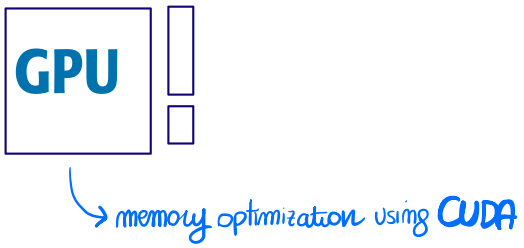
IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 5, 2021, accepted October 11, 2021, date of publication October 25, 2021, date of current version October 29, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3122466

Fast Implementation of SHA-3 in GPU Environment



HOJIN CHOI AND SEOG CHUNG SEO, (Member, IEEE)
Department of Financial Information Security, Kookmin University, Seoul 02707, South Korea

Corresponding author: Seog Chung Seo (scseo@kookmin.ac.kr)

This work was supported by the Institute of Information and communications Technology Planning and Evaluation (IITP) Grant by the Korean Government through Ministry of Science and ICT (MSIT) (Development of Fast Design and Implementation of Cryptographic Algorithms Based on GPU/ASIC) under Grant 2021-0-00540.

ABSTRACT Recently, Graphic Processing Units (GPUs) have been widely used for general purpose applications such as machine learning applications, acceleration of cryptographic applications (especially, blockchains), etc. The development of CUDA makes this General-Purpose computing on GPU possible. In particular, currently GPU technology has been widely used for server-side applications so as to provide fast and efficient service to a number of clients. In other words, servers need to process a large amount of user data and execute authentication process. Verifying the integrity of transmitted data is essential for ensuring that the data is not modified during transmission. Hash functions are the cryptographic algorithm which can verify the integrity of data and there are SHA-1, SHA-2, and SHA-3 standard hash functions. In 2015, Keccak algorithm was selected for SHA-3 competition by NIST. However, until now, software implementations of SHA-3 have not provided enough performance for various applications. In addition, SHA-3 and SHAKE using SHA-3 are being used in many Post-Quantum Cryptosystems (PQC) submitted to NIST PQC competition. Therefore, SHA-3 optimization research is required in the software environment. We propose an optimized SHA-3 software implementation on GPU environment. For performance efficiency, we propose several techniques including optimization of SHA-3 internal process, inline PTX optimization, optimized memory usage, and the application of asynchronous CUDA stream. As a result of applying the proposed optimization method, our SHA-3(512) (resp. SHA-3(256)) implementation without CUDA stream provides a maximum throughput of 88.51 Gb/s (resp. 171.62 Gb/s) on RTX2080Ti GPU. Furthermore, without the application of CUDA stream, our SHA-3(512) software on GTX1070 provides about 49.73% improved throughput compared with the previous best work on GTX1080, which shows the superiority of our proposed optimization methods. Our optimized SHA-3 software on GPU can be efficiently used for block-chain applications and several PQCs (especially, key generation process in Lattice-based cryptosystems).

INDEX TERMS Graphic Processing Unit (GPU), secure hash function, Secure Hash Algorithm (SHA)-3, software optimization, NVIDIA CUDA, parallel processing, cryptography, optimization.

I. INTRODUCTION

Graphics Processing Unit (GPU) architecture is designed for image processing and graphics processing, and is an effective device for parallel processing of data many threads. Recently, GPU architectures are used in various fields such as deep learning, machine learning, and cryptographic algorithms [1]–[7]. In addition, as GPU hardware specifications improve, General-Purpose computing on Graphics Processing Units(GPGPU) general-purpose computing technology has emerged that can handle applications on GPU architectures.

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Grosu.

Utilizing GPGPU technology, GPU architecture has begun to be used in many applications such as including embedded systems, artificial intelligence, driverless vehicles, and cloud computing etc. GPU architecture is effective for parallel processing of data, especially in server environments that allow access by many clients. For example, the server stores the client's data. The client's data is stored as ciphertext. In this process, the server encrypts the client's data or when the data is large, the GPU architecture enables effective encryption through parallel processing. In addition, when a client accesses the server, the server is required to authenticate the client and verify data integrity. In this process, the server uses a cryptographic hash function for authentication and data integrity verification.

Standard cryptographic hash functions are Secure Hash Algorithm (SHA)-1, SHA-2, and SHA-3. In 2015, National Institute of Standards and Technology (NIST) selected Keccak Algorithm as SHA-3 through a contest. Since, many researchers have proposed to SHA-1 and SHA-2 attack methods [8], [9], and in 2017, SHA-1 a real collision pair was discovered [10]. SHA-2 has a structure similar to SHA-1, and some researchers have proposed a SHA-2 attack method similar to the SHA-1 attack method [11]. In addition, SHA-3 shows 2 times slower performance than SHA-1 and SHA-2 in a software environment [12]. Therefore, optimization study of SHA-3 in software environment is required. In addition, SHA-3 is used in the National Institute of Standards and Technology (NIST) Post Quantum Cryptography(PQC) contest submission algorithm.

In 1994, Shor published an effective factorization algorithm in the quantum computing environment [13]. As a computing environment at the time, Shor's proposal was a theoretical algorithm. However, recently, many quantum computing development methods have been proposed. In fact, in 2019, Google proposed a quantum computing development method of 53 qubits [14]. In addition, in 2020, IBM developed the Quantum Volume 64, 65-qubit Quantum Hummingbird processor. In addition, IBM is developing a 127-qubit IBM Quantum Eagle Processor [15], [16]. With advances in quantum computers, Shor's effective factorization algorithm becomes feasible. Effective factorization calculation methods proposed Shor are critical to current public key cryptographic algorithms. The existing public key cryptographic algorithm system is designed based on the RSA algorithm. The RSA algorithm is a public key cryptographic algorithm based on the mathematical challenges of discrete logarithm and factorization, and the development of quantum computing is critical to the RSA cryptographic algorithm. Therefore, an alternative algorithm to the existing public key encryption algorithm is required. To replace RSA and RSA-based digital signature schemes, NIST is running a PQC competitive contest. Currently, the cryptographic algorithm submitted to NIST-PQC round 3 use SHA-3 and SHAKE to construct key and nonce establishment [17]–[20].

From the time Keccak algorithm was submitted to the NIST standard, research on SHA-3 optimization methods in GPU environment has continued until now [21]–[25]. Kim *et al.* proposed a SHA-3 optimization method in an 8-bit AVR environment [25]. We ported the optimization method proposed by Kim *et al.* to GPU environment. Dat *et al.* proposed a SHA-3 optimization technique in GPU environment [22]. They proposed an optimization method mainly related to memory access. We customized the optimization method of Dat *et al.* to our implementation. In addition, Dat *et al.* extended their optimal methods using three CUDA streams in [24].

In this paper, we propose an optimization method for SHA-3 in GPU environment. Our optimization methods are efficient processing that internal processing of SHA-3 and memory access on the GPU. We extended and customized the idea from Kim *et al.*'s SHA-3 implementation on the 8-bit AVR environment to optimize the performance of SHA-3 software on GPU environment [25]. In addition, we minimized the constant value table of SHA-3 internal functions. Our implementation reduces the number of memory accesses through constant table minimization.

Finally, our paper presents an effective implementation method: optimized internal operations, application of Parallel Thread eXecution (PTX), and CUDA streams. PTX is an inline assembly language available in CUDA C, and CUDA stream is a technology that can effectively handle memory copying and kernel functions. To the best of our knowledge, our optimization implementation is the fastest implementation of the GPU architecture.

A. CONTRIBUTIONS

The contribution of this paper is as follows:

- **OPTIMIZATION OF SHA-3 INTERNAL PROCESS** To optimize the performance of SHA-3 software in a GPU environment, we extended and customized the SHA-3 implementation idea by Kim *et al.* methods [25]. Actually, we carefully configured the idea from Kim *et al.* to make it effective in the SHA-3 implementation of GPU environment. In addition, we present a SHA-3 implementation using PTX inline assembly, which can make full use of the coarse-grained registers and arithmetic instructions for the optimum performance. In our experiments, algorithms implemented with our hand-crafted PTX codes outperforms than codes generated by the CUDA compiler. Furthermore, out implementation provides the optimal performance compared with the previous best result. In addition, our optimization technique can also be applied to SHAKE and PQC algorithms using SHAKE.
- **STRATEGY FOR OPTIMIZED MEMORY ACCESS OPERATION** SHA-3 operation process requires a pre-operation to store and load the constant values (π offset, ρ offset and ι offset) into memory. GPU architecture has high latency in memory loads and stores. Therefore, for effective computational processing in the GPU architecture, the number of memory accesses needs to be minimized. In this paper, we proposed a method to minimize the size of the constant table in the SHA-3 internal process. Our optimization method removed the π constant table, ι constant table and the ρ constant table, and processed the internal process through direct indexing. In this process, our implementation, reduced the number of 75 * *rounds* memory accesses compare to general SHA-3 implementation. In addition, to minimizing the number of memory accesses for loading plaintext data, our implementation applies coalesced memory accesses. Our SHA-3(512) implementation, performance with internal operation optimization methods and memory access optimization method provides maximum throughput of 88.51GB/s

