# NTT-PIM: Row-Centric Architecture and Mapping for Efficient Number-Theoretic Transform on PIM

Jaewoo Park*, Sugil Lee† and Jongeun Lee†§
*Department of Physics, †Department of Electrical Engineering
Ulsan National Institute of Science and Technology (UNIST), Ulsan, Korea
{hecate64,sglee17,jlee}@unist.ac.kr

***Abstract*— Recently DRAM-based PIMs (processing-in-memories) with unmodified cell arrays have demonstrated impressive performance for accelerating AI applications. However, due to the very restrictive hardware constraints, PIM remains an accelerator for simple functions only. In this paper we propose NTT-PIM, which is based on the same principles such as no modification of cell arrays and very restrictive area budget, but shows state-of-the-art performance for a very complex application such as NTT, thanks to features optimized for the application's characteristics, such as in-place update and pipelining via multiple buffers. Our experimental results demonstrate that our NTT-PIM can outperform previous best PIM-based NTT accelerators in terms of runtime by $1.7 \sim 17\times$ while having negligible area and power overhead.**

***Keywords*—Processing-in-memory (PIM), fully homomorphic encryption (FHE), number theoretic transform (NTT), DRAM, row buffer**

## I. Introduction

The recent success of AI has caused important changes in the computing landscape, one of which being the renewed interest in PIM (processing-in-memory) [1]–[3]. Though the idea of PIM dates back to 70's, recent approaches to AI-PIM by DRAM makers take a more principled approach [4], such as limiting the size of compute hardware, providing a full SW stack, and keeping memory arrays and the DRAM interface intact, which has led to successful demonstrations of AI-PIM in the industry setting [4], [5].

On the other hand, current AI-PIM architectures [6], [7] can only support very simple functions such as matrix-vector multiplication (MVM). In this paper we extend the scope of DRAM-based PIM by optimizing an industry-designed PIM architecture [7] for a nontrivial non-AI application. In particular, we target Fully Homomorphic Encryption (FHE) [8], where the most important function is NTT (Number-Theoretic Transform). In addition to being memory-bound, NTT has highly irregular memory access patterns, which is a main difference compared to MVM or AI applications.

In this paper we propose a novel PIM architecture and a mapping method for NTT on PIM. One of the key challenges in finding efficient mapping is asymmetric memory access time depending on whether consecutive accesses to a bank are to the same row (*buffer hit*) vs. different rows (*buffer conflict*). While previous works on accelerating FHE workloads [9], [10] use large on-chip memory to increase data reuse and hide memory latency, PIM must economize on chip area, which is another key challenge. Exploiting the recursive structure of the NTT computation, our mapping algorithm divides the problem into smaller ones, and uses a different mapping strategy depending on the input size, categorized into three regimes based on architectural parameters. One important architectural parameter is the size and number of local buffers. We show that whereas a single-buffer architecture is extremely inefficient for NTT, providing at least one auxiliary buffer would greatly enhance mapping efficiency through a technique called *in-place update*. We also propose and evaluate the use of more buffers through *pipelining*, which increases hardware overhead minimally but can improve performance significantly by both hiding memory latency and reducing the number of row activations. Our architecture and mapping scheme also allows for bank-level parallelism (i.e., running different NTT functions in each bank), which can be naturally exploited by FHE applications with linear speedup.

Our experimental results based on architectural parameters of HBM2E demonstrate that our PIM architecture has very little hardware overhead, less than half of Newton's [7], which is already at a tiny level, yet can deliver state-of-the-art performance for NTT acceleration. Compared with the previous best PIM-based NTT accelerators [11], [12], our architecture does not modify cell arrays, and our solution can support arbitrary polynomial length and modulo values, and yet ours can deliver up to $1.7 \sim 17\times$ speedup at the NTT level (except the bit reversal, which is common in all the compared works).

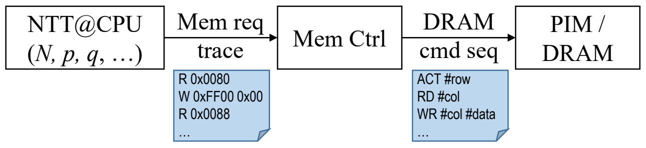


Fig. 1: From software to DRAM commands.

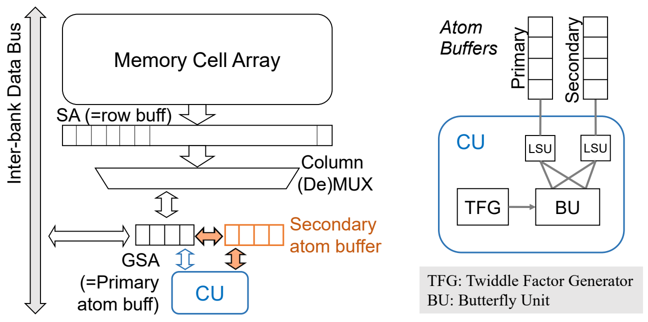


Fig. 2: Left: Bank datapath, where colored parts are our extension. Right: Compute Unit (CU).

