Coala: Coalescion-based Acceleration of Polynomial Multiplication for GPU Execution

Homer Gamil[1], Oleg Mazonka[2], Michail Maniatakos[2]
[1]*NYU Tandon School of Engineering, New York, USA*
[2]*New York University Abu Dhabi, Abu Dhabi, UAE*

***Abstract*—In this study, we introduce `Coala`, a novel framework designed to enhance the performance of finite field transformations for GPU environments. We have developed a GPU-optimized version of the Discrete Galois Transformation (DGT), a variant of the Number Theoretic Transform (NTT). We introduce a novel data access pattern scheme specifically engineered to enable coalesced accesses, significantly enhancing the efficiency of data transfers between global and shared memory. This enhancement not only boosts execution efficiency but also optimizes the interaction with the GPU's memory architecture. Additionally, `Coala` presents a comprehensive framework that optimizes the allocation of computational tasks across the GPU's architecture and execution kernels, thereby maximizing the use of GPU resources. Lastly, we provide a flexible method to adjust security levels and polynomial sizes through the incorporation of an in-kernel RNS method, and a flexible parameter generation approach. Comparative analysis against current state-of-the-art techniques reveals significant improvements. We observe performance gains of $2.82\times$ - $17.18\times$ against other DGT works on GPUs for different parameters, achieved concurrently with equal or lesser memory utilization.**

***Index Terms*—GPU, DGT, NTT, FHE, Polynomial Multiplication, Finite Field Transformations, Hardware Acceleration**

I. Introduction

The advent of outsourced computation has heightened privacy concerns, underlining the escalating need for privacy-preserving technologies. Although cryptographic schemes like Fully Homomorphic Encryption (FHE) [1] offer promising solutions to ensure privacy, their practical application is often marred by significant performance overheads, limiting widespread adoption [2]. Consequently, enhancing the efficiency of these cryptographic primitives is imperative, with a particular focus on optimizing fundamental operations such as multiplication, which is notably less efficient than addition or subtraction due to its $\mathcal{O}(n^2)$ complexity.

Finite field transformations, such as the Number Theoretic Transform (NTT), emerge as a crucial technique in addressing computational challenges, enabling multiplication in $\mathcal{O}(n)$ time. However, the transformation process itself introduces considerable computational costs, with complexity reaching $\mathcal{O}(n \log n)$. This underscores the importance of performance optimization in this domain, with hardware acceleration standing out as a viable strategy.

Options like CPUs [3]–[5], FPGAs [6]–[9], and ASICs [10]–[14] hold significant potential for optimization. However, GPUs currently enjoy widespread popularity due to their extensive utilization in Large Language Models (LLMs) and other high-performance computing applications. As a result, repurposing GPUs for cryptographic applications has become highly valuable. Despite GPUs' popularity and potential for cryptographic acceleration, it is crucial to acknowledge their architectural limitations. While adept at parallel processing, GPUs are primarily optimized for graphics rendering. Consequently, they encounter challenges when tasked with cryptographic computations, particularly with algorithms reliant on sequential processing or specialized non-native instructions.

The Discrete Galois Transformation (DGT) stands out as a promising candidate to NTT for GPU execution, primarily due to its enhanced capability for further parallelization. This attribute makes DGT particularly advantageous, as it aligns closely with the parallel processing strengths of GPUs, offering a pathway to significantly improved efficiency in computing finite field transformations.

Despite these advancements, prior research often fall short on boosting finite field transformation performance via GPU execution. Oftentimes existing works overlook the unique architectural constraints inherent to GPUs, and instead transfer techniques designed for alternative hardware platforms, such as FPGAs [15] and ASICs [10], without fully adapting to the nuances of GPU architecture. These adaptations fail to effectively parallelize workload tasks, leverage GPUs' native bit sizes, distribute computational loads efficiently, and optimize data access patterns to minimize transfer overheads. This work proposes a series of optimizations across both algorithmic and hardware-specific levels, tailored to the needs of GPU architectures. By addressing these challenges, we aim to unlock the full potential of GPUs for efficient cryptographic computations. In summary, our contributions include:

- We develop an optimized version of the Discrete Galois Transformation (DGT), a variant of the Number Theoretic Transform (NTT), specifically tailored for GPU computation. This enhancement significantly boosts execution efficiency, making it suitable for real-world applications.
- We introduce a novel data access pattern scheme engineered to enable coalesced accesses, enhancing the efficiency of data transfers between global and shared memory. This approach minimizes latency and maximizes throughput, which are critical factors for performance.
- We present a framework that identifies the most efficient allocation of computational tasks across the GPU and its multiple kernels, optimizing the use of GPU resources and enhancing overall performance.