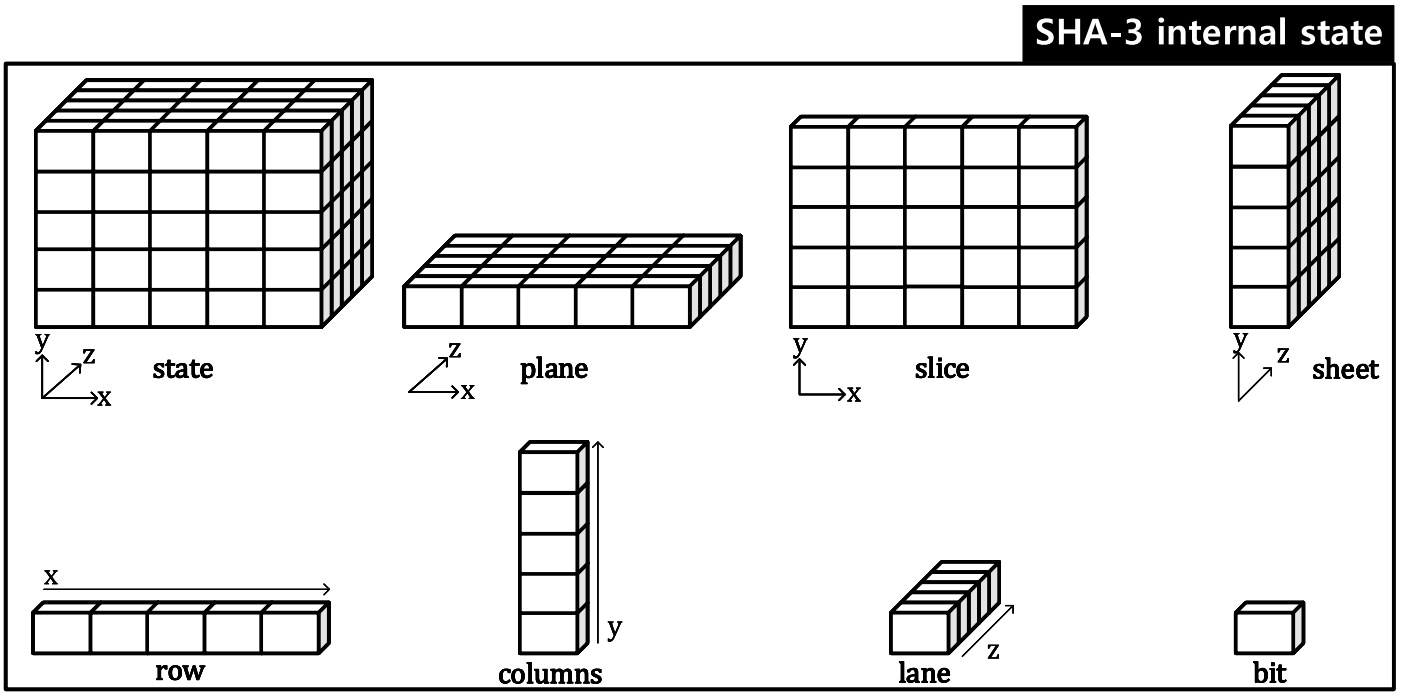


FIGURE 1. SHA-3 internal state [26].

in RTX 2080Ti environment and 30.20Gb/s in GTX 1070 environment without the application of CUDA stream. In addition, our SHA-3(256) implementation performance provides maximum throughput 59.87Gb/s in GTX 1070 environment and 171.62Gb/s in RTX 2080Ti environment.

- **APPLICATION OF CUDA STREAM** CUDA stream is an object in an asynchronous implementation of CUDA, a function that supports the GPU processor to receive data inputs and compute them simultaneously. In this paper, we proposed an effective parallel message copy of input data and kernel execution methods by applying CUDA stream and we provided the result of SHA-3 implementation using CUDA stream. Also, in SHA-3 implementation, performance using CUDA stream, we present the result of comparing Dat *et al.*'s proposal and our implementation [22], [24]. Our optimized implementation of SHA-3 using 3 CUDA streams provides the maximum throughput of 95.67Gb/s in a GTX 1070 environment. Compared to the performance of Dat at al.'s implementation with 3 CUDA streams in the GTX 1080 environment, our implementation provides a performance improvement of up to 73.53%, which proves the superiority of the proposed optimization methods.

II. RELATED WORKS

A. SHA-3 ALGORITHM

1) OVERVIEW OF SHA-3

In 2015, NIST selected the algorithm proposed by Keccak team as the standard hash function SHA-3 [26]. SHA-3 consist of sponge structure and internal operation process different from the existing standard hash functions SHA-1 and SHA-2. Sponge structure is resistant to the proposed attack methods of existing hash functions. SHA-1 and SHA-2 hash functions have a similar structure, many attack methods have been proposed [8], [9], and a collision pair for SHA-1 was actually discovered in 2017 [10]. In addition, in NIST PQC competition, many candidate algorithms are used SHA-3 and SHAKE for key setting and random number generation (SHAKE is an algorithm that utilizes the SHA-3 sponge structure) [17]–[20].

TABLE 1. SHA-3 internal values [26].

b	25	50	100	200	400	800	1600
l	0	1	2	3	4	5	6
w	1	2	4	8	16	32	64

2) THE INTERNAL STATE

In the SHA-3 algorithm, the size of the internal state depends on w. w and size of the state are shown in Table 1. The internal state is represented in the form of a three-dimensional cube using w as the z axis. The maximum value of x and y is fixed at 5, and the value of z consists of the maximum w. Therefore, the size of the internal state is $x \times y \times z$ bits. The structure of the internal state is shown in Figure 1. SHA-3 operates through internal state and sponge structure. Sponge structure is divided into an absorbing process and a squeezing process. Figure 2 shows the SHA-3 sponge structure.

Absorbing process is the process of absorbing message data into an internal state. Absorbing process first initializes the internal state to '0'. During the absorbing process, one block of r-bit messages and the internal state r-bit are

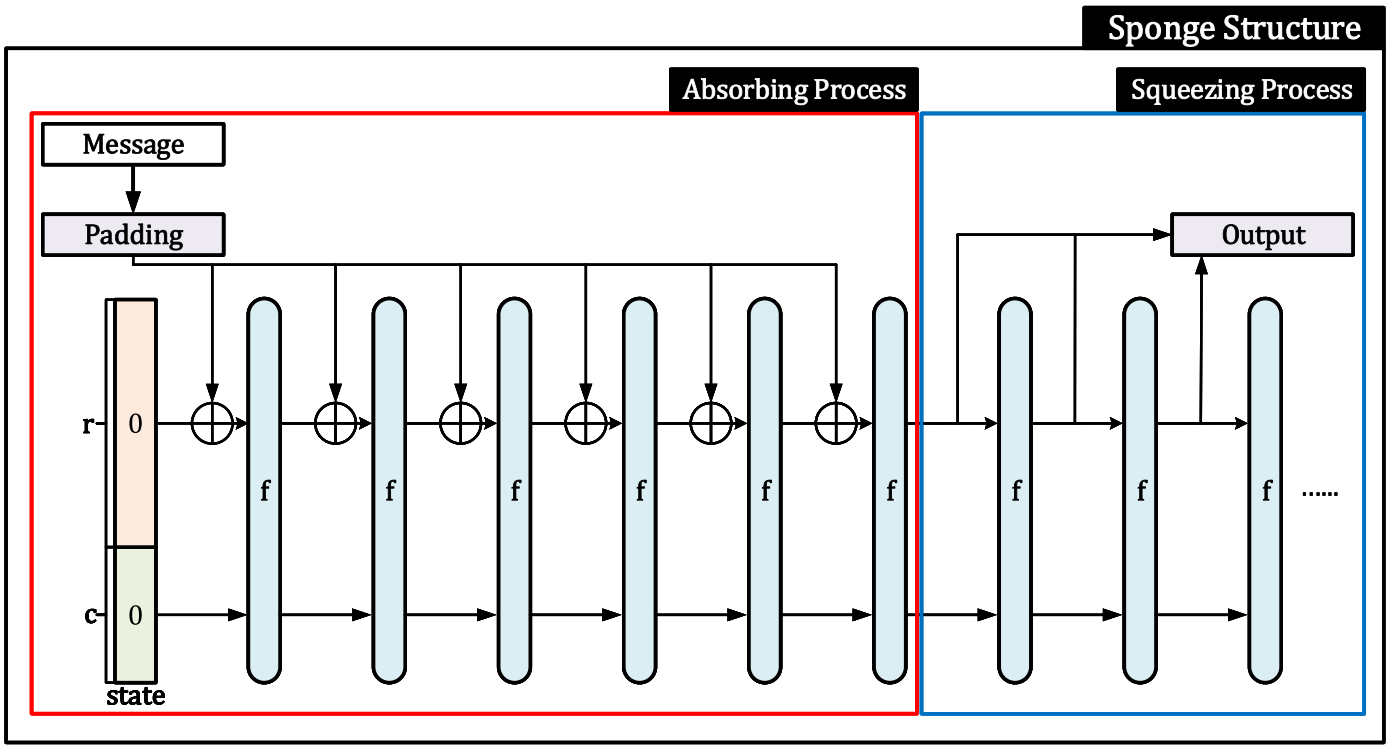


FIGURE 2. Sponge structure [26].

eXclusive OR(XOR)ed. After that the internal state is updated via the f function. This process repeats until all message blocks have been used. Squeezing process is a function of extracting a hash digest from an internal state, which is executed after the absorbing process is completed. When absorbing process is complete, the hash digest is allocated as much as the digest length in the internal state. If the requested hash digest length is larger than the internal state length, the output value is allocated as much as the internal state size and the internal state is updated again through the f function. After that, the value is extracted from the state as much as the remaining length.

