Analysis of Parallel Implementation of Pilsung Block Cipher On Graphics Processing Unit

Siwoo Eum[1][0000−0002−9583−5427],
Hyunjun Kim[1][0000−0001−6757−6109],
Minho Song[2][0009−0007−6277−0069], and
Hwajeong Seo[2][0000−0003−0069−9061]

[1]Department of Information Computer Engineering,
Hansung University, Seoul (02876), South Korea,
[2]Department of Convergence Security,
Hansung University, Seoul (02876), South Korea,
{shuraatum, khj930704, smino0906, hwajeong84}@gmail.com

Abstract. This paper focuses on the GPU implementation of the Pilsung block cipher used in the Red Star 3.0 operating system developed in North Korea. The Pilsung block cipher is designed based on AES. One notable feature of the Pilsung block cipher is that the table calculations required for encryption take longer than the encryption process itself. This paper emphasizes the parallel implementation of the Pilsung block cipher by leveraging the parallel processing capabilities of GPUs and evaluates the performance of the Pilsung block cipher. Techniques for optimization are proposed, including the use of Pinned memory to reduce data transfer time and work distribution between the CPU and GPU. Pinned memory helps optimize data transfer, and work distribution between the CPU and GPU needs to be considered for efficient parallel processing. Performance measurements were performed using the Nvidia GTX 3060 laptop for evaluation, comparing the results of applying Pinned memory usage and work distribution optimization. As a result, optimizing memory transfer costs was found to have a greater impact on performance improvement. When both techniques were applied together, approximately a 1.44 times performance improvement was observed.

Keywords: Pilsung · GPU · Pinned Memory · Parallel implementation · Block cipher

1 Introduction

Pilsung block cipher [1] is a special type of block cipher used in the Red Star 3.0 operating system, developed in North Korea. It is built upon the Advanced Encryption Standard(AES) [2] and provides effective security features. However, a notable characteristic of Pilsung block cipher is that the computation of tables required for encryption takes longer than the encryption process itself.

This paper focuses on the implementation of Pilsung block cipher on Graphics Processing Units (GPUs). GPUs are processors specialized in handling massive parallel computations, offering high-performance parallel processing capabilities. Consequently, leveraging GPUs allows for efficient encryption of large datasets within shorter time frames. This paper focuses on leveraging GPU parallelism to carry out the parallel implementation of the Pilsung block cipher.

The paper is structured as follows: In Chapter 2, an explanation of the Pilsung block cipher and GPUs. Chapter 3 introduces the techniques used for optimizing the implementation on GPUs. Chapter 4 evaluates the performance of the implemented techniques. Finally, Chapter 5 presents the conclusions of the paper.

2 Background

2.1 Pilsung Block cihper [1]

Pilsung block cipher is an algorithm based on the SPN(Substitution-Permutation Network) structure, which is derived from the AES block cipher. SPN is one of the structures used in symmetric key cryptography, where the input block undergoes substitution and permutation stages to carry out the encryption process.

The block length of Pilsung is 128 bits, and the key length is 256 bits. As it is based on the AES block cipher, it shares the same round functions (SubByte, ShiftRows, MixColumns, AddRoundKey) as AES. However, there are differences between Pilsung and AES in the SubByte and ShiftRows functions, as well as in the key expansion algorithm.

The Pilsung cipher introduces some differences in the SubByte, ShiftRows, and key schedule compared to the AES round functions.

Key Schedule The key schedule in Pilsung also deviates from AES. While AES employs a specific key schedule algorithm, Pilsung utilizes the SHA-1 hash function for key expansion. The input key is hashed using SHA-1, resulting in a 160-bit hash value. This hash value is then used in the Pilsung key schedule algorithm to generate the round keys for encryption and decryption.

SubByte A substitution operation is performed using Sbox tables. While AES utilizes a single table for the operation, Pilsung employs a different Sbox table for each round, resulting in a total of 160 tables (16 bytes * 10 rounds). These tables are generated by applying an arbitrary bit permutation, using the round keys through the Rao-Sandelius shuffle, to the original Sbox table. The substituted values obtained from these generated tables undergo a final step involving bitwise XOR with 3.

