Maximizing GPU Parallelism for a High-performance Cryptanalysis System

Sangyub Kim
Korea University
Seoul, South Korea
sangy_kim@korea.ac.kr

Youngjoo Shin
Korea University
Seoul, South Korea
syoungjoo@korea.ac.kr

***Abstract*—Modern cryptanalysis techniques utilize GPUs, owing to their parallel processing power, to significantly improve the speed of cryptanalysis tasks. Thus, recent research directions have focused on maximizing parallelism to complete large-scale cryptanalysis in a short time. For this, previous works utilized NVIDIA's technologies such as Hyper-Q, Multi-Process Service (MPS), and Multi-Instance GPU (MIG) for their cryptanalysis system. However, no effort has been made to evaluate whether integrating these technologies improves parallelism compared to a single one. In this paper, we design and implement a cryptanalysis system that integrates MPS and MIG. To evaluate this system, we develop three versions of Strassen's matrix multiplication programs for large-scale cryptanalysis. We evaluate the performance of these programs using CUDA, cuBLAS, and a combination of cuBLAS and MPI, respectively. Our results show that with both MPS and MIG enabled, the computation time for matrix multiplication is approximately $1.42\times$, $1.28\times$, and $2.02\times$ faster.**

***Index Terms*—Cryptanalysis System, Multi-Process Service, Multi-Instance GPU, Parallel Computing**

I. Introduction

Based on extensive research on various cryptographic algorithms, there is also research on cryptanalysis. This includes cryptanalysis using GPUs, which is a notable field.

Previous research efforts have focused on reducing the computation time for cryptanalysis tasks by utilizing GPUs. Cianfriglia et al. [2] proposed a novel framework for breaking encryption by finding linear polynomials in the key bits, called a cube attack [4], which utilizes GPUs to speed up the computation. Additionally, optimized cryptanalysis algorithms [15] have been developed by addressing some of the limitations of GPUs (e.g., limited number of registers available to each thread, slow access time to global memory) through CUDA programming.

Expanding on previous work, there is a need to reduce the time needed to process cryptanalysis operations that leverage large-scale data. Consequently, we present the need for a system capable of parallelizing large-scale data in a single GPU environment, i.e., when the system has limited GPU resources.

One of NVIDIA's latest parallelization technologies is Multi-Process Services (MPS) [10], which allows multiple GPU kernels belonging to different CUDA contexts to be managed as a single context. It allows concurrent processing of GPU kernels from different CUDA contexts on the same GPU, thereby increasing parallelism. Another is Multi-Instance GPU (MIG) [12], which partitions a single GPU to enable parallel processing of multiple GPU kernels.

In this paper, we propose a parallel processing environment that leverages both MPS and MIG. This environment partitions a single GPU into multiple instances, enabling efficient parallel processing of GPU operations. This configuration allows for a greater number of GPU tasks to be handled concurrently, thereby reducing overall processing time. Consequently, we design and implement a high-performance cryptanalysis system based on this environment.

To evaluate our system, we apply matrix multiplication acceleration techniques and assess its performance. For parallel computation, we develop three Strassen's matrix multiplication [14] programs using shared source code [7]. These programs were implemented using three different approaches: CUDA, CUDA Basic Linear Algebra Subroutine (cuBLAS) [11], and a combination of cuBLAS and Message Passing Interface (MPI) [13]. We measure computation time both with and without the proposed cryptanalysis system. The performance evaluation indicates improvements of approximately $1.42\times$, $1.28\times$, and $2.02\times$, respectively.

The main contributions of our study are summarized as follows:

- We design and implement a parallelism-maximizing system by integrating MPS and MIG on a single GPU system.
- We implement cryptanalysis programs, i.e., Strassen's matrix multiplication programs, using three methods, apply them to our system, and evaluate their performance.

The structure of this paper is as follows. Section II underscores related work to the current project and Section III describes some background of this paper. Section IV proposes the design of a cryptanalysis system using these technologies and Section V implements the proposed system. In Section VI, we present a performance evaluation of the system, and Section VII discusses the pros and cons of MPI technology. We finally conclude in Section VIII.