978-3-9826741-0-0/DATE25/© 2025 EDAA

II. Preliminaries

A. Number Theoretic Transform

The Number Theoretic Transform (NTT) is a specialized version of the Discrete Fourier Transform (DFT) operating over the ring of integers modulo q ($\mathbb{Z}_q$). Crucial in cryptography, the NTT efficiently multiplies high-degree polynomials, reducing complexity from $O(n^2)$ to $O(n \log n)$. A polynomial $a = [a_0, a_1, \ldots, a_{N-1}]$ of degree N is transformed into a vector $\hat{a} = [\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{N-1}]$, where each coefficient is represented in the transformed domain. The NTT has a computational cost of $O(n \log n)$, defined by:

$$\hat{a}_k = \sum_{n=0}^{N-1} a_n \cdot w^{kn} \mod q, \quad k = 0, \ldots, N-1. \quad (1)$$

where w is a primitive Nth root of unity modulo q. The inverse NTT (iNTT), reconstructing the original polynomial, is:

$$a_n = N^{-1} \sum_{k=0}^{N-1} \hat{a}_k \cdot w^{-kn} \mod q, \quad n = 0, \ldots, N-1. \quad (2)$$

where N^{-1} is the multiplicative inverse of N modulo q.

For efficient NTT computation, two key transformation techniques are used, with their butterfly units (core execution unit): the *Gentleman-Sande* [16] (decimation in frequency) and the *Cooley-Tukey* algorithm [17] (decimation in time).

B. Discrete Galois Transform

The Discrete Galois Transform (DGT) [18] is a variant of the Discrete Fourier Transform (DFT) that operates within the Galois Field GF ($\hat{q}^2$), different from the Number Theoretic Transform (NTT) which uses integers modulo a prime. In DGT, elements are Gaussian integers, $u = u_{\text{re}} + iu_{\text{im}}$, with $u_{\text{re}}, u_{\text{im}} \in \mathbb{Z}_p$ and $i = \sqrt{-1}$. These undergo arithmetic operations similar to complex numbers modulo $\hat{q}$. By converting integer vectors to vectors of Gaussian numbers, DGT halves the transformation stages required from $\log_2(N)$ to $\log_2(N/2)$, streamlining computations into real and imaginary parts. This efficiency makes DGT particularly advantageous for performing negacyclic convolutions, using an $(N/2)$-point DGT instead of an N-point NTT.

C. GPU Architecture

Unlike CPUs, optimized for serial processing, Graphics Processing Units (GPUs) excel in parallel processing, executing thousands of threads simultaneously. The GPU computational structure is organized as follows [19]: *Threads* - smallest execution units, *Blocks* - three-dimensional sets of threads sharing data via shared memory, and *Grids* - three-dimensional sets of blocks processing different data segments. The memory structure in GPUs includes: *Registers* - fastest, thread-specific memory, *Shared Memory* - block-specific, faster than global memory but limited accessibility, *Global Memory* - slowest, accessible by all execution units.

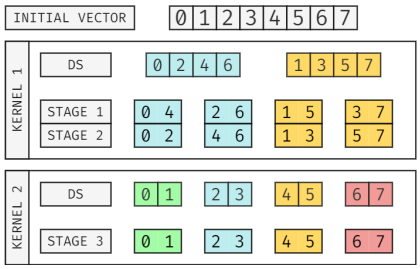

Fig. 1. Computational breakdown of butterfly pairs of the Gentleman-Sande Transformation for N=8 in two kernels

III. Methodology

A. Transformation Breakdown Over Multiple Kernels

1) GPU Constraints and Optimization Strategies: In applications like Fully Homomorphic Encryption, large datasets can cause data transfer bottlenecks. Optimizing these processes involves minimizing data transfer by efficiently using the shared memory of the GPU. However, GPU architecture imposes constraints that affect shared memory utilization:

1) GPU architecture limits threads per block to κ, capping transformation size per kernel: $N = 2\kappa$ for NTT and $N = 4\kappa$ for DGT, requiring more blocks for larger tasks.
2) Shared memory is restricted to individual thread blocks, limiting inter-block data interaction.

These constraints make it difficult to distribute data across blocks, which is essential for transformations that require all elements to interact. Ensuring data dependency within the same block is crucial to avoid slower global memory access.

2) Response to GPU Constraints: GPU architecture supports up to κ threads per block, typically 1024. Each thread processes two elements in radix-2 butterfly operations, enabling a 2κ-point NTT or 4κ-point DGT per kernel by dividing tasks between real and imaginary parts. For polynomials up to 4κ, a single block handles the load, optimizing shared memory use and reducing latency. For polynomials larger than 4κ, more kernels or blocks are required due to the need for more cores than a single block can support. The most promising approach when dealing with this limitation is to distribute the computation across multiple kernels.

