

BP-NTT: Fast and Compact in-SRAM Number Theoretic Transform with Bit-Parallel Modular Multiplication

Jingyao Zhang*, Mohsen Imani[†], Elaheh Sadredini*

*University of California, Riverside [†]University of California, Irvine
{jzhan502, elahehs}@ucr.edu, m.imani@uci.edu

Abstract—Number Theoretic Transform (NTT) is an essential mathematical tool for computing polynomial multiplication in promising lattice-based cryptography. However, costly division operations and complex data dependencies make efficient and flexible hardware design to be challenging, especially on resource-constrained edge devices. Existing approaches either focus on only limited parameter settings or impose substantial hardware overhead. In this paper, we introduce a hardware-algorithm methodology to efficiently accelerate NTT in various settings using in-cache computing. By leveraging an optimized bit-parallel modular multiplication and introducing costless shift operations, our proposed solution provides up to $29\times$ higher throughput-per-area and $10\text{--}138\times$ better throughput-per-power compared to the state-of-the-art.

I. INTRODUCTION

With the rise of cloud computing and the Internet of Things (IoT), concerns about data privacy and security are escalating, especially for vulnerable edge devices. Lattice-based cryptography is the most promising candidate to serve as the foundation of future information security due to its superior balance of security and operational speed. Currently, three of four NIST-standardized post-quantum cryptography algorithms (PQCs) [1] and almost all homomorphic encryption (HE) schemes are based on lattice-based cryptography. Typically, lattice-based cryptography is primarily based on the hardness of two problems: module learning with error (for PQCs) and ring learning with error (for HEs). The algorithms based on these two problems involve polynomial operations, such as modular addition and modular multiplication. With a complexity of $O(N^2)$, the modular multiplication of polynomials is the most time-consuming operation.

To mitigate the computing bottleneck, number-theoretic transform (NTT) is commonly employed to accelerate polynomial modular multiplication with the principles of the Fast Fourier Transform (FFT), which lowers the computing complexity of polynomial multiplication to $O(N\log N)$. However, the complicated data dependencies among different NTT stages and the required division operations make efficient hardware-based acceleration challenging.

To accelerate NTT effectively, ASIC/FPGA-based hardware acceleration designs are proposed [2]–[4]. Although performance is enhanced, they still suffer from frequent data movement between processing components and memory, which inhibits further performance growth. To eliminate the data movement bottleneck, in-memory computing techniques for cryptography algorithms are proposed [5]–[10]. The chal-

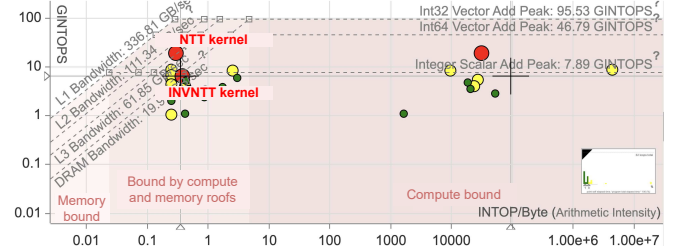


Fig. 1: Roofline model for lattice-based cryptography.

lenges with existing solutions are that they (1) expand the trusted computing base to off-chip memories [8], thus, introducing security vulnerabilities; (2) introduce complex peripheral circuits [8], [9], thus, incurring high area overhead; or (3) are specialized only for NTT processing [10], thus, sacrificing generality and flexibility. These restrictions make it even more challenging to enable secure computing on vulnerable and resource-constrained edge devices.

To analyze the computational bottleneck of the NTT, modular multiplication, and reduction (these kernels count for more than 50% of the computation in PQC algorithms based on our profiling on CPU) and to answer the question of "where is the best place to compute in the memory hierarchy?", we first generate the roofline model of the lattice-based cryptography algorithms, such as CRYSTAL-Dilithium [11] and CRYSTAL-Cyber [12] using Intel Advisor [13] (Fig. 1). Our observation is that the main performance bottleneck for these kernels is the L1 and L2 bandwidth, and they are not bounded by the memory bandwidth bottleneck.

Based on these insights, we re-purpose existing on-chip 6T SRAM arrays into large vector computation units and co-design them with a novel bit-parallel modular multiplication algorithm and our proposed implicit shift operations to enable energy-efficient, fast, and low-overhead NTT acceleration, especially for the IoT devices. Our proposed solution, BP-NTT (Bit-Parallel NTT), addresses the inefficiency of existing schemes while preserving safety and flexibility. Since only the chip itself can be considered a trusted computing base with any off-chip data requiring encryption [14], our proposed solution provides data confidentiality by not offloading the plaintext to off-chip memories. Enabled by our proposed data organization and modular multiplication, a single 256×256 SRAM subarray in BP-NTT design can support up to a 250-point polynomial with 256-bit coefficients or a 4500-point polynomial with 14-bit coefficients, which covers requirements of the lattice-based

PQC algorithms (256/1024-point polynomial with 14/16/32-bit coefficients) [11], [12], [15] and three security levels of HE under the BKZ.qsieve model (1024-point polynomial with 16/21/29-bit coefficients) [16].

In summary, the paper contributes the following:

- We present a compact and low-overhead in-SRAM NTT acceleration design while preserving the generality (i.e., capable of expanding to other crypto kernels) and flexibility (i.e., to easily adjust the bitwidth, polynomial order, and modulus). Moreover, these arrays incur minimal hardware modification compared to conventional SRAM arrays (less than 2%) and can be used for normal cache operations when they are not used as a crypto accelerator.
- To provide a compact and high-throughput computation, we present a bit-parallel data layout, which enables a costless shift for $\sim 50\%$ of the shift operations in NTT and its inverse (in other words, #shifts in our bit-parallel design is half of the prior bit-serial solutions). In addition, our design exploits the inspiration from carry-save adder, which eliminates the carry propagation operation in SRAM arrays and enables higher parallelism.
- Through simulation, we validate the correctness of the proposed bit-parallel modular multiplication algorithm. Our evaluations reveal that BP-NTT achieves up to an order of magnitude higher throughput-per-area and up to two orders of magnitude higher throughput-per-power compared to the state-of-the-art in-memory, ASIC, and FPGA solutions, on the same technology node and similar parameter settings, thus, making it a low-cost and energy-efficient option for edge devices.

II. BACKGROUND

A. Ring Learning with Errors (R-LWE)

The R-LWE problem is whether it is possible to create $pk = a * sk + e$ given a public vector pk and a vector, where a is a uniformly sampled vector across $\mathcal{R} \equiv \mathbb{Z}_q[x] / \langle x^n + 1 \rangle$, e is a Gaussian distributed error vector of small absolute value, n is a power of 2, q is a prime number, and sk is a secret vector. Consequently, when utilizing a method based on lattice-based cryptography, each encryption/decryption must conduct at least one polynomial multiplication.

B. Number Theoretic Transform (NTT)

In general, NTT – which is a generalization of DFT over quotient rings – utilizes the principles of FFT to lower the complexity of polynomial multiplication from $O(N^2)$ to $O(N \log N)$, where N is the number of polynomial terms. Algorithm 1 illustrates the popular in-place Cooley-Tukey NTT [17]. The algorithm receives the polynomial a as input and the n -th roots on \mathcal{R} , and yields the NTT-transformed polynomial a as its output. Then, polynomial a can be multiplied with the NTT-transformed polynomial b element-by-element, and the result will be converted from NTT format to standard format using inverse NTT, as $ab = \text{NTT}^{-1}(\text{NTT}(a) * \text{NTT}(b))$.

To avoid the costly division required by modular multiplication in NTT, Montgomery multiplication [18] is generally

Algorithm 1 Cooley-Tukey-based In-place NTT Algorithm.

Input: $a = (a_{n-1}, \dots, a_0) \in \mathcal{R}$, and pre-computed n -th roots $\zeta_0, \dots, \zeta_{n-1}$ of unity w_n in \mathbb{Z}_q with bit-reversed order
Output: $a = \text{NTT}(a)$ in bit-reversed order

```

1:  $k = 0$ 
2: for  $\text{len} = n/2$ ;  $\text{len} > 0$ ;  $\text{len} \gg 1$  do
3:   for  $\text{idx} = 0$ ;  $\text{idx} < n$ ;  $\text{idx} = j + \text{len}$  do
4:      $z = \zeta[+ + k]$ 
5:     for  $j = \text{idx}$ ;  $j < \text{idx} + \text{len}$ ;  $j = j + 1$  do
6:        $t = z * a[j + \text{len}] \bmod q$ 
7:        $a[j + \text{len}] = a[j] - t \bmod q$ 
8:        $a[j] = a[j] + t \bmod q$ 

```

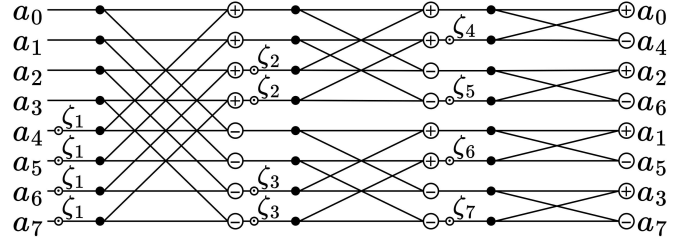


Fig. 2: Cooley-Tukey butterfly with 3-stage communication.

used. In NTT, the multipliers ζ_k is pre-computed, which helps avoid the cost of Montgomery domain transformation [19]. One characteristic of NTT is the intricate data communication between adjacent stages. As depicted in Fig. 2, for an 8-point polynomial, the NTT requires three stages, and the data dependencies between the stages are complicated, resulting in increased data movement, long wires, and higher area overhead when designing ASIC and FPGA-based hardware accelerators. For in-memory architecture, this inefficiency is translated to a higher number of shift operations to align the data for bitline computation.

C. Computing in SRAM

In-SRAM processing is capable of bitline computing [20] by activating more than one row in the SRAM sub-array. A diagram of the logical operations that can be performed in 6T SRAM is shown in Fig. 3. AND and NOR operations are realized in

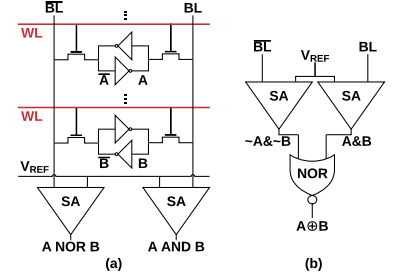


Fig. 3: In-SRAM bitline operations.

several activated wordlines and sense amplifiers (SAs), as displayed in Fig. 3(a). If all the cells in the active rows that are wired to the bitline BL have the value ‘1’, then the SA on the BL will be able to detect a voltage larger than V_{ref} , which is an element-wise AND operation. Only if all the cells in the activated rows connected to the corresponding \overline{BL} contain ‘1’, the SA on the \overline{BL} senses a voltage greater than V_{ref} , which in turn requires that all the cells in the activated rows connected to the corresponding BL contain ‘0’. This means the SA can do element-wise NOR operations.

As can be seen in Fig. 3(b), the XOR operation can be accomplished by combining the capabilities of the logical bit-wise AND and NOR operations. Cache Automaton [21] uses a sense-amplifier cycling mechanism to read out multiple bits in a single time slot, hence reducing input symbol match time. Compute Cache [22] increases the number of logical operations by modifying the SA architecture based on the NOR, AND, and XOR operations stated in [20]. In BP-NTT, we utilize the XOR capability presented in [20] and 1-bit shifting (details in Section IV-C).

III. RELATED WORK

ASIC/FPGA solutions: Ni et al. [2] reduce lattice-based algorithm latency by accelerating Montgomery multiplication with a three-stage pipeline and systolic array. Banerjee et al. [3] construct a reconfigurable cryptographic processor with a low-power modular arithmetic core to improve hardware overhead and energy efficiency. Xing et al. [4] improve latency and hardware complexity by proposing a negative wrapped convolution method to execute polynomial multiplication on FPGAs. Although the above approaches increase performance, they still suffer from frequent PE-to-memory data movement. Moreover, their designs are limited to a few parameter settings.

In-Memory solutions: Nejatollahi et al. [10] propose CryptoPIM, a ReRAM-based NTT accelerator that uses a *Shift-Add-based* reduction scheme with the Gentleman-Sande algorithm. However, (1) the fixed interconnection among ReRAM arrays increases hardware overhead and makes it inflexible to support other cryptography kernels, and (2) it only supports a limited number of moduli which limits the flexibility. Park et al. [9] present RM-NTT based on vector-matrix multiplication instead of FFT-like computation to reduce latency. However, it increases memory footprint and energy consumption. In general, ReRAM-based solutions suffer from device variation, which results in reliability problems. In addition, with in-ReRAM computing, the data is located off-chip in plaintext, which makes it vulnerable to memory and bus attacks. Li et al. [8] present MeNTT with a new modular multiplication method and NTT hardware mapping. However, the fixed routing among SRAM arrays and heavy near-memory peripheral circuitry introduces large area/energy overhead and inflexibility. Unlike prior work, this paper aims to design a flexible, secure, high-performance, low-overhead, and energy-efficient NTT accelerator.

