Two Algorithms for Fast GPU Implementation of NTT

Ali Şah Özcan
Faculty of Engineering and Natural Sciences
Sabanci University
Istanbul, Turkey
Email: alisah@sabanciuniv.edu

Erkay Savaş
Faculty of Engineering and Natural Sciences
Sabanci University
Istanbul, Turkey
Email: erkays@sabanciuniv.edu

***Abstract*—The number theoretic transform (NTT) permits a very efficient method to perform multiplication of very large degree polynomials, which is the most time-consuming operation in fully homomorphic encryption (FHE) schemes and a class of non-interactive succinct zero-knowledge proof systems such as zk-SNARK. Efficient modular arithmetic plays an important role in the performance of NTT, and therefore it is studied extensively. The access pattern to the memory, on the other hand, may play much greater role, as the NTT execution time is mostly memory-bound due to large degree polynomials. In this paper, we propose two algorithms for fast computation of NTT on a class of graphical processing units (GPU) by optimizing the memory access patterns. We present an approach i) to optimize the number of accesses to slow global memory for thread synchronization, and ii) to make better use of spatial locality in global memory accesses. It turns out that by controlling certain parameters in CUDA platform for general-purpose GPU computing (GPGPU) such as kernel count, block size and block shape, we can affect the performance of NTT. To best of our knowledge, this work is unique for it suggests a recipe for selecting optimum CUDA parameters to obtain the best NTT performance for a given polynomial degree. Our implementation results on various GPU devices for all power-of-two polynomial degrees from 2^{12} to 2^{28} show that our algorithms compare favorably with the other state-of-the-art GPU implementations in the literature with the optimum selection of these three CUDA parameters.**

***Index Terms*—Number Theoretic Transform, GPU, Homomorphic Encryption, zk-SNARK, Hardware Acceleration.**

I. Introduction

Advanced cryptographic schemes such as homomorphic encryption [1] and zero-knowledge proofs [2] are staple building blocks in many privacy-critical applications including electronic voting [3], privacy-preserving machine learning [4], privacy-preserving cryptocurrency [5], smart contracts [6], and many others. One can even safely say the promise of a dependable and fair digital age cannot come to pass without them [7]–[9]. As a result, the research community invested decades of intense effort and energy to supply *practicable* solutions [10]–[15], as their intended functionality can only be achieved with prohibitively high computation costs [16]. Although an outstanding progress have been made on efficient implementation of those schemes [17]–[19], there is still pressing need for further improvement thereof to match up the latency and throughput requirements of real-life applications [20].

The multiplication of very large degree polynomials (e.g., the multiplication in polynomial rings $\mathcal{R} = \mathbb{Z}[x]/\Phi_n(x)$, where $\Phi_n(x)$ is the cyclotomic polynomial of degree n), is (one of) the most time and resource consuming operation in both homomorphic encryption and modern zero-knowledge proof schemes such as zk-SNARK [10], [11]. One widely adopted method for fast and memory efficient polynomial multiplication is based on number theoretic transform (NTT) [21], where fast implementation can be achieved via hardware acceleration on GPU [22]–[28] and FPGA devices [29]–[34]. Besides academia, industry is also keenly interested in the acceleration of the cryptographic primitives and organize the competitions to promote interest therein[1].

New generation FPGA devices, with extra high bandwidth memory communication and relatively low-energy consumption, stand one of the best candidates for acceleration. GPU devices, on the other hand, can also be profitably utilized in acceleration of many cryptographic primitives due to their extraordinary computational resources, easy integration with software libraries, and superior general-purpose computing capabilities.

Current GPU devices consist of thousands of parallel running threads and high bandwidth memory hierarchy featuring on-chip and off-chip memory. Computations intended to run on GPU are performed by invoking GPU kernel functions, and applications running on a host CPU device can call many kernels to offload some of its computation to GPU. Invoking each kernel function incurs an overhead in execution time due to the fact that access to off-chip memory is required for thread synchronization. The threads are grouped into blocks, whose size and shape are configurable and the threads in the same group enjoy faster synchronization. Various factors such as the number of kernels, block size, and block shape may have major impact on the performance of the application. Therefore, developing efficient GPU applications necessitates novel algorithm design and systematic approach in addition to code optimization specific to GPU architectures.

[1] https://www.zprize.io/

In this paper, we aim to develop and implement efficient algorithms to compute NTT for high degree polynomial multiplication on a class of GPU devices. We can summarize our contributions as follows.

- We propose two algorithms for efficient computation of NTT designed to fully exploit the outstanding parallel computing capabilities of GPU with carefully optimized memory access patterns. One of the algorithms is a form of well-known recursive algorithm while the other based on the Four-Step algorithm. We aim two degree ranges for the polynomials: i) $n \in [2^{12}, 2^{16}]$ intended for homomorphic encryption applications, and ii) $n \in [2^{20}, 2^{28}]$ for the zk-SNARK protocol. The algorithms are flexible and parametric as they can easily be adapted to work with any value of n provided that it is a power of two.
- We show that the performance of the algorithms is highly dependent on the selection of parameters such as the number of kernels, block size and block shape. We, then, propose a systematic approach to find out their optimum selection for the fastest implementation given a polynomial degree. The approach helps determine the selection of a specific parameter by considering the interplay of several parameters to improve computational aspects of an implementation such as fast access to global memory.
- We implement both algorithms on various GPU devices using all possible optimization techniques and present our implementation results for time efficiency. We provide both latency and throughput results, which suggest the performance of each algorithm varies depending on the ring dimension as well as the specific GPU device. Also, the timing results confirm that our algorithms compare favorably with other state-of-the-art algorithms for GPU in the literature.

The remaining of the paper is organized as follows. Section II provides the mathematical background of Number Theoretic Transform and presents the so-called Merge and 4-Step NTT algorithms for its efficient computation. Section III reviews the GPU architecture and its working principles and points outs common practices to use GPU devices efficiently while Section IV briefly explains the notation used throughout the paper. Section V presents two different NTT algorithms customized for efficient GPU implementation. Section VI presents the implementation results and compares them with those of the state-of-the-art implementations in the literature. The paper is concluded in Section VII with final remarks capturing the achievements and contribution.

II. Preliminaries

The number theoretic transform (NTT) is a form of Discrete Fourier Transform (DFT) defined over the ring of integers $\mathbb{Z}_q$. In cryptography, it is commonly used for multiplication of high degree polynomials as it reduces quadratic complexity of the schoolbook multiplication to $O(n \log n)$. The coefficients of an $(n-1)$-degree polynomial can be thought as a vector of integers, $\boldsymbol{a} = [a_0, a_1, \ldots, a_{n-1}]$, which can be transformed to another vector $\bar{\boldsymbol{a}} = [\bar{a}_0, \bar{a}_1, \ldots, \bar{a}_{n-1}]$ using NTT. The definition of the m-point NTT can be given as

$$\bar{a}_i = \sum_{j=0}^{n-1} a_j \omega^{i \times j} \mod q \text{ for } i = 0, 1, \ldots, m \qquad (1)$$

where $m \geq n$. For NTT to be defined, we need the existence of a constant value value $\omega \in \mathbb{Z}_q$., which can have two types

- $\omega \in \mathbb{Z}_q$: the primitive n-th root of unity in $\mathbb{Z}_q$, which satisfies the conditions $\omega^n \equiv 1 \pmod q$ and $\omega^i \neq 1 \pmod q$ $\forall i < n$, where $q \equiv 1 \pmod n$.
- ψ, where $\psi \in \mathbb{Z}_q$: the primitive $2n$-th root of unity, which satisfies the conditions $\psi^{2n} \equiv 1 \pmod q$ and $\psi^i \neq 1 \pmod q$ $\forall i < 2n$, where $q \equiv 1 \pmod{2n}$. Note that $\omega = \psi^2 \bmod q$ and $\psi^n \bmod q = -1$.

The polynomial ring $\mathcal{R}_q = \mathbb{Z}_q / \Phi(x)$ consists of polynomials of degree at most $n-1$, where $\Phi(x)$ is the cyclotomic polynomial of degree n. Multiplication in $\mathcal{R}_q$ requires polynomial multiplication followed by division with $\Phi(x)$. The multiplication can be performed using NTT, which takes the coefficient vectors of two polynomials, $\boldsymbol{a}$ and $\boldsymbol{b}$, and computes $\bar{\boldsymbol{a}} = NTT(\boldsymbol{a})$ and $\bar{\boldsymbol{b}} = NTT(\boldsymbol{b})$. We can perform the component-wise multiplication afterward as $\bar{c}_i = \bar{a}_i \cdot \bar{b}_i \bmod q$, for $i = 0, 1, \ldots, m$. Then, we apply the inverse NTT operation on the resulting vector $\bar{\boldsymbol{c}}$ to compute $c \in \mathcal{R}_q$. Generally, a final division operation by $\Phi(x)$ is applied to obtain the final result. When n is a power of two, we have $\Phi(x) = x^n + 1$, and due to negacyclic convolution, only n-point NTT operations are performed without the final reduction by $\Phi(x) = x^n + 1$.

A. Two Algorithms for Number Theoretic Transform (NTT)

In this section, we explain two different algorithms to compute number theoretic transform (NTT). For brevity, we only include algorithms for forward NTT while the details of the inverse NTT can be found in Appendix A as well as in the literature [35].

1) Merged NTT Algorithm

The first algorithm (Algorithm 1) is the classical iterative method that processes the input vector elements in $\log_2 n$ outer loop iterations, which have to be executed sequentially (see **do-while** loop in Algorithm 1). In each outer iterations, there are $n/2$ *butterfly* operations in every outer loop iterations. Depending on the NTT algorithm, either Cooley-Tukey (CT) (Algorithm 2) or Gentleman-Sande (GS) (Algorithm 6 in Appendix A) are used for butterfly operation. Simply speaking, a butterfly operation reads two elements of the vector, process them to compute two integers and writes the results back to their place in the vector. The access patterns for read and write operations change in every outer iterations.

Algorithm 1, used to compute forward NTT operation, is referred as **Merge-NTT** as there is neither pre- nor post-processing operations and the bit reverse operation is deferred. Different powers of the primitive root of unity (ψ) are used in butterfly operations depending on the iteration and part of the vector. The algorithm pre-computes and stores them in the array, Ψ_{br} in bit-reversed order. The inverse NTT operation

Algorithm 1 Merge Forward NTT (**Merge-NTT**)

Input: $a(x) \in \mathbb{Z}_q[x]/(x^n+1)$ polynomial standard-order
Input: $\Psi_{br}[k] = \psi^{br(k)} \pmod q$ **for** $0 < k \leq n-1$ (Powers of ψ stored in bit-reverse order)
Input: $n = 2^l$, q ($q \equiv 1 \mod 2n$)
Output: $\boldsymbol{a} \leftarrow NTT(\boldsymbol{a})$ in bit-reversed order

1: $t \leftarrow n$; $m \leftarrow 1$
2: **do**
3: $t \leftarrow t/2$
4: **for** i from 0 by 1 to m **do**
5: $j_1 \leftarrow 2it$
6: $j_2 \leftarrow j_1 + t - 1$
7: **for** j from j_1 **by** 1 **to** $j_2 + 1$ **do**
8: $a_j, a_{j+t} \leftarrow \mathbf{CT}(a_j, a_{j+t}, \Psi_{br}[m+i], q)$
9: **end for**
10: **end for**
11: $m \leftarrow 2 \times m$
12: **while** $m < n$
13: **return** a

Algorithm 2 Cooley-Tukey Butterfly (**CT**)

Input: U, V, Ψ, q
Output: $\bar{U}$, $\bar{V}$

1: $\bar{U} \leftarrow U + (V \times \Psi) \pmod q$
2: $\bar{V} \leftarrow U - (V \times \Psi) \pmod q$
3: **return** $\bar{U}, \bar{V}$

(Algorithm 8 in Appendix A) can roughly be considered as the NTT algorithm being executed backward, which uses the GS butterfly operation.

2) Four-Step NTT Algorithm

The four-step method [36], described in Algorithm 3, is another way to compute NTT (and inverse NTT) operations. In the first stage, it arranges the input vector into a two-dimensional matrix of n_1-by-n_2, where $n = n_1 \times n_2$ (Steps 1-5 of Algorithm 3). In the second stage, it performs n_2 NTT operations of n_1-point each (Steps 7-9). In the third stage, the matrix elements are multiplied with certain powers of the primitive root of unity, ψ (Steps 11-15). And finally, the algorithm performs n_1 NTT operations of n_2-point each (Steps 16-18). The **4-Step NTT** algorithm computes much smaller (n_1-point and n_2-point against n-point NTT operations) and independent NTT computations, which exploits the parallel processing and locality of memory access much better than the **Merge NTT** algorithm. Nevertheless, the transpose operations (Steps 6, 10, and 19) can be challenging as they can cause a very fragmented memory access, which can be very disruptive for threads in a GPU device. The version of the **4-Step NTT** algorithm for computing inverse NTT operation is given in Algorithm 7 in Appendix A.

Algorithm 3 Four-Step NTT (**4Step-NTT**)

Input: $n_1, n_2 \leq n$ and $n_1 \times n_2 = n$
Input: $a(x) \in \mathbb{Z}_q[x]/(x^n-1)$ in polynomial standard-order
Input: $\Omega[k] = \Omega^{br(j)\times i} \pmod q$ **for** $0 < k \leq n-1$, **for** $0 < j \leq n_1 - 1$ **for** $0 < i \leq n_2 - 1$
Input: $\Omega_{0_{br}}[k] = \omega_0^{br(k)} \pmod q$ **where** $\omega_0 = \Omega^{(n/n_1)} \pmod q$, **for** $0 < k \leq n_1 - 1$
Input: $\Omega_{1_{br}}[k] = \omega_1^{br(k)} \pmod q$ **where** $\omega_1 = \Omega^{(n/n_2)} \pmod q$, **for** $0 < k \leq n_2 - 1$
Output: $\boldsymbol{a} \leftarrow NTT(\boldsymbol{a})$ in bit-reversed order

1: **for** i from 0 by 1 to n_1 **do** ▷ 1) Vector to matrix
2: **for** j from 0 by 1 to n_2 **do**
3: $B_{i,j} \leftarrow a_{i \times n_2 + j}$
4: **end for**
5: **end for**
6: $B = B^T$ ▷ Transpose operation
7: **for** j from 0 by 1 to n_2 **do** ▷ 2) n_2, n_1-point NTTs
8: $B_j \leftarrow \mathbf{NTT}(B_j, \Omega_{0_{br}}, n_1, q)$
9: **end for**
10: $B = B^T$ ▷ Transpose operation
11: **for** i from 0 by 1 to n_1 **do** ▷ 3) Correction step
12: **for** j from 0 by 1 to n_2 **do**
13: $B_{i,j} \leftarrow B_{i,j} \times \Omega_{i \times n_2 + j} \pmod q$
14: **end for**
15: **end for**
16: **for** i from 0 by 1 to n_1 **do**
17: $B_i \leftarrow \mathbf{NTT}(B_i, \Omega_{1_{br}}, n_2, q)$ ▷ 4) n_1, n_2-point NTTs
18: **end for**
19: $B = B^T$ ▷ Transpose operation
20: **for** j from 0 by 1 to n_2 **do** ▷ Matrix to vector
21: **for** i from 0 by 1 to n_1 **do**
22: $a_{j \times n_1 + i} \leftarrow B_{j,i}$
23: **end for**
24: **end for**
25: **return** A

III. GPU Architecture

GPU is a computing device that facilitates exceptional parallel processing capability due to that fact that it supports extremely high number of threads. Therefore, its instruction throughput far exceeds the one that can possibly be sustained by a conventional general-purpose CPU, which features comparably modest number of threads. As CPU and GPU are designed for different applications with different design principles, quantity, speed and computational power of CPU and GPU threads are also different. A CPU sustains fewer number of threads (in the order of tens) that can complete computationally more involved operations faster while recent GPU devices can support as high as 15K threads. But the GPU threads are computationally less capable running at slightly slower clock speeds. Therefore, the performance comparison of GPU and CPU can be involved and depends on benchmarks as pointed out in [37] and advantage of GPU over CPU is overestimated at times.

Configurable hardware platforms such as FPGA, which can offer superior parallelization with better energy efficiency than GPU. However, being easier to program and integrate with applications running on CPU and containing more on-chip and off-chip memory, GPU can be a strong alternative to accelerate

memory-bound applications such as advanced cryptographic operations, which heavily rely on arithmetic on extremely large mathematical objects; e.g. polynomial rings of very high dimensions.

CUDA-enabled GPU is a parallel computing device that can be used for general-purpose programming via CUDA®, which is a general purpose parallel computing platform and programming model. The CUDA software environment allows developers to use high-level programming languages such as C++ and Fortran.

In conclusion, GPUs are powerful devices, which proved to be accessible and easily programmable accelerators for a wide range of applications. However, it is essential to acquire deep insight into its micro-architectural details and to design algorithms to harness their computational resources.

A. High-Level Architecture of GPU

From a very high level, we can consider that a GPU platform consists of two main parts: i) GPU chip, and ii) off-chip memory. The GPU chip contains the main computation units known as streaming multiprocessors (SM) and on-chip memory that implements registers and shared memory. The local, global, constant, and texture memory types are all implemented in off-chip memory, for which the GDDR is used; a memory technology offering higher bandwidth and more power-efficient communication when compared with the DDR technology used in CPU memory systems. Nevertheless, the off-chip memory is still much slower than registers and on-chip shared memory.

A GPU contains an array of Streaming Multiprocessors (SM), which create, manage, schedule and execute threads on its functional units. An SM contains L1 cache (which is used to implement shared memory) and registers that are accessible to the threads scheduled to run on the SM. Execution happens in groups of 32 parallel threads called *warps*, in which threads start together at the same program address, but they can execute independently as they maintain separate states. An SM partitions threads into warps and its warps schedulers schedule them for execution on functional units of SM.

Another programming abstraction is thread block (or simply *block*), which can contain more than one warp. For instance, a block can contain as many as 32 warps or 1024 threads in current GPU devices. For easy indexing of threads, a block can be defined one, two or three dimensional. All threads in a block are scheduled to run on the same SM (accessing the same shared memory), which is capable of executing more than one block.

A *grid* is formed by combining multiple blocks, each of which contains the same number of threads. As the number of threads in a block is limited, grids are used to run larger number of threads in parallel than that can fit in a single block. Namely, different blocks in a grid may run in different SMs, and therefore, threads in different blocks do not use the same shared memory. Similar to blocks, a grid can be indexed one, two or three dimensional. We refer a particular configuration of block or grid as its *shape*.

Kernel is a function that is executed on a GPU device, which takes also the number of threads and blocks as arguments. As more than one block is used to execute a kernel, when threads from different blocks have to synchronize, this can be done by terminating the current kernel and start another if the computation is expected to continue. Starting and terminating kernels incur significant timing overhead, therefore an algorithm that limits the thread synchronization within a block is more efficient.

The *compute capability* of a GPU device, which determines hardware features and/or instructions available, is given by a version number and should be known to algorithm designers for developing better algorithms and their efficient implementations on GPU devices. The compute capability of a GPU device determines some pertinent information to our work such as the maximum number of blocks, warps, threads, number of 32-bit registers, maximum amount of shared memory per SM. In this paper, we use GPU devices with the compute capability 8.6 and some of its relevant features are listed in Table I.

TABLE I
CONFIGURATION OF COMPUTE CAPABILITY 8.6

Maximum number of blocks per SM	16
Maximum number of warps per SM	48
Maximum number of threads per SM	1536
Number of 32-bit registers per SM	64 K
Maximum number of 32-bit registers per thread	255
Maximum amount of shared memory per SM	100 KB

B. GPU Memory Hierarchy

There are different types of memory in the GPU, and the access pattern to these memory types plays a very important role in terms of memory latency performance. Each memory type has its advantages and disadvantages and can be used for different purposes accordingly. Table II lists these memory types and their specifications.

TABLE II
THE VARIOUS PRINCIPAL TRAITS OF THE MEMORY TYPES

Memory Types	Scope	Life Time	Access Latency
Register	1 Thread	Kernel	$1\times$
Shared	All threads in block	Kernel	$1\times$
Local	1 Thread	Kernel	$\approx 100\times$
Global	All threads + host	Application	$\approx 100\times$

In the hierarchical structure of GPU memory system, the *global memory* is in the highest level implemented in GDDR (off-chip memory), and therefore, its size is larger than any other GPU memory type. Its access latency is slower than other memory types, because of the overhead of accessing the off-chip memory. Additionally, since the global memory allocations persist for the lifetime of a GPU application, data in global memory can be shared among kernels and all threads

in a GPU can access global memory regardless of their block. The *local memory*, rather than a physical memory, is an abstraction of global memory, which is local to the thread and used to hold variables when register space is not sufficient.

The *shared memory* persists for the lifetime of a kernel, due to the fact that they are implemented in the L1 cache (on-chip memory) of an SM. As the shared memory is private to blocks, all threads of a block can access the same shared memory and synchronize. Furthermore, the shared memory, which is smaller in size, is fast and its access latency is comparable to that of registers. Consequently, memory-dependent operations can be profitably performed in the shared memory instead of the global memory. However, threads from different blocks can use only the global memory to share data.

The last on-chip memory type, the *register file* consists of 32-bit registers, which can be accessed in one clock cycle. The register file size is 64K 32-bit registers per SM and each thread has at most 255 32-bit registers.

C. Coalesced & Uncoalesced Access to Global Memory

As discussed in Section III-B, each kernel needs to access global memory to load and store data. Therefore, accessing global memory plays a very important role in high performance GPU programming. The data in the off-chip global memory are delivered to the CUDA cores via caches. Each time the global memory is accessed, the entire memory block is fetched and placed in a cache line of the same size as the memory block. Therefore, the line sizes of L1 and L2 cache memories have particularly significant impact on the performance of CUDA programs. As the line sizes of both cache memories are 128 B, a group of threads accessing consecutive addresses that coincide with a memory block of 128 B contributes to achieving the global memory access latency values listed in Table II. This is known as *coalesced* access in the GPU terminology. For instance, 16 threads accessing 8 B `unsigned long long` data types each, which happen to be consecutive in the same memory block, maximizes the performance as the entire 128 B block is brought to the cache with one global memory access operation. Otherwise, uncoalesced and strided accesses occur and the latency figures in Table II cannot be achieved as the same amount of data requires more than one memory block to be brought to the cache. One can affect the coalesced access following good programming practices.

