Parallel Advanced Encryption Standard (AES)

Implementation and Analysis

Wu Yuheng, Li Zongze

This document is served as ShanghaiTech University CS121 Parallel Computing course project report. In this project, we implemented ECB-AES and CTR-AES in parallel using OpenMP, MPI and CUDA. We also did a series or performance test and scalability test as well as analyze these graphs so as to draw our conclusion to parallel AES algorithm. Moreover, we use nvprof to profile our CUDA executable file and optimize our algorithm. Our result will rank the speedups from high to low as CUDA, OpenMP, MPI.

1 Introduction

The Advanced Encryption Standard (AES), is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001. Its strength lies in the use of a substitution-permutation network operating on fixed-size blocks of data. With key lengths of 128, 192, or 256 bits, AES provides a robust defense against both conventional and advanced cryptographic attacks. This versatility and resilience make AES a cornerstone in ensuring the confidentiality and integrity of digital communication, financial transactions, and data storage across diverse domains.

Figure 1 shows the algorithm for a 128-bit AES that can encrypt a piece of 128-bit plain-text to a piece of 128-bit cipher-text using a 128-bit key. The process involves a series of carefully orchestrated steps, including key expansion, initial round key addition, multiple rounds of substitution-permutation operations, and final round key addition. 128, 192 and 256 bits AES respectively requires 10, 12, 14 round of these steps.

1.1 Paralleling AES Algorithm

Paralleling the AES algorithm itself is hard. Since the encryption rounds strictly rely on the previous rounds, the only possible way currently is to implement pipelining. Considering that software-level pipelining requires huge workload and we do not have hardware support for hardware-level pipelining, we tried to add OpenMP to the AES SubBytes and ShiftRows. The result turns out to be a 4x slower in performance due to OpenMP overhead outweighing the benefit of parallelization since the iteration time is only 16.

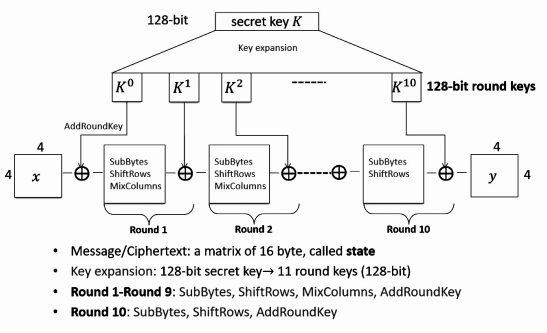

Figure 1: AES Algorithm

So instead of attempting to boost the AES algorithm, we decided to speedup the AES block ciphering process. That is, when encrypting a long dataset, for instance 1 GB, since AES length are fixed to 128 bit, 192 bit or 256 bit, the a naive and inefficient way is to execute the AES for a lot of times to encrypt the plain-text on steps. So the AES block ciphering algorithm is critical to overall performance.

1.2 Parallelized AES Block Ciphering

There are several existed way to parallel AES Block Ciphering. We will introduce ECB, CBC, CTR below.

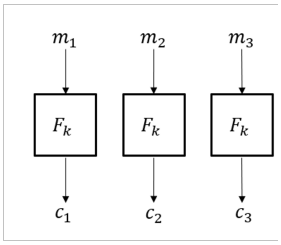

Figure 2: ECB Algorithm

1.2.1 ECB In ECB mode, each block of plaintext is independently encrypted using the same key, resulting in a corresponding block of ciphertext. ECB is easy to be parallized. The key characteristic of it is that identical blocks of plaintext will always encrypt to identical blocks of ciphertext, regardless of their position in the message. So ECB is not IND-EAV secure but is IND-CPA secure.

1.2.2 CBC In CBC mode, each block of plaintext is XORed with the previous ciphertext block before encryption, introducing a chaining mechanism that prevents identical plaintext blocks from producing identical ciphertext blocks. CBC is both IND-EAV secure and IND-CPA secure. However, the CBC algorithm is serial and can only be parallelized through pipelining.