3) f FUNCTION

In the SHA-3 sponge structure, f function consists of a total of five processes (θ, ρ, π, ι, and χ). Figure 3 shows the four internal processes(ρ, π, ι, and χ) of SHA-3.

θ process is an operation that changes the value of *lane*. The *lane* to be changed in θ operation is updated through the *sheet* values on both sides. Five *lane* belonging to *sheet* are combined into one *lane* through XOR operation. When the operation is finished, three *lane* are combined into one *lane* through XOR operation. Algorithm 1 shows the θ process.

ρ process is the operation of calculating the Rotation Left Shift (ROTL) of *lane*. Each *lane* has a different offset value of ROTL. The π process is the process that changes the location of *lane*. χ process is the operation of updating values through NOT and OR operations between *lane*s belonging to the same *plane*. ι process is the process that XORed with Round Constant (RC) values set in *lane* corresponding to $x = 0$ and $y = 0$. The internal state is updated through the five processes, and the five processes are repeated for a set number of rounds.

Algorithm 1 θ process in SHA-3 f functions [26]

Require: *internal state Array A*
Ensure: *internal state Array A′*
1: **for** $x = 0$ to 4 **do**
2: **for** $z = 0$ to $w - 1$ **do**
3: $C[x, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]$;
4: **end for**
5: **end for**
6: **for** $x = 0$ to 4 **do**
7: **for** $z = 0$ to $w - 1$ **do**
8: $D[x, z] = C[(x - 1) \bmod 5, z] \oplus C[(x + 1) \bmod 5, (z - 1) \bmod w]$;
9: **end for**
10: **end for**
11: **for** $x = 0$ to 4 **do**
12: **for** $y = 0$ to 5 **do**
13: **for** $z = 0$ to $w - 1$ **do**
14: $A'[x, y, z] = A[x, y, z] \oplus D[x, z]$;
15: **end for**
16: **end for**
17: **end for**

B. GPU ENVIRONMENT

GPU architecture was developed for graphics processing and high-definition image processing. With the development of GPU architecture, GPU architecture shows efficient performance, in parallel data processing. Based on this, High-Performance Computing (HPC) environments began to use GPUs. Figure 4 shows a summary of the GPU environment introduced in this section.

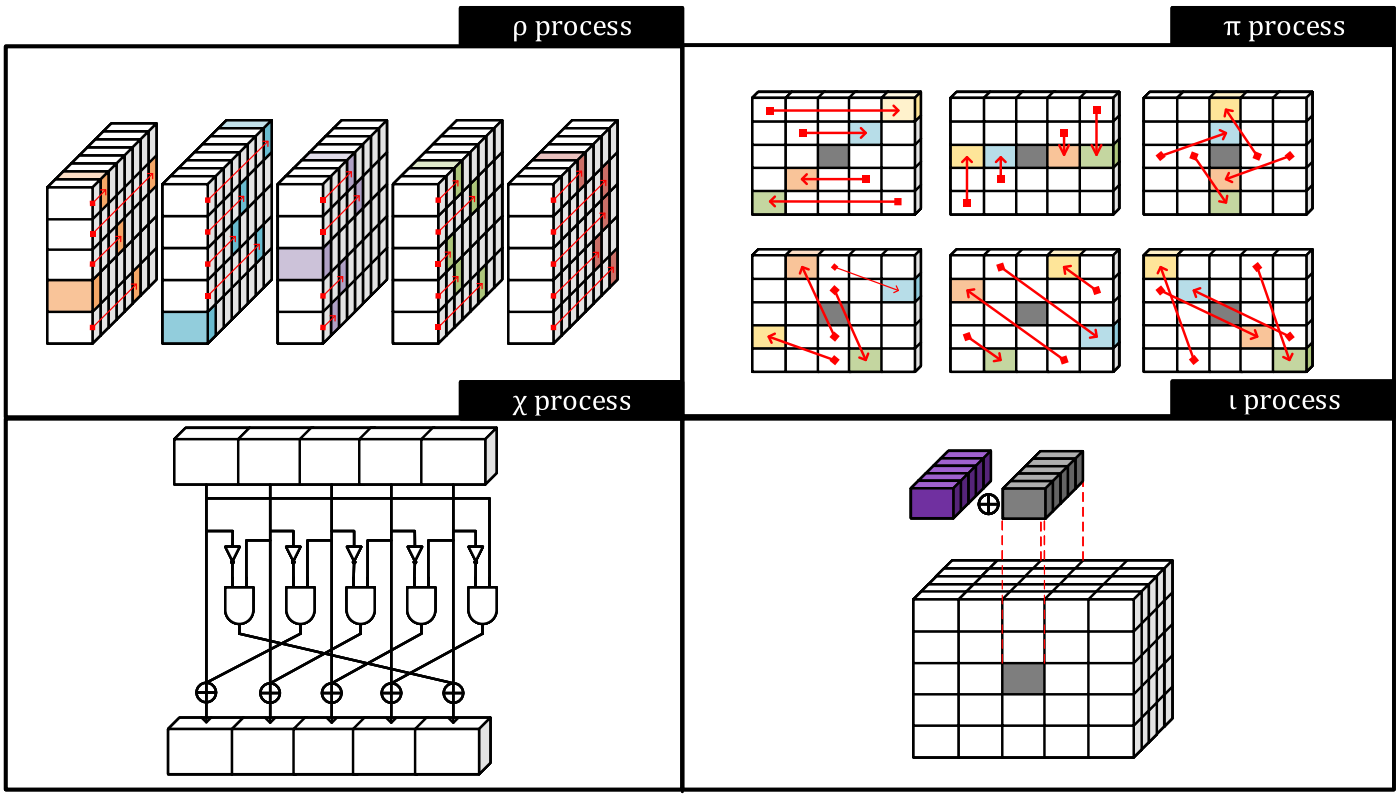


FIGURE 3. ρ, π, χ and ι process in SHA-3 f function [26].