D. Theoretical Occupancy

Occupancy is defined as the ratio between the number of actual active warps on an SM and the maximum *possible* number of active warps on the SM. As the occupancy plays an important role in efficient utilization of GPU resources (and ultimately the application performance), it is extremely essential to calculate the maximum theoretical occupancy of GPU before launching a kernel. When a kernel is created, the number of threads in a block, is determined as well as the shared memory size that will be available in the block. Since each block runs in one SM, the size of the register per thread is also fixed in the block. These three parameters (namely, block dimensions, the shared memory size, the number of registers per thread) directly affect the maximum theoretical occupancy. A block can include at most 1024 threads, which may not be necessarily optimum for achieving maximum theoretical occupancy. The reason is that increasing block size limits the resources per thread. For instance, for GPUs with compute capability 8.6, shared memory capacity per SM and is 100 KB, while maximum shared memory per thread block is 99 KB. Also, the L1 cache is configurable and its some parts can be used for shared memory and some parts can be used for data loaded or stored by the L2 cache. If the shared memory size is too high, a bottleneck occurs because there is little space left for data loaded or stored by the L2 cache. Therefore, in order to allocate more resources to threads one can consider deploying smaller block sizes such as 512 or even 256. This way, more warps can be active at a time. One can follow the NVIDIA documentation and guidelines to achieve higher occupancy rates. In Section V, we report the achieved occupancy rates and how it affects the overall timings of our NTT implementation (see the discussion of the optimum size for a thread block).

TABLE III
NOTATION AND SYMBOLS

symbol	explanation
n	The ring dimension
q	The modulus
ψ	The twiddle factor
Ψ	The array of twiddle factor powers
$bDim$ $\{bDim.x, \ldots\}$	# of threads in a block # of threads in each block dimension
bID $\{bID.x, \ldots\}$	Block ID Block ID in each dimension
tID $\{tID.x, \ldots\}$	Thread ID Thread ID in each dimension
bc and kc	# of blocks and # of kernels
koc	Array of # of outer loop iterations per kernel
$offset$	Difference between indices of inputs of a butterfly operation in a kernel or iteration

IV. NOTATION AND GPU DEVICES

In this section, we provide Table III, which includes a quick reference to symbols and notation frequently used throughout the paper.

We also detail the configurations of three GPU devices used in this work in Table IV. While not all GPU devices are of the same architecture (RTX 3060Ti and A100 are Amphere and RTX 4090 ADA) both have 64 ALUs in one SM. Three main features in both architectures are the determining factor in performance of NTT algorithms: i) Number of threads, ii) memory bandwidth, and iii) core frequency.

V. NTT ALGORITHMS FOR GPU

This section presents two different NTT algorithms and their implementations on GPU devices; **Merge-NTT** and **4Step-NTT**. We will detail their steps and explain the rationale in the specific design choices, which are directly dictated by the micro-architecture of GPU devices.

TABLE IV
HARDWARE FEATURES OF THE TESTBED

Feature	GPU		
	GPU-A	GPU-B	GPU-C
Architecture	RTX 3060Ti	A100 80 GB	RTX 4090
Threads	4864	6912	16384
Boost Freq.	1665 MHz	1410 MHz	2520 MHz
Memory Size	8 GB	80 GB	24 GB
Memory Type	GDDR6	HBM2e	GDDR6X
Memory Bus	256 bit	5120 bit	384 bit
Bandwidth	448 GB/s	1935 GB/s	1008 GB/s

A. Merge-NTT GPU Algorithm & Its Implementation

The **Merge-NTT** consists of two parts: i) **NTT host**, runing on the host device (i.e., general-purpose CPU), which determines the block size, block and grid shapes and the number of kernels etc., and ii) **NTT kernel** which performs the actual NTT computation on GPU. As observable in Algorithm 1, **Merge-NTT** consists of 3 main parts: i) the outer loop (the "**do-while**" loop starting in line 2), ii) the inner loop (the "**for**" loop in line 4), and iii) butterfly operation (**CT** Coley-Tukey) in line 8). In the first outer loop iteration, there is one NTT operation operating on the entire elements of the input vector. The number of *independent NTT operations* doubles from one iteration to the next operating on the separate parts of the input vector. For instance, in the second iteration, we have two independent NTT operations operating the first and the second half of the input vector, respectively. We can even assign indices to the NTT operations in a outer loop iteration, increasing from left to right for easy reference to them.

While the outer loop needs to be executed sequentially as there is data dependency between an iteration of the outer loop and the next, the inner loop iterations are independent and, therefore, suitable for parallelization. The threads in a block can perform all iterations of the inner loop concurrently and use the `__syncthreads()` intrinsic function for synchronization provided that $bDim \geq n/2$ where n is the ring dimension. Otherwise, more than one block is needed for full parallelization of the inner loop and the synchronization becomes problematic as the only way for that is via global memory, which results in using multiple kernels. For example, when $n = 2^{11}$, where there are 1024 **CT** operations in each inner loop iteration, one block (and one kernel) suffices to implement NTT operation as the maximum number of threads in CUDA-capable GPUs is 1024.

When $n > 2^{11}$, however, multiple kernels will be needed and the approach adopted in [22] uses a new kernel for each outer iteration until the iteration number $\log_2 n - \log_2 2bDim$. Therefore, the number of kernels can be calculated using the formula $kc = \log_2 (n/2bDim)$. For example, when $n = 2^{15}$ there are 15 outer iterations, the number of kernels needed can be computed as 4 for $bDim = 1024$. When the ring dimension increases to 2^{20} and 2^{24}, 9 and 13 kernels are needed, respectively, which renders the approach in [22] prohibitively inefficient for high ring dimensions. The work in [23], adopting a completely different approach, performs all outer iterations up to $\log_2 n - \log_2 2bDim$ in a single kernel, whereby some iterations of the inner loop is serialized. The second kernel is used thereafter as inter-block dependency is no longer an issue. This way, access to global memory is reduced using only two kernels. But, unfortunately, it can only perform NTT operation up to $n = 2^{15}$ as the number of registers in the SM is insufficient to get all vector elements from the global memory at the start of the kernel.

We can keep track of the number of outer loop iterations performed in each kernel in an array, named koc (kernel outer iteration count). For example, for the method in [22], the number of kernels (kernel count) $kc = \log_2 n - \log_2 2bDim$ and the elements of the array can be written as $koc[0] = \log_2 2bDim$ and $koc[i] = 1$ for $i \geq 1$. For [23], we have $kc = 2$ and $koc[0] = log_2 2bDim$, $koc[1] = \log_2 n - \log_2 2bDim$ with $n \leq 2^{15}$. Nevertheless, the partitioning of the outer loop iterations into kernels can be done in different ways to obtain a better GPU implementation as demonstrated in this paper.

Suppose the function `Partition` (see Algorithm 4) gets the ring dimension n, the maximum number of threads in a block (typically $mbd = 1024$ in CUDA-enabled GPU devices), and returns the optimal partition for n along with the block size $bDim$, without the particular shapes of a block and grid; namely their dimensions in different coordinates, which may vary depending on the particular kernel. Note that the returned block size is not necessarily the maximum block size and it turns out a smaller block size may be advantageous as will be shown in the subsequent sections. For example, when $bDim = 1024$ and $n = 2^{18}$ we can perform NTT in two kernels with seven outer loop iterations in the first kernel and 11 in the second kernel. Namely, we can write $kc = 2$, $koc[0] = 11$ and $koc[1] = 7$. In another example, when $bDim = 1024$ and $n = 2^{24}$, we have $kc = 3$, $kc[2] = 2, kc[1] = kc[0] = 11$. On the other hand, when we use a smaller block size such as $bDim = 256$, then we have $kc = 3$, $kc[2] = 6, kc[1] = kc[0] = 9$. Note that a kernel cannot perform more than $\log_2 2bDim$ outer loop iterations. One particular contribution of ours is that we propose such a novel access model to global memory that blocks in the kernels for $i \geq 1$ (i.e., those except the last kernel running the last $\log_2 2bDim$) perform the computations using shared memory in all outer loop iterations.

The proposed approach for determining the block size, kernel count, grid and kernel shapes is detailed in Algorithm 4, which is intended to execute on the host device to invoke kernels. In line 1 of the algorithm, the function `Partition` takes the ring dimension n and the maximum block size on CUDA-enable GPU (by default $mbd = 1024$) and returns the block size ($bDim \leq 1024$), the number of kernels ($kc \geq 2$ for $n > 2bDim$) and the array koc, whose elements keep the number of outer iterations in the corresponding kernel in reverse index; e.g., $i = 0$ and $i = kc - 1$ represents the last and the first kernels, respectively. The function `Partition` relies on empirical investigation to a certain extent as shown in the subsequent sections.

In partitioning the outer loop iterations into kernels, it may

Algorithm 4 NTT HOST

Input: $A[n], PsiTable[n], n, q, mbd$
Output: $A[n]$ ▷ In-place calculation

1: $\{bDim, kc, koc\} \leftarrow \texttt{Partition}(n, mbd = 1024)$ ▷ Optimal partition
2: $bc \leftarrow n/(2 \times bDim)$ ▷ # of blocks
3: $olc \leftarrow log_2(n)$ ▷ # of outer loop iterations
4: $oc \leftarrow -1$
5: **for** i **from** 0 by 1 **to** $kc - 1$ **do**
6: $oc \leftarrow oc + koc[i]$
7: $ko[i] \leftarrow 2^{oc}$
8: $olc \leftarrow olc - koc[i]$
9: **if** $i = 0$ **then**
10: $kgs[i] \leftarrow [1, bc]$
11: $kbs[i] \leftarrow [bDim/ko[i], ko[i]]$
12: **else**
13: $kgs[i] \leftarrow [bc/(2^{olc}), 2^{olc}]$
14: $kbs[i][1] \leftarrow (2 \times ko[i-1])/kgs[i][1]$
15: $kbs[i][0] \leftarrow bDim/kbs[i][1]$
16: **end if**
17: **end for**
18: $m \leftarrow 1$
19: **for** i **from** $kc - 1$ by -1 **to** 0 **do**
20: $dim3\ \mathbf{B}(kgs[i][0], kgs[i][1])$
21: $dim3\ \mathbf{T}(kbs[i][0], kbs[i][1])$
22: $\mathbf{NTT} \lll \mathbf{B}, \mathbf{T} \ggg (A, PsiTable, m, ko[i], koc[i], q)$
23: $m \leftarrow m \times (2^{koc[i]})$
24: **end for**

seem intuitive to perform as many as possible iterations in last kernels and more likely fewer number of them in earlier kernels. Different partitioning schemes, however, can also benefit the memory access performance; but care must be taken on deciding the optimal partitioning.

