International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 07 Issue: 07 | July - 2023 SJIF Rating: 8.176 ISSN: 2582-3930

Enhancing GPU Performance for SHA-3 Algorithm: Optimizing Hashing Operations in a Parallel Computing Environment

M.Vasuki[1], A. Karunamurthy[2], G. Pothyeswari[3]
[1] Associate Professor, Department of Master of Computer Application, Sri Manakula Vinayagar Engineering College, Puducherry-605 107.India
[2] Associate Professor, Department of Master of Computer Application, Sri Manakula Vinayagar Engineering College, Puducherry-605 107.India
[3] Student, Department of Master of Computer Application, Sri Manakula Vinayagar Engineering College, Puducherry-605 107.India
pothyeswaripothyes@gmail.com

ABSTRACT

Software implementation of Hash function have not been able to offer satisfactory performances for various application thus far. Additionally, SHA-3 and SHAKE, which utilize SHA-3, are extensively utilized in many Post Quantum Cryptosystem (PQC) . Therefore, there is a need for research to optimize SHA-3 in the software environments. Our proposal involves an optimized software implementation of SHA-3 on a GPU environment. To improve performance efficiency, we suggest various techniques such as optimizing the internal processes of SHA-3, inline PTX optimization, efficient memory usage, and asynchronous CUDA stream application.

After implementing these optimization methods, our SHA-3(512) (and SHA-3(256)) algorithm provides a maximum throughput of 88.51 Gb/s (and 171.62 Gb/s) on the RTX2080Ti GPU without CUDA stream. The proposal aims to optimize the software implementation of SHA-3 in a GPU environment to enhance performance efficiency. The suggested techniques include internal process optimization of SHA-3, inline PTX optimization, efficient memory usage, and the application of an asynchronous CUDA stream. After applying these optimization methods, our SHA-3(512) and SHA-3(256) algorithms provide a maximum throughput of 88.51 Gb/s and 171.62 Gb/s, respectively, on the RTX2080Ti GPU without CUDA stream.

INTRODUCTION

The architecture of Graphics Processing Units (GPUs) is primarily intended for graphics and image processing, making it a highly effective device for parallel data processing across multiple threads. In recent times, GPUs have found widespread usage across diverse fields like deep learning, machine learning, and cryptographic algorithms

Furthermore, as hardware specifications continue to advance, there has been an emergence of General-Purpose computing on Graphics Processing Unit (GPGPU) that enables the handling of various applications on GPU architectures.

With the advent of GPGPU technology, GPU architecture has found applications in various fields such as embedded systems, artificial intelligence, driverless vehicles, and cloud computing, among others.

The architecture is particularly effective in parallel data processing, especially in server environments that accommodate multiple clients. For instance, the server can store the client's data as ciphertext, which can then be encrypted by the server or processed effectively through parallel processing using GPU architecture, particularly when the data is substantial. Additionally, when clients access the server, the server must authenticate the clients and verify the data integrity. In this process, the server employs cryptographic hash functions for authentication and data integrity verification.

Standard cryptographic hash functions, such as Secure Hash Algorithm (SHA)-1, SHA-2, and SHA-3, are commonly used. In 2015, Keccek Algorithm was selected as SHA-3 by the National Institute of Standards and Technology (NIST) through a contest. Despite their widespread usage, many researchers have proposed attack methods to SHA-1 and SHA-2 [8], [9]. In 2017, a real collision pair was discovered in SHA-1 [10]. Since SHA-2 has a structure similar to SHA-1, some researchers have also proposed a SHA-2 attack method similar to the SHA-1 attack method [11]. Furthermore, SHA-3 demonstrates performance two times slower than SHA-1 and SHA-2 in a software environment [12]. Therefore, optimization studies of SHA-3 in software environments are necessary. Notably, SHA-3 is employed in the National Institute of Standards and Technology (NIST) Post Quantum Cryptography (PQC) contest submission algorithm.

In 1994, Shor proposed an effective factorization algorithm in the quantum computing environment [13]. At the time, Shor's proposal was purely theoretical since quantum computing was not yet developed. However, many quantum computing development methods have been proposed since then. In 2019, Google proposed a quantum computing development method using 53 qubits [14]. In 2020, IBM developed the Quantum Volume 64, a 65-qubit Quantum Hummingbird processor, and is also working on developing the 127-qubit IBM Quantum Eagle Processor [15], [16].

These advances in quantum computing make Shor's effective factorization algorithm feasible, which is crucial to the current public key cryptographic algorithms. The RSA algorithm is a public key cryptographic algorithm designed based on the mathematical challenges of discrete logarithm and factorization. However, the development of quantum computing poses a threat to RSA cryptographic algorithms. Thus, alternative algorithms to the existing public key encryption algorithm are required. To replace RSA and RSA-based digital signature schemes, NIST is conducting a PQC competitive contest. Currently, the cryptographic algorithm submitted to NIST-PQC round 3 uses SHA-3 and SHAKE to construct key and nonce establishment [17]-[20].

Several research studies have focused on optimizing SHA-3 in a GPU environment since the algorithm was submitted to the NIST standard. Kim et al. proposed an optimization method for SHA-3 in an 8-bit AVR environment, which we adapted to the GPU environment. Dat et al. proposed a memory access-related optimization method for SHA-3 in the GPU environment, which we customized for our implementation. They also extended their optimization methods using three CUDA streams in a separate study.

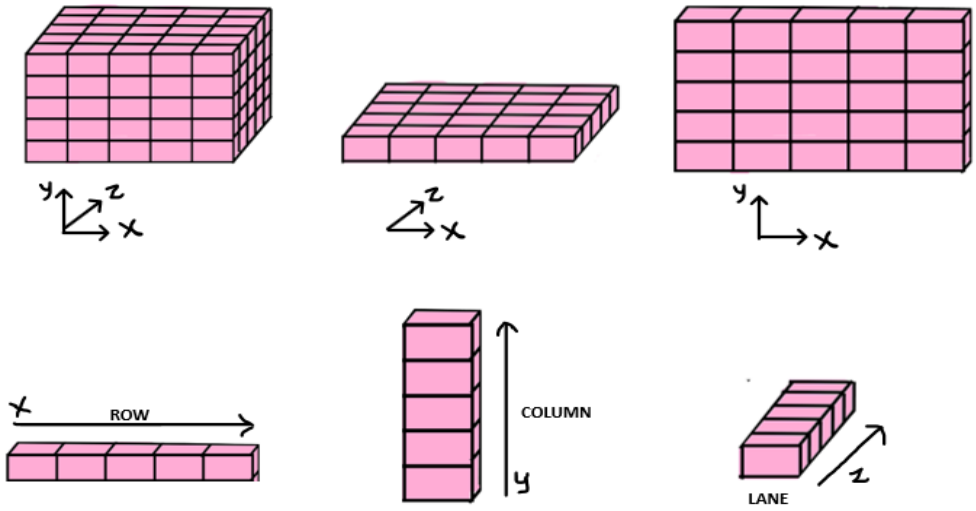

FIGURE 1: Internal state of SHA-3

The objective of this paper is to present a novel optimization method for SHA-3 in a GPU environment. Our approach focuses on improving the performance of SHA-3 software on the GPU by optimizing internal processing and memory access. To achieve this, we build on the work of Kim et al. who developed an SHA-3 implementation for an 8-bit AVR environment, and adapt their ideas for use in a GPU environment. Additionally, we reduce the constant value table of SHA-3 internal functions to minimize the number of memory accesses required for processing. Overall, our implementation offers significant performance improvements for SHA-3 in a GPU environment.

In summary, our paper proposes an efficient implementation approach for SHA-3 in GPU environments, utilizing optimized internal operations and the application of Parallel Thread eXecution (PTX) and CUDA streams. PTX is an inline assembly language that is available in CUDA C, and CUDA stream is a technology that is effective in managing memory copying and kernel functions. Based on our knowledge, our optimization implementation is currently the fastest among GPU architectures.