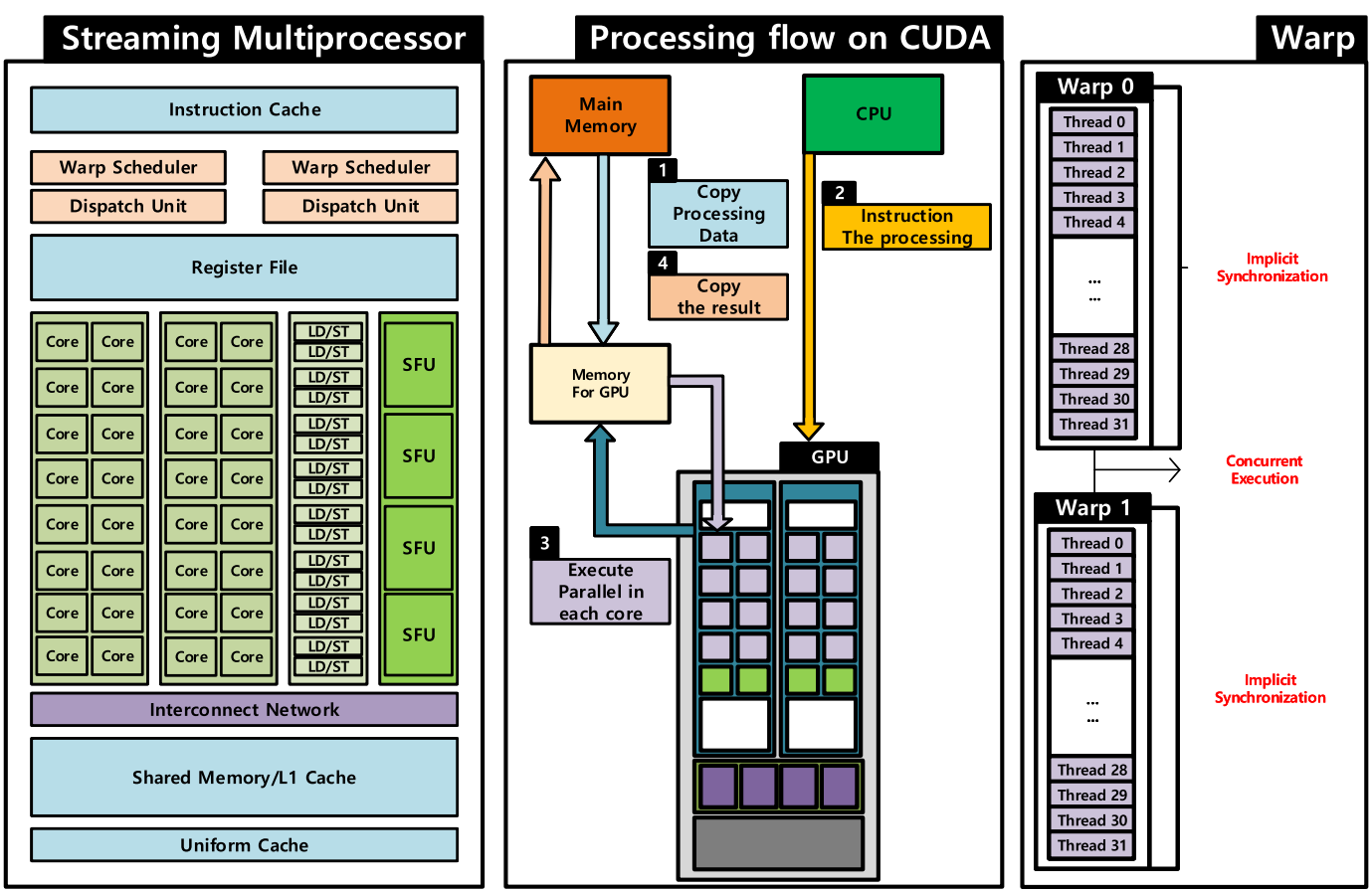


FIGURE 4. GPU environment summary.

GPU architecture consists of several Streaming Multiprocessors (SM). SM is designed to support the execution of hundreds of threads simultaneously. This means that there are multiple SMs for each GPU architecture, allowing thousands of threads to run concurrently. The warp is the basic unit that runs in SM and manages 32 threads. All threads belonging to warp execute the same command at the same time. When the kernel function starts, the grid's thread block is divided in SM. The threads in each block are divided into warp, and one warp is composed of 32 threads.

In 2006, NVIDIA announced the Fermi architecture and developed the CUDA as a GPGPU technology that could run on GPU devices. CUDA is a technology that allows developers to easily write algorithms that can run on GPUs in a variety of programming languages. Through CUDA technology, developers can access the instruction set and memory of the GPU, and CUDA was developed to use the GPU effectively.

Programs developed in the GPU environment using CUDA can effectively process a lot of data in parallel. Especially on servers, GPU architecture is used more efficiently to handle authentication and message integrity for many users. Also, GPU uses many threads to parallel computations (parallel processing of encryption/decryption of bulk data or parallel processing of hash operations to verify data integrity etc.).

C. EXISTING RESULTS OF SHA-3 IMPLEMENTATION

Dat *et al.* proposed an optimization method for storing a constant values used in SHA-3 internal functions [22]. SHA-3 uses constant value table in internal function (θ process, ι process and π process).

Dat *et al.* proposed a data store method with the fastest memory access after storing three constant values in the memory area of the GPU. In addition, Dat *et al.* compared the data memory access time of one-dimensional and two-dimensional arrays of internal state. Dat *et al.* proposed that one-dimensional arrays are more efficient for memory access. Finally, Dat *et al.* proposed the most efficient block and thread configuration in warp units. Dat *et al.* suggested 128 threads per block as the most efficient warp unit in a GTX 1080 environment. Their optimization method shows a maximum throughput of 20.5Gb/s in a GTX 1080 environment. In addition, they proposed SHA-3 implementation using three CUDA streams [24]. Using CUDA stream, they showed a maximum throughput of 64.58Gb/s.

Kim *et al.* proposed a SHA-3 optimization method in the 8-bit AVR environment [25]. Kim *et al.* proposed optimization methods that minimize memory access by changing the order of SHA-3 internal processes. In particular, in the θ process, values that are used repeatedly were pre-computed and stored in memory. Also, in Kim *et al.*'s implementation of SHA-3, the π process was handled implicitly through direct indexing. As a result, they proposed an optimization technique that minimizes memory access by changing the order of operations in the SHA-3 internal process. Kim *et al.*'s implementation of SHA-3 in a 8-bit AVR environment provided 1,073 Clock Per Byte(CPB).

III. PROPOSED OPTIMIZATION TECHNIQUES FOR SHA-3

In this section, we proposed an optimization technique for SHA-3. In particular, we proposed an optimization method for GPU environment, porting related work methods, and an optimization method through GPU technology.

A. DESIGN CRITERIA OF THE PROPOSED SOFTWARE

In this section, we summarized a CPU-GPU usage technology. CUDA technology usage of GPGPU is as follows. First, host(CPU) performs the message transfer process to device (GPU). Device stores the messages it receives from the host. Second, device operates tasks through the kernel function. The kernel function consists of the source code and the command process that generated by the host. During this process, when host performs the message transfer, host can set the number of threads and blocks used by the device. Devices use configured blocks and threads. And Devices operate a kernel function. In this paper, we configured the kernel function of the device as SHA-3.

B. OVERALL STRUCTURE OF THE PROPOSED SOFTWARE

In this section, we summarized the optimization method for SHA-3 each process. Figure 5 is an overall summary of our optimization method. Unlike other device devices, GPU architectures have high memory access latency. For certain kernel functions, the memory access latency is longer than the operation time of the kernel function. Therefore, in the GPU architecture, the number of memory accesses must be small for the optimization of the algorithm. Therefore, our optimization implementation applied a coalesced memory access.

In GPU architecture, algorithm optimization should consider not only memory access latency, but also the host's memory transfer latency. Our implementation has applied data copy parallelism over CUDA streams. CUDA streams allow one SM to perform the data copy process while another SM executes a device kernel function. In other words, our implementation's CUDA stream utilization is as follows: First, we divided the data into certain lengths. Second, one stream receives one block of data. After that, one SM executes kernel functions. During the second step, another CUDA stream receives one block of data. After that, SM executes the kernel function. In other words, our implementation utilizes CUDA streams to perform the kernel function and message copy process in parallel. Finally, in our implementation, all threads each compute SHA-3. That is, one thread computes a hash digest of one message.

C. COARSE GRAIN IMPLEMENTATION METHOD

In GPU architecture, algorithm implementation methods are generally divided into fine grain method and coarse grain method. In coarse grain method, one thread handles one operation. In fine grain method, multiple threads process a single operation. The fine grain method is efficient when the task is computationally demanding. The coarse grain

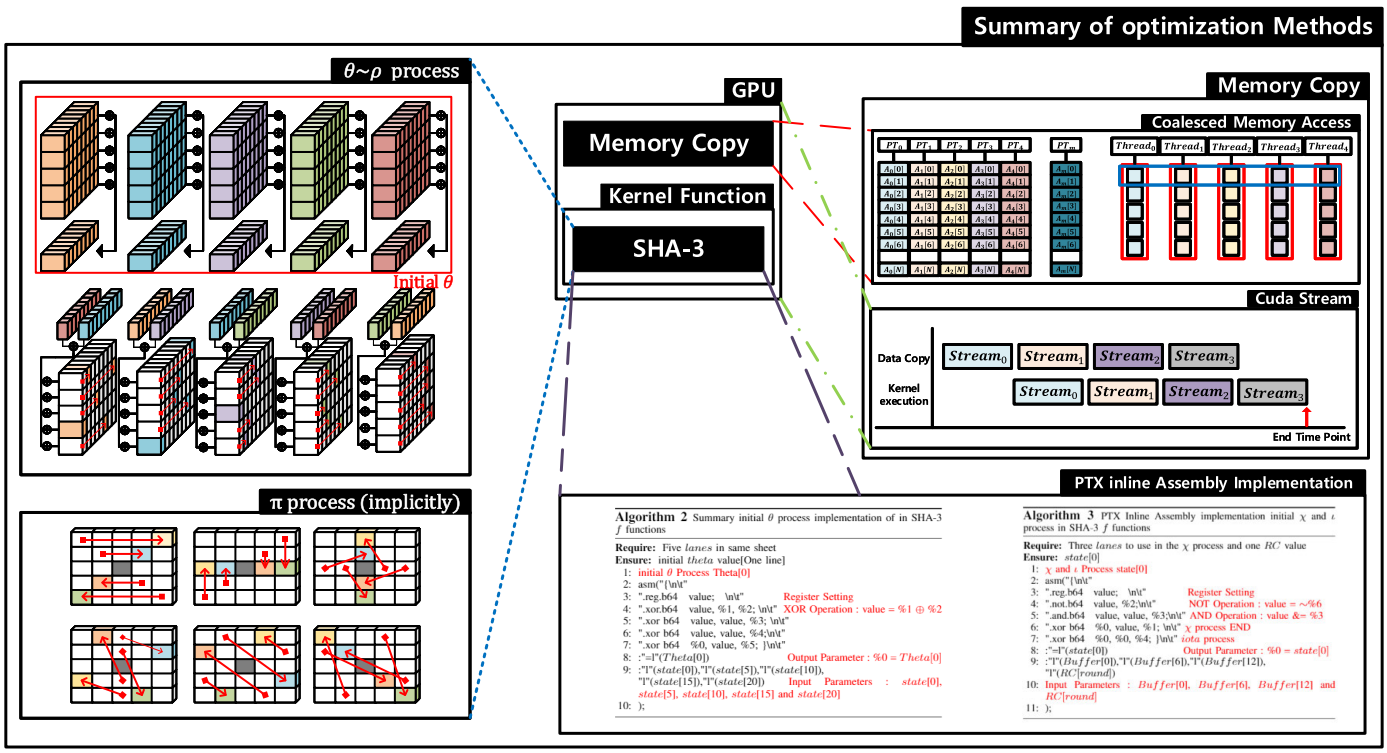


FIGURE 5. Summary of our optimization method.

method is efficient when the task requires less computation. Algorithms that apply the fine grain method can increase throughput, but there is a problem of high latency. In addition, the fine grain method involves thread synchronization. Therefore, in fine grain method, the thread synchronization process causes latency and the coarse grain method incurs memory access latency. Our implementation using coarse grain method minimizes the memory access latency by using CUDA streams. In addition, the coarse grain method does not have a thread synchronization process. Therefore, our implementation applies the coarse grain method.

