CryptoPIM: In-memory Acceleration for Lattice-based Cryptographic Hardware

Hamid Nejatollahi*§, Saransh Gupta†§, Mohsen Imani†
Tajana Simunic Rosing†, Rosario Cammarota‡, Nikil Dutt*
*University of California, Irvine, USA
†University of California, San Diego, USA
‡ Intel Labs, USA

***Abstract*—Quantum computers promise to solve hard mathematical problems such as integer factorization and discrete logarithms in polynomial time, making standardized public-key cryptosystems insecure. Lattice-Based Cryptography (LBC) is a promising post-quantum public key cryptographic protocol that could replace standardized public key cryptography, thanks to the inherent post-quantum resistant properties, efficiency, and versatility. A key mathematical tool in LBC is the Number Theoretic Transform (NTT), a common method to compute polynomial multiplication. It is the most compute-intensive routine and requires acceleration for practical deployment of LBC protocols. In this paper, we propose CryptoPIM, a high-throughput Processing In-Memory (`PIM`) accelerator for NTT-based polynomial multiplier with the support of polynomials with degrees up to 32k. Compared to the fastest FPGA implementation of an NTT-based multiplier, CryptoPIM achieves on average 31x throughput improvement with the same energy and only 28% performance reduction, thereby showing promise for practical deployment of LBC.**

***Index Terms*—Lattice-based Cryptography, Acceleration, Number Theoretic Transform, Homomorphic Encryption, Processing in Memory**

I. Introduction

Shor's algorithm can solve integer factorization and discrete logarithm in polynomial time [1], which gives quantum computers the ability to break standardized public-key cryptosystems based on RSA and ECC. The response of the cryptography community to such an urgent threat is the introduction and evaluation of quantum-resistant algorithms - in the families of lattice-, multivariate-, hash-, code-, and isogeny-based cryptography [2]. Moreover, the National Institute of Standards and Technology (NIST) announced a contest in 2017 to evaluate and recommend candidates for the standardization of quantum-resistant public-key cryptosystems. Among the candidate schemes, Lattice-Based Cryptography (`LBC`) schemes are the most promising due to their versatility and superior performance for building quantum-resistant security mechanisms such as digital signature and key agreement. Besides the basic public-key encryption schemes, `LBC` lays the foundations to construct protocols to compute on the encryption data, i.e., homomorphic encryption [3].

Secure `Standard LBC` schemes involve expensive matrix-vector multiplications and large key sizes (e.g., 11kB for Frodo [4]). In contrast, `Ideal LBC` schemes are variants that provide superior performance and memory footprint by performing the computation on polynomial rings without compromising security. The hardness of such schemes is based on the Ring Learning With Error (RLWE) [5] problem, in which polynomial multiplication is the most time-consuming routine. Polynomial multiplication is usually performed using the Number Theoretic Transform (NTT), a variant of the Discrete Fourier Transform (DFT) for polynomial rings on finite arithmetic fields [6]. The process is slow due to the repeated use of expensive operations like multiplication and modulo reduction. Fortunately, polynomial operations carry a significant amount of parallelism that can be leveraged with suitable hardware architecture.

§Hamid Nejatollahi and Saransh Gupta contributed equally to this research.

An attractive solution for leveraging parallelism in highly-parallel applications on data inputs with a large memory footprint is to enable memory to perform the computation, rather than hinder performance due to expensive data movements [7]. This approach, often referred to as Processing In-Memory `PIM`, reduces communication costs between the processor and memory by performing most computations directly in memory. Emerging Non-Volatile Memory (NVM) technology, e.g., Resistive Memory - memristor - can be effectively used for various PIM techniques such as in-memory search [8], bit-wise and addition operations [9], [10]. This enables the memory to perform computations while serving sufficient storage utilizing superior characteristics of the NVM, such as high density, low-power consumption, and scalability [11]. Previous work has shown that PIM is highly energy-efficient and fast for data-intensive tasks [12], [13], [14].

In this work, we propose the design of a high-throughput in-memory architecture for NTT-based polynomial multiplication, for its use to accelerate `LBC` cryptosystems defined on the RLWE problem. This work makes the following contributions:

- This work is the first to propose a PIM architecture for NTT-based polynomial multiplication, to the best of the authors' knowledge.
- This work proposes an architecture for fast in-memory multiplication and modulo operations in-memory.
- This work proposes `NTT`-specific fixed-function switches to overcome the irregular memory access issue in NTT. Our switches enable parallel data transfer between multiple `NTT` stages.
- This work proposes the design of a configurable pipelined architecture, which can support as large as 32k, accommodating requirements for public-key cryptographic systems for data at rest and in communication, and data in use via homomorphic encryption cryptosystems defined on RLWE lattices, e.g., BGV.

The rest of the paper is organized as follows: Section II provides the background for `LBC` and NTT-based multipliers; Sections III describes the CryptoPIM. Section IV discusses the results. Finally, we conclude the paper in Section V.

