Accelerating Finite Field Arithmetic for Homomorphic Encryption on GPUs

Neal Livesay*, Gilbert Jonatan†, Evelio Mora‡, Kaustubh Shivdikar*, Rashmi Agrawal§, Ajay Joshi§, José L. Abellán¶, John Kim†, David Kaeli*
*Northeastern University, §Boston University, †KAIST University, ‡Universidad Católica de Murcia, ¶Universidad de Murcia
{n.livesay, shivdikar.k}@northeastern.edu, gilbertjonatan@kaist.ac.kr, eamora@ucam.edu, {rashmi23, joshi}@bu.edu, jlabellan@um.es, jjk12@kaist.edu, kaeli@ece.neu.edu

***Abstract*—Fully Homomorphic Encryption (FHE) is a rapidly developing technology that enables computation directly on encrypted data, making it a compelling solution for security in cloud-based systems. In addition, modern FHE schemes are believed to be resistant to quantum attacks. Although FHE offers unprecedented potential for security, current implementations suffer from prohibitively high latency. Finite field arithmetic operations, particularly the multiplication of high-degree polynomials, are key computational bottlenecks. The parallel processing capabilities provided by modern Graphical Processing Units (GPUs) make them compelling candidates to target these highly parallelizable workloads. In this article, we discuss methods to accelerate polynomial multiplication with GPUs, with the goal of making FHE practical.**

COMPUTER SECURITY continues to grow in importance as computation and storage of sensitive data is increasingly outsourced to cloud-based computing services. Moreover, with the rapid development of quantum computers, we expect to see new classes of threats looming that can defeat the security of long-trusted cryptosystems.

Fully Homomorphic Encryption (FHE) is an emerging technology with the potential to address both of these problems. FHE enables computation directly on encrypted data, concealing both operands and results from wily attackers on untrusted computing servers (see Figure 1). The security of many modern FHE schemes is based on the hardness of the Ring Learning with Errors problem [1], which is presumed to be resistant to quantum attacks.

A major hurdle to deploying FHE in real-world applications is overcoming the high computational costs associated with its workloads. Fortunately, FHE workloads are highly parallelizable, making FHE a strong candidate for GPU acceler-

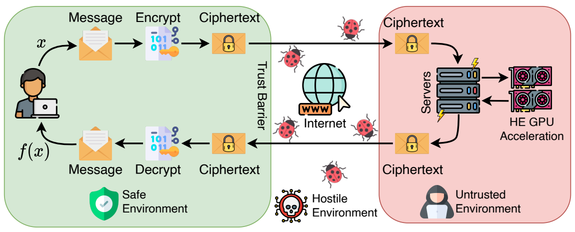

Figure 1. FHE enables computation on encrypted operands, providing strong security in the cloud-computing era.

ation. We focus our attention on two key computational bottlenecks: (1) *modular reduction*, or the computation of the remainder in integer division; and (2) the *Number Theoretic Transform* used to efficiently multiply polynomials. We begin by presenting background on these bottlenecks, followed by some potential approaches to accelerate them on a GPU. To that end, we leverage a number of architectural features such as shared memory and intrinsic instructions.

We build on our preliminary exploration into accelerating FHE [2]. The main contributions of this work include: a modulus selection methodology for Barrett reduction; design and evaluation of a "lazy" Barrett-based NTT optimization; evaluation of a Residue Number System based extension of our prior NTT implementation, including an evaluation of 32-bit vs 64-bit word sizes; and an evaluation of multiple methods for computing "$1/N$-scaling" in inverse NTTs.

Barrett Reduction

We begin with an investigation of one of the most well-known and widely-used modular reduction algorithms: Barrett reduction. In particular, we discuss the costly *correctional subtraction* operation appearing in computations involving Barrett reduction, and present novel methods to significantly decrease the number of these operations.

Background: Modular Arithmetic and the Correctional Subtraction

Let $x \bmod q$ denote the remainder of a non-negative integer x divided by a positive integer q. The naive method for performing *modular reduction* (i.e., the computation of $x \bmod q$) is via an integer division operation: $x \bmod q = x - \lfloor x/q \rfloor \times q$. However, integer division is computationally expensive. Moreover, integer division typically takes a number of cycles that is highly dependent on the data, making it susceptible to side-channel attacks [3]. Fortunately, there are a number of attractive alternatives for performing modular reduction—especially in conjunction with arithmetic operations such as addition and multiplication—that avoid undesirable integer division operations.