RELATED WORKS

A. SHA-3 ALGORITHM

1) OVERVIEW OF SHA-3

NIST selected the Keccak team's algorithm as the standard hash function SHA-3 in 2015. Unlike SHA-1 and SHA-2, SHA-3 uses a sponge structure and unique internal operations. This structure is specifically designed to resist proposed attack methods used against other hash functions. Unlike SHA-1 and SHA-2, which have shown vulnerability to various attack methods [8], [9], a collision pair for SHA-1 was discovered in 2017 [10]. Additionally, during the NIST PQC competition, SHA-3 and SHAKE were widely utilized by candidate algorithms for key establishment and random number generation. Notably, SHAKE makes use of the SHA-3 sponge structure [17]–[20].

b	25	50	100	200	400	800	1600	3200	6400
l	0	1	2	3	4	5	6	7	8
w	1	2	4	8	16	32	64	128	256

TABLE 1: Internal value OF SHA-3

2) THE INTERNAL STATE

The internal state size in the SHA-3 algorithm is determined by the value of w, as indicated in Table 1. The internal state is visualized as a three-dimensional cube, with w serving as the z-axis. The maximum values for x and y are fixed at 5, and the value of z corresponds to the maximum value of w. Consequently, the internal state size is given by x × y × z bits. The structure of the internal state can be seen in Figure 1. SHA-3 operates by utilizing both the internal state and the sponge structure, which consists of an absorbing process and a squeezing process.

The absorbing process involves incorporating message data into an initial internal state set to '0'. In this process, each block of r-bit messages is XORed with the r-bit internal state, and then the internal state is updated using the f function. This process continues until all message blocks have been processed. Once the absorbing process is finished, the squeezing process is performed to extract a hash digest from the internal state.

Once the absorption process concludes, the hash digest is assigned a space equal to the length of the digest within the internal state. In cases where the desired hash digest length exceeds the length of the internal state, the output value is allocated space equal to the internal state size, and the internal state is subsequently updated using the f function. Following this, the value is extracted from the state to fill the remaining length.

3) f FUNCTION

The SHA-3 sponge structure incorporates a series of five processes (θ, ρ, π, ι, and χ) within its f function. Among these processes, ρ, π, ι, and χ are depicted in Figure 1, illustrating the internal operations of SHA-3.

ALGORITHM process in SHA-3

```
for x=0 to 5 do
for z=0 to w-1 do
C [x, z] =A [x, 0, z] ⊕A [x, 1, z] ⊕A [x, 2, z]⊕ A [x, 3, z]⊕ A [x, 4, z]⊕ A [x, 5, z];
end for
end for
for x=0 to 5 do
for z=0 to w-1 do
D [x, y] =C[(x-1) mod 6, z]⊕C [(x+1) mod 6, (z-1) mod w];
end for
end for
for x=0 to 5 do
for y=0 to 6 do
for z=0 to w-1 do
A' [x, y, z] = a [x, y, z] ⊕D [x, z];
end for
end for
end for
```

The θ process involves modifying the value of a lane by updating it based on the neighboring sheet values through an XOR operation. In this process, five lanes belonging to a sheet are combined into a single lane using XOR. Once the θ operation is complete, three lanes are further combined into one lane through XOR. Algorithm 1 illustrates the θ process. On the other hand, the ρ process calculates the Rotation Left Shift (ROTL) for each lane, with each lane having a unique offset value for the ROTL operation.

The π process involves altering the lane positions, while the χ process updates values by performing NOT and OR operations on lanes within the same plane. The ι process XORs the Round Constant (RC) values with the lane corresponding to x = 0 and y = 0. The internal state undergoes updates through these five processes, which are repeated for a specified number of rounds.

B. GPU ENVIRONMENT

The GPU architecture was originally created for graphics processing and high-definition image processing. Over time, it has demonstrated exceptional performance in parallel data processing. As a result, High Performance Computing (HPC) environments started adopting GPUs. Refer to Figure 2 for an overview of the GPU environment discussed in this section.

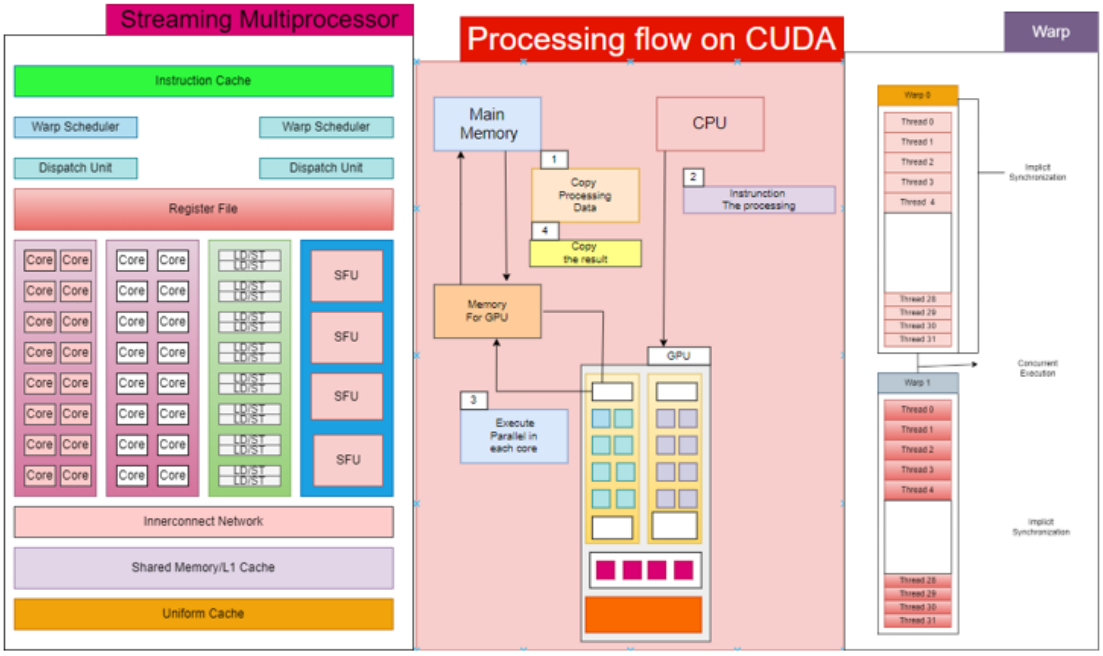

FIGURE 2: Summary of GPU environment

The GPU architecture comprises multiple Streaming Multiprocessors (SM), each designed to support the simultaneous execution of numerous threads. This enables concurrent execution of thousands of threads by having multiple SMs within a GPU architecture. Within an SM, the warp acts as the fundamental unit, managing 32 threads. All threads within a warp execute the same command simultaneously. When the kernel function begins, the thread block of the grid is divided into SMs. The threads within each block are further divided into warps, with each warp consisting of 32 threads. In 2006, NVIDIA introduced the Fermi architecture and developed CUDA as a GPGPU technology compatible with GPU devices. CUDA simplifies the process of writing algorithms that can be executed on GPUs, supporting various programming languages.

CUDA technology enables developers to access the instruction set and memory of the GPU, providing effective utilization of the GPU. By developing programs in the GPU environment with CUDA, data processing in parallel can be performed efficiently, particularly on servers where GPU architecture is employed to handle authentication and ensure message integrity for numerous users. Additionally, the GPU leverages multiple threads to enable parallel computations such as bulk data encryption/decryption or hash operations for data integrity verification.

EXISTING RESULTS FOR SHA-3 USAGE

An optimization method has been proposed to store constant values utilized in the internal functions (θ process, ι process, and π process) of SHA-3. These internal functions of SHA-3 depend on a table of constant values.

An alternative proposal introduces a technique for achieving the fastest memory access when storing data by utilizing three constant values in the memory area of the GPU. In their study, Dat et al. compared the data memory access time between one-dimensional and two-dimensional arrays of the internal state and determined that one-dimensional arrays offer greater efficiency in terms of memory access. They also put forth an optimized configuration for blocks and threads in warp units, suggesting that utilizing 128 threads

per block yields the highest efficiency in a GTX 1080 environment. According to their optimization approach, the maximum throughput achievable in a GTX 1080 environment is 20.5 Gb/s. Additionally, they proposed an SHA-3 implementation employing three CUDA streams, demonstrating a maximum throughput of 64.58 Gb/s.

An alternative proposal aims to optimize SHA-3 specifically for the 8-bit AVR environment. This optimization approach focuses on reducing memory access by altering the sequence of SHA-3's internal processes. In particular, the θ process precomputes and stores frequently used values in memory to minimize repeated calculations. Furthermore, the implementation of SHA-3 implicitly manages the π process through direct indexing. Consequently, the proposal suggests rearranging the order of operations within SHA-3's internal process as an optimization technique that reduces memory access.

PROPOSED SYSTEM FOR SHA-3 USAGE

In this section, we present an optimization technique for SHA-3. Specifically, we focus on optimizing SHA-3 for the GPU environment. We also discuss the process of adapting existing methods and utilizing GPU technology for optimization.

A. DESIGN CRITERIA:

This section presents a summary of CPU-GPU utilization technology, focusing specifically on the utilization of CUDA technology for GPGPU. The utilization process involves the host (CPU) transferring messages to the device (GPU), where the messages are stored. Subsequently, the device carries out tasks using the kernel function, which comprises the source code and the command process generated by the host. Throughout this process, the host has the ability to define the number of threads and blocks to be utilized by the device, which the device will employ to execute the configured kernel function. In our research, we configured the device's kernel function to perform SHA-3.

B. OVERALL STRUCTURE:

This section presents a summary of the optimization method employed for each process of SHA-3. GPU architectures, unlike other devices, exhibit high memory access latency. In certain kernel functions, the memory access latency exceeds the operation time of the kernel function itself. As a result, minimizing the number of memory accesses is crucial for optimizing the algorithm in the GPU architecture. To achieve this, our optimization implementation incorporates coalesced memory access. When optimizing algorithms in the GPU architecture, it is essential to consider not only the memory access latency but also the latency associated with memory transfers between the host and the GPU.

Our implementation incorporates data copy parallelism using CUDA streams, enabling concurrent execution of data copy and device kernel functions on different streaming multiprocessors (SMs). In our approach, we divided the data into specific lengths and assigned each block to a separate stream. One SM executes kernel functions while another stream receives and processes the next block of data. This utilization of CUDA streams allows for simultaneous execution of kernel functions and message copying. Ultimately, in our implementation, each thread performs SHA-3 computations independently, with one thread dedicated to computing the hash digest for each message.

C. IMPLEMENTATION MEHTOD:

In GPU architecture, algorithm implementation methods in general are categorized into fine grain and coarse grain approaches. In the coarse grain method, each thread handles a single operation, while in the fine grain method, multiple threads collaborate to process a single operation. The fine grain method proves efficient for computationally intensive tasks, but it suffers from high latency. Additionally, thread synchronization is required in the fine grain method, further contributing to latency. On the other hand, the coarse grain method is efficient for tasks with lower computational demands and avoids the need for thread synchronization. However, it incurs memory access latency. To address this, our implementation employs the coarse grain method with CUDA streams, effectively minimizing memory access latency. Furthermore, our approach eliminates the need for thread synchronization, making the coarse grain method more suitable for our implementation.

D. OPTIMIZATION OF SHA-3:

In this section, we present optimization methods for the internal process of SHA-3. In a related study, Kim et al. introduced an optimization technique for SHA-3 specifically designed for the 8-bit AVR environment. Their method aimed to minimize memory access by altering the order of the SHA-3 internal process. Another related work by Dat et al. focused on optimizing constant values used in the SHA-3 internal algorithm. Since GPU environments consist of global memory, constant memory, and shared memory, each with varying memory access latency, the SHA-3 operation process necessitates a preliminary operation to store constant values in GPU memory. Dat et al. stored constant values in different memory areas and assessed the memory access latency to propose the most effective memory storage method. In our approach, we tailored Kim et al.'s optimization technique to suit the GPU environment.

Our focus was on optimizing the internal process of SHA-3 based on the method proposed by Kim et al. Our proposed technique is more efficient than the earlier approach of storing the SHA-3 internal function constant values in memory. Furthermore, we introduced the implementation of PTX Inline assembly using CUDA C.

E. STRATEGY FOR OPTIMIZED MEMORY ACCESS OPERATION

1. APPLICATION OF COALESCED MEMORY ACCESS

Memory access time on GPU architecture significantly impacts performance, leading to considerable degradation. During the operation of both the GPU and CPU, data for processing is transmitted from the host (CPU) to the device (GPU). This transferred data is then copied to the device's memory. Subsequently, when the GPU accesses this memory, it does so in warp-sized units. A warp comprises 32 threads, and memory copying occurs at the warp level. When a warp accesses memory, all threads within that warp simultaneously process the same memory instruction.

During the memory access process, each thread in a warp retrieves the required data elements and assigns them to individual threads. If the memory storage address of the data is contiguous, the warp can access the data through a single cache line. This implies that contiguous data elements enable efficient memory access, known as coalesced memory access. However, when coalesced memory access is not

possible (due to the memory request spanning multiple cache lines or non-contiguous memory addresses), the cache line is accessed multiple times within the warp, resulting in multiple memory accesses. To address this issue, we propose a method in this paper that rearranges the data elements to facilitate coalesced memory access. Figure 3 illustrates the application of coalesced memory access.

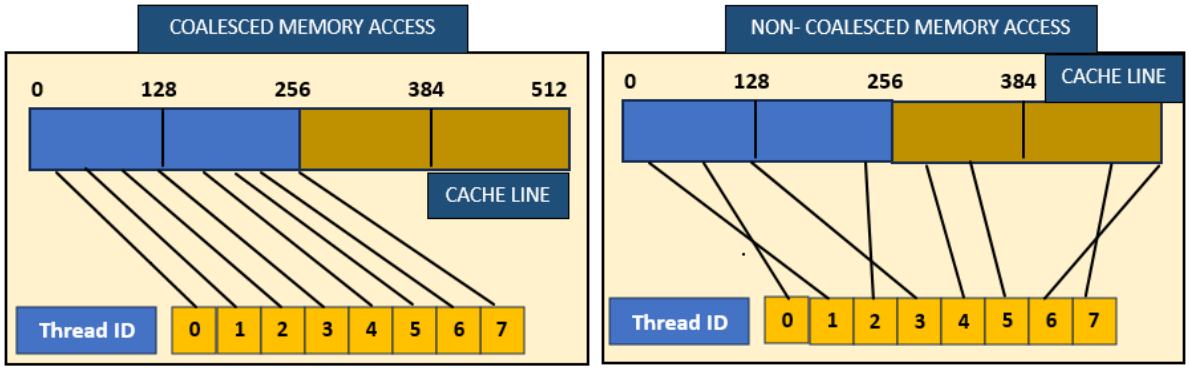

FIGURE 3: Coalesced memory access

Data is typically stored in memory using the row-by-row sequential storage method. In this method, the first element of each row becomes the first element of the stored data. When processing data elements, each thread accesses memory in warp units. If the data elements are stored in memory as a one-dimensional array, each thread accesses the first element of each data element.However, if the data elements do not belong to the same cache line, the warp will need to access multiple cache lines. Additionally, when data is sequentially stored in rows, the addresses of the i-th data element of each data exhibit a non-sequential structure.

Specifically, when each message element exceeds the size of a warp unit or a cache line, it leads to increased inefficiency in memory access. The solution to address these inefficiencies is to implement the coalesced memory access method. For the method to work, it requires that the data addresses of each thread within a warp are sequential or contiguous. Consequently, when the warp accesses the stored data, it only accesses a few cache lines. To implement this technique, the data elements need to be stored in a column-wise fashion, where the first element of each data element is stored in the first row of the array.

Nevertheless, in order to modify the storage approach for plaintext, it is necessary for the architecture to access the memory where the plaintext is currently stored. This modification in the GPU architecture's plaintext storage methodology leads to a decline in performance due to increased memory accesses for modifications and storage. Consequently, the task of altering the plaintext storage method should be carried out on the CPU architecture. In our implementation and experimental assessments of SHA-3, the CPU architecture handles the plaintext storage modifications, while the GPU architecture focuses on executing the kernel function (SHA-3) operations. Moreover, our experimental measurements encompassed both the CPU architecture and the time taken by the GPU architecture.

2. APPLICATION OF CUDA STREAM

When a kernel function is called by the host, the data needs to be transmitted to the device. However, during this transmission process, the kernel functions of the GPU remain inactive. Consequently, the kernel function does not function while a message is being transmitted. To address this limitation, the CUDA stream performs message copying, parallel copying of kernel functions, and parallel operations. To utilize the CUDA stream, the host divides a lengthy message into blocks of a specific size and transmits each block to the device. As the GPU copies a message block, a kernel function is executed on the data. Simultaneously, while the kernel function is being executed on the GPU, the memory area of the GPU proceeds to copy messages to the next block.