D. OPTIMIZATION OF SHA-3 INTERNAL PROCESS

In this section, we proposed optimization methods for the SHA-3 internal process. In a related work, Kim *et al.* proposed a SHA-3 optimization technique in the 8-bit AVR environment [25]. Kim *et al.* proposed optimization method that minimized memory access by changing the order of SHA-3 internal process. In other related work, Dat *et al.* proposed an optimization method for constant values used in the SHA-3 internal algorithm [22]. In GPU environment, there are global memory, constant memory, and shared memory, each memory access latency is different. The SHA-3 operation process requires a pre-operation to store the constant values in GPU memory. Dat *et al.* stored constant values in each memory area and calculated the memory access latency. After that, Dat *et al.* proposed the most effective memory storage method.

We proposed a method to customize the optimization technique proposed by Kim *et al.* to the GPU environment. We focused on SHA-3 internal process optimization method proposed by Kim *et al.* and we proposed a more efficient technique than storing the SHA-3 internal function constant value memory as proposed by Dat *et al.* [22], [25]. In addition, we proposed the PTX Inline assembly implementation provided by CUDA C.

1) CUSTOMIZING TO OTHER OPTIMIZATION TECHNIQUE

Kim *et al.*'s first optimization method uses the fact that *lanes* belonging to the same *sheet* use the same value in the θ process [25]. In other words, in θ process, the values used in each *sheet* are pre-calculated and uploaded into memory, and then the stored values are called and used when each *lane* is calculated. When the optimization method is applied in θ process, the operation of five *lane* combines is replaced by one *lane* combine operation and five memory accesses.

Kim *et al.*'s second optimization method minimized memory access by changing the order of the internal processes of f function. The standard hash function implementation consists of θ -> ρ & π -> χ -> ι. Kim *et al.*'s optimization method combined ρ process with θ process and replaced π process with implicit direct indexing. By replacing the π process with implicit direct indexing, the constant value table of the π process is not used. Therefore, the constant values of the π process are not required, and the memory accesses and the memory loads to the constant value table are also removed.

We ported Kim *et al.*'s optimization method to GPU environment. That is, ρ process and θ process are combined in SHA-3 internal f function in our implementation. After that, π process was omitted through direct indexing. By processing

Algorithm 2 PTX Inline Assembly implementation initial θ process in SHA-3 f functions

Require: Five *lanes* in same sheet
Ensure: initial *theta* value[One line]

```
initial θ Process Theta[0]
asm("{\n\t"
".reg.b64   value;   \n\t"          Register Setting
".xor.b64   value, %1, %2; \n\t"    XOR Operation: value = %1 ⊕ %2
".xor b64   value, value, %3; \n\t"
".xor b64   value, value, %4;\n\t"
".xor b64   %0, value, %5; }\n\t"
:"=l" (Theta[0])                    Output Parameter: %0 = Theta[0]
:"l"(state[0]),"l"(state[5]),"l"(state[10]),
 "l"(state[15]),"l"(state[20])      Input Parameters: state[0], state[5],
state[10], state[15] and state[20]
);
```

Algorithm 3 PTX Inline Assembly implementation χ and ι process in SHA-3 f function

Require: Three *lanes* to use in the χ process and one *RC* value
Ensure: *state*[0]

```
χ and ι Process state[0]
asm("{\n\t"
".reg.b64   value;   \n\t"          Register Setting
".not.b64   value, %2;\n\t"         NOT Operation: value = ~%6
".and.b64   value, value, %3;\n\t"  AND Operation: value &= %3
".xor b64   %0, value, %1; \n\t"    χ process END
".xor b64   %0, %0, %4; }\n\t"      iota process
:"=l"(state[0])                     Output Parameter: %0 = state[0]
:"l"(Buffer[0]),"l"(Buffer[6]),"l"(Buffer[12]), "l"(RC[round])
Input Parameters: Buffer[0], Buffer[6], Buffer[12] and RC[round]
);
```

the π process through direct indexing, the constant value table used in the π process can be removed.

2) PTX INLINE IMPLEMENTATION

PTX is an inline assembly language available in CUDA C. Assembly language has the advantage of being able to directly correspond to machine language through instructions and can communicate directly with the machine (architecture). In other words, the execution speed of instructions is the fastest among programming languages. PTX is a GPU assembly language provided by CUDA C. PTX instruction can be effectively reduced the general instruction transmission and machine language conversion process. In this paper, we proposed a SHA-3 implementation method using PTX assembly implementation.

Algorithm 2 is a summary of θ process. Algorithm 2 proposes a calculation method for one *lane* of the initial θ process. That is, a value used repeatedly in the θ process are calculated using *lanes* belonging to the same *sheet*. θ process is performed using the PTX instruction. In θ process, the number of input parameters is 25 *lanes*, and a total of 5 combined *lanes* are output values.

Algorithm 3 summarizes χ and ι processes using PTX inline assembly. The three input parameters of the function are *lanes*. χ process updates the value with three *lanes* belonging to the same *plane*. π process must be completed before χ process. In our implementation, π process is handled through implicit indexing. That is, before the χ process, the location where *lane* moves through the π process is calculated, and then directly indexed into the input parameters of the algorithm and processed.

E. STRATEGY FOR OPTIMIZED MEMORY ACCESS OPERATION

In this section, we proposed an optimization method using the characteristics of GPU environment. In particular, we proposed an optimization method focusing on memory storage method and memory access latency in the GPU environment.

1) CONSTANT TABLE REDUCTION METHODS

There are many memory areas in GPU architecture. In CUDA C implementation, commonly used memory are shared memory, global memory and constant memory. Global memory is an area of memory that can be accessed by all elements of the GPU architecture. Global memory is the largest capacity, but has the slowest latency of memory access. Shared memory is a shared by threads belonging to the same block. In other words, threads belonging to the same block can share operation values in shared memory, but cannot access shared memory in other blocks. Memory access latency increases when a large number of threads access the same memory. Constant memory is a very small read-only memory. Memory access latency for cached data is very low. Constant memory has a low memory access latency when all threads belonging to the same warp read the same data.

TABLE 2. Dat et al's memory type of tables throughput performance(Hashing 512 blocks with 512 threads per block) [22].

Memory type			Throughput
π Table	ρ Table	ι Table	[GB/s]
Global	Global	Global	1.212
Constant	Constant	Constant	14.207
Constant	Constant	Shared	14.847
Constant	Shared	Constant	12.572
Constant	Shared	Shared	12.336
Shared	Constant	Constant	1.232
Shared	Constant	Shared	1.219
Shared	Shared	Constant	1.215
Shared	Shared	Shared	1.225

The internal process of f function uses several constant value tables. This constant value tables are stored in the GPU memory area. Stored constant tables are accessed during SHA-3 operation. Therefore, distributing the constant table in each memory area is also an important performance factor. Dat *et al.* stored the constant table used in SHA-3 in each memory area, and measured the performance of the access time [22]. The performance of each memory area is specified in the Table 2.