IV. IMPLEMENTATION

A. Overview

BP-NTT can be realized either by re-purposing a portion of the L1/L2/L3 cache or by placing a separate in-SRAM accelerator next to the on-chip caches. Fig. 4 represents an integration of BP-NTT in the last-level cache (LLC). Each LLC slice has several SRAM banks, and each bank has usually four subarrays. We repurposed one subarray for memory-mapped command/control instructions (*CTRL/CMD*) and the rest as vector computing units to perform NTT computation, as shown in Fig. 4(b). Different banks performing the same operations can share *CTRL/CMD* subarray. The instruction sets

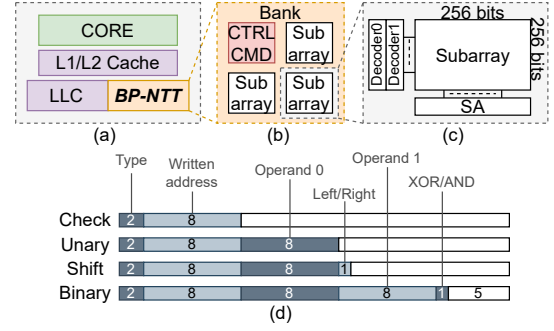


Fig. 4: (a-c) Hierarchical view of BP-NTT-enabled system. (d) Control signals for different operations in BP-NTT.

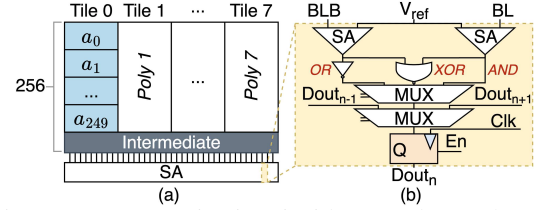


Fig. 5: (a) Data organization inside one data subarray. (b) The structure of the sense amplifier supporting OR/XOR/AND logic operation and 1-bit left/right shift.

for the BP-NTT are designed and represented in Fig. 4(d). Our design incurs minimal hardware changes to the SRAM arrays, i.e., less than 2% area overhead compared to the conventional SRAM, thus, enabling a low-overhead solution.

B. Data Organization

In BP-NTT, polynomial coefficients are arranged in distinct rows of the same tile. Fig. 5(a) shows eight independent polynomials in eight tiles within the same array, with 250 rows for coefficients and 6 rows for intermediate variables. In this configuration, each row in a tile stores 32-bit polynomial coefficients. Our design can be easily reconfigured to accommodate n tiles with $\lfloor 256/n \rfloor$ -bit coefficients. When the number of polynomial coefficients is less than the tile's capacity, we can place coefficients from other polynomials in unused rows to save memory. On the other hand, if the number of polynomial coefficients exceeds the tile's capacity, the excess coefficients can be stored in adjacent tiles and merged during computation using the 1-bit shift operation.

Unlike prior methods method [23], our method doesn't require shift operations to compute distinct coefficients (i.e., costless shift) thanks to our tile-based data layout that place coefficients in the rows of the same tile so that they can share the bitlines. This means that we can simply select the required coefficient by having their row addresses as the input of the decoders. Although bit-serial alignment doesn't require shifting [24] as well, it requires long columns (e.g., 4096 rows for a 128-point 32-bit polynomial), which is very uncommon, especially for resource-constrained edge devices.

C. Sense Amplifier Design

As shown in Fig. 5(b), we modify conventional SRAM SAs to enable in-memory bit-parallel modular multiplication and

Algorithm 2 In-memory Bit-Parallel Modular Multiplication

Input: n -bit $A = (a_{n-1}, \dots, a_0)$, $B = (b_{n-1}, \dots, b_0)$, $M < R = 2^n$, where $n > 2$ and $M \perp R$
Output: $ABR^{-1} \bmod M$

```

1:  $Sum := (s_{n-1}, \dots, s_0) = 0$  // Initialize
2:  $Carry := (c_{n-1}, \dots, c_0) = 0$ 
3:  $P := Sum + Carry << 1$  //  $P = 0$ 
4: for  $i = 0, n-1$  do
5:   if  $a_i == 1$  then // Implicit compare
6:      $c1, s1 = \{Sum \& B, Sum \oplus B\}$ 
7:      $Carry << 1$  // ← Observation 1
8:      $c2, Sum = \{Carry \& s1, Carry \oplus s1\}$ 
9:      $Carry = c1 \mid c2$  //  $P = P + a_i B$ 
10:   end if
11:    $m = (LSB(Sum) == 1) ? M : 0$  //  $m = M$  or  $0$ 
12:    $c1, s1 = \{Sum \& m, Sum \oplus m\}$ 
13:    $s1 >> 1$  // ← Observation 2
14:    $c2, s2 = \{s1 \& c1, s1 \oplus c1\}$ 
15:    $c3, Sum = \{Carry \& s2, Carry \oplus s2\}$ 
16:    $Carry = c2 \mid c3$  //  $P = P + m; P >> 1$ 
17: end for

```

addition/subtraction in NTT. By using NOR and inverse gates, BP-NTT can perform XOR and OR. A MUX and a latch are introduced to implement 1-bit bidirectional shift operations.