## II. Background and Related Work

### A. DRAM and PIM

Software running on a CPU issues memory requests, which are handled by the memory controller (MC) (see Fig. 1). Understanding DRAM timing, MC schedules memory requests and generates low-level DRAM commands such as row activate and column read/write.

This work was supported by the Samsung Advanced Institute of Technology, Samsung Electronics Co., Ltd., by IITP grant (No. 2020-0-01336, Artificial Intelligence Graduate School Program (UNIST)), and by the Free Innovative Research Fund of UNIST (1.170067.01).

§J. Lee is the corresponding author (E-mail: jlee@unist.ac.kr).

A DRAM chip consists of multiple banks, which share command, address, and data buses. At the chip and bank level, the unit of a memory transaction is a DRAM *atom* (32B in HBM), which ultimately comes from a row of DRAM cells of a bank [13]. When a row is activated, the contents of the cells belonging to the row are copied to the bitline sense amplifiers (BLSAs), also known as *row buffer* (e.g., 1 KB in size). A subsequent read command latches part of the row buffer (a DRAM atom) to the global sense amplifiers (GSAs), sending it out via chip I/O (see Fig. 2).

Recent PIM works add extra logic inside DRAM banks [5] or modify the dram die structure [4]. DRAM based PIM has been able to successfully integrate computing logic inside existing DRAM technology, significantly reducing the memory bottleneck of machine learning applications. While the processing unit of FIMDRAM [5] is located in an independent memory cell and uses an additional bus to send data among banks, Newton [4] has a MVM logic inside the memory bank, completely eliminating the data transfer between the logic and the memory. The limitation of Newton [4] is that the datapath is rather simple and fixed, and it is hard to store intermediate data. This constraint makes it hard to map algorithms with intermediate data and complex memory access such as NTT.

MeNTT [11] is a 6T-SRAM based PIM accelerator for NTT. It computes NTT in a bit-serial fashion, which is not very efficient for high-precision arithmetic such as 64-bit or 32-bit. Also it is inflexible in terms of modulo and maximum polynomial length. CryptoPIM [12] is a ReRAM-based PIM accelerator targeting polynomial multiplication. While it is expected to have low cost due to the advantage of ReRAM technology, ReRAM technology has issues with fabrication such as variation and faults. Also CryptoPIM is inflexible in terms of modulo and maximum polynomial length.

### *B. FHE and NTT*

Latest FHE schemes [8], [14] are based on Ring Learning-With-Errors (Ring-LWE) [15], which encodes vectors as polynomials with coefficients of finite fields. A typical polynomial can be defined as $R_q = \mathbb{Z}_q[X]/(X^N + 1)$, a polynomial whose length is a power of 2 and each of its coefficients is a modulo of a prime number $q$. The product of two polynomials can be computed efficiently in the NTT domain [16]. NTT is a generalization of the discrete Fourier transform (DFT) on the finite field, where complex multiplication is replaced with an integer multiplication followed by a modulo operation. For a given polynomial $a = a_0 + a_1x + a_2x^2 + ... + a_{N-1}x^{N-1}$, we can associate a vector $\mathbf{a} = (a_0, a_1, ...a_{N-1})$. Then the polynomial multiplication $a * b$ can be computed using NTT, which is more efficient than direct multiplication.

$$\mathbf{a} * \mathbf{b} = NTT^{-1}(NTT(\mathbf{a}) \odot NTT(\mathbf{b})) \quad (1)$$

where $\odot$ represents the element-wise product and $NTT^{-1}$ is inverse NTT. The inverse NTT computation is mathematically identical to the original NTT except that the twiddle factor $\omega$ is replaced with its inverse.

NTT shares the characteristic irregular memory access pattern with FFT (Fast Fourier Transform). A size-$N$ NTT can be computed in $\log N$ stages, each of which involves $N/2$ parallel butterfly unit (BU) operations, followed by bit reversal (see Fig. 3). There is a large body of work on efficient parallel FFT algorithms [17], [18], e.g., Pease [17], which is well suited for FPGAs and ASICs due to its regular structure, and Stockham [18], which is self-sorting. However, those algorithms often require multiple ($\log N$) shuffling stages, which means more frequent interactions with CPU (unless hardware for shuffling is added, increasing cost) and may not be any better than the

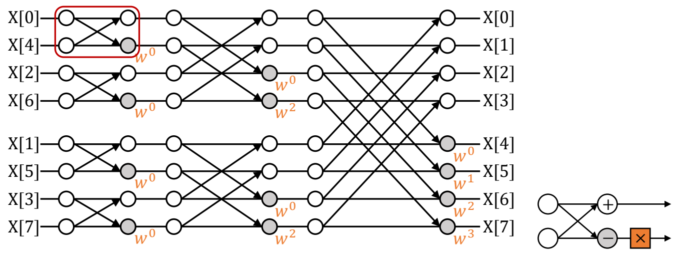
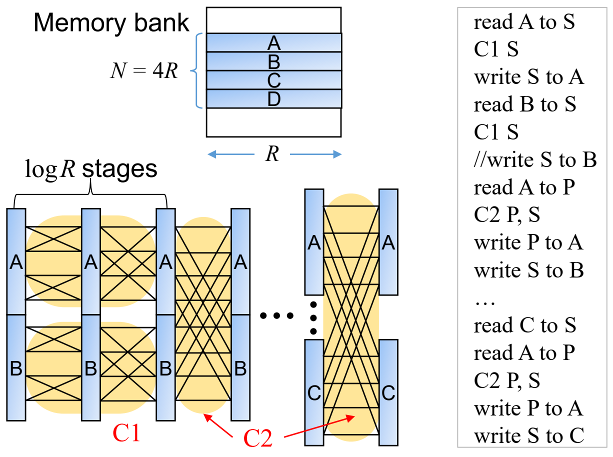


Fig. 3: Left: Dataflow of NTT algorithm (based on Cooley-Tukey FFT, bit-reversal is omitted). Right: A butterfly unit (BU) operation consisting of two ModAdd/Sub and one ModMult operations.



```
read A to S
C1 S
write S to A
read B to S
C1 S
//write S to B
read A to P
C2 P, S
write P to A
write S to B
...
read C to S
read A to P
C2 P, S
write P to A
write S to C
```
Fig. 4: Mapping example ($P$ and $S$ represent local buffers).

Cooley-Tukey FFT algorithm [19]. Also intra-row data reuse can be easily and fully exploited by recursive Cooley-Tukey FFT, which we use in this work. We assume that bit reversal is performed by software running on a CPU, which is a common assumption in previous PIM approaches [11], [12]. Further, bit reversal can be avoided altogether when all NTT-domain operations are element-wise operations [11].

## III. Our Architecture and Mapping

### *A. Analysis of NTT Computation*

For size-$N$ NTT, the number of compute operations is $O(N \log N)$, and the amount of data transfer is either $O(N)$ if the entire input fits in the local memory, whose size is denoted by $M$, or $O(N)$ *per stage* otherwise. In recursive Cooley-Tukey FFT, the first $\log M$ stages (assuming $M$ is a power of 2) can be partitioned into $\frac{N}{M}$ independent blocks, each of which would fit in the local memory (see Fig. 4). Therefore, the total amount of data transfer when $N \geq M$ is $O(N + N(\log N - \log M))$, and the compute-to-data-transfer ratio (CDR) is $O(\frac{\log N}{1+\log \frac{N}{M}}) \leq O(\log N)$, where equality holds when $M = N$. The CDR cannot be increased further by such techniques as batching. Tiling (also known as blocking) is also difficult due to the irregular memory access pattern. All of the above suggests that NTT is memory-bound and PIM can be a good candidate to accelerate NTT. However, DRAM memories have very asymmetric access time. Also the very limited size of a row compared to $N$ makes the problem of mapping NTT to PIM challenging.

### *B. Necessity of An Auxiliary Buffer*

Suppose $N$ is small enough to fit in a row buffer. Then the NTT function can be implemented efficiently by a small compute unit (CU) within a bank that can read two words of a row (performed by two **load** $\mu$-ops), do a butterfly unit (BU) operation, and write the result back to the input locations (by two **store** $\mu$-ops). Every access to memory (i.e., load/store) except the first will be a buffer hit, which is optimal.

However, if $N$ is larger than the row buffer size (denoted by $R$), there is no efficient mapping that realizes maximal data reuse, unless there is an extra buffer. With a single buffer (i.e., GSA only), assuming CU is equipped with two scalar registers for two BU inputs (see Fig. 2), *every BU operation* requires two loads and two stores, and about half of them require row activation, which greatly increases memory access time.

### C. Sufficiency of One Auxiliary Buffer

Suppose that $N = 4R$. As illustrated in Fig. 4, the first $\log R$ stages can be vertically partitioned into 4 independent blocks (only two are shown), each of which can be processed efficiently with one row activation, resulting in just 4 ($= \frac{N}{R}$) row activations for the $\log R$ stages.

