An Exploration of Parallelization Techniques for Cryptographic Algorithms

Arjun Manjunatha Rao
Shreyas Belur Sampath

May 5, 2023

1 Summary

We have implemented parallelization techniques for the AES [DR99], ChaCha [B+08], and ED25519 [BCJZ21] cryptographic algorithms by utilizing both OpenMP and CUDA. In addition, we have conducted a thorough analysis of the performance on the GHC machine cluster, local machines, and Google Colaboratory. As a result of our analysis, we have extracted speedup values that reflect the performance improvements achieved through our parallelization techniques and provided justifications.

2 Background

2.1 The AES Algorithm

The Advanced Encryption Standard (AES) is a widely used symmetric-key encryption algorithm that operates on blocks of data, with each block being 128 bits long. The algorithm itself consists of several phases of encryption, which are explained below:

- Key expansion: This is the phase where an initial secret key is expanded into a larger size based on the number of rounds in the encryption. The new key is split across different rounds, where the number of rounds depends on the secret key size. For a secret key size of 16 bytes, the number of rounds in the encryption would be 10.
- Substitution: In this phase, a pre-defined substitution table known as the S-box replaces each block's byte with a corresponding byte from the S-box.

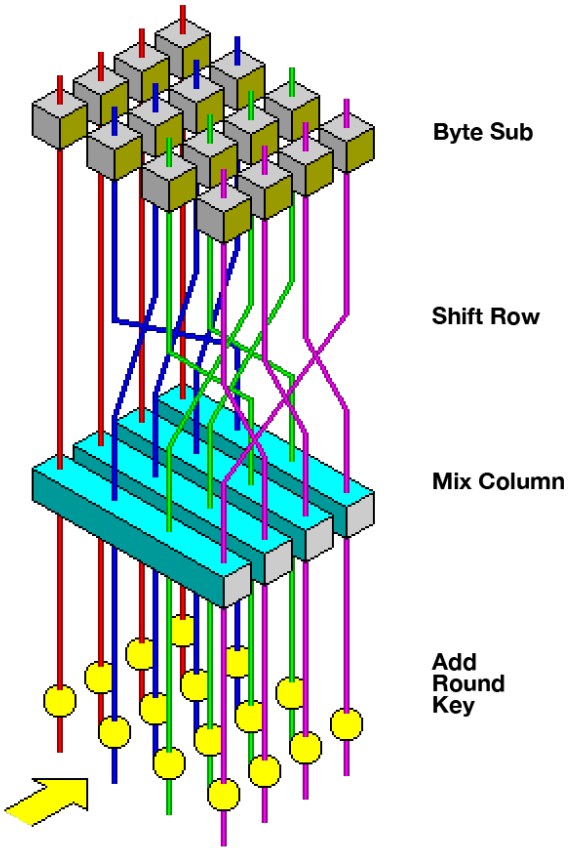

Figure 1: A visualization of one AES round of encryption

- Permutation: In this phase, an operation known as ShiftRows is performed, which involves shifting the rows of the block cyclically by a certain number of bytes.
- Mixing: The mixing component is called the MixColumns operation, and involves applying a Galois-field matrix multiplication to the columns of the block.
- Addition of Round Key: This is the phase where the round key is added to the block before the next set of operations begin

2.2 The ChaCha Algorithm

The ChaCha algorithm is a fast and simple cryptographic algorithm, which produces a stream of pseudo-randomly generated numbers. This stream is used to encrypt the data, earning it the category of a stream cipher. The main components of the ChaCha algorithm are described below:

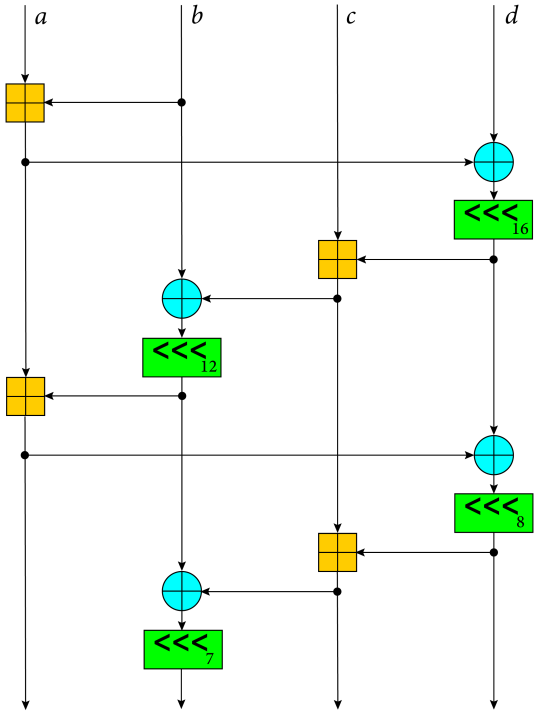

Figure 2: A magnification into one ChaCha20 quarter-round

- Key initialization: In the key initialization phase, the initial 356-bit key is appended with a counter, nonce, and a constant in the form of a 4x4 matrix, where every entry is a value of size 32-bits. The constant used for the setup is "expand 32-byte k", which provides multiple advantages, including the prevention of an all-zero block during randomization. The counter is usually set to 0 in this phase.

- Quarter-round: A quarter-round is a series of operations applied on the state matrix which was generated in the key setup phase. The quarter-rounds are applied across both the columns and diagonals of the state matrix, and repeated 10 times, in order to give a total of 20 rounds of shuffling to the key values.

- Keystream XORing: The obtained state matrix is read in a particular pattern and XORed with the plaintext till the generated keystream is used up, after which the next stream is generated by repeating the operations mentioned above.

2.3 The ed25519 Algorithm

The history of cryptography can be divided into two distinct periods: the classical era and the modern era, with a significant transition point taking place in 1977. This pivotal year marked the introduction of two groundbreaking cryptographic methods: the RSA [Mil09] algorithm and the Diffie-Hellman key exchange algorithm [DH22]. Regarding RSA, the simpler operation involves multiplying two prime numbers, while the more intricate counterpart necessitates factoring the resulting product into its constituent primes. This type of algorithm, which is facile in one direction but challenging in the other, is commonly referred to as a "trapdoor function." An elliptic curve denotes the collection of points that satisfy a precise mathematical equation, which can be represented as follows: $y^2 = x^3 + a * x + b$.

To establish an elliptic curve cryptosystem, one can designate a prime number as a limit, select a curve equation, and specify a public point on that curve. In this system, a private key is represented by a numerical value, while a public key corresponds to the public point subject to a scalar multiplication of the private key. The operation of obtaining the private key from the public key in an elliptic curve cryptosystem is commonly referred to as the elliptic curve discrete logarithm function, which serves as the desired trapdoor function.

The operations we look at in the Edwards curve cryptosystem are as follows:

- Key pair generation: This involves the selection of a random 256-bit value as the private key. This private key is then utilized to generate the corresponding public key through the process of point multiplication on the ed25519 elliptic curve. The resulting public key is a 256-bit value that is obtained by hashing the point derived from the private key and is used to verify digital signatures created by the corresponding private key.

- Signature: A digital signature is created by computing a hash of the message being signed and then performing a series of mathematical operations on that hash using the signer's private key and the ed25519 elliptic curve. The resulting signature is a fixed-length value that is unique to both the message and the signer's private key.

- Verification: Verifying a digital signature involves recomputing the hash of the signed message and applying a series of mathematical operations to the signature using the sender's public key and the ed25519 elliptic curve. If the resulting value matches the hash of the message, the signature is considered valid. This process allows the recipient to confirm the authenticity of the message and the identity of the sender, without needing access to the sender's private key.

- Scalar Addition: Scalar Addition in ed25519 refers to the process of adding a scalar value to an elliptic curve point. This operation involves multiplying the scalar value

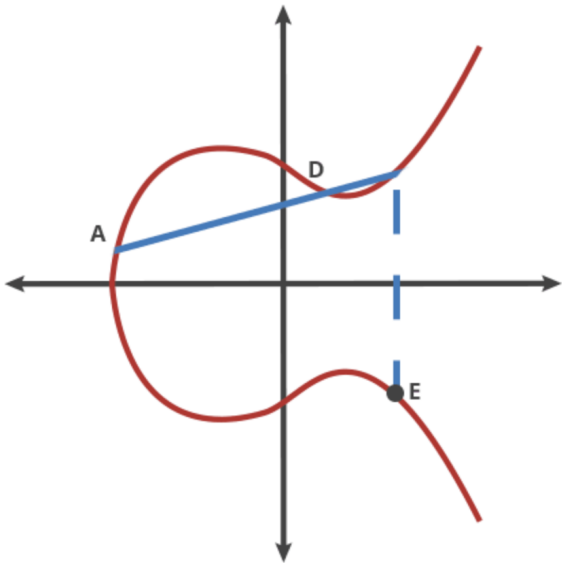

Figure 3: Scalar addition part of the trapdoor function for the elliptical curve

with the standard base point of the ed25519 elliptic curve to obtain a new point, and then adding the original point to the new point. The resulting point represents the sum of the two input points and is also a point on the same elliptic curve.