ShiftRows A distinct permutation operation is carried out for each round. Similar to the Sbox generation, a random byte permutation is created by utilizing the round keys and applied to the operation. This ensures different permutations for each round in Pilsung.

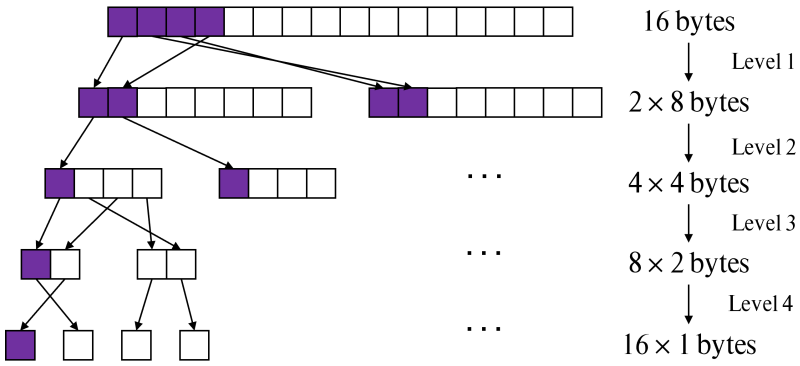

Fig. 1: Rao–Sandelius shuffle in Pilsung block cipher.

Rao-Sandelius Shuffle [3–5] To generate permutations, Pilsung utilizes the Rao −Sandelius shuffle. This shuffle first randomly divides an array into two equal halves and recursively shuffles each half. For shuffling 16 bytes, four stages of shuffling are required. Algorithm 1 provides a high-level description of this permutation implemented in Pilsung. Figure 1 illustrates a visualization of the algorithm. The randomness for the four levels of shuffling, used to generate the permutation at round i, is obtained from the corresponding round key, RKi. The randomness for the first and second level shuffles is taken from the front portion of the round key, while the randomness for the third and fourth level shuffles is taken from the rear portion of the round key.

These modifications in SubByte, ShiftRows, and the key schedule contribute to the unique characteristics and security properties of the Pilsung cipher, differentiating it from the standard AES round functions.

2.2 Graphics Processing Units

The use of GPUs has become an integral and widespread component in modern computing systems. These GPUs are highly parallel processors with significant arithmetic and memory bandwidth capabilities that far surpass those of CPUs [6–8]. For our study, we utilized an Nvidia RTX 3060 Laptop GPU, which boasts an impressive 3,840 cores and operates at a clock rate of 1,702MHz. It is important to note that clock rates may vary depending on the specific GPU manufacturer. The GPU we used is designed with the Ampere architecture, which has a Compute Capability (CC) of 8.3. CC refers to the device's ability to perform computations.

To leverage the parallel processing power of the GPU, we employed the Compute Unified Device Architecture (CUDA), a GPGPU (General-Purpose Computing on Graphics Processing Units) technology. CUDA allows programmers to write parallel processing code using the C language. Developed and main-

tained by Nvidia, CUDA requires an Nvidia GPU and the corresponding stream processing driver. The CUDA GPU architecture comprises functional kernels, threads, blocks, grids, and warps (bundles of 32 threads) that run on the GPU. Multiple warps execute concurrently on a Streaming Multi-processor (SM), enabling efficient parallel computation [8, 9].

3 Implementation technique

GPU computing leverages multiple cores to perform computations on large amounts of data simultaneously, enabling efficient processing of large datasets. Therefore, in GPU-based implementations, the transfer of data from the host (CPU) to the device (GPU) is an important consideration that needs to be optimized. Additionally, it is not always the best approach to perform all encryption operations solely on the GPU, as the optimal performance of parallel implementations requires a balanced division of tasks between the CPU and GPU.