The question is whether the other stages can be done efficiently. Let us assume that there is an auxiliary buffer of size $R$. We call GSAs and the extra buffer *primary* ($P$) and *secondary atom buffer* ($S$), respectively. For the $(1+\log R)$-th stage, a naïve implementation may require a third buffer for the output, since the two buffers $P$ and $S$ are occupied by the two inputs. But observing that each input element is used exactly twice, we can schedule operations such that we complete each BU operation before moving on to the next, which allows us to store the output of a BU operation directly to its input locations (*in-place update*). The combination of BU-grained scheduling and in-place update not only eliminates the need for a separate output buffer, but also allows us to fully reuse input data in $P$ and $S$, which is true for the following stages as well. Each *stage* (not a BU op) after the first $\log R$ stages requires 4 ($= \frac{N}{R}$) pairs of reads and writes, but half of the writes can be made a buffer hit, resulting in at most $3\frac{N}{2R}$ row activations, which is much less than if no in-place update and no third buffer are used.

### D. New DRAM Commands and Our PIM Architecture

One problem with adding an extra row buffer is a very large area overhead. However, the extra buffer needs only to be of a DRAM atom size, $N_a$. In fact, GSAs are also of a DRAM atom size [13], [20].

To realize the mapping outlined above, CU needs to support the following operations.

- **C1** takes input from one buffer, $S$, and performs the NTT function minus bit-reversal, which includes $\log N_a$ stages of $\frac{N_a}{2}$ BU operations per stage. The result is stored into the same buffer as the input.
- **C2** takes input from two buffers, $P$ and $S$, and performs one $N_a$-way vectorized BU operation. The result is stored back into $P$ and $S$. Note that C1 and C2 must perform modulo add/mult [6] in order to support NTT.

The memory hierarchy of our PIM architecture is as follows. The CU has two operand registers (as well as other registers, e.g., for twiddle factors), which can be seen as L0 memory. The $P$ and $S$ buffers are L1 memory, and data movement between L0 and L1 is accomplished by load/store $\mu$-ops, which are very fast (2 cycles) and part of C1 and C2. The memory banks are L2 memory, and data transfer between L1 and L2 is managed by explicit commands, which we simply call **read** and **write**. These CU-read/write are similar to column read/write, taking about a dozen cycles, except that data transfer stops at $P$ or $S$ instead of chip I/O.

**Algorithm 1** Intra-atom NTT compute: $\mathrm{C1}(S) \rightarrow S$

**Inout** $S$ : vector reg for both input and output
**Param** $\omega_0, r_\omega$ : init value & step size for twiddle factor gen.

```
1:  $\omega_s$ : twiddle factor for each stage, initialized to 1
2:  for ($stage = 1; stage \leq \log N_a; stage$ +=1) do
3:      let $\omega \leftarrow \omega_0$, $m \leftarrow 2^{stage-1}$
4:      for ($k = 0; k < N_a; k$ +=$2m$) do
5:          for ($j = 0; j < m; j$ +=1) do
6:              let $a \leftarrow S[k+j]$
7:              let $b \leftarrow S[k+j+m]$
8:              $S[k+j] \leftarrow (a+b) \mod q$
9:              $S[k+j+m] \leftarrow (a-b) \cdot \omega \mod q$
10:             $\omega \leftarrow \omega \cdot \omega_s \mod q$
11:         end for
12:     end for
13:     $\omega_s \leftarrow \omega_s \cdot r_\omega \mod q$
14: end for
```

**Algorithm 2** Inter-atom NTT compute: $\mathrm{C2}(P, S) \rightarrow P, S$

**Inout** $P, S$ : vector regs for both input and output
**Param** $\omega_0, r_\omega$ : init value & step size for twiddle factor gen.

```
1: let $\omega \leftarrow \omega_0$
2: for ($j = 0; j < N_a; j$ +=1) do
3:     let $a \leftarrow P[j]$, $b \leftarrow S[j]$
4:     $P[j] \leftarrow (a+b) \mod q$
5:     $S[j] \leftarrow (a-b) \cdot \omega \mod q$
6:     $\omega \leftarrow \omega \cdot r_\omega \mod q$
7: end for
```

## IV. Implementation Detail

### A. Host Interface and Architecture

From the software perspective, our NTT function can be invoked as a write request (see Fig. 1), which contains NTT parameters as "write data". We assume $N$ to be a power of 2. The input data ($N$ 32b integers) is assumed to be already in the memory; thus, only the address is passed. The MC generates a sequence of DRAM commands to fulfill the NTT function (see Fig. 4). The result is stored at the same location as the input, and a write response is given to the request initiator to signal that NTT is completed. The MC needs to be modified to implement the mapping algorithm presented in the next section.

Algorithms 1 and 2 describe the C1 and C2 commands. The BU in Fig. 2 has two operand registers. Each buffer is single-ported, but a small crossbar switch allows full connectivity between buffers and BU registers. To do computation, CU first loads input from $P$ and/or $S$ to its registers, does a BU operation, and stores the results, which is repeated in a pipelined fashion for all elements available. In our architecture the size of a DRAM atom ($N_a$) is 8.