The difference between Dat *et al.*'s optimization method and our method is as follows. First, we removed the π table by applying the optimal method of Kim *et al.* [25]. Second, we have added removals for other constant value tables. Our implementation handled the ρ and ι process through direct indexing. As a result, our implementation does not use the SHA-3 constant table. In our implementation, SHA-3 π, ρ,

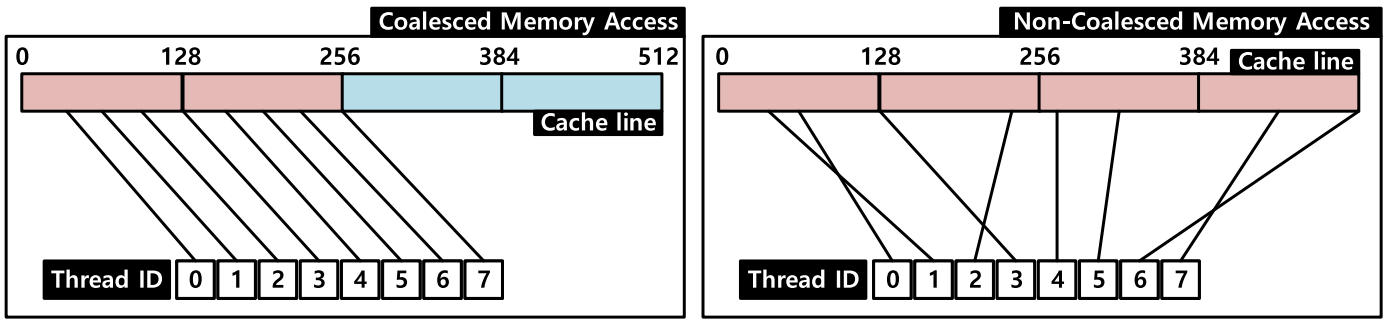


FIGURE 6. Coalesced memory access.

and ι process constant tables are all indexed through direct indexing. So in SHA-3 internals, our implementation does not have memory access. In other words, when our implementation of SHA-3 in GPU, a total of 24 * 75 table accesses are reduced compared to Dat et al's implementation [22].

2) APPLICATION OF COALESCED MEMORY ACCESS

On GPU architecture, memory access time causes a lot of performance degradation. While the GPU and CPU are running, the host (CPU) transmits data to be processed to the device (GPU). The transferred data is copied to the device's memory. After that, when the GPU accesses this memory, GPU accesses the message to be copied in units of warp. A warp consists of 32 threads, and memory copying is performed in units of warp. When warp accesses memory, all threads belonging to the same warp unit process the same memory instruction.

When warp accesses memory process, each thread access the data elements needed and the data is assigned to each thread. In this process, If the data's memory storage address is contiguous, Warp handles data access through one cache line. In other words, when the data elements are contiguous, memory access can be effectively achieved. This is called coalesced memory access. When a coalesced memory access method is not performed (when the memory request does not belong to one cache line or the memory address is not continuous), the cache line is accessed several times inside the warp. In other words, warp perform multiple memory accesses. In this paper, we proposed a method of changing data elements so that coalesced memory access is advantageous. Figure 6 shows the application of coalesced memory access.

When storing data in memory, the storage data method generally uses the row-by-row sequential storage method. In stores data in memory, the first element of each row will be the first element of each data. Each thread accesses memory in warp units to process each data element. When data elements are stored in memory as a one-dimensional array, each thread accesses the first element of the data element. If each data element does not belong to the same cache line, the warp will must access many cache lines. When data is sequentially stored in rows, the i-th data element address of each data has a non-sequential structure. This part is explained in Figure 7 Non-Coalesced Memory Access part. In particular, more inefficient memory access occurs if each message element is larger than a warp unit or one cache line.

To solve these inefficient memory accesses, the coalesced memory access method must be applied. Each thread belonging to the same warp is valid when its data address is sequential or contiguous. In this case, when the warp accesses the stored data, it accesses a small number of cache lines. To apply this technique, the data elements must be stored column-wise. That is, the first element of each data element is stored in the first row of the array. This method is explained in the coalesced memory access part of Figure 7.

However, to change the storage method of plaintext, the architecture must access the memory in which the plaintext is stored. When the GPU architecture changes the way plaintext is stored, memory accesses for changes and memory accesses for storage cause performance degradation. Therefore, the plaintext storage method must be performed on the CPU architecture. In our SHA-3 implementation and experimental measurements, the CPU architecture performs plaintext storage changes, and the GPU architecture performs kernel function (SHA-3) operations. In addition, our experimental measurements included both the CPU architecture and the time performed by the GPU architecture.

F. APPLICATION OF CUDA STREAM

When host calls a kernel function, host must transmit the data to the device. In this process, the kernel functions of the GPU do not work. Based on this fact, the kernel function does not work in the process of transmitting a message, CUDA stream perform message copying, parallel copying of kernel functions, and a parallel operation. To use CUDA stream, the host divides a long message into blocks of a certain size and transmits each message block to the device. When a message block is copied by the GPU, a kernel function is executed on the data. While the kernel function is executed on the GPU, the memory area of the GPU proceeds to copy messages to the next block.

In addition, CUDA stream can effectively work with multiple messages rather than long messages. When several messages are calculated as a general single stream, each message is combined into one message and transmitted to the GPU, or a kernel function is separately called and calculated

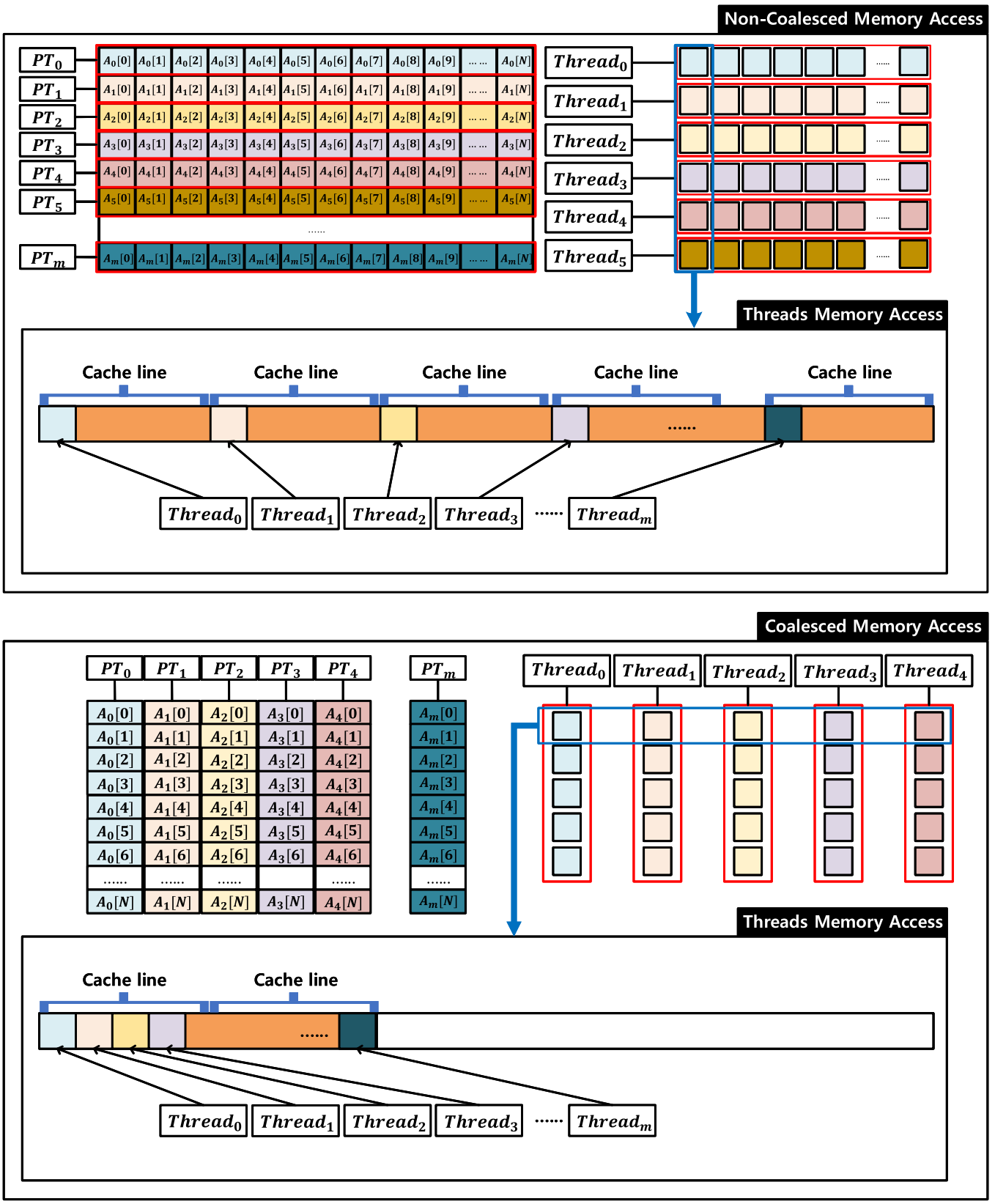


FIGURE 7. Application of coalesced memory access.

for each message. By applying CUDA streams, the process of combining multiple messages is not performed. That is, when CUDA stream is applied, each stream can obtain effective performance by processing one message element or the number of message elements per the total number of streams. We proposed a method of dividing one message into several blocks and calculating the hash digest through each stream. In our implementation, each stream is designed with a

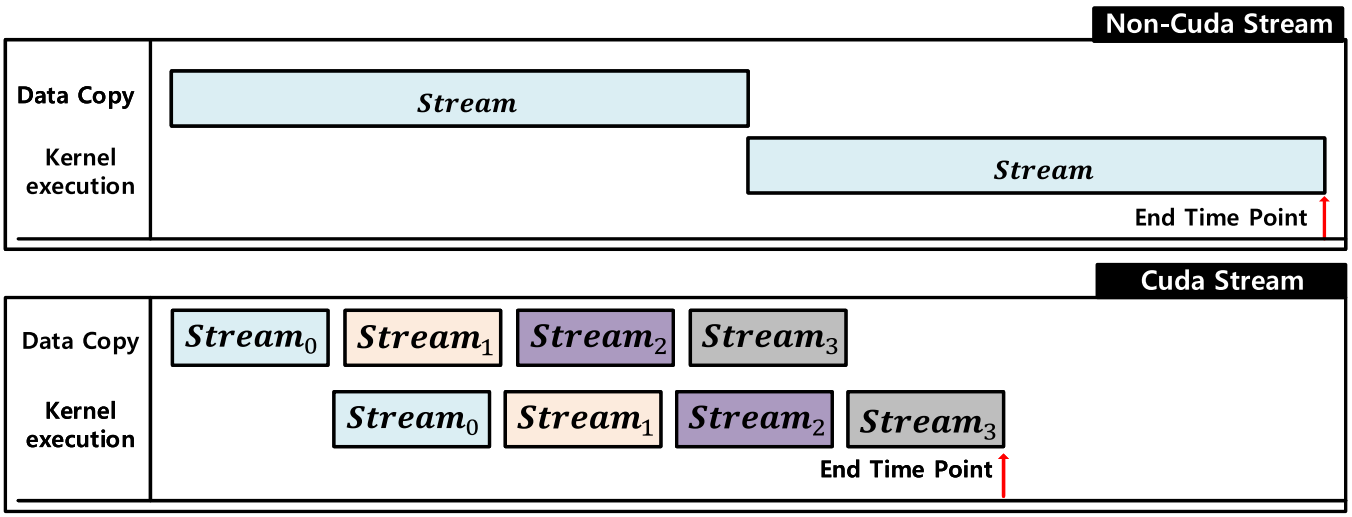


FIGURE 8. CUDA stream.

TABLE 3. GPU architecture specs.

	GTX 1070	GTX 1080	RTX 2080 Ti
Cores	1,920	2,560	4,352
Core clock speed	1,506MHz	1,607MHz	1,350MHz
boost clock speed	1,683MHz	1,733MHz	1,545MHz
Memory clock	2,002MHz	1,251MHz	1,750MHz
Memory bus	256-bit	256-bit	352-bit
Bandwidth	256.3Gb/s	320.3Gb/s	616.0Gb/s
Relative Performance	1	1.18	1.75

structure that computes a hash digest for a message block. Figure 8 shows a SHA-3 design plan using CUDA streams.

IV. PERFORMANCE ANALYSIS

In this section, we present SHA-3 operation performance results. In addition, we present the performance differences between our implementations and other implementations. Our performance measurement results were conducted in the environment of NVIDIA GTX 1070 architecture and RTX 2080Ti architecture, and the CPU was conducted in Intel(R) Core(TM) i7-10700K(3.60GHz) environment. The performance measurement unit used the maximum throughput (GiGa-byte per Second).

Table 3 is the specs of the GPU architecture used in our paper. Before measuring performance and checking results in GPU architecture, it is necessary to understand the structure of GPU architecture. Each GPU architecture has a different number of cores, core clock speed, and boost clock speed. The GTX 1070 architecture provides 1,920 cores, a core clock speed of 1,506 MHz, and a boost clock speed of 1,683 MHz. The performance measurement environment of Dat *et al.* is GTX 1080, and GTX 1080 provides 2,560 cores, 1,607 MHz core clock speed, and 1,733 MHz boost clock speed. The number of cores, core clock speed, and boost clock speed affect the performance of GPU architecture, and GTX 1080 architecture shows more efficient performance than GTX 1070 architecture. The RTX 2080 Ti architecture provides 4,352 cores, a core clock speed of 1,350 MHz, and a boost clock speed of 1,545 MHz. The RTX 2080 Ti architecture shows lower performance in core clock speed and boost clock speed than other GPU architectures, but the number of cores that process operations is roughly doubled. As a result, assuming the performance of the GTX 1070 is 1, the performance efficiency of the GTX 1080 is 1.18 times and that of RTX 2080 Ti is 1.75 times.

A. ANALYSIS WITHOUT THE APPLICATION OF CUDA STREAM

In this section, we present the performance measurement without applying CUDA stream. We compared the performance of SHA-3 with a single CUDA stream.

1) SHA-3(256)

Table 4 shows the performance of SHA-3(256). We measured performance on GTX 1070 architecture and RTX 2080 Ti architecture. In GTX 1070 environment, our implementation's the maximum throughput of SHA-3(256) is 59.87 Gb/s. And our implementation's the maximum throughput of SHA-3(256) in the RTX 2080 Ti environment is 171.62Gb/s.

2) SHA-3(512)

Table 5 shows the performance measurement results of SHA-3(512). In addition, table 5 shows a comparison with Dat *et al.*'s optimized implementation performance. In GTX 1070 environment, our optimized implementation has a maximum throughput of 30.20Gb/s. In RTX 2080 Ti architecture, our optimized implementation has a maximum throughput of 88.51 Gb/s. In addition, In addition, the results of our experiments conducted in the GTX 1070 environment show a performance improvement of about 20% compared to Dat *et al.*'s implementation. When comparing our implementation results measured in the GTX 1070 environment with Dat *et al.*'s experimental results measured in the GTX 1080 environment, our optimized implementation has a performance improvement of up to 49.73% compared to Dat *et al.*'s implementation. Considering the hardware

TABLE 4. Performance of SHA-3(256) without CUDA stream.

Plaintext(byte)	Number of Blocks	Number of Threads	Our Works (GTX 1070) [Gb/s]	Our Works (RTX 2080Ti) [Gb/s]
65536	1024	64	*47.48*	*90.51*
131072	1024	128	*54.13*	*136.96*
262144	2048	128	*56.35*	*144.85*
524288	4096	128	*57.38*	*146.89*
1048576	8192	128	*58.25*	*148.42*
2097152	16384	128	*59.27*	*153.33*
4194304	16384	256	*59.58*	*162.58*
8388608	16384	512	*59.87*	*171.62*

TABLE 5. Performance of SHA-3(512) without CUDA stream(figures in bracket are the performance enhancement compare with Dat *et al.* [22]).

Plaintext(byte)	Number of Blocks	Number of Threads	Dat et al. (GTX 1080) [Gb/s]	Our Works (GTX 1070) [Gb/s]	Our Works (RTX 2080Ti) [Gb/s]
65536	1024	64	19.28	*24.56(+27.39%)*	*48.78(+153.00%)*
131072	1024	128	19.98	*28.10(+40.64%)*	*69.10(+245.85%)*
262144	2048	128	20.29	*29.63(+46.03%)*	*73.48(+262.15%)*
524288	4096	128	20.38	*29.58(+45.14%)*	*73.58(+261.04%)*
1048576	8192	128	20.47	*30.15(+47.29%)*	*78.43(+283.15%)*
2097152	16384	128	20.51	*30.17(+49.73%)*	*80.24(+291.22%)*
4194304	16384	256	None	*30.20*	*83.47*
8388608	16384	512	None	*30.19*	*88.51*

TABLE 6. Performance of SHA-3(512) with three CUDA stream(figures in bracket are the performance enhancement compare with Dat *et al.* [22]).

Number of Message	Dat et al. [GTX 1080] 3 streams[Gb/s]	Our Works [GTX 1070] 3 streams[Gb/s]	Our Works [RTX 2080Ti] 3 streams[Gb/s]
65536	51.86	*75.53(+45.64%)*	*168.84(+225.56%)*
131072	53.91	*93.55(+73.53%)*	*252.49(+368.35%)*
262144	55.13	*95.06(+72.43%)*	*265.41(+381.43%)*
524288	64.07	*95.62(+49.24%)*	*268.52(+319.10%)*
1048576	64.58	*95.67(+48.14%)*	*271.82(+320.90%)*

specifications of the GTX 1070 architecture and GTX 1080 architecture, our performance improvement rate is expected to be higher than 49.73%.

In addition, Lee *et al.* proposed a study result of SHA-3 optimization in CUDA-SSL/TLS format [27]. Lee *et al.* effectively designed the internal structure of the f function using the warp shuffle method. Lee et al's implementation in GTX 1080 environment computes 38 million hashes per second. Our SHA-3(512) implementation in the GTX 1070 environment computes 449 million hashes per second, and our implementation provides a 11.81x performance improvement over that of Lee *et al.*

B. ANALYSIS WITH THE APPLICATION OF CUDA STREAM

In this section, we present the SHA-3 performance results using CUDA stream. A previous work by Dat *et al.* showed a throughput of up to 64.58 Gb/s using three CUDA streams [24]. We present the results of SHA-3 implementation in GTX 1070 and RTX 2080Ti architecture environments. The performance measurement result is the SHA-3 performance result using three CUDA streams.

TABLE 7. Performance of SHA-3(256) with three CUDA stream.