Dependencies grow with the number of stages computed within a kernel. For example, as shown in Figure 1, computing three stages in one kernel requires $2^3 = 8$ elements to be in the same memory block, thus forming a dependency set (DS). Splitting the stages across two kernels reduces the dependency set size, allowing for strategic allocation across blocks. For polynomials of size n, this approach translates to a dependency set size of $l_k = 2^{\texttt{stages}_k}$ and a number of sets $d_k = n/l_k$. This setup allows processing up to $\log_2(4\kappa)$ stages per kernel. Typically, one or two kernels suffice for cryptographic computations, efficiently handling polynomial sizes up to 2^{24} with $\kappa = 1024$. For larger tasks, additional kernels accommodate higher degrees.

Algorithm 1 Vector Packing

Require: A, N, $stages_1$
Ensure: Array A is reordered according to the Vector Packing algorithm

function VECTORPACKING(A, N, $stages_1$)
  $m \leftarrow N/2$
  $step \leftarrow 1 \ll (\text{LOG2}(m) - stages_1 + 1)$
  Initialize array $temp$ with size N
  $index \leftarrow 0$
  for $i \leftarrow 0$ **to** $step - 1$ **do**
    for $j \leftarrow i$ **to** $m - 1$ **by** $step$ **do**
      $temp[index] \leftarrow A[j](real)$
      $temp[index + m] \leftarrow A[j + m](imag)$
      $index \leftarrow index + 1$
    end for
  end for
  for $i \leftarrow 0$ **to** $N - 1$ **do**
    $A[i] \leftarrow temp[i]$
  end for
end function

B. Data Transfer Optimization

As previously mentioned, the key to accelerating transformation processes on GPUs is to minimize data transfer distances by effectively utilizing shared memory. However, transferring data from global to shared memory introduces significant overheads if not well managed. In this context, coalesced accesses are crucial for improving data movement efficiency and reducing latency, optimizing overall GPU performance. This section introduces two methodologies aimed at refining data access between global and shared memory to address latency and bandwidth challenges. These approaches are detailed below:

1) **Data Packing Mechanism:** A packing mechanism is introduced to enable coalesced data transfers into GPU shared memory.
2) **Optimized DGT Algorithm:** A customized DGT algorithm is proposed to facilitate fixed memory access patterns, minimizing latencies.

1) Optimizing Global Memory Access: Coalesced accesses occur when threads access consecutive memory addresses that align with memory block sizes (typically `128 Bytes` in most GPU caches). This alignment allows an entire data block to be loaded in a single operation, improving efficiency and reducing latency. Optimizing global-to-shared memory transfers minimizes transactions and maximizes performance.

a) Vector Packing Algorithm: To enable coalesced accesses, we introduce a **Packing** stage, shown in Figure 2. Algorithm 1, describes the vector packing process, applied before and after the first kernel. Packing organizes dependency sets for efficient global-to-shared memory transfers. This procedure is applied to polynomial vectors to align data access patterns.

The choice of packing/unpacking method depends on the transformation technique used. The Gentleman-Sande (GS) method uses a packing approach starting with $m = N$, halving the value each round, while Cooley-Tukey starts with $m = 1$, doubling each round. Due to space, Cooley-Tukey's packing algorithm is not included, but follows a similar pattern.

These algorithms identify the stride τ between elements of dependency sets, which varies based on polynomial degree and stages computed per kernel. For example, for a polynomial of degree $n = 8$, in the GS method, the first two stages (Kernel 1) create two sets $[0, 2, 4, 6], [1, 3, 5, 7]$, while computing one stage (Kernel 2) creates four sets $[0, 1], [2, 3], [4, 5], [6, 7]$, ensuring aligned memory access for both real and imaginary components. The stride (Δ) can be generalized as:

For Gentleman-Sande (GS):

$$\Delta_{k\{i\}} = 2^{\log_2(N) - \tau_{k\{i\}}} = m_{k\{i+1\}}$$

For Cooley-Tukey (CT):

$$\Delta_{k\{i\}} = 2^{\tau_{k\{i-1\}}} = m_{k\{i-1\}}$$

b) Twiddle Factor Packing Algorithm: To improve computational efficiency during the execution of each core, we made the decision to precompute the twiddle factors. Similar to the polynomial vector, the twiddle factor vector also undergoes a packing procedure to optimize its memory layout. The twiddle packing algorithm addresses a more complex scenario where an array of size $n - 1$ must distribute values across multiple blocks, as some sets share the same twiddle factor. This algorithm identifies the required twiddle factors for each dependency set, collects them, and packs them in a manner similar to the vector packing process.

2) Optimizing Shared Memory Access: While shared memory access does not necessitate coalescing akin to global memory, it is imperative to organize access patterns carefully to circumvent bank conflicts, which can severely impede performance. To enhance the efficiency of GPU execution for DGT algorithms, we propose modifications to the conventional approaches, as outlined in Algorithms 2, and 3. The traditional GS and CT algorithms exhibit complex, dynamic access patterns, particularly due to the variable separation distance (m) between element pairs (e.g., i and $i + m$) that evolves with each computation stage. Such variability complicates memory access patterns, undermining the optimization potential for shared memory utilization.