II. Related work

In this section, we discuss previous studies related to our work.

Cube attack with GPU. Cianfriglia et al. [2] presented a framework for a cube attack [4] that leverages GPU processing to enhance the efficiency of parallel search. The cube attack is a cryptanalysis technique applicable to various symmetric key algorithms. They successfully demonstrated this attack on two different well-known stream ciphers: Trivium [3] and Grain-128 [5].

In their study, they compared the execution times of an OpenMP-based parallel CPU version and a GPU-based version of the cube attack. The results of the performance evaluation highlighted that the GPU-based version performs faster, depending on the size of the free variables. In the case of Trivium, NVIDIA Kepler showed $80\times$ and NVIDIA Pascal $630\times$ speedups, while for Grain-128, Kepler indicated $154\times$ speedup, and Pascal showed $930\times$ speedup.

This study introduced the first GPU-optimized cube attack framework specifically designed to maximize parallelization. It demonstrated the advantages of using GPUs by emphasizing the speedup compared with the CPU-parallel benchmark, the performance dependence on system parameters, and a comparison between different GPU architectures.

While this attack simply leveraged the computational power of the GPU, our work increases parallelism by allowing the GPU's operations to process in parallel.

Brute force attack with GPU. Cihangir Tezcan [15] presented a GPU-based brute force cryptanalysis approach for lightweight cryptographic algorithms. Brute force attacks have become practical, and algorithms that use short keys have been abandoned, but there are still ISO/IEC standard ciphers that use short keys. He demonstrated an exhaustive key search attack on the bit-oriented block ciphers such as DES/3DES and ISO/IEC 18033-3 PRESENT.

Since most programming languages operate at the byte level rather than the bit level, bit-oriented algorithms like DES and PRESENT require specialized handling. For this, his experiments used a table-based CUDA implementation to bypass bit-level operations and avoid performance degradation. His results indicated a speedup of at least $1.22\times$ using NVIDIA MX 250 for PRESENT and showed that a DES key could be obtained in less than a year using an NVIDIA RTX 3070 GPU.

All GPUs have various architectural constraints that degrade performance, and the study provided an optimized CUDA implementation to resolve these limitations. They focused on optimizing the algorithm for these different architectures and resources through GPU programming.

In contrast to this algorithm-level optimization through GPU programming, our work focuses on establishing a parallel execution environment to enhance overall performance.

III. Background

A. Multi-Process Service

A single GPU operation can run across multiple GPU cores simultaneously, enhancing GPU utilization and significantly reducing idle time. However, these operations must exist within the same CUDA context, meaning that operations in different CUDA contexts cannot execute concurrently [1]. Each CUDA application has state information about the hardware resources it needs to operate, which is called the CUDA context. This restriction exists because GPU kernels in the same context can run in parallel through the concurrent scheduler, while those in different contexts must execute sequentially via the time-sliced scheduler [8].

Multi-Process Service (MPS) is a CUDA API designed to address this limitation and improve parallelism. It allows multiple GPU kernels belonging to different CUDA contexts to be managed as a single context, bypassing the time-sliced scheduling. This enables CUDA applications to share GPU resources more effectively, allowing the GPU kernels of each process to run concurrently. MPS consists of three components: an MPS control daemon process, an MPS server process, and an MPS client runtime. The control daemon coordinates the connection between the MPS client and the MPS server, with the server providing MPS services to the client. Also, environmental settings for the MPS server can be specified by setting MPS-related environment variables (e.g., `CUDA_MPS_PIPE_DIRECTORY`).

The MPS workflow consists of three steps. In step 1, users initiate an MPS control daemon process. In step 2, the control daemon launches an MPS server. In step 3, the MPS client submits its kernel to the execution environment managed by the MPS server.

B. Multi-Instance GPU

Multi-Instance GPU (MIG) allows a single GPU to be partitioned into up to seven independent GPU instances, enabling multiple CUDA applications to utilize distinct GPU instances concurrently. This physical isolation provided by MIG allows each application to interact with its assigned virtual GPU instance as if it were a physical GPU.

Each MIG instance includes an L2 cache, memory, and dedicated Streaming Multiprocessors (SMs). It ensures secure isolation to prevent one client's activities from negatively impacting other client's operations. As each instance operates with independent GPU resources, resource contention is effectively eliminated.

MIG provides flexibility in creating various partitions and configuring different combinations of GPU instances. It is noteworthy that the MIG is only supported on the latest NVIDIA GPUs (Ampere and later). For example, the NVIDIA A100 GPU can be decomposed into eight 5GB memory slices and seven SM slices. Users can create a 4g.20gb GPU Instance (GI) profile with four 5GB memory slices and four SM slices. NVIDIA also offers "GPU Instance Profiles" options allowing flexible and efficient resource management to respond to various workloads.

C. Strassen's algorithm

Cryptanalysis algorithms often rely on matrix operations, and one such example is the Hill cipher [6], a substitution cipher that utilizes matrix multiplication and inverse matrices

for message encryption and decryption. Hill ciphers are rooted in concepts from linear algebra, making them more mathematical compared to other ciphers. In the Hill cipher, encryption involves multiplying each message block by an invertible matrix modulo 26, and decryption requires multiplying each block by the inverse of the encryption matrix.

Therefore, various algorithms and studies have been developed to improve the computational complexity of matrix multiplication. In particular, Strassen's algorithm is more efficient than standard matrix multiplication for the following reasons. First, it has a lower computational complexity of $O(n^{2.81})$ compared to the $O(n^3)$ complexity of standard matrix multiplication. Second, it is well-suited for parallelism as it divides a large matrix into smaller submatrices and processes them recursively.

Strassen's algorithm achieves matrix multiplication by performing a small number of multiplications and a large number of additions and subtractions compared to standard matrix multiplication. It performs better for certain matrix sizes, which is achieved through the following process. First, users partition each matrix into four equal-sized submatrices,

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

where A and B represent the input matrices, and matrix C is the result of multiplying A and B. Users calculate seven intermediaries $M_1, M_2, M_3, M_4, M_5, M_6$ and M_7 from the partial matrices A and B and use them to determine the final matrix elements $C_{1,1}, C_{1,2}, C_{2,1}$ and $C_{2,2}$. The formulas for calculating the intermediaries and the final matrix elements are shown below.

$$\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) & C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} & C_{1,2} &= M_3 + M_5 \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) & C_{2,1} &= M_2 + M_4 \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) & C_{2,2} &= M_1 - M_2 + M_3 + M_6 \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} & & \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) & & \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) & &
\end{aligned}$$

IV. Design

This section describes the design of a high-performance cryptanalysis system that integrates MPS and MIG to maximize GPU parallelism. We design three distinct environments: one with solely MIG, another with only MPS, and a third with integrated MPS and MIG. Here, we assume the execution of a cryptanalysis program that involves splitting and aggregating, such as Strassen's matrix multiplication.

A. Parallel processing system with MIG

When MIG is enabled, applications can run concurrently on distinct GPU instances. MIG allows a single GPU to be divided into multiple MIG instances, each with its own memory and SMs. These instances are physically isolated, ensuring that activities on one instance do not impact other instances running on the same GPU. MIG not only enhances overall GPU utilization but also guarantees Quality of Service (QoS).

By leveraging MIG, we can create up to seven instances to allocate to various cryptanalysis applications. Utilizing separate GPU resources for each instance enables us to parallelize these applications, allowing multiple operations to be processed simultaneously and efficiently without interference.

B. Parallel processing system with MPS

We design an environment where MPS is enabled. This configuration allows the CUDA-based cryptanalysis program to leverage MPS services and be scheduled within the same context for concurrent execution.

When we execute an MPS client in an environment with the MPS control daemon, the daemon activates and assigns an MPS server. The MPS server runs under the same user ID as the requesting user. Once the MPS server is running, the MPS service can be provided to our application whenever we execute the CUDA application.