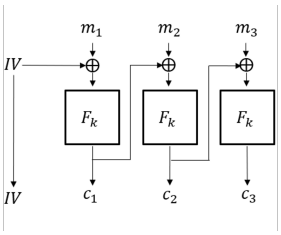

Figure 3: CBC Alogrithm

1.2.3 CTR In CTR mode, each block of plaintext is encrypted by combining it with the output of a counter that increments for each subsequent block. This counter, often combined with a nonce (number used once), generates a unique keystream for each block, which is then XORed with the plaintext to produce the corresponding ciphertext. CTR mode provides good parallelism as well as good security. We also found CTR a ideal fit for CUDA, which will be decribed later in this report.

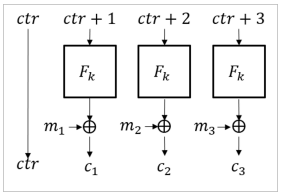

Figure 4: CTR Algorithm

2 Related Work

In 2011, Kozak, Maskiewicz, and Stpiczynski focused on the computational power of General-Purpose Graphics Processing Units (GPGPUs) to boost AES [4]. Following this, in 2013, Keivanloo and Amiri investigated the parallelization of AES, employing OpenMP as tool. [3]. In year 2015 Varsha C. and Rajalakshmi's strides in implementing parallel AES on Field-Programmable Gate Arrays (FPGAs)[1]. Building upon this trend in 2016, Wang, Jia, and Feng explored the parallelization of AES through the utilization of CUDA[5]. In 2018 when Ezzati-Jivan, Ezzati-Jivan, and Akbari worked on the parallelization of AES on multi-core processors and discussed the challenges and benefits of parallelizing AES on multi-core architectures [2]. These pivotal works collectively showcase the evolution of parallel AES algorithms across different computing platforms and different algorithms.

3 Implementation

We use OpenMP, MPI and CUDA to implement our AES parallelization experiment and use chrono to timing test results. *Figure 5* is the MPI uml graph. For the first stage experimentation, we use all ECB-AES for fair comparision. -O3 compiler option offers us a approximate 3x speedup.

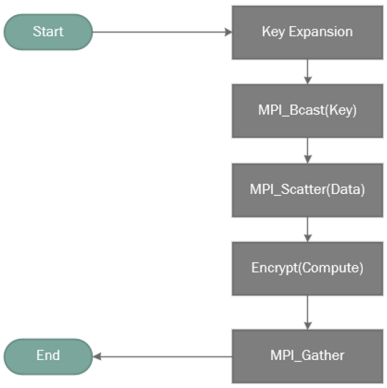

Figure 5: MPI Procedure

3.1 Multiplication Optimization

In MixColumn operation in AES, we need to conduct multiplications over $\mathbb{F}_{2^8}$ field. The procedures of $a * b$ are as follows.

$$\mathbb{F}_{2^8} = \mathbb{Z}_2[X]/(X^8 + X^4 + X^3 + X + 1)$$

$$\begin{aligned} a &= a_0 + a_1X + ... + a_7X^7 \\ b &= b_0 + b_1X + ... + b_7X^7 \\ c &= (a * b) mod (X^8 + X^4 + X^3 + X + 1) \\ &= (C_0 mod 2) + (C_1 mod 2)X + ... + (C_7 mod 2)X^7 \\ &= c_0 + c_1X + ... + c_7X^7 \end{aligned}$$

Since the field is only 2^8 in size, we can use a $256 * 256$ array look-up to replace the multiplication calculation. It's a trade between $1\ byte(uint8\_t) * 256 * 256 = 64KB$ space and O(1) look-up time. With this technique, the OpenMP and MPI based AES achieves a huge speedup.

It is also a interesting result that this multiplication optimization does not work well on GPU. We infer that's because GPU has thousands of cores, with which the work done on each core

is so little that the compute time of multiplication are smaller than the time to access device memory.