In this section, we will discuss the techniques applied to achieve parallel implementation of the Pilsung block cipher on the GPU. First, we will explain methods to reduce data transfer time. Then, we will explore the distribution of tasks between the CPU and GPU.

3.1 Using Pinned Memory

The performance of GPU implementation is influenced by various factors, and one of them is the data transfer between the Host (CPU) and the Device (GPU). In GPU implementation, data is copied from the Host to the Device before executing GPU kernel functions, and the resulting computation is copied back from the Device to the Host. Although it may seem like a simple copying process, it can significantly impact performance. For example, let's assume a block size of 16 bytes (128 bits). If we consider a configuration with 1024×128 blocks and 256 threads, the total amount of data that needs to be copied would be 512MB (16×1024×128×256).

By default, when allocating memory using the Malloc() function in the Host, it is considered Pageable memory. Pageable memory cannot be directly copied to GPU memory. To copy Pageable memory to GPU memory, the GPU driver needs to allocate Pinned memory. The data is first copied from Pageable memory to Pinned memory and then from Pinned memory to GPU memory.

In Figure 2To skip the step of copying data from Pageable memory to Pinned memory, it is possible to allocate Pinned memory directly in the Host when allocating memory. This can be achieved using CUDA's supported functions such as cudaMallocHost() or cudaHostAlloc(). By utilizing these functions, Pinned memory can be directly allocated in the Host, bypassing the copying process from Pageable memory to Pinned memory [10].

By employing Pinned memory allocation, the data transfer between the Host and the Device can be optimized, leading to improved performance in GPU implementations.

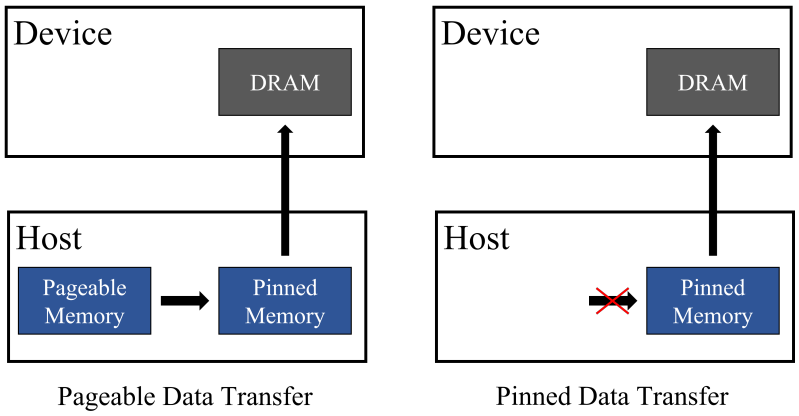

Fig. 2: Processes of Pageable and Pinned Data Transfer.

3.2 Work Distribution

In Pilsung, the SubByte function uses different Sboxes for each round, so an operation to expand the tables is performed before the round function operates. Similarly, in the ShiftRows function, random permutations (utilizing Pbox) specific to each round are applied, rather than fixed permutations.

The expansion operations of the Sbox and Pbox tables are based on the round keys generated through key expansion, and the expanded tables are utilized for encryption. Since different tables are used for each round and each byte in the Sbox, the expansion requires 160 repetitions, resulting in significant computational cost.

In Figure 3 represents an overview of the Pilsung block cipher process. A straightforward approach in implementing encryption using GPUs is to perform all encryption steps in all threads. This includes key scheduling and table expansion processes. Round keys are generated through key scheduling, and Sboxes and Pboxes used for encryption are generated through table expansion. If all threads follow the same process, they will produce the same values when the master key is the same. Performing the same computations in all threads to obtain the same value ultimately becomes inefficient, and it leads to the utilization of more memory to store the computation results.

To optimize the implementation, this paper suggests conducting key expansion on the host and table expansion on the device (GPU). As the table expansion involves the same operation repeated for the number of rounds, it can be parallelized. By dividing the GPU threads for parallel computations, instead of expanding the tables through 160 repetitions when implemented on the host, it can be achieved with only 16 repetitions using 10 threads.

In addition to the performance improvement achieved through parallel computing, there is an additional performance benefit of reducing the size of data that needs to be copied from the Host to the Device. When the table expansion

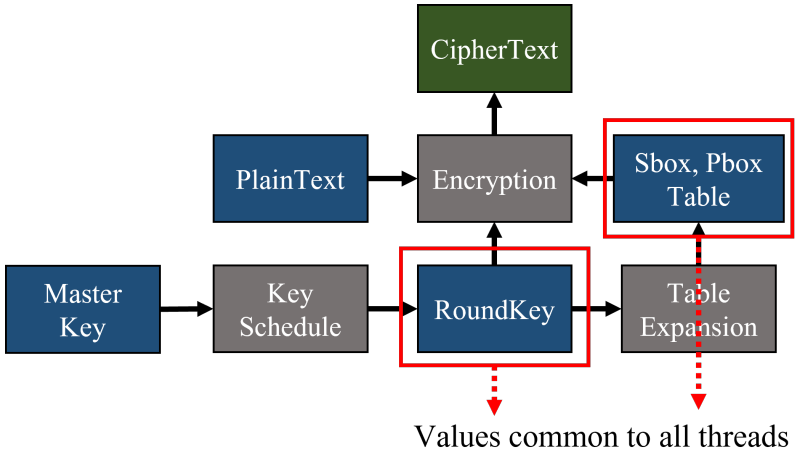

Fig. 3: Overview of the Pilsung block cipher

is performed on the Host, the cost of copying increases due to the expanded table. However, if the expansion is carried out on the Device, the table is defined and expanded internally on the Device, resulting in a reduced data size that needs to be copied from the Host to the Device. After the work distribution, the approximate process is as shown in Figure 4.

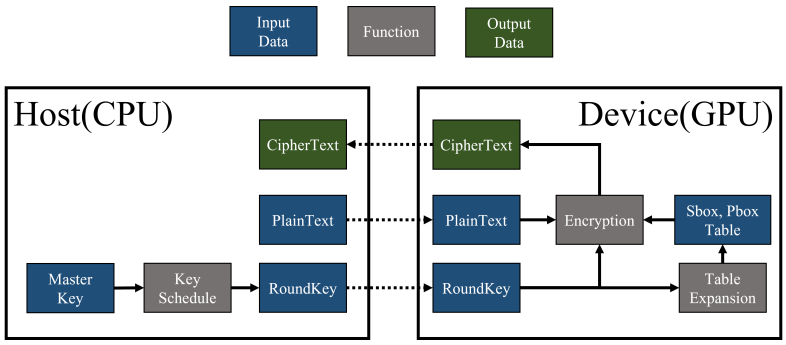

Fig. 4: After the work distribution, the approximate process of the Pilsung block cipher

4 Evaluation

For performance evaluation, the implementation and benchmarking were conducted on a Nvidia GTX 3060 Laptop. The implementation was done using

Visual Studio with Cuda 11.8 runtime version. The project was built in Release mode to measure the performance during execution. The number of blocks was fixed at 65536 (1024×64), and the measurement was performed by varying the number of threads. The performance measurement scope includes the time (in milliseconds) from the key expansion function, which includes memory copying (excluding encryption value copying), to the encryption function. The size of the plaintext data is determined by the Grid size multiplied by the Thread size and the block length (16 bytes). Hence, when the number of threads is 32, the size of the plaintext data is 32MB (1024×64×32×16).

To assess the computational throughput during kernel execution, we utilized the Nsight Compute tool provided by NVIDIA [11]. Nsight Compute is a GPU kernel profiling tool that allows for the analysis and optimization of CUDA-based applications. Developers can analyze GPU kernel execution time, memory access patterns, warp behavior, and more. The measurement results are presented in the following Table 2.

The performance measurement is conducted by dividing it into four implementations based on Using the Pinned memory and optimization of table expansion. The first implementation does not apply any optimization, the second implementation applies work distribution, the third implementation uses Pinned memory, and the last implementation applies all methods. The measurement results are presented in the following Table 1.

Table 1: Performance result(BlockSize: 1024*64, Pinned: Using Pinned mermory, Opti: Optimized Table Expansion, unit: ms)

Type	**Threads**					
	32	64	128	256	512	1024
None	15.9106	25.4039	42.3677	75.9655	145.1255	288.7281
Opti	13.6967	20.9236	35.2862	68.9094	138.1196	280.5977
Pinned	13.4598	20.6596	33.0835	59.8967	113.6430	224.7214
Opti&Pinned	11.3909	15.6775	26.9073	52.7457	104.4272	219.8976

Table 2: Performance results in terms of computational throughput (unit: %)

Type	**Threads**					
	32	64	128	256	512	1024
None	87.82	92.28	96.42	94.96	96.35	92.40
Opti&Pinned	26.09	48.74	84.75	94.44	91.54	76.16

In Table 1, The results of None and Opti demonstrate performance improvement through work distribution. Looking at 256 Thread as a reference in Table 2, we can observe an approximate 1.1x performance improvement with the application of task distribution. This confirms that performing table expansion operations on the GPU rather than the CPU can yield performance enhancements.

Next, we can observe performance improvement through the use of Pinned memory by examining the results of None and Pinned. Similarly, considering 256 Thread as a reference, we can see an approximate 1.26x performance improvement with the utilization of Pinned memory. This confirms that storing data directly in Pinned memory instead of Pageable memory can yield performance enhancements when using the GPU.

Ultimately, when comparing the two results, optimizing the memory transfer cost has shown greater performance improvement compared to task distribution. This indicates that in parallel implementations using the GPU, memory transfers have a significant impact on performance. Consequently, when both techniques are applied, we observed an approximate 1.44x performance improvement.

5 Conclusion

In this paper, we conducted an optimized implementation of the Pilsung block cipher on the GPU by utilizing the usage of Pinned memory and work distribution. We observed significant performance improvement simply by using Pinned memory, which allowed us to understand the impact of data copying on performance in GPU implementations. There are additional research and techniques available to reduce data copying costs, beyond the usage of Pinned memory. We believe that comparing the performance by applying various techniques and methodologies would yield valuable research results. Furthermore, since these techniques are not limited to the Pilsung block cipher, they can be applied to various block cipher GPU implementations.

References

1. "A brief look at north korean cryptography." `https://www.kryptoslogic.com/blog/2018/07/a-brief-look-at-north-korean-cryptography/`. July 2018.
2. J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.
3. C. R. Rao, "Generation of random permutations of given number of elements using random sampling numbers," *Sankhyā: The Indian Journal of Statistics, Series A*, pp. 305–307, 1961.
4. M. Sandelius, "A simple randomization procedure," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 24, no. 2, pp. 472–481, 1962.
5. C. Chuengsatiansup, E. Ronen, G. G. Rose, and Y. Yarom, "Row, row, row your boat: How to not find weak keys in pilsung," *The Computer Journal*, p. bxac092, 2022.
6. J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

7. H. Choi and S. C. Seo, "Fast implementation of sha-3 in gpu environment," *IEEE Access*, vol. 9, pp. 144574–144586, 2021.
8. K. Iwai, N. Nishikawa, and T. Kurokawa, "Acceleration of aes encryption on cuda gpu," *International Journal of Networking and Computing*, vol. 2, no. 1, pp. 131–145, 2012.
9. "CUDA C Programming Guide V6.0." `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Accessed: 2022-05-11.
10. D. Negrut, R. Serban, A. Li, and A. Seidl, "Unified memory in cuda 6.0. a brief overview of related data access and transfer issues," *SBEL, Madison, WI, USA, Tech. Rep. TR-2014-09*, 2014.
11. "Nsight Compute - nvida documentation center." `https://docs.nvidia.com/nsight-compute/NsightCompute/index.html`. Accessed: 2022-08-24.