IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 14 April 2025, accepted 5 May 2025, date of publication 14 May 2025, date of current version 27 May 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3570024

RESEARCH ARTICLE

High-Performance Number Theoretic Transform on GPU Through radix2-CT and 4-Step Algorithms

ALISAH OZCAN, ARSALAN JAVEED, AND ERKAY SAVAS, (Member, IEEE)
Faculty of Engineering and Natural Sciences, Sabancı University, 34956 Istanbul, Türkiye

Corresponding author: Alisah Ozcan (alisah@sabanciuniv.edu)

This work was supported by European Union's Horizon Europe Research and Innovation Program under Grant 101079319.

ABSTRACT The number theoretic transform (NTT) provides a practical and efficient technique to perform multiplication of very large degree polynomials typically found in fully homomorphic encryption (FHE), lattice-based cryptography, and non-interactive succinct zero-knowledge proof systems such as zk-SNARK. In this paper, we focus on this aspect and present two robust algorithms for efficient NTT using readily available GPU cards as hardware accelerators. These algorithms are based on the radix-2 Cooley-Tukey (CT) and 4-Step techniques, which are rooted in classical FFT research. To this end, our algorithms leverage novel strategy to optimize memory access patterns adaptive to input size, which often is very large. Our approach: 1) reduces and optimizes the number of accesses required for global memory for thread synchronization on the GPU device, and 2) systematically improves and enhances the use of spatial locality. We achieve this effect by carefully controlling parameters such as the number of kernels, thread block size and shape, and thread layout, which directly impact overall NTT performance. The proposed optimizations enable our NTT implementation to handle very large polynomial sizes up to 2^{28}, which are usually a limiting factor in existing approaches, and achieve remarkable performance. To the best of our knowledge, our proposed technique is unique and provides a recipe for selecting suitable configurable parameter combinations to achieve top performance for a given polynomial degree. Furthermore, we perform thorough experiments and empirically assess the performance of our proposed algorithms on three mainstream commercial GPU cards by NVIDIA. Finally, we demonstrate that our algorithms compare favorably and outperform an existing commercial-grade open-source implementation in this arena.

INDEX TERMS Graphical processing unit, homomorphic cryptography, hardware acceleration, number theoretic transform, polynomial arithmetic.

I. INTRODUCTION

Advanced cryptographic schemes, such as homomorphic encryption [1], [2], [3] and zero-knowledge proofs [4], serve as fundamental building blocks in numerous privacy-critical applications, including electronic voting [5], privacy-preserving- machine learning [6], cryptocurrency [7], smart contracts [8], and so on to name a few. It is reasonable assertion that establishment of a reliable and just digital era depends crucially on these technologies [9]. Consequently, the research community has invested significant time and enormous efforts to theorize and build practical solutions of real-world significance in this arena [10], [11], [12], [13], [14], [15]. However, among the notable challanges of practical significance, tackling high computation costs [16] and reducing it down remains an active area of research interest.

Although, significant strides have been made in efficiently implementing several of Fully Homomorphic Encryption (FHE)-schemes for practical usage [17], [18], [19]. However, additional enhancements are required to meet the latency and throughput demands of real-world applications, particularly with the emergence of AI-enabled ubiquitous applications in domains that demand stringent privacy preservation [20], driven by legal and ethical obligations.

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

In contemporary lattice-based cryptographic schemes and FHE-schemes, the multiplication of very large degree polynomials, often spanning lengths far exceeding 2^{16} and involve 64-bit wide-and-beyond individual elements, poses a critical bottleneck. This bottleneck becomes pronounced when these multiplications need to be repeated numerous times, thereby directly impacting the application throughput [21]. Typical of these settings, the multiplication utilizes polynomial rings $\mathcal{R} = \mathbb{Z}[x]/\Phi_n(x) = \mathbb{Z}[x]/(x^n \pm 1)$, where $\Phi_n(x)$ is the cyclotomic polynomial of degree n which is some power of two.

Lately, efforts to accelerate FFT-like algorithms, such as the NTT (a special case of FFT), have gathered attention [28]. These efforts leverage the architecture and computational capabilities of underlying hardware platforms. CPUs, especially those equipped with Advanced Vector Extensions (AVX), are being used for acceleration in these settings [29], albeit with some limitations compared to GPUs. The limited core count in CPUs restricts the degree of achievable parallelism, even with AVX instructions that allow vectorized operations on multiple data points simultaneously. Moreover, the complexity of managing vector instructions and the overhead of context-switching in multithreaded scenarios can detriment meaningful speedup gains, which is undesirable for compute-intensive cryptographic applications. Furthermore, the complex memory hierarchy with multiple levels of cache introduces undesirable latency for such algorithms, as opposed to the more straightforward and high-bandwidth memory access available in GPUs. In contrast, leveraging FPGAs for acceleration presents distinct opportunities and challenges. They offer highly customizable hardware configurations tailored to specific algorithms, enabling efficient parallel processing and fine-grained control over data flow. However, notable challenges include design complexity, longer design-synthesis times, limited on-chip resources, vendor support, cost and scalability issues become often prohibitive [30], [31].

Beyond traditional graphical rendering, modern GPUs have undergone remarkable advancements and now play a crucial role in a wide array of computational tasks of scientific interest. Current GPUs are capable of executing thousands of parallel threads and are equipped with high-bandwidth memory hierarchies, both on-chip and off-chip. A single GPU thread is a sequence of instructions executed on a GPU core, designed to run in parallel with thousands of other threads simultaneously in large-scale computations. These threads are organized and executed within a GPU kernel, a function designed to run in parallel across thousands of execution threads on the streaming multiprocessors (SMs), the cores of the GPU responsible for execution. Threads are grouped into blocks, which can be adjusted in size and shape, allowing threads within the same block to synchronize more efficiently. Key factors such as the number of kernels, block size, and block shape play a critical role in determining application performance. As a result, achieving efficient GPU implementations necessitates mature algorithm design, systematic approach to tune parameters, and diligent code optimizations tailored for GPU architecture at hand [32].

In this work, our motivation lies in leveraging the parallel processing capabilities and high-bandwidth memory of GPUs to present and implement two efficient algorithms to compute NTT in both directions, for high degree polynomial multiplication on three mainstream NVIDIA GPU cards of practical significance. To this end, we summarize our contributions as follows:

- We present two algorithms rooted in classical FFT research, namely the *radix2-CT* and *4-Step*, designed for efficient NTT computation optimized to leverage the parallel computing capabilities of GPU cards. These algorithms are tailored with strategic memory patterns to enhance memory access efficiency. We target two distinct ranges for polynomial degrees: i) $n \in [2^{12}, 2^{16}]$, optimized for homomorphic encryption applications, and ii) $n \in [2^{20}, 2^{28}]$, suited for the zero-knowledge zk-SNARK protocol. The algorithms are designed to be flexible and parametric, capable of handling any power-of-two value of n within the specified ranges.
- We demonstrate that the performance of aforementioned algorithms is sensitive to configurable parameters such as kernel count, block size, and block shape. Furthermore, we propose a systematic approach to determine the optimal parameter selection for achieving the fastest implementation for a given polynomial degree. This approach optimizes the access to global memory on the GPU.
- We empirically evaluate the performance of both algorithms on three modern GPU cards, leveraging all optimization techniques, and report both latency and throughput results. Furthermore, we demonstrate that our proposed implementation surpasses an industrial-grade open-source alternative. Our implementation is made available on GitHub [33].

The remainder of the paper is organized as follows: Section II provides a review of the literature; Section III discusses background information to aid comprehension; Section IV explores the GPU programming model; Section V presents two NTT algorithms customized for efficient GPU implementation; Section VI-B reports our experimental findings; Section VII discusses threats to validity; and finally, Section VIII concludes the paper with final remarks and conclusions.

II. RELATED WORK

Accelerating Number Theoretic Transform for various applications remains an active research topic [34], [35], [36], [37] in different application contexts [34], [38], [39], [40] including homomorphic encryption [41], [42] and postquantum cryptography [43]. To this end, researchers have relied upon leveraging novel memory designs [44], [45],

[46] for CPUs, to migrating altogether towards accelerator hardware such as GPUs [42], [47], [48], FPGAs [49], [50], [51] and more recently DPUs [52], [53].

Li et al. [44] introduce MeNTT, a novel in-memory NTT accelerator optimized for efficient computation within a variant of SRAM array design. Their design includes specialized peripherals for rapid modular operations and employs a novel mapping strategy to streamline data flow between NTT stages, simplifying routing among SRAM columns. These advancements result in significant reductions in latency and energy consumption compared to previous approaches. Park et al. [45] introduce RM-NTT, a resistive random access memory (RRAM)-based compute-in-memory (CIM) design aimed at accelerating NTT operations. RM-NTT employs a vector-matrix multiplication approach instead of traditional FFT-like methods. Twiddle factors are stored in modified forms within RRAM arrays to enable efficient in-place computation, complemented by a Montgomery reduction algorithm. These optimizations notably decrease NTT operation latency compared to both CIM and non-CIM-based accelerators. Li and Geng et al. [46] introduce a high-throughput and compact accelerator for Polynomial Matrix Multiplication (PMM) using NTT, based on a crossbar CIM architecture. Their design prioritizes optimizing PMM algorithms tailored to their approach and introduces an innovative bit-mapping strategy to minimize area and energy consumption. This design achieves a significant latency improvement over existing ones.

Li et al. [47] propose a faster GPU-based NTT implementation for TFHE applications, utilizing matrix multiplication for a decomposed small-point NTT. The design includes a merging preprocessing method to reduce modulo operations by combining multiple inputs. Furthermore, logical left shifts are used instead of arithmetic multiplication when multiplying the merged result by rotation factors, enhancing computational efficiency. This approach efficiently implements a 1024-point NTT, achieving a speedup ratio of approximately 2.5x according to experimental results. Durrni et al. [48] argue that while NVIDIA's cuFFT library provides FFT computation capabilities, there are significant opportunities to enhance the performance of FFT and NTT algorithms on modern GPUs through specialized operations and architectural advancements. The authors present four major optimizations that leverage tensor cores and warp-shuffle instructions. Extensive evaluations on 32-bit integers and small-parameter space demonstrates that their approach outperforms existing GPU-based implementations, achieving up to a 4x speedup with some limitations. Goey et al. [42] introduce several techniques to enhance the performance of NTT implementation on a GPU, including register-based storage of twiddle factors and multi-stream asynchronous computation, taking advantage of new GPU-architecture features. Applied to a lesser-known FHE scheme, the proposed NTT implementation achieves homomorphic multiplications in 0.27 ms on a GTX1070 desktop GPU and 7.49 ms on a Jetson TX1 embedded system. The findings highlight the practicality of the implementation for both server environments and embedded systems.

Fan et al. [79] introduce TensorFHE, leveraging CUDA and Tensor Cores (TCUs) to accelerate NTT and other operations for CKKS with ring dimensions up to 2^{16}. TCUs are specialized units in recent GPUs, optimized for high-throughput matrix operations with low-precision arithmetic (int8, fp16), which limits flexibility and requires a specialized programming approach. TensorFHE aims to maximize the number of FHE operations within a given timeframe by employing operation-level batching to enhance data parallelism. It utilizes a hierarchical decomposition of CKKS into reusable kernels to reduce pipeline stalls, applies TCU-based acceleration for NTT kernels, and implements data layout optimizations. While TCUs accelerate NTT computations, CUDA cores handle parallelized algorithms for other kernels. However, TCUs' fixed-size matrix operations and limited precision constrain their efficiency for NTT with large ring dimensions, requiring careful algorithmic adjustments.