II. Background and Related Work

A lattice $L \subset \mathbb{R}^n$ is defined as all the integer linear combinations of basis vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^n$. The hardness of lattice-based schemes are based on two mathematically hard problems: short integer solution (SIS) and, more commonly, learning with errors (LWE). Given the pair (A,pk) as a matrix of constants sampled uniformly at random in $\mathbb{Z}_q^n$ and the public key, the learning with error problem is defined as finding the secret key sk, where $pk = (A * sk + e) \bmod q$, and e is a small error vector that is sampled from a Gaussian distribution.

LWE-based schemes are impractical to be implemented on resource-constrained devices due to their large keys. At the same security level, Ring-LWE (RLWE) reduces the key size by a factor of n, where n is the degree of the polynomial ring. In Ring-LWE (RLWE), a derivation of LWE in which A is implicitly defined as a vector a in the ring $\mathcal{R} \equiv \mathbb{Z}_q[x]/\langle x^n + 1 \rangle$. Arithmetic operations for a Ring-LWE-based scheme are

Algorithm 1 NTT-based Polynomial Multiplier

```
Initialization: w is the n-th root of unity and φ is the 2n-th root of unity
  (φ^2 = w mod q); w^{-1} and φ^{-1} are the inverse of w modq and φ modq,
  respectively.
Precompute: {w^i, w^{-i}, φ^i, φ^{-i}}     ▷ w^i, w^{-i} are in reversed order,
  while φ^i, φ^{-i} are in normal order
function POLY_NTT(A, B)
    bitrev(A); bitrev(B)
    for i = 0 to n − 1 do
        ā_i ← a_i φ^i; b̄_i ← b_i φ^i
    end for
    Ā ← NTT_GS(ā, w)
    B̄ ← NTT_GS(b̄, w)
    C̄ = Ā.B̄
    bitrev(C̄)
    c̄ ← NTT_GS(C̄, w^{-1})
    for i = 0 to n − 1 do
        c_i ← c̄_i φ^{-i}
    end for
    Return C
end function
```

performed over a Z_p, the ring of integers modulo p where n (degree of the polynomial) is a power of two, p is a large prime number, and $x^n + 1$ is an irreducible polynomial degree n. The quotient ring R_p includes polynomials with degree less than n in Z_p, that defines $R_p = Z_p/[x^{n+1}]$ in which coefficients of polynomials are in [0,p). Degrees of the polynomials in RLWE-based schemes vary between 256 [15] and 1024 [16] for public-key encryption and between 4k and 32k for homomorphic encryption evaluation [17].

The polynomial multiplication operation is commonly computed using the Number Theoretic Transform (NTT). Two polynomials ($a = a(n-1) \cdot x^{n-1} + \ldots + a(0)$ and $b = b(n-1) \cdot x^{n-1} + \ldots + b(0)$) are first transformed into the NTT domain ($A = A(n-1) \cdot x^{n-1} + \ldots + A(0)$ and $B = B(n-1) \cdot x^{n-1} + \ldots + B(0)$). Second, the multiplication of the two transformed polynomials is computed by performing coefficient-wise multiplications, i.e., as follows: $C = \sum_{i=0}^{n-1} A(i) \cdot B(i) \cdot x^i$. The final result, $c = a * b$, is computed by applying the the inverse number theoretic transform (NTT^{-1}) on C. A common method to perform the number theoretic transform is Gentleman-Sande (GS) [18], which receives the input in the reverse order and produces the output in the normal order. Similar to [19], we employ the GS method to compute both forward and inverse number theoretic transforms. It involves changing the order of the coefficients in the vector representation (i.e., bit-reverse). Algorithm 1 describes the NTT-based polynomial multiplier using the GS method.

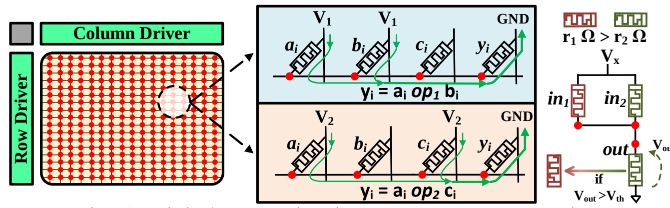

Fig. 1: Digital processing in memory (PIM) overview.

Algorithm 2 The Gentleman-Sande in-place NTT algorithm

```
To compute the NTT and NTT^{-1}, twiddle is set to {w^i} and {w^{-i}}
  for all i ∈ [0, n/2−1], respectively. Output is A in the frequency domain
  (bit-reversed order).
function NTT_GS(A,twiddle)
    for i=0 to log2n do
        for idx=0 to n/2 do
            st ← idx & ((1<<i)-1)
            j ← ((idx & !(1<<i - 1) <<1)&(n-1) + st
            j' ← j + (1<<i)
            W ← twiddle[j>>(i+1)]
            T ← A[j]
            A[j] ← (T + A[j'])mod q
            A[j'] ← W*(T - A[j'])mod q
        end for
    end for
end function
```

Numerous researches focus on implementations of classical cryptographic schemes for servers and constraint devices [20], [21]. In the scope of LBC schemes, researches on the acceleration of the NTT-based polynomial multiplications narrow the focus on the area and performance of the polynomial multiplier and leave the energy unexplored [22] [23].