4 Results and Analysis

4.1 Baseline

We use an implemented library *openssl/aes* as our baseline. We calculate the time period of program starting from raw data fetched to DRAM and ending at the cipher-text arrives at DRAM.

Openssl also provides a shell interface in Ubuntu, but that CLI counts the time of fetching data from disk to DRAM, so we use *openssl/aes* API to implement a ECB-AES as baseline.

Also, we found through *htop* command monitor that *openssl/aes* is not single-threaded. Instead, it will use several threads (not all) to do the encryption.

4.2 Test Platform

Table 1: Hardware Specifications

Processor	2013 Intel Xeon E5-2697 v2 × 2 (24C48T)
RAM	Micron DDR3 1600 MHz 16GB ECC × 4 (64GB)
GPU1	NVIDIA GeForce GTX 970 (profile)
GPU2	NVIDIA GeForce RTX 3080 Ti (compute)

4.3 Sequential Result

Figure 6 shows our sequential program performance. We can infer from the graph that as file size grows, our sequential program is always about 7x slower than the *openssl* lib. The result is quite reasonable because *openssl* is a highly optimized library and it uses more than one thread to do the operation.

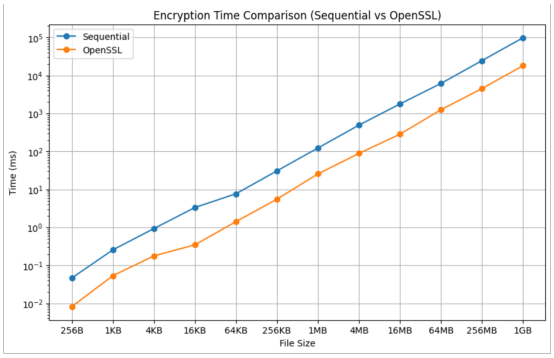

Figure 6: Sequential Performance

4.4 OpenMP

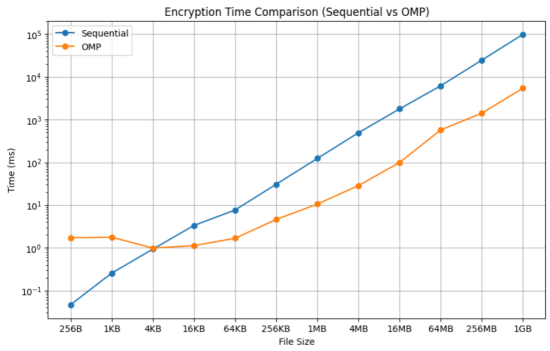

Figure 7: OpenMP Versus Serial

Figure 7 is the result of OpenMP versus the result of serial algorithm. We can dipict from the picture that at larger file size, for example, 1 GB, OpenMP are more than 10x faster than se-

rial algorithm. However, under small file size, especially smaller than 4KB, OpenMP have not that good performance due to multi-thread overhead.

4.5 Overall Performance

MPI and CUDA result are shown in the overall graph (*Figure 8*).

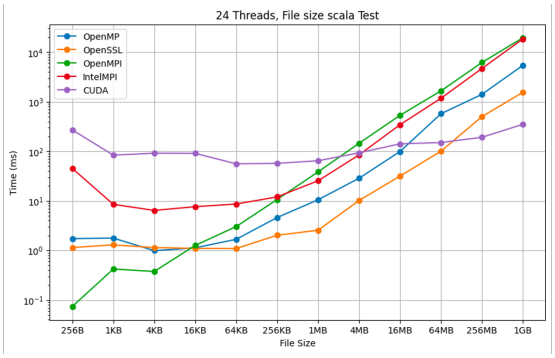

Figure 8: Overall Result

We can see that on CPU runs, the *openssl* run on ECB mode and OpenMP parallel is as fast as expected, due to its optimization. The second fastest one is OpenMP on our serial program, which is around 7x slower than *openssl*. The slowest one is MPI because MPI is designed to have good performance on distributed memory systems and it has the largest overhead.

CUDA is special to perform as an almost horizontal line. CUDA has relatively bad performance on small file sizes due to cudaMemcpy() is time-consuming compared to DRAM copy. However, on larger file sizes CUDA is extremely fast and its ascending trend is slow. On 1GB file size, CUDA performs nearly 10x of speedup than *openssl* and nearly 100x speedup than MPI. And after 1GB file size, we can infer that CUDA will have huge gaps with CPU runs.

It is also worth analyzing the difference between IntelMPI and OpenMPI. IntelMPI, is as our expected, always has a slower performance than OpenMP. However, OpenMPI have surprisingly fast performance on small file sizes. We had excludes our own experiment factors through turning off hyperthread, fix thread to small numbers and change our test machine from Intel to AMD processors, and the only difference between OpenMPI run and IntelMPI run is that we change the MPI in shell. So we think this is because some subtle implementation of OpenMPI that leads to this result.

5 CUDA Optimization

We noticed a fact during our experiment that the compute time of CUDA (At the level of 10^-5 second) is way more less than the wall time (At the level of 10^0 second). So we use *nvprof* in *NVIDIA CUDA Toolkit 12.3* to profile the CUDA executable file.

Figure 9: Profiling Result before Optimization

Figure 9 is the profiling result. We can see that the computing time of CUDA AES is nearly negligible and on contrary, the cudaMemcpy() time accounts for a large part.

So the CTR-AES algorithm naturally comes to our mind that since we only need to encrypt the counter in CTR, we do not need to execute the first cudaMemcpy(). Instead, we only need

to calculate the counter by threadIdx in kernel function.

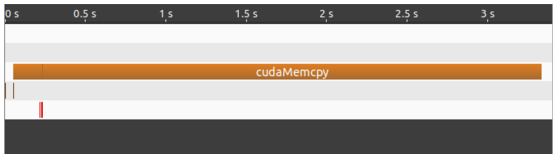

Figure 10: Profiling Result after Optimization

Figure 10 is the profiling result after optimization. We can see that the first cudaMemcpy() has been eliminated. And the speedup we've measured is 18% on average. Besides, the security of this encryption has been extended.

Also, we need to explain why CTR is only an ideal fit for CUDA. On OpenMP and MPI runs, memcpy is never a bottleneck. The most time-comsuming operation is the AES encryption calculation. Adding CTR to these runs will provide little performance improvement.

6 Conclusion

From what has been discussed above, we can safely jump to the conclusion that CPU paralleling is suitable for file under small size and on a large GB-level dataset, using GPU to accelerate calculation is completely worthy. Besides, under very small size, using only the serial algorithm would be the best choice.

For further work, it is a good idea to seek optimization under distributed systems. Also, it would be interesting to figure out the cause of the special result for OpenMPI under small file sizes. Improving paralleling algorithm would also be a ever-lasting problem to be solved.

7 Existing Code Work

We refer to a popular Github repository created by dhuertas/AES which is a AES serial implementation in C. All of the other work in the code and report, including but not limited to paralleling implementation, profiling, shell scripts and datasets are all done by ourselves.

References

[1] Varsha C. and R. Rajalakshmi. Parallel implementation of the advanced encryption standard algorithm on fpga. *International Journal of Advanced Research in Computer and Communication Engineering*, 4(5):329–332, 2015.

[2] Naser Ezzati-Jivan, Roohollah Ezzati-Jivan, and Behzad Akbari. Parallel aes on multicore processors. In *2018 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 2018.

[3] Iman Keivanloo and Mohammad Javad Amiri. Parallel implementations of aes based on the parallel programming model openmp. *International Journal of Computer Applications*, 65(1):26–32, 2013.

[4] Karol Kozak, Lukasz Maskiewicz, and Piotr Stpiczynski. Parallel aes algorithms for gpgpus. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2011.

[5] Xiaoxia Wang, Weijia Jia, and Xianjin Feng. Parallel aes encryption algorithm based on cuda. In *2016 8th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*. IEEE, 2016.