Hafeez et al. [76] suggest that while NTT is a suitable choice, its parameter limitations often prevent direct application in lattice-based schemes like Saber and Sable for IoT applications, prompting the exploration of alternative techniques to accelerate polynomial convolution. To this end, a GPU-based TMVP polynomial convolution utilizing tensor cores is presented. The special structure of the Toeplitz matrix facilitates a prearrangement and memory placement according to the reduction pattern of these schemes, enabling a subsequent matrix-vector product that maximizes parallelism and results in significant performance gains on tensor cores.

Ji et al. [78] focus on optimizing NTT for the Kyber scheme, which typically involves a ring dimension of 2^8. They propose three strategies that entail a depth-first search approach to improve computational efficiency and reduce global memory access overhead by strategically utilizing a limited number of registers as intermediate buffers. Fixed-size coefficient slices are processed in each butterfly layer and number of slices is determined by the distance between a given butterfly units to the last layer of the transform. Coefficients are fetched from global memory in batches in equal-length slices of specified order for efficient parallel processing. However, a limitation of the sliced layer merging scheme is that its efficiency is constrained by the fixed number of registers, which may not be sufficient for larger ring dimensions or more complex transformations, potentially diminishing the gains due to increased scheduling overhead.

Ngoc et al. [77] propose a flexible butterfly unit for NTT on GPUs, designed for large prime moduli. These large primes can be decomposed into smaller, more manageable sub-primes that fit into machine word size using the RNS. This approach demonstrates the feasibility of creating a

processing pipeline for each individual RNS prime, with butterfly units for each prime logically organized into separate planes. By mapping each plane and its butterfly units onto thread blocks and grids, multiple planes can be processed concurrently, achieving effective parallelism.

Yang et al. [75] present a FHE library on GPU supporting BFV, BGV and CKKS schemes leveraging a design referred as weighted-NTT, to maximize parallelism while minimizing total IO latency within the hierarchical GPU memory. Their NTT is developed for RNS polynomial representation, a technique that manages computations involving elements larger than the native machine word size by utilizing a predefined set of smaller moduli. The NTT employs a single kernel, where each thread block performs a residue-wise transform for each modulus. Within each block, individual threads handle a fixed set of predetermined l-coefficients and execute the corresponding radix-l butterfly operations. For large polynomials, this kernel is recursively invoked to process segments of the polynomial when memory is constrained. However, this approach becomes impractical for very large n due to excessive register usage, which adversely impacts SM occupancy and overall performance. The parameter l can be reduced to decrease latency, provided sufficient shared memory is available. Nonetheless, the presented design at most support upto 2^{17} polynomial rings. Jung et al. [74] employ Kim et al. [41] implementation that explores the trade-offs and performance challenges of applying FFT-based optimizations to the NTT. They analyzed the performance of the Cooley-Tukey and Stockham algorithms with various radix values, leveraging shared memory. Furthermore, they argue that NTT performance is constrained by main-memory bandwidth due to its algorithmic characteristics. To alleviate this bottleneck, they propose a novel on-the-fly root generation technique, which partly computes the twiddle factors dynamically and subsequently avoids a certain fraction of modulo operations during the full twiddle factor generation phase thus, reducing the size of precomputed tables while minimizing main memory accesses. This approach achieves an additional average speedup of 9.5% while reducing the cost of modular multiplication for rings up to 2^{17}. Wang et al. [72] accelerate NTT by leveraging GPU as part of a broader effort to optimize multiple operations within the BFV scheme in an approach named HE-Booster, for parameter spaces dealing with ring dimension up to 2^{16}. The approach by design is a multi-GPU capable, developed to enhance the throughput of all five phases of BFV involving polynomial arithmetic, and NTT. They argue that the $\log n$ compute stages in NTT can be executed concurrently by inter-thread local synchronization approach which they propose to enhance thread-level parallelism and to facilitate high-throughput data processing. By avoiding intermediate global synchronization and minimizing kernel launches, their method reduces significant execution overhead. Although, local synchronization improves performance for smaller ring dimensions, it may suffer as ring dimensions increase beyond 2^{16} as they introduce more complex dependencies and greater memory overhead, leading upto synchronization inefficiencies and diminished performance gains. In contrast, our approach supports ring dimensions beyond 2^{16} by optimizing global memory access. We achieve this through an input-specific arrangement of thread blocks and grids, which leverages shared memory for the coordination of concurrent small-sized sub-transforms thus, needing minimal inter-thread synchronization, eliminating need for special mechanisms. Ozcan et al. [64] utilize inner and outer iteration-based kernels in their implementation and employ shared memory during certain outer iterations for thread block coordination. Their method executes the outer iterations sequentially, unlike ours which is a concurrent approach. Futhermore their appoach is constrained by the limited size of register memory that prevents it from processing NTT operations for inputs larger than 2^{15}. Moreover, their design relies on a maximum of two kernels, the first employs only two thread blocks and register memory whereas, the second handles the remainder of outer iterations. By comparison our approach not only demonstrates significant speed improvements, being 1.68x to 3.42x faster for rings 2^{12} to 2^{15} on the same GPU model, but also can efficiently batch process multiple NTTs unlike theirs.

Although our research utilizes GPUs to accelerate the NTT transform with two algorithms, it distinguishes itself from previous studies in several aspects. We opted for a fixed radix-2 design over mixed-radix to optimize GPU memory access, which is critical for larger polynomial rings exceeding 2^{16}. Briefly, radix-2 design processes two elements at a time, enabling pairwise uniform memory access patterns and needs low register usage per thread thus, collectively maximizing GPU occupancy and alleviates bank conflicts. Whereas, larger or mixed-radix designs although, process more elements per step to reduce the total number of computation stages, they lead up to irregular memory access patterns, that manifests into increased memory latency which is an undesirable side-effect. Additionally, larger-radix designs require higher register usage per thread, which beyond a certain limit, negatively impacts GPU occupancy and diminishes performance gains. Furthermore, the number of kernels used for computation plays a vital role in optimizing performance for a specified ring dimension. Using an optimal number of kernels, rather than a fixed number, ensures better load balancing and alleviates hardware underutilization. Larger ring dimensions demand more resources and computational stages, and a fixed kernel configuration may either oversaturate GPU resources or fail to utilize them efficiently. Thus, in this work, we build upon these observations and present an approach that dynamically determines the optimal number of kernels required for a given polynomial ring dimension. Additionally, our method systematically configures the launch parameters to ensure coalesced memory access, adaptive to input size. Furthermore, we conducted evaluations and experiments

using three mainstream GPU models suitable for domestic, scientific, and industrial markets. In contrast, existing studies typically concentrate on a single GPU model, such as the outdated RTX 940-1020, according to modern standards. Our focus remains exclusively on desktop and server computing applications, excluding embedded/IoT scenarios, which might be explored in future work requiring adjustments to suit resource-constrained settings. Our experiments validate the practicality of our approach for large ring dimensions up to 2^{28}, making it well-suited for FHE schemes that require larger parameters for security guarantees and involve significant computational intensity. Previous GPU-based studies often limit their analysis to smaller ring dimensions, up to size 2^{16}, and occasionally speculate on the feasibility of scaling to larger ring dimensions without empirical validation. We demonstrate a practical speedup of up to 1.3x on average for the largest ring dimension of 2^{28} compared to an industrial-grade open-source implementation [54].

III. PRELIMINARIES AND BACKGROUND INFORMATION

To aid the understanding of the rest of the paper, this section provides essential background information, notations, and relevant concepts.

A. NOTATIONS AND ELABORATIONS

In this section, we provide Table 1, which includes a quick reference to symbols and notation frequently used throughout the paper.

TABLE 1. Symbolic notations and their elaborations.

Notation	Elaboration
n	the ring dimension (e.g. 2^{14})
q	the modulus value (usually a prime)
ψ_i/ω_i	a twiddle factor
Ψ/Ω	a vector of powers of the twiddle factors
$bDim$	num. of threads in a block
$\{bDim.x, \ldots\}$	num. of threads in each block dimension
bID	block Id
$\{bID.x, \ldots\}$	block Id in each dimension (e.g. x, y, z)
tID	thread Id
$\{tID.x, \ldots\}$	thread Id in each dimension
bc and kc	num. of thread blocks and num. of gpu kernels
koc	array of num. of outer loop iterations per kernel
$offset$	difference between indices of inputs of a butterfly operation in a kernel or iteration

B. GRAPHICS PROCESSING UNIT (GPU)

In recent years, interest in utilizing GPUs for non-graphics tasks has surged, accompanied by advancements in programming languages and tools that simplify GPU programming efforts. However, mastering full GPU hardware utilization remains a complex skill that requires significant time investment to achieve proficiency. GPU computing, known as GPGPU, has been pivotal over the past decade, transitioning from fixed graphics operations to user-friendly third-party languages for general purpose programming. Despite initial developmental challenges, platheora of GPU programs showcased their ability to outperform CPUs for specific algorithms, expanding their adoption and accessibility [55].

The adoption of GPU computing primarily stems from its substantial performance edge over CPUs, with a significant performance gap attributable to physical per-core limitations and architectural disparities between the two processor types. Parallelism represents a viable approach for enhancing performance, particularly for applications with parallelizable workloads that are well-suited for execution on GPUs [56].

In addition to GPUs, other accelerator cores such as field-programmable gate arrays (FPGAs) and the cell broadband engine (Cell BE) [56], [57] have also gained significant interest and success across various application areas. Accessibility to GPUs remains a major driver behind GPGPU computing, as significant percentage of modern commodity computers already come equipped with a dedicated GPU, whereas FPGAs and Cell BE are typically found only in specialized setups.

Currently, the PC market is dominated by three major GPU vendors. Although, Intel leads in integrated and low-performance market-segment, whereas AMD and NVIDIA dominate high-performance and discrete graphics [58]. Nonetheless, in academic and industrial sectors, NVIDIA remains the prime choice, prompting our work's focus on NVIDIA GPUs, although the discussed concepts and techniques are generally applicable to AMD GPUs as well.

C. NUMBER THEORETIC TRANSFORM

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 $\omega \in \mathbb{Z}_q$, known as primitive root of unity with $\omega^n \equiv 1 \pmod q$. For rings of $\mathcal{R}_q = \mathbb{Z}_q/(x^n+1)$, where n is a power of two, the negacyclic transformation can be used with $\psi^{2n} \equiv 1 \pmod q$, where $\omega = \psi^2 \bmod q$.

After NTT is applied to two polynomials $a, b \in \mathcal{R}_q$, then the multiplication can be done using element-wise multiplication of the resulting vectors, $\bar{\boldsymbol{a}}$ and $\bar{\boldsymbol{b}}$. Inverse NTT operation is then applied to obtain the final result in $\mathcal{R}_q$.

D. MODULAR REDUCTION ALGORITHMS

Modular multiplication and modular reduction of long integers are essential operations in implementing most cryptographic algorithms. While the Karatsuba divide-and-conquer

approach is a popular method for modular multiplication, modular reduction is more complex and time-consuming. Therefore, an efficient implementation of the modular reduction operation is crucial for the overall performance of the cryptographic algorithm [59].

To this end, we followed existing research and selected the Barrett [60], Goldilocks [61], and Plantard [62] algorithms for modular reduction. It is noteworthy that while the former two algorithms remain widely used, the latter is relatively new. Next we present a brief account of these algorithms.

Barret reduction (Algorithm 1) performs efficient modular reduction on input C and is particularly well-suited for scenarios where multiple reductions need to be performed with the same modulus q. It involves a preprocessing step to compute a value μ that can be reused, which directly speeds up the reduction process. Computation of μ takes the number of bits k and a base $b = 2$ for binary values. The algorithm then first performs an approximate reduction through a sequence of simple operations (lines 1- 5) followed by a conditional correction step (line 6- 10) on the final result $Cout$.

