 KB in size, with the speedup increasing near linearly for larger files. Both encryption and decryption were able to operate in under 5 ms for all tests conducted, increasing nearly logarithmically for larger files. This becomes much more scalable than the CPU solution, as larger files can be encrypted in a fraction of the time with this GPU implementation.

File Size (bytes)	Encryption Speedup	Decryption Speedup
100	1.0504	1.5077
1000	4.6800	3.7254
2000	4.4289	4.3416
5000	5.8979	4.6229
10000	5.3419	4.1482
50000	15.8069	13.7137
100000	32.3091	27.8543

Table 6: Speedups obtained between AES CPU and GPU implementations

Overall, we were able to optimize our CPU implementation of AES in order to achieve a significant speed improvement by utilizing CUDA for parallelized execution and calculation.

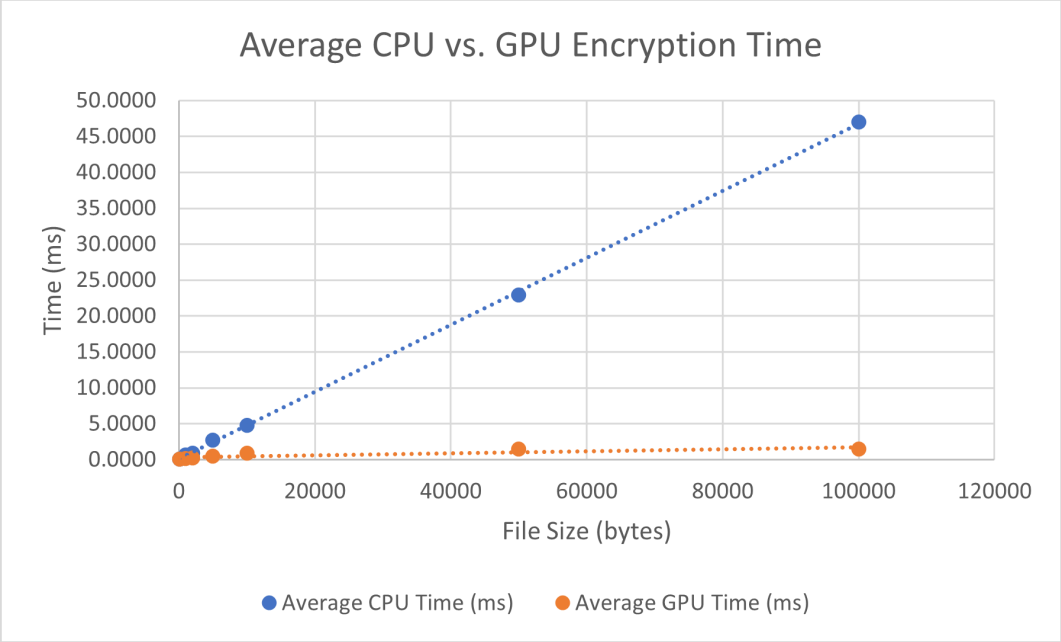

Figure 12: AES encryption time on CPU vs GPU

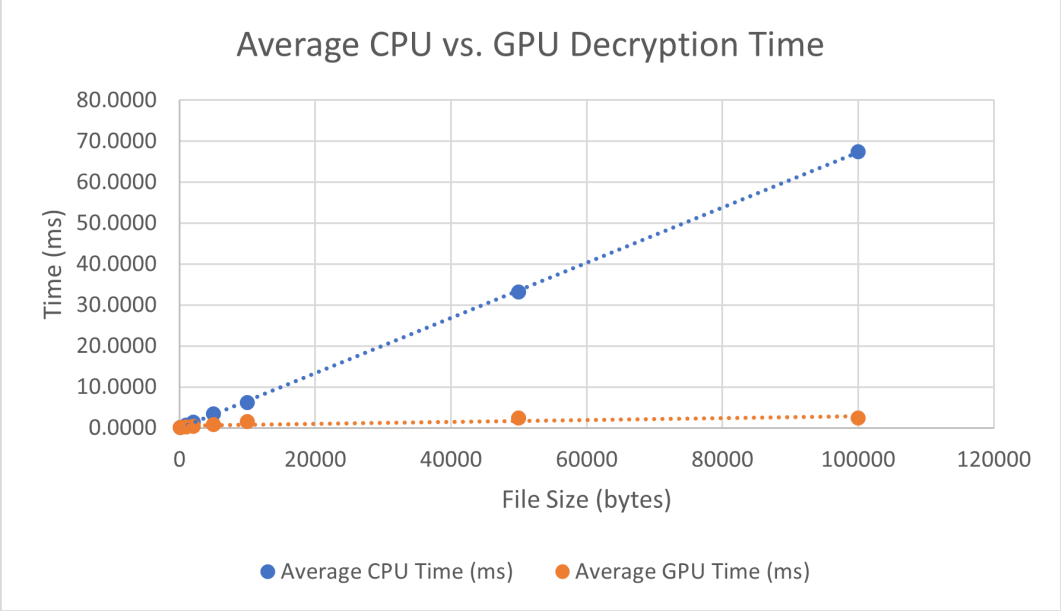

Figure 13: AES decryption time on CPU vs GPU

These accelerations are highlighted in Figures 12 and 13.

5 Discussion

5.1 RSA Challenges and Pitfalls

One of the challenges which was encountered when implementing RSA was associated with the bit length of the keys. In traditional RSA, a minimum of 512 bit prime numbers are typically used to generate the private and public keys to ensure maximum security. The initial planned solution for this project was to use a 1024 bit, however, it was quickly discovered that there's no data primitive data types which can host a key of this many bits. This meant that in order to use a traditional RSA key bit length of 1024, the key would need to be stored as an array. The decision was made to avoid using this approach as it would require refactoring all of the mathematical operations involved in RSA such that they would be operable with array based keys. This would've reduced the understanding of what the algorithm was mathematically accomplishing and subtracted from learning outcomes.