D. In-memory Bit-parallel Modular Multiplication

With the aforementioned data arrangement, a polynomial can fit the data compactly into a subarray, which is not possible with a bit-serial design. Another benefit of our suggested BP-NTT is its capacity to conduct bit-parallel modular multiplication. Traditional carry-propagation-based multiplication [8], [24] is not suitable for in-memory parallel computation because once carry propagation is introduced, the higher bit must wait for the carry propagation of the lower bits. This hinders maximally exploiting in-memory computing's high parallelism for efficiency.

BP-NTT takes advantage of our proposed in-memory bit-parallel modular multiplication algorithm so that it can perform operations in bit-parallel (e.g., on a 32-bit word) which reduces the latency. Our proposed algorithm (Algorithm 2) is based on Montgomery's algorithm for modular multiplication, which can produce $ABR^{-1} \bmod M$ for the input polynomials A and B , where R is 2^n , n is the bitwidth, and M is the modulo. Notably, although our approach does not directly output $AB \bmod M$ (same as Montgomery algorithm), the twiddle factors can be pre-computed by multiplying them to R in advance to make the final result ($AB = (AR)BR^{-1} \bmod M$) as expected without extra conversion.

As shown in Algorithm 2, after initialization, our algorithm determines whether to add partial sum P to multiplier B based on the presence of '1' in A . In our design, twiddle factor A is hidden in the control commands. Control commands are generated from twiddle factors and stored before starting the NTT computation. For example, if $P = P + B$ is performed on the third iteration, the third bit a_2 of twiddle factor A must be 1. In line 11, the least significant bit (LSB) of Sum is used to determine m 's value. If $LSB = 1$, $m = M$; otherwise,

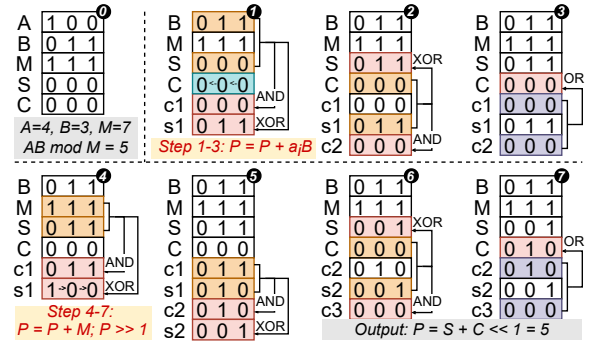


Fig. 6: A 3-bit example of the proposed in-memory bit-parallel modular multiplication.

$m = 0$. The partial sum is then added to m and shifted to the right. We can get an n -bit Sum and $Carry$ after n iterations.

The proposed algorithm uses two variables, Sum and $Carry$, to represent the sum of two addends to enable in-memory bit-parallel modular multiplication. Inspired by carry-save adder design [27], Sum and $Carry$ are derived directly using bitwise operations (AND, XOR, OR) and kept distinct throughout the computation.

The number of bits in the vanilla algorithm can be reduced from $n + 1$ to n based on two observations: (1) the highest bit of the $Carry$ is always 0 after each iteration; and (2) the lowest bit of the result in $P = P + M$ is always 0. Thus by shifting $Carry$ 1-bit to the left (line 7) and $s1$ 1-bit to the right (line 13), all the computations can be done within n columns, which is crucial for exploiting parallelism in in-SRAM computing. A 256-column SRAM array with 32-bit operands can only perform seven modular multiplications in parallel if $n + 1$ bits (33 columns) is required, whose throughput is 12.5% worse than our proposed method.

Fig. 6 depicts an example of bit-parallel modular multiplication in SRAM. Inputs are $A = 4$, $B = 3$, and $M = 7$. We use A to directly represent AR because $A = AR \bmod M$. Due to the lowest two bits of A , P remains 0 after two iterations. In step 1 of the third iteration, B and Sum are bit-parallelly ANDed and XORed to generate $c1$ and $s1$. $Carry$ is shifted to the left by one bit to align with Sum for subsequent addition operations. Step 2 performs AND and XOR on $Carry$ and $s1$ to produce $c2$ and Sum . In step 3, $c1$ and $c2$ are ORed to get $Carry$. In step 4, M and Sum produce $c1$ and $s1$. For addition, $s1$ is shifted to the right by one bit. Steps 5-7 are the same as 1-3. Finally, we can obtain the result $P = 001 + 010 << 1 = 5$.