We generate twiddle factors on-the-fly using a method similar to [21], which allows us to use all memory bandwidth for accessing input polynomial.

To pass parameters ($q$, $\omega_0$, $r_\omega$), a value (16-bit) is placed on the global buffer, which is visible to all banks, and subsequently loaded to a scalar register in the CU of a bank (in multiple cycles for higher precision values).

The secondary atom buffer can be implemented using SRAM cells (6T/cell) plus inverters (2T/cell) to provide complementary signals needed.

### B. NTT Computation Mapping

Given the memory hierarchy of our PIM architecture, there are three mapping regimes as illustrated in Fig. 5.

1) Intra-atom (when $N \leq N_a$): applies to the first $\log N_a$ stages, and uses C1 command.
2) Intra-row (when $N_a < N \leq R$, where $R$ is the row buffer size): applies to the next $\log \frac{R}{N_a}$ stages.
3) Inter-row (when $N > R$): applies to the rest of the stages. Both intra-row and inter-row use C2 command.

Both intra-row and inter-row are "inter-atom", but their difference is that only inter-row requires intermittent row activate commands.

The MC generates DRAM commands by dividing a given NTT function or its dataflow graph (DFG) such as Fig. 3 as follows . First, note that there are two ways to divide a DFG: stage-wise (horizontally) or data-wise (vertically). Second, observe that all data dependence is within a row during the first $\log R$ stages, whereas in the latter stages all data dependence is across rows but BU operations are vectorized with at least $R$ ways. (The analogous applies to an atom buffer as well.) Therefore, we propose to divide the first $\log R$ stages of a DFG vertically into $\frac{N}{R}$ independent row-sized blocks, each of which is divided again horizontally into (i) the first $\log N_a$ stages, which are handled using intra-atom mapping, and (ii) the rest of the stages, which are handled using intra-row mapping. The latter stages are in the inter-row regime, for which we process a DFG stage-by-stage and each stage in the sequential order. Since each C2 command in this regime always involves at least two row activations, the processing order does not make much difference in performance, but increasing the size or number of buffers does, which we discuss next.

## V. Pipelining Optimization

While a dual-buffer architecture can fully realize data reuse up to intra-row mapping, it suffers from frequent row activations in inter-row mapping. Also even if full data reuse is achieved, it does not necessarily mean the highest throughput possible. Fig. 6 illustrates the difference between executing two CU operations consecutively (without pipelining) vs. with pipelining. In the case of pipelined execution, read commands for the second compute operation can start before write commands for the first compute finish, thereby hiding some memory access latency.

On the other hand, pipelining requires more buffers. In the case of intra-atom mapping, pipelining is possible even with a single auxiliary buffer (since our mapping uses only one buffer, we can use the other buffer for pipelining), but more buffers mean more overlap. In general, to overlap $n$ executions requires $n$ times as many buffers. For the other mapping regimes, more than one auxiliary buffers are necessary for pipelining. Also C1 and C2 commands need to take operands from any buffers, including one primary and multiple secondary buffers. Thus in addition to increased buffer area, hardware overhead of pipelining includes a larger crossbar switch in CU (see Fig. 2). CU-read/write commands and C1/C2 commands need more parameters too, but having no immediate field, they have enough bits to encode additional parameters.

Pipelining has an unexpected positive effect in inter-row mapping: reduced number of row activations. Equipped with more buffers thanks to pipelining, we can group same-row read/write commands together as illustrated in Fig. 6c, eliminating some row activations in the process. Note that the likelihood of finding same-row commands within the two CU compute operations is quite high, since the inter-row regime has highly vectorized BU operations.

The MC is responsible for generating pipelined schedules. One issue is that since the command bus is shared between all DRAM commands, we need to choose which command to issue first between read/write vs. compute. It can be decided rather easily, since the latency of each command is known in advance.

TABLE I: Architecture Parameters

| Architecture Parameters | | Timing Param. (cycles) | |
|---|---|---|---|
| DRAM atom size | 32B | CL | 14 |
| # of columns per row | 32 | tCCD | 2 |
| # of rows per bank | 32,768 | tRP | 14 |
| # of ranks | 1 | tRAS | 34 |
| # of banks | 1 | tRCD | 14 |
| | | tWR | 16 |

TABLE II: PIM Area Overhead ($N_b$: # of all atom buffers)

| Architecture | $N_b$ | Area (mm$^2$) | % |
|---|---|---|---|
| A DRAM bank | | 4.2208 | – |
| Newton [7] | | 0.0474 | 1.123 |
| NTT-PIM | 1 | 0.0213 | 0.504 |
| | 2 | 0.0232 | 0.550 |
| | 4 | 0.0263 | 0.624 |
| | 6 | 0.0285 | 0.676 |

## VI. Experiments

### A. Experimental Setup

To evaluate the performance of our NTT-PIM, we have developed our in-house PIM simulator, which consists of a front-end driver and DRAMsim3 [22] working in tandem. The front-end driver serves two purposes: to generate DRAM command sequence simulating the memory controller including our mapping algorithm, and to verify the functionality of our NTT function as executed in DRAMsim3.[1]

Table I lists the parameters used in our evaluation, which are based on HBM2E. While all our results are based on running NTT on a single bank, FHE applications can naturally run multiple NTT functions using multiple banks.