- Key Exchange: Key exchange in ed25519 is a process that enables two parties to securely establish a shared secret key over an insecure communication channel. This is achieved by utilizing the properties of the ed25519 elliptic curve, where each party generates a private-public key pair and sends their public key to the other party. The two parties then perform scalar multiplication on their own private key and the other party's public key, respectively, to derive a shared secret key that is identical for both parties. This shared key can then be used for subsequent communication using symmetric encryption, providing confidentiality and authenticity for the exchanged data.

3 Approach

3.1 The AES Algorithm

3.1.1 OpenMP exploration

The most computationally intensive part of the AES algorithm is the encryption round, where several operations such as substitution, row shifting, column mixing, and addition of

round keys are performed. The initial implementation [Mey23] we selected for the project only encrypted one message with a single block of data. Since the encryption round is a sequential process that requires intermediate results from each iteration for the subsequent iteration, utilizing the "parallel for" construct did not yield any performance gains.

To overcome this limitation, we expanded the implementation to encrypt multiple messages with multiple blocks of data. Both of these enhancements can be parallelized, resulting in a substantial increase in speed. The optimal number of threads played a crucial role in obtaining a desirable speedup since a high number of threads can generate significant overhead, particularly for cryptographic algorithms, which are characterized by very low execution times.

3.1.2 CUDA exploration

For the CUDA implementation of the AES algorithm, a strategy similar to that of OpenMP was adopted. To enable parallelization over messages and blocks in a message, we created a two-dimensional array to store all the messages. However, creating a two-dimensional array on the GPU can be a complex process. We worked around this issue by flattening the 2D array and using an offset to access messages. This approach did not complicate the kernel since each message could be accessed by utilizing the block ID of the thread. The number of blocks and threads in each block were determined by the number of messages and message length, respectively. Each GPU block worked on one message, while every thread in a GPU block worked on a block of the corresponding message. It should be noted that the maximum number of threads in a GPU block is 1024, which limited the number of blocks in a message. However, we could remove this barrier by assigning a thread to work on multiple blocks. Doing so would require computation beforehand to get the lower and upper limits of the block indices to compute.

During our implementation of cryptographic algorithms on the GPU, a critical decision we had to make was regarding the key expansion process. In particular, we had to decide whether to perform the key expansion on the GPU or the CPU. Performing the key expansion on the GPU would require transferring all the necessary data to the GPU, which could result in substantial overhead. On the other hand, performing the same computation on the CPU would be slower but would not have the same overheads. Since we used a single key to encrypt all the messages, the key expansion had to be performed only once. After careful consideration, we decided to perform the key expansion on the CPU since the copy overheads to the GPU would be significant and almost nullify any potential speedup we could obtain

by using the GPU. This decision was made based on the trade-off between the overhead of data transfer and the potential speedup that could be achieved.

3.2 The ChaCha Algorithm

3.2.1 Baseline implementation

The initial implementation of the ChaCha algorithm was sourced from Ginurx's implementation of the same algorithm [Zhu20]. However, the code contained certain optimizations like loop unrolling that we had to eliminate to accurately gauge the impact of loop parallelizers, such as OpenMP, on performance. Our objective was to determine the extent to which parallelization could improve performance as compared to a regular sequential implementation.

3.2.2 OpenMP exploration

As a stream cipher, ChaCha generates a new block of the keystream only when the current block is completely used up to encrypt the message. Due to the stream nature of the cipher, it is impossible to determine the length of the message beforehand, making precomputation of keystream blocks impractical. The only feasible area where OpenMP could be utilized to obtain performance improvements was the encryption of different messages. We accomplished this by parallelizing the process using the "parallel for" construct using 16 threads.

3.2.3 ISPC exploration

We also investigated the use of an ISPC implementation for the ChaCha algorithm, which involved adopting an SPMD (Single Program, Multiple Data) approach. The approach centered on precomputing a predetermined number of blocks in the keystream, based on the SIMD (Single Instruction, Multiple Data) width of the CPU. However, it is worth noting that this approach works optimally when the number of blocks in the message is a multiple of the machine's SIMD width. Otherwise, there would be unnecessary computation.