Efforts have evaluated the energy as well as the area and performance of NTT accelerators [24][25][26][27]. We compare our results to the fastest FPGA implementation of the NTT-based multiplier in [19] for the vector sizes $n \in$ (256, 512, 1024) in terms of performance, energy, and throughput. Besides, we report the performance, energy, and throughput for vector sizes $n \in$ (256, 512, 1024, 2k, 4k, 8k, 16k, 32k) to show how scalable is the proposed design.

While the works mentioned above try to accelerate NTT, they do not perform well for higher degree polynomials. Processing higher degree polynomial, even with NTT, involves a massive amount of computations. Hence, the application performance suffers due to (i) less than required on-chip memory and (ii) limited availability of complex cores.

Prior work has proposed processing in memory (PIM), which is an architecture for performing in-situ computations, mainly in memory [7], [28], [12]. Recent works in PIM enable highly efficient bitwise operations in memory [10], [28] and extend the operations to implement complex functions like floating-point arithmetic [12]. Besides, PIM-based architectures promise to provide large dense memory and extensive parallel computing capacity. Figure 1 shows a high-level implementation of a PIM based architecture for bitwise computation [10]. The memory block on the left is a crossbar of ReRAM (resistive RAM) cells, where cells in the same row share a wordline, and cells in the same column share a bitline. Such cells have two possible states. The cells change state when the voltage across them crosses the device threshold. A memory cell is present at the intersection of a wordline and bitline. To implement bitwise functions, it applies a voltage V at the input bitlines and ground the output bitline. The result of the computation is generated in the output cell. The operation performed is dependent on V. Furthermore, the same operation can be executed in parallel over multiple rows of the block, enabling parallel vector-wide operations.

In the prior work, domain-specific PIM blocks have been proven effective in applications using MapReduce (based on 3D stacking) [29], nearest neighbor search (using computational logics beside DRAM) [8], and parallel graph processing (based on 3D DRAM) [13]. Many software interface designs have also been proposed for heterogeneous computing platforms to use the accelerators in systems with coherent access to the host memory, e.g., IBM CAPI [30], Intel HARP [31], and HMC 2.0 [32].

In this work, we, for the first time, propose CryptoPIM, a high-throughput PIM accelerator for `NTT`-based polynomial multiplication. The proposed design optimizes the basic `NTT` operations, introduced new inter-block switches, and finally uses a configurable architecture to support polynomial multiplications for polynomials with degrees higher than ever supported on an accelerator, i.e., up to 32k and beyond.

III. CryptoPIM for RLWE Polynomial Multiplier

In this section, we detail CryptoPIM, a PIM design for NTT-based polynomial multiplication. We start by analyzing the base algorithms to identify the primitive operations involved in the execution of an RLWE polynomial multiplication. Then we show how these operations can be architected in a PIM to present a high-speed and highly efficient architecture for NTT-based polynomial multiplier.

A. RLWE Polynomial Multiplication Primitives

Algorithm 1 illustrates an algorithm to perform RWLE polynomial multiplication. Most notably, Algorithm 1 contains two NTT for the polynomial to multiply, element-wise multiplications between the coefficients of the transformed polynomials, and an inverse NTT to reduce the result to the corresponding output polynomial. The `bitrev()` only changes the sequence of data reads and not the data itself. Algorithm 2 shows that NTT is further composed of element-wise vector addition, subtraction, and multiplication operations. Hence, NTT-based polynomial multiplication essentially comprises of bit-reversal and element-wise modular addition, subtraction, and multiplication. Each arithmetic operation is followed by a modulo operation (`mod q`).

B. PIM-based polynomial arithmetic

Section II shows that a PIM based designs [12], [28], [33], [34] can implement arithmetic functions with high parallelism and energy efficiency. Moreover, using such designs for modular arithmetic, as required in our case, can further increase their benefits. Integer operations do not involve tracking the decimal point (for fixed-point) or iterative data-dependent shifts (for floating-point). This simple computation logic of integer operations, combined with the vector-based operations in polynomial multiplication, make it a suitable candidate for PIM.

*1) **Data organization in CryptoPIM:*** A memory block is an array of memory cells, where each memory cell represents a bit. N continuous memory cells in a row represent an N-bit number, with the first cell storing the Most Significant Bit (MSB). For a block with r rows and c columns, each row stores c/N numbers, with the entire block having a capacity of $(c/N) \times r$ N-bit numbers.