For example, Algorithm 1 specifies a simple and efficient computation of the modular reduction of a sum. Observe that, assuming a and b are each *reduced* (i.e., they lie in $[0, q)$), then either $a + b$ is reduced or $a + b$ lies in $[q, 2q)$ and requires a single *correctional subtraction* to become reduced (see lines 2–3). Correctional subtractions are a common feature in modular reduction and arithmetic algorithms. Fortunately, correctional subtractions can be implemented without branch instructions, avoiding control divergence that can significantly impact performance on GPUs.

Algorithm 1 Baseline modular addition algorithm

```
Require: $0 \leq a, b < q$, $\mathrm{len}(q) \leq \beta - 1$
Ensure: $\mathrm{sum} = (a + b) \bmod q$
1: $\mathrm{sum} \leftarrow a + b$
2: if $\mathrm{sum} \geq q$ then
3:     $\mathrm{sum} \leftarrow \mathrm{sum} - q$
4: return $\mathrm{sum}$
```

Note that the bit-length (i.e., the number of bits in the binary representation) of the modulus q is limited to be at most one less than the word length β, preventing overflow of the intermediate operations (e.g., $a + b$).

Algorithm 2 specifies the classical Barrett reduction algorithm commonly used to efficiently reduce products. The key idea underlying Barrett reduction is that it replaces a computation of the quotient $\lfloor x/q \rfloor$ with a computation of an "approximate quotient"

$$\mathrm{quot} = \left\lfloor \frac{\left\lfloor \frac{x}{2^{m-1}} \right\rfloor \left\lfloor \frac{2^{2m}}{q} \right\rfloor}{2^{m+1}} \right\rfloor, \tag{1}$$

where $m = \mathrm{len}(q)$. Provided the "Barrett constant" $\mu = \lfloor \frac{2^{2m}}{q} \rfloor$ is precomputed, the approximate quotient can be efficiently computed without integer division, as the divisions by powers of two

Algorithm 2 Classical Barrett reduction

```
Require: $m = \text{len}(q) \leq \beta - 2,\ 0 \leq x < 2^{2m}$, $\mu = \lfloor \frac{2^{2m}}{q} \rfloor$
Ensure: rem $= x \bmod q$
1: $c \leftarrow \lfloor \frac{x}{2^{m-1}} \rfloor$
2: quot $\leftarrow \lfloor \frac{c \times \mu}{2^{m+1}} \rfloor$
3: rem $\leftarrow x -$ quot $\times q$
4: if rem $\geq q$ then
5:     rem $\leftarrow$ rem $- q$
6: if rem $\geq q$ then
7:     rem $\leftarrow$ rem $- q$
8: return rem
```

Algorithm 3 BetterBarrett reduction: Dhem–Quisquater with $(\alpha, \beta) = (m+1, -2)$

```
Require: $m = \text{len}(q) \leq \beta - 2,\ 0 \leq x < 2^{2m}$, $\mu = \lfloor \frac{2^{2m+1}}{q} \rfloor$
Ensure: rem $= x \bmod q$
1: $c \leftarrow \lfloor \frac{x}{2^{m-2}} \rfloor$
2: quot $\leftarrow \lfloor \frac{c \times \mu}{2^{m+3}} \rfloor$
3: rem $\leftarrow x -$ quot $\times q$
4: if rem $\geq q$ then
5:     rem $\leftarrow$ rem $- q$
6: return rem
```

in lines 1–2 can be implemented via bit shifts (which are highly efficient operations on GPUs).

The number of correctional subtractions required to fully reduce x via classical Barrett reduction equals the difference $\lfloor x/q \rfloor -$ quot between the actual and the approximate quotient. Provided x is sufficiently small (i.e., less than 2^{2m}), this number is either zero, one, or two (lines 4–7).

Omitting Correctional Subtractions

In the original paper introducing Barrett reduction, Paul Barrett remarked—without further qualifications—that his algorithm requires a second conditional subtraction “only in 1% of cases” [4]. This motivates the following question: Is it possible to modify Barrett’s algorithm so that the second correctional subtraction instruction (i.e., lines 6–7) may be omitted?

Improved Quotient Approximation To address this question, we first consider an approach that involves modifying the classical quotient approximation. Dhem and Quisquater [5] introduced the following generalization of Barrett’s approximate quotient, with parameters α and β:

$$\text{quot} = \left\lfloor \frac{\lfloor \frac{x}{2^{m+\beta}} \rfloor \lfloor \frac{2^{m+\alpha}}{q} \rfloor}{2^{m-\beta}} \right\rfloor. \qquad (2)$$

Each instantiation (α, β) corresponds to both an upper bound on the number of required correctional subtractions and an upper bound on the bit-length of the modulus in the corresponding reduction algorithm. These two upper bounds are directly related, and thus, present a trade-off. Instantiations requiring as few as zero and as many as three correctional subtractions have demonstrated favorable performance for various applications [6].