	GTX 1070	GTX 1080	RTX 2080 Ti
Cores	1,920	2,560	4,352
Cores clock speed	1,506 MHz	1,607MHz	1,350MHZ
Boost clock speed	1,683MHz	1,733MHz	1,545MHz
Memory clock	2,2002MHz	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

Furthermore, CUDA streams offer efficient handling of multiple messages instead of lengthy messages. Instead of merging each message into a single stream and transmitting it to the GPU, or invoking and calculating a kernel function separately for each message, CUDA streams eliminate the need for message consolidation. With the application of CUDA streams, each stream can achieve optimal performance by processing individual message elements or a specific number of message elements per total number of streams. Our approach involves dividing a message into several blocks and computing the hash digest for each block using dedicated streams. Specifically, each stream in our implementation is designed to compute a hash digest for a message block.

This section presents the performance results of the SHA-3 operation. Additionally, a comparison of our implementations and other implementations is provided to highlight the performance differences. The performance measurements were conducted using the NVIDIA GTX 1070 and RTX 2080Ti architectures, while the CPU measurements were performed on the Intel(R) Core(TM) i7-10700K (3.60GHz) environment. The performance was measured using the maximum throughput, represented in GiGa-bytes per Second. Table 2 presents the specifications of the GPU architecture utilized in our paper. Prior to measuring performance and reviewing results in the GPU architecture, it is essential to comprehend the structure of the GPU architecture. Each GPU architecture varies in terms of the number of cores, core clock speed, and boost clock speed.

The GTX 1070 architecture boasts 1,920 cores, with a core clock speed of 1,506 MHz and a boost clock speed of 1,683 MHz. In comparison, the GTX 1080 architecture utilized by Dat et al. features 2,560 cores, a core clock speed of 1,607 MHz, and a boost clock speed of 1,733 MHz. The number of cores, core clock speed, and boost clock speed all influence the performance of GPU architectures, with the GTX 1080 architecture demonstrating greater efficiency than the GTX 1070 architecture. Conversely, the RTX 2080 Ti

architecture offers 4,352 cores, a core clock speed of 1,350 MHz, and a boost clock speed of 1,545 MHz. Although the core and boost clock speeds of the RTX 2080 Ti architecture are lower compared to other GPU architectures, the doubled number of cores enables it to process operations more effectively. Consequently, assuming a performance level of 1 for the GTX 1070, the performance efficiency of the GTX 1080 is 1.18 times that of the GTX 1070, while the RTX 2080 Ti achieves a performance efficiency of 1.75 times.

CONCLUSION:

The frequency of SHA-3 usage is increasing due to its incorporation into algorithms submitted to the PQC 3-round contest. Moreover, the adoption of SHA-3 is recommended as a defense against continuous collision pair attacks, which target existing hash functions like SHA-1 and SHA-2. In our research, we present an optimized parallel implementation method for SHA-3 on GPU systems. Specifically, our optimized implementation of SHA-3(256) on the GTX 1070 architecture achieved a maximum throughput of 59.87 Gb/s, while on the RTX 2080 Ti architecture, it reached a maximum throughput of 171.62 Gb/s.

In addition to these impressive results, our optimized implementation of SHA-3(512) in the GTX 1070 environment achieves a maximum throughput of 30.20 Gb/s. Moreover, when running on the RTX 2080 Ti environment, the maximum throughput reaches an outstanding 88.51 Gb/s. Additionally, by not utilizing CUDA stream, our SHA-3(512) software on the GTX1070 GPU exhibits a remarkable 49.73% improvement in throughput compared to the previous best work on the GTX1080, highlighting the superiority of our proposed optimization methods.

Our optimization implementation stands out as the fastest implementation of the GPU architecture, as far as our knowledge goes. Notably, our optimization method proves highly effective for PQC algorithms employing SHAKE and SHA-3 formats. Furthermore, our optimization method finds valuable applications in authentication processors and cryptographic algorithms based on hash functions.

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-DateExpanding-the-Computational-Power-of-its-IBM-Cloud-AccessibleQuantum-Computers

[16] IBM’s Roadmap for Scaling Quantum Technology. Accessed: Oct. 24, 2021. [Online]. Available: https://research.ibm.com/blog/ibmquantum-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 modulelattice-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/ntru20190330.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