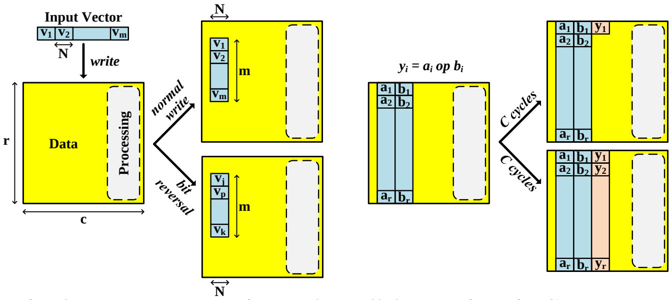

Fig. 2: Data representation and parallel operations in CryptoPIM.

In PIM, each row has some data columns and some processing columns. While data columns store inputs and other relevant data like pre-computed factors, processing columns are primarily used for performing intermediate operations and storing temporary results. However, the data and processing columns are physically indistinguishable and their roles can be changed on-the-fly. An input vector with m N-bit values is stored in data columns such that each N-bit number occupies the same N columns in m rows. This is illustrated in Figure 2

*2) **Polynomial multiplication in CryptoPIM:*** We implement the functions in polynomial multiplication to PIM as follows:

Bit-reversal: Bit-reversal changes the sequence of data read. In the case of PIM, where an input vector is stored over different rows in a memory block, a bit-reversal operation is equivalent to changing the row to which a value is written (shown in Figure 2. This can be easily achieved while writing the vector to the block. The arrangement can either be hard-coded or be flexible according to the target application.

Addition/Subtraction: The state-of-the-art PIM designs perform vector-wide addition. For subtraction, 2's complement is taken for one of the inputs (subtrahend) and then the addition is performed. We use similar techniques where basic bitwise operations are cascaded to implement a 1-bit adder. Then, these multiple such 1-bit additions are used to implement an N-bit operation. Although a single N-bit addition/subtraction may be a slow operation, r of such operations can be executed in parallel in a $r \times c$ memory block without any additional overhead, as shown in Figure 2. The latency of N-bit addition is $6N + 1$ cycles [10] and for subtraction is $7N + 1$.

Multiplication: An N-bit multiplication operation is broken into a simple shift and addition of partial products. First, the partial products are generated using bitwise operations and stored in the same row as operands. Because CryptoPIM works with bit-level memory access, instead of explicit data shift, shifting operation is translated to selecting appropriate columns of the memory block. Similar to addition/subtraction, r multiplication operations can also execute in parallel in a memory block which provides efficient vector-wide operations. The work in [35] proposed full-precision multipliers. However, the bitwise operations used by them are expensive. Instead, we combine the algorithm in [35] with the low latency bitwise operations proposed in [10]. As a result, N-bit multiplication in CryptoPIM takes $6.5N^2 - 11.5N + 3$ cycles, significantly less than the $13N^2 - 14N + 6$ cycles of [35].

Modulo: While the addition of two N-bit numbers can result in an $(N + 1)$-bit output, a multiplication may give an output with $2N$ bits. However, to maintain the consistency in number bit-width with minimum effect on the algorithmic correctness, each computation undergoes a modulo operation. Modulo operations traditionally involve expensive division operations. To enable modulo operations in memory, we use Barrett reduction [36] and Montgomery reduction [37] after

Algorithm 3 Reduction algorithms using shifts and adds. $q = 7681$ for $n \leq 256$, $q = 12289$ for n=512, 1024, and $q = 786433$ for n$\geq$ 2048

```
1:  function BARRETT REDUCE(a)
2:      if q == 12289 then
3:          u ← ((a<<2)+(a)) >> 16; u ←(u<<13) + (u<<12) + u
4:      end if
5:      if q == 7681 then
6:          u ← a>>13; u ← (u<<13) - (u<<9) - u
7:      end if
8:      if q == 786433 then
9:          u ← a>>20; u ← (u<<19) + (u<<18) + u
10:     end if
11:     Return (a − u)
12: end function
13: function MONTGOMERY REDUCTION(a)
14:     if q == 12289 then
15:         u ← (a<<13) + (a<<12) - a; u ← u & ((1<<18)-1)
16:         u ← (u<<13) + (u<<12) + u; u ← (u + a)>>18
17:     end if
18:     if q == 7681 then
19:         u ← (a<<13) - (a<<9) + a; u ← u & ((1<<18)-1)
20:         u ← (u<<13) - (u<<9) - u; u ← (u + a)>>18
21:     end if
22:     if q == 786433 then
23:         u ← (a<<19) - (a<<18) + a; u ← u & ((1<<32)-1)
24:         u ← (u<<19) + (u<<18) - u; u ← (u + a)>>32
25:     end if
26:     Return u
27: end function
```

each addition and multiplication, respectively. Further, in NTT, the modulo factor (q in our case) is generally fixed for a specific size of the vector or polynomial's degree. For instance, we set $q = 12289$ for vector sizes of 1024 and 512, according to NewHope [16], and $q = 7681$ for vector sizes of 256 and less according to Kyber [15]. For vector sizes of 2k, 4k, 8k, 16k, and 32k we set $q = 786433$ according to the Microsoft SEAL library [17]. We exploit this limited set of possible qs to make reductions more efficient in PIM. Instead of naively using multiple multiplications, we first convert these reduce operations into the successive shift and add/subtract operations as shown in Algorithm 3. Since we have bit-level access, we perform only the necessary bit-wise computations. For example, in the line 15 of Algorithm 3, $\{v \leftarrow u$ & ((1<<18)-1)$\}$ resets to 0 all but the 17 LSBs of the output of $\{u \leftarrow$ (a<<13) - (a<<9) + a$\}$. Here, we compute only 17 LSBs of u.

Since q is the same for all the values in a vector, they require same shift (i.e. same column activations in CryptoPIM) and can hence be completely parallelized over the entire vector in CryptoPIM. For each reduction, latency is dependent on the total number of bitwise additions involved in it. Table I lists the latency of each reduction in CryptoPIM.

TABLE I: Execution time (cycles) for modulo operation

q	7681	12289	786433
Barrett Reduction	261	239	429
Montgomery Reduction	683	461	1083

C. CryptoPIM Building Blocks

The two basic building blocks of CryptoPIM are PIM-enabled memory block and fixed-function switch. A memory block implements one phase or stage of the polynomial multiplication. Two adjacent memory blocks are connected using the CryptoPIM fixed-function switches as shown in Figure 3 Each memory block is a PIM enabled an array of 512 × 512 memory cells and can process a vector of length 512 at a time.

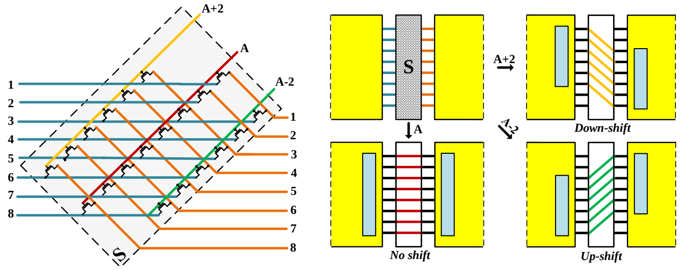

Fig. 3: (Left) Detailed circuit of fixed-function switch with 8 inputs/outputs and shift factor of 2. (Right) All possible connections for s=2.

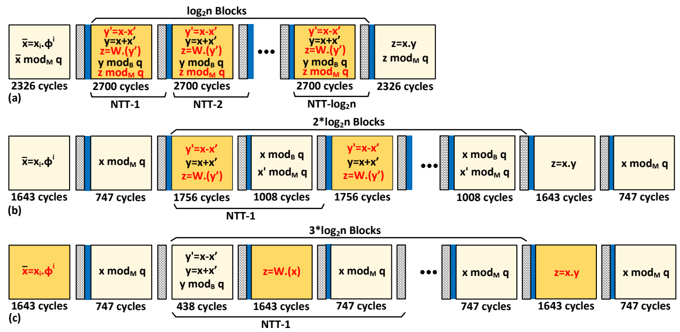

Fig. 4: Detailed stage-by-stage breakdown of (a) Area-efficient pipeline, (b) naive pipeline, and (c) CryptoPIM pipeline. The region between two consecutive blue bars represents a pipeline stage. The slowest stage in a pipeline is colored darker. The slowest operation or sequence of operations in the pipeline is colored red.

The fixed-function switches, unlike typical crossbar switches, enable only three types of connections: a direct rowA-to-rowA, rowA-to-row(A+s), rowA-to-row(A-s). A fixed-function switch allows just one, the in-hardware-coded value of s. However, this 's' may be different (but fixed) for different instances of this fixed-function switch in the hardware. Instead, traditional crossbar switches provide a connection between any possible input/output combination leading to an exponential increase in logic requirement with an increase in inputs or outputs. Our fixed-function crossbar switch has just three logic switches per row. The number of switches per row is independent of the number of inputs/outputs. These switches can read up to one entire column of one block and write it to the next block in parallel, requiring as many cycles as the bit-width of data. Hence, transferring data between two blocks in NTT requires only 3*bitwidth cycles, one each for A-to-A, A-to-(A+s), and A-to-(A-s) transfer.

All computation steps in Algorithm 1 and 2 are implemented using these building blocks. Each vector-wide data operation along with the involved modulo reduction is implemented in an independent memory block. Hence, we have a memory block each for $a_i\phi^i$, $b_i\phi^i$, $\bar{A}.\bar{B}$, $\bar{c}_i\phi^{-i}$. Then, the log_2n stages in NTT uses a memory block each. Each (NTT) stage block is connected to the next stage block using the switches.