Many prior studies and open-source libraries (e.g., the well-known OpenFHE library [7]) use the instantiation $(\alpha, \beta) = (m+3, -2)$, which requires at most one correctional subtraction and restricts the modulus length to at most four less than the word length (e.g., a 28-bit modulus on a single-precision 32-bit word on a GPU).

To the best of our knowledge, the instantiation $(\alpha, \beta) = (m+1, -2)$ does not appear in any paper or open-source library. This apparently overlooked instantiation, which we call *Better-Barrett*, is specified in Algorithm 3. Similar to the instantiation used by OpenFHE, BetterBarrett requires at most one correctional subtraction. However, BetterBarrett only restricts the modulus length to at most two less than the word length,

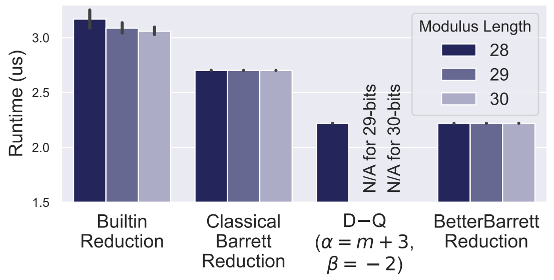

Figure 2. Runtimes (on NVIDIA’s V100 GPU) of modular reduction implementations for 28-, 29-, and 30-bit moduli. The “builtin reduction” uses the CUDA % construct for modular reduction. BetterBarrett has a $1.22\times$ speedup over classical Barrett reduction for 30-bit prime moduli chosen uniformly at random.

making it an attractive alternative for applications where large moduli are desirable (such as FHE).

Observe that BetterBarrett shares the same restriction on its modulus as classical Barrett reduction. As such, BetterBarrett may be viewed as a more performant replacement for classical Barrett reduction (as shown in Figure 2). We are not aware of a practical context where classical Barrett is preferable to BetterBarrett.

Modulus Selection Next, we examine a second approach for optimizing Barrett reduction that takes the choice of modulus into consideration. Recall that classical Barrett reduction requires at most two correctional subtractions to fully reduce an integer x in $[0, 2^{2\text{len}(q)})$. This prompts an interesting question: Are there any moduli q for which classical Barrett reduction requires at most one correctional subtraction to reduce any integer in $[0, (q-1)^2]$ (and thus any product, the common use case for Barrett reduction)?

A naive approach to determining whether a given modulus q satisfies this property involves a brute-force verification that $\lfloor x/q \rfloor - \text{quot} \leq 1$ for all x in $[0, (q-1)^2]$. We proved in Theorem 1 that the number of required verifications can be reduced by a factor of q.

Theorem 1:
Barrett reduction requires at most k correctional subtractions to fully reduce any x in $[0, L]$ if

$$j - \left\lfloor \frac{\lfloor \frac{j \times q}{2^{m+\beta}} \rfloor \times \mu}{2^{m-\beta}} \right\rfloor \leq k \tag{3}$$

for all j in $[0, \lfloor L/q \rfloor]$.

Proof:
The claim follows from the fact that the quotient function $x \mapsto \lfloor x/q \rfloor$ and approximate quotient function $x \mapsto \text{quot}(x)$ are monotone increasing, with the former remaining constant on intervals $[j \times q, j \times (q+1))$ for each integer j. ■

Using a search algorithm based on Theorem 1, we found that moduli q which require at most one correctional subtraction to fully reduce a product via classical Barrett reduction are surprisingly common. For example, of the 395 30-bit primes q that are "negacyclic-friendly" for $N = 2^{16}$ (i.e., moduli that are relevant for FHE), 192 require at most one correctional subtraction to fully reduce a product. This provides yet another alternative to the widely used Dhem–Quisquater reduction corresponding to $(\alpha, \beta) = (m+3, -2)$ that does not impose a further restriction on the modulus length.

In general, this modulus selection approach may be useful in contexts—such as implementations of NTT based on the Residue Number System—where a relatively large set of moduli must be selected from a heavily restricted class of candidates. A currently prevalent modulus selection approach involves simply choosing the largest several moduli from the candidate set; see e.g., the Microsoft Simple Encrypted Arithmetic Library (SEAL).

Polynomial Multiplication

Next, we shift our attention from modular multiplication to polynomial multiplication. More specifically, we focus on the Number Theoretic Transform (NTT), an algorithm commonly used to efficiently implement polynomial multiplication in the context of Fully Homomorphic Encryption. Throughout, we fix a degree-bound N and represent a polynomial $\sum_{i=0}^{N-1} a_i x^i$ as an N-dimensional *coefficient vector* $\mathbf{a} = (a_0, a_1, \ldots, a_{N-1})$.