E. Implicit Shift in NTT

Inefficient hardware and complex data dependencies make NTT acceleration challenging. The product of modular multiplication should be added to or subtracted from polynomial coefficients (lines 7-8 in Algorithm 1), which is difficult with in-memory computing due to shifting overhead. The implicit shift design prevents word shifting. Thanks to our data organization, which places all polynomial coefficients in the same tile, we can perform bitline computing by implicitly

TABLE I: Comparing BP-NTT with state-of-the-art solutions on a 256-point polynomial.

	Design	Coef. Bitwidth	Max f (MHz)	Latency (μ s)	Tput. (KNTT/s)	Energy (nJ)	Area (mm ²)	Tput./Area (KNTT/s/mm ²)	Tput./Power (KNTT/mJ)
BP-NTT (45nm)	In-SRAM	16	3.8K	61.9	258.6	34	0.063	4.1K	230.7
MeNTT (45nm*) [8]	In-SRAM	14	218	15.9	62.8	47.8	0.173	364	20.9
CryptoPIM (45nm) [10]	ReRAM	16	909	68.7	553.3	2.6K	0.152†	3.6K	14.7
RM-NTT (45nm*) [9]	ReRAM	14	249	0.45	2.2K	602	0.289†	7.7K	1.67
LEIA (45nm*) [25]	ASIC	14	267	0.6	1.7K	44.1	1.77	940.6	22.7
Sapphire (45nm*) [3]	ASIC	14	64	20.1	49.7	236.3	0.354	140.1	4.23
FPGA (45nm*) [26]	FPGA	16	164	24.3	41.2	3061	-	-	-
CPU [10]	x86	16	2K	85	11.8	570K	-	-	-

* Technology nodes are projected to 45nm for an apples-to-apples comparison with BP-NTT.

† These solutions do not provide the area consumption. For the sake of comparison, we optimistically estimate their area overhead based on the area of memory subarrays used in these designs (i.e., we ignore the peripheral overhead).

selecting the corresponding row of operands, thus, avoiding shift operations. If we want to add the product of modular multiplication with the k -th polynomial coefficient, we only need to activate the rows where the product and the n -th coefficient are located, to enable bitline computing and perform the standard addition with 1-bit shifts. As a result, compared to MeNTT [8], our BP-NTT consumes less energy due to the reduced shifting overhead.

V. EVALUATION

A. Evaluation Methodology

In this section, we analyze the latency, throughput, area, energy, throughput-per-area, and throughput-per-power of BP-NTT performing NTT for 256-point polynomials with 14-bit and 16-bit coefficients (as they are the most common parameter choices in PQC algorithms [11], [12]), and compare them to existing ASICs, FPGAs, and in-memory designs, as shown in Table I. The correctness of the proposed bit-parallel modular multiplication has been validated for various bitwidths. The array size of BP-NTT is 256×256 following the ARM Cortex-M0+ microcontroller [28]. We then investigate BP-NTT performance on different parameter configurations. PyMTL3 [29] and OpenRAM [30] are utilized to construct SRAM arrays, and Synopsys Design Compiler and Cadence Innovus are utilized to calculate read and write latency and area consumption, respectively. Additionally, due to the lack of area consumption analysis in CryptoPIM [10] and RM-NTT [9], we utilize the Destiny simulator [31] to optimistically estimate only the subarray areas, and we do not account for their complex peripheral circuitry.

B. Memory Footprint and Area Comparison

BP-NTT occupies the least amount of memory footprint and introduces the least amount of hardware overhead compared to other in-memory designs. Fig. 7 shows the data layout of BP-NTT, MeNTT, and RM-NTT for computing NTT with 32-bit, 128-point polynomial. BP-NTT requires only 4288 SRAM cells (i.e., 134 rows and 32 columns), however, MeNTT and RM-NTT require 16,640 (130 rows and 128 columns) and 524,288 (128 rows and 4096 columns) ReRAM cells, respectively. In addition, BP-NTT is capable of performing all NTT-related operations within an SRAM subarray, thanks

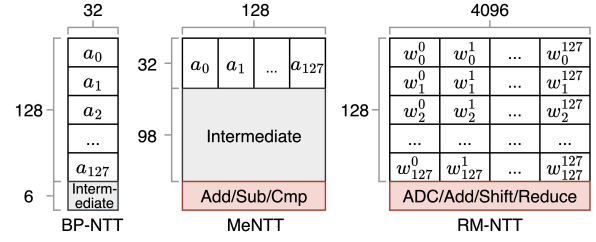


Fig. 7: Comparison of different in-memory designs for NTT on a 32-bit 128-point polynomial.

to our compact data organization and the proposed bit-parallel modular multiplication. However, MeNTT requires peripheral circuitry for addition, subtraction, and comparison operations, and RM-NTT requires additional shift and reduction modules, resulting in a large area and energy consumption overhead.