Plaintext(byte)	Our Works (GTX 1070) [Gb/s]	Our Works (RTX 2080Ti) [Gb/s]
16384	*111.04*	*201.25*
32768	*143.10*	*334.42*
65536	*151.38*	*426.88*
131072	*185.16*	*489.97*
262144	*188.86*	*503.85*
524288	*189.39*	*508.84*
1048576	*190.02*	*512.31*

Table 6 shows the performance results of SHA-3(512). In GTX 1070 environment, our implementation results show a throughput of up to 95.67Gb/s, and the maximum performance improvement rate is 73.53% compared to Dat *et al.* In RTX 2080 Ti environment, the result of our SHA-3(512) implementation shows a throughput of up to 271.82Gb/s, and the maximum performance improvement rate is 381.43% compared to the performance of Dat *et al.* Table 7 shows the performance measurement results of SHA-3(256) using three CUDA streams. In a GTX 1070

environment, our SHA3(256) optimized implementation has a maximum throughput of 190.02Gb/s. In an RTX 2080 environment, our implementation has a maximum throughput of 512.31 Gb/s.

V. CONCLUDING REMARKS

As SHA-3 and SHAKE are used in algorithm submitted to the PQC 3-round contest, the frequency of use SHA-3 increases. In addition, the use of SHA-3 is recommended as a continuous collision pair attack method has been proposed for the existing hash functions SHA-1 and SHA-2. In this paper, we proposed a SHA-3 optimal parallel implementation method in GPU environment. As a result, our SHA-3(256) optimized implementation on the GTX 1070 architecture provided a maximum throughput of 59.87 Gb/s. Our SHA-3 (256) optimized implementation on the RTX 2080 Ti architecture provided a maximum throughput of 171.62 Gb/s. Not only that, for the SHA-3(512) implementation, our optimized implementation in the GTX 1070 environment has a maximum throughput of 30.20 Gb/s, and the RTX 2080 Ti environment provides a maximum throughput of 88.51 Gb/s. Furthermore, without the application of cuda stream, our SHA-3(512) software on GTX1070 GPU provides about 49.73% improved throughput compared with the previous best work on GTX1080, which shows the superiority of our proposed optimization methods. To the best of our knowledge, our optimization implementation is the fastest implementation of the GPU architecture. Our optimization method can be applied to PQC algorithms using SHAKE and SHA-3 formats. In addition, our optimization method can be applied to authentication processors and hash function-based cryptographic algorithms.

REFERENCES

[1] I. Baldini, S. J. Fink, and E. R. Altman, "Predicting GPU performance from CPU runs using machine learning," in *Proc. 26th IEEE Int. Symp. Comput. Architecture High Perform. Comput. (SBAC-PAD)*. Paris, France: IEEE Computer Society, Oct. 2014, pp. 254–261.

[2] M. Amaris, R. Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram, "A comparison of GPU execution time prediction using machine learning and analytical modeling," in *Proc. IEEE 15th Int. Symp. Netw. Comput. Appl. (NCA)*, A. Pellegrini, A. Gkoulalas-Divanis, P. di Sanzo, and D. R. Avresky, Eds. Boston, MA, USA: IEEE Computer Society, Oct. 2016, pp. 326–333.

[3] J. Lew, D. A. Shah, S. Pati, S. Cattell, M. Zhang, A. Sandhupatla, C. Ng, N. Goli, M. D. Sinclair, T. G. Rogers, and T. M. Aamodt, "Analyzing machine learning workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Madison, WI, USA, Mar. 2019, pp. 151–152.

[4] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server," in *Proc. 11th Eur. Conf. Comput. Syst. (EuroSys)*, C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds., London, U.K., Apr. 2016, pp. 4:1–4:16.

[5] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. H. Liu, and C. Guo, "Tiresias: A GPU cluster manager for distributed deep learning," in *Proc. 16th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, J. R. Lorch and M. Yu, Eds. Boston, MA, USA: USENIX Association, Feb. 2019, pp. 485–500.

[6] S. An and S. C. Seo, "Highly efficient implementation of block ciphers on graphic processing units for massively large data," *Appl. Sci.*, vol. 10, no. 11, p. 3711, May 2020.

[7] S. An and S. C. Seo, "Efficient parallel implementations of LWE-based post-quantum cryptosystems on graphics processing units," *Mathematics*, vol. 8, no. 10, p. 1781, Oct. 2020.

[8] M. Stevens, "New collision attacks on SHA-1 based on optimal joint local-collision analysis," in *Advances in Cryptology—EUROCRYPT 2013* (Lecture Notes in Computer Science), vol. 7881, T. Johansson and P. Q. Nguyen, Eds. Athens, Greece: Springer, May 2013, pp. 245–261.

[9] G. Leurent and T. Peyrin, "SHA-1 is a shambles: First chosen-prefix collision on SHA-1 and application to the PGP web of trust," in *Proc. 29th USENIX Secur. Symp. (USENIX Security)*, S. Capkun and F. Roesner, Eds. Berkeley, CA, USA: USENIX Association, Aug. 2020, pp. 1839–1856.

[10] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full SHA-1," in *Advances in Cryptology—CRYPTO 2017* (Lecture Notes in Computer Science), vol. 10401, J. Katz and H. Shacham, Eds. Santa Barbara, CA, USA: Springer, Aug. 2017, pp. 570–596.

[11] S. K. Sanadhya and P. Sarkar, "New collision attacks against up to 24-step SHA-2," in *Progress in Cryptology—INDOCRYPT 2008* (Lecture Notes in Computer Science), vol. 5365, D. R. Chowdhury, V. Rijmen, and A. Das, Eds. Kharagpur, India: Springer, Dec. 2008, pp. 91–103.

[12] R. K. Dahal, J. Bhatta, and T. N. Dhamala, "Performance analysis of SHA-2 and SHA-3 finalist," *Int. J. Cryptogr. Inf. Secur.*, vol. 3, no. 3, pp. 720–730, 2013.

[13] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, Santa Fe, NM, USA: IEEE Computer Society, Nov. 1994, pp. 124–134.

[14] F. Arute *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, pp. 505–510, Oct. 2019.

[15] *IBM Delivers its Highest Quantum Volume to Date, Expanding the Computational Power of its IBM Cloud-Accessible Quantum Computers*. Accessed: Oct. 24, 2021. [Online]. Available: https://newsroom.ibm.com/2020-08-20-IBM-Delivers-Its-Highest-Quantum-Volume-to-Date-Expanding-the-Computational-Power-of-its-IBM-Cloud-Accessible-Quantum-Computers

[16] *IBM's Roadmap for Scaling Quantum Technology*. Accessed: Oct. 24, 2021. [Online]. Available: https://research.ibm.com/blog/ibm-quantum-roadmap

[17] H. Singh, "Code based cryptography: Classic McEliece," *CoRR*, vol. abs/1907.12754, pp. 1–45, Jul. 2019.

[18] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, and D. Stehlé, "CRYSTALS—Kyber: A CCA-secure module-lattice-based KEM," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 634, Jun. 2017.

[19] NTRU. *Algorithm Specifications and Supporting Documentation*. Accessed: Oct. 24, 2021. [Online]. Available: https://ntru.org/f/ntru-20190330.pdf

[20] *SABER: Mod-LWR Based KEM (Round 3 Submission)*. Accessed: Oct. 24, 2021. [Online]. Available: https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf

[21] P. Cayrel, G. Hoffmann, and M. Schneider, "GPU implementation of the Keccak hash function family," in *Information Security and Assurance* (Communications in Computer and Information Science), vol. 200, T. Kim, H. Adeli, R. J. Robles, and M. O. Balitanas, Eds. Brno, Czech Republic: Springer, Aug. 2011, pp. 33–42.

[22] T. N. Dat, K. Iwai, and T. Kurokawa, "Implementation of high speed hash function keccak using CUDA on GTX 1080," in *Proc. 5th Int. Symp. Comput. Netw. (CANDAR)*. Aomori, Japan: IEEE Computer Society, Nov. 2017, pp. 475–481.

[23] C. Wang and X. Chu, "GPU accelerated Keccak (SHA3) algorithm," *CoRR*, vol. abs/1902.05320, pp. 1–11, Feb. 2019.

[24] T. N. Dat, K. Iwai, T. Matsubara, and T. Kurokawa, "Implementation of high speed hash function Keccak on GPU," *Int. J. Netw. Comput.*, vol. 9, no. 2, pp. 370–389, 2019.

[25] Y. Kim, H. Choi, and S. C. Seo, "Efficient implementation of SHA-3 hash function on 8-bit AVR-based sensor nodes," in *Information Security and Cryptology—ICISC 2020* (Lecture Notes in Computer Science), vol. 12593, D. Hong, Ed. Seoul, South Korea: Springer, Dec. 2020, pp. 140–154.

[26] *FIPS 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Accessed: Oct. 24, 2021. [Online]. Available: https://csrc.nist.gov/publications/detail/fips/202/final

[27] W.-K. Lee, X.-F. Wong, B.-M. Goi, and R. C.-W. Phan, "CUDA-SSL: SSL/TLS accelerated by GPU," in *Proc. Int. Carnahan Conf. Secur. Technol. (ICCST)*, Madrid, Spain, Oct. 2017, pp. 1–6.