Our approach fundamentally reimagines data access by maintaining constant access locations within each core/thread throughout the computation process. This methodological pivot, illustrated in Figure 3, streamlines access patterns, ensuring consistent, predictable memory interactions within the GPU's execution units. The adapted algorithms, delineated in 2 and 3, demonstrate these principles in action for both forward and backward transformations. They retain the core essence of their respective methods, however, unlike their predecessors, our modified algorithms maintain fixed access indices per core/thread while adjusting the destination indices for computed results, thereby simplifying and stabilizing the access patterns across all computation stages.

For the Gentleman-Sande DGT, the algorithm accommodates σ threads with 2σ elements residing on the same block. Each stage positions the interacting pairs σ indices apart;

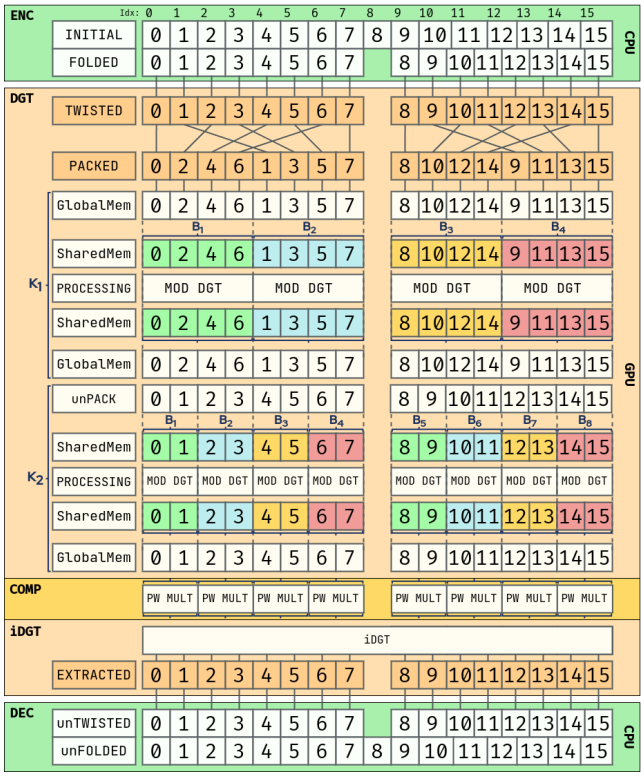

Fig. 2. Encoding, Forward Transformation, and Decoding phases for a process with $N = 16$. Each cell in the vector represents the position/index of the value. The computation requires $\log_2(N/2) = 3$ stages: the first two stages are executed in Kernel 1, and the final stage is completed in Kernel 2.

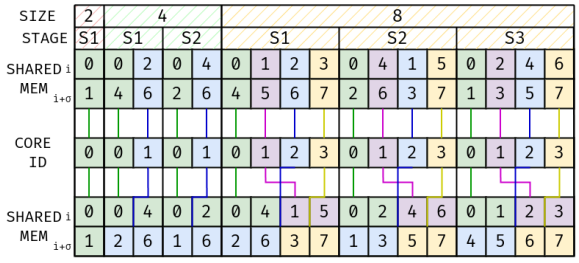

Fig. 3. Proposed Modified DGT (MOD DGT) on Dependency Set sizes of 2 and 4 (as example Fig. 2) and 8 (generalized).

therefore, core i consistently pulls data from positions i and $i + \sigma$. After computation, the data are then pushed back to shared memory, now adjacent to one another at positions $2i$ and $2i+1$. This arrangement creates a shifting motion of data from stage to stage, ensuring that data are always accurately positioned so each core consistently accesses the correct data pair with a fixed address. This strategy guarantees that each memory transaction is executed adjacently, enhancing throughput and minimizing latency. By eliminating any stride or multiplicative adjustments in the indexing, the data transfers remain linear and straightforward, leading to efficient and coalesced memory accesses.

In contrast, the Cooley-Tukey DGT operates in reverse. With the Cooley-Tukey method, m grows from 1 to $N/2$

Algorithm 2 Modified Gentleman-Sande Algorithm for DGT

Input: Gaussian vector a of length n, root w, modulo q, precomputed ω
Output: DGT-transformed vector a
for $c = 0$ to 1 **do**
  for $m = n/2$ to 1 by $m/2$ **do**
    for $i = 0$ to $n/2$ **do**
      $(a[2 * i][c], a[2 * i + 1][c]) \leftarrow$ BUTTERFLY$(a[i][c], a[i + n/2][c], \omega, q)$
    end for
  end for
end for

Algorithm 3 Modified Cooley-Tukey Algorithm for DGT

Input: Gaussian vector a of length n, primitive root w, modulo q, precomputed ω
Output: DGT-transformed vector a
$\sigma \leftarrow n/2$
for $c = 0$ to 1 **do**
  for $m = 1$ to $n/2$ by $2m$ **do**
    for $i = 0$ to $n/2$ **do**
      $(u, v) \leftarrow (a[2i][c], a[2i + 1][c])$
      $(a[i][c], a[i + \sigma][c]) \leftarrow$ BUTTERFLY(u, v, ω, q)
    end for
  end for
end for

during each stage, unlike the contraction from $N/2$ to 1 as in the GS method. For each core i, data are pulled from the shared memory at positions $2i$ and $2i + 1$, and subsequently, the results are pushed back to positions i and $i + \sigma$. This approach aligns with the CT technique's decimation in time, ensuring each stage logically expands the computation span.

The core operation in this process is the butterfly unit, which performs one addition, one subtraction, and one multiplication operation, followed by the respective modulus reductions. The inputs to the butterfly unit are the pulled values u and v, the precomputed $twiddle$, and the respective modulus q, while the outputs are the to-be-pushed values u' and v'. To enhance the efficiency of the primitive multiplication operation, crucial in DGT calculations, we adopt the Barrett reduction method [20].

The primary advantage of this revised strategy includes enhanced shared memory efficiency by eliminating the variability and complexity of traditional access patterns. By ensuring that each core/thread consistently accesses the same indices, we substantially reduce the likelihood of bank conflicts, thus optimizing the shared memory bandwidth and minimizing latency. Furthermore, this approach facilitates improved predictability in memory access, allowing for more effective compiler optimizations and GPU scheduler efficiency.

C. Workload Distribution

This section outlines our method for efficiently distributing the DGT workload for large polynomial degrees while addressing GPU constraints. Figure 4 shows our transformation distribution strategy across one kernel. The following subsections delve into each aspect of this strategy, detailing the parallelization approach behind workload distribution.

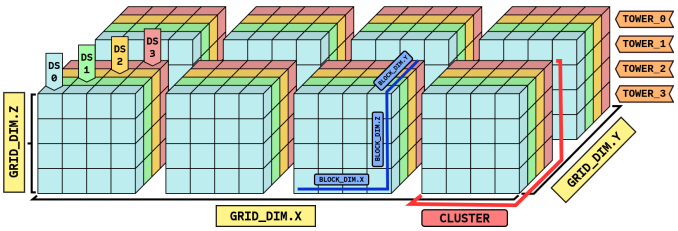

Fig. 4. Workload Distribution of a Single Kernel for Forward DGT Transformation Across the GPU

1) Gaussian Component Breakdown: The first layer of workload distribution is achieved by selecting the transformation algorithm. As discussed earlier, the Discrete Galois Transformation reduces transformation stages from $\log_2(N)$ to $\log_2(N/2)$ for a vector of size N, with each stage requiring Gaussian operations instead of integers. While integer butterfly operations require three primitive operations, Gaussian operations need six, split between real and imaginary components. These six operations can be divided in two segments, allowing independent processing of real and imaginary parts.

This separation enables a $\log_2(N/2)$ transformation for both real and imaginary parts, effectively doubling parallelization. On the GPU, this is managed by using `GridDIM.y`, where blocks with `BlockID.y` $= 0$ handle the real part and those with `BlockID.y` $= 1$ handle the imaginary part, providing the first level of parallelization.

2) Stage Partitioning & Optimization Through Dependency Set Clustering: As stages are split across multiple kernels, dependency sets are formed. A basic approach assigns dependency sets to `GridDIM.x` and their size to `blockDIM.x`. However, this leads to small blocks, which underutilize GPU resources. Each GPU block invokes an overhead for execution, therefore too many small blocks introduce execution overhead without sufficient work, consequently reducing efficiency. To solve this, we cluster multiple dependency sets within the same block as long as the thread count remains below the limit of κ. We reorganize one-dimensional blocks into two dimensions, packing multiple sets along `blockDIM.y`, forming *clusters* to optimize thread utilization. As shown in Figure 4, the cluster and block dimensions are:

$$\texttt{BlockDIM.x} = l_{k\{\delta\}},$$
$$\texttt{BlockDIM.y} = \lambda,$$
$$\texttt{GridDIM.x} = \frac{d_{k\{\delta\}}}{\lambda}$$

The optimal size of clusters, λ, depends on the number of stages and polynomial degree. Exploring different values ensures the best thread utilization per block and kernel.

3) Exploiting Native 32-bit Arithmetic through RNS Partitioning: Although GPUs support larger bit sizes, they are optimized for 32-bit operations. Previous approaches that use 64-bit sizes or floating-point arithmetic do not fully utilize GPU capabilities and ignore the efficiency difference between floating-point and integer units.

TABLE I
OPTIMAL EXECUTION TIMES (T_{TOWERS}) IN μS FOR A FORWARD POLYNOMIAL TRANSFORMATION, ALONGSIDE THE RESPECTIVE PARAMETER SETS ($Cluster_{\text{TOWERS}}$, $Pair_{\text{TOWERS}}$) FOR VARYING POLYNOMIAL DEGREES ACROSS DIFFERENT TOWER BIT-SIZES.

	RTX 2080									A100								
N	C_1	P_1	T_1	C_2	P_2	T_2	C_4	P_4	T_4	C_1	P_1	T_1	C_2	P_2	T_2	C_4	P_4	T_4
1024	8	0-9	**3.55**	2	0-9	**3.52**	2	0-9	**3.52**	2	1-8	**8.99**	2	4-5	**8.89**	4	3-6	**8.90**
2048	16	0-10	**4.58**	4	0-10	**4.58**	2	0-10	**4.61**	16	4-6	**9.15**	16	5-5	**9.22**	16	2-8	**9.09**
4096	4	4-7	**5.40**	8	3-8	**5.54**	32	4-7	**6.04**	32	3-8	**9.50**	16	4-7	**9.35**	8	3-8	**9.40**
8192	4	6-6	**5.67**	8	5-7	**6.17**	16	4-8	**7.04**	1	6-6	**9.69**	1	4-8	**9.79**	32	3-9	**10.11**
16384	2	6-7	**6.47**	8	5-8	**7.39**	8	4-9	**10.15**	2	7-6	**10.04**	2	4-9	**10.34**	4	5-8	**11.52**
32768	4	6-8	**7.84**	16	5-9	**10.94**	32	4-10	**16.93**	8	6-8	**10.78**	2	6-8	**11.65**	4	6-8	**14.11**
65536	8	6-9	**11.81**	16	6-9	**17.72**	8	6-9	**28.76**	2	7-8	**12.19**	2	7-8	**14.37**	8	6-9	**19.87**

Our system leverages native 32-bit performance while providing adaptable security levels through the residue number system (RNS). RNS represents large integers as smaller residues, allowing for parallel operations and faster computation. We implement 32-bit towers, where the number of towers t determines the security level (e.g., 4 towers offer 128-bit computation).

This multi-tower architecture is mapped onto `GridDim.Z`, enabling parallel execution of t towers, with each tower's computations isolated in different hardware sections to avoid conflicts. Our parameter generation mechanism supports a large number of towers by generating the roots of unity needed for the transformation, ensuring both high performance and customizable security without unnecessary overhead.

IV. Experimental Evaluation

We conducted a study of our proposed methods across two setups: an `Nvidia RTX 2080` (clock frequency-focused) and an `Nvidia A100 80GB` (memory bandwidth-focused).

A. Experimental Setup

Our experiments assess trade-offs in optimizing polynomial degree N, tower size, cluster dimension, and stage distribution:

- *Polynomial Degree*: We vary polynomial degrees ($N = 2^{10} \dots 2^{16}$) to evaluate performance across applications.
- *Bit Size (Towers)*: We test different bit sizes, from one tower (32-bit) to four towers (128-bit), to explore computational performance vs. security trade-offs.
- *Cluster Dimension*: We test the impact of packing different numbers of dependency sets (1-32) into a block.
- *Stage Distribution*: We evaluate the distribution of computational stages over multiple kernels for different polynomial degrees.

B. Framework & Results

We designed a framework to identify optimal GPU execution parameters by systematically exploring workload distributions across two kernels, including stage distribution, cluster packaging, and tower configurations. It employs a Pareto front to determine the fastest configurations for each workload.

Figure 5 illustrates this exploration. The analysis shows that equal stage distribution across kernels often leads to better performance, though certain cluster configurations may favor more stages in kernel 1. The optimal configuration is highly

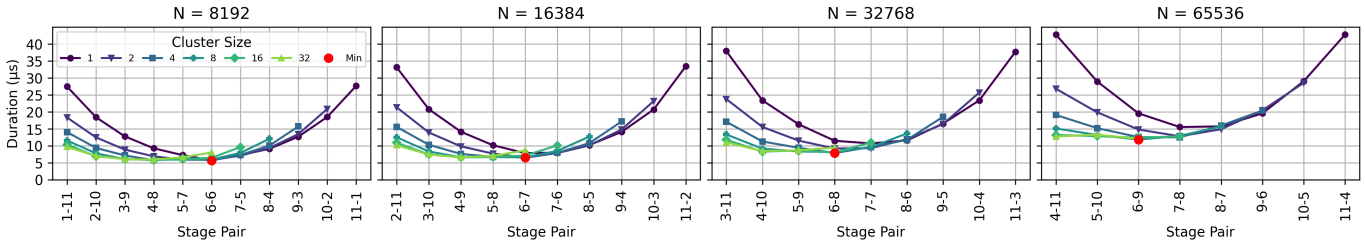

Fig. 5. Systematic exploration of all possible computational workload distributions across two kernels, measuring forward transformation runtime, and the Pareto front for a single tower (32-bit) on the GPU `RTX 2080`.

TABLE II
COMPARATIVE ANALYSIS OF VARIOUS DGT AND NTT IMPLEMENTATIONS ACROSS MULTIPLE GPU PLATFORMS

Method	Work	Device	$\log_2(N)$	$\log_2(q)$	Duration (µs)	Speedup using 2080	Speedup using A100
DGT	[21]	V100	11	63	26.0	5.68×	2.82×
			12	63	34.0	6.14×	3.64×
			13	63	40.0	6.48×	4.09×
			14	63	78.0	10.55×	7.54×
			15	63	188.0	17.18×	16.14×
NTT	[22]	GTX 980	14	55	51.0	6.90×	4.93×
			15	55	73.0	6.67×	6.27×
		GTX 1080	14	55	33.0	4.47×	3.19×
			15	55	36.0	3.29×	3.09×
		V100	14	55	29.0	3.92×	2.80×
			15	55	39.0	3.56×	3.35×
	[23]	A100	14	62	13.3	1.80×	**1.29**×
			16	62	16.5	0.93×	**1.15**×
		V100	14	30	8.7	1.34×	0.87×
			16	30	13.1	1.11×	1.07×
			14	62	11.5	1.56×	1.11×
			16	62	16.4	0.93×	1.14×
	[24]	GTX 3070	12	56	15.78	2.85×	1.69×
			13	56	17.47	2.83×	1.78×
			14	56	20.13	2.72×	1.95×
			15	56	21.79	1.99×	1.87×
			16	56	28.22	1.59×	1.96×
	[25]	GTX 690	14	64	56.0	7.58×	5.42×
			15	64	71.2	6.51×	6.11×

dependent on polynomial degree and workload characteristics, requiring empirical evaluation to find the best parameter set.

Table I shows the optimal execution times ($\mathtt{T_{towers}}$) for varying polynomial degrees and tower configurations. Results show that the `RTX 2080` excels at smaller polynomial degrees (up to 2.5× speedup for $N = 1024$), while the `A100` outperforms `RTX 2080` for larger degrees (1.45× faster for $N = 65536$ with four towers). These differences stem from GPU hardware: the `RTX 2080` benefits from its higher clock frequency (1710 MHz vs. 1410 MHz for `A100`) in smaller polynomial computations, while the `A100` excels in large-scale computations due to its superior memory bandwidth (2.04 TB/s vs. 448 GB/s for `RTX 2080`), making it more adept at managing larger polynomial degrees where memory transfer becomes the limiting factor.

C. Related Work

Table II compares our work with other NTT and DGT implementations, evaluating latency metrics across polynomial degrees and bit-sizes on `NVIDIA RTX 2080` and `NVIDIA A100 80GB`. Following GPU research standards, we compare across devices for broader context. To ensure fairness, we also compare using the same GPUs as in our study, showing performance gains. Alves' Hierarchical DGT [21] uses a fixed modulus and Solinas primes, limiting the flexibility of cryptographic solutions. In contrast, our method supports a broader range of prime moduli, achieving 2.82× - 17.18× faster results. Özerk et al. [22] optimize NTT for non-native 64-bit arithmetic, which is resource-heavy and lacks flexibility in handling different $\log_2(q)$. Our method is 2.80× - 6.90× faster, offering better flexibility and efficiency. Shivdhikar et al. [23] use kernel fusion and modular reduction but do not fully utilize 32-bit arithmetic for 64-bit computations, limiting performance. Our methodology generally outperforms Shivdhikar's, but in some cases, their work shows better results. It is important to note, however, that this only occurs in cases where we use a clock frequency-focused GPU for a memory bandwidth-heavy task, or a memory bandwidth-focused GPU for a clock frequency-heavy task. Nonetheless, on an equal comparison between the two works (both using `A100` platforms), our method demonstrates speedups of 1.29× and 1.15× for $\log_2(N) = 14$ and 16, respectively, showcasing better efficiency and scalability. The cuHE library [25] uses CRT but is not optimized for maximum performance, leading to slower runtimes. Our method improves performance by 5.42× - 7.58×. Wang's NTTfusion [24] reduces synchronization overhead, but our stage management techniques further enhance throughput and reduce latency, making our method 1.59× - 2.85× faster.

V. CONCLUSION

In this study, we have introduced `Coala`, a coalescion-based framework that accelerates polynomial multiplication in GPUs using a customized Discrete Galois Transformation. Comparative results show performance gains of 2.82× - 17.18× against state-of-the-art work running DGT using GPUs. This framework establishes a robust base for future developments in cryptography acceleration using GPUs.

ACKNOWLEDGMENTS

This research was supported by the NYUAD Global Ph.D. Fellowship and the NYUAD Center for Cyber Security.

REFERENCES

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[2] E. Chielle, H. Gamil, and M. Maniatakos, "Real-time private membership test using homomorphic encryption," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1282–1287.

[3] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, "E3: A framework for compiling c++ programs with encrypted operands," *Cryptology ePrint Archive*, 2018.

[4] E. Chielle, O. Mazonka, H. Gamil, and M. Maniatakos, "Coupling bit and modular arithmetic for efficient general-purpose fully homomorphic encryption," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 4, pp. 1–28, 2024.

[5] "Microsoft SEAL (release 3.3.2)," https://github.com/Microsoft/SEAL, 2019, microsoft Research, Redmond, WA.

[6] G. Li, D. Chen, G. Mao, W. Dai, A. I. Sanka, and R. C. Cheung, "Algorithm-hardware co-design of split-radix discrete galois transformation for kyberkem," *IEEE Transactions on Emerging Topics in Computing*, 2023.

[7] K. Derya, A. C. Mert, E. Öztürk, and E. Savaş, "Coha-ntt: A configurable hardware accelerator for ntt-based polynomial multiplication," *Microprocessors and Microsystems*, vol. 89, p. 104451, 2022.

[8] X. Hu, J. Tian, M. Li, and Z. Wang, "Ac-pm: An area-efficient and configurable polynomial multiplier for lattice based cryptography," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 2, pp. 719–732, 2022.

[9] W. Tan, S.-W. Chiu, A. Wang, Y. Lao, and K. K. Parhi, "Parentt: Low-latency parallel residue number system and ntt-based long polynomial modular multiplication for homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, 2023.

[10] M. Nabeel, D. Soni, M. Ashraf, M. A. Gebremichael, H. Gamil, E. Chielle, R. Karri, M. Sanduleanu, and M. Maniatakos, "Cofhee: A co-processor for fully homomorphic encryption execution," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–2.

[11] M. Nabeel, H. Gamil, D. Soni, M. Ashraf, M. A. Gebremichael, E. Chielle, R. Karri, M. Sanduleanu, and M. Maniatakos, "Silicon-proven asic design for the polynomial operations of fully homomorphic encryption," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[12] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[13] M. Nabeel, H. Gamil, J. Knechtel, and M. Maniatakos, "Mcs-ntt: Multi-chip system design for ntt acceleration," in *2024 IFIP/IEEE 32nd International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2024, pp. 1–4.

[14] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th annual international symposium on computer architecture*, 2022, pp. 711–725.

[15] A. C. Mert, E. Öztürk, and E. Savaş, "Fpga implementation of a run-time configurable ntt-based polynomial multiplication hardware," *Microprocessors and Microsystems*, vol. 78, p. 103219, 2020.

[16] W. M. Gentleman and G. Sande, "Fast fourier transforms: for fun and profit," in *Proceedings of the November 7-10, 1966, fall joint computer conference*, 1966, pp. 563–578.

[17] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[18] R. E. Crandall, "Integer convolution via split-radix fast galois transform," *Center for Advanced Computation Reed College*, 1999.

[19] D. Guide, "Cuda c programming guide," *NVIDIA, July*, vol. 29, p. 31, 2013.

[20] A. Ş. Özcan and E. Savaş, "Two algorithms for fast gpu implementation of ntt," *Cryptology ePrint Archive*, 2023.

[21] P. G. M. Alves, J. N. Ortiz, and D. F. Aranha, "Faster homomorphic encryption over gpgpus via hierarchical dgt," in *International Conference on Financial Cryptography and Data Security*. Springer, 2021, pp. 520–540.

[22] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savaş, "Efficient number theoretic transform implementation on gpu for homomorphic encryption," *The Journal of Supercomputing*, vol. 78, no. 2, pp. 2840–2872, 2022.

[23] K. Shivdikar, G. Jonatan, E. Mora, N. Livesay, R. Agrawal, A. Joshi, J. L. Abellán, J. Kim, and D. Kaeli, "Accelerating polynomial multiplication for homomorphic encryption on gpus," in *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, 2022, pp. 61–72.

[24] Z. Wang, P. Li, R. Hou, and D. Meng, "Nttfusion: Efficient number theoretic transform acceleration on gpus," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 2023, pp. 357–365.

[25] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *Cryptography and Information Security in the Balkans: Second International Conference, BalkanCryptSec 2015, Koper, Slovenia, September 3-4, 2015, Revised Selected Papers 2*. Springer, 2016, pp. 169–186.