As shown in Table I, BP-NTT provides at least $2.4 \times$ – $4.6 \times$ lower area overhead compared to the state-of-the-art in-memory designs. This is because their data arrangement and operation mode (i.e., bit-serial computing in MeNTT and vector-matrix-based NTT in RM-NTT) necessitates more memory capacity to support NTT acceleration.

C. Throughput-per-Area & Throughput-per-Power

BP-NTT is a design that exhibits the highest throughput-per-power (TP) when compared to other designs. This is due to its energy efficiency, high performance, and minimal overhead, rendering it a suitable candidate for IoT devices. The proposed bit-parallel modular multiplication and data organization enable costless shift operations, contributing to BP-NTT's superior performance. In addition, BP-NTT provides excellent throughput-per-area (TA) by making minimal modifications to existing standard arrays, and it only requires one array to perform several NTTs with different parameters, resulting in a low-overhead design. In general, ReRAM-based in-memory designs offer higher throughput due to their efficient analog computing capabilities. However, their computing schemes, including cascaded pipeline and vector-matrix-based NTT, require a large number of arrays and memory cells, resulting in high energy consumption. BP-NTT outperforms ASIC/FPGA designs by up to $30 \times$ and $50 \times$ in TA and TP metrics, respectively, due to the latter's frequent data movement.

D. Latency Comparison

The higher latency of BP-NTT compared to ASIC/FPGA designs is due to the fact that ASIC/FPGA can fine-tune algorithms so that unnecessary operations on the critical path can be minimized for certain algorithms and parameter settings. On the other hand, hardware generalization and flexibility are sacrificed. In-memory designs have lower latency than BP-NTT because to handle inefficient modular multiplication and complex inter-stage data communication, either they target a specific modulus and fixed inter-array connections, or they introduce a large number of combinational logic circuits (comparators, adders, subtractors, and even Montgomery reduction module), thereby, sacrificing flexibility and adding unnecessary area overhead. In contrast, BP-NTT can accomplish all NTT operations within one subarray with minimal modifications to the SA and without any extra dedicated modules.

E. Flexibility Analysis

The BP-NTT is flexible to support NTT calculations with different parameters. Figure 8(a) shows the clock count and energy overhead for a 256×256 BP-NTT design plus 6 rows for intermediate data using 2-bit to 64-bit coefficients with a polynomial order of 256. As the bitwidth increases, both the clock count and energy overhead grow. The reason for the steeper increase in energy overhead is that as bitwidth grows, the number of NTTs that can be computed in parallel in a fixed-size subarray decreases.

Fig. 8(b) shows the number of clock cycles and energy consumption for performing NTT on BP-NTT with different polynomial orders at a bitwidth of 16. The steeper increase in clock count and energy consumption compared to Fig. 8(a) is because as the polynomial order rises, not only does the number of NTTs that can be computed in parallel in the subarray decrease but also the additional shift overhead in a single NTT is introduced. However, these additional overheads can be effectively avoided by using the larger subarray or interconnection of multiple subarrays.

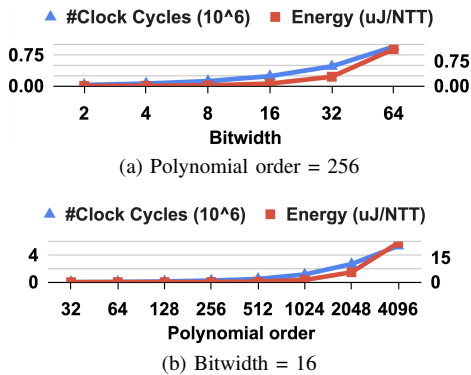


Fig. 8: Performance and energy consumption of BP-NTT (a) with different bitwidths when polynomial order is 256, (b) with different polynomial orders when bitwidth is 16.

VI. CONCLUSION

In this paper, we present BP-NTT, an in-SRAM architecture for flexible, secure, high-performance, low-overhead, energy-

efficient acceleration of NTT. By utilizing bit-parallel modular multiplication, we take advantage of the parallelism of in-SRAM computing to enhance performance. By incorporating our proposed data organization, the majority of shift operations can be eliminated with minimal hardware modifications. Our evaluation results indicate that BP-NTT can achieve a significant improvement in throughput-per-power (up to $138\times$) over the latest ASIC/FPGA and in-memory designs.