Background: The Number Theoretic Transform

The naive approach to multiplying polynomials requires the computation of order N^2 products and sums. This can be reduced to order $N \log(N)$ using the celebrated Fast Fourier Transform (FFT). FFT consists of an iteration of *stages*, in which pairs of coefficients are transformed via a *butterfly operation*. There are numerous variations of the butterfly operation, but a classical baseline is the Cooley–Tukey (CT) butterfly defined below:

$$\text{butt}_{\zeta}^{\text{CT}} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} a_0 + a_1 \\ a_0 - \zeta \times a_1 \end{pmatrix} \tag{4}$$

In the context of FHE, the term "polynomial multiplication" typically refers to *negacyclic convolution*, a slight variant of the usual polynomial multiplication. On hardware platforms such as CPUs and GPUs, negacyclic convolution is commonly implemented via an efficient combination of two specialized variants of the FFT, known

Algorithm 4 The Number Theoretic Transform

```
Require: $N = 2^n$, $q$ negacyclic-friendly prime, twiddle factors $\mathbf{\Psi}_{\mathrm{br}}$ in bit-reversed order, $\mathbf{a} = (a_0, a_1, \ldots, a_{N-1}) \in (\mathbb{Z}_q)^N$
 1: $m \leftarrow 1$
 2: $k \leftarrow N/2$
 3: while $m < N$ do
 4:     for $i = 0$ to $m - 1$ do
 5:         $jFirst \leftarrow 2 \times i \times k$
 6:         $jLast \leftarrow jFirst + k - 1$
 7:         $\zeta \leftarrow \mathbf{\Psi}_{\mathrm{br}}[m + i]$
 8:         for $j = jFirst$ to $jLast$ do
 9:             $\begin{pmatrix} \mathbf{a}[j] \\ \mathbf{a}[j+k] \end{pmatrix} \leftarrow \mathrm{butt}_{\zeta}^{\mathrm{CT}} \begin{pmatrix} \mathbf{a}[j] \\ \mathbf{a}[j+k] \end{pmatrix}$
10:     $m \leftarrow 2 \times m$
11:     $k \leftarrow k/2$
12: return $\mathbf{a}$
```

Algorithm 5 A lazy Barrett-based butterfly

```
Require: $0 \le a_0, a_1 \le 2(q-1)$, $0 \le \zeta < q$, $m = \mathrm{len}(q) \le \beta - 2$, $q < 2^{m-1/2}$
Ensure: $0 \le b_0, b_1 \le 2(q-1)$,
    $b_0 \bmod q = (a_0 + \zeta \times a_1) \bmod q$,
    $b_1 \bmod q = (a_0 - \zeta \times a_1) \bmod q$
1: $\mathrm{prod} \leftarrow \zeta \times a_1$
2: $r \leftarrow \mathrm{barrett}_q^{\mathrm{trunc}}(\mathrm{prod})$
3: $b_0 \leftarrow a_0 + \mathrm{prod}$
4: $b_1 \leftarrow a_0 - \mathrm{prod}$
5: if $b_0 \ge 2q$ then
6:     $b_0 \leftarrow b_0 - 2q$
7: if $b_1 \ge 2q$ then
8:     $b_1 \leftarrow b_1 - 2q$
9: return $\begin{pmatrix} b_0 \\ b_1 \end{pmatrix}$
```

as the *merged NTT* (specified in Algorithm 4) and the *merged inverse NTT (iNTT)*. In this case, the negacyclic convolution of $\mathbf{a}$ and $\mathbf{b}$ can be computed via the expression $\frac{1}{N}\mathrm{iNTT}(\mathrm{NTT}(\mathbf{a}) \odot \mathrm{NTT}(\mathbf{b}))$. For further details, see [8]. We henceforth omit the modifier “merged” for brevity.

Lazy Butterfly Optimization

Harvey introduced a specification for an efficient “lazy” variant of the butterfly operation [9]. An NTT implemented with Harvey’s lazy butterfly optimization omits a correctional subtraction in each modular product. Although the intermediate operations are partially computed (hence, the qualifier “lazy”), the final result is correct. This results in a total of $N(\frac{\log N}{2} - 1)$ fewer correctional subtractions for each NTT.

Harvey’s optimization is specified for NTTs based on Shoup’s method, an approach for computing NTTs that requires precomputed modifications to the twiddle factors to be implemented efficiently. We propose a Barrett-based version of Harvey’s lazy butterfly that requires no precomputed modifications.

Our lazy Barrett-based butterfly is specified in Algorithm 5. We define the *truncated Barrett reduction function* $\mathrm{barrett}_q^{\mathrm{trunc}}$ to be Barrett reduction (e.g., BetterBarrett) with a correctional subtraction omitted (i.e., lines 4–5).