In this configuration, the cryptanalysis program is divided into multiple GPU kernels and executed sequentially. These GPU kernels are managed as a single CUDA context using MPS, even though they belong to different CUDA contexts. As a result, they can be efficiently scheduled by a concurrent scheduler rather than a time-sliced scheduler, enabling parallel execution.

C. Parallel processing system integrated MPS and MIG

We design an environment that integrates MPS and MIG to maximize GPU parallelism. In this design, multiple MPS servers can operate on a single GPU. While a typical single GPU setup involves only one MPS server, our design allows for the operation of multiple servers. Each server can run on a different MIG instance, and MPS clients utilizing the same GPU resources (i.e., MIG instance) as the MPS server can receive MPS services.

In our system, a single GPU is partitioned into n MIG instances, as described in Section IV-A. Each MIG instance functions as an independent GPU, allowing multiple operations to be performed within each instance. An MPS control daemon can be launched on each MIG instance, enabling the possibility of running a maximum of seven MPS servers on a single GPU. CUDA applications using an MIG instance can then receive MPS services from the MPS server within that instance. The environment for each MPS instance follows the same as in Section IV-B.

Consequently, this integrated environment leverages both technologies to increase the maximum number of clients. Each client can utilize the MPS service to process operations in parallel. The subsequent section details the process of implementing this environment based on our design.

V. Implementation

This section details the implementation of the proposed high-performance cryptanalysis system that integrates MPS

and MIG. The implementation is based on the three environments described in the previous section. In this setup, we assume the use of a single NVIDIA A100 GPGPU.

A. Cryptanalysis system with MIG

We utilize MIG to split a single A100 GPGPU into four MIG instances, each of type 1g.20gb, combining a 20GB memory slice with one SM slice. Given that the A100 GPU contains 108 SMs, each 1g.20gb instance can utilize 15 SMs.

To execute a CUDA application with specific GPU resources, we use the `CUDA_VISIBLE_DEVICES` environment variable. First, we identify the identifier (ID) of each MIG instance using the NVIDIA System Management Interface (`nvidia-smi`) [9] command. We then set the environment variable to the ID of the desired MIG instance and execute the cryptanalysis program. It allows us to control which GPU resources the cryptanalysis program will use.

B. Cryptanalysis system with MPS

To initiate the MPS service environment, we start by launching the MPS control daemon. Upon the first connection attempt by an MPS client, the control daemon triggers the launch of the MPS server. The MPS client subsequently submits the CUDA context of each GPU kernel to the execution environment managed by the MPS server. The execution environment effectively manages these contexts within a single shared context known as the MPS CUDA context. This setup allows GPU kernels from different CUDA contexts to run in parallel.

The MPS control daemon is launched using a straightforward command `nvidia-cuda-mps-control`, and can be started either in the background or foreground. The control daemon does not immediately start the MPS server. When an MPS client, such as the CUDA driver, attempts to establish a connection with the daemon, the daemon starts the MPS server. From then on, with the MPS server activated, the MPS client communicates with the server to receive MPS services.

Communication between the MPS control daemon, MPS server, and MPS clients is facilitated through named pipes and UNIX domain sockets, utilizing the `/tmp/nvidia-mps` directory by default. Once this environment is constructed, the MPS service is ready for use with any GPU kernel.

C. Cryptanalysis system integrated MPS and MIG

We construct Section V-A and apply the procedures described in Section V-B for each instance to implement an environment integrating MPS and MIG. To achieve this integration, additional environment variables must be configured. Specifically, the MPS-related environment variables mentioned in Section III-A need to be set, allowing for the execution of multiple MPS servers and facilitating communication between each server and client.

We create four 1g.20gb instances using the same single A100 GPGPU as in Section V. For each instance, we follow the procedures described in Section V-B and configure two key environment variables: `CUDA_MPS_PIPE_DIRECTORY` and `CUDA_VISIBLE_DEVICES`. The former specifies the communication directory for each instance, while the latter designates the ID of each MIG instance.