VII. ACKNOWLEDGMENTS

This work is funded, in part, by the Hellman Fellowship from the University of California, NSF #2127780, Semiconductor Research Corporation, and Office of Naval Research.

REFERENCES

- [1] Selected algorithms 2022 - post-quantum cryptography [online].
- [2] Z. Ni, D.-E.-S. Kundi, M. O'Neill, and W. Liu, "High-performance systolic array montgomery multiplier for sike," in *ISCAS*, 2021.
- [3] U. Banerjee, T. S. Ukyab *et al.*, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *arXiv*, 2019.
- [4] Y. Xing and S. Li, "A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga," *IACR TCHES*, 2021.
- [5] J. Zhang, H. Naghibijouybari *et al.*, "Sealer: In-SRAM AES for high-performance and low-overhead memory encryption," in *ISLPED*, 2022.
- [6] J. Zhang and E. Sadredini, "Inhale: Enabling high-performance and energy-efficient in-SRAM cryptographic hash for IoT," in *ICCAD*, 2022.
- [7] Z. Wang, C. Liu, A. Arora, L. John *et al.*, "Infinity stream: Portable and programmer-friendly in-/near-memory fusion," in *ASPLOS*, 2023.
- [8] D. Li *et al.*, "Mentt: A compact and efficient processing-in-memory number theoretic transform (ntt) accelerator," *IEEE VLSI*, 2022.
- [9] Y. Park, Z. Wang, S. Yoo *et al.*, "Rm-ntt: An rram-based compute-in-memory number theoretic transform accelerator," *IEEE JXDC*, 2022.
- [10] H. Nejatollahi, S. Gupta, M. Imani *et al.*, "Cryptopim: In-memory acceleration for lattice-based cryptographic hardware," in *DAC*, 2020.
- [11] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky *et al.*, "Crystals-dilithium: A lattice-based digital signature scheme," *IACR TCHES*, 2018.
- [12] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky *et al.*, "Crystals-kyber: a cca-secure module-lattice-based kem," in *EuroS&P*, 2018.
- [13] Design code for parallelism and offloading with intel® advisor [online].
- [14] V. Costan *et al.*, "Intel sgx explained," *Cryptology ePrint Archive*, 2016.
- [15] P.-A. Fouque, J. Hoffstein, P. Kirchner *et al.*, "Falcon: Fast-fourier lattice-based compact signatures over ntru," *NIST CSRC*, 2018.
- [16] M. Albrecht, M. Chase *et al.*, "Homomorphic encryption standard," in *Protecting Privacy through Homomorphic Encryption*, 2021.
- [17] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, 1965.
- [18] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, 1985.
- [19] O. Mazonka, E. Chieffe *et al.*, "Fast and compact interleaved modular multiplicationbased on carry save addition," in *ICCAD*, 2022.
- [20] S. Jeloka *et al.*, "A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory," *IEEE JSSC*, 2016.
- [21] A. Subramaniyan, J. Wang *et al.*, "Cache automaton," in *MICRO*, 2017.
- [22] S. Aga, S. Jeloka *et al.*, "Compute caches," in *HPCA*, 2017.
- [23] Y. Zhang, L. Xu, Q. Dong, J. Wang, D. Blaauw *et al.*, "Recryptor: A reconfigurable cryptographic cortex-m0 processor with in-memory and near-memory computing for iot security," *IEEE JSSC*, 2018.
- [24] C. Eckert, X. Wang, J. Wang, A. Subramaniyan *et al.*, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *ISCA*, 2018.
- [25] S. Song, W. Tang, T. Chen, and Z. Zhang, "Leia: A 2.05 mm² 140mw lattice encryption instruction accelerator in 40nm cmos," in *CICC*, 2018.
- [26] H. Nejatollahi *et al.*, "Exploring energy efficient quantum-resistant signal processing using array processors," in *ICASSP*, 2020.
- [27] J. G. Earle, "Latched carry save adder circuit for multipliers," 1967.
- [28] LPC1225fbd48 | arm cortex-m0 | 32-bit MCU | NXP [online].
- [29] S. Jiang *et al.*, "Pymtl3: A python framework for open-source hardware modeling, generation, simulation, and verification," *MICRO*, 2020.
- [30] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "Openram: An open-source memory compiler," in *ICCAD*, 2016.
- [31] M. Poremba, S. Mittal, D. Li, J. S. Vetter *et al.*, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *DATE*, 2015.