Our *lazy Barrett-based Number Theoretic Transform* is defined by making the following modifications to Algorithm 4: (1) further constraining the modulus by $q < 2^{\mathrm{len}(q)-1/2}$; (2) replacing the butterfly in line 9 with the lazy butterfly specified in Algorithm 5; and (3) applying a correctional subtraction to each entry of the output greater than or equal to q.

Theorem 2:
The lazy Barrett-based NTT is correct.

Proof:
It suffices to show that the input to $\mathrm{barrett}_q^{\mathrm{trunc}}$ in line 2 of Algorithm 5 lies in the prescribed domain, $[0, 2^{2\mathrm{len}(q)})$. If ζ lies in $[0, q)$ and a_1 lies in $[0, 2(q-1))$, then $0 \le \zeta \times a_1 \le 2(q-1)^2$. This is bounded above by $2^{2\mathrm{len}(q)}$ since $q < 2^{\mathrm{len}(q)-1/2}$. ■

Background: The Residue Number System

To guarantee a sufficient level of security for FHE, it is necessary that the set $\mathbb{Z}_Q$ of coefficients be sufficiently large. For example, it is common to assume $\mathrm{len}(Q) > 1000$. To efficiently implement NTT over such large coefficients on a hardware platform with significantly smaller word lengths (e.g., the 32- or 64-bit data types available on many GPUs), we use the well-known Residue Number System (RNS).

To use the RNS, we choose Q to be a product of distinct word-sized, negacyclic-friendly primes $q_1, q_2, \ldots, q_\ell$, each of equal length. Then a computation of an NTT with respect to Q can be replaced by a computation of ℓ NTTs with respect to the ℓ distinct prime moduli. The number ℓ, called the *number of limbs*, is inversely related to the modulus length m by $\ell = \frac{Q}{m}$. This implies an interesting trade-off between the *workload size* (i.e., the total number of butterflies computed in ℓ NTTs) and the *operational complexity* (i.e., the complexity of arithmetic operations with respect to the m-bit moduli).

Efficient Methods to Accelerate Polynomial Multiplication on GPU

Given the large size N and a large modulus Q for each polynomial in FHE, polynomial multiplication requires a significant amount of computation. Fortunately, polynomial multiplication algorithms based on NTT (and, more generally, FFT) are highly parallelizable and can exploit the massive number of threads and high memory throughput on modern GPUs.

GPU architectures are based on the *Single Instruction Multiple Thread (SIMT)* programming model where a group of threads (called *warps*) execute instructions over a collection of data in parallel (in a lockstep manner). These warps are grouped into *blocks*, which are further grouped into *grids*. In particular, we target the NVIDIA V100 GPU, which consists of 128 KB L1 data and instruction caches per streaming multiprocessor (SM), a multi-banked shared L2 cache (6.1 MB), and 16 GB HBM2 global memory. Each SM also has a low latency user-configurable cache called shared memory (each configurable in size up to 96 KB). As we will explain, we tailor our implementations by applying incremental optimizations that are aimed at harnessing these underlying architectural features to speed up our FHE workloads.

We obtain performance metrics for our kernels using the following tools: the NVIDIA Binary Instrumentation Tool (NVBit) for tracing memory transactions, the Nsight Compute kernel profiler for fetching performance counters, and the Nsight Systems performance analysis tool to obtain kernel scheduler performance and measure synchronization overheads. Our experiments provide insights into our kernel implementations and are aimed at mitigating the performance impacts of microarchitectural bottlenecks.

For all of our implementations, we use BetterBarrett to implement modular multiplication. We consider implementations based on both 32-bit and 64-bit data types, using 30-bit and 62-bit moduli, respectively (the largest permitted bit-lengths for BetterBarrett reduction). We fix $N = 2^{16}$ for the NTT size. We use $\text{len}(Q) = 1860$ for the modulus length. Using the Residue Number System, this choice of modulus length translates to 62 and 30 limbs for our 32-bit and 64-bit implementations, respectively. For each limb, we use the merged NTT algorithm.

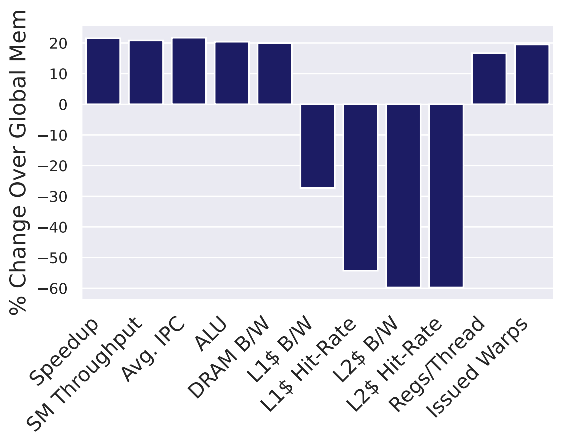

Figure 3. Performance analysis of single-limb NTT kernels employing shared memory optimization in comparison to global memory kernels.

GPU Implementations and Analysis

Next, we discuss a series of incremental optimizations targeting the GPU architecture.

Our baseline NTT implementation employs the *hybrid-kernel optimization* [10], which consists of a combination of single-stage and multi-stage GPU kernels. This approach leverages the following observation: after k stages of NTT have been executed, the remaining computation can be subdivided into 2^k sub-computations, each operating on a distinct set of $N/2^k$ coefficients.

We outline the hybrid-kernel approach for $N = 2^{16}$ that has 16 stages. During the first five stages, a naive *single-stage kernel* approach assigns one kernel per stage and synchronizes between kernels with inter-block synchronization to ensure all blocks have access to updated data. In practice, this is achieved by using a barrier on kernel finish. After the fifth stage, the remaining computation can be subdivided into sub-computations, each operating on a distinct set of $2^{11} = 2048$ coefficients. For each of these sub-computations, a *multi-stage kernel* approach assigns the associated set of coefficients to a 1024-thread block. Multi-stage kernels exchange inter-block synchronizations for intra-block synchronizations, where the barrier only affects the threads of a block, ensuring that the data in shared memory is updated so we can continue with the next stage. This does not affect the rest of the blocks, allowing them to continue computation without interruption.

Additionally, multi-stage kernels effectively leverage shared memory for efficient coefficient storage. Using shared memory mitigates redundant memory accesses (reducing memory pressure on the L1 and L2 caches) while increasing compute throughput. In Figure 3, we see the impact of shared memory on single-limb NTT and inverse NTT kernels. Incorporating shared memory provides over 20% improvement in the compute throughput of NTT, while significantly reducing the demands on L1 and L2 data caches (shared memory bandwidth does not contribute to L1 and L2 cache bandwidth, but does contribute to DRAM bandwidth).

Radix Configurations Furthermore, we experimented with various configurations of higher-radix butterflies. The *radix* of a butterfly corresponds to the number of its operands. An increase in the radix results in a decrease in both the number of stages in each NTT and the number of kernel launches (and the associated kernel launch overhead). However, an increase in the radix can also reduce the amount of parallelism as each thread processes more data. This trade-off can be seen in Figure 4. We achieved the best performance with a mixed 4/16-radix implementation.

Hierarchical NTT To further increase block-level parallelism, we implemented a (two-dimensional) hierarchical NTT [11]. We refer to our implementation as our *2D NTT*. In our 2D NTT, a set of 2^{16} coefficients are arranged into a $2^8 \times 2^8$ row-major matrix. Processing consists of two steps. In the first step, a 2^8-point NTT is performed on each of the 2^8 columns. In the second step, a 2^8-point NTT is performed on each of the 2^8 rows. Block-level synchronizations

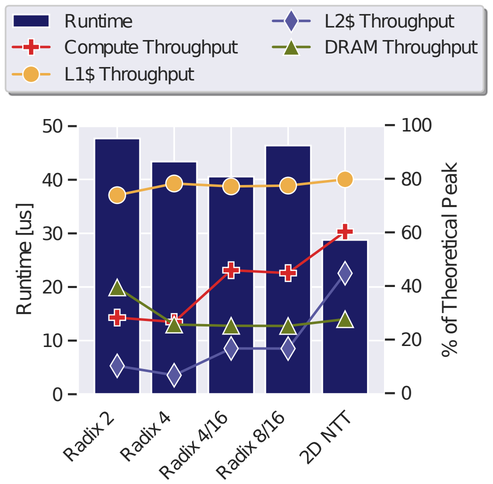

Figure 4. Performance metrics for single-limb NTT kernels with various radix configurations.

occur between these two steps. We assign each of the 2^8-point NTTs to a single block, resulting in 256 blocks per step. Each step maps to a kernel, where each thread computes a single butterfly per stage and performs thread-level synchronizations between stages. This approach allows us to make extensive use of shared memory, while reducing the pressure on L1 and L2 data caches.

32-bit versus 64-bit Word Lengths The choice of word length is associated with a trade-off between operational complexity and workload size (i.e., the number of NTTs, or *limbs*, in the workload). Figure 5 presents performance metrics for both 32-bit and 64-bit implementations over various values of the modulus Q. The evaluation is based on two assumptions. First, we fix the NTT size as $N = 2^{16}$. In practice, the value of N depends on the value of Q. Second, we model the relationship between the number of limbs ℓ and the modulus length m as $\ell = \lceil \frac{Q}{m} \rceil$. As seen in Figure 5, the baseline NTT runtimes for the 32-bit and 64-bit implementations are similar. This result aligns with prior work [12] that found a “negligible difference” in the runtimes associated with these word lengths.