In Algorithm 4, bc stands for the number of blocks in the grid (block count) while olc represents the number of outer loop iterations remaining to be performed. The kernel offset ko, keeps the difference between the indices of the vector elements in the butterfly operation at the start of a kernel. For instance, in the last kernel, in which each block processes $2bDim$ vector elements, $ko = bDim$ while $ko = n/2$ in the first kernel.

After the block dimension $bDim$, the number of kernels kc, and the number of iterations in each kernel are determined in Step 1 of Algorithm 4, shapes of grids and kernels are computed in lines between Steps 5 and 18. Block and grid shapes, which simply pertain to their dimensionality, are determined by taking into account the memory dependencies between the outer loop iterations of the NTT algorithm. Both kgs (kernel grid shape) and kbs (kernel block shape) are two-dimensional arrays and their elements keep track of dimensions of grids and blocks in each kernel.

With two-dimensional access structure, one can arrange the blocks of the grid into different *block groups*. The first and second dimensions of kgs designate the number of blocks in each group and the number of block groups, respectively. The blocks, executing the same NTT operation and thus accessing the same range of vector elements, are organized into the same block group.

Example 1: Supposing $n = 2^{24}$ and $bDim = 1024$, the number of blocks is $bc = 8192$. Assume also $koc = [11, 11, 2]$. In the first iteration of the first kernel, all threads in the blocks access the entire input vector, then all blocks in the kernel belongs to the same group. Therefore, we have $kgs[2] = [8192, 1]$. As the first kernel iterates two times ($koc[2] = 2$), the second kernel will process four independent parts of the input vector in four independent NTT operations in its first iteration. Then, we can have four groups of blocks, i.e., $kgs[1] = [2048, 4]$. The final kernel has blocks, which are processing their own parts of the vector. Then, one can think there are as many block groups as the number of blocks. Thus, we have $kgs[0] = [1, bc]$.

In a similar fashion, we can group the threads in a block, as well. While the first dimension of kbs designates the number of threads in each group, the second does the number of thread groups.The grouping strategy depends on the number of iterations in the kernel. The goal is simply to ensure that the threads in a block will access the same vector elements in all outer iterations performed in the kernel.

Example 2: In Example 1, the first kernel performs the first two iterations of the outer loop as $koc[2] = 2$. Then, a block is grouped into two thread groups with 512 threads in each; namely, $kbs = [512, 2]$. This way, one group of threads will be accessing the vector elements in the second iteration, which are processed by the other thread group in the first iteration. As the first and second groups are in the same block, they use the same shared memory, which will eliminate accessing the global memory. In the second kernel, as there are 11 iterations, the block is organized into 1024 thread groups with a single thread in each group; namely $kbs = [1, 1024]$. Finally, in the last kernel, we can place all threads in the same group as a block is guaranteed to access the same $2bDim$ elements of the vector (i.e., $kbs = [1024, 1]$).

As observed in Example 2, the thread groups are excessively fragmented in the second kernel. Since threads in the same block process the vector elements that are located in distant locations in memory, this can adversely affect the memory access performance due to uncoalesced access pattern. Especially, if we can place the threads that access the same memory block (i.e., 128 B) in the same group, memory access will be optimized. Then, different partitioning of outer loop iterations into kernels should be considered.

Example 3: Suppose $koc = [11, 7, 6]$ for $n = 2^{24}$ and $bDim = 1024$. Then we will have

$$kgs = [[1, 8192], [128, 64], [8192, 1]] \text{ and}$$

$$kbs = [[1024, 1], [16, 64], [32, 32]].$$

Here, in the first two kernels, there are 32 and 16 threads in thread groups, respectively. For instance, in the first iterations of the second kernel, 16 threads access 16 consecutive vector

elements from the global memory, which is likely to be kept in the same memory block. If each thread accesses 8 B data types, this will result in a perfect match with the size of the memory block of 128 B.

Working with the maximum block dimension of $bDim = 1024$ may not always result in optimum performance, as good *occupancy* rate cannot be achieved due to poor resource utilization as explained in Section III. For instance, if we use 64-bit arithmetic (8 B) in the computation of NTT, then, each block uses 2048×8 B of the shared memory for the operands of the butterfly operation. This turns out to result in only a poor occupancy rate of 66% as an SM in a GPU device with compute capability 8.6 can run maximum of 1534 threads (see Table I). If, however, $bDim = 256$, then the maximum theoretical occupancy will be achieved as an SM can run more blocks at the same time with each block using $512 \times 8 = 4$ KB. If, for example, the SM runs 6 blocks of 256, then computation uses 24 KB of the shared memory.

To see the effects of the occupancy rate on the performance we ran a set of experiments. We executed our NTT algorithm with two different block dimensions, $bDim = 1024$ and $bDim = 256$, on three GPU devices. As seen in Table V, better occupancy rate can lead to more than 10% improvement in execution times.

TABLE V
EFFECT OF THE BLOCK DIMENSION ON PERFORMANCE WITH $n = 2^{17}$.

$bDim$	koc	**GPU-A**	**GPU-B**	**GPU-C**
1024	[11,6]	30.1 μs	24.3 μs	12.2 μs
256	[9, 8]	25.5 μs	21.0 μs	12.2 μs

From the discussions, we can conclude that there are couple of factors that determine the overall performance: block dimension, the number of kernels, the number of outer iterations in the kernels, kernel and grid shapes. As all the internal architectural details of the GPU devices and the scheduling of threads in SMs are known to a certain extent, an exact formula for choosing the best value of a factor cannot be given. For example, maximum block dimension of $bDim = 1024$ for NTT computation of high ring dimensions is not optimum due to poor occupancy rate. However, it is not easy to determine whether $bDim = 256$ or $bDim = 128$ is better although both enjoy maximum occupancy. Depending on the ring dimension and other factors, $bDim = 128$ may not fully utilize coalesced access to the memory. All our experiments support that using $bDim = 256$ is the optimum choice. For the number of outer iterations in the kernels koc, we rely on experimental observations and manual adjustments to a certain extent.

Using the configuration obtained in Algorithm 4 for block dimension, kernel count, kernel and block shapes, the **NTT Kernel** function in Algorithm 5 is called in Steps 19 and 24 of Algorithm 4. The index $GAddr$ in Algorithm 5 is used to access the global memory for the elements of the input vector A when the kernel is started. The threads access the global memory with $GAddr$ for array elements, which are placed in the shared memory. The size of the shared memory for each thread block depends on the block dimension $BDim$ and the size of input vector elements, w (e.g. $w = 4$ B or $w = 8$ B), and can be computed as $2 \times BDim \times w$. In order to exploit the coalesced access to the global memory, the threads are organized into groups, whose member threads access the global memory with consecutive indices of the input vector.

Algorithm 5 NTT KERNEL (NTT)

Input: $A[n], PsiTable[n], m, ko, koc, q$
Output: $A[n]$
1: $t_1 \leftarrow bDim.x \times bDim.y$ ▷ Block dimension
2: $\ell_1 \leftarrow tID.y \times (ko/2^{koc-1})$ ▷ Offset btw. thread groups
3: $\ell_2 \leftarrow bDim.x \times bID.x$ ▷ offset within an NTT
4: $\ell_3 \leftarrow 2 \times ko \times bID.y$ ▷ offset for an NTT
5: $GAddr \leftarrow tID.x + \ell_1 + \ell_2 + \ell_3$ ▷ For global mem.
6: $SAddr \leftarrow tID.x + tID.y \times bDim.x$ ▷ For shared mem.
7: $PsiAddr \leftarrow tID.x + \ell_1 + \ell_2 + \ell_3/2$ ▷ For twiddle factors
8: $offset_G \leftarrow ko$
9: $offset_S \leftarrow bDim.x \times bDim.y$
10: **SMem**$[SAddr] \leftarrow A[GAddr]$
11: **SMem**$[SAddr + offset_S] \leftarrow A[GAddr + offset_G]$
12: **for** i from 0 **by** 1 **to** $koc - 1$ **do**
13: $u \leftarrow \lfloor SAddr/t_1 \rfloor \times t_1 + SAddr$
14: $PsiIn \leftarrow m + \lfloor PsiAddr/ko \rfloor$
15: CT(**SMem**$[u]$, **SMem**$[u + t_1]$, $PsiTable[PsiIn], q)$
16: syncthreads()
17: $m \leftarrow m \times 2$
18: $t_1 \leftarrow t_1/2$
19: $ko \leftarrow ko/2$
20: **end for**
21: $A[GAddr] \leftarrow$ **SMem**$[SAddr]$
22: $A[GAddr + offset_G] \leftarrow$ **SMem**$[SAddr + offset_S]$

Example 4: In a hypothetical GPU, assume $n = 256$ and $bDim = 4$ and the partition is $kc = 3$ and $koc = [3, 3, 2]$. Then, we can compute the grid and block shapes as $kgs = [[1, 32], [8, 4], [32, 1]]$ and $kbs = [[4, 1], [1, 4], [2, 2]]$, respectively. In the first kernel, the blocks access the entire range of vector elements (as there is one NTT operation), therefore, there is one block group with 32 elements as $bc = 32$ and $bID.x \in [0, 31], bID.y = 0$. The execution of the two outer loop iterations of the first block of the first kernel is depicted in Table VI. There are two groups of threads in each block and threads in the same group access the consecutive elements of the input vector. Thus, we have $tID.x \in [0, 1]$ and $tID.y \in [0, 1]$. For example, the two threads in the first group access the four elements with indices [0,128] and [1,129], respectively. Note that, in the second iterations, there is no access to the global memory as all vector elements are already in the shared memory. The offset value of the indices between the first and second group of threads is calculated as $ko/2^{koc-1} = 64$ (See ℓ_1 in the second step of Algorithm 5).

As mentioned previously, thread blocks are organized as a two-dimensional array in grids and $bID.x$ and $bID.y$ are indices of a particular block. Here, $bID.y$ is the index of the

TABLE VI
THE EXECUTION OF THE FIRST BLOCK OF THE FIRST KERNEL IN EXAMPLE 4

bID	*tID*	*GAddr*	*SAddr*	*Corr.GAddr*	iteration
[0,0]	[0,0]	**[0, 128]**	[0,4]	[0,128]	0
			[0,2]	[0,64]	1
	[1,0]	**[1, 129]**	[1,5]	[1,129]	0
			[1,3]	[1,65]	1
	[0,1]	**[64, 192]**	[2,6]	[64,192]	0
			[4,6]	[128,192]	1
	[1,1]	**[65, 193]**	[3,7]	[65,193]	0
			[5,7]	[128,192]	1

NTT sub-block while $bID.x$ is the offset within the NTT sub-block.

Example 5: In Example 4, as there is a single NTT operation in the first iteration of the first kernel, there is one block group in a grid; namely we have $bID.y = 0$ for all block groups. To calculate the index of A in the global memory, we need to compute the offset value $\ell_2 = bDim.x \times bID.x$. For example, when $bID.x = 1$, the offset value for the index of A in global memory will be 2 as $bDim.x = 2$. See Table VII for the execution of the first three blocks of the first kernel for the first thread groups.

TABLE VII
THE EXECUTION OF THE FIRST THREE BLOCKS OF THE FIRST KERNEL IN EXAMPLE 4

bID	*tID*	*GAddr*	*SAddr*	*Corr.GAddr*	iteration
[0,0]	[0,0]	**[0, 128]**	[0,4]	[0,128]	0
			[0,2]	[0,64]	1
	[1,0]	**[1, 129]**	[1,5]	[1,129]	0
			[1,3]	[1,65]	1
	...				
[1,0]	[0,0]	**[2, 130]**	[0,4]	[2,130]	0
			[0,2]	[2,66]	1
	[1,0]	**[3, 131]**	[1,5]	[3,131]	0
			[1,3]	[3,67]	1
	...				
[2,0]	[0,0]	**[4, 132]**	[0,4]	[4,132]	0
			[0,2]	[4,68]	1
	[1,0]	**[5, 133]**	[1,5]	[5,133]	0
			[1,3]	[5,69]	1

As there are $bDim.x$ threads in each thread group $bID.x$ executing the same NTT operation, we need to add the offset value $\ell_2 = bDim.x \times bID.x$ (see Algorithm 5, Step 3) within an NTT operation to the index used to access to global memory at the start of a kernel. Finally, another offset value $\ell_3 = 2 \times ko \times bID.y$ (Algorithm 5, Step 4) is added to the global memory index. Here, $bID.y$ is the index of the NTT operation in outer loop iterations, which needs to be multiplied by twice the kernel offset value of ko.

As mentioned earlier, $tID.y$ is the index of a thread group in the same block, which refers to coalesced thread groups. For example, assume $bDim = 256$ and $kbs = [16, 16]$ (i.e., $tID.x \in [0, 15]$). As stated in Section III, since L1 cache memory and L2 cache memory line sizes are 128 bytes and if we use vector elements of 8 B (64-bit), for the best value of minimum thread group size we should have $bDim.x \geq 16$. Otherwise, the number of clock cycles increases when accessing global memory due to the increase in the number of uncoalesced accesses, which leads to decrease in the bandwidth and increase in the latency of computation.

Except for the last kernel, $bDim.x$ decreases as the number of outer iterations in kernels increases. Thus, using fewer outer loop iterations in those kernels must be considered. For example, in Algorithm 4, if the kernel performs as many as the maximum number of outer iterations (i.e., $log_2(2bDim)$), then we will cause the worst memory access pattern as $bDim.x = 1$.

Example 6: Assume $n = 2^{24}$, $bDim = 1024$, $bc = 8192$. The partitioning the outer loop iterations as $koc = [11, 11, 2]$ will lead to $bDim.x = 512, 1$, in the first and the second kernels, respectively. However, if we use $koc = [11, 7, 6]$, we will have $bDim.x = 32, 16$ for the first two kernels, which will result in much better global memory access pattern.

Reducing the number of outer iterations in a kernel, on the other hand, will increase the total number of kernels. Therefore, the number of accesses to the global memory will increase; as explained in Section III, which is not good for the overall latency. However, the best NTT implementation can be achieved if a balance is found between the number of accesses to global memory and the number of clock cycles spent accessing global memory.

Example 7: We examine the effect of the kernel count with a concrete example, with $n = 2^{18}$ and $bDim = 256$. We can use two different partitions: $kc_1 = 2$ and $koc_1 = [9, 9]$; and $kc_2 = 3$ and $koc_2 = [9, 5, 4]$. The results are given in Table VIII. As

TABLE VIII
EFFECT OF THE NUMBER OF KERNELS ON PERFORMANCE WITH $n = 2^{18}$ AND $bDim = 256$.

koc	*bDim.x*	**GPU-A**	**GPU-B**	**GPU-C**
[9,9]	[256, 1]	53.0 μs	31.8 μs	21.2 μs
[9, 5, 4]	[256, 16, 32]	41.7 μs	28.4 μs	15.5 μs

can be observed in Table VIII, when $kc = 2$, $bDim.x = 1$ for the first kernel, which will result in inferior latency. On the other hand, when three kernels are used, as the global memory access patterns are much better (due to $bDim.x \geq 16$), the latency values are improved. As can be seen from the results, using extra kernels may be advantageous depending on input parameters, if global memory access patterns result in poor performance despite fewer number of kernels.

B. GPU Implementation of **4Step-NTT**

As can be observed in Algorithm 3 given for **4Step-NTT**, the algorithm consists of six main operation blocks: i) transpose of $n_1 \times n_2$ matrix B (Step 6), ii) n_2 n_1-point NTT of rows of B (Steps 7-9), iii) transpose of $n_2 \times n_1$ matrix B (Step 10), iv) multiplication with twiddle factors (Steps 11-15), v) n_1 n_2-point NTT of columns of B (Steps 16-18), and

vi) transpose of $n_1 \times n_2$ matrix B (Step 19). Here, the NTT operation blocks can be performed in parallel as there are n_2 or n_1 independent NTT operations in each block.

Note that the vector-to-matrix and matrix-to-vector operations do not have to performed explicitly. Note also that the first and the last transpose operations are not necessary and can be skipped provided that both NTT and inverse NTT operations do not perform them. This naturally necessitates modification in data access patterns, which will not pose any significant performance penalty. The transpose operation, on the other hand, can be prohibitively expensive for large matrices, and eliminating them results in improvement in the latency. Finally, the multiplication with twiddle factors in fourth operations block can be incorporated into the second NTT operation block. With these optimizations, the **4Step-NTT** algorithm is simplified to have only two NTT operation blocks and a transpose operation in between.

In both NTT operation blocks, there are many independent NTT operations of much smaller sizes than those used in the **Merge-NTT** algorithm, which can be performed in parallel. The number and the sizes of NTT operations in the first and second NTT blocks can be important in the performance of its GPU implementation. Although in the original **4Step-NTT** algorithm, it is suggested that B be a square matrix (or as close to square matrix as possible), namely $n_1 \approx n_2$, the algorithm works with various selections of n_1 and n_2. Then, we need to determine specific values of of n_1 and n_2 for a given n to optimize the second transpose operation, which may be problematic as it can result in costly memory accesses.

Example 8: Consider the ring dimension of $n = 2^{22}$, where the coefficients of input polynomial can be arranged into a $2^{11} \times 2^{11}$ matrix; i.e., $n_1 = n_2 = 2^{11}$. Then, the first NTT block consists of 2^{11}-point NTT operations. And, considering $bDim \leq 1024$, one block can only process at most $2bDim/n_1 = 4$ NTT operations. Consequently, only a small number of threads in a block will access the consecutive addresses in the global memory during the subsequent transpose operation. This will lead to sub-optimal access pattern to the global memory, which adversely affects the latency. Thus, after the first NTT operation block, it will be more efficient to terminate the kernel and launch another that performs transpose operation through shared memory. Although using an additional kernel for the transpose increases the number of global memory accesses, from latency perspective this turns out to be more efficient compared to storing the matrix elements in transposed format directly to global memory in the same kernel after the first NTT operation block. This is due to the fact that performing the transpose by the existing threads of the kernel blocks through global memory will lead to uncoalesced accesses by the threads, resulting in increased latency for global memory access. Instead, that storing the matrix elements to the global memory before the transpose and then loading them in a new kernel will enable to perform the transpose in the shared memory can be much more efficient. Therefore, using an additional kernel enables the transpose process to be performed with much lower latency.

Alternatively, using a smaller value of n_1 can be advantageous to improve the memory access pattern during the transpose operation performed in the same kernel as the first block of NTT operations. For instance, we can use the dimensions $n_1 = 2^7$ and $n_2 = 2^{15}$ for $n = 2^{22}$. This way, each block's shared memory can be considered as two-dimensional array, each row of which corresponds to an independent NTT operation and by this means as many as $2bDim/n_1 = 16$ NTT operations can be performed for $n_1 = 128$. After all NTT operations completed, columns of the array are read by threads and stored to the global memory exploiting the advantages of coalesced accesses. In summary, using a suitably small values of n_1, one can eliminate the extra kernel for the transpose operation without the adverse effect of sub-optimal global memory accesses.

Table IX shows the timing results of **4Step-NTT** based on five different cases for $n = 2^{22}$ for the two matrix dimensions $(n_1, n_2) \in \{(2^{11}, 2^{11}), (2^7, 2^{15})\}$ on three different GPU devices, where different implementation techniques are applied.

TABLE IX
EFFECT OF THE MATRIX DIMENSION ON THE PERFORMANCE OF THE 4STEP NTT ALGORITHM

n	**case**	$[n_1, n_2]$	**GPU-A**	**GPU-B**	**GPU-C**
2^{22}	1	$[2^{11}, 2^{11}]$	1219.29 μs	439.99 μs	342.51 μs
	2	$[2^{11}, 2^{11}]$	1226.70 μs	413.65 μs	256.77 μs
	3	$[2^{11}, 2^{11}]$	893.27 μs	302.35 μs	190.70 μs
	4	$[2^7, 2^{15}]$	780.11 μs	296.16 μs	175.92 μs
	5	$[2^7, 2^{15}]$	617.26 μs	252.71 μs	142.84 μs

The first three cases in Table IX capture the effect of transpose operations in performance. For instance, in cases 1 and 2, the first and last transpose operations explained in Algorithm 3 are included in timings. In case 1, all three transpose operations described in Algorithm 3 are performed in the NTT kernels, which has an adverse effect on performance because of uncoalesced access to the global memory. In case 2, on the other hand, using an additional kernel for each transpose results in much better performance compared to case 1 for A 100 and RTX 4090.

As explained earlier, when used in an application, there is no need to execute the first and the last transpose operations. In case 3, only one transpose operation is performed in a separate kernel, which leads to significant acceleration.

In cases 4 and 5 are we use a rectangular matrix $(n_1, n_2) = (2^7, 2^{15})$, where there is a significant speedup in comparison with the first three cases. The difference between cases 4 and 5 is that case 5 does not use an additional kernel for transpose operation. Since n_1 is small, many NTT operations can be performed in the same GPU block. After the NTT operation, the transpose operation can be performed in the same kernel via reading the columns of the shared memory.

The dimension of the second block of NTT operations also plays an important role and very large values of n_2 can lead to performance penalties. A balance between n_1 and n_2 should be reached to achieve the best performance. Table X contains the

matrix dimensions for all ring sizes of interest, which is found to give the best performance in each case experimentally.

TABLE X
MATRIX SIZES FOR 4STEP NTT IMPLEMENTATION

n	$n_1 \times n_2$	n	$n_1 \times n_2$
2^{12}	$2^5 \times 2^7$	2^{19}	$2^5 \times 2^{14}$
2^{13}	$2^5 \times 2^8$	2^{20}	$2^5 \times 2^{15}$
2^{14}	$2^5 \times 2^9$	2^{21}	$2^6 \times 2^{15}$
2^{15}	$2^6 \times 2^9$	2^{22}	$2^7 \times 2^{15}$
2^{16}	$2^7 \times 2^9$	2^{23}	$2^7 \times 2^{16}$
2^{17}	$2^5 \times 2^{12}$	2^{24}	$2^8 \times 2^{16}$
2^{18}	$2^5 \times 2^{13}$		

VI. RESULTS

In this section, we present the GPU implementation results of the **Merge-NTT** and the **4STEP-NTT** algorithms on three different GPU machines, whose architectural details are given in Table IV. Only NTT results and comparisons are included in this section since INTT is just an inverse model of NTT, and its results are almost identical to NTT results.

In our NTT implementations, three different types modular reduction which are optimized in assembly, are applied. Two of them are Barret [24] and Plantard [38] reduction, which can work with any NTT friendly prime, while the third is the Goldilock reduction [39], if the 64-bit goldilock prime (i.e., $q = 2^{64} - 2^{32} + 1$) is employed. The goldilock reduction method, which consists of fewer number of instructions, is used for more efficient modular arithmetic. Here, we also show the effects of a fast goldilock modular reduction method on the overall performance of an NTT operation. Lastly, we include performance comparison of the proposed algorithms with those in the literature.

All the measurements in the subsequent tables are kernel timings only; namely, they do not include the transfer time from the CPU to the GPU. While taking the measurements, the **cudaEventRecord** function, which is one of CUDA's native functions, is used. It is a reliable function for measuring time as it measures all GPU activity from the time kernels are invoked to the terminations of the kernels. To ensure stability, all scenarios are repeated at least 50 times (as many as 1000 times for most cases) depending on the ring dimension, and the average values of the results are presented. In addition, the timings of all implementations are taken on devices with the same CUDA driver and same operating system[2].

A. Comparison of Merge-NTT and 4STEP-NTT Algorithms

In this section, we compare the **Merge-NTT** and **4Step-NTT** algorithms in terms of both latency and throughput. The latency results of a single NTT for **Merge-NTT** and **4Step-NTT** algorithms are given in Table XI with three different GPUs in Table IV, where the better timings are in bold. As can be observed from the table, the timings of the two algorithms are generally close to each other. For relatively small values of n, the **Merge-NTT** algorithm tend to perform better than the **4Step-NTT** algorithm. For very large values of n, however, the **4Step-NTT** algorithm's performance is superior, due to the fact that the former algorithm becomes memory-bound with the increase of n and that better spatial locality of the latter algorithm becomes advantageous. Another observation from the table is that the architectural differences can affect the performance to a certain extent. For intance, while the **Merge-NTT** is slower than the **4Step-NTT** on RTX 4090 for $n = 2^{20}$, the opposite is true for other GPU devices.

TABLE XI
TIMINGS OF NTT ALGORITHMS FOR **Single** FORWARD NTT ON DIFFERENT GPUS IN μs (**Latency**)

	MERGE	**4-STEP**
$logn$	GPU-A / GPU-B / GPU-C	GPU-A / GPU-B / GPU-C
12	8.32 / **12.57** / 7.64	**7.92** / 12.62 / **7.29**
13	8.43 / 12.94 / 7.74	**8.30** / **12.64** / **7.53**
14	9.15 / **12.97** / **7.80**	**8.95** / 13.70 / 8.08
15	10.49 / **13.84** / **8.03**	**10.36** / 14.41 / 8.49
16	14.79 / **15.93** / **8.66**	**14.33** / 16.34 / 9.04
17	24.66 / **21.94** / **11.81**	**23.97** / 22.57 / 11.86
18	41.70 / **28.38** / 15.46	**40.75** / 28.89 / **15.37**
19	**85.25** / 43.38 / 24.54	87.30 / **43.19** / **23.13**
20	**164.02** / **72.12** / 39.40	165.85 / 72.60 / **38.33**
21	321.88 / 142.29 / 70.68	**317.61** / **135.83** / **66.37**
22	647.32 / 272.99 / **135.19**	**617.26** / **252.71** / 142.84
23	1422.32 / 602.16 / **377.91**	**1221.85** / **499.01** / 422.91
24	3448.09 / 1152.67 / 1020.95	**2514.96** / **1016.50** / **969.92**

All time measurements are taken for the 64-bit Goldilock prime.

In HE implementations, as many independent NTT operations are executed concurrently, we also need to measure the throughput. GPU has many threads to execute many NTT operations in parallel. In fact, running a single NTT does not reveal the real potential of a GPU device as its resources are not fully utilized, especially for smaller values of n. Table XII presents the timing results for different number of concurrently running NTT operations (i.e., 4, 16, 32,64, 128). It is clear from the table that GPU devices can compute many NTT operations in parallel without increasing the execution time significantly. However, when the GPU resources are fully utilized, the execution of NTT operations is serialized. For instance, on RTX 4090, when $n = 2^{14}$ while four NTT operations take 8.04 μs, 16 of them take only slightly more time, 11.48 μs when **Merge-NTT** is used. The increase in execution times is more salient for RTX 3060 Ti and A 100, which support fewer number of threads. Nevertheless, we observe the same effect for RTX 4090 also for higher ring dimensions, which require more resources.

When performances of the three GPUs are compared, both algorithms run faster on RTX 3060Ti than A 100 at low ring dimensions when the number of NTT operations is low. This is due to the fact that the clock frequency is more dominant than the bandwidth; in other words the operations are compute-bounded for those input sizes. However, when

[2]Ubuntu 20.04 LTS and CUDA version 12.1

TABLE XII
TIMINGS OF GPU IMPLEMENTATIONS OF BATCH FORWARD NTT IN μs

n	NTT count	GPU-A † / ‡	GPU-B † / ‡	GPU-C † / ‡
2^{12}	4	8.67 / 8.67	**12.24** / 12.98	**7.59** / 7.72
	16	**12.27** / 12.63	**13.91** / 14.20	**7.86** / 8.14
	32	**17.09** / 17.41	**15.65** / 16.98	**8.54** / 9.25
	64	**26.90** / 28.69	**19.63** / 20.91	10.67 / 10.68
	128	**53.04** / 54.71	**27.85** / 28.97	15.62 / **15.25**
2^{13}	4	**9.45** / 9.82	**13.25** / 13.66	**7.65** / 8.04
	16	**17.52** / 18.53	**16.20** / 17.14	**8.80** / 9.51
	32	**28.00** / 29.24	**20.22** / 21.73	**11.08** / 11.36
	64	**54.41** / 57.89	**29.10** / 30.48	16.05 / **15.92**
	128	**101.53** / 104.55	**46.07** / 48.11	24.89 / **24.72**
2^{14}	4	**13.24** / 13.70	**14.94** / 15.10	**8.04** / 8.55
	16	**29.80** / 31.90	**21.10** / 22.34	**11.48** / 11.70
	32	**56.19** / 60.98	**30.63** / 31.76	16.82 / 16.80
	64	**103.47** / 110.90	**48.90** / 51.29	26.36 / 26.32
	128	**197.25** / 209.85	**88.52** / 90.58	47.11 / **45.16**
2^{15}	4	**19.41** / 20.46	**17.26** / 18.59	**9.19** / 9.86
	16	**57.71** / 62.48	**32.26** / 33.21	17.55 / **17.31**
	32	**106.48** / 112.25	**51.85** / 53.47	27.81 / **27.41**
	64	**200.83** / 211.89	**93.86** / 94.58	49.83 / **47.72**
	128	**389.93** / 407.62	192.83 / **185.17**	90.66 / **85.48**
2^{16}	4	**33.64** / 34.28	**23.26** / 24.30	12.46 / 12.49
	16	113.16 / **112.93**	**55.92** / 56.30	30.76 / **28.56**
	32	**219.09** / 211.73	100.77 / **99.30**	53.85 / **49.66**
	64	422.00 / **406.88**	196.43 / **188.83**	97.16 / **91.00**
	128	819.44 / **803.18**	366.00 / **354.03**	**184.86** / 192.16

†: MERGE NTT
‡: 4STEP NTT
All time measurements are taken for the 64-bit Goldilock prime.

TABLE XIII
TIMINGS (μs) OF GPU IMPLEMENTATION OF OUR SINGLE FORWARD NTT AND THEIR COMPARISON WITH THE WORKS IN LITERATURE

			$\log_2 n$				
Work	Device	$\log_2 q$	12	13	14	15	16
[25]	Titan V	60	-	-	44.1	84.2	-
[27]	RTX 2080 Ti	64*	-	-	-	83.3	-
[28]	GTX 1070	64*	-	-	57.8	-	-
[26]	GTX 1070	64*	-	-	66.8	-	-
[22]	V 100	55	-	-	29	39	-
[23]	RTX 3060 Ti	62	14.0	14.9	19.1	35.9	-
[24]	A 100	62	-	-	13.3	-	16.5
[24]	V 100	62	-	-	11.5	-	16.4
T.W.	RTX 3060 Ti	64*	8.32	8.43	9.15	10.49	14.79
T.W.	A 100	64*	12.57	12.94	12.97	13.84	15.93
T.W.	RTX 4090	64*	7.64	7.74	7.78	8.03	8.66
T.W.	RTX 3060 Ti	60	8.45	8.67	9.38	11.45	15.00
T.W.	A 100	60	12.72	13.22	13.27	14.56	16.15
T.W.	RTX 4090	60	7.60	7.65	7.77	8.20	8.86

T.W.: This Work (**MERGE NTT**)
*: uses constant prime $q = 2^{64} - 2^{32} + 1$

the ring dimension and the number of NTT operations are high, the execution becomes memory-bound. Then, A 100 performs much better than RTX 3060 Ti with its superior memory bandwidth. RTX 4090 has a relatively high memory bandwidth, albeit only half as much as A100. Nonetheless, thanks to the new ADA architecture, RTX 4090 has many more threads and operates at a very high frequency. Thus, RTX 4090 is the best of the three GPU devices for all instances (see Table XI and Table XII). However, as the operations become memory-bound for higher values of n, the difference in the timing results obtained on RTX 4090 and A 100 becomes mush less visible. For example, for $n = 2^{22}$ **Merge-NTT** takes 272.99 μs and 135.19 μs on A100 and on RTX 4090, respectively, which translates into a speedup of $2.02\times$ (see Table XI). However, the speedup values dramatically drop to $1.59\times$ for $n = 2^{23}$ and $1.13\times$ for $n = 2^{24}$.

In our all experiments, we report the timing results for the 64-bit goldilock primes, for which the modular reduction is very fast. On the other hand, when residue number system (RNS) is used to work with much larger modulus, as in the case of Homomorphic Encryprion (HE) schemes, using random primes with a fast reduction algorithm can be advantageous as it decreases the number of base moduli in the RNS [40]. For HE applications, special goldilock prime $Q = 2^{64} + 2^{32} + 1$ is used *carrier* for smaller primes q_i employed in RNS arithmetic. When carrier prime is used for NTT, there is an upper bound for RNS prime bases, imposed by the inequality $q_i^2 n < Q$. For instance, each base can be at most 25-bit and 24-bit, respectively, for the ring sizes $n = 2^{14}$ and $n = 2^{15}$.

Thus, we also implemented the **Merge-NTT** algorithm using 60-bit NTT-friendly random primes with Barret and Plantard reduction to see the performance penalty due to their usage. For the ring dimension range of $[2^{12}, 2^{24}]$, using 64-bit goldilock primes offers maximum 25.5% speedup over the NTT implementation with random primes for A100 (the figures are very close for the other GPU devices). Consequently, we can conclude that using NTT-friendly random primes can be more advantageous for HE applications using RNS arithmetic.

B. Comparison of NTT Results with Related Works in the Literature

The literature contains several works that accelerates NTT (and related operations) using old as well as the contemporary GPU devices. They either use a randomly chosen NTT-friendly primes or specials primes (e.g., goldilock primes), which offer faster modular reduction. We compare our timing results of single NTT with those in the literature for the parameter sets suitable for the HE algorithms in Table XIII. The table lists the results for both 60-bit random primes implemented with Plantard Reduction and the 64-bit goldilock prime implemented with Goldilock Reduction. Considering the architectural differences, our **MERGE NTT** algorithm compares favorably with all works in the literature. More results including those taken with three different modular reduction methods, Goldilock, Plantard, and Barret Reductions, respectively, are available Table XV in Appendix.

TABLE XIV
TIMINGS (μs) OF GPU IMPLEMENTATION OF OUR SINGLE FORWARD NTT FOR HIGHER VALUES OF n WITH $\log q = 253$ (q IS BLS12-377 PRIME)

	GPU-B	**GPU-C**
$logn$	**T.W. / SPPARK**	**T.W. / SPPARK**
19	**230.68** / 246.66	**111.60** / 132.96
20	**416.47** / 515.72	**206.88** / 259.88
21	**854.36** / 937.38	**424.25** / 504.40
22	**1690.40** / 1912.60	**959.88** / 1075.67
23	**3511.66** / 4284.51	**2027.27** / 2362.28
24	**7294.05** / 8060.20	**4178.86** / 4484.67
25	**10021.3** / 16952.4	**5205.58** / 9131.73
26	**20962.8** / 38815.0	**12144.4** / 20213.7
27	**43772.7** / 74886.3	**28104.0** / 38177.9
28	**92813.9** / 179563.0	**56198.5** / 82987.3

T.W.: This Work (**MERGE NTT**)

We also compare our results with the GPU implementation (known as **SPPARK**), the winner of 2022 zprize competition[3], which is intended for ZK-SNARK protocols working with much higher ring dimensions and a larger 253-bit random prime. For a fair comparison, we adopt the SPARKK implementation of the Montgomery multiplication algorithm in our CUDA code. Only the kernels of SPPARK for NTT are included in the timings excluding the kernels for LDE or bit reverse order operations. In addition, the execution times of same number of NTT operations are measured for both SPPARK and our implementations and their averages are calculated. The average timings of a single NTT operation for ring dimensions between 2^{19} and 2^{28} are reported in Table XIV[4]. The results in Table XIV show that our implementation is $6.9/93.5\%$ and $7.3/75.4\%$ min/max faster than SPPARK on A 100 and RTX 4090, respectively. In addition, our implementation also supports higher ring dimensions than 2^{28} as the SPPARK implementation.

VII. CONCLUSION

In this work, we presented two NTT algorithms which are specifically designed for GPU implementations. The first algorithm is based on an iterative version while the other on a version know as the 4-step algorithm. The main objective of the proposed algorithms is to optimize the memory access latency by optimizing the number of CUDA kernel invocations, and by determining the optimum sizes and shapes of the thread blocks to take advantage of coalesced access to the global memory. We provided two types of timing results for both algorithms on three powerful GPUs: i) execution time of a single NTT operation (*latency*) and ii) the execution times of many concurrently running NTT operations (throughput). Throughput is a more important metric for the performance of NTT in homomorphic encryption applications as they execute a multitude of them in parallel. Nevertheless, all other works on the subject report only latency results for a very limited input parameters such as ring dimension and coefficient modulus size, which may be misleading to asses the performance of an NTT algorithm. Therefore, ours is a unique work in the literature by providing a very extensive timing results on a wide range of input parameter set and three different GPU devices. Our latency results suggest that the two algorithms perform comparably with very close timing results. The throughput results, on the other hand, indicate that the the iterative algorithm performs markedly better than the 4-step algorithm. While the later algorithm is usually preferred in the literature for its better spatial locality, our results suggest that the former can be an alternative for homomorphic encryption applications. Also, when compared with the best known implementation for very high degree polynomials in the range of $[2^{12}, 2^{28}]$, our implementation is superior.

As future work, we are planning to develop a method for the twiddle factor generation in the iterative NTT algorithm as it suffers from the increased number of them processed in a large portion of iterations. Other optimization techniques such as higher radix butterfly circuits can be incorporated to further accelerate both algorithms.

[3]https://www.zprize.io/

[4]The extended version of the execution times comparison for ring dimensions between 2^{12} and 2^{28} is available in Table XVI in Appendix.

ACKNOWLEDGMENT

Erkay Savaş was supported by the European Union's Horizon Europe research and innovation programme under grant agreement No: 101079319. Ali Şah Özcan was supported by Ulvetanna Inc[5]. In addition, the authors would like to thank Enes Recep Türkoğlu and Can Ayduman for their support in developing a Python model of the four-Step NTT algorithm.

REFERENCES

[1] C. Gentry, "Computing arbitrary functions of encrypted data," *Commun. ACM*, vol. 53, no. 3, p. 97–105, mar 2010. [Online]. Available: https://doi.org/10.1145/1666420.1666444

[2] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems (extended abstract)," in *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, R. Sedgewick, Ed. ACM, 1985, pp. 291–304. [Online]. Available: https://doi.org/10.1145/22145.22178

[3] D. Bernhard and B. Warinschi, "Cryptographic voting - A gentle introduction," in *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, ser. Lecture Notes in Computer Science, A. Aldini, J. López, and F. Martinelli, Eds., vol. 8604. Springer, 2013, pp. 167–211. [Online]. Available: https://doi.org/10.1007/978-3-319-10082-1_7

[4] M. Al-Rubaie and J. M. Chang, "Privacy-preserving machine learning: Threats and solutions," *IEEE Security Privacy*, vol. 17, no. 2, pp. 49–58, 2019.

[5] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," *IACR Cryptol. ePrint Arch.*, p. 349, 2014. [Online]. Available: http://eprint.iacr.org/2014/349

[6] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 839–858.

[7] K. Sako, "Cryptography and digital transformation," in *Recent Advances in Industrial and Applied Mathematics*, T. Chacón Rebollo, R. Donat, and I. Higueras, Eds. Cham: Springer International Publishing, 2022, pp. 159–171.

[5]https://www.ulvetanna.io/

[8] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Comput. Surv.*, vol. 51, no. 4, jul 2018. [Online]. Available: https://doi.org/10.1145/3214303

[9] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, *Homomorphic Encryption Standard*. Cham: Springer International Publishing, 2021, pp. 31–62. [Online]. Available: https://doi.org/10.1007/978-3-030-77287-1_2

[10] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 326–349. [Online]. Available: https://doi.org/10.1145/2090236.2090263

[11] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "Snarks for C: verifying program executions succinctly and in zero knowledge," *IACR Cryptol. ePrint Arch.*, p. 507, 2013. [Online]. Available: http://eprint.iacr.org/2013/507

[12] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, M. Fischlin and J. Coron, Eds., vol. 9666. Springer, 2016, pp. 305–326. [Online]. Available: https://doi.org/10.1007/978-3-662-49896-5_11

[13] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," *SIAM Journal on Computing*, vol. 43, no. 2, pp. 831–871, 2014. [Online]. Available: https://doi.org/10.1137/120868669

[14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2633600

[15] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[16] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 113–124. [Online]. Available: https://doi.org/10.1145/2046660.2046682

[17] "Microsoft SEAL (release 3.6)," https://github.com/Microsoft/SEAL, Nov. 2020, microsoft Research, Redmond, WA.

[18] "PALISADE Lattice Cryptography Library (release 1.11.5)," https://palisade-crypto.org/, 2021.

[19] S. Halevi and V. Shoup, "Algorithms in helib," in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 554–571.

[20] V. Sidorov, E. Y. F. Wei, and W. K. Ng, "Comprehensive performance analysis of homomorphic cryptosystems for practical data processing," *CoRR*, vol. abs/2202.02960, 2022. [Online]. Available: https://arxiv.org/abs/2202.02960

[21] R. C. Agarwal and C. S. Burrus, "Fast convolution using fermat number transforms with applications to digital filtering," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 22, pp. 87–97, 1974.

[22] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savas, "Efficient number theoretic transform implementation on GPU for homomorphic encryption," *J. Supercomput.*, vol. 78, no. 2, pp. 2840–2872, 2022. [Online]. Available: https://doi.org/10.1007/s11227-021-03980-5

[23] A. Özcan, C. Ayduman, E. R. Türkoğlu, and E. Savaş, "Homomorphic encryption on GPU," *IEEE Access*, pp. 1–1, 2023.

[24] K. Shivdikar, G. Jonatan, E. Mora, N. Livesay, R. Agrawal, A. Joshi, J. L. Abellan, J. Kim, and D. Kaeli, "Accelerating polynomial multiplication for homomorphic encryption on GPUs," in *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2022, pp. 61–72. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SEED55351.2022.00013

[25] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 264–275.

[26] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

[27] Z. Zheng, "Encrypted cloud using GPUs," Master's thesis, KU Leuven, 2020.

[28] J.-Z. Goey, W.-K. Lee, B.-M. Goi, and W.-S. Yap, "Accelerating number theoretic transform in gpu platform for fully homomorphic encryption," *The Journal of Supercomputing*, vol. 477, pp. 1455–1474, 2021. [Online]. Available: https://doi.org/10.1007/s11227-020-03156-7

[29] A. C. Mert, E. Karabulut, E. Öztürk, E. Savas, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Trans. Computers*, vol. 71, no. 11, pp. 2829–2843, 2022. [Online]. Available: https://doi.org/10.1109/TC.2020.3017930

[30] K. Derya, A. C. Mert, E. Öztürk, and E. Savas, "Coha-ntt: A configurable hardware accelerator for ntt-based polynomial multiplication," *IACR Cryptol. ePrint Arch.*, p. 1527, 2021. [Online]. Available: https://eprint.iacr.org/2021/1527

[31] A. C. Mert, E. Öztürk, and E. Savas, "FPGA implementation of a run-time configurable ntt-based polynomial multiplication hardware," *Microprocess. Microsystems*, vol. 78, p. 103219, 2020. [Online]. Available: https://doi.org/10.1016/j.micpro.2020.103219

[32] F. Hirner, A. C. Mert, and S. S. Roy, "Proteus: A tool to generate pipelined number theoretic transform architectures for fhe and zkp applications," Cryptology ePrint Archive, Paper 2023/267, 2023, https://eprint.iacr.org/2023/267. [Online]. Available: https://eprint.iacr.org/2023/267

[33] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1295–1309. [Online]. Available: https://doi.org/10.1145/3373376.3378523

[34] H. Zhao, D. Ding, F. Wang, P. Hua, N. Wang, Q. Wu, and Z. Chai, "Hardware acceleration of number theoretic transform for zk-snark," *Engineering Reports*, vol. n/a, no. n/a, p. e12639. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/eng2.12639

[35] A. Özcan, C. Ayduman, E. R. Türkoğlu, and E. Savaş, "Homomorphic encryption on GPU," *IEEE Access*, pp. 1–1, 2023.

[36] D. H. Bailey, "Ffts in external or hierarchical memory," in *Supercomputing '89:Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, 1989, pp. 234–242.

[37] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 451–460, jun 2010. [Online]. Available: https://doi.org/10.1145/1816038.1816021

[38] T. Plantard, "Efficient word size modular arithmetic," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1506–1518, 2021.