Before launching the MPS control daemon, we set the `CUDA_MPS_PIPE_DIRECTORY` and `CUDA_VISIBLE_DEVICES` environment variables for each of the four instances. This configuration ensures that MPS servers and MPS clients communicate exclusively through specific directories, preventing any sharing of MPS sockets. This setup enables the MPS server to operate independently on each instance.

To verify that each program is functioning correctly on its designated GPU instance, two methods are employed. First, we inspect the log output from the MPS control daemon to identify any error messages that might indicate the use of an incorrect GPU instance. Second, we confirm that the MPS server is running. It's important to note that the MPS control daemon does not immediately create an MPS server upon startup; it generates an MPS server when it receives a connection request from an MPS client. If the MPS server is inactive, it may indicate that no operation has been requested for that specific GPU instance.

VI. Evaluation

To evaluate the design and implementation of the proposed high-performance cryptanalysis system, we utilized three of Strassen's matrix multiplication programs. The first program implemented Strassen's matrix multiplication algorithm in the CUDA for GPU acceleration. The second program utilized the matrix multiplication function of the cuBLAS API to implement Strassen's algorithm. The third program extends the second approach by incorporating MPI to enable parallel operations.

We first describe the experimental environment setup, followed by an evaluation of the system's performance across the three Strassen matrix multiplication programs.

A. Experimental setup

The hardware resources and software configuration used to set up the high-performance cryptanalysis system are described in Table I. We employed an Intel Xeon CPU to establish a stable system environment and selected an A100 GPGPU to leverage both MPS and MIG. Specifically, we utilize MIG to build an experimental environment by partitioning one A100 GPU into four virtual GPU instances. Each MIG instance (1g.20gb) is allocated one SM slice and 20GB of memory.

TABLE I: Experimental setup for configuring system.

Configuration	Detail
CPU	Intel Xeon Silver 4210
GPU	NVIDIA Ampere 100 GPU 80G
	MIG 1g.20gb (4 instances)
Operation System	Ubuntu 20.04.6 LTS
Kernel	5.15.0-78-generic

In addition, Table II lists the drivers and applications installed to enable parallelism in the system. First, a dedicated drive for the GPU device was installed to ensure proper recognition by the operating system. To enable parallel programming using the CUDA language, we installed a CUDA driver compatible with the GPU device. Specifically, for the utilization of the cuBLAS API, we need to install CUDA 11.0 or later. Finally, we prepared the environment for parallelism by installing the OpenMPI compiler to support MPI-based parallel operations.

TABLE II: List of installed applications and drivers.

Type	Device or Technology		Detail
Driver	Device	GPU	nvidia-driver 535.54.03
	Tech.	CUDA	cuda-driver-12.2
Application	Tech.	MPI	openmpi-4.1.5
		cuBLAS	cuBLAS API 12.02.01
		Compiler	gcc 9.4.0
			nvcc 12.2.91

To evaluate the performance of the designed system, we utilize a case study focused on matrix multiplication acceleration. As mentioned in Section III-C, numerous matrix multiplication algorithms have been studied to enhance the computational complexity. Among them, we employed Strassen's algorithm to further enhance the parallelism within our system.

Strassen's algorithm, based on the divide-and-conquer approach, divides the input matrix into smaller submatrices and aggregates the results of these partial matrices to obtain the final output matrix. The key advantage of using Strassen's algorithm in this study is its capability to perform intermediate computations in parallel. The division and aggregation process is highly suitable for the proposed system, as it task partitioning, parallel execution of subtasks, and reduced memory consumption.

Additionally, we developed a bash script to execute the cryptanalysis program (i.e., Strassen's matrix multiplication program) on the MIG instances. This script takes two large-scale matrices as input and divides each matrix into four partial matrices. Then it provides them as input to each program and runs each of these four programs on different MIG instances so that the operations can be processed in parallel. At the end of each program's execution, it combines the results from each program to create a final result matrix.

This system effectively divides a large matrix into four smaller partial matrices, performing all calculations in parallel. For instance, it can concurrently execute four multiplications of 8×8 matrices to compute the multiplication of two 16×16 matrices.

We denote the environment with both MPS and MIG enabled as ESEG (Section V-C) and the environment with both MPS and MIG disabled as DSDG. In the ESEG setup, a single GPU is split into four GPU instances (1g.20gb), with each of the four MPS servers running on a different GPU instance. Additionally, the environment with only MIG enabled is DSEG (Section V-A), and with only MPS enabled, that is ESDG (Section V-B).

B. CUDA

We first evaluate a matrix multiplication program implemented in the CUDA, as illustrated in Fig. 1a. The matrix size ranged from 64(2^6) to 8,192(2^{13}), with matrix elements represented as float data types.

For a matrix of size 8,192, the computation takes approximately 2287.9 ms in the DSDG environment. However, in the ESEG environment, the time is reduced to around 1608.9 ms, indicating a performance improvement of approximately $1.42\times$ when both technologies are integrated.

Furthermore, the result of ESDG shows an improvement of approximately $1.24\times$, while the result of DSEG improves by approximately $1.11\times$. These results highlight the potential for performance enhancements by enabling MPS and MIG individually.

C. cuBLAS API

To achieve faster computation, we utilized the cuBLAS API, which provides optimized matrix multiplication functions, to implement Strassen's matrix multiplication program. The performance results are shown in Fig. 1b. The matrix size and element data type were consistent with those used in the previous experiment.

A comparison of Fig. 1a and Fig. 1b reveals the faster overall computation time achieved with the cuBLAS API. For a matrix size of 8,192, the program implemented with the cuBLAS API is approximately $1.57\times$ faster than CUDA alone.

We also assessed the effects of the MPS and MIG on this program's performance and found optimal results in the ESEG environment, with a performance improvement of approximately $1.28\times$ compared to the DSDG environment.

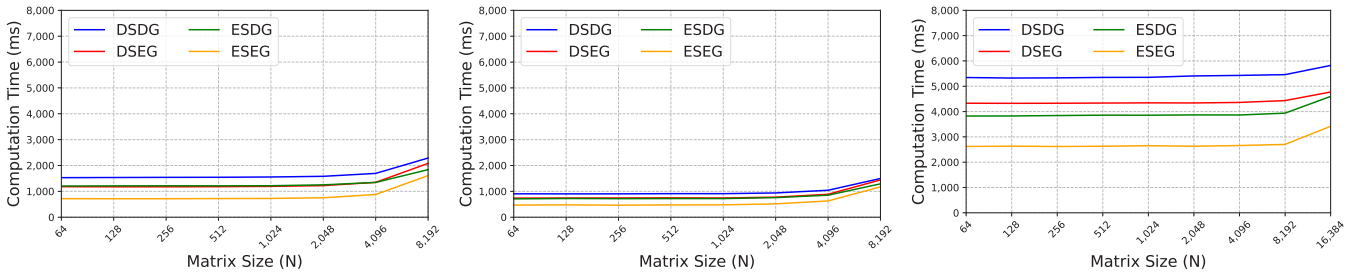
D. cuBLAS API and MPI

The third program builds on the implementation described in Section VI-C, with the addition of MPI for distributed processing. In this setup, the master node distributes tasks to worker nodes, which then process these tasks in parallel using MPI. This allows the input matrix to be divided into smaller submatrices, enabling parallel calculations on these smaller matrices.

Fig. 1c illustrates the performance of this program. The element data type is the same as in the previous experiment, while the matrix size ranges from 64(2^6) to 16,384(2^{14}).

As shown in the figure, the ESEG environment achieves the highest performance, showing reduced matrix multiplication times compared to the other cases. For a matrix size of 8,192, enabling both MPS and MIG results in a performance improvement of approximately $2.02\times$ compared to using no parallelization technologies. Similarly, for a matrix of size 16,384, the performance improvement is about $1.71\times$ with both technologies enabled.

In comparing ESDG and DSEG, we observe that enabling only MPS yields better performance than enabling only MIG.



(a) CUDA (b) cuBLAS (c) cuBLAS and MPI

Fig. 1: Performance of parallel cryptanalysis system for Strassen's matrix multiplication.

This difference is likely due to MIG providing some parallelism while scheduling for other CUDA contexts is managed by the concurrent execution scheduler.

VII. Discussion

We assert that a primary advantage of an MPI-based distributed system is its capacity for significant memory savings.

A comprehensive analysis of the results in Fig. 1 indicates that matrix multiplication programs implemented with the cuBLAS API and MPI achieve an overall slower computation speed. This is primarily due to the overhead introduced by the MPI-based distributed system, which inherently involves numerous parallel processes and overhead. Furthermore, there is a scheduling latency associated with receiving MPS services, as MPI increases the number of nodes.

However, it's worth noting that without an MPI-based distributed system, matrix multiplication calculations for a matrix size of 16,384(2^{14}) cannot be performed. The reason for this is that the OS kills the process due to memory pressure, which we confirmed experimentally. Consequently, we expect that the proposed MPI-based distributed system will enable matrix multiplications on more large-scale data which is a notable contribution.

VIII. Conclusion

In this paper, we presented the design and implementation of a high-performance cryptanalysis system to accelerate large-scale data processing. This system integrates two key technologies, MPS and MIG. MPS allows GPU kernels from different contexts to execute concurrently by sharing GPU resources, while MIG partitions a single GPU into multiple instances to facilitate parallel execution of programs. Our case study on matrix multiplication acceleration demonstrated that the proposed system achieved notable performance improvements, with up to a 2.02× increase. We expect that our work has wide application in various cryptanalysis technologies, offering promising performance enhancements.

Acknowledgment

This research was supported by a National Research Foundation of Korea (NRF) grant, funded by the Korean government (MSIT) (No.2023R1A2C2006862).

References

[1] T. Bradley, "Hyper-q example," *NVidia Corporation. Whitepaper v1. 0*, 2012.

[2] M. Cianfriglia, S. Guarino, M. Bernaschi, F. Lombardi, and M. Pedicini, "A novel gpu-based implementation of the cube attack: Preliminary results against trivium," in *Applied Cryptography and Network Security: 15th International Conference, ACNS 2017, Kanazawa, Japan, July 10-12, 2017, Proceedings 15*. Springer, 2017, pp. 184–207.

[3] C. De Canniere, "Trivium: A stream cipher construction inspired by block cipher design principles," in *International Conference on Information Security*. Springer, 2006, pp. 171–186.

[4] I. Dinur and A. Shamir, "Cube attacks on tweakable black box polynomials," in *Advances in Cryptology-EUROCRYPT 2009: 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings 28*. Springer, 2009, pp. 278–299.

[5] M. Hell, T. Johansson, A. Maximov, and W. Meier, "A stream cipher proposal: Grain-128," in *2006 IEEE International Symposium on Information Theory*. IEEE, 2006, pp. 1614–1618.

[6] L. S. Hill, "Cryptography in an algebraic alphabet," *The American Mathematical Monthly*, vol. 36, no. 6, pp. 306–312, 1929.

[7] ijleesw, "matmul-omp-cuda," 2018. [Online]. Available: https://github.com/ijleesw/matmul-omp-cuda

[8] S. Kim, J. Oh, and Y. Kim, "An execution performance analysis of applications using multi-process service over gpu," *Korea Network Operations and Management Review Volume 19 Number 1*, p. 60, 2019.

[9] NVIDIA, "NVIDIA System Management Interface program," 2016. [Online]. Available: https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf

[10] ——, "Multi-Process Service," 2022. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[11] ——, "cuBLAS Release 12.3," 2023. [Online]. Available: https://docs.nvidia.com/cuda/cublas/

[12] ——, "Multi-Instance GPU User Guide," 2023. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html

[13] OpenMPI, "Open MPI: Open Source High Performance Computing," 2023. [Online]. Available: https://www.open-mpi.org/

[14] V. Strassen *et al.*, "Gaussian elimination is not optimal," *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[15] C. Tezcan, "Key lengths revisited: Gpu-based brute force cryptanalysis of des, 3des, and present," *Journal of Systems Architecture*, vol. 124, p. 102402, 2022.