We see how our DRAM throughput is impacted by the use of shared memory. In our 2D NTT implementation, we use shared memory throughout the entire computation, whereas the baseline implementation requires five stages to read coefficients from global memory directly.

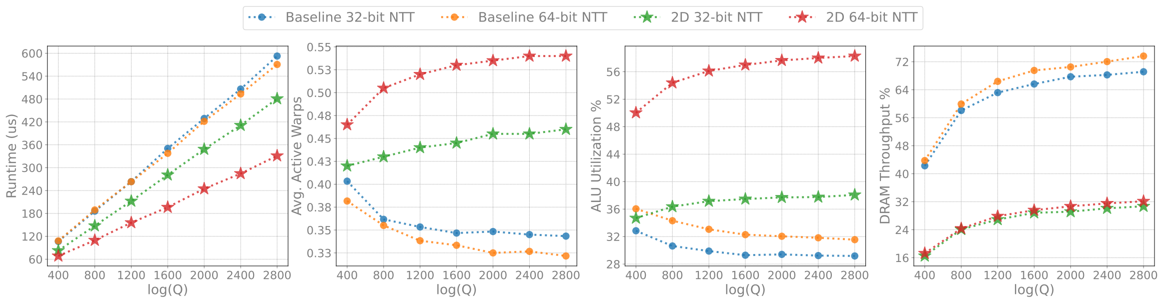

Figure 5. Performance metrics for 32-bit and 64-bit implementations of the Number Theoretic Transform for various values of $\log(Q)$.

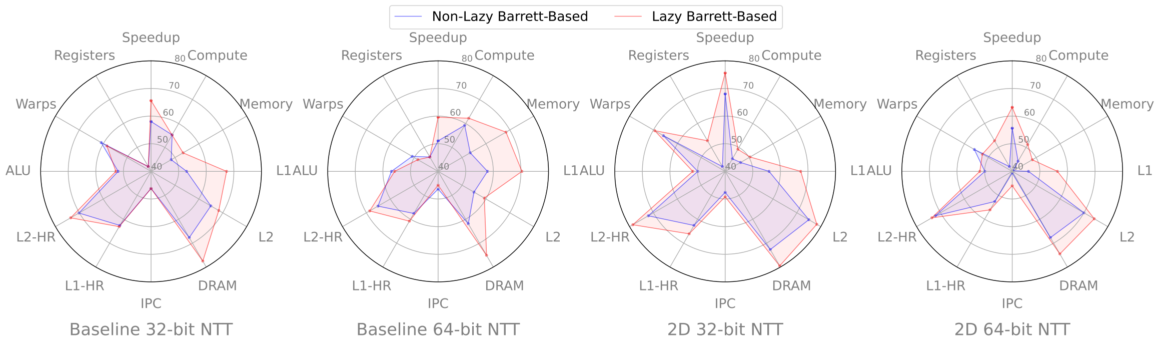

Figure 6. Performance of lazy and non-lazy Barrett-based NTTs (as compared to an implementation using CUDA's builtin % construct for modular reduction).

The resulting latency from those accesses reduces the number of active warps as they must wait for data to arrive, resulting in a degradation of compute throughput.

In 2D NTT, 64-bit operations take more cycles to execute than 32-bit operations, thus 64-bit operations are more effective in hiding memory latencies. In addition, we find that the reduced workload size resulting from using 64-bit operations further reduces the impact of extra execution cycles associated with 64-bit operations.

Comparison of $1/N$-Scaling Methods If $\mathbf{a}$ and $\mathbf{b}$ are polynomials, their product can be computed via the following expression: $\frac{1}{N}\mathrm{iNTT}(\mathrm{NTT}(\mathbf{a}) \odot \mathrm{NTT}(\mathbf{b}))$, where NTT denotes the Number Theoretic Transform, iNTT its inverse, and $\odot$ denotes the Hadamard product. In this section, we compare the performance of three methods of performing the $1/N$-*scaling*: two which are commonly used, and a third method using pre-computation. The naive approach simply scales the output of iNTT by $1/N$.

However, the $1/N$-scaling does not need to occur at the end. It is straightforward to verify that scaling commutes with butterfly operations, stages in NTTs, and entire NTTs (and their inverses). This enables a second approach to perform $1/N$-scaling, which consists of scaling the output of each butterfly by $1/2$. We refer to this approach as *butterfly scaling*. Note that $1/2$-scaling can be efficiently computed on the GPU's ALU via bit-shifts and sums. A third approach involves moving half of these $1/2$-scalings to pre-computation in the twiddle factor array, which we refer to as *twiddle factor scaling*.