[39] M. Hamburg, "Ed448-goldilocks, a new elliptic curve," *IACR Cryptol. ePrint Arch.*, p. 625, 2015. [Online]. Available: http://eprint.iacr.org/2015/625

[40] W. Dai, Y. Doröz, Y. Polyakov, K. Rohloff, H. Sajjadpour, E. Savas, and B. Sunar, "Implementation and evaluation of a lattice-based key-policy ABE scheme," *IEEE Trans. Inf. Forensics Secur.*, vol. 13, no. 5, pp. 1169–1184, 2018. [Online]. Available: https://doi.org/10.1109/TIFS.2017.2779427

APPENDIX

A. Additional Algorithms

Algorithm 6 Gentleman-Sande Butterfly (**GS**)

Input: U, V, Ψ, q
Output: $\bar{U}$, $\bar{V}$
1: $\bar{U} \leftarrow U + V \pmod q$
2: $\bar{V} \leftarrow (U - V) \times \Psi \pmod q$
3: **return** $\bar{U}, \bar{V}$

Algorithm 7 Four-Step INTT (**4Step-INTT**)

Input: $n_1, n_2 \leq n$ and $n_1 \times n_2 = n$
Input: $A(x) \in \mathbb{Z}_q[x]/(x^n - 1)$ in bit-reversed order
Input: $\Omega[k] = \Omega^{-br(j)\times i} \pmod q$ **for** $0 < k \leq n - 1$, **for** $0 < j \leq n_1 - 1$ **for** $0 < i \leq n_2 - 1$
Input: $\omega_{0_{br}}[k] = \omega_0^{-br(k)} \pmod q$ **where** $\omega_0 = \Omega^{-(n/n_1)} \pmod q$, **for** $0 < k \leq n_1 - 1$ (Powers of ω_0 stored in bit-reverse order)
Input: $\omega_{1_{br}}[k] = \omega_1^{-br(k)} \pmod q$ **where** $\omega_1 = \Omega^{-(n/n_2)} \pmod q$, **for** $0 < k \leq n_2 - 1$ (Powers of ω_1 stored in bit-reverse order)
Output: $A \leftarrow INTT(A)$ in polynomial standard-order
1: **for** i from 0 by 1 to n_1 **do** ▷ Vector to matrix
2: **for** j from 0 by 1 to n_2 **do**
3: $B_{i,j} \leftarrow A_{i \times n_2 + j}$
4: **end for**
5: **end for**
6: $B = B^T$ ▷ 1^{st} transpose operation
7: **for** j from 0 by 1 to n_2 **do** ▷ n_2, n_1-point INTTs
8: $B_j \leftarrow$ **INTT**(B_j, ω_0, n_1, q)
9: **end for**
10: $B \leftarrow B^T$ ▷ 2^{nd} transpose operation
11: **for** i from 0 by 1 to n_1 **do**
12: **for** j from 0 by 1 to n_2 **do**
13: $B_{i,j} \leftarrow B_{i,j} \times \Omega[i \times n_2 + j] \pmod q$
14: **end for**
15: **end for**
16: **for** i from 0 by 1 to n_1 **do**
17: $B_i \leftarrow$ **INTT**(B_i, ω_1, n_2, q) ▷ n_1, n_2-point INTTs
18: **end for**
19: $B = B^T$ ▷ 3^{rd} transpose operation
20: **for** j from 0 by 1 to n_2 **do** ▷ Matrix to vector
21: **for** i from 0 by 1 to n_1 **do**
22: $A_{j \times n_1 + i} \leftarrow B_{j,i}$
23: **end for**
24: **end for**
25: **for** k from 0 by 1 to n **do**
26: $A_k \leftarrow (A_k \times n^{-1}) \pmod q$
27: **end for**
28: **return** A

B. Additional Tables

Algorithm 8 Merge Inverse NTT (**Merge-INTT**)

Input: $\bar{\boldsymbol{a}} \in \mathbb{Z}_q^n$ in bit-reversed order
Input: $\Psi_{rev}[k] = \Psi^{-br(k)} \pmod q$ **for** $0 < k \leq n - 1$ (power of Ψ^{-1} stored in bit-reverse order)
Input: $n = 2^l$, q ($q \equiv 1 \mod 2n$)
Output: $a(x) \in \mathbb{Z}_q[x]/(x^n + 1)$ standard-order
1: $t \leftarrow 1$; $m \leftarrow n$
2: **do**
3: $j_1 \leftarrow 0$; $h \leftarrow m/2$
4: **for** i from 0 by 1 to h **do**
5: $j_2 \leftarrow j_1 + t - 1$
6: **for** j from j_1 by 1 to $j_2 + 1$ **do**
7: $\bar{a}_j, \bar{a}_{j+t} \leftarrow$ **GS**$(\bar{a}_j, \bar{a}_{j+t}, \Psi_{rev}[h + i], q)$
8: **end for**
9: $j_1 \leftarrow j_1 + 2 \times t$
10: **end for**
11: $t \leftarrow 2 \times t$
12: $m \leftarrow m/2$
13: **while** $m < n$
14: **for** i from 0 by 1 to n **do**
15: $a_i \leftarrow (\bar{a}_i \cdot n^{-1}) \pmod q$
16: **end for**
17: **return** a

Algorithm 9 Goldilock Reduction [39]

Input: $C = a \times b$, where $a, b < q \equiv 2^{64} - 2^{32} + 1$
Output: C_{out} ($C \mod q$)
1: $X_3 \leftarrow C \gg 96$
2: $X_2 \leftarrow (C \gg 64) \& (2^{32} - 1)$
3: $X_1 \leftarrow C \& (2^{64} - 1)$
4: $C_{out} \leftarrow X_1 + (X_2 \times (2^{32} - 1)) - X_3$
5: **if** $C_{out} \geq q$ **then** $C_{out} \leftarrow C_{out} - q$
6: **else** $C_{out} \leftarrow C_{out}$
7: **end if**

Algorithm 10 Plantard Reduction [38]

Input: a, b *where* $a, b < q$; $k = \lceil log_2(q) \rceil$ *and* $w = 2^{k+2}$
Output: C_{out} ($a \times b \mod q$)
1: $\sigma \leftarrow (-2^{2w}) \mod q$ ▷ Pre-computed
2: $\tilde{b} \leftarrow b \times \sigma \mod q$ ▷ Pre-computed
3: $\tilde{b} \leftarrow \tilde{b} \times (q^{-1} \mod 2^{2w}) \mod 2^{2w}$ ▷ Pre-computed
4: $T \leftarrow a \times \tilde{b} \mod 2^{2w}$
5: $T \leftarrow T \gg w$
6: $T \leftarrow (T \times (q + 1)) \gg w$
7: **if** $T \equiv q$ **then** $C_{out} \leftarrow 0$
8: **else** $C_{out} \leftarrow T$
9: **end if**

TABLE XV
TIMINGS OF MERGE NTT ALGORITHM FOR SINGLE FORWARD NTT ON DIFFERENT GPUS WITH THREE DIFFERENT MODULAR REDUCTION ALGORITHMS IN μs (LATENCY)

	GPU-A	**GPU-B**	**GPU-C**
$logn$	† / ‡ / ♠	† / ‡ / ♠	† / ‡ / ♠
12	8.32 / 8.45 / 8.93	12.57 / 12.72 / 12.95	7.64 / 7.60 / 7.76
13	8.43 / 8.67 / 9.31	12.94 / 13.22 / 13.42	7.74 / 7.65 / 7.80
14	9.15 / 9.38 / 9.71	12.97 / 13.27 / 13.46	7.80 / 7.77 / 8.06
15	10.49 / 11.45 / 11.88	13.84 / 14.56 / 14.69	8.03 / 8.20 / 8.23
16	14.79 / 15.00 / 17.67	15.93 / 16.15 / 17.37	8.66 / 8.86 / 8.96
17	24.66 / 28.24 / 28.45	21.94 / 22.45 / 24.09	11.81 / 12.94 / 12.52
18	41.70 / 45.26 / 49.90	28.38 / 28.46 / 32.36	15.46 / 16.33 / 17.44
19	85.25 / 94.46 / 97.88	43.38 / 41.67 / 51.74	24.54 / 24.18 / 27.97
20	164.02 / 179.82 / 188.30	72.12 / 68.87 / 89.69	39.40 / 38.36 / 47.73
21	321.88 / 351.73 / 373.72	142.29 / 135.10 / 182.17	70.68 / 66.64 / 87.19
22	647.32 / 735.41 / 759.05	272.99 / 262.17 / 342.66	135.19 / 129.97 / 168.43
23	1422.32 / 1596.05 / 1607.2	602.16 / 580.07 / 727.43	377.91 / 465.20 / 409.42
24	3448.09 / 3882.28 / 3596.6	1152.67 / 1122.70 / 1445.5	1020.95 / 1173.72 / 1028.55

†: Goldilock Reduction used. (64bit)
‡: Plantard Reduction used. (60bit)
♠: Barret Reduction used. (60bit)

Algorithm 11 Barrett Reduction [24]

Input: $C = a \times b$, where $a, b < q$; $k = \lceil log_2(q) \rceil$; $\mu = \lfloor \frac{2^{2k}}{q} \rfloor$
Output: C_{out} $(C \mod q)$
1: $r \leftarrow C \gg (k-2)$
2: $r \leftarrow r \cdot \mu$
3: $r \leftarrow r \gg (k+3)$
4: $r \leftarrow q \cdot r$
5: $C_{out} \leftarrow (C - r)$
6: **if** $C_{out} >= q$ **then** $C_{out} \leftarrow C_{out} - q$
7: **else** $C_{out} \leftarrow C_{out}$
8: **end if**

TABLE XVI
TIMINGS (μs) OF OUR SINGLE FORWARD NTT AND COMPARISON WITH SPPARK'S IMPLEMENTATION WITH $\log q = 253$ (q IS BLS12-377 PRIME)

	GPU-B	**GPU-C**
$logn$	**T.W. / SPPARK**	**T.W. / SPPARK**
12	26.79 / **23.33**	15.38 / **11.71**
13	28.88 / **28.08**	16.25 / **15.84**
14	29.97 / **28.48**	**17.04** / 18.73
15	**31.72** / 37.73	**18.50** / 20.35
16	48.27 / **47.14**	**21.64** / 27.58
17	**77.10** / 77.61	**39.44** / 42.04
18	120.06 / **116.45**	**64.04** / 65.81
19	**230.68** / 246.66	**111.60** / 132.96
20	**416.47** / 515.72	**206.88** / 259.88
21	**854.36** / 937.38	**424.25** / 504.40
22	**1690.40** / 1912.60	**959.88** / 1075.67
23	**3511.66** / 4284.51	**2027.27** / 2362.28
24	**7294.05** / 8060.20	**4178.86** / 4484.67
25	**10021.3** / 16952.4	**5205.58** / 9131.73
26	**20962.8** / 38815.0	**12144.4** / 20213.7
27	**43772.7** / 74886.3	**28104.0** / 38177.9
28	**92813.9** / 179563.0	**56198.5** / 82987.3

T.W.: This Work (**MERGE NTT**)