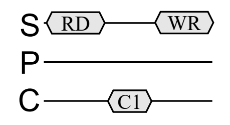
### B. Synthesis Results

Since PIM is supposed to be fabricated using a memory technology, which is not open, it is hard to estimate the area overhead of our architecture accurately. Instead, we provide a comparison with Newton [7] in terms of hardware overhead, which should assure that our CU is small enough to be fabricated inside a memory bank. We have implemented both Newton's compute hardware, mostly consisting of 16 16-bit floating-point MACs, and our CU in Fig. 2 in Verilog RTL, and synthesized them with Synopsys Design Compiler using Samsung 65 nm standard-cell library. The area of auxiliary atom buffers is estimated with CACTI 7.0 at 65 nm. We have verified that our designs meet the timing requirement for 1200 MHz, which is the operating frequency of HBM2E.

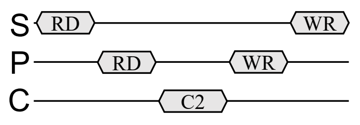
We have fully pipelined the BU inside CU, which supports ModAdd/Sub and ModMult for arbitrary modulo values using Montgomery reduction algorithm [23]. The latency of C1 and C2 is 15 and 10 cycles, respectively.

Table II summarizes area results, which show that our hardware overhead is small, being less than half of Newton's, which is already

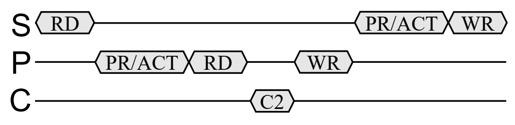
[1]The front-end driver is written in Python, and runs iteratively with DRAMsim3 to implement two-way communication between the two, which was added to double-check the correctness of timing and functionality.



(a) Intra-atom mapping

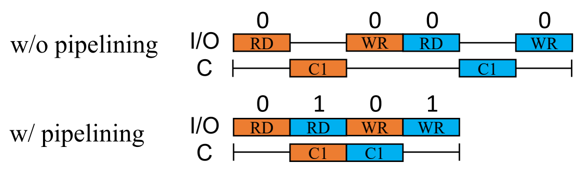


(b) Intra-row mapping

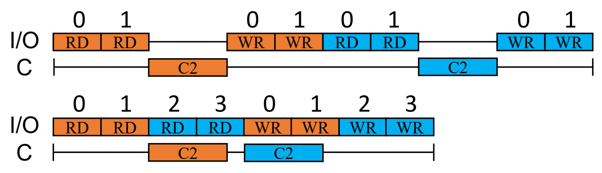


(c) Inter-row mapping

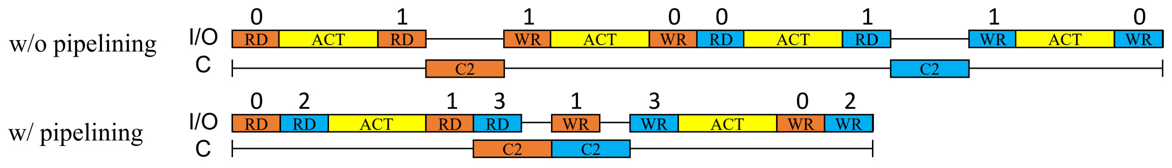
Fig. 5: Timing diagram for three mapping regimes (S/P: secondary/primary atom buffer, C: CU). Initial row activate is omitted.



(a) Intra-atom mapping ($N_b$ = 1 vs. 2)



(b) Intra-row mapping ($N_b$ = 2 vs. 4)



(c) Inter-row mapping ($N_b$ = 2 vs. 4)

Fig. 6: When two CU operations are executed without vs. with pipelining. Same color means dependence, numbers buffer indices, and $N_b$ is the number of buffers (incl. primary). Performance improvement is due to (i) overlapping memory latency with compute latency, and in the case of inter-row mapping, also to (ii) reducing the number of row activations.

at a tiny level. Also the additional overhead of having multiple atom buffers seems marginal. To give a rough comparison, we have also estimated the area of a single DRAM bank using CACTI-3DD [24] DDR4 model at 32 nm, which is the most advanced node supported by the tool.[2]

### C. Performance and Effect of Using Multiple Buffers

Fig. 7 compares performance of our PIM architecture under various values of $N$ (polynomial length) and $N_b$. First we note that without auxiliary buffers, there is no performance advantage even compared with a software execution, whereas even just one auxiliary buffer can improve performance by an order of magnitude.