We collected performance metrics for each of these three methods, for both our 32-bit and 64-bit baseline inverse NTT implementations. As expected, the naive $1/N$-scaling was the slowest. Butterfly scaling provided 1.29% and 3.85% speedups respectively. Twiddle factor scaling provided the most significant speedups at 3.13% and 4.86% speedups respectively. This was also to be expected, as much of the computation is moved into pre-computation.

Lazy Arithmetic To investigate the performance improvements provided by our lazy Barrett-based NTT algorithm, we collected performance metrics for both lazy and non-lazy variants for both the baseline and 2D NTT kernels. Our results are summarized in Figure 6. In our baseline implementations, laziness provides 4.3% and 6.7% speedups for our 32-bit and 64-bit baseline implementations, respectively. In our 2D implementations, laziness gives 2.6% and 9.0% speedups respectively. This performance improvement is largely due to an increase in memory throughput (DRAM, L1 cache, and L2 cache throughput), as shown in Figure 6.

Conclusion

Finite field arithmetic operations are significant computational bottlenecks in Fully Homomorphic Encryption, impeding the widespread deployment of FHE (and lattice-based cryptography more generally) in real-world systems. We present optimizations to classical algorithms that can decrease the number of correctional subtractions required, resulting in significant performance improvements. In addition, in highly parallel workloads such as Number Theoretic Transforms, we can obtain further speedup by carefully mapping kernels to many-core systems (e.g., GPUs). We found that applying all of the proposed optimizations processes a $2.08\times$ speedup over a state-of-the-art NTT implementation. For FHE to be deployed in next-generation systems, there still remain performance barriers to overcome. Some potential paths forward include using larger word sizes, as well as leveraging multi-GPU systems.

Acknowledgment

This work was supported in part by the Institute for Experiential AI, the Harold Alfond Foundation, the NSF IUCRC Center for Hardware and Embedded Systems Security and Trust (CHEST), the RedHat Collaboratory, and project grant PID2020-112827GB-I00 funded by MCIN/AEI/10.13039/501100011033.

REFERENCES

1. V. Lyubashevsky, C. Peikert, and O. Regev, “On ideal lattices and learning with errors over rings,” *Advances in Cryptology – EUROCRYPT 2010*, Lecture Notes in Computer Science, vol. 6110, pp. 1–23, 2010.
2. 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,” *IEEE Int. Symp. on Secure and Private Execution Environ. Des. (SEED)*, pp. 61–72, 2022.
3. M. Scott, “A note on the implementation of the number theoretic transform,” *IMA Int. Conf. on Cryptography and Coding – IMACC 2017*, Lecture Notes in Computer Science, vol. 10655, pp. 247–258, 2017.
4. P. Barrett, “Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor,” *Advances in Cryptology – CRYPTO’ 86*, Lecture Notes in Computer Science, vol. 263, pp. 311–323, 1986.
5. J.-F. Dhem and J.-J. Quisquater, “Recent results on modular multiplications for smart cards,” *Smart Card Research and Applications*, Lecture Notes in Computer Science, vol. 1820, pp. 336–352, 1998.
6. N. Emmart, F. Zheng, and C. Weems, “A new variant of the Barrett algorithm applied to quotient selection,” *IEEE 25th Symp. on Comput. Arithmetic (ARITH)*, pp. 138–144, 2018.
7. A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, R. V. Saraswathy, K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, “OpenFHE: open-source fully homomorphic encryption library,” *Proc. of the 10th Workshop on Encrypted Comput. and Applied Homomorphic Cryptography*, pp. 53–63, 2022.
8. T. Pöppelmann, T. Oder, and T. Güneysu, “High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers,” *Progress in Cryptology – LATINCRYPT 2015*, Lecture Notes in Computer Science, vol. 9230, pp. 346–365, 2015.
9. D. Harvey, “Faster arithmetic for number-theoretic transforms,” *J. Symb. Comp.*, vol. 60, pp. 113–119, 2014.
10. Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savaş, “Efficient number theoretic transform implementation on GPU for homomorphic encryption,” *J. Supercomput.*, vol. 78, no. 2, pp. 2840–2872, 2022.
11. D. H. Bailey, “FFTs in external or hierarchical memory,” *Proc. of the 1989 ACM/IEEE Conf. on Supercomput.*, vol. 4, no. 1, pp. 234–242, 1990.
12. S. Kim, W. Jung, J. Park, and J. H. Ahn, “Accelerating number theoretic transformations for bootstrappable homomorphic encryption on GPUs,” *IEEE Int. Symp. on Workload Characterization (IISWC)*, pp. 264–275, 2020.