Despite our efforts to implement the ISPC approach for ChaCha, we did not obtain the desired results. We ran into several issues specific to the ISPC compiler, which ultimately hindered our progress. As a result, we had to abandon this approach and focus on other techniques to optimize the algorithm's performance.

3.3 The ed25519 Algorithm

3.3.1 Baseline implementation

The baseline for our implementation was borrowed from a portable implementation [Pet22] of Ed25519 based on the SUPERCOP "ref10" implementation. All the code part of the repository was pure ANSI C without any dependencies, except for the random seed generation which uses standard OS cryptography APIs.

3.3.2 CUDA exploration

The operations involved in ed25519, such as point multiplication and scalar addition, are highly interdependent and cannot be easily parallelized. This means that attempting to perform these operations in parallel using CUDA may not result in significant performance improvements and may even lead to increased overhead due to the synchronization of threads and cache contention.

Additionally, ed25519 operations involve multiple data dependencies and a large amount of memory access, which can lead to performance degradation if the data is not correctly aligned in memory. CUDA, while effective in many parallel computing scenarios, may not be well-suited for the specific memory access patterns and data dependencies required for efficient ed25519 operation parallelization.

However, ed25519 operations for different messages in the context of verifying digital signatures can be batched together. In this scenario, the signer can batch multiple signatures for different messages into a single batch, which can then be verified together in a single operation. This can result in significant performance improvements, as the computation of ed25519 operations is highly optimized for batch processing.

Keeping this in mind, we parallelized five important operations as part of the cryptosystem. For scalar addition, kernels were written to execute the operation for a batch of private keys simultaneously. Two versions of this batch kernel were written, one using the same scalar to be added to all the keys, while the other involving multiple scalars in an array with a mapping of which ones to be added to the respective keys. Similarly, kernels were written for key exchange and key pair generation such that multiple pairs of independent keys could be parallelly generated or exchanged. Finally, the sign and verify methods were also parallelized using separate kernels. The batch signature kernel involved batching multiple secretes simultaneously while the batch verification kernel verified these independent key pairs in parallel. In addition to these, kernels for multiple key pairs were also written to support the generic case of signature and verification of independent secrets and signatures with corresponding key pairs.

4 Results

4.1 The AES Algorithm

We achieved substantial speedups on the AES algorithm. For the OpenMP implementation, the speedup graphs plotted are shown below. There are two graphs, one with changing message lengths, implying changing number of blocks, and the other with changing number of messages. Note that the message length was fixed at 64 for the second graph.