An additional challenge which was encountered when implementing RSA was associated with the encryption and decryption of blocks. When originally attempting to exponentiate the blocks using the private and public keys, it was found that the resulting value was large enough that it was unable to fit in a long long data type before the modulus could be applied to it. After researching how this issue was dealt with in traditional RSA, it was discovered that the exponentiation in RSA is taken by multiplying the block value in place for the exponent amount of iterations, followed by taking the modulus of the block value after every iteration. By using this technique, the issue was resolved.

5.2 AES Challenges and Pitfalls

With the general goal being to parallelize these encryption techniques, a frequent question was which the algorithm's aspects should be optimized. This report focuses on block cipher parallelization, but in the case of AES, the commutative property of certain steps would allow their subroutines to be placed in a kernel and operate on a subsection of a chunk.

References

[1] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," National Science Foundation grant MCS76-14294, and the Office of Naval Research grant number N00014-67-A-0204-0063. Author's Address: Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 1978 [Online]. [Online]. Available: https://people.csail.mit.edu/rivest/Rsapaper.pdf

[2] E. Zabala, "The rijndael animation," Accessed Nov. 10, 2021 [Online], 2008 [Online]. [Online]. Available: https://formaestudio.com/rijndaelinspector/archivos/Rijndael_Animation_v4_eng-html5.html

[3] V. Adamchik, "Great Theoretical Ideas in CS, Cryptography and RSA," Accessed Nov. 10, 2021 [Online]. [Online]. Available: https://www.cs.cmu.edu/~arielpro/15251/Lectures/lecture24.pdf

[4] W. Buchanan, "Generating random prime with n bits," Asecuritysite, Nov. 10, 2021 [Online]. [Online]. Available: https://asecuritysite.com/encryption/getprimen

[5] krishna Sundarram, "Implementing aes," 2015. [Online]. Available: https://blog.nindalf.com/posts/implementing-aes/