Software-Pipelining The Number Theoretic Transform on Neural Processing Units: A Loop-Fission Methodology

Henri Vermeersch[1,2], Thomas de Ruijter[1],
Quinten Norga[1], Xander Pottier[1] and Ingrid Verbauwhede[1,2]

[1] COSIC KU Leuven, Leuven, Belgium, {firstname.lastname}@kuleuven.be
[2] Belfort Labs, Leuven, Belgium, {firstname.lastname}@belfortlabs.com

Abstract. Lattice-based post-quantum cryptography (PQC) and fully homomorphic encryption (FHE) rely heavily on the number theoretic transform (NTT) for efficient polynomial multiplication. While GPUs and FPGAs have been widely explored for NTT acceleration, neural processing units (NPUs), a rapidly growing class of AI accelerators, remain largely unexplored for cryptographic workloads. In this work, we develop a methodology for accelerating the NTT on NPUs, using AMD's AI Engine (AIE) architecture as a representative NPU to develop and benchmark it. At its core is *manual loop fission*, in which large and complex loops are split into smaller primitives, enabling highly efficient software pipelining and avoiding memory stalls or register spilling. Applying it, we develop efficient Barrett reduction primitives and use these to construct three high-throughput, low-latency NTT implementations (radix-2, radix-4, and radix-8), achieving significant performance gains over the AIE compiler and prior literature. On AMD's first-generation AIE specifically, we achieve speedups of 8.27× to 13.2× in throughput and 2.44× to 10.3× in latency per AIE tile in comparison with prior work. Our implementations support arbitrary NTT-friendly moduli and polynomial sizes up to degree 4096 for 14-bit moduli and 2048 for 30-bit moduli. Finally, we discuss the portability of our techniques to other NPU architectures, and identify which architectural features are most beneficial for NTT acceleration.

Keywords: Number Theoretic Transform · Neural Processing Unit · AI Engine · Barrett reduction · Post-Quantum Cryptography · Fully Homomorphic Encryption

1 Introduction

A wide range of lattice-based cryptographic schemes rely on polynomial arithmetic over the rings $\mathbb{Z}_q[x]/(x^n \pm 1)$. Examples include the recently standardized ML-KEM (CRYSTALS-Kyber [Kyb24]) and ML-DSA (CRYSTALS-Dilithium [Dil24]) for post-quantum cryptography (PQC), as well as fully homomorphic encryption (FHE) [CGGI18, CKKS16] and lattice-based zero-knowledge proofs (ZKP) [LNP22]. The computations in lattice-based schemes mostly involve modular reductions, element-wise operations, matrix multiplications, and number theoretic transforms (NTT). Among these, NTTs and their modular reductions generally form the dominant bottleneck and are critical for performance. Moreover, although element-wise operations and matrix multiplications suit platforms such as GPUs or NPUs well, NTTs are hard to parallelize due to their sequential nature and intricate indexing.

GPUs [JZG+25, OJS25, FZZ+25] and FPGAs [LBRS25, DBBS25, ZKA+25, KHF25] have received significant attention for NTT acceleration. A less explored, yet increasingly

relevant, alternative is *neural processing units* (NPUs) [RL24]. These AI-specialized accelerators combine SIMD data paths, explicit scratchpad memories, and deterministic execution at low power. Compared with GPUs, they offer a superior compute-to-power ratio and reduced latency, attractive for compute-intensive workloads such as FHE and lattice-based ZKPs, where power is an increasingly important deployment constraint. NPUs are deployed at scale in datacenters and edge devices. Examples include Google's TPUs [JYP+17], NVIDIA/Groq LPUs [ARS+20], AMD's AI Engines (AIE) in Versal [Xil18] and XDNA/XDNA2 chips [SKP24], and others [BDL25, SIF+25].

In this work, we develop a methodology for accelerating the NTT on NPUs, and we use AMD's first-generation AIE as a representative datacenter NPU on which to develop and evaluate it. The AIE array in the Versal Adaptive SoC is a tiled array of Very Long Instruction Word (VLIW) SIMD processors with high-bandwidth, bank-partitioned local memories and deterministic, cycle-accurate execution. All of these features are shared by many NPU designs. The AIE Array ships in datacenter-class Versal cards [Xil18], and newer AIE-derived NPUs are integrated into CPU-class chips [SKP24], making them immediately deployable.

Related Work. To our knowledge, only three prior works have studied NTT acceleration on AIE [BZLZ25, AA24, NKNT24], of which two report clear throughput results [BZLZ25, AA24]. However, through suboptimal communication paths and algorithmic choices, these works fail to fully exploit the NPU architecture and features to achieve maximal performance gain, limiting their conclusions about the platform's potential.

Our Contributions. In this work, we first evaluate whether NPUs can serve as competitive platforms for accelerating lattice-based cryptography. Second, we identify which NPU architectural features are most beneficial, thereby providing feedback to architecture designers. Our approach is a general methodology, which we develop and evaluate on the first-generation AIE architecture as a representative case study. More specifically, with the source code available[1]:

- We describe and motivate a promising implementation workflow to implement highly efficient kernels on this architecture. It is largely based on *manual loop fission* [AK01], the splitting of complex loop bodies into simpler loops which can be pipelined more efficiently. Our workflow combines this with manual design of loops that can be fully software pipelined by the compiler, manual loop unrolling, careful memory bank placement, and integration of permutations in the compute data path. This workflow is not specific to the NTT and applies to other compute-intensive algorithms on similar architectures (Section 3).
- We develop high-throughput Barrett reduction primitives that utilize approximately 75% of available multipliers per cycle for `int16` and nearly 100% for `int32`. Our parameter and rounding function choices ensure all computations use single-word arithmetic and maximize the headroom for lazy reduction (Section 4).
- We develop three NTT implementations (radix-2, radix-4, and radix-8) that improve over the state of the art by factors of 8.27× to 13.2× in throughput and 2.44× to 10.3× in latency per tile (Section 5). This is achieved through constant-geometry transforms, in-data path permutations, and partial and lazy modular reduction. We provide results for polynomial sizes relevant to both PQC ($n \approx 256$) and some FHE schemes such as TFHE ($n \approx 2048$) [CGGI18]. We observe that the use of constant-geometry transforms on platforms supporting the corresponding permutations is a key factor in achieving high performance (Section 5).

[1] https://github.com/KULeuven-COSIC/NTTs_on_AMD_AI_Engines_AIE1

- We present an efficient bit-reversal unit using vector shuffles that enables inverse NTT computation via the forward implementations (Section 6).
- We show that the methodology is not tied to the first-generation AIE: we discuss the portability of our techniques to other NPU platforms, identify which architectural features benefit NTT acceleration, and briefly discuss how our algorithms port to the second-generation AIE (Section 7).

We provide implementations based on both `int16` and `int32` datatypes, where the former is more efficient but the latter supports larger parameters.

2 Preliminaries

2.1 Number Theoretic Transform (NTT)

The Number Theoretic Transform (NTT) is a finite field analogue of the Discrete Fourier Transform (DFT), operating in $\mathbb{Z}_q$ (with q a prime number) instead of $\mathbb{C}$. The definitions of the NTT and inverse NTT (iNTT) are given in Equation 1 and Equation 2, respectively, where ω_n is a primitive n-th root of unity in $\mathbb{Z}_q$.

$$\hat{a}_l = \sum_{k=0}^{n-1} a_k \omega_n^{kl} \mod q \quad (1) \qquad\qquad a_k = n^{-1} \sum_{l=0}^{n-1} \hat{a}_l \omega_n^{-kl} \mod q \quad (2)$$

Equation 1 and Equation 2 imply a complexity of $\mathcal{O}(n^2)$, but the Fast Fourier Transform (FFT) algorithm can be applied to reduce this to $\mathcal{O}(n\ \log(n))$, as in the case of the DFT. For a detailed coverage of FFT algorithms, we refer to the literature [LZ22, DCH+24, CYY+22]. Here, we provide a short summary of the Cooley-Tukey or Decimation-in-time (DIT) FFT [LZ22] for a general radix r. Let n be divisible by r and substitute $l = \frac{n}{r} l_1 + l_2$ and $k = rk_1 + k_2$ in Equation 1, where $l_1, k_2 = 0, 1, \ldots, r-1$ and $l_2, k_1 = 0, 1, \ldots, n/r-1$. A binomial product then appears in the exponent of ω_n, which expands to $nk_1l_1 + \frac{n}{r}k_2l_1 + rk_1l_2 + k_2l_2$, where the first term can be dropped since $\omega_n^{nk_1l_1} = (\omega_n^n)^{k_1l_1} = 1$. Also, note that $\omega_n^m = \omega_{n/m}$. After some rearrangements we have

$$\hat{a}_{\frac{n}{r}l_1+l_2} = \sum_{k_2=0}^{r-1} \left(\sum_{k_1=0}^{\frac{n}{r}-1} a_{rk_1+k_2} \omega_{n/r}^{k_1l_2} \right) \omega_n^{k_2l_2} \omega_r^{k_2l_1} \mod q \quad (3)$$

The inner sum may be interpreted as r NTTs of size n/r. The results of these NTTs are first multiplied element-wise with the set of twiddle factors $\omega_n^{k_2l_2}$ and then used in n/r NTTs of size r (the outer sum) which can be interpreted as r-by-r matrix-vector multiplications, which we call the radix-r butterflies. The inner sum may now be subdivided further such that the full transform of size n will eventually be divided into $\log_r(n)$ layers, each performing an element-wise multiplication with twiddle factors followed by n/r matrix multiplications of size r-by-r. This means that a larger radix incurs a higher computational cost. However, larger radix transforms may be more efficiently mapped to the hardware architecture.

2.2 Barrett reduction

Barrett reduction [Bar87][2] approximates modular reduction by replacing division by q with a precomputed multiplication and shift. We define the relevant primitives and their

[2]With precomputation of the constant twiddle factor, Montgomery reduction requires the same number of instructions and memory as Barrett reduction, and most of this work's ideas carry over to Montgomery reduction.

output ranges. Let $[\![\,]\!] : \mathbb{R} \to \mathbb{Z}$ denote an integer rounding function, such as the nearest integer ($\lfloor\,\rceil$), floor ($\lfloor\,\rfloor$), or ceiling ($\lceil\,\rceil$) functions. We follow the notation of Becker et al. [BHK+21].

Definition 1 (Barrett multiplication and reduction)**.** Let a and ω be integers with ω known beforehand (the *twiddle factor* in the context of NTT), $[\![\,]\!] : \mathbb{R} \to \mathbb{Z}$ an integer rounding function and K a parameter. The Barrett multiplication of a and ω is then

$$\mathbf{bar}_q^{[\![\,]\!]}(a,\omega) := a\omega - \left[\!\!\left[\frac{a\omega'}{R}\right]\!\!\right] q \quad \text{with} \quad R = 2^K \quad \text{and} \quad \omega' = \left[\!\!\left[\frac{\omega R}{q}\right]\!\!\right] \tag{4}$$

where ω' is called the Barrett multiplier and is computed beforehand. A Barrett reduction is defined as the specific case where $\omega = 1$. From [BHK+21], we have the following output range bound for the Barrett primitives using the nearest integer rounding function:

Theorem 1. *Let a and ω be integers. We have the following bound for Barrett multiplication and Barrett reduction using the nearest integer function:*

$$\left|\mathbf{bar}_q^{\lfloor\,\rceil}(a,\omega)\right| \leq \frac{q}{2}\left(1 + \frac{|a|}{R}\right)$$

Similarly in the case of the floor and ceiling functions, we have an output range for a positive integer a of $0 \leq \mathbf{bar}_q^{\lfloor\,\rfloor}(a,\omega) < q\left(1+\frac{a}{R}\right)$ and $0 \geq \mathbf{bar}_q^{\lceil\,\rceil}(a,\omega) > -q\left(1+\frac{a}{R}\right)$, respectively. Since the output of the nearest integer Barrett primitives may be negative, we call this *partial reduction*. A reduction to $[0, q)$ is called a *full reduction* and can be obtained from a nearest integer partial reduction using a conditional addition of q.

2.3 AI Engine architecture

The first-generation AIE array is part of AMD's Versal Adaptive SoC family [Xil18], designed for datacenter applications. The chip also integrates dual-core Arm processors, FPGA fabric, and Digital Signal Processing (DSP) engines. The AIE array is a two-dimensional matrix of tiles, where each tile consists of an identical compute element (an *AI Engine*) and a memory element, running at a fixed frequency of 1.25 GHz (see Figure 1). For a detailed description, we refer to the AMD documentation [AMDd, AMDc, AMDa, AMDb]; here we summarize the aspects most relevant to this work.

Programming model. Kernels are written in C++ using two APIs: the AI Engine API [AMDa], which is platform-independent and provides coarse-grained vector operations, and the AIE Intrinsics [AMDb], which map nearly one-to-one to machine instructions for fine-grained control. Since the AIE toolchain does not support hand-written assembly, we steer the compiler toward efficient schedules by choosing the right intrinsics and API calls, structuring loops carefully, and controlling register usage at the assembly level.

Communication between tiles. The communication options are illustrated in Figure 1. Each memory element has a capacity of 32 KB divided into four banks of 8 KB. Each bank can serve one 256-bit load or store per cycle. However, if multiple requests (from the same tile or from different tiles) target the same bank in the same cycle, a *memory stall* delays one of the requests by one cycle, stalling the corresponding AIE. Tiles communicate through: (i) 256-bit/cycle direct memory access to their own and three neighboring memory elements (blue lines), (ii) a streaming interconnect connecting all tiles at 32 bits/cycle per connection (black lines), or (iii) a 384-bit/cycle cascade path snaking through the array (green lines).

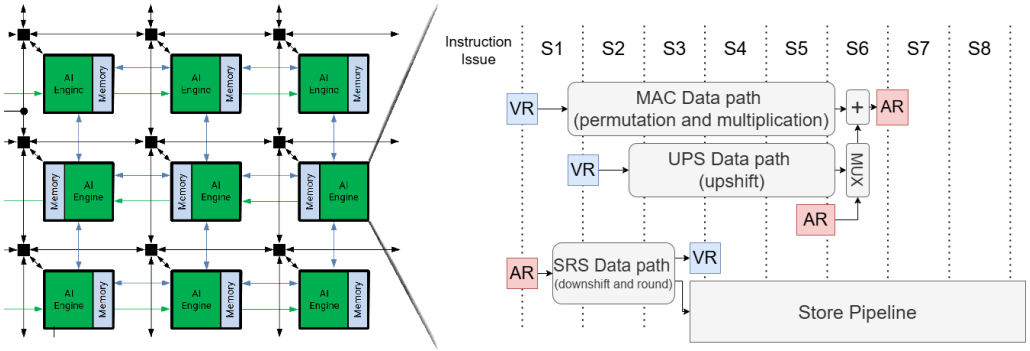


Figure 1: Left: diagram of the two-dimensional array of AIE tiles, taken from [AMDd]. Right: the MAC, UPS, and SRS data paths. The load/store pipelines for the `VLDA`/`VLDB` and `VST` instructions are not shown. The blue boxes are the vector register file and the red boxes are the accumulator register file. The "+" component denotes the accumulation in the MAC data path and the MUX component is a selector. The pipeline stages indicate how long each instruction takes and in which cycle (after instruction issue) the operands are read and the results are stored. The dotted lines mark the pipeline registers that separate successive stages.

SIMD vectors and VLIW data paths. Each AIE operates on 256-bit SIMD vectors, consisting of 16 lanes of `int16` or 8 lanes of `int32`. Next to vectors, there are also accumulators which have the same number of lanes but wider bit widths for storing intermediate results. There are 8 vector registers and 8 accumulator registers. Every AIE contains multiple independent data paths (see below), each with fixed latency and single-cycle throughput, that can each start a different operation concurrently. This makes it a VLIW architecture. From the programmer's perspective, the task is to structure computations so that the main compute data paths are utilized in every cycle, which is the central challenge addressed in this paper.

We now describe the key data paths and their instructions, illustrated in Figure 1. See Figure 2 for an example of an algorithm in assembly.

- **Memory:** Two load units (`VLDA`, `VLDB`) and one store unit (`VST`), each concurrently transferring one 256-bit vector per cycle with a 7-cycle latency. These can access the local memory element and the memory elements of three neighboring tiles.
- **Shift-round-saturate (SRS):** Performs an arithmetic right shift (division by a power of two) on each lane of an accumulator register, rounds the result according to a configurable rounding mode, and writes to a vector register (`VMOV.SRS`, 3-cycle latency). This path also supports a perfect shuffle permutation. Converting an accumulator to a vector and storing that vector can be done in one instruction (`VST.SRS`, 8-cycle latency), bypassing the vector register file.
- **Upshift (UPS):** Multiplies each lane of a vector by a power of two and writes the result to an accumulator (`VMOV.UPS`, 6-cycle latency but the source register is read in the second stage, so we can issue the instruction one cycle before the source vector is available).
- **Multiply-accumulate (MAC):** This is the main compute path. Starting from two vector registers, it applies a configurable permutation across them, performs element-wise multiplication between them and then stores the result as an accumulator or potentially adds the result to an existing accumulator. Instructions include `VMUL` (store result to an accumulator register), `VMAC` (accumulate with an existing

accumulator register), and `VMAC.UPS` (simultaneously upshift a vector register into an accumulator in the UPS data path and add it to the multiplication result), all with 6-cycle latency. The MAC can perform $8192/(\beta_1 \cdot \beta_2)$ multiplications per cycle for operands of β_1 and β_2 bits. The `VSEL` instruction (2-cycle latency) uses the permutation units in the MAC data path without multiplication. The permutation flexibility of the MAC data path is greater for `int32` than for `int16`. For `int16`, achieving certain permutations requires combining instructions across the MAC and SRS data paths.

2.4 Software pipelining and II approximation

The AIE compiler applies *software pipelining* to loops with independent iterations, overlapping the execution of successive iterations in time across the VLIW data paths [AMDc, p.67]. The compiler uses modulo scheduling [Rau95, KN11] to find the schedule. The key metric is the *Initiation Interval* (II): the number of cycles between starting successive iterations. Hence, the II represents the throughput of the loop in cycles per iteration. For *one* loop with N iterations, the total cycle count is $\text{PREAMBLE} + \text{II} \cdot N + \text{POSTAMBLE}$, where the preamble and postamble are the cycles required to fill and drain the pipelines of the data paths. For simplicity, we approximate this with only the term $\text{II} \cdot N$ assuming N is large and call this the *II approximation*. For a full kernel consisting of *multiple* loops executed in sequence, we can approximate the full kernel duration by summing the $\text{II} \cdot N$ terms of each loop. This is because preambles and postambles are typically small compared to $\text{II} \cdot N$ and the postamble of one loop overlaps with the preamble of the next. It is important to note that we will report loop duration almost exclusively in the II approximation for the rest of the text.

Figure 2 illustrates software pipelining for the computation $r_j = \lfloor (a_j + b_j q)/2^{16} \rfloor$, for $0 \leq j < n$, with 32-bit operands. Processing 8 lanes per iteration in `int32`, the naive loop requires 14 cycles per iteration (due to the instruction latencies), while the software-pipelined version achieves $\text{II} = 1$ cycle per iteration (with four instructions issued in every cycle, each instruction belonging to a different iteration of the loop and using a different data path). The memory buffers a, b, and r must reside in different memory banks to avoid memory stalls as they are all accessed in the same cycle (see Subsection 2.3).

Resource constraints impose a lower bound on the II. For instance, two stores per iteration force $\text{II} \geq 2$ since there is only one store unit. The *minimum* II (MinII) is the maximum lower bound across all data paths. Note that the MinII is not always achievable, nor is the compiler guaranteed to find it [KN11].

```
For (i = 0; i < n/8; i ← i + 1):
1: VLDA wr0 ← b[8i:8i+8]
2: VLDB wd0 ← a[8i:8i+8]
    ...
8: VMAC.UPS bm0 ← wd0 + wr0 ⊙ wc0
    ...
14:VST.SRS(s=16) bm0 → r[8i:8i+8]
LOOP BODY END: GOTO For
```

```
PREAMBLE...
For (i = 0; i < n/8; i ← i + 1):
1: VLDA wr0 ← b[8i:8i+8]
   |VLDB wd0 ← a[8(i−1):8(i−1)+8]
   |VMUL.UPS bm0 ← wd0 + wr0 ⊙ wc0
   |VST.SRS(s=16) bm0 → r[8(i−13):8(i−13)+8]
LOOP BODY END: GOTO For
POSTAMBLE...
```

Figure 2: An example of a naive loop schedule (left) and a software pipelined schedule (right) for the same loop with $N = \frac{n}{8}$ iterations. The `VST.SRS(s=16)` indicates that the accumulator undergoes a right shift by 16 bits with rounding in the SRS data path.

3 Manual Loop Fission for Efficient Software Pipelining

Loop fission (also known as loop distribution) [AK01] is a well-known compiler optimization that splits a loop containing multiple independent computations into separate, simpler loops. Modern compilers for general-purpose CPUs often apply this transformation automatically, but we find that the AIE compiler does not do so effectively for our workloads.

In this section, we explore how manually applying loop fission can enable the modulo scheduler to achieve near-optimal initiation intervals for each smaller loop, resulting in substantial performance gains. We propose a concrete workflow for implementing high-performance kernels on these architectures in Subsection 3.3. We remark that the content of this section generalizes beyond the NTT and beyond AIE: any compute-intensive kernel on a VLIW/SIMD architecture with limited registers and bank-partitioned memory stands to benefit from the same approach. We start by identifying a few reasons why the compiler achieves better schedules for smaller loops and how loop fission can be used to exploit this:

- **Scheduling complexity.** A complex loop body increases the probability that the compiler fails to find a schedule at the MinII, either because no valid schedule exists for the MinII or because the search space is too large. For the smaller loops, we can reason about achievability and assist the compiler through intrinsics [AMDb], register pinning, and manual unrolling.

- **Register pressure.** A complex loop often requires values to remain live for many cycles (due to long data path latencies), driving up register pressure and forcing register spilling to memory since there are only 8 vector and 8 accumulator registers. Moreover, a loop with multiple operations may require vector moves between vector registers or accumulator registers but the move unit can only move one vector or one *half* of an accumulator per cycle, so the moves can easily become a bottleneck. Loop fission acts as *controlled spilling*: intermediate results are stored in dedicated memory buffers between loops, avoiding uncontrolled spills and the vector-move bottleneck.

- **Configuration registers.** Different operations may require different data path configurations. In a single loop, reconfiguring these registers for different operations adds dependencies and latencies; separate loops avoid this overhead.

3.1 Methodology

Given a mathematical formula to implement with our kernel, we can directly map the mathematical operations to instructions in the MAC or SRS data paths, of which the MAC data path is usually the main bottleneck, posing a fundamental mathematical lower bound on the achievable II. Ideally, we should find an implementation of the formula where the MAC or SRS data path is fully utilized in every cycle, so that the achieved II matches this mathematical lower bound. However, for complex formulas such as those in an NTT layer, the compiler often fails to obtain a schedule that achieves this lower bound based on the MAC and SRS data paths only, due to the problems identified above. Specifically, complex loops may add overhead such that the MinII is not determined by the instructions directly mapped from the formula. In such cases, splitting the kernel into smaller loops that are executed sequentially on the same AIE can be highly beneficial.

Note that the controlled spilling by splitting loops increases the number of memory instructions. Hence, we must pay attention that the memory instructions do not become the new bottleneck for the smaller loops.

3.2 Trade-offs and constraints

Loop fission increases the total pipeline preamble/postamble overhead. We mitigate the overhead by only splitting when it yields a measurable II improvement, and by overlapping loop preambles and postambles where the stores of one loop use a different bank than the loads of the next. Loop fission also increases memory consumption, since we now require intermediate buffers to store partial results, which in turn increases the likelihood of memory stalls. To avoid memory stalls entirely, we make sure that all memory buffers used by the same loop reside in different memory banks as they might be accessed simultaneously due to the VLIW architecture. We call this the *bank-disjointness invariant*. Note that, for a fair reporting of the hardware resource usage, we constrain ourselves not to use memory banks from tiles that are not in use by our algorithm. Hence we have four memory banks per AIE to work with, although multiple buffers belonging to different loops may share the same bank or different logical buffers may share the same physical memory section.

3.3 General implementation workflow

Combining all the above insights, we now describe our general workflow for designing kernels on the AIE architecture. This workflow is not specific to the NTT and can be applied to other compute-intensive algorithms on similar architectures:

1. Directly implement the mathematical formula in one large loop by mapping the mathematical operations to instructions in the MAC or SRS data paths, and determine a lower bound on the II based on the number of compute operations.

2. Simplify the original loop by applying loop fission, splitting the computation into simpler loops with their own input and output buffers. For each loop, make sure to preserve the bank-disjointness invariant and ensure that the MAC data path is effectively utilized in every cycle and hence remains the MinII bottleneck.

3. Determine which permutations are required and how to realize them inside the MAC and SRS data paths during instruction execution without increasing the II.

4. For each loop, reason about how to achieve the MinII with an efficient assembly schedule through register pinning and manual loop unrolling.

5. Implement the loop in C++ using the AIE API and intrinsics, structuring the source code so that the compiler is guided toward the intended schedule.

If one of the above steps cannot be achieved, then fission the original loop differently.

It is important to note that all code is written in C++ using the AIE API [AMDa] and intrinsics [AMDb]. Since the AIE compiler does not support hand-written assembly but does report the generated assembly code, the developer must manually design an efficient assembly schedule and then guide the compiler towards that efficient machine code by carefully choosing intrinsics, controlling loop structure, and managing register allocation at the C++ level.

4 Barrett Reduction Primitives

In this section, we implement the Barrett reduction primitives, based on the proposed methodology, ensuring the compiler can schedule them efficiently on the AIE architecture. We start by describing and motivating our choices of the Barrett primitive parameters and then present the kernel implementations of partial and full reduction for both `int16` and `int32` datatypes. Following the workflow of the previous section, each implementation

will consist of one or more software pipelined loops executed in sequence. The compiler-generated assembly code is presented[3] in Figure 3.

4.1 Barrett primitive parameters

Value of parameter K. Becker et al. [BHK+21] argue that, in a single-word implementation, $R = 2^K$ should be as large as possible to increase the precision of the approximation. In this work, we use the same principle to maximize the headroom for *lazy reduction*, i.e. to accumulate several Barrett-reduced values before reducing the sum. However, the Barrett multiplier grows with K and we must ensure it still fits in the available word size. Let $\beta \in \{16, 32\}$ denote the machine word size. From Definition 1, the Barrett multiplier satisfies $\omega' \in \left[-\frac{R}{2}, \frac{R}{2} - 1\right]$ when ω is reduced to $\left[-\frac{q-1}{2}, \frac{q-1}{2}\right]$. We therefore maximize K subject to ω' fitting in a β-bit signed two's complement integer (with range $[-2^{\beta-1}, 2^{\beta-1} - 1]$), which yields $K = \beta$.

Nearest integer rounding. Our SRS unit supports all three integer rounding functions. We choose the nearest integer function because it gives the most headroom for lazy reduction. The floor and ceiling functions have output ranges of the same size as the nearest integer rounding function, but their ranges are entirely positive or entirely negative. This halves the number of outputs that can be accumulated before overflow occurs.

We now quantify this headroom for lazy reduction. For $K = \beta$, using $|a| \leq 2^{\beta-1} = R/2$ in Theorem 1, we obtain a more specific output bound for the Barrett primitive:

$$\left|\mathbf{bar}_q^{\lfloor\,\rceil}(a, \omega)\right| \leq \frac{q}{2}\left(1 + \frac{|a|}{R}\right) \leq \frac{q}{2}\left(1 + \frac{R}{2R}\right) \leq \frac{3}{4}q \tag{5}$$

To fit the output in a β-bit signed integer, the prime modulus must satisfy $q < \frac{4}{3}2^{\beta-1}$. For even smaller q, k partially reduced values (each bounded by $\frac{3}{4}q$) can be summed as long as $k < \frac{2^{\beta-1}}{\frac{3}{4}q}$. The superscript $\lfloor\,\rceil$ will usually be omitted for brevity.

4.2 Kernel implementations

For a partial reduction of a modular multiplication (Equation 4), the computation per element is $\mathbf{bar}_q(a_i, \omega_i) = m_i - m_i' q$, with $m_i := a_i\omega_i$ and $m_i' := \lfloor a_i\omega_i'/R \rceil$. These are vectorized in batches of 16 (`int16`) or 8 (`int32`) lanes, with a_i, ω_i, and ω_i' stored consecutively. For reduction without multiplication ($\omega_i = 1$), the twiddle loads and the $a_i\omega_i$ multiplication are omitted.

The Barrett reduction maps well to the AIE hardware: the multiplications $a_i\omega_i$ and $a_i\omega_i'$ use `VMUL` instructions, and $m_i - m_i' q$ uses a `VMAC` instruction, while the division by $R = 2^K$ and rounding use the SRS data path. All values except $m_i = a_i\omega_i$ fit in the β-bit word size. For m_i, only the lower β bits are needed because $\mathbf{bar}_q(a_i, \omega_i)$ fits in β bits (lying in $[-\frac{3}{4}q, \frac{3}{4}q]$), so the upper bits of m_i and $m_i' q$ cancel[4] and so we don't have to store those upper bits.

We now present the kernel implementations for both datatypes, each consisting of one or more loops following the loop fission strategy of Subsection 3.3.

[3] We present the resulting compiler-generated assembly of the Barrett reduction primitives for two reasons. First, to illustrate the loop schedules and corresponding performance. Second, to show how similar optimizations can be achieved on other VLIW/SIMD platforms and how future platforms may be designed for efficient cryptographic primitives.

[4] Formally: $\mathbf{bar}_q(a_i, \omega_i) = m_i - m_i' q = \mathbf{mod}_{2^\beta}(m_i - m_i' q) = \mathbf{mod}_{2^\beta}(\mathbf{mod}_{2^\beta}(m_i) - m_i' q)$.

Partial reduction primitives for `int16`. With 16 lanes per operation, the computation proceeds as follows. A first `VMUL` instruction computes $a_i\omega'_i$, which is then passed through the SRS data path (`VMOV.SRS`) to perform the division by R and rounding, producing m'_i. A second `VMUL` instruction then computes the final result $a_i\omega_i - m'_i q$ in one step, exploiting the fact that 32 multiplications of 16-bit values can be performed per cycle[5]. The two `VMUL` instructions are separated by nine cycles (due to the SRS latency), but both consume the input vector a_i. Since more iterations are in flight than there are registers available, a_i cannot remain in a register for those nine cycles and must be loaded a second time. This yields a total of four vector loads (a_i twice, ω_i, and ω'_i), one store, and two passes each through the MAC and SRS data paths. The MinII of 2 is achievable and the compiler-generated schedule is shown in Figure 3d.

For reduction without multiplication ($\omega_i = 1$), the loads of ω_i and ω'_i (both constant for all i) are unnecessary and the first multiplication $a_i\omega_i$ is omitted. However, the two remaining MAC passes (for $a_i\omega'_i$ and $a_i - m'_i q$) still dominate, so the II remains 2.

Partial reduction primitives for `int32`. With 8 lanes and the MAC data path being able to perform 8 multiplications of 32-bit values per cycle, the theoretical MinII is 3: one cycle each for the two products $m_i = a_i\omega_i$ and $m'_i = a_i\omega'_i$, and one cycle for $m_i - m'_i q$ (computed as a `VMAC` instruction). Attempting to implement this in a single loop, the register file and configuration register constraints from Section 3 prevent the compiler from achieving $\mathrm{II} = 3$. We therefore apply loop fission, splitting the computation into two loops. The first loop (Figure 3a) computes m'_i through a load, multiplication, SRS pass, and store, achieving $\mathrm{II} = 1$. The second loop (Figure 3c) loads a_i, ω_i, and m'_i, computes $m_i = a_i\omega_i$ via `VMUL` and then $m_i - m'_i q$ via `VMAC`, and stores the result, achieving $\mathrm{II} = 2$. The combined $\mathrm{II} = 3$ matches the theoretical minimum.

Without the $a_i\omega_i$ multiplication, the second loop needs only one MAC instruction (for $a_i - m'_i q$) and achieves $\mathrm{II} = 1$ (Figure 3b). The combined II is then $1 + 1 = 2$ and matches the theoretical minimum of the two multiplications.

Full reduction primitives. Although we will not use full reductions in our NTT implementations, we also implement full reduction primitives as they are useful in other contexts and the ideas can generalize to other NPU platforms.

As a shorthand, let's denote $r_i := \mathbf{bar}_q(a_i, \omega_i) \in [-\frac{3}{4}q, \frac{3}{4}q]$ the result of the partial Barrett reduction. Converting a partial reduction result $r_i \in [-\frac{3}{4}q, \frac{3}{4}q]$ to a full reduction $r'_i \in [0, q)$ requires adding q whenever r_i is negative. We implement this branchlessly by extracting a sign mask $s_i \in \{-1, 0\}$. Specifically, we arithmetic right-shift the β-bit value r_i by β positions in the SRS data path using the floor rounding mode. Because the shift is arithmetic, the sign bit is replicated across all bit positions: the result is $s_i = -1$ (all ones) if r_i was strictly negative and $s_i = 0$ if r_i was non-negative. The full reduction is then $r'_i = r_i - s_i q$, computed with a single `VMAC`.

A single-loop implementation with $\mathrm{II} = 2$ (the minimum, set by the two SRS passes) is not achievable in practice. Applying loop fission, we split into two loops: the first extracts the sign mask s_i via a load, UPS upshift to convert to an accumulator, SRS right-shift, and store ($\mathrm{II} = 1$), and the second computes $r'_i = r_i - s_i q$ via two loads for r_i and s_i, a `VMAC`, and a `VST.SRS` ($\mathrm{II} = 1$). The combined cost is 1+1=2 *additional* II for all primitives.

For a `int32` Barrett multiplication, however, we can do better since the sign extraction can be implemented in the second loop for the partial reduction (Figure 3c). That loop has $\mathrm{II} = 2$ but uses the SRS and store data paths only once, leaving one idle SRS slot and

[5] To make the compiler emit $a_i\omega_i - m'_i q$ as a single instruction, we use an intrinsic from the multiply-and-subtract suite [AMDb].

```
PREAMBLE...
For (i = 0; i < n/8; i ← i + 1):
1: VLDA wd0 ← a[8i:8i+8]
   |VLDB wc0 ← ω'[8i:8i+8]
   |VMUL bm0 ← wd0 ⊙ wc0
   |VST.SRS(s=32) bm0 → m'[8(i−13):8(i−13)+8]
LOOP BODY END: GOTO For
POSTAMBLE...
```

(a) `int32` Barrett multiplication and reduction (loop 1/2). For Barrett reduction, the load of ω'_i disappears as it is a constant for all i.

```
PREAMBLE...
For (i = 0; i < n/8; i ← i + 1):
1: VLDA wd0 ← a[8i:8i+8]
   |VLDB wr0 ← m'[8i:8i+8]
   |VMAC.UPS bm0 ← wd0 − wr0 ⊙ wc0
   |VST.SRS(s=0) bm0 → r[8(i−13):8(i−13)+8]
LOOP BODY END: GOTO For
POSTAMBLE...
```

(b) `int32` Barrett reduction (loop 2/2). Assume that `wc0` holds eight copies of q.

```
PREAMBLE...
For (i = 0; i < n/8; i ← i + 1):
1: VLDA wd0 ← a[8i:8i+8]
   |VLDB wc1 ← ω[8i:8i+8]
   |VMAC bm0 ← bm0 − wr0 ⊙ wc0
   |VST.SRS(s=0) bm0 → r[8(i−7):8(i−7)+8]
2: VLDA wr0 ← m'[8i:8i+8]
   |VMUL bm0 ← wd0 ⊙ wc1
LOOP BODY END: GOTO For
POSTAMBLE...
```

(c) `int32` Barrett multiplication (loop 2/2). Assume that `wc0` holds eight copies of q.

```
PREAMBLE...
For (i = 0; i < n/16; i ← i + 1):
1: VLDA wr0 ← a[16i:16i+16]
   |VLDB wr1 ← ω'[16i:16i+16]
   |VMUL bm1 ← wd0 ⊙ wr2 + wd1 ⊙ wr3
   |VST.SRS(s=0) bm1 → r[16(i−11):16(i−11)+16]
2: VLDA wd0 ← a[16(i−4):16(i−4)+16]
   |VLDB wr2 ← ω[16(i−4):16(i−4)+16]
   |VMUL bm0 ← wr0 ⊙ wr1 + wr0 ⊙ wr1
   |VMOV.SRS(s=17) bm0 → wd1
LOOP BODY END: GOTO For
POSTAMBLE...
```

(d) `int16` Barrett multiplication and reduction. Assume that `wr3` holds 16 copies of $-q$. For Barrett reduction, the loads of ω_i and ω'_i and the multiplication $a_i\omega_i$ disappear.

Figure 3: Implementation of partial Barrett reduction and multiplication primitives

one idle store slot per iteration, where we can insert the sign extraction[6]. A separate loop then computes $r'_i = r_i - s_i q$ at II = 1, reducing the total overhead to one additional II.

4.3 Results

Table 1 summarizes the cycle counts of all proposed Barrett primitives. We report the number of cycles per batch (16 or 8 lanes) as the sum of II over all the loops since one additional batch requires one additional iteration of each loop, adding II to the total cycle count. For partial Barrett multiplication, the effective multiplier utilization is 75% for `int16` and nearly 100% for `int32`. Recall that the prime modulus is limited to $q < 2^{\beta+1}/3$.

5 NTT Implementations

We present three NTT schemes based on radices $r = 2, 4, 8$, all starting from Equation 3 and sharing six design principles: (i) our loop-fission workflow with free permutations in

[6]However, simply adding a `VST.SRS(s=32) bm0` → $s_{[(8(i-7):8(i-7)+8)]}$ would not work as `bm0` is overwritten in that second cycle by the VMUL instruction of the subsequent iteration, issued six cycles earlier. This is fixed by manually unrolling the loop and alternating accumulators between odd and even iterations.

Table 1: Number of cycles per batch for the reduction primitives (sum of II). The batch size is 16 lanes for `int16` and 8 lanes for `int32`. Note that we ignore the loop preamble and postamble overhead, which is amortized across iterations.

Primitive	`int16`	`int32`
Barrett multiplication (partial)	2 cycles (1 loop)	3 cycles (2 loops)
Barrett reduction (partial)	2 cycles (1 loop)	2 cycles (2 loops)
Barrett multiplication (full)	4 cycles (3 loops)	4 cycles (3 loops)
Barrett reduction (full)	4 cycles (3 loops)	4 cycles (4 loops)

the compute data paths, (ii) all reductions use Barrett primitives, (iii) all twiddle factors and Barrett multipliers are precomputed and stored in the memory tiles, (iv) only partial Barrett reductions (no full reductions), (v) lazy reduction of accumulations to minimize Barrett calls, and (vi) constant-geometry transforms [Pea68] to reuse the same kernel(s).

We always describe the `int16` case, with `int32` analogous. The radix-2 and radix-4 implementations use the Barrett primitives of Section 4, while radix-8 uses a new specialized reduction block. Each scheme also has a custom unit for its butterfly and data mapping.

Radix trade-offs. Recall from Subsection 2.1 that for radix r, each of the $\log_r(n)$ layers consists of a stage of elementwise twiddle-factor multiplications followed by a stage of n/r modular r-by-r matrix multiplications (the radix-r butterflies). Different radices offer different trade-offs. Radix-2 (Subsection 5.1) minimizes $\mathbb{Z}_q$ operations and has butterflies without reductions, but requires the most layers and hence the most twiddle-multiplication stages, giving the largest number of reductions as no lazy reduction is possible. Larger radices have fewer layers, and thus fewer elementwise multiplication stages, at the cost of more complex butterflies with more modular multiplications. We explore two strategies for higher radices. The first reduces all non-trivial butterfly multiplications with Barrett operations, beneficial only for radix-4, whose butterfly matrix needs only two modular multiplications (Subsection 5.2). The second strategy leaves butterfly multiplications unreduced, producing double-width outputs, and then uses wider Barrett multiplications between layers, favoring radix-8 (Subsection 5.3).

Pipeline architecture. Since the precomputed twiddle factors and Barrett multipliers do not all fit in one tile's memory, we distribute the $\log_r(n)$ layers across a pipeline of AIE tiles (e.g. one tile per layer), each storing only its own layer's factors. Thanks to constant-geometry transforms, all tiles run the same kernel(s), improving load balancing. Tiles communicate via shared memory banks (the last loop of a tile writes to a bank read by the next tile), avoiding the slow streaming interconnect.

Normalized metric. To compare radices fairly, we combine throughput and tile usage into the area-throughput product (the area multiplied by the time per sample).

5.1 Radix-2 scheme (R2)

High-level overview. Our first implementation uses the radix-2 FFT. The full transform is performed in a pipeline of $\log_2(n)$ AIE tiles, each performing one layer in one kernel consisting of the following three blocks:

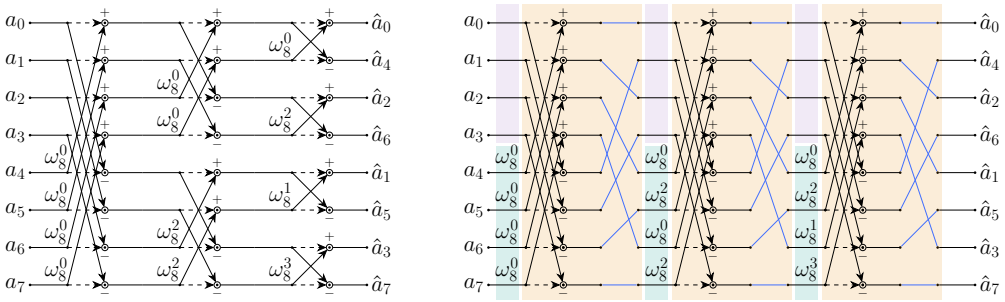
1. the partial Barrett reduction primitive on the top half of the coefficients (2 cycles per 16 lanes for `int16`, 2 cycles per 8 lanes for `int32`, purple box in Figure 4b);
2. the partial Barrett multiplication primitive on the bottom half of the coefficients (2 cycles per 16 lanes for `int16`, 3 cycles per 8 lanes for `int32`, green box in Figure 4b);

3. the radix-2 butterflies fused with a perfect-shuffle permutation in one loop without reduction (2 cycles per 32 lanes for `int16`, 4 cycles per 32 lanes for `int32`, orange box in Figure 4b). The data mapping below explains why we perform this permutation.

Summing the three blocks gives $\frac{2}{16}\frac{n}{2} + \frac{2}{16}\frac{n}{2} + \frac{2}{32}n = 0.1875n$ cycles per layer for `int16` and $\frac{2}{8}\frac{n}{2} + \frac{3}{8}\frac{n}{2} + \frac{4}{32}n = 0.4375n$ for `int32`, in the II approximation. Blocks 1 and 2 use the primitives of Section 4 directly. Block 3 is explained below.

Data mapping (blocks 1 and 2). The forward NTT is shown in Figure 4a. The first layer is highly efficient because the coefficients to be Barrett reduced lie in the top half of the buffer and those to be Barrett multiplied lie in the bottom half. This lets us use the optimized SIMD Barrett primitives of Section 4, and this structure will also enable an efficient implementation of the radix-2 butterflies and permutation.

Since the structure of the first layer is so beneficial, we may force each layer to have this same structure by applying a *perfect shuffle* permutation on the coefficients after each layer, as illustrated in Figure 4b. In this case, each layer has the same structure and each AIE tile in the pipeline implements the exact same kernel. This type of implementation is called a constant-geometry FFT [Pea68].



(a) Naive Cooley-Tukey NTT (b) Cooley-Tukey NTT with perfect shuffles.

Figure 4: Illustration of the efficient data mapping for the radix-2 scheme. The purple, green, and orange boxes correspond to the three blocks of the kernel.

Butterfly and permutation loop (block 3). Finally, one loop performs the butterfly additions and subtractions together with the perfect-shuffle permutation. The result is left unreduced (lazy reduction), so only additions and subtractions are needed, which means the butterfly can be implemented in the MAC data path as a matrix-vector multiply with a matrix of ± 1 entries. Since $2n$ multiplications are needed per layer and the MAC data path performs 32 multiplications per cycle, the theoretical minimum is $\frac{2n}{32}$. Each iteration of the loop computes 16 consecutive butterflies with two vector loads (one from each half of the buffer), two `VMUL` instructions for the permutation and matrix multiplication, and two `VST` instructions to store the 32 consecutive outputs. For `int32`, the ± 1 matrix entries fit in 16 bits, so the butterfly uses 32-bit-by-16-bit multiplications, of which 16 can be performed per cycle. This achieves the theoretical minimum of two cycles per 32 buffer elements for `int16` and four for `int32`, in the II approximation.

Constraints and inverse transform. The memory constraint (fitting input, output, twiddle factors, Barrett multipliers, and one $n/2$-element temporary buffer in four 8 KB banks) limits the transform to $n \leq 2^{12}$ for `int16` and $n \leq 2^{11}$ for `int32`. The modulus is constrained to $q < \frac{4}{3} \cdot \frac{2^{15}}{2} = \frac{2^{16}}{3}$ for `int16` and $q < \frac{4}{3} \cdot \frac{2^{31}}{2} = \frac{2^{32}}{3}$ for `int32` to ensure butterfly outputs fit in 16/32-bit signed integers without reduction (lazy reduction) due

to Equation 5. The inverse transform uses the Gentleman-Sande configuration with an *unshuffle* permutation after each layer. The structure mirrors the forward transform, with a different permutation, achieving the same performance in the II approximation.

5.2 Radix-4 scheme (R4)

High-level overview. Unlike the radix-2 scheme, whose butterflies contain no multiplications, the radix-4 scheme uses the partial Barrett multiplication primitive for *all* multiplications, both the between-layer twiddle multiplications and the two non-trivial modular multiplications with ω_4 inside the butterfly. Only additions and subtractions are left unreduced. In exchange, it needs only half as many layers. Starting from the canonical radix-4 formulation in Equation 6, we convert it to the actual butterfly computation in Equation 7. The full transform thus runs in a pipeline of $\log_4(n)$ AIE tiles, each computing one radix-4 layer in a kernel of three blocks:

1. the partial Barrett multiplication primitive applied to the four inputs and their twiddle factors, producing A_c^1 up to A_c^4 in Equation 7 ($\frac{2}{16} \cdot 4\frac{n}{4} = 0.125n$ cycles per layer for `int16`, 3 cycles per 8 lanes for `int32`);

2. a second partial Barrett multiplication primitive for the two multiplications with ω_4 inside the butterfly, producing A_c^5 and A_c^6 in Equation 7 ($\frac{2}{16} \cdot 2\frac{n}{4} \approx 0.0625n$ cycles per layer for `int16`);

3. performing and summing two matrix multiplications of Equation 7, computed without reduction (8 cycles per 64 outputs for `int16`, 16 for `int32`, explained below).

Summing the three blocks gives $\frac{2}{16}4\frac{n}{4} + \frac{2}{16}2\frac{n}{4} + \frac{8}{64}n \approx 0.3125n$ cycles per layer for `int16` and $\frac{3}{8}4\frac{n}{4} + \frac{3}{8}2\frac{n}{4} + \frac{16}{64}n \approx 0.8125n$ for `int32`. For `int16` the throughput is thus about 1.67× worse than radix-2, but the design uses only half the AIE tiles, making it more efficient in the area-throughput product: fewer reductions are needed overall, and the Barrett reductions traded for Barrett multiplications are equally costly in `int16`. For `int32` the gains diminish, as Barrett multiplications are slower than reductions.

Blocks 1 and 2 reuse the partial Barrett multiplication primitive of Section 4 directly. Block 3 is the new radix-4 butterfly loop. We derive the butterfly structure (blocks 1 and 2) first, then the matrix-multiplication loop (block 3).

Butterfly structure (blocks 1 and 2). For layer L, with $0 \leq c < n/4$, both the twiddle multiplication and butterfly stages are given by Equation 6, where $n_L = r^{L+1}$, '$\overset{\text{B}}{\odot}$' denotes element-wise partial Barrett multiplication, $l_2(c) = \text{bitrev}(\mathbf{mod}^{\lfloor\,\rfloor}_{n_{L-1}}(c))$, and A and B are the input and output buffers. As in radix-2, this is a constant-geometry structure: each butterfly takes inputs from four quarters of the buffer and outputs are placed consecutively. We note that the inner two rows of the radix-4 butterfly must be swapped for this to work.

We reduce the two modular multiplications with ω_4 in the butterfly, so only two partial Barrett multiplications are needed: $A_c^5 := \omega_4 \overset{\text{B}}{\cdot} A_c^2$ and $A_c^6 := \omega_4 \overset{\text{B}}{\cdot} A_c^4$. We then rewrite the matrix-vector multiplication with all entries in $\{-1, 0, 1\}$ and A_c^5, A_c^6 as additional inputs (bringing us closer to Equation 7), which produces little further growth. When $q < 2^{15}/3$, a sum of four partially reduced values (each bounded by $\frac{3}{4}q$) still fits in the `int16` range, so no intermediate reductions are needed (lazy reduction).

$$\begin{bmatrix} B_{4c+0} \\ B_{4c+1} \\ B_{4c+2} \\ B_{4c+3} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & \omega_4 & -1 & -\omega_4 \\ 1 & -\omega_4 & -1 & \omega_4 \end{bmatrix} \underbrace{\left(\begin{bmatrix} A_{c+0} \\ A_{c+\frac{n}{4}} \\ A_{c+2\frac{n}{4}} \\ A_{c+3\frac{n}{4}} \end{bmatrix} \overset{\mathrm{B}}{\odot} \begin{bmatrix} 1 \\ \omega_{n_L}^{l_2(c)} \\ \omega_{n_L}^{2l_2(c)} \\ \omega_{n_L}^{3l_2(c)} \end{bmatrix} \right)}_{\left[A_c^1; A_c^2; A_c^3; A_c^4\right]} \tag{6}$$

$$\begin{bmatrix} B_{4c+0} \\ B_{4c+1} \\ B_{4c+2} \\ B_{4c+3} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ -1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} A_c^2 \\ A_c^4 \\ A_c^5 \\ A_c^6 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} A_c^1 \\ A_c^3 \end{bmatrix} \tag{7}$$

Butterfly matrix multiplication (block 3). For radix-2, we kept two vectors per iteration in registers. Radix-4 would require 6 vectors (A_c^1 up to A_c^6, each for 16 values of c), which is infeasible. We therefore split the matrix multiplication into two terms (Equation 7), computed with a `VMUL` and a `VMAC` instruction respectively. Each instruction computes 32 multiplications and both terms require 8 multiplications with ± 1, so processing 16 values of c per iteration takes four `VMUL` and four `VMAC`. Concretely, we load the 16-lane vectors of A_c^2, A_c^4, A_c^5 and A_c^6 and compute the first term, each `VMUL` writing a different accumulator. Then we load A_c^1 and A_c^3 and apply four `VMAC`. Finally, we store the four accumulators with `VST.SRS`. This gives 8 cycles per 64 outputs in the II approximation for `int16`, and 16 cycles per 64 outputs for `int32` with 16-bit-by-32-bit arithmetic.

Constraints and inverse transform. The memory constraints limit the scheme to $n \leq 2^{11}$ for `int16` and $n \leq 2^{10}$ for `int32`. Due to the lazy reduction, we also have $q < 2^{15}/3$ for `int16` and $q < 2^{31}/3$ for `int32`. For the inverse transform, we apply a bit-reversal (Section 6) to the forward NTT outputs and then reuse the same implementation with different twiddles.

5.3 Radix-8 scheme (R8)

High-level overview. Unlike the radix-4 scheme, which reduces non-trivial modular multiplication inside the butterfly, the radix-8 scheme leaves all modular multiplications inside the radix-r butterfly *unreduced*, producing 32-bit outputs. These are then multiplied with the twiddle factors between layers and the result is reduced back to 16 bits, using a new 32-bit-by-16-bit Barrett multiplication block. Each of the $\log_8(n)$ layers is thus built from two blocks, both explained below:

1. the radix-8 butterfly with 16-bit inputs, computed without reduction so that its outputs are 32-bit (38 cycles per 16 butterflies, i.e. $0.297n$ cycles per layer);
2. the 32-to-16-bit Barrett multiplication reducing the between-layer twiddle products to 16 bits (5 cycles per 16 lanes, i.e. $\frac{5}{16}n = 0.3125n$ cycles per layer).

Together these two blocks need more than four memory banks due to intermediate buffers, so they are split across two AIE tiles, giving $2\log_8(n)$ tiles in total. The throughput is determined by the slower tile, at $0.3125n$ cycles per radix-8 layer. This is about $1.67\times$ worse than radix-2 while using $\frac{2\log_8(n)}{\log_2(n)} \approx 0.67\times$ the tiles, making it less efficient in the

area-throughput product ($1.67 \times 0.67 \approx 1.1$). This is the least efficient scheme on the first-generation AIE, but it has the highest arithmetic intensity and will benefit most from the doubled compute throughput on AIE2 (Section 7). An `int32` version is not feasible as it would require 64-bit intermediates. The inverse transform follows the same approach as radix-4. We explain the butterfly first, then the new 32-to-16-bit Barrett multiplication block and the motivation for choosing $r = 8$ for this strategy.

Radix-8 butterfly (block 1). Ideally, since 32 16-bit-by-16-bit multiplications can be performed per cycle, we would need 32 cycles per 16 butterflies. In reality, without going into details, our implementation requires 38 cycles per 16 butterflies with a similar approach to the radix-4 scheme. This is more than the ideal 32 cycles due to register constraints.

32-to-16-bit Barrett multiplication (block 2). Our implementation exploits the fact that only the lower 16 bits of m_i and m'_i are needed for the final result (the upper bits cancel, as in the footnote of Section 4). It is implemented in three loops: (i) multiply the 32-bit coefficient with the 32-bit Barrett multiplier to compute m'_i (1 cycle per 8 lanes); (ii) truncate m'_i to its lower 16 bits via a pass through the MAC data path (1 cycle per 16 lanes); (iii) compute the 32-by-16-bit twiddle multiplication $m_i = a_i\omega_i$ and the 16-by-16-bit reduction $m_i - m'_i q$ in two MAC instructions (2 cycles per 16 lanes). The total is 5 cycles per 16 reductions. Since all elements must be reduced, this yields a total of $\frac{5}{16}n$ cycles per layer.

Radix. We chose a radix of $r = 8$ for this approach by balancing the $O(r^2)$ cost of the butterfly against the number of layers. Each layer has n/r butterflies of r^2 multiplications each, at a cost of $\frac{r^2 \cdot n/r}{32} = \frac{rn}{32}$ cycles for one full layer, plus $\frac{5}{16}n$ cycles for the Barrett reductions. The total cost for all layers is $\left(\frac{rn}{32} + \frac{5n}{16}\right) \log_r(n)$. The minimum integer solution is $r = 9$, but we prefer the power-of-two $r = 8$ with a negligibly higher cost.

5.4 Results

Implementation results. Table 2 gives the throughput and latency of all *implemented* schemes for the maximum n permitted by their memory constraints, as measured using the Vitis hardware simulator. The number of AIE tiles (**#AIE**) equals the number of pipeline stages. We compute the latency by multiplying the throughput in cycles per transform by the number of pipeline stages. The reported values exclude inverse normalization and full reduction. For a rough comparison of resource efficiency across transform sizes, we report the normalized cost $C = \#\text{AIE} \cdot T/(n \log_2 n)$, where T is the throughput in cycles per transform and lower values indicate better efficiency. Among the `int16` schemes, radix-8 has the highest normalized cost on AIE1, so we do not discuss it further in this section.

Table 2: Vitis hardware-simulator results for the *implemented* schemes at the maximum n permitted by their memory constraints, excluding inverse normalization and full reduction.

Scheme	n	**#AIE**	**Throughput**	**Latency**	C
Radix-2 `int16` CT forward	2^{12}	12 ($\log_2(n)$)	952 cc	11,424 cc	0.2324
Radix-2 `int16` GS inverse	2^{12}	12 ($\log_2(n)$)	918 cc	11,016 cc	0.2241
Radix-4 `int16`	2^{11}	6 ($\lceil \log_4(n) \rceil$)	808 cc	4,848 cc	0.2152
Radix-8 `int16`	2^{11}	8 ($2\lceil \log_8(n) \rceil$)	770 cc	6,160 cc	0.2734
Radix-2 `int32`	2^{11}	11 ($\log_2(n)$)	1,028 cc	11,308 cc	0.5020

Analysis. Table 3 reports the kernel duration, utilized AIE tiles, and parameter constraints of the radix-2 and radix-4 schemes, each measured at its maximum achievable n. We report both the II approximation (**II**) and cycle-accurate simulation (**S**). The simulation exceeds the II approximation due to pipeline prologues/epilogues and synchronization overhead, confirming that the II approximation is a useful but slightly optimistic predictor. A radix-4 `int32` implementation was not developed, but the same optimizations apply, so we give its II approximation.

To verify the simulation, we also performed physical measurements (**P**) on the AIE device for the radix-2 `int16` implementation, taking the slope of a linear regression on latency measurements over increasing batch sizes. The resulting throughput of 944 cycles per layer is close to the simulated 952, a difference we attribute to measurement noise, from which we conclude that the simulations are accurate. A power estimation using AMD's Power Design Manager yields a steady-state consumption of about 2.964 W.

Comparing the radix-2 and radix-4 schemes. A fair comparison uses the same n, so take the case of `int16` with $n = 2^{11}$. From the gap between simulation and II approximation, we estimate a fixed loop and synchronization overhead of $952 - 768 = 184$ cycles for radix-2. The radix-2 kernel thus requires $0.1875 \cdot 2^{11} + 184 \approx 568$ cycles and the radix-4 kernel 808 cycles. However, a radix-4 kernel performs the equivalent of two radix-2 stages, needing only half as many AIE tiles for a full transform, so it is more efficient overall: by about $\frac{0.1875n \cdot 2}{0.3125n} \approx 1.2\times$ in both latency and area-throughput product in the II approximation. The gap is even larger in practice, as radix-2 has more overhead (184 cycles) than radix-4 (168) due to less overlap of loop preambles and postambles. The radix-4 scheme is therefore preferable when its stricter parameter constraints are met, and radix-2 otherwise. For `int32`, radix-4 is again more efficient, though by a smaller margin.

Table 3: Duration in cycles, utilized AIE tiles, and parameter constraints for the radix-2 and radix-4 kernels. For the radix-2, we report the numbers of the forward or inverse implementation, whichever is slower. **II** = II approximation, **S** = cycle-accurate simulation, **#AIE** = number of AIE tiles.

		Kernel duration in cycles			**Parameter constraints**	
		Radix-2	**Radix-4**		**Radix-2**	**Radix-4**
`int16`	n	2^{12}	2^{11}	n	$\leq 2^{12}$	$\leq 2^{11}$
	II	768 $(0.1875n)$	640 $(0.3125n)$	q	$< \frac{2^{16}}{3}$	$< \frac{2^{15}}{3}$
	S	952	808			
	#AIE	12 $(\log_2(n))$	6 $(\lceil \log_4(n) \rceil)$			
`int32`	n	2^{11}	2^{10}	n	$\leq 2^{11}$	$\leq 2^{10}$
	II	896 $(0.4375n)$	832 $(0.8125n)$	q	$< \frac{2^{32}}{3}$	$< \frac{2^{31}}{3}$
	S	1028	-			
	#AIE	11 $(\log_2(n))$	5 $(\lceil \log_4(n) \rceil)$			

Comparison with prior AIE work. Table 4 compares our results with the existing AIE-based NTT implementations. The speedup is computed as $\frac{\text{prior throughput} \times \#\text{AIE}_{\text{prior}}}{\text{our throughput} \times \#\text{AIE}_{\text{ours}}}$ for throughput and analogously for latency, thus normalizing for the number of tiles used. We use our radix-2 scheme for comparison as the parameters exceed the radix-4 constraints.

We briefly explain why prior work achieves suboptimal performance. Ahmed and Arafin [AA24] use four parallel pipelines for large polynomial degrees but depend on the streaming interconnect with explicit engine-to-engine communication and data mapping stages; moreover, their use of Harvey's butterfly algorithm [Har14] introduces branch

statements ill-suited to SIMD execution. Bao et al. [BZLZ25] implement a radix-2 scheme with Montgomery reduction but similarly rely on the slow streaming interconnect, lose cycles to explicit communication stages, and do not apply partial reduction.

Comparison with other platforms. A direct comparison with GPU and FPGA implementations is challenging because these platforms differ fundamentally in clock frequency, power budget, and parallelism model. For context, Bao et al. [BZLZ25] report that their AIE-based design already outperforms a Tesla V100 GPU by 1.89× and an RTX 3090 GPU by 2.7× to 4.05× for the parameter set in Table 4. This suggests that our implementations offer superior performance compared with GPU implementations while operating at about 80× lower power (our power estimation yields a result of about 2.964 W for the `int16` implementation compared to over 250 W for a V100). FPGA-based NTT implementations [LBRS25, DBBS25, ZKA+25, KHF25] can offer comparable latency but typically at lower throughput and with much higher design effort. The key advantage of NPUs is the combination of software programmability, deterministic timing, and energy efficiency.

Table 4: Comparison with prior AIE-based work for a full NTT. TP = throughput. Speed-up is normalized per AIE tile: speed-up $= \frac{\text{prior metric} \times \#\text{AIE}_{\text{prior}}}{\text{our metric} \times \#\text{AIE}_{\text{ours}}}$.

		Full NTT			Speed-up	
	n, q	#AIE	Latency	TP	Latency	TP
SANTA [AA24][1]	2^{12}, 2^{14}	47	30,175 cc	2,011 cc	-	-
This work (R2)[2]	2^{12}, 2^{14}	12	11,424 cc	952 cc	10.3×	8.27×
Bao et al. [BZLZ25][3]	2^{11}, 2^{30}	16	19,000 cc	9,332 cc	-	-
This work (R2)[4]	2^{11}, 2^{30}	11	11,308 cc	1,028 cc	2.44×	13.2×

[1] Best design for $n = 2^{12}$: 47 AIE, latency 30,175 cc, estimated 15 pipeline stages; throughput $30{,}175/15 \approx 2{,}011$ cc.
[2] Pipeline of $\log_2(2^{12}) = 12$ tiles; latency $= 12 \times 952 = 11{,}424$ cc with 952 cc the simulation throughput for the `int16` radix-2 scheme.
[3] Best design for $n = 2^{11}$ with 4 PU (16 AIE) from [BZLZ25, Table 3]: latency 15.2 µs = 19,000 cc at 1.25 GHz; throughput = 1.25 GHz/133,954 = 9,332 cc.
[4] Pipeline of $\log_2(2^{11}) = 11$ tiles; latency $= 11 \times 1{,}028 = 11{,}308$ cc with 1,028 cc from Table 3 for `int32`.

Support for smaller ring dimensions. To amortize the overhead of kernel launches and loop preambles/postambles, we targeted the largest transform size allowed by the memory constraints. However, due to the native structure of the NTT, the implementation for a large n can efficiently compute smaller NTTs in parallel with no additional cost. This is particularly relevant for PQC applications such as ML-KEM and ML-DSA (both ring size 256). Table 5 provides the throughput and latency estimates.

Consider N independent NTTs of size n/N. This batching may be implemented in multiple ways. Firstly, we may assume the N input sets are interleaved in the input buffer. Using the regular $n = 4096$ implementation with perfect shuffles, the N output sets will be separated in the output buffer, each in bit-reversed order. Alternatively, we may perform a grouped shuffle on groups of N consecutive elements, which preserves the interleaving in the output. In this case, the same kernel can be used with different configuration parameters for the shuffle units, so the execution time does not increase. Alternative batching approaches are also possible, such as ones using separated instead of interleaved inputs, as long as the required permutation can be implemented efficiently.

Throughput and Latency. Our NTT designs improve over the literature in both latency and throughput. However, to fully utilize the pipeline throughput, multiple independent

Table 5: Estimated performance for PQC-relevant ring dimensions using the radix-2 `int16` scheme. Smaller transforms are computed in parallel within the $n = 4096$ kernel.

Ring size	**256**	**512**	**1024**	**2048**	**4096**
#Parallel NTTs	16	8	4	2	1
#Stages (#AIE tiles)	8	9	10	11	12
Throughput (cc)	952	952	952	952	952
Latency (cc)	7,616	8,568	9,520	10,472	11,424

NTT invocations must be in flight simultaneously, since our designs distribute the NTT layers across multiple AIE tiles in a pipeline. Real-world workloads easily provide this parallelism. In FHE, a single FHEW gate bootstrapping requires on the order of 10,000 NTTs [CGGI18], and the BFV/BGV/CKKS schemes require NTTs for multiple limbs at the same time. Furthermore, in PQC server applications, a server handling thousands of TLS handshakes generates thousands of concurrent ML-KEM or ML-DSA operations.

As another example, TFHE canonically makes use of ring degrees 2^9 up to 2^{12}, which align well with our supported parameters. It provides a natural use case for this throughput-oriented execution model as TFHE usually contains many bootstraps in parallel. FPT, a streaming TFHE accelerator [BDTV23], demonstrates this execution model by keeping 12 ciphertexts in flight, implying at least 24 NTTs are performed in parallel. This is more than enough to saturate our pipelines.

Full polynomial multiplication pipeline. A complete polynomial multiplication composes two forward NTT pipelines, an element-wise modular multiplication, and an inverse NTT pipeline. For radix-4 and radix-8, the product passes through the bit-reversal unit before entering the inverse-NTT pipeline, adding a tile but not reducing throughput.

Full reduction and normalization factor. All variants require a full-reduction pass when canonical outputs in $[0, q)$ are needed. This can be implemented with one or two additional pipeline stages without worsening the throughput.

The inverse NTT also requires multiplication by $n^{-1} \bmod q$. For all schemes except radix-2 `int32`, we can fold this factor into the final-layer twiddle factors without making that pipeline stage more expensive. For radix-2 `int32`, however, doing so would increase the cost of the final pipeline stage, since 32-bit Barrett multiplications are more expensive than 32-bit Barrett reductions. We then need either to add a pipeline stage for normalization or to merge normalization with the full reduction stages using full Barrett multiplications.

6 Bit-reversal unit

Since forward and inverse NTTs differ only in their twiddle factors (Equation 1, Equation 2), we can implement the inverse NTT for radix-4 and radix-8 using the forward implementation preceded by a bit-reversal permutation (all three schemes produce bit-reversed outputs). A naive bit-reversal moves one coefficient at a time, requiring about one cycle per coefficient. Using vector shuffle operations, we achieve up to 4× speedup.

Notation. We may characterize permutations by describing the mapping from the index of a coefficient in the input buffer to the index in the output buffer. Let the string of bits $\boldsymbol{a_{N-1}a_{N-2}\ldots a_0}$ be the binary representation of an index in the input buffer of size 2^N, with each bit $\boldsymbol{a_i} \in \{0, 1\}$. We introduce the notation $\boldsymbol{a_{[i<j]}}$ with $i < j$ as a shorthand for the *increasing* string of bits $\boldsymbol{a_i a_{i+1} \ldots a_{j-1} a_j}$, and $\boldsymbol{a_{[i>j]}}$ with $i > j$ as a shorthand for the *decreasing* string of bits $\boldsymbol{a_i a_{i-1} \ldots a_{j+1} a_j}$. After a perfect shuffle permutation is

performed, the index of the coefficient is converted from $\boldsymbol{a_{N-1}a_{N-2}\ldots a_0} = \boldsymbol{a_{[N-1>0]}}$ to $\boldsymbol{a_{[N-2>0]}a_{N-1}}$, a rotation of the bits to the left. We call the inverse permutation an *unshuffle* permutation, corresponding to a rotation of the bits to the right. A bit-reversal operation reverses the order of the bits, resulting in $\boldsymbol{a_{[0<N-1]}}$.

6.1 Vector-based bit-reversal permutation

A bit-reversal can be decomposed into a recursive series of perfect shuffle permutations [VL92]: if both halves of the buffer are already bit-reversed, one perfect shuffle on the full buffer completes the reversal. Applied recursively, this requires $N-1$ levels. Now consider a buffer of 2^{N-M} vectors of size 2^M each. Performing only the last M shuffle levels converts an index $\boldsymbol{a_{[N-1>0]}}$ to $\boldsymbol{a_{[N-1-M>0]}a_{[N-M<N-1]}}$; the remaining $N-M$ high-order bits can be reversed using vector moves in one loop (Figure 5a).

Following Lokhmotov and Mycroft [LM07], we further optimize by using *in-place* shuffle butterflies (Figure 5b), which implicitly unshuffle the vector ordering. For $N = 2M$, this eliminates the vector-move phase entirely. As an example, in the case of $M = 4$ and $N = 8$, the four levels of in-place butterflies are illustrated in Equation 8, where left-arrows indicate element shuffles and right-arrows indicate the implicit unshuffle of vector positions:

$$\begin{aligned}\boldsymbol{a_7a_6a_5\underleftarrow{a_4\overrightarrow{a_3}a_2a_1a_0}} &\Rightarrow \boldsymbol{a_7a_6\underleftarrow{a_5\overrightarrow{a_3a_2}a_1a_0a_4}} \Rightarrow \boldsymbol{a_7\underleftarrow{a_6\overrightarrow{a_2a_3a_1}a_0a_4a_5}} \\ &\Rightarrow \boldsymbol{\underleftarrow{a_7\overrightarrow{a_1a_2a_3a_0}a_4a_5a_6}} \Rightarrow \boldsymbol{a_0a_1a_2a_3a_4a_5a_6a_7}\end{aligned} \tag{8}$$

The case of $N = 2M$ is useful for ML-KEM with $N = 8$ as the `int16` vector length is $2^M = 16$ ($M = 4$) [Kyb24]. For $N > 2M$, we again only perform the *last* M levels, then only a bit-reversal of bits $N-M-1$ down to M must be performed, which only requires "local" vector moves *within* groups of 2^{N-2M} consecutive vectors. Note that if $N = 2M+1$, then a bit-reversal of $N-2M = 1$ bit is required, which is the identity permutation. When $N < 2M$, we may simply use vectors of size $2^{M'}$, where $M' = \lfloor N/2 \rfloor$, and apply one of the above cases.

In conclusion, we perform a bit-reversal with a complexity of $\mathcal{O}(M2^{N-M})$ instructions instead of $\mathcal{O}(2^N)$. This is a speedup of $4\times$ for $M = 4$ (16-lane `int16` vectors).

6.2 Implementation and results

For our implementation, we focus on the `int16` datatype with vector length $2^M = 16$ ($M = 4$) and polynomial sizes 2^N for $N = 8, 9, 10, 11, 12$. We require four perfect shuffle levels, followed by local vector moves for $N = 10, 11, 12$.

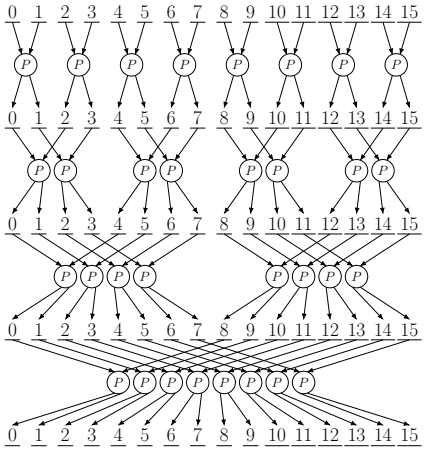
To increase arithmetic intensity, we perform two shuffle levels per loop by loading vectors in groups of four (e.g. yellow and blue groups in Figure 5b). Each iteration uses two `VSEL` instructions and stores four vectors in bit-reversed order. The critical resource is the store unit (MinII = 4), but 5 cycles are needed due to register constraints[7].

As four levels must be performed, two of these loops are required. One last loop implements the local vector moves in one cycle per vector. In total, $\frac{2\cdot 5/4+1}{16}2^N = 0.21875\cdot 2^N$ cycles are required in the II approximation. The simulation results are given in Table 6.

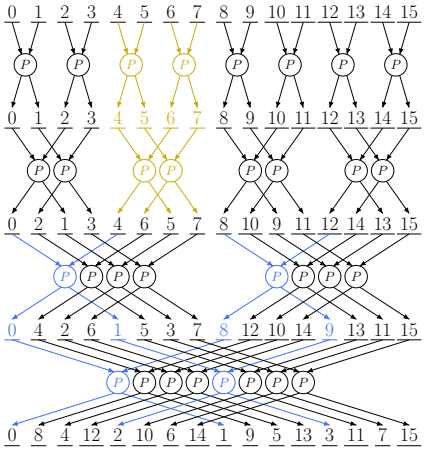
Table 6: The number of cycles required for the bit-reversal kernel in `int16`.

N	8	9	10	11	12
II Approximation	56	112	224	448	896
Simulation	100	156	268	492	940

[7] With a single memory bank, loading the next iteration's vectors takes four cycles, requiring one extra cycle to avoid overwriting live registers.



(a) Last four regular perfect shuffle levels. The elements within vectors are bit-reversed, but additional vector moves are required to complete the bit-reversal.



(b) Last four perfect shuffle levels with in-place butterfly operations. The output vectors are bit-reversed w.r.t. Figure (a) so no extra vector moves are required.

Figure 5: Comparison between (a) regular and (b) in-place interleaving operations for $N = M + 4$. The in-place operations result in a bit-reversal of the vectors. Furthermore, (b) illustrates the partitioning into groups every two steps, as explained in Subsection 6.2, by highlighting one group in the first two levels and in the last two levels.

7 Portability and Application to Second-Generation AIE

Many other NPUs share AIE-like features, including SIMD execution, high-bandwidth local SRAM, and dedicated permutation hardware. We therefore discuss portability from three perspectives: which optimizations are architecture-agnostic, which architectural features of AIE are beneficial for NTT acceleration and how the algorithms map to AIE2.

7.1 Portable optimization techniques

Several of our key optimizations are not tied to the AIE architecture and can be applied on any platform with sufficient SIMD width and memory bandwidth:

- **Loop fission for instruction pipelining.** Splitting complex loops into simpler ones to help the compiler scheduler applies to any VLIW or deeply pipelined SIMD processor, even if the compiler is more advanced.
- **Barrett reduction with optimized parameters.** Our choice of $K = \beta$ and nearest-integer rounding maximizes the headroom for lazy reduction, but requires a down-shift operator supporting nearest-integer rounding. Most accelerator cards support the down-shift but may lack a flexible rounding mode, in which case the sub-optimal rounding functions of Section 4 can be used. For architectures with only unsigned arithmetic support, the floor function becomes optimal.
- **Constant-geometry NTT.** Constant-geometry transforms [Pea68] make every NTT layer execute the same code with the same data access pattern, eliminating layer-dependent control flow and simplifying load balancing. The hardware must, however, support the required permutations efficiently, ideally at no extra cost.
- **Lazy and partial reduction.** Computing with partially reduced values and only reducing when overflow would otherwise occur (lazy reduction) are purely algorithmic

optimizations that apply identically on any platform.

7.2 Architectural features beneficial for NTT acceleration

We identify three AIE features that are particularly valuable for cryptographic workloads:

- **Arithmetic right shift with configurable rounding.** A right shift is essential for efficient Barrett reduction and the configurable rounding allows for optimal choice of the rounding mode. Furthermore, since this operation is implemented in a separate data path, we can perform right shifts and multiplications in parallel.
- **In-data path permutations.** This enables free data mapping for constant-geometry transforms, avoiding separate permutation passes.
- **High-bandwidth bank-partitioned memory.** The direct 256-bit/cycle memory access to neighboring tiles enables pipelined NTT layers without relying on a slow streaming interconnect. It is essential for effective loop fission, which produces many small loops that must be fed with data at high bandwidth.

7.3 Second-generation AIE (AIE-ML/AIE2)

As a concrete example of porting our designs to a different architecture, we provide a short summary of how the algorithms map to the second-generation AIE (AIE2). The most important difference between AIE2 and AIE is a doubled multiply throughput per cycle at the same memory bandwidth, which shifts the bottleneck from compute to memory for the `int16` radix-2 and radix-4 kernels. The new architecture also has 24 vector registers instead of 8, and it has independent data paths for permutation, addition and multiplication.

We believe all optimizations in this work should carry over to this newer architecture. For `int16`, the radix-8 scheme may now become most attractive because its high arithmetic intensity better matches the increased compute-to-memory ratio.

8 Overview of optimization contributions

This section gives an overview of the most important optimizations for the radix-2 and radix-4 NTT schemes and their Barrett primitives, ranked in decreasing order of importance according to their contribution compared to prior art on the AI Engines for NTTs.

1. **Loop fission:** manual loop fission, software pipelining, loop unrolling, register pinning, and overlapping loop preambles and postambles.
2. **Memory, data layout, and communication:** conflict-free memory-bank placement, shared-memory pipelines across tiles, constant-geometry transforms, and permutations fused into compute operations.
3. **Barrett arithmetic:** precomputed twiddle factors and Barrett multipliers, the choice $K = \beta$ and nearest-integer rounding, and fused multiply-subtract operations.
4. **Reduction strategy:** partial modular reduction and lazy reduction of butterfly additions and subtractions.
5. **Radix-4 structure:** fewer layers and reductions, butterfly reformulation with $\{-1, 0, 1\}$ coefficients, and split matrix computations to manage register pressure.

9 Conclusion and Future work

We presented a general methodology for accelerating the NTT on NPUs, using AMD's first-generation AIE as a representative case study. Our main finding is that manual loop fission, a well-known compiler optimization, is essential for achieving near-optimal performance on VLIW/SIMD NPU architectures where current compilers fail to apply it effectively. Combined with careful Barrett reduction parameter selection, constant-geometry transforms, and in-data path permutations, this approach yields large speedups over the prior state-of-the-art. More broadly, our work shows that NPUs designed for AI workloads are competitive platforms for lattice-based cryptography.

Our work also leaves several extensions open. First, for specific moduli, Barrett reduction may be replaced by a more specialized technique such as K-Red [BNAMK21, LN16]. Second, when q is small enough (as for ML-KEM with $q = 3329$ [Kyb24]), additional lazy reduction can eliminate all intermediate Barrett reductions in the radix-2 scheