GPU-Accelerated AES Algorithm

E4750_2024Fall_<CQCQ>_report
<Qiuhong, Chen, qc2335>
Columbia University

Abstract

AES is a widely used encryption protocol for web communication and storage. The conventional CPU-based AES implementation has poor performance when dealing with large data, while the rise of GPU provides an approach to solve this problem. This project investigates the possible acceleration of the AES128 algorithm using GPU and CUDA platforms, including T-table, constant memory, block shared memory, and warp shuffling. We conduct experiments with 5 implementations with different datasets and find the anticipated results. The project code is in the GitHub repository [1].

Key Words*: block cipher, AES, parallel, GPU*

1. Overview

1.1 Problem in a Nutshell

Today, data encryption is ubiquitous in web communication and storage, where block cipher plays a significant role in encrypting arbitrary lengths of data. Among the block ciphers, Advanced Encryption Standard (AES) combines speed and security and becomes one of the industrial standards. However, conventional CPU-based implementations of AES algorithms have poor performance when dealing with large data. This is because CPU-based implementation is bounded by computation resources which neglects that most operations involved in AES are homogeneous. On the other hand, GPU provides abundant computation cores, which are aimed at converting computation-bounded problems into memory-bounded problems. Thus, GPU acceleration should be a good solution.

In this project, we use the Nvidia GPU and CUDA platforms to accelerate both the encryption and decryption process of the AES algorithm. We follow the approaches adopted by prior experiments, primarily A. Abdelrahman et al. [2], and we expect to produce similar or anticipated results on our machine.

1.2 Prior Work

Di Biagio et al. implemented AES-CTR using GT8800. The result shows that the coarse-grained implementation, where a block is computed by a single thread, performs better than the fine-grained implementation, where multiple blocks collaborate to compute one block. They also found speedup to put T-box in shared memory instead of constant memory [3]. Thus, in this project, we will focus on coarse-grained implementations of the AES algorithm.

A. Abdelrahman et al. implemented AES-128 ECB Encryption on three different GPU architectures (Kepler, Maxwell, and Pascal). The results show a new speedup with 32bytes/thread granularity. The work also demonstrates the use of warp shuffling to utilize the register file for inter-thread communication [2]. Thus, in this project, three memory optimizations will be considered: constant memory, shared memory, and warp shuffling.

2. Description

2.1. Goal and Objectives

The goal of this project is to implement the possible accelerations of the AES128 algorithm and compare their performance. We first develop a standard AES128 serial implementation, where the output of each step is identical to the online calculator[5]. Second, we develop a T-table optimized fast serial implementation. Third, we develop a naive parallel T-table implementation, where all tables are stored in the constant memory. Fourth, we develop an improved parallel implementation, where the table is loaded to shared memory. The last implementation puts the round keys in the register file and accesses them by warp shuffling, an inter-thread communication interface supported by CUDA.

2.2. Problem Formulation and Design

Mode	Description	Parallelization potential
Electronic Codebook (ECB)	For a given key, the forward cipher function is applied directly and independently to each block of the plaintext.	Suitablefor parallelization
Cipher Block Chaining (CBC)	Each successive plaintext block is exclusive-ORed with the previous output/ciphertext block to produce the new input block. The forward cipher function is applied to each input block to produce the ciphertext block.	Decryption is suitable for parallelization
Cipher Feedback (CFB)	The feedback of successive ciphertext segments into the input blocks of the forward cipher to generate output blocks that are exclusive-ORed with the plaintext to produce the ciphertext ,and vice versa.	Not suitable for parallelization
Output Feedback (OFB)	The iteration of the forward cipher on an IV to generate a sequence of output blocks that are exclusive-ORed with the plaintext to produce the ciphertext, and vice versa.	Not suitable for parallelization
Counter (CTR)	The application of the forward cipher to a set of input blocks, called counters, to produce a sequence of output blocks that are exclusive-ORed with the plaintext to produce the ciphertext, and vice versa.	Suitable for parallelization
XEX-based tweaked-codebook mode with ciphertext stealing (XTS)	IEEE standard, IEEE Std 1619-2007, which a method of encryption for data stored in sector-based devices	Suitable for parallelization

AES128 is a symmetric encryption algorithm from the AES family that uses a fixed block size of 128 bits and a 128-bit key. As a block cipher, it can divide the plain text into multiple blocks and thus encrypt the plain text of arbitrary length. According to how these blocks influence each other, there are several block cipher modes, which provide different levels of security, as listed in the above figure [2]. Among these modes, the most fundamental ECB mode encrypts each block independently and is suitable for parallelization. Considering ECB mode is easy to implement (although less secure) and can extend to the practical CTR mode, we adopt ECB mode in our implementation.

As the algorithm of AES128 is well elaborated in previous works, we will illustrate it in a more concise way and focus more on aspects concerning our optimizations.

In AES128, both encryption and decryption concern 11 rounds. Before that, we first need to expand another 10 keys from the original key. These keys are precomputed in the host machine and transferred to GPU memory at runtime, so we won’t elaborate on this part.

Four kinds of operations are involved in each round of encryption: byte substitution, byte permutation, matrix multiplication, and round key addition. Rounds are identical except for the first and last rounds. In decryption, the operations are inversed and performed in the reverse order.

- Byte Substitution. An 8-bit-to-8-bit bijection map (or lookup table) substitutes a byte to another byte.
- Byte Permutation (or ShiftRows). All 16 bytes in a block are re-permuted in a certain way.
- Matrix Multiplication (or MixColumns). The 16 bytes in a block are seen as a 4×4 matrix, which is multiplied by another 4×4 matrix in one of the Galois Fields of 2^8. As this part concerns matrix multiplication and Gaoilis Field multiplication, it is computationally expensive.
- Round-key Addition. The 128-bit block is exclusive-ORed with the corresponding round key.

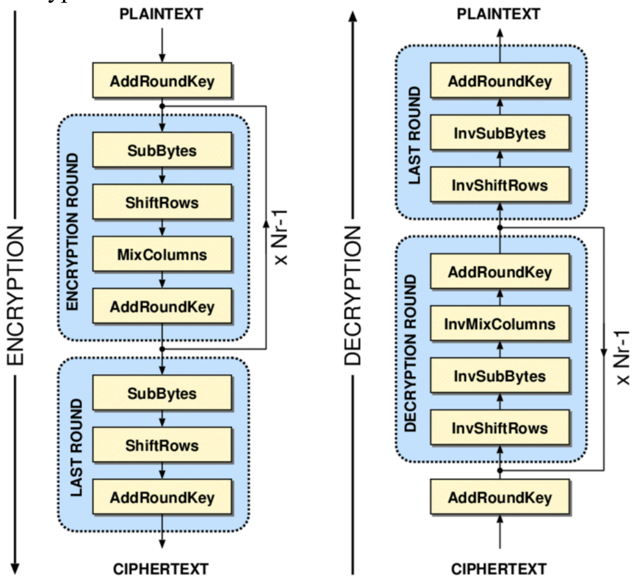
In the fast AES, four lookup tables substitute the four operations. Each lookup table is a 16-bit-to-16-bit map. The operations of each round except for the first and last round are defined as Equation (1):

$$e_j = T_0[a_{j \bmod 4}] \oplus T_0[a_{(j+1) \bmod 4}] \oplus T_0[a_{(j+2) \bmod 4}] \oplus T_0[a_{(j+3) \bmod 4}] \oplus K_j$$

Where $j \in \{0, 1, 2, 3\}$, e_j is the j-th word in the input block, a_j is the j-th word in the output block, and K_j is the j-th word of the round key.

The last round only concerns byte substitution and byte permutation, so we can keep the standard implementation for this part without sacrificing performance.

The overall procedure can be expressed with the flowchart [4]. There are 11 rounds for both encryption and decryption. For encryption, the first round only concerns AddRoundKey. The middle 9 rounds can be optimized by T-table. The last round does not include the MixColumn (or Byte Permutation) operation compared to the middle rounds. The decryption is simply the reverse process of encryption.



2.3 System and Software Design

According to the algorithm described above, we can come up with the following Pseudocode.

For standard AES encryption algorithm:

```
Input: a 16-byte state, a 4-word Key
Output: a 16-byte state
RoundKey=KeyExpansion(key)
AddRoundKey(state,RoundKey[0])
for i=0 ; i<10 ; i+=1
        ByteSubstitution(state,sbox)
        BytePermutation(state)
        MatrixMultiplication(state)
        AddRoundKey(state,RoundKey[i])
ByteSubstitution(state,sbox)
BytePermutation(state)
AddRoundKey(state,RoundKey[10])
```

For fast AES encryption algorithm:

```
Input: a 16-byte state, a 4-word Key
Output: a 16-byte state
RoundKey=KeyExpansion(key)
AddRoundKey(state,RoundKey[0])
for i=0 ; i<10 ; i+=1
        Equation (1)
ByteSubstitution(state,sbox)
BytePermutation(state)
AddRoundKey(state,RoundKey[10])
```

As mentioned above, we have five implementations in the experiments, and each of them adds a bit of improvement.

1. Serial standard implementation. This approach is used as a reference to check the correctness of the output. It follows the procedure and test output provided by the CrypTool Portal [5].
2. Serial fast implementation. This approach compresses standard operations to a series of T-table lookups. There are four T-tables, each is an int32 array of size 256.
3. Parallel implementation with constant memory. The parallelization granularity is multiple blocks handled by a single thread, depending on the data size. As a result, a thread needs to access four T-tables, one S-box, and 11 round keys. These data are constant and can be precomputed on the host. In this implementation, we put all these data in the constant memory.
4. Parallel implementation with shared memory. This implementation put all four T-tables, one S-box, and 11 round keys in the thread block's shared memory. Before encryption & decryption, the 512 threads in a block collaborate to load the data into shared memory.
5. Parallel implementation with warp shuffling. Among the T-table, S-box, and round keys, the 11 round keys occupy the smallest space (11*4=44 bytes) and can fit into the register file. Before encryption & decryption, all threads in a warp create local variables to store all the round keys. During computation, the threads in the warp access each other's local variables via a warp shuffling interface to get the needed round keys.

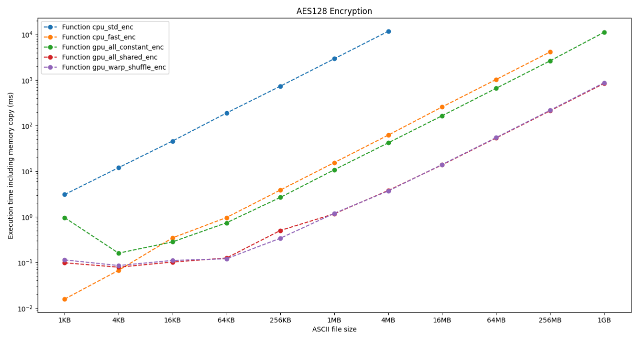
3. Results and Discussion

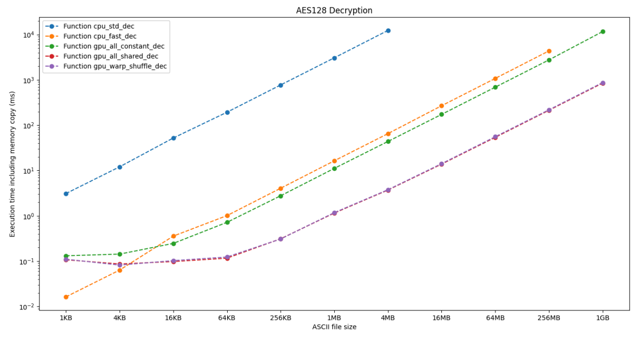
3.1. Tools and platforms

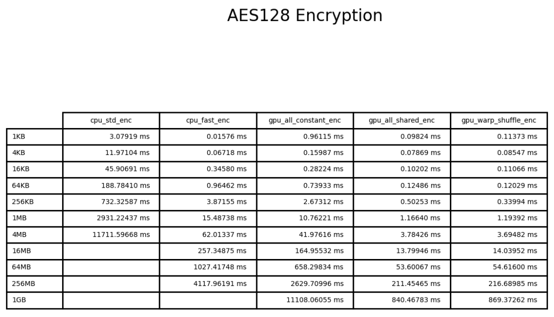
The tools and platforms in this implementation are listed below:

Language: CUDA C++
Build tools: CMake 3.2
Compiler: CUDA 12.6, G++ 12.3
Host platform: Ubuntu 22.04.5 LTS in x86/64 architecture
GPU platform: Nvidia L4 GPU with driver version 560.35.05

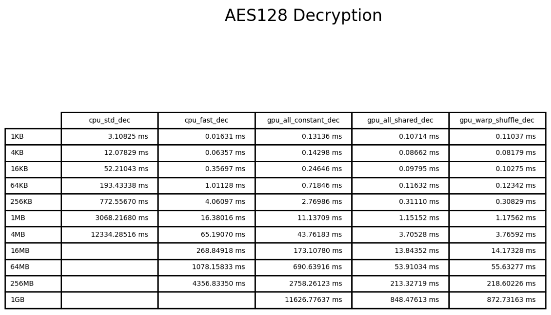
3.2. Results






AES128 Encryption

	cpu_std_enc	cpu_fast_enc	gpu_all_constant_enc	gpu_all_shared_enc	gpu_warp_shuffle_enc
1KB	3.07919 ms	0.01576 ms	0.96115 ms	0.09824 ms	0.11373 ms
4KB	11.97104 ms	0.06718 ms	0.15987 ms	0.07869 ms	0.08547 ms
16KB	45.90691 ms	0.34580 ms	0.28224 ms	0.10202 ms	0.11066 ms
64KB	188.78410 ms	0.96462 ms	0.73933 ms	0.12486 ms	0.12029 ms
256KB	732.32587 ms	3.87155 ms	2.67312 ms	0.50253 ms	0.33994 ms
1MB	2931.22437 ms	15.48738 ms	10.76221 ms	1.16640 ms	1.19392 ms
4MB	11711.59668 ms	62.01337 ms	41.97616 ms	3.78426 ms	3.69482 ms
16MB		257.34875 ms	164.95532 ms	13.79946 ms	14.03952 ms
64MB		1027.41748 ms	658.29834 ms	53.60067 ms	54.61600 ms
256MB		4117.96191 ms	2629.70996 ms	211.45465 ms	216.68985 ms
1GB			11108.06055 ms	840.46783 ms	869.37262 ms



AES128 Decryption

	cpu_std_dec	cpu_fast_dec	gpu_all_constant_dec	gpu_all_shared_dec	gpu_warp_shuffle_dec
1KB	3.10825 ms	0.01631 ms	0.13136 ms	0.10714 ms	0.11037 ms
4KB	12.07829 ms	0.06357 ms	0.14298 ms	0.08662 ms	0.08179 ms
16KB	52.21043 ms	0.35697 ms	0.24646 ms	0.09795 ms	0.10275 ms
64KB	193.43338 ms	1.01128 ms	0.71846 ms	0.11632 ms	0.12342 ms
256KB	772.55670 ms	4.06097 ms	2.76986 ms	0.31110 ms	0.30829 ms
1MB	3068.21680 ms	16.38016 ms	11.13709 ms	1.15152 ms	1.17562 ms
4MB	12334.28516 ms	65.19070 ms	43.76183 ms	3.70528 ms	3.76592 ms
16MB		268.84918 ms	173.10780 ms	13.84352 ms	14.17328 ms
64MB		1078.15833 ms	690.63916 ms	53.91034 ms	55.63277 ms
256MB		4356.83350 ms	2758.26123 ms	213.32719 ms	218.60226 ms
1GB			11626.77637 ms	848.47613 ms	872.73163 ms

By running the test script, we derive two graphs and two tables. The two graphs show the encryption & decryption time of different implementations under different data sizes. The time and data sizes are all in log scale. The time of serial implementation records only the computation time, because the input & output buffer are both in host memory. The time of parallel implementation includes both the computation time and memory transfer time between the host and GPU. The two tables record the specific times in the two graphs.

3.3. Discussions

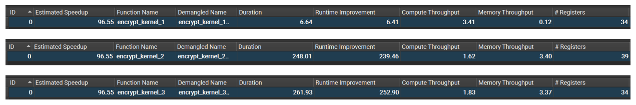
We have the following analysis of the graph:

- The two graphs are similar. This is because decryption concerns the same amount of operations as encryption.
- The fast AES serial implementation is much faster than the standard AES implementation. This is because, in the fast algorithm, the Galois field matrix multiplication is replaced with lookup tables, which need much fewer steps. What's more, the computation is around 4 words instead of 16 bytes, which is more efficient for the Arithmetic and Logic Unit (ALU).
- When the data size is small (<4KB), the fast serial implementation is the fastest. This is because when the data size is small, the latency of host-device memory transfer dominates the execution time. Only when the data size is large, the computation and memory bandwidth can dominate the execution time.

When the data size is large enough, the execution time of the implementations shows: serial standard AES >> serial fast AES > parallel AES with constant memory >> parallel AES with shared memory ≈ parallel AES with warp shuffling. This derives the following conclusions:

- By launching 1024 threads in parallel, the execution is faster than the serial implementation. This proves that AES128 is parallelizable.
- The line is almost straight when the data size is large. This means that the AES algorithm has O(n) time complexity, and these implementations have good scalability.
- The shared memory implementation is significantly faster than the constant memory implementation. This is because all memory access operations in computation now use shared memory. Intuitively, the shared memory is 5 cycles away from the processor, while the constant memory is O(100) cycles away.
- The warp shuffling approach is not helpful for the execution. This result is coherent with the result shown in the work of A. Abdelrahman et al. This is because most of the memory accesses in the warp shuffling implementation are still shared memory access, which dominates the execution time. What's more, to use memory shuffling, we need to synchronize all threads in a warp, which makes the execution time bounded by the slowest thread.

3.4. Profiling



ID	Estimated Speedup	Function Name	Demangled Name	Duration	Runtime Improvement	Compute Throughput	Memory Throughput	# Registers
0	96.55	encrypt_kernel_1	encrypt_kernel_1..	6.64	6.41	3.41	0.12	34

ID	Estimated Speedup	Function Name	Demangled Name	Duration	Runtime Improvement	Compute Throughput	Memory Throughput	# Registers
0	96.55	encrypt_kernel_2	encrypt_kernel_2..	248.01	239.46	1.62	3.40	39

ID	Estimated Speedup	Function Name	Demangled Name	Duration	Runtime Improvement	Compute Throughput	Memory Throughput	# Registers
0	96.55	encrypt_kernel_3	encrypt_kernel_3..	261.93	252.90	1.83	3.37	34

The profiling files can be found in the code repository.

The profiling results from top to down are using constant memory, shared memory, and warp shuffling. The most significant difference is the throughput of the second and third records is much larger than the first record. This shows the efficiency of constant memory is poor because it spends most of the time waiting for constant memory access. The throughput of the shared memory approach and warp shuffling approach are similar, which is consistent with the fact that their execution time is similar.

The correctness of the implementation is verified by comparing all the serial and parallel output with the standard AES128 implementation, over input data of 1GB.

4. Further Work

This project shows the effectiveness of GPU in completing AES encryption & decryption tasks, especially under large datasets. It is a practice that tries to reproduce the results of previous works, and it is implemented independently from scratch.

New GPU architectures improve and rebalance the computation and memory access speed, so it is crucial for the program to find the most suitable parameter based on the specific architecture. In the future, we can fine-tune the program under different GPU architectures to achieve best performance under different inputs.

What's more, although ECB mode is fundamental, CTR mode is more widely used in practice due to its security. We can extend the implementations to CTR mode and find its best hyper-parameters.

5. Conclusion

This project conducts experiments on five implementations of AES128, and successfully gets the anticipated result. We learned how to construct a CUDA C++ program and how to use the warp shuffling approach for inter-thread communication. Further work will emphasize fine-tuning on a specific GPU architecture and extending to CTR block-cipher mode,

6. Acknowledgements

I want to thank Professor Kostic who gave us such nice lectures. This course is one of the most useful that I attended during my study at this university.

7. References

[1] Qiuhong Chen, e4750-2024fall-project-cqcq-qc2335. https://github.com/eecse4750/e4750-2024fall-project-cqcq-qc2335

[2] A. A. Abdelrahman, M. M. Fouad, H. Dahshan and A. M. Mousa, "High performance CUDA AES implementation: A quantitative performance analysis approach," 2017 Computing Conference, London, 2017, pp. 1077-1085, doi: 10.1109/SAI.2017.8252225.

[3] A. D. Biagio, A. Barenghi, G. Agosta and G. Pelosi, "Design of a parallel AES for graphics hardware using the CUDA framework," 2009 IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, 2009, pp. 1-8, doi: 10.1109/IPDPS.2009.5161242.

[4] An image encryption method based on selective AES coding of wavelet transform and chaotic pixel shuffling - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/An-illustration-of-the-AES-encryption-and-decryption-schemes-3_fig4_333612056 [accessed 17 Dec 2024]

[5] "CrypTool Portal," CrypTool Portal. https://legacy.cryptool.org/en/cto/aes-step-by-step

8. Appendices

The GitHub repository includes the code and results. The profiling results of the three parallel implementations can also be found.