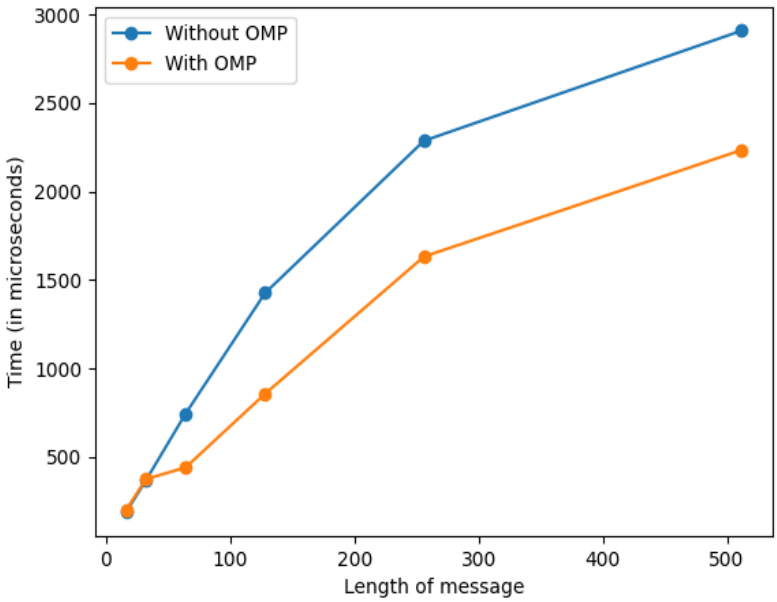

Figure 4: OpenMP speedup for AES with varying message length

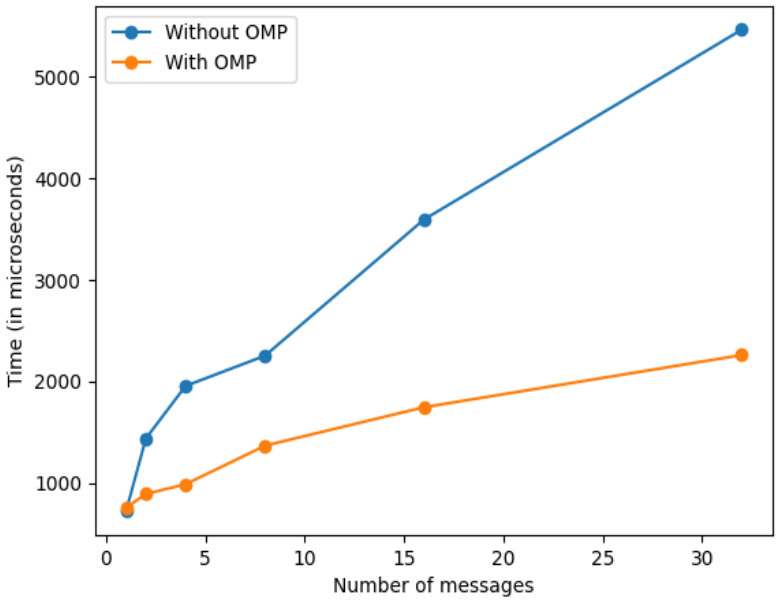

Figure 5: OpenMP speedup for AES with varying number of messages

We observed that the speedup achieved by parallelizing the outer loop of the AES algorithm was greater over the number of messages as compared to that over the message lengths. This outcome was expected, as the parallelization of the outer loop resulted in the distribution of the entire inner loop among the threads, as opposed to block-level parallelization achieved by parallelizing the inner loop. It was noted that the extent of computation that was parallelized played a crucial role in determining the degree of speedup achieved. Since all messages have the same length, task scheduling would not be beneficial as there is no significant difference in the workload distribution across threads.

For the CUDA implementation, the execution times are shown in the table below for varying number of messages. Note that since the number of threads in the GPU block were maxed out to 1024 as previously mentioned, the message length would have not much variance.

Table 1: AES speedup on an NVIDIA GeForce RTX 2080 GPU

Number of Messages	GPU Time (in μs)	CPU Time (in μs)
1	16.45	725
16	16.83	3776
64	19.16	9535
128	23.81	17150
256	30.72	31703

The obtained results show that the GPU times are remarkably low, yielding a speedup of nearly 50x even for the smallest number of messages (1). The GPU's massive number of cores and its architecture are responsible for this behavior. Since the messages are divided among blocks, which can be processed by different cores, the time taken is minimal for low message counts, with the time increasing as the number of messages grows. On the other hand, the sequential implementation on the CPU performs the worst, with no form of parallelization. However, though the speedup with GPU computation increases as the amount of memory being processed increases, the cost of copying the output data back to the CPU memory also increases and can lead to negative effects at high values.

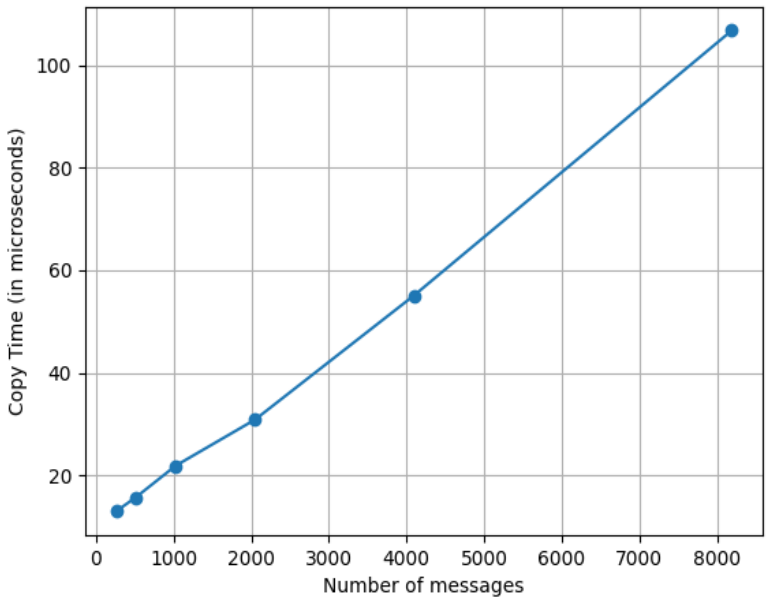

Figure 6: GPU to CPU memory copy time vs Number of messages

Note the steady doubling of the copy times as the number of messages increases.

4.2 The ChaCha Algorithm

With the OpenMP implementation, the ChaCha algorithm was found to perform better than its sequential counterpart. The speedup graph is shown below.

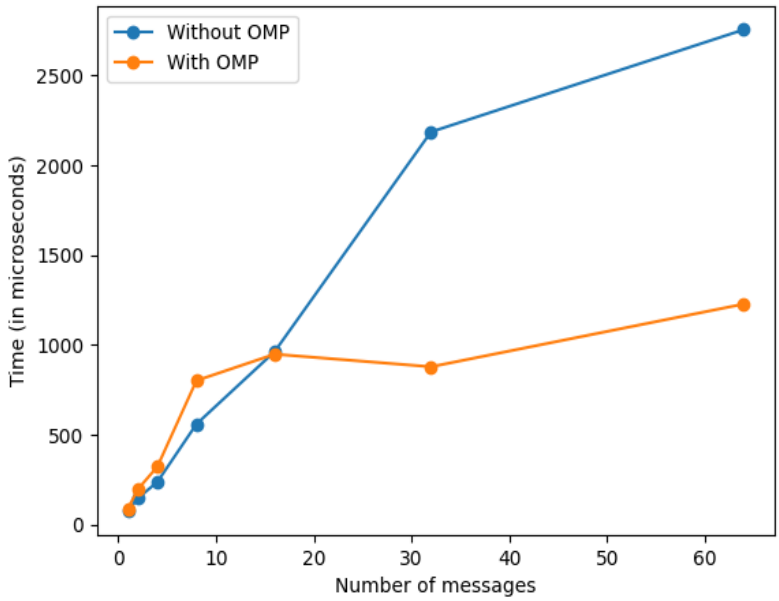

Figure 7: OpenMP speedup for ChaCha20 with varying number of messages

The initial impact of using the OpenMP implementation of ChaCha is a decrease in performance, which is primarily due to the inherent lower computation time of the sequential ChaCha, which amplifies the overhead associated with thread creation in OpenMP. However, as the number of messages increases, the speedup also increases and the gap widens.

In addition to the speedup graph, we analyzed the cache references with and without the OpenMP implementation for the ChaCha algorithm. The cache misses for the OpenMP implementation are consistenly lesser when compared to the vanilla version. Some numbers to support this claim (extracted from perf-stat):
With OMP:

- Cache misses: 74302
- Cache references: 250824
- Miss rate: 29.623%

Without OMP:

- Cache misses: 52002
- Cache references: 129818
- Miss rate: 40.058%

One possible explanation for this phenomenon is that when multiple threads execute in parallel, there is a reduced likelihood of data eviction, resulting in a lower cache miss rate. However, since there are more threads involved in the computation, the number of cache references also tends to be higher.

4.3 The ed25519 algorithm

With the help of kernels written for batching operations for parallel execution, a clear and substantial speedup was seen for four of the operations when the time taken per application of operation was measured. The experiments were run for a range of batch sizes/total number of messages. The initial configuration tested 262144 messages divided into 1024 blocks with 256 threads. The final configuration included 32 threads with the split as 1 block with 32 threads. The configurations were tweaked such that, each configuration with half the number of messages as the previous one was tested. The first few dimensions included halving the number of blocks and further halving the number of threads once the number of blocks reached 1. The results for each of these configurations expressed by the product of a number of blocks and threads as a total number of messages is as seen below in the graphs below. Through the empirical results, it was quite clear that the speedup for all operations stagnated or flattened after reaching a certain number of messages. In terms of the Number of Blocks * Number of Threads, keeping the number of threads at 256 and halving the number of Blocks every time from 1024 to 16 had almost no impact on the execution time for Key generation. A similar experiment showed that 32 blocks and above had similar runtime for key exchange while the threshold in terms of the number of blocks for signature and verification was 16. The experiment shows that on increasing the number of messages beyond these values, the data parallelism for the increased number of messages is limited. However, on further decreasing the number of blocks up to 1 and then halving the number of threads (effectively halving the number of messages/keypairs), we see that the run time per keypair/message is almost doubling. This linear trend is clearly visible beyond the threshold number of messages as indicated by the graphs. Finally, looking at each operation we see that verification is quite a heavy operation followed by Key exchange compared to others. The peak speed-up for each of the operations was as follows (All GPU times are acquired by running on an Nvidia Tesla T4 while CPU time is run time on Intel Pentium B970 @ 2.3GHz used by the baseline):

- Key Generation - 512 * 256 keys batched gives 302 times speedup on the GPU over the sequential version run on CPU considering a single operation from the batch.
- Key Exchange - 256 * 256 keys batched gives 550 times speedup on the GPU over the

sequential version run on CPU considering a single operation from the batch.

- Signature - 512 * 256 keys batched gives 273 times speedup on the GPU over the sequential version run on CPU.
- Verification - 1024 * 256 keys batched gives 347 times speedup for a single key pair and 72 times for multiple key pairs on the GPU over the sequential version run on the CPU.

In ed25519, for example, signing a message involves a single-point multiplication operation with a single secret key, whereas verifying a signature with multiple public keys involves multiple point additions and multiplications with multiple public keys. Each additional public key that must be used in the verification process increases the amount of computation required, which can result in a longer processing time.

Additionally, it's worth noting that the verification process is typically performed by a third party, such as a recipient or a verifier, whereas the signing process is typically performed by the message sender. The third party may need to verify multiple signatures with multiple public keys, which can further increase the overall verification time.

These reasons may help explain the apparent increase in time for verification and how the use of multiple key pairs affects execution differently from how the signature is affected.

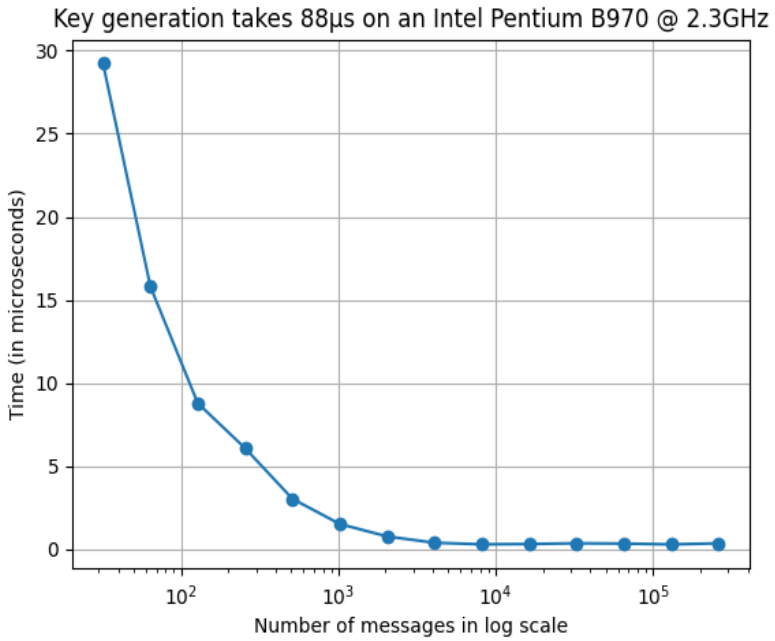

Figure 8: Key generation on NVIDIA Tesla T4 GPU

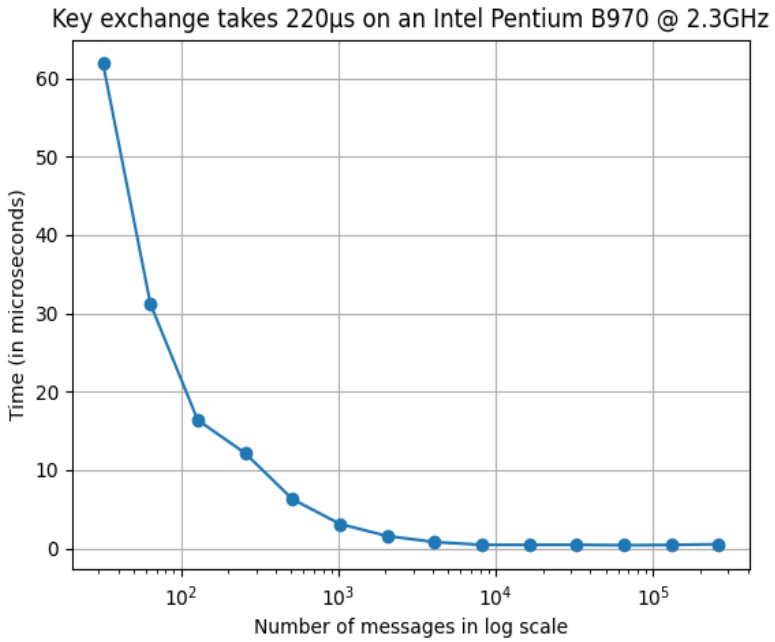

Figure 9: Key exchange on NVIDIA Tesla T4 GPU