In contrast, Goldilocks reduction (Algorithm 2) is generally preferred for cases where the prime modulus q is of the form $2^a - 2^b + 1$, which has origins in elliptic-curve cryptography [61]. The name *Goldilocks* typically refers to one such special prime of the aforementioned form. In a nutshell, this algorithm relies on decomposing the input C into parts based on the modulus form (lines 1-3), followed by reduction through simple operations on the decomposed parts by exploiting the special form of the modulus (lines 4-9). For our use case in this work, we utilized the prime value $2^{64} - 2^{32} + 1$ for 64-bit modular reduction.

Plantard reduction (Algorithm 3) is a relatively new approach in the area of modular reduction, designed with an emphasis on efficiency in certain cryptographic settings such as NTT [62]. It leverages properties of the modulus (generally used in lattice-based cryptography) to improve the reduction process. Broadly, the steps involve precomputing a value σ based on the modulus q and w (lines 1), followed by a series of modular arithmetic along with binary shifts (lines 2- 6). The final result is brought within range by conditionally applying a correction step (lines 7- 11).

Overall, we have selected the Barrett, Plantard, and Goldilocks algorithms for modular reduction in our subsequent NTT implementations. This selection aims to evaluate and compare their performance, determining if one algorithm outperforms the others or if they are equivalently effective.

Three type of modular reduction algorithms that we employed in this work.

Algorithm 1 Barrett Reduction [60]

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** $C_{out} \geq q$ **then**
7 $\quad C_{out} \leftarrow C_{out} - q$
8 **else**
9 $\quad C_{out} \leftarrow C_{out}$
10 **end**

Algorithm 2 Goldilock Reduction [61]

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**
6 $\quad C_{out} \leftarrow C_{out} - q$
7 **else**
8 $\quad C_{out} \leftarrow C_{out}$
9 **end**

Algorithm 3 Plantard Reduction [62]

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 $\sigma \leftarrow (-2^{2w}) \mod q$
2 $\tilde{b} \leftarrow b \times \sigma \mod q$
3 $\tilde{b} \leftarrow \tilde{b} \times (q^{-1} \mod 2^{2w}) \mod 2^{2w}$
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**
8 $\quad C_{out} \leftarrow 0$
9 **else**
10 $\quad C_{out} \leftarrow T$
11 **end**

E. BUTTERFLY OPERATIONS

In the context of Number Theoretic Transform (NTT), butterfly operations play a crucial role in transforming input data into its NTT-domain representation and vice-versa. These operations are analogous to those in the Fast Fourier Transform (FFT) and involve combining pairs of elements from different *butterfly wings* (due to graphical appearance) [63] of the transform array during each recursive partition of the input while transforming input elements to output. For the forward transform direction, a specialized

variant of the butterfly operation known as Cooley-Tukey (CT) (Algorithm 5) is employed, while for the inverse direction, the Gantleman-Sande (GS) (Algorithm 6) variant is used. Both CT and GS primarily differ in the order in which they decompose and reconstruct transformed data based on the direction of the transform. At a high level typical butterfly operation performs modular- addition, subtraction and multiplication on the input pairs, such that each stage doubles the size of butterfly wings in the dataflow graph until the end of tranform. Furthermore, the butterfly operations in NTT employ a precomputed scaling value typically known as the twiddle factor ψ, which is in fact the *kth* power of the *nth* root of unity employed in these transforms. In most cases, a precomputed table of twiddle factors is leveraged [26] to facilitate the efficiency of these butterfly operations. Last but not least, for an input size n elements, usually $n \log_2 n$ butterfly operations take place during one complete transform.

F. TWO ALGORITHMS FOR NUMBER THEORETIC TRANSFORM (NTT)

In this section, we provide a brief overview of the radix2-CT (radix-2 Cooley-Tukey) and 4-Step algorithms for computing the NTT, typically used in CPU implementations. For brevity, we focus on the forward transform, noting that the inverse transform follows a similar process. Interested readers can find a detailed account in [64].

1) RADIX2-CT ALGORITHM

The *radix2-CT* algorithm (Algorithm 4) is the classical iterative method which processes the input elements of vector in $\log_2 n$ outer loop iterations, which execute sequentially (see *do-while* loop, line 4).

Two types of butterfly operations.

In each outer iteration $n/2$ *butterfly* operations are carried out. More precisely, a butterfly operation (*CT* line 8) processes two input elements from vector a_j, a_{j+t} and a twiddle factor Ψ integer, computes $a_j \leftarrow (a_j + \Psi a_{j+t})$ and $a_{j+t} \leftarrow (a_j - \Psi a_{j+t})$. To this end, the access patterns for read and write operations change in every outer iteration by the value of t. Depending on the iteration and the segment of vector being processed, different powers of the primitive root of unity *psi* (also known as twiddles) are employed in butterfly operations. Generally, the twiddle factors are precomputed and stored in a separate vector (Ψ_{br}), typically indexed in a bit-reversed order. Bit-reversing facilitates the data access for the algorithm in the correct order, enabling inplace computation to optimizing memory access patterns and improving overall efficiency.

2) 4-STEP NTT ALGORITHM

The 4-Step algorithm, first introduced in a classic paper [65], was designed to efficiently compute the FFT transform when the input size n is a composite number, $n = n1 \times n2$. In our work, we adapted this algorithm to compute the NTT transform (Algorithm 7) and subsequently implemented it on GPU, as NTT is a special form of FFT.

The 4-Step NTT algorithm essentially breaks down the larger transform into smaller transforms of sizes $n1$ and $n2$ (lines 8, 17), allowing for flexibility in choosing the specific **NTT** implementation variant for each, such as *naive*, *radix2-CT*, or even another 4-Step. When a 4-Step NTT is applied within a top-level 4-Step, this recursive setup is termed a 7-Step NTT. Conversely, using naive or radix2-CT methods within a 4-Step configuration yields what can be considered a hybrid algorithm. However, for clarity, we refrain from using the term *hybrid*, as we consider the overarching 4-Step algorithm a unique approach, independent of the sub-NTT variant. In our implementation,

Algorithm 4 Radix2-CT NTT (Forward)

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 ← n;
2  m ← 1;
3  while m < n do
4      for i from 0 by 1 to m do
5          j1 ← 2it;
6          j2 ← j1 + t − 1;
7          for j from j1 by 1 to j2 + 1 do
8              a_j, a_{j+t} ← CT(a_j, a_{j+t}, Ψ_br[m + i], q)
9          end
10     end
11     m ← 2 × m
12 end
13 return a
```

Algorithm 5 Cooley-Tukey *(CT)* Butterfly

input : u, v, Ψ, q
output: $\bar{u}, \bar{v}$

```
1  ū ← u + (v × Ψ) (mod q);
2  v̄ ← u − (v × Ψ) (mod q);
3  return ū, v̄
```

Algorithm 6 Gentleman-Sande *(GS)* Butterfly

input : $\bar{u}, \bar{v}, \Psi, q$
output: u, v