D. CryptoPIM Architecture

1) CryptoPIM Pipeline: The block-by-block modular architecture provides an opportunity to increase the throughput by pipelining. Figure 4a shows the most area-efficient NTT pipeline for the 16-bit data-width. Here, the computation and its corresponding modulo operation are performed in the same block. For 16-bit datatype and n=256, this results in 2700 cycles per stage. Note that even though the logic performed in each stage have different latencies, the latency of the slowest stage determines the stage latency while pipelining. Now, the data computation and its modulo are completely independent operations and can be performed in separate blocks, leading to the pipeline shown in Figure 4b and stage latency of 1756 cycles. However, this comes at the cost of increasing the number of stages and hence, the total latency of one polynomial multiplication. We further optimized the pipeline by combining Montgomery reduction, addition/subtraction, and Barrett reduction in the same stage as shown in Figure 4c. We obtain the final stage latency of 1643 cycles.

2) Configurable Architecture: CryptoPIM consists of a ReRAM memory chip with several memory banks. A set m cascaded memory blocks map to one memory bank. A memory bank takes in 512 parallel inputs in the first block and output 512-element wide vector. Hence, it can only process polynomials with degrees up to 512. However, the degree of the polynomials in RLWE/FHWE-based schemes generally ranges up to 32k. We design CryptoPIM architecture such that many of these banks can be dynamically arranged in the form of b_{soft} *softbanks*. A softbank consists of b_m parallel memory banks. Each softbank is responsible for processing vector-wide operations for a polynomial. Then, two softbanks dynamically form a *superbank* which completely processes multiplication between two input polynomials. To enable this configurability, CryptoPIM uses additional switches at the intersection of different banks and softbanks to allows data communication between them.

We optimize our hardware to support 32k degree polynomials in memory. A 32k NTT pipeline has 49 blocks (from Figure 4). Hence, each bank has 49 memory blocks. We further need 64 such memory banks for each input polynomial, requiring 128 memory banks per 32k polynomial multiplication. If the degree of input polynomial is higher than 32k, CryptoPIM divides the inputs into segments of 32k and iteratively uses the hardware to compute to one polynomials portion at a time. On the other hand, if the input polynomial degree is less than 32k, we dynamically configure CryptoPIM into multiple superbanks to enable parallel multiplication of multiple polynomial pairs.

IV. Evaluation

A. Evaluation Setup

We use an in-house cycle-accurate C++ simulator, which emulates CryptoPIM functionality. We used HSPICE for circuit-level simulations and calculate energy consumption and performance of all the CryptoPIM operations in the 45nm process node. We used System Verilog and Synopsys *Design Compiler* to implement and synthesize the CryptoPIM controller. The robustness of all proposed circuits, i.e., interconnect, has been verified by considering 10% process variations on the size and threshold voltage of transistors using 5000 Monte Carlo simulations. A maximum of 25.6% reduction in resistance noise margin was observed for RRAM devices. However, this did not affect the operations in CryptoPIM due to the high R_{OFF}/R_{ON}. We adopt an RRAM device with VTEAM model [38]. The device parameters [9] are chosen to fit practical devices [39] with switching delay of 1.1ns (= cycle time in CryptoPIM). The behavior of bitwise PIM operations has been demonstrated using a fabricated chip in [40].

B. Performance and Energy Consumption of CryptoPIM

Figure 5 shows the latency and throughput of non-pipelined and pipelined CryptoPIM over different degrees n of polynomial multiplications. The latency of CryptoPIM increases with

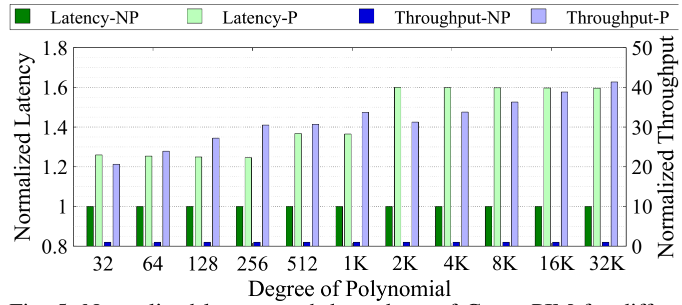

Fig. 5: Normalized latency and throughput of CryptoPIM for different degrees of polynomial. NP and P represent non-pipelined and pipelined designs respectively.