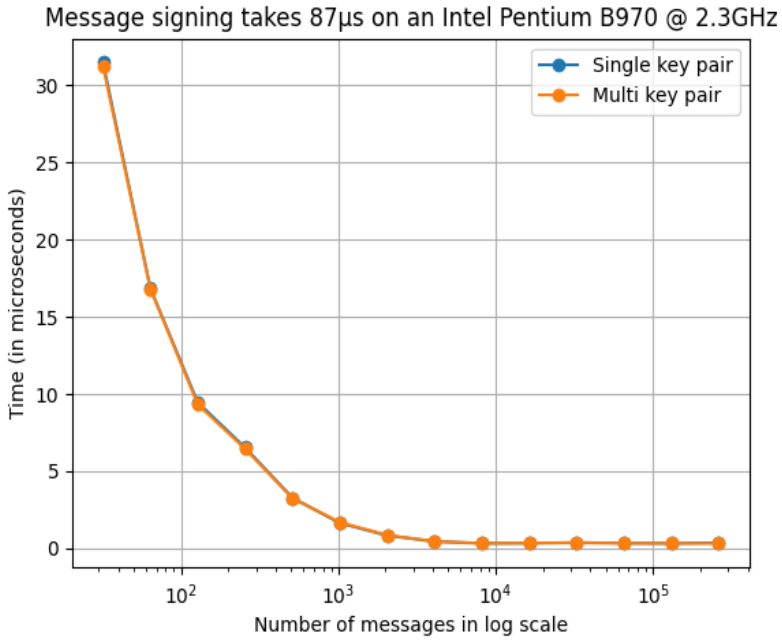

Figure 10: Message signature on NVIDIA Tesla T4 GPU

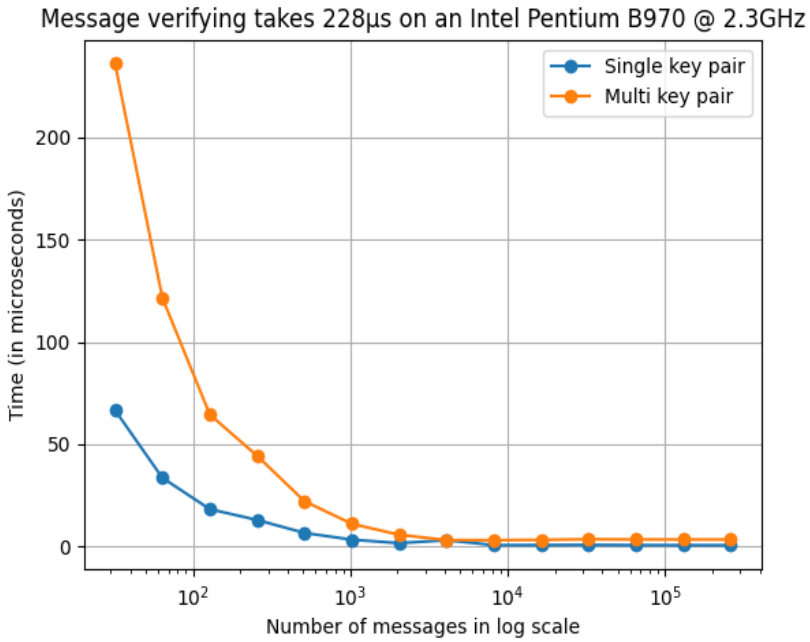

Figure 11: Message verification on NVIDIA Tesla T4 GPU

5 Work Distribution

Credit distribution: 50%-50%

- AES - Research, OpenMP, and CUDA implementation (Shreyas)
- ChaCha - Research and Initial Parallelization and Optimizations (Arjun)
- ChaCha - OpenMP implementation (Shreyas)
- Ed25519 - Research, Initial Parallelization, and CUDA implementation (Arjun)
- Performance analysis and Benchmarking (Arjun and Shreyas)
- Write-up (Arjun and Shreyas)

References

[B+08] Daniel J Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Citeseer, 2008.

[BCJZ21] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. The provable security of ed25519: theory and practice. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1659–1676. IEEE, 2021.

[DH22] Whitfield Diffie and Martin E Hellman. New directions in cryptography. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*, pages 365–390. 2022.

[DR99] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[Mey23] Meysam. aes-in-c. https://github.com/m3y54m/aes-in-c, 2023.

[Mil09] Evgeny Milanov. The rsa algorithm. *RSA laboratories*, pages 1–11, 2009.

[Pet22] Orson Peters. Ed25519. https://github.com/orlp/ed25519, 2022.

[Zhu20] Jeffrey Zhuang. chacha20-c. https://github.com/Ginurx/chacha20-c, 2020.