Moreover, adding more buffers gives very significant speed up of about $1.5 \sim 2.5\times$ depending on $N$. As expected, having multiple auxiliary buffers proves more effective when $N$ is larger, which is because at larger $N$, a bigger portion of runtime is accounted for by inter-row mapping and inter-row mapping benefits more from pipelining.

### D. Sensitivity to Clock Frequency

To see the effect of lower clock frequency we have varied the frequency from 300 MHz to 1200 MHz. The computation time of the CU increases in proportion to the inverse of clock frequency, but the absolute latency of DRAM memory access time (in ns) is kept constant. The result is summarized in Fig. 8. Since most of the latency of NTT-PIM is due to DRAM memory operations, the performance of NTT-PIM is quite tolerable under lower frequencies, still achieving $3 \sim 7\times$ speedup compared to CPU. The performance of NTT with long polynomial lengths tend to be more robust on lower frequencies, slowing down $1.65\times$ only when the clock frequency drops by $4\times$.

[2]We intentionally use an older technology for logic, to simulate the effect of implementing logic with a memory technology, which would be slower and take larger area.

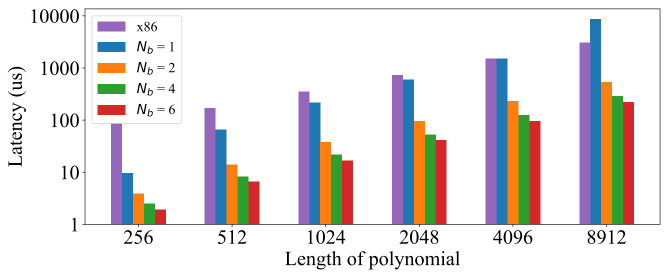


Fig. 7: Sensitivity to $N_b$, the number of buffers.

### E. Comparison with Previous Work

Table III compares our NTT-PIM with previous PIM-based NTT accelerators as well as x86 CPU and FPGA, in terms of latency and power. Our NTT-PIM achieves speedup of minimum $1.7\times$ up to $17\times$ depending on the polynomial size. It is important to note that ours is much more flexible than some of the compared works. For instance, CryptoPIM [12] has a limitation that the modulo is fixed (a severe drawback for FHE, which runs multiple NTTs using different modulo values) and both CryptoPIM and MeNTT [11] limit the maximum polynomial size. Ours has no such restriction. Moreover, the power consumption of MeNTT [11] is very low, which is largely due to the fact that its maximum polynomial size is very small (1K). While our NTT-PIM's latency increases exponentially as the polynomial length increases, it is expected of any scheme supporting arbitrary polynomial length. After all, the number of operations increases as $O(N \log N)$. But it has also to do with the fact that longer polynomials require frequent row activations due to larger portion of inter-row mapping.

TABLE III: Comparison with Previous Work

| Design | | NTT-PIM | | | MeNTT | CryptoPIM | x86 CPU | FPGA |
|---|---|---|---|---|---|---|---|---|
| Method | | DRAM | | | 6T-SRAM | RRAM | Software | - |
| Bitwidth | | 32 | | | 14 / 16 | 16 / 32 | 32 | 16 |
| # of atom buffers ($N_b$) | | 2 | 4 | 6 | | | | |
| Latency (ns) | $N$: 256 | 3.90 | 2.50 | 1.94 | 23$^{\ddagger}$ | 68.57$^{\dagger}$ | 84.81 | 21.56$^{\dagger}$ |
| | 512 | 14.16 | 8.33 | 6.58 | 26$^{\ddagger}$ | 75.90$^{\dagger}$ | 168.96 | 47.64$^{\dagger}$ |
| | 1024 | 38.19 | 21.62 | 16.89 | 34.3$^{\dagger}$ | 83.12$^{\dagger}$ | 349.41 | 101.84$^{\dagger}$ |
| | 2048 | 95.84 | 53.03 | 41.18 | - | 363.90 | 736.92 | - |
| | 4096 | 230.45 | 124.95 | 96.62 | - | 392.69 | 1503.31 | - |
| Energy (nJ) | $N$: 256 | 0.80 | 0.49 | - | 0.144$^{\ddagger *}$ | 68.67$^{\dagger}$ | 570.60 | 2.15$^{\dagger}$ |
| | 512 | 4.77 | 2.67 | - | 0.324$^{\ddagger *}$ | 75.90$^{\dagger}$ | 1179.52 | 5.28$^{\dagger}$ |
| | 1024 | 13.86 | 7.16 | - | 0.868$^{\dagger *}$ | 83.12$^{\dagger}$ | 2483.77 | 12.52$^{\dagger}$ |
| | 2048 | 36.68 | 18.98 | - | - | 363.60 | 5273.07 | - |
| | 4096 | 93.08 | 48.93 | - | - | 421.78 | 10864.64 | - |

*Note.* 1. † indicates 16-bit bitwidth and ‡ 14-bit.
2. *very small memory, supporting only $N \leq 1024$.

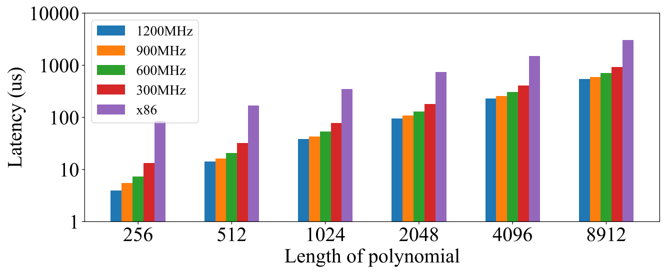


Fig. 8: Sensitivity to clock frequency ($N_b$ = 2).

## VII. Conclusion

We presented NTT-PIM, a novel PIM architecture supporting more flexible data movement within and across bank rows, and a mapping method for NTT functions. To address the challenges of highly irregular memory access patterns and very restricted area budget, our solution exploits the characteristics of the algorithm, such as BU op-level scheduling, in-place update, and pipelining using multiple buffers. Our experimental results demonstrates that even *without modifying cell arrays*, PIM can provide state-of-the-art performance at very little area and power overhead for important functions such as NTT.

Our architecture and mapping is designed to support bank-level parallelism, and while we expect near-linear speed up as the number of banks increases, a more thorough investigation at the system level is left for future work.

## References

[1] G. Hilson, "AI drives renewed interest in PIM," *EE Times*, 2021. [Online]. Available: https://www.eetimes.com/ai-drives-renewed-interest-in-pim

[2] N. S. Kim, "A journey to a commercial-grade processing-in-memory (PIM) chip development," in *HPCA-27*, 2021, keynote speech.

[3] L. Ke *et al.*, "Near-memory processing in action: Accelerating personalized recommendation with AxDIMM," *IEEE Micro*, vol. 42, no. 1, pp. 116–127, 2022.

[4] S. Lee *et al.*, "A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based accelerator-in-memory supporting 1TFLOPS MAC operation and various activation functions for deep-learning applications," in *ISSCC*, vol. 65, 2022, pp. 1–3.

[5] Y.-C. Kwon *et al.*, "25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications," in *ISSCC*, vol. 64, 2021, pp. 350–352.

[6] A. C. Mert *et al.*, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Transactions on Computers*, vol. 71, no. 11, pp. 2829–2843, 2022.

[7] M. He *et al.*, "Newton: A dram-maker's accelerator-in-memory (AiM) architecture for machine learning," in *MICRO-53*, 2020, pp. 372–385.

[8] J. H. Cheon *et al.*, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi *et al.*, Eds. Cham: Springer International Publishing, 2017, pp. 409–437.

[9] M. S. Riazi *et al.*, "HEAX: An architecture for computing on encrypted data," in *ASPLOS*, 2020, p. 1295–1309.

[10] N. Samardzic *et al.*, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54*, 2021, p. 238–252.

[11] D. Li *et al.*, "MeNTT: A compact and efficient processing-in-memory number theoretic transform (NTT) accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 5, pp. 579–588, 2022.

[12] H. Nejatollahi *et al.*, "CryptoPIM: In-memory acceleration for lattice-based cryptographic hardware," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[13] N. Chatterjee *et al.*, "Architecting an energy-efficient dram system for gpus," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 73–84.

[14] J. Fan *et al.*, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, p. 144, 2012.

[15] V. Lyubashevsky *et al.*, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23.

[16] T. Pöppelmann *et al.*, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *Progress in Cryptology – LATINCRYPT 2012*, A. Hevia *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 139–158.

[17] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *J. ACM*, vol. 15, no. 2, p. 252–264, apr 1968.

[18] P. N. Swarztrauber, "FFT algorithms for vector computers," *Parallel Comput.*, vol. 1, no. 1, p. 45–63, aug 1984.

[19] D. Harris *et al.*, "Vector radix fast Fourier transform," in *ICASSP '77. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, 1977, pp. 548–551.

[20] V. Seshadri *et al.*, "The processing using memory paradigm:in-dram bulk copy, initialization, bitwise AND and OR," 2016. [Online]. Available: https://arxiv.org/abs/1610.09603

[21] A. Aysu *et al.*, "Low-cost and area-efficient fpga implementations of lattice-based cryptography," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 81–86.

[22] S. Li *et al.*, "DRAMsim3: A cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[23] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[24] K. Chen *et al.*, "CACTI-3DD: Architecture-level modeling for 3d die-stacked DRAM main memory," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 33–38.