```
1  u ← ū + v̄ (mod q);
2  v ← (ū − v̄) × Ψ (mod q);
3  return u, v
```

Algorithm 7 4-Step 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** $0 < k \leq n_1 - 1$;
(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: $\boldsymbol{a} \leftarrow NTT(\boldsymbol{a})$ in bit-reversed order

1 **for** *i from* 0 *by* 1 *to* n_1 **do**
2 **for** *j from* 0 *by* 1 *to* n_2 **do**
3 $B_{i,j} \leftarrow a_{i\times n_2+j}$
4 **end**
5 **end**
6 $B = B^T$ *(Transpose operation)*
7 **for** *j from* 0 *by* 1 *to* n_2 **do**
8 $B_j \leftarrow \boldsymbol{NTT}(B_j, \Omega_{0_{br}}, n_1, q)$
9 **end**
10 $B = B^T$ *(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**
15 **end**
16 **for** *j from* 0 *by* 1 *to* n_2 **do**
17 $B_i \leftarrow \boldsymbol{NTT}(B_i, \Omega_{1_{br}}, n_2, q)$
18 **end**
19 $B = B^T$ *(Transpose operation)*
20 **for** *j from* 0 *by* 1 *to* n_2 **do**
21 **for** *i from* 0 *by* 1 *to* n_1 **do**
22 $a_{j\times n_1+i} \leftarrow B_{j,i}$
23 **end**
24 **end**
25 ***return*** *A*

we tailor the radix2-CT approach to carryout these inner transforms.

Broadly, 4-Step approach in the first stage arranges the input vector into two-dimensional n_1-by-n_2 matrix, where $n = n_1 \times n_2$ (line 1-5). In the second stage, it performs n_1-point NTT operations for each of the n_2 rows (lines 7-9). In the third stage, the matrix elements are multiplied by twiddles (specific powers of the primitive root of unity), ψ (lines 11-15). Finally, the algorithm performs n_1-point NTT operations for each of the n_2 (lines 16-18).

The 4-Step algorithm performs smaller, independent n_1-point and n_2-point NTTs compared to the n-point NTT operations as in the radix2-CT algorithm, thereby it better suits for a parallel processing and memory access locality point of view. However, the transpose operations (lines 6, 10, and 19) can be a hurdle, as they may lead to fragmented memory access, which is disruptive for threads on GPU.

IV. GPU PROGRAMMING MODEL

A GPU contains an array of Streaming Multiprocessors (SMs), where each SM consist of a fixed number of N Streaming Cores (SCs), that manage and execute computation *threads*. Besides inter-generational differences, GPU cards also vary in the number of SMs. Increasing the number of SMs provides horizontal scaling to the GPU device, enabling it to process more tasks simultaneously or complete a given task more quickly, provided the task fully utilizes the parallelism capabilities of the GPU.

Typically, threads execute in groups of 32 called *warps*, which start together but can execute independently. Warps execute concurrently on SMs, allowing multiple groups of threads to run in parallel, thus maximizing the computational efficiency and acheivable throughput. A thread block (or simply *block*) can typically contain up to 1024 threads and be logically organized into one, two, or three dimensions for indexing convenience. All threads in a block run on the same SM, accessing the same shared memory, and an SM can execute multiple blocks [66].

A *grid* is formed by combining multiple blocks, each containing the same number of threads. Since a block's thread count is limited, grids allow for running more threads in parallel than a single block can accommodate. Different blocks in a grid may run on different SMs, so threads in different blocks do not share the same memory. Akin to blocks, a grid can be indexed in one, two, or three dimensions, referred to as its *shape*.

A *kernel* is a function executed on a GPU, taking the number of threads and blocks as arguments. When threads from different blocks need to synchronize, the current kernel must terminate, and another must start which usually incurrs significant timing overhead [67]. From a practical standpoint, one must refrain from excessive and premature kernel executions in succession with suboptimal block-and-grid sizes, as kernel-launches incur significant overhead. Wherever possible, the execution of the current kernel should be reused. This approach is particularly suitable for algorithms involving partition-and-merge steps.

A. GPU MEMORY HIERARCHY

The memory system of a GPU is hierarchical, designed to optimize the performance of parallel tasks. It broadly includes three types of memory: *global*, *shared*, and *local*. Global memory, the largest in size, is implemented as an off-chip DDR array. Consequently, its access latency is higher compared to other memory types. However, data in global memory can be shared across kernels and threads regardless

of their block. On the other hand shared memory is smaller but faster parallel data cache, typically measured in kilobytes per SM, is shared by threads within the same thread block, enabling threads to communicate, synchronize, and store reusable data thereby, reducing global memory latency [68]. Local memory is the smallest which is specific to each thread and used as scratchpad memory to store automatic variables.

B. PRACTICING OPTIMIZED MEMORY ACCESS AND OCCUPANCY

In our implementation, we optimize memory access patterns through aligning data and thread operations with the warp size and adhere to memory alignment boundaries. This alignment ensures that threads within a warp can access contiguous data-elements whenever possible, facilitating coalesced access. Coalesced access significantly reduces global memory latency and enhances its bandwidth utilization. The key reason behind this improvement is that when global memory is accessed, a full memory block (typically 128B) is fetched and brought into shared memory. If a thread block reads or writes consecutive elements in a full block, this reduces fragmentation in global memory as a side-effect. In contrast, fragmented access to global memory can drastically impacts overall performance.

Furthermore, the degree of GPU utilization plays a crucial role in overall computation throughput. Utilization is measured in terms of occupancy, where low occupancy indicates under-utilization of hardware resources and viceversa. Occupancy is defined as the ratio between the number of active warps actually executing and the maximum possible number of active warps on a SM. To this end, profiling data from a specific implementation reveals valuable occupancy information, dictating further development efforts to maximize GPU utilization, if needed. This optimization is essential for leveraging the full computational power of the GPU at hand.

V. TWO NTT ALGORITHMS FOR GPU

In this section, we discuss the internals of the proposed algorithms and their GPU implementation. The algorithms are *radix2-CT* and *4-Step* NTT. Furthermore, we explain the rationale behind specific design choices made from an optimization perspective, which are largely influenced by the microarchitectural features of the underlying GPU devices we use. In the following, we present the detailed account of each algorithm.

A. RADIX2-CT NTT GPU ALGORITHM & ITS IMPLEMENTATION

The radix2-CT NTT algorithm for GPU operates in two parts: i) *NTT host*, which runs on the host device (i.e., general-purpose CPU). This part determines the optimal thread block size, block and grid shape, and the number of kernels, and ii) *NTT kernel*, which performs the actual NTT computation on the GPU based on the launch-time kernel parameters from the first step.

As observed in Algorithm 4, broadly it comprises on an *outer* do-while and an *inner* for loop. At the beginning, during the first outer loop iteration, one NTT operation is carried out on the entirety of the length of input vector. However, in subsequent iterations the number of *independent* NTTs double from one iteration to the next, operating on separate parts of input vector. For instance, during the second iteration, the two independent NTTs are performed on first and second half of the input vector, respectively. For ease of indexing, one can assign indices to each of the individual NTT operations in the outerloop from left-to-right manner.

While the outer loop needs to be executed sequentially, since a data-dependency exist from one iteration to the next. However, iterations of the inner loop are independent and therby can be parallelized and executed concurrently by thread blocks. The threads of a block can concurrently carryout all iterations of the inner loop and when needed can utilize a builtin intrinsic function (`__syncthreads`) to synchronize across blocks before next iteration of outer loop. However, this technique can be leveraged as long as $bDim \geq n/2$. However, this becomes expensive for the viceversa as it would need more-than-one additional thread blocks for full parallelization of the inner loop, for which the synchronization becomes costier, as the only way for this is via global memory and multiple kernels. Consider the case, when $n = 2^{11}$ there are 1024 CT operations in each inner loop iteration, one block (and one kernel) suffices to acheive the desired NTT, given the fact that the maximum number of threads in CUDA-capable GPUs are 1024.

When $n > 2^{11}$, however, multiple kernels will be needed and the approach adopted in [69] 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 $kc = \log_2 (n/2bDim)$. When the ring dimension increases to 2^{28}, 17 kernels are needed, which renders the approach in [69] prohibitively inefficient for high ring dimensions. The work in [64], 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.

On the contrary, for cases where $n > 2^{11}$, multiple kernels are needed and such is the approach followed in our earlier work [69] which launched a new kernel for each outer iteration, until iteration count reaches to ($\log_2 n - \log_2 2bDim$) but soon in practice runs into limitations and performance bottlenecks if n exceeds 2^{15}, nonetheless one can determine the number of kernels kc needed through ($kc = \log_2 (n/2bDim)$). For a large enough ring dimension $n \leq 2^{28}$ with aforementioned approach 17 kernels would be needed, which is prohibitively expensive. An incremental

Algorithm 8 Radix2-CT NTT Host

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

1: $\{bDim, kc, koc\} \leftarrow$ `Partition`$(n, mbd = 1024)$
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** $kc - 1$
6: $oc \leftarrow oc + koc[i]$;
7: $ko[i] \leftarrow 2^{oc}$;
8: $olc \leftarrow olc - koc[i]$
9: **if** $i = 0$
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**
18: **for** i **from** $kc - 1$ by -1 **to** 0
19: $dim3$ **B**$(kgs[i][0], kgs[i][1])$
20: $dim3$ **T**$(kbs[i][0], kbs[i][1])$
21: **NTT** $\lll$ **B**, **T** $\ggg$ $(A, PsiTable, m, ko[i], koc[i], q)$
22: $m \leftarrow m \times (2^{koc[i]})$
23: **end**

improvment was later introduced in [64] which employed a single kernel but carried out complete ($\log_2 n - \log_2 2bDim$) iterations and thus, saved additional and expensive kernel launches. Still, the resource limitation can only leeway the NTT operation for up to $n = 2^{15}$ due to insufficient registers in the SM to load all vector elements from 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 [69], 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 [64], 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.

For instance, 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 all kernels perform the computations using shared memory in all outer loop iterations.

The proposed method for determining block size, kernel count, and the shapes of the grid and blocks is employed in Algorithm 8. This algorithm runs on the host device first and later invokes the actual GPU kernels for NTT. The function `Partition` (line 1) takes the ring dimension n and the maximum block size for the GPU (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*. The elements of the *koc* array store the number of outer iterations for each corresponding kernel in reverse index order; i.e, $i = 0$ represents the last kernel, while $i = kc - 1$ represents the first kernel. The `Partition` function relies on a lookup table based on empirical assessments for the given input parameter combinations whose further details will be discussed in the subsequent sections. Although, partitioning the outer loop iterations into kernels can be done in several valid ways, and intuitively, one might prefer to perform more iterations in later kernel invocations than in earlier ones. However, in practice, partitioning schemes directly influence memory access performance. Therefore, careful consideration must be given to deciding the optimal partitioning, which is why we relied on empirical investigation.

In Algorithm 8, *bc* stands for the number of blocks in the grid (block count) while 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. The shapes of grids and layout of kernels are determined between lines(5-16) of Algorithm 8. The block and grid shapes, which pertain to their dimensionality, are determined by considering 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, with their elements tracking the dimensions of grids and blocks in each kernel.

The blocks and grid have a cartesian access structure and thus can be organized into different *block groups*. We organize the blocks executing the same NTT operation into the same group and thus facilitate the accessing of relevant range of data elements. The first and second dimensions of *kgs* designate the number of blocks in each group and the number of block groups, respectively. Below, we explore the rationale behind the specific way for kernel launches, including block and thread arrangements, with several examples to illustrate how these choices relate to the input size and facilitate coalesced access to global memory.

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., $kgs[1] = [2048, 4]$. For the final kernel we have $kgs[0] = [1, bc]$.[1]

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 the block shape $kbs = [[1024, 1], [16, 64], [32, 32]]$. Here, in the first iteration 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 may not be achieved due to poor resource utilization as explained in Section IV. For instance, using $bDim = 1024$ turns out to result in only a poor occupancy rate of 66% as a SM in a GPU device with compute capability 8.6 can run maximum of 1534 threads. 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.

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 2, better occupancy rate can lead to more than 10% improvement in execution times for two GPU devices.

TABLE 2. 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 preceding 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 specified. 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 degree.

Using the configuration obtained in Algorithm 8 for block dimension, kernel count, kernel and block shapes, the **NTT Kernel** function in Algorithm 9 is called in Step 20 of Algorithm 8. The index *GAddr* in Algorithm 9 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.

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 3. 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 9).

[1] Cuda Technical Specification provides us with a maximum grid.x, grid.y and grid.z as $(2^{31}-1)$, 65535, and 65535 respectively. Therefore, if the grid.y is higher than 65535 for ring dimension, use a new kernel type that has grid.x and grid.y index switched.

Algorithm 9 Radix2-CT NTT Kernel

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 a NTT
4: $\ell_3 \leftarrow 2 \times ko \times bID.y$ ▷ offset for a 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** $koc - 1$
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**
21: $A[GAddr] \leftarrow$ **SMem**$[SAddr]$
22: $A[GAddr + offset_G] \leftarrow$ **SMem**$[SAddr + offset_S]$

TABLE 3. 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

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 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 4 for the execution of the first three blocks of the first kernel for the first thread groups.

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 9, Step 3)

TABLE 4. 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

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 9, 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.

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 8, 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 IV, 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 5. As can be observed in Table 5, 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.

TABLE 5. 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

B. GPU IMPLEMENTATION OF 4-STEP NTT

In Algorithm 7 given for 4-Step NTT, 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. Finally, the multiplication with twiddle factors in Step 13 of In Algorithm 7 can be incorporated into the second group of NTT operations. With these optimizations, the 4-Step NTT algorithm is simplified to have only two groups of NTT operations and a transpose operation in between.

In both NTT operation groups, there are many independent NTT operations of much smaller sizes (i.e., n_1 and n_2 values) than those used in the radix2-CT 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 4-Step NTT algorithm, it is suggested that $n_1 \approx n_2$, 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 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 group of NTT operations 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 group 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 6 shows the timing results of 4-Step 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 6. Effect of the matrix dimension on the performance of the 4-Step 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 6 capture the effect of transpose operations in performance. For instance, in cases 1 and 2, the first and last transpose operations explained in Algorithm 7 are included in timings. In case 1, all three transpose operations described in Algorithm 7 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 GPU-B (A 100) and GPU-C (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 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 7 contains the matrix dimensions for all ring dimensions of interest, which is found to give the best performance in each case experimentally.

TABLE 7. Matrix Sizes for 4-Step NTT Implementation.

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

VI. EXPERIMENTAL SETUP AND RESULTS

In this section, we describe the experiments we conducted to empirically evaluate our GPU implementation of the 4-Step and radix2-CT NTT algorithms and report the results. We begin by describing our experimental setup, followed by various facets of our experimentation, which include: comparing the latency and throughput of both algorithms across subject GPU cards; evaluating performance against existing works for small ring dimensions, and against a commercial-grade implementation for large ring dimensions; and finally assessing the impact of different modular reduction methods on the performance of both algorithms.

TABLE 8. Overview of the three models of GPU- A, B, C, we employed for experimentation, together with their featured specifications. The actual model number of each card is boldcased to standout for the reader.

Feature	GPU-A	GPU-B	GPU-C
Model	**RTX 3060Ti**	**A100**	**RTX 4090**
Architecture Family	Ampere	Ampere	Ada
Release Date	12'2020	05'2020	09'2022
Cuda Cores (i.e. Num. Threads)	4864	6912	16384
Boost Clock (GHz)	1.66	1.41	2.52
Memory Size (GB)	8	80	24
GPU Memory Bandwidth (GB/s)	448	1935	1008
Memory Bus Size (bit)	256	5120	384
Thermal Power (W)	220	300	450
Supported PCIe (gen)	4.0	4.0	4.0
Target Market	Consumer	Industrial	Consumer
Target Usecase	Gaming	HPC	Gaming/HPC

A. SETUP

To conduct experiments, we utilized the following hardware and software setup. We carried out all experiments on a Linux (Debian distribution) machine equipped with an AMD Ryzen 3955WX CPU, 64GB of main memory, running kernel version 5.15.0, and utilizing CUDA SDK version 12.1. The mainboard provides a number of PCIe slots for the pluggable insertion of a GPU card of choice. It was our intention to evaluate our implementation on as many relevant GPU cards as possible; however, acquiring a broad range of cards was infeasible primarily due to availability and budget constraints. Nonetheless, we settled for the following three different GPU cards, whose details and notable features are listed in Table 8.

A quick glance through Table 8 establishes the fact that our choice of these cards is reasonable within the latest trends, as they span facets such as the target market ranging from typical to high-end consumers, as well as their suitability for moderate to large-scale scientific research. Moreover, the capabilities of these cards range from 4k to 18k CUDA high-performance cores, with onboard memory ranging from 8 to 40 GB, and PCIe generation 4.0 powered host-to-GPU data transfer capabilities. As stated earlier, achieving exhaustive coverage of GPU cards would not be possible; however, we believe that our evaluation across the triad of these cards covers ample ground to propose experimentally optimal parameters for a performant NTT within the scope of GPU implementation. For readers seeking optimal NTT parameters for similar purposes but utilizing a different GPU card than ours, we suggest referencing the results of the card closest in capability to their own.

B. RESULTS

In this section, we present the GPU implementation results of the 4-Step and radix2-CT NTT algorithms on the experimental setup described in the preceding section. Here we only report results for the forward direction of NTT transform and exclude the results of inverse direction, since from computation perspective the transform direction is insignificant and the results are almost identical.

In our software implementation of both algorithms, three different modular reduction methods which are selectable at compile time. Moreover, the implementation of reduction methods was done in CUDA PTX assembly for performance reasons, in accordance with similar works. It is notable that Barret and Plantard reduction can work with any NTT-friendly prime whereas, for Goldilocks reduction we employed $2^{64} - 2^{32} + 1$ 64-bit prime, whose rationale is already discussed in (Section III-D).

To measure elapsed time for performance evaluation we record the time spent during the execution of a transform job on the actual device for a given ring dimension n, the subject algorithm and the underlying GPU card. The time measurement mechanism we employ is the built-in `cudaEventRecord` method from the CUDA SDK, which is part of APIs for profiling different aspects of CUDA code execution. Each experimental scenario is repeated between 50 to 100 times, and mean values are reported.

C. PERFORMANCE COMPARISON BETWEEN RADIX2-CT AND 4-STEP IMPLEMENTATION

In this section, we compare the *radix2-CT* and *4-Step* NTT implementation for GPU, both in terms of both latency and throughput. The latency results of a single NTT for both algorithms are numerically given in Table 9 across the three subject GPUs cards (Table 8), and the timings which standout are in boldcased. The trend in the results is graphically visualized in Figure 1.As can be observed that, the timings

of the two algorithms are generally close to each other albeit with minor differences for same n and increases non linearly across increasing n.

As can be observed from Table 9, the timings of the two algorithms are generally close to each other. For relatively small values of n, the radix2-CT algorithm tend to perform better than the 4-Step algorithm. However, for very large values of n, 4-Step 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 pays off. Another observation from the results is that the architectural differences can come into play to alter performance to a certain extent. For intance, while radix2-CT is slower than the 4-Step on GPU-C (RTX 4090) for $n = 2^{20}$, whereas, the opposite is true for GPU-B (A100) considering from specification perspective both of them stand nearby, same observation is true for GPU-A.

Overall, for a highlevel numerical summary which is agnostic to underlying GPU card, both radix2-CT and 4-Step algorithms on average exhibit $9.5\mu s$ and $9.2\mu s$ for $n = 2^{12}$; $28.5\mu s$ and $28.3\mu s$ for $n = 2^{18}$; and $1873.5\mu s$ and $1499.6\mu s$ for $n = 2^{24}$ for a single transform across the evaluated parameter space.

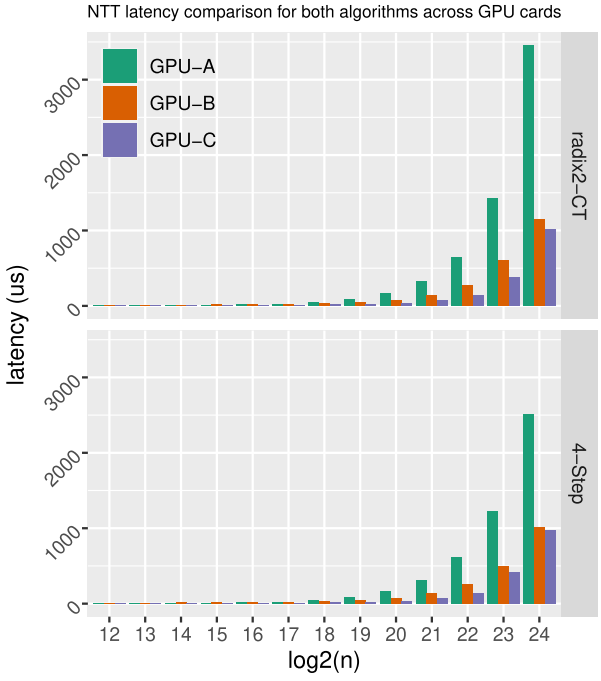


FIGURE 1. Latency (μs) comparison across GPU cards for both algorithms for varying ring dimensions.

Although, the latency figures may provide a quick glance on the computational performance however, for certain practical applications throughput figures may be more relevant as one may desire to carry out many independent NTTs concurrently in a batch. To this end, we employed five different sized of (i.e, 4, 16, 32, 64, 128) transforms to study the batching behavior of both algorithms for n ranging between $2^{12} - 2^{16}$.

TABLE 9. Comparison of performance via mean execution times (μs) for single forward NTT operations in both radix2-CT and 4-Step implementations, across varying ring sizes and three distinct GPU card models (Table 8). Highlighted in bold are the superior timings within each row. These experiments utilize a 64-bit Goldilocks prime modulus for modular reduction operations.