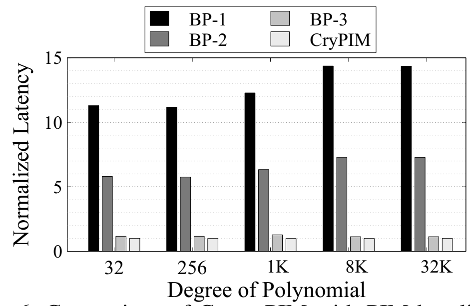

Fig. 6: Comparison of CryptoPIM with PIM baselines.

an increase in n, primarily due to the increased number of NTT stages. However, the pipelined-throughput remains the same for the degrees processed in the same bit-width. This is because the latency of one stage in CryptoPIM depends on bit-width and not n. The energy consumption of the design increases along with n. This is due to an increase in both the number of stages and well as the number of parallel computations in each stage.

As evident from the results, pipelining increases the throughput tremendously with some latency overhead. For smaller degrees (n $\leq$ 1024), average throughput improves by 27.8$\times$, while incurring 29% latency overhead. When n $>$ 1024, the throughput improvement increases to 36.3$\times$, with 59.7% increment in latency. This happens because the latency of multiplication increases exponentially with the bit-width of inputs. For $n > 1024$ (32-bit inputs), the execution time of multiplication is 6.8$\times$ that of the second slowest operation. On the other hand, for n $\leq$ 1024 (16-bit inputs), multiplication is just 2.3$\times$ slower than the second slowest operation. Hence, the pipeline is comparatively more balanced in 16-bit than 32-bit. The energy consumption of CryptoPIM increases on average by just 1.6% for the pipelined design. While pipelining increases the number of stages, the underlying computations don't increase. Hence, the total amount of logic is the same in both the pipelined and non-pipelined versions. The small increase in energy is due to increased block-to-block transfers.

C. Comparison with state-of-the-art PIM

To show the efficiency of our optimizations, we compare CryptoPIM with multiple PIM baselines. The first baseline PIM (BP-1) uses the operations proposed in [35] while utilizing the same building blocks and architecture as CryptoPIM. BP-2 is BP-1 with its N-bit multiplication replaced with the multiplication in CryptoPIM. BP-3 is BP-2 with the reduction operations converted to shift and adds. Figure 6 shows the latency of the three baselines and CryptoPIM for different degrees of polynomial multiplication. To have a fair comparison, we compare the baselines with the non-pipelined version of the design. We observe that BP-2 is on average 1.9× faster than BP-1. This shows that optimized CryptoPIM multiplications significantly improve CryptoPIM latency. Moreover, BP-3 is 5.5× faster than BP-2, which shows that the shift and add based reduction is more efficient than multiplication based reduction. Finally, CryptoPIM is 1.2× faster than BP-3, showing that CryptoPIM modulo reductions are highly optimized. As a result, CryptoPIM is 12.7× faster than the state-of-the-art PIM (BP-1).

D. Comparison with CPU and FPGA

Table II shows the comparison of the pipelined-CryptoPIM in terms of latency, energy, and throughput to the FPGA (on Xilinx Zynq UltraScale+) and software (on an X86 CPU at 2GHz) implementations. Compare to the CPU implementation, CryptoPIM on average achieves 7.6x, 111x, and 226x improvement in the performance, throughput, and energy, respectively. For the sizes suitable for public-key encryption (256, 512, and 1024) CryptoPIM achieves on average 31x throughput improvement with the same energy and less than %30 reduction in the performance.

TABLE II: Comparison of the CryptoPIM to the FPGA and CPU implementation of the NTT-based polynomial multiplier. Throughput is defined as number of the polynomial multiplications per seconds. Energy is the require energy to multiply two polynomials.

Design	N	Bitwidth	Latency (us)	Energy (uJ)	Throughput
X86 (gem5)	256	16	84.81	570.60	11790
	512	16	168.96	1179.52	5918
	1k	16	349.41	2483.77	2861
	2k	32	736.92	5273.07	1365
	4k	32	1503.31	10864.64	665
	8k	32	3066.76	22385.51	326
	16k	32	6256.20	46123.84	159
	32k	32	12762.65	95032.33	78
NTT-based [19] (FPGA)	256	16	21.56	2.15	46382
	512	16	47.63	5.28	20995
	1k	16	101.84	12.52	9819
	2k-32k	-	-	-	-
CryptoPIM Pipelined	256	16	68.67	2.58	553311
	512	16	75.90	5.02	553311
	1k	16	83.12	11.04	553311
	2k	32	363.60	82.57	137511
	4k	32	392.69	178.62	137511
	8k	32	421.78	384.17	137511
	16k	32	450.87	822.21	137511
	32k	32	479.95	1752.15	137511

V. Conclusion

The (NTT) is the most time-consuming routine in ideal lattice-based cryptographic protocols. In this paper, we proposed a high-throughput PIM accelerator for NTT-based polynomial multiplication. CryptoPIM, enables fast execution of the polynomial multiplication with the support of polynomials with degrees up to 32k, accommodating requirements for public-key cryptographic systems for data at rest and in communication, and data in use, i.e., homomorphic encryption. CryptoPIM achieves on average 31x throughput improvement with the same energy consumption and 28% latency reduction as compared to the fastest NTT-based polynomial multiplier implemented on FPGA.

Acknowledgments

This work was supported in part by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, SRC-Global Research Collaboration grant, and also NSF grants 1527034, 1730158, 1826967, and 1911095.

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Jour on Comp*, 1997.
[2] H. Nejatollahi *et al.*, "Post-quantum lattice-based cryptography implementations: A survey," *ACM CSUR*, 2019.
[3] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *STOC*, 2009.
[4] J. Bos *et al.*, "Frodo: Take off the ring! practical, quantum-secure key exchange from lwe," in *CCS*, 2016.
[5] V. Lyubashevsky *et al.*, "On ideal lattices and learning with errors over rings," in *EUROCRYPT*, 2010.
[6] H. Nejatollahi *et al.*, "Trends, challenges and needs for lattice-based cryptography implementations: Special session," in *CODES*, 2017.
[7] M. Gokhale *et al.*, "Processing in memory: The terasys massively parallel pim array," *Computer*, 1995.
[8] C. C. del Mundo *et al.*, "Ncam: Near-data processing for nearest neighbor search," in *MEMSYS*, 2015.
[9] N. Talati *et al.*, "Logic design within memristive memories using memristor-aided logic (magic)," *NANO*, 2016.
[10] S. Gupta *et al.*, "Felix: Fast and energy-efficient logic in memory," in *ICCAD*, 2018.
[11] Y. Xie, *Emerging Memory Technologies: Design, Architecture, and Applications*, 2013.
[12] M. Imani *et al.*, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *ISCA*, 2019.
[13] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, 2016.
[14] D. Lavenier *et al.*, "Dna mapping using processor-in-memory architecture," in *BIBM*, 2016.
[15] R. Avanzi *et al.*, "Crystals-kyber," NIST, Tech. Rep., 2017.
[16] T. Poppelmann *et al.*, "Newhope," NIST, Tech. Rep., 2017.
[17] H. Chen *et al.*, "Simple encrypted arithmetic library - seal v2.1," Cryptology ePrint Archive, 2017.
[18] W. M. Gentleman *et al.*, "Fast fourier transforms: For fun and profit," in *AFIPS*, 1966.
[19] H. Nejatollahi *et al.*, "Exploring energy efficient quantum-resistant signal processing using array processors," Cryptology ePrint Arch, 2019.
[20] J. Toldinas *et al.*, "Energy efficiency comparison with cipher strength of aes and rijndael cryptographic algorithms in mobile devices," 2011.
[21] S. A. Ansarmohammadi *et al.*, "Fast and area efficient implementation for chaotic image encryption algorithms," *CADS*, 2015.
[22] T. Pöppelmann *et al.*, "Accelerating homomorphic evaluation on reconfigurable hardware," in *Lecture Notes in Computer Science*, 2015.
[23] D. B. Cousins *et al.*, "An fpga co-processor implementation of homomorphic encryption," in *HPEC*, 2014.
[24] H. Nejatollahi *et al.*, "Synthesis of flexible accelerators for early adoption of ring-lwe post-quantum cryptography," *TECS*, 2020.
[25] H. Nejatollahi *et al.*, "Domain-specific accelerators for ideal lattice-based public key protocols," *Cryptology ePrint Archive*, 2018.
[26] U. Banerjee *et al.*, "Sapphire: A configurable crypto-processor for post-quantumlattice-based protocols," *TCHES*, 2019.
[27] H. Nejatollahi *et al.*, "Flexible ntt accelerators for rlwe lattice-based cryptography," *ICCD*, 2019.
[28] D. Fujiki *et al.*, "Duality cache for data parallel acceleration," in *ISCA*, 2019.
[29] S. H. Pugsley *et al.*, "Ndc: Analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads," in *ISPASS*, 2014.
[30] J. Stuecheli *et al.*, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, 2015.
[31] P. K. Gupta, "Xeon+ fpga platform for the data center," in *CARL*, 2015.
[32] M. Technology, "Hybrid memory cube," 2017.
[33] N. Talati *et al.*, "mmpu—a real processing-in-memory architecture to combat the von neumann bottleneck," in *Applications of Emerging Memory Technology*, 2020.
[34] S. Gupta *et al.*, "Exploring processing in-memory for different technologies," in *GSVLSI*, 2019.
[35] A. Haj-Ali *et al.*, "Imaging-in-memory algorithms for image processing," *TCAS I*, 2018.
[36] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *CRYPTO*, 1986.
[37] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, 1985.
[38] S. Kvatinsky *et al.*, "Vteam: A general model for voltage-controlled memristors," *TCAS II*, 2015.
[39] S. Kvatinsky *et al.*, "Magic memristor aided logic," *TCAS II*, 2014.
[40] B. C. Jang *et al.*, "Memristive logic-in-memory integrated circuits for energy-efficient flexible electronics," *Adv Functional Materials*, 2018.