	radix2-CT			*4-Step*		
$log_2(n)$	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**

It is evident from Table 10 and Figure 2 that the employed GPUs are capable of computing concurrent NTTs without significant increase in latency as long as available resources on the GPU are available and not completely exhausted. However, past that point the NTTs are serialized and this reflects in the observed latency and througput figures. For instance, on GPU-C, when $n = 2^{14}$ while 4 NTT operations in a batch take 8.04 μs, 16 of them take only slightly more time, 11.48 μs with radix2-CT, similarly 4-Step takes 4.55 μs and 11.70 μs. In contrast, when $n = 2^{16}$ while 64 NTT operations in a batch take 97.16 μs and 128 take 184.8 μs with radix2-CT, which almost doubles, similarly 4-Step takes 91.00 μs and 192.16 μs following the same trend reflecting the available GPU resources getting into contention. Although, it should be kept in mind that choosing a suitable n and an practical batch size for purpose of maximal throughput is largely dictated by the resources of the GPU card, its capabilities and is often a decision based on tradeoff.

Lastly, from a numerical standpoint, making a clear distinction between the superiority of radix2-CT and 4-Step is challenging, as their performance varies and is sensitive to the underlying GPU architecture. However, as observed in Table 10, radix2-CT in batched operations tends to perform more consistently and is less sensitive to architectural differences across GPUs. This stability enables a more direct and meaningful comparison with existing studies (Table 11). Batch NTT results are generally more representative for benchmarking, especially for ring dimensions ($2^{12} - 2^{16}$), where performing multiple batch NTTs is often preferable to a single NTT—a practice well established in existing literature.

D. COMPARISON WITH THE LITERATURE

From the existing literature on GPU-accelerated NTT, a few works comparable to ours [64], [72], [74], [75] are

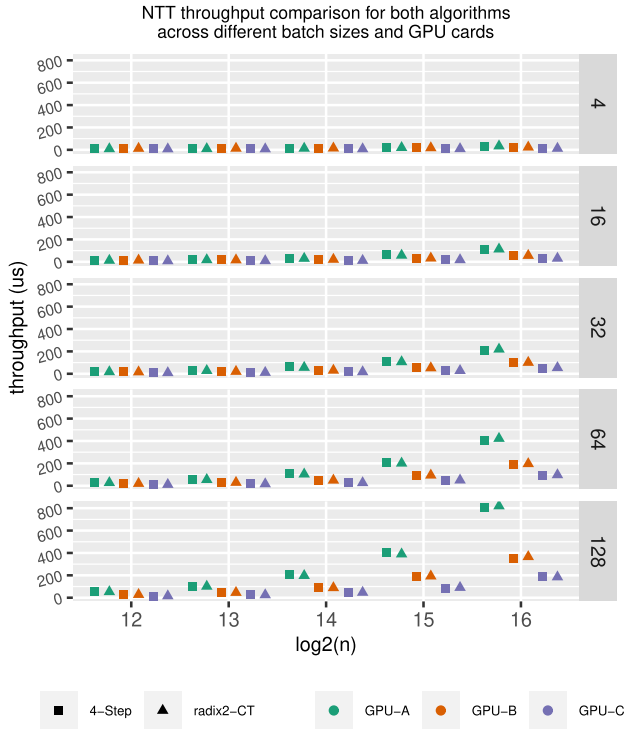


FIGURE 2. NTT throughput comparison across 5 different batch sizes for both algorithms across three GPU cards.

TABLE 10. Timings of GPU implementations of Batch Forward NTT in μs.

		radix2-CT			*4-Step*		
$log_2(n)$	batch	GPU-A	GPU-B	GPU-C	GPU-A	GPU-B	GPU-C
12	4	8.67	**12.24**	**7.59**	8.67	12.98	7.72
	16	**12.27**	**13.91**	**7.86**	12.63	14.20	8.14
	32	**17.09**	**15.65**	**8.54**	17.41	16.98	9.25
	64	**26.90**	**19.63**	10.67	28.69	20.91	10.68
	128	**53.04**	**27.85**	15.62	54.71	28.97	**15.25**
13	4	**9.45**	**13.25**	**7.65**	9.82	13.66	8.04
	16	**17.52**	**16.20**	**8.80**	18.53	17.14	9.51
	32	**28.00**	**20.22**	**11.08**	29.24	21.73	11.36
	64	**54.41**	**29.10**	16.05	57.89	30.48	**15.92**
	128	**101.53**	**46.07**	24.89	104.55	48.11	**24.72**
14	4	**13.24**	**14.94**	**8.04**	13.70	15.10	8.55
	16	**29.80**	**21.10**	**11.48**	31.90	22.34	11.70
	32	**56.19**	**30.63**	16.82	60.98	31.76	16.80
	64	**103.47**	**48.90**	26.36	110.90	51.29	26.32
	128	**197.25**	**88.52**	47.11	209.85	90.58	**45.16**
15	4	**19.41**	**17.26**	**9.19**	20.46	18.59	9.86
	16	**57.71**	**32.26**	17.55	62.48	33.21	**17.31**
	32	**106.48**	**51.85**	27.81	112.25	53.47	**27.41**
	64	**200.83**	**93.86**	49.83	211.89	94.58	**47.72**
	128	**389.93**	192.83	90.66	407.62	**185.17**	**85.48**
16	4	**33.64**	**23.26**	12.46	34.28	24.30	12.49
	16	113.16	**55.92**	30.76	**112.93**	56.30	**28.56**
	32	**219.09**	100.77	53.85	211.73	**99.30**	**49.66**
	64	422.00	196.43	97.16	**406.88**	**188.83**	**91.00**
	128	819.44	366.00	**184.86**	**803.18**	**354.03**	192.16

discussed in (Section II), which primarily address smaller rings. However, to the best of our knowledge, we have not encountered studies addressing larger rings, and thus, a comparison with our work is not straightforward. However, we use our informed judgment to strive for a reasonable comparison. Notably, most existing works utilize small n, limiting their results. Although we compare our results with such works for comparable n, for larger ring dimensions ($n \geq 2^{19}$ to 2^{28}), we compare against an existing open-source, commercial-grade implementation known as SPPARK [54], as a fair competitor. Moreover, in our assessment, the radix2-CT algorithm is a suitable candidate for comparison due to its classical heritage. For modular reduction, we use 60-bit random primes with the Plantard method and a Goldilocks reduction with 64-bit prime in following set of experiments.

We report latency (μs) results in Table 11 for ($n \leq 2^{17}$) and a high level summary comparison for each n, we pick the minimum value both from results of existing work and this work category from Table 11 and observe a 21% overall speedup respectively from follows: 17.3% (9.2 vs 7.60 for $n = 2^{12}$); 20.3% (9.6 vs 7.65 for $n = 2^{13}$); 30.0% (11.1 vs 7.77 for $n = 2^{14}$); 29.56% (11.4 vs 8.20 for $n = 2^{15}$); 22.68% (11.2 vs 8.66 for $n = 2^{16}$); and 7.8% (12.2 vs 11.7 for $n = 2^{17}$).

TABLE 11. Latency (μs) comparison with existing literature for $n \leq 2^{17}$.

			$\log_2 n$					
Work	Device	$\log_2 q$	12	13	14	15	16	17
[72]	Titan V	60	-	-	44.1	84.2	-	-
[73]	RTX 2080 Ti	64*	-	-	-	83.3	-	-
[74]	GTX 1070	64*	-	-	57.8	-	-	-
[75]	GTX 1070	64*	-	-	66.8	-	-	-
[70]	V 100	55	-	-	29	39	-	-
[71]	RTX 3060 Ti	62	14.0	14.9	19.1	35.9	-	-
[60]	A 100	62	-	-	13.3	-	16.5	-
[60]	V 100	62	-	-	11.5	-	16.4	-
[79]	GPU-A	60	11.8	11.8	13.6	14.6	16.9	23.3
[79]	GPU-C	60	9.2	9.6	11.1	11.4	11.2	12.2
[78]	GPU-A	60	14.0	10.9	13.7	15.2	16.4	19.7
[78]	GPU-C	60	12.3	11.4	15.1	15.4	15.5	16.7
T.W.	GPU-A	64*	8.32	8.43	9.15	10.49	14.79	27.8
T.W.	GPU-B	64*	12.57	12.94	12.97	13.84	15.93	-
T.W.	GPU-C	64*	7.64	7.74	7.78	8.03	8.66	11.7
T.W.	GPU-A	60	8.45	8.67	9.38	11.45	15.00	24.7
T.W.	GPU-B	60	12.72	13.22	13.27	14.56	16.15	21.9
T.W.	GPU-C	60	7.60	7.65	7.77	8.20	8.86	11.8

*: uses the 64-bit Goldilocks prime; **T.W.**: This Work uses *radix2-CT* algorithm

For high ring dimensions ($2^{19} \leq n \leq 2^{28}$), we evaluate the radix2-CT algorithm and the SPPARK implementation on GPU-B and GPU-C, omitting GPU-A due to its early resource exhaustion for larger n in the experiments. The latency (μs) results are presented in Table 12, and the trends are visualized in Figure 3. A quick glance reveals that our implementation outperforms SPPARK on both GPUs across stated n values, with average improvements of 6.9%/30.8% (min/max) on GPU-B and 3.8%/25.6% (min/max) on GPU-C, respectively. To maintain a level playing field, we employed the Montgomery reduction routine in this half of experiments, as SPPARK exclusively uses it due

to its design specifications for supporting zk-SNARKs and working with very large moduli, such as the BLS12-377, which is 377 bits in length.

Among other relevant works [76], [77], [78], [79] where a direct comparison is non-trivial, we make our best effort to provide a fair comparison of numerical results. Regarding TensorFHE [79], it reports a throughput of 1, 764, 953 NTT operations per second across $n \in \{2^{12}, 2^{13}, 2^{14}\}$ on GPU-B type platform, on average. In our comparable experimental setting, we achieve an average throughput of 2,930,746 NTT operations per second on GPU-B using the radix2-CT algorithm (Table 10). Flexible-GPU [77] reports a latency of 315 μs for $n = 2^{16}$. In comparison, our approach, averaged across both algorithms, reports a latency of 13 μs, reflecting a significant comparative performance. Lastly, HiKyber [78] and Efficient-TMVP [76] report notable latency figures of 1.2 and 0.5 μs for $n = 2^8$ on GPU. However, our work neither targets nor is designed for such small n values, as our primary focus remains on significantly larger sized transforms, exclusively utilizing CUDA cores on the GPU.

All in all, we observe that our obtained comparative results are encouraging and support avenues for optimization and performance gains even further.

TABLE 12. Latency (μs) of radix2-CT algorithm against SPPARK on GPU-B and GPU-C using BLS12-377 prime modulus).

	radix2-CT			
	GPU-B		GPU-C	
$log_2(n)$	T.W.	SPPARK [54]	T.W.	SPPARK [54]
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	**15251.5**	16952.4	**8620.7**	9131.73
26	**31388.1**	38815.0	**17728.4**	20213.7
27	**65097.4**	74886.3	**36780.5**	38177.9
28	**137242.0**	179563.0	**78886.7**	82987.3

T.W.: This Work

E. IMPACT BY MODULAR REDUCTION TYPE

In this set of experiments, we aimed to determine whether a particular reduction method significantly impacts the overall performance of our implementation. As discussed in Section III-D, we employed Plantard, Barrett, and Goldilocks reduction methods, outlining their distinctions concerning the subject modulus value q. To this end, we selected the radix2-CT algorithm and configured it at compile time for each of the three reduction types. We then evaluated it across typical ring dimensions ($2^{12} \leq n \leq 2^{24}$) with q of at most 64-bits. As with previous experiments, these tests were conducted on all three GPUs (Table 8).

As before, we measured the latency, with the results presented in Table 13 and visualized in Figure 4. Overall,

FIGURE 3. Latency (μs) comparison against SPPARK on GPU-B and GPU-C.

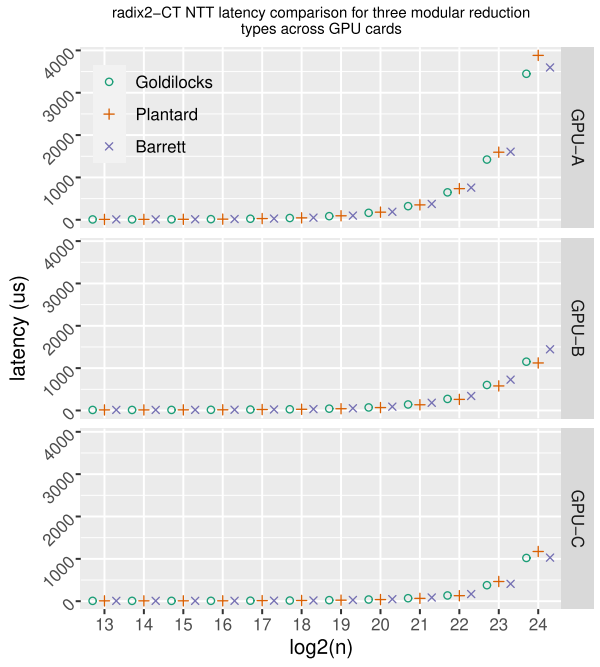


FIGURE 4. Latency (μs) comparison of radix2-CT algorithm employing three modular reduction types.

we observe that all three modular reduction methods exhibited similar performance, with none showing a significant advantage over the others in terms of latency. For instance $n = 2^{24}$ Goldilocks reduction outperforms the other methods meagerly on GPU-A, but Plantard method remains performant on GPU-B, and Barret on GPU-C marginally.

While the Plantard reduction algorithm saves one multiplication in comparison with the Barret reduction, the input size of the former algorithm is larger than the latter. Thus,

TABLE 13. latency (μs) comparison of radix2-CT algorithm employing three modular reduction types and evaluated across three GPUs.

	GPU-A	GPU-B	GPU-C
$log_2 n$	† / ‡ / ♠	† / ‡ / ♠	† / ‡ / ♠
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

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

the Plantard algorithm can outperform the Barret algorithm only if the GPU platform has good memory access latency. This can be observed in Figure 4, where the Plantard outperforms the other reduction algorithms on GPU-B, which is a high memory bandwidth device. On the other hand, on low memory bandwidth GPU-A, the Plantard method did not outperform the other two for the aforementioned reason.

In summary, the choice of modular reduction method alone is not always a decisive factor in the overall performance of the NTT algorithm. It may also depend on the performance characteristics of the underlying compute platform such as memory bandwidth and clock speed.

F. FURTHER DISCUSSION

Our experiments in the preceding sections also highlight the role of GPU clock frequency on performance, as expected. For example, in Section VI-C, where $n = 12 - 14$, we found that both of our algorithms performed better on GPU-A compared to GPU-B. This is because GPU-A, despite having fewer resources, operates at a higher clock frequency. When comparing two GPUs with similar resource availability for a given n, the one with the higher clock frequency typically delivers greater throughput and lower latency (see Table 9 and Table 10).

However, clock frequency increases are limited by the physical constraints of the hardware. Beyond a certain point, further improvements in performance require horizontal scaling, i.e., increasing the number of resources. For instance, GPU-C, which has a lesser memory bandwidth and but way more compute threads in comparison to GPU-B, outperforms GPU-B despite both being industrial-grade workload capable. This demonstrates that additional resources can play a crucial role in performance when clock frequency enhancements reach their limits as evident from Table 12.

1) PERFORMANCE - A MATHEMATICAL TAKE

At a juncture, a reader may seek mathematical model entailing the performance of the radix2-CT and the 4-Step algorithms, which require $n/2 \log n$, and $n/2 \log n + n$ modular multiplications, respectively, where modular multiplication dominates the compute-bound operations. Despite the 4-Step algorithm is more involved computationally, it is not usually the slower of the two algorithms. This and the high input sizes imply that NTT is of more memory-bound than compute-bound especially for large n memory-bound and one needs a model that captures the memory access efficiency of the two algorithms to better evaluate the implementation results.

To this end, in the following we outline such a model. However one should remain wary, that developing an exact mathematical model is only approximate without taking into account all the latent microarchitectural behavior of the underlying hardware especially, the distributed GPU memory hierarchy. This prime limitation stems from the lack of availability of an existing formal model in the literature that accurately represent the behavior of global, shared, and thread memory for modern NVIDIA GPUs, among proprietary implementation details of the GPU vendor. Nevertheless, we attempt to describe the performance behavior of presented algorithms based on performance metrics [80], which are reported by using Nvidia's profiling tool `nsight nsys` as part of SDK.

Among more than 100 available profile metrics, we consider the following ones which relate to *memory efficiency* i.e. *GLE, GSE* - global memory (l)oad/(s)tore efficiency; *GMT* - global memory throughput; *SLE, SSE* - shared memory (l)oad/(s)tore efficiency; *SME* - overall shared memory efficiency; *SMT* - shared memory throughput. The efficiency metrics are measured in percentage and, throughput ones in GB/s. Although a single *SME* metric exists for shared memory, there is no direct equivalent

for global memory. To establish a *GME* (*Global Memory Efficiency*) metric, we compute a weighted average of the global memory load efficiency (*GLE*) and global memory store efficiency (*GSE*). The weights for this average may be equal or differ depending on the specifics of the underlying platform. For our evaluation, we considered them to have equal weightage.

The performance for both algorithms is outlined in expression (2) as proportional relation, where T represents the total time taken, n the input polynomial size, and two constants tg and ts as typical (g)lobal and (s)hared memory access overhead which are specific to GPU card model, and kc the number of kernels invoked, as more kernels increase overhead due to the launch associated overhead.

$$T \propto t_g \cdot kc \cdot \left(\frac{2n}{GMT \times GME}\right) + t_s \cdot \left(\frac{n \log n}{SMT \times SME}\right) \quad (2)$$

Typically, t_g is much greater than t_s, with estimates suggesting a ratio of 20 : 1 [73], implying that fragmented access of global memory significantly deteriorates overall performance. Metrics such as global memory throughput (*GMT*) and shared memory throughput (*SMT*) are card-dependent, with higher-end GPUs offering better performance in these areas. The performance P is inversely related to the execution time T, expressed as $P = \frac{1}{T}$.

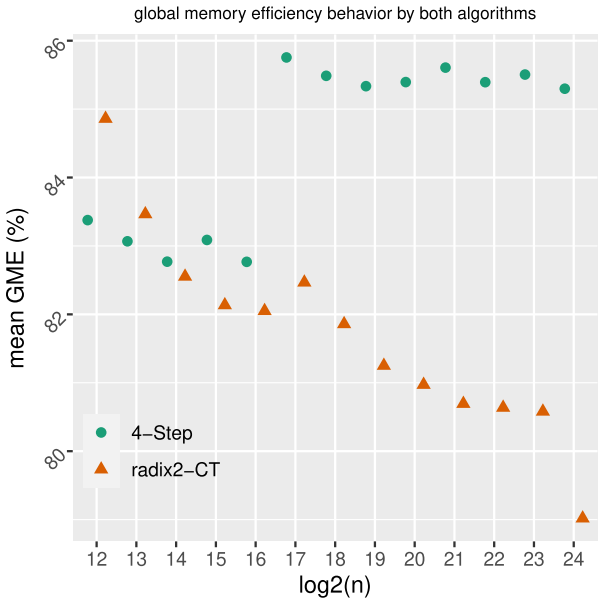


FIGURE 5. Observed mean global memory efficiency (GME)% behavior for both algorithms.

The performance of Algorithm 9, is heavily influenced by how efficiently global memory is utilized through uncoalesced access in relation to n. The first half of expression (2) related to *GME* significantly underlines this effect, as the combined impact of kc and t_g has more weight than the second half. Attributed to the implementation, up to $2n$ global memory loads/stores are performed across the required kc kernels in a single n-point transform. As successive kernels are executed for larger n, global memory access becomes increasingly uncoalesced, progressively degrading *GME* and impacting performance. Moreover, the number of outer iterations in a kernel *koc*, which also increases as n increases, may also reduce *GME* due to uncoalesced access, on average of $\approx 2\%$ with every twofold increase as illustrated in Figure 5.

Conversely, the 4-Step algorithm uses fewer kernels ($k \leq 3$) and leverages shared memory for minimal but coalesced global memory access. This behavior is evident in Figure 5, where *GME* efficiency remains nearly consistent across varying values of n. As discussed earlier (Section V), the input n is decomposed into optimally sized $n_1 \times n_2$ sub-transforms, facilitating shared memory usage while minimizing fragmented access to global memory. The choices of n_1 and n_2 are made to reduce bank conflicts, thereby preventing lower *SME*. Furthermore, selecting n_1 appropriately can eliminate the need for an additional kernel launch dedicated to the mandatory transpose step, allowing for transposition to occur in shared memory by reusing an existing kernel to avoid an additional kernel launch overhead. Overall, the use of fewer kernels and improved *SME* results in better performance. Our evaluations confirm that selecting the optimal n_1 can achieve up to 80% *SME*, while suboptimal choices lead to a 60% drop in efficiency.

Lastly, one might ask how well our performance model explains the actual results of both algorithms. To address this, Figure 1 presents our experimental evaluations across GPU-A, GPU-B, and GPU-C. The 4-Step algorithm generally outperforms the radix2-CT algorithm on GPU-A, which can be attributed to the lower memory bandwidth of GPU-A and confirms our hypothesis that both algorithms are memory-bound and the one with better *GME* usually wins. That the efficiency of the 4-Step algorithm is more pronounced in GPU-A is a further evidence for our hypothesis. On the other hand, for GPU-B and GPU-C, the performance is nearly identical, with only minor differences. A look at (2) shows that, if and only if *GME* to be comparable for both algorithms, performance is affected by *SME*, where 4-Step clearly has the advantage. However, for GPU-B and GPU-C, this distinction is less apparent, likely due to the latent architectural optimizations in these cards, which our model does not account for the reasons outlined earlier in this section. Also, when *GME* and *SME* are comparable, then the modular multiplications can be a determining factor for the performance, where the radix2-CT algorithm has slight advantage over the 4-Step algorithm.

VII. THREATS TO VALIDITY

A notable external threat to the validity of this work lies in the evaluation on the CUDA SDK version and the choice of GPU cards. As with any evolving software system, different versions of the SDK may introduce changes in features, internal optimizations, or even deprecate core platform components, often with unforeseen consequences.

Additionally, a user application built with a certain SDK version may perform inconsistently when evaluated on different hardware. Moreover, macro and microarchitectural differences in GPU hardware and specifications can lead to inconsistent results. To address these concerns, we utilized the latest SDK version available to us and tested on three reputable GPU cards. Although we established a strategy to optimize memory access by prioritizing performance and adjusting configurable parameters, empirically supporting the desired effect, however, remains the possibility of needing to fine-tune these parameters even further based on a weighted approach to the specific hardware and software setup at hand that is different from ours.

Among other external threats, the absence of a standardized benchmark dataset for NTT, and input data distribution, may inadvertently influence NTT performance figures. The absence of a standardized dataset introduces the risk that performance evaluations could vary significantly across studies due to differences in input size, data-distribution, and batching order. While we employed random input values from a uniform distribution, consistent with existing literature, there remains a possibility that specific input distributions such as sparse, semi-structured, or highly correlated data, could result in suboptimal memory access patterns, workload imbalances, or latent impacts on the hardware scheduling. These factors could affect the generalizability of performance results. To address these concerns, exhaustive testing across a wide range of input distributions might provide deeper insights. However, for non-trivial input sizes, such testing is neither practical nor feasible and thus, we chose to rely on random-valued inputs for a given n in this work.

VIII. CONCLUDING REMARKS AND FUTURE WORK

In this work, we present two robust algorithms, namely 4-Step and radix2-CT, to perform the number theoretic transform (NTT) on three well-regarded GPU cards by NVIDIA. The performance of a naive implementation of these algorithms would suffer considerably and would not be desirable. To address this, we utilized a systematic strategy to determine optimal configurable parameters for GPU programming, such as kernel count, thread count, and block and grid shapes. We demonstrate that informed configuration of these parameters significantly impacts the global memory access patterns of the GPU device and plays a vital role in overall performance for non-trivial input sizes. Furthermore, we evaluate and report empirically the latency of a single NTT transform and the practical throughput for batch-mode processing, which is generally sought after in homomorphic encryption as it allows for the concurrent execution of multiple transforms. In contrast to existing works, we demonstrate that our approach can effectively handle large input workloads upto 2^{28}. Additionally, our implementation outperformed an existing third-party alternative for large inputs. We also experimented with three different modular reduction techniques to determine if one performs considerably better than the others. Overall, both of our presented algorithms performed comparably across all three modular reduction methods on all three GPU devices used in the experimentation.

For future work, we plan to experiment with and integrate on-the-fly generation of twiddle factors, rather than precomputing them to reduce memory usage. Additionally, we are considering the exploration and incorporation of high-radix butterfly circuits in future variants of our implementation.

REFERENCES

[1] C. Gentry, "Computing arbitrary functions of encrypted data," *Commun. ACM*, vol. 53, no. 3, pp. 97–105, Mar. 2010.

[2] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Comput. Surveys*, vol. 51, no. 4, pp. 1–35, Jul. 2018.

[3] 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," in *Protecting Privacy Through Homomorphic Encryption*. Cham, Switzerland: Springer, 2021, pp. 31–62.

[4] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems (Extended abstract)," in *Proc. 17th Annu. ACM Symp. Theory Comput.*, R. Sedgewick, Ed., Jan. 1985, pp. 291–304.

[5] D. Bernhard and B. Warinschi, "Cryptographic voting—A gentle introduction," in *Foundations of Security Analysis and Design*, vol. 8604, A. Aldini, J. López, and F. Martinelli, Eds., Cham, Switzerland: Springer, 2013, pp. 167–211.

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

[7] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," *IACR Cryptol. ePrint Arch.*, vol. 2014, p. 349, May 2014.

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

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

[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 *Proc. 3rd Innov. Theor. Comput. Sci. Conf.*, New York, NY, USA, Jan. 2012, pp. 326–349.

[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.*, vol. 2013, 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* (Lecture Notes in Computer Science), vol. 9666, M. Fischlin and J. Coron, Eds., Cham, Switzerland: Springer, 2016, pp. 305–326.

[13] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," *SIAM J. Comput.*, vol. 43, no. 2, pp. 831–871, 2014.

[14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, pp. 1–36, Jul. 2014.

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

[16] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in *Proc. 3rd ACM Workshop Cloud Comput. Secur. Workshop*, New York, NY, USA, Oct. 2011, pp. 1–15.

[17] Microsoft Res., Redmond, WA, USA. (Nov. 2020). *Microsoft SEAL*. [Online]. Available: https://github.com/Microsoft/SEAL

[18] (2021). *PALISADE Lattice Cryptography Library*. [Online]. Available: https://palisade-crypto.org/

[19] S. Halevi and V. Shoup, "Algorithms in helib," in *Advances in Cryptology*, J. A. Garay and R. Gennaro, Eds., Berlin, Germany: Springer, 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," 2022, *arXiv:2202.02960*.

[21] Y. Cui, Y. Zhang, Z. Ni, S. Y. C. Wang, and W. Liu, "High-throughput polynomial multiplier for accelerating saber on FPGA," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 70, no. 9, pp. 3584–3588, Sep. 2023.

[22] A. Skavantzos and T. Stouraitis, "Complex multiplication using the polynomial residue number system," in *Advances in Computing and Control*. Cham, Switzerland: Springer, 2006, pp. 61–70.

[23] W. Liu, S. Fan, A. Khalid, C. Rafferty, and M. O'Neill, "Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA," *IEEE Trans. Very Large Scale Integration (VLSI) Syst.*, vol. 27, no. 10, pp. 2459–2463, Oct. 2019.

[24] Z.-Y. Wong, D. C.-K. Wong, W.-K. Lee, and K.-M. Mok, "High-speed RLWE-oriented polynomial multiplier utilizing Karatsuba algorithm," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 68, no. 6, pp. 2157–2161, Jun. 2021.

[25] I. K. Paksoy and M. Cenk, "TMVP-based multiplication for polynomial quotient rings and application to saber on ARM cortex-M4," *Cryptol. ePrint Arch.*, vol. 2020, p. 1302, Jan. 2020.

[26] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *Proc. 15th Int. Conf. Cryptol. Netw. Secur.*, Milan, Italy. Cham, Switzerland: Springer, Jan. 2016, pp. 124–139.

[27] (2022). *Zprize Accelerating NTT Operations on an FPGA*. [Online]. Available: https://www.zprize.io/prizes/accelerating-ntt-operations-on-an-fpga

[28] J. Woodruff, J. Armengol-Estapé, S. Ainsworth, and M. F. P. O'Boyle, "Bind the gap: Compiling real software to hardware FFT accelerators," in *Proc. 43rd ACM SIGPLAN Int. Conf. Program. Lang. Design Implement.*, Jun. 2022, pp. 687–702.

[29] F. Boemer, S. Kim, G. Seifu, F. D. M. de Souza, and V. Gopal, "Intel HEXL: Accelerating homomorphic encryption with Intel AVX512-IFMA52," in *Proc. 9th Workshop Encrypted Comput. Appl. Homomorphic Cryptography*, Nov. 2021, pp. 57–62.

[30] J. Hoozemans, J. Peltenburg, F. Nonnemacher, A. Hadnagy, Z. Al-Ars, and H. P. Hofstee, "FPGA acceleration for big data analytics: Challenges and opportunities," *IEEE Circuits Syst. Mag.*, vol. 21, no. 2, pp. 30–47, 2nd Quart., 2021.

[31] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS today: Successes, challenges, and opportunities," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, pp. 1–42, Dec. 2022.

[32] P. Hijma, S. Heldens, A. Sclocco, B. van Werkhoven, and H. E. Bal, "Optimization techniques for GPU programming," *ACM Comput. Surveys*, vol. 55, no. 11, pp. 1–81, Mar. 2023, doi: 10.1145/3570638.

[33] (2024). *Two NTT Algorithms for GPU*. [Online]. Available: https://github.com/Alisah-Ozcan/GPU-NTT

[34] Prasetiyo, S. Hong, Y. F. Arthanto, and J.-Y. Kim, "Accelerating deep convolutional neural networks using number theoretic transform," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 70, no. 1, pp. 315–326, Jan. 2023.

[35] D. Strelák, C. Ó. S. Sorzano, J. M. Carazo, and J. Filipovic, "A GPU acceleration of 3-D Fourier reconstruction in cryo-EM," *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 5, pp. 948–959, Sep. 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID

[36] X. Li, Y. Liu, F. Jiang, C. Li, Y. Fu, W. Zhang, and J. Xu, "NEOCNN: NTT-enabled optical convolution neural network accelerator," in *Proc. 38th ACM Int. Conf. Supercomputing*, May 2024, pp. 352–362.

[37] L. He, Y. Zhao, R. Gao, Y. Du, and L. Du, "SFC: Achieve accurate fast convolution under low-precision arithmetic," 2024, *arXiv:2407.02913*.

[38] A. Pedrouzo-Ulloa, J. R. Troncoso-Pastoriza, and F. Pérez-González, "Number theoretic transforms for secure signal processing," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 5, pp. 1125–1140, May 2017.

[39] R. A. Fiorini, "How random is your tomographic noise? A number theoretic transform (NTT) approach," *Fundamenta Informaticae*, vol. 135, nos. 1–2, pp. 135–170, 2014.

[40] J. R. de Oliveira Neto, J. B. Lima, and D. Panario, "The design of a novel multiple-parameter fractional number-theoretic transform and its application to image encryption," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 30, no. 8, pp. 2489–2502, Aug. 2020.

[41] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on GPUs," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2020, pp. 264–275.

[42] J.-Z. Goey, W.-K. Lee, B.-M. Goi, and W.-S. Yap, "Accelerating number theoretic transform in GPU platform for fully homomorphic encryption," *J. Supercomput.*, vol. 77, no. 2, pp. 1455–1474, Feb. 2021.

[43] H. Zhao, D. Ding, F. Wang, P. Hua, N. Wang, Q. Wu, and Z. Chai, "Hardware acceleration of number theoretic transform for zk-SNARK," *Eng. Rep.*, vol. 2023, Feb. 2023, Art. no. e12639.

[44] D. Li, A. Pakala, and K. Yang, "MeNTT: A compact and efficient processing-in-memory number theoretic transform (NTT) accelerator," *IEEE Trans. Very Large Scale Integration (VLSI) Syst.*, vol. 30, no. 5, pp. 579–588, May 2022.

[45] Y. Park, Z. Wang, S. Yoo, and W. D. Lu, "RM-NTT: An RRAM-based compute-in-memory number theoretic transform accelerator," *IEEE J. Explor. Solid-State Comput. Devices Circuits*, vol. 8, pp. 93–101, 2022.

[46] M. Li, H. Geng, M. Niemier, and X. S. Hu, "Accelerating polynomial modular multiplication with crossbar-based compute-in-memory," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Oct. 2023, pp. 1–9.

[47] H. Li, D. Pan, J. Li, and H. Wang, "Parallel accelerating number theoretic transform for bootstrapping on a graphics processing unit," *Mathematics*, vol. 12, no. 3, p. 458, Jan. 2024.

[48] S. Durrani, M. S. Chughtai, M. Hidayetoglu, R. Tahir, A. Dakkak, L. Rauchwerger, F. Zaffar, and W.-m. Hwu, "Accelerating Fourier and number theoretic transforms using tensor cores and warp shuffles," in *Proc. 30th Int. Conf. Parallel Architectures Compilation Techn. (PACT)*, Sep. 2021, pp. 345–355.

[49] B. Li, Y. Yan, Y. Wei, and H. Han, "Scalable and parallel optimization of the number theoretic transform based on FPGA," *IEEE Trans. Very Large Scale Integration Syst.*, vol. 32, no. 2, pp. 291–304, Feb. 2024.

[50] Y. Tian, Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "FPGA acceleration of number theoretic transform," in *Proc. 36th Int. Conf. High Perform. Comput.* Cham, Switzerland: Springer, Jan. 2021, pp. 98–117.

[51] A. Hartshorn, H. Leon, N. Qiao, and S. Weber, "Number theoretic transform (NTT) FPGA accelerator," Worcester Polytech. Inst., Worcester, MA, USA, E-project 051420-162339, 2020, pp. 1–37.

[52] G. Jonatan, H. Cho, H. Son, X. Wu, N. Livesay, E. Mora, K. Shivdikar, J. L. Abellán, A. Joshi, D. Kaeli, and J. Kim, "Scalability limitations of processing-in-memory using real system evaluations," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 52, no. 1, pp. 63–64, Jun. 2024.

[53] S. Datta, R. A. G. Antonio, A. R. S. Ison, and J. M. Rabaey, "A programmable hyper-dimensional processor architecture for human-centric IoT," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 3, pp. 439–452, Sep. 2019.

[54] *Supranational's Performance Primitives for Arguments of Knowledge*. [Online]. Available: https://github.com/supranational/sppark/tree/main

[55] V. Rosenfeld, S. Breß, and V. Markl, "Query processing on heterogeneous CPU/GPU systems," *ACM Comput. Surveys*, vol. 55, no. 1, pp. 1–38, Jan. 2023.

[56] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 4–13, Jan. 2013.

[57] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta, "CellSs: Making it easier to program the cell broadband engine processor," *IBM J. Res. Develop.*, vol. 51, no. 5, pp. 593–604, Sep. 2007.

[58] J. Leong and M.-J. Chen, "Nvidia and the great east-west semiconductor game," *Asian Case Res. J.*, vol. 27, no. 2, pp. 147–174, Jun. 2023.

[59] C. H. Lim, H. S. Hwang, and P. J. Lee, "Fast modular reduction with precomputation," in *Proc. Korea-Jpn. Joint Workshop Inf. Secur. Cryptol.*, 1997, pp. 65–79.

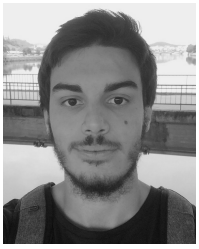
[60] K. S. A. G. Jonatan, E. Mora, N. Livesay, R. Agrawal, A. Joshi, J. L. Abellan, J. Kim, and D. Kaeli, "Accelerating polynomial multiplication for homomorphic encryption on GPUs," in *Proc. IEEE Int. Symp. Secure Private Execution Environ. Design (SEED)*, Los Alamitos, CA, USA, Sep. 2022, pp. 61–72.

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

[62] T. Plantard, "Efficient word size modular arithmetic," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 3, pp. 1506–1518, Jul. 2021.

[63] M. Ayinala, Y. Lao, and K. K. Parhi, "An in-place FFT architecture for real-valued signals," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 60, no. 10, pp. 652–656, Oct. 2013.

[64] A. S. Özcan, C. Ayduman, E. R. Turkoglu, and E. Savas, "Homomorphic encryption on GPU," *IEEE Access*, vol. 11, pp. 84168–84186, 2023.
[65] D. H. Bailey, "FFTs in external or hierarchical memory," in *Proc. Supercomputing : Proceedings ACM/IEEE Conf. Supercomputing*, Nov. 1989, pp. 234–242.
[66] S. Cook, *CUDA Programming: A Developer's Guide To Parallel Computing With GPUs* (Applications of GPU Computing Series). Amsterdam, The Netherlands: Elsevier, 2012. [Online]. Available: https://books.google.com.tr/books?id=EX2LNkSqViUC
[67] S. Kim, C. Jung, and Y. Kim, "Comparative analysis of GPU stream processing between persistent and non-persistent kernels," in *Proc. 13th Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2022, pp. 2330–2332.
[68] *Cuda Memory Hierarchy*. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[69] 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, Feb. 2022.
[70] Z. Zheng, "Encrypted cloud using GPUs," Master's thesis, KU Leuven, Leuven, Belgium, 2020.
[71] W. Dai and B. Sunar, "CuHE: A homomorphic encryption accelerator library," in *Proc. Int. Conf. Cryptography Inf. Secur. Balkans*. Cham, Switzerland: Springer, Jan. 2016, pp. 169–186.
[72] Z. Wang, P. Li, R. Hou, Z. Li, J. Cao, X. Wang, and D. Meng, "HE-booster: An efficient polynomial arithmetic acceleration on GPUs for fully homomorphic encryption," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 4, pp. 1067–1081, Apr. 2023.
[73] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. Hoboken, NJ, USA: Wiley, 2014.
[74] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, pp. 114–148, Aug. 2021.
[75] H. Yang, S. Shen, W. Dai, L. Zhou, Z. Liu, and Y. Zhao, "Phantom: A CUDA-accelerated word-wise homomorphic encryption library," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 5, pp. 4895–4906, Sep. 2024.
[76] M. A. Hafeez, W.-K. Lee, A. Karmakar, and S. O. Hwang, "Efficient TMVP-based polynomial convolution on GPU for post-quantum cryptography targeting IoT applications," *IEEE Internet Things J.*, vol. 11, no. 13, pp. 23428–23443, Jul. 2024.
[77] P. Duong-Ngoc, T. X. Pham, H. Lee, and T. T. Nguyen, "Flexible GPU-based implementation of number theoretic transform for homomorphic encryption," in *Proc. 19th Int. SoC Design Conf. (ISOCC)*, Oct. 2022, pp. 259–260.
[78] X. Ji, J. Dong, T. Deng, P. Zhang, J. Hua, and F. Xiao, "HI-kyber: A novel high-performance implementation scheme of kyber based on GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 6, pp. 877–891, Jun. 2024.
[79] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang, "TensorFHE: Achieving practical computation on encrypted data using GPGPU," in *Proc. IEEE Int. Symp. High-Performance Comput. Archit. (HPCA)*, Feb. 2023, pp. 922–934.
[80] *Cuda Kernel Profile Metrics*. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference



ALISAH OZCAN received the B.S. and M.S. degrees in electronics engineering program from Sabancı University, Istanbul, Türkiye, in 2020 and 2023, respectively. His research interests include RFIC design, homomorphic encryption, lattice-based cryptography, and cryptographic hardware and software design.

ARSALAN JAVEED received the B.S. degree in telecommunication engineering from the National University of Computer and Emerging Sciences, Islamabad, Pakistan, in 2011, and the M.S. and Ph.D. degrees in computer science and engineering from Sabancı University, Istanbul, Türkiye, in 2015 and 2022, respectively. His research interests include software engineering, testing, and security.

ERKAY SAVAS (Member, IEEE) received the B.S. and M.S. degrees in electrical engineering from the Department of Electronics and Communications Engineering, Istanbul Technical University, in 1990 and 1994, respectively, and the Ph.D. degree from the Department of Electrical and Computer Engineering (ECE), Oregon State University, in June 2000. He had worked for various companies and research institutions before he joined Sabancı University, in 2002. His research interests include applied cryptography, data and communication security, privacy in biometrics, security and privacy in data mining applications, embedded systems security, and distributed systems. He is a member of ACM, the IEEE Computer Society, and the International Association of Cryptologic Research (IACR). He is currently an Associate Editor of IEEE TRANSACTIONS ON COMPUTERS and *Journal of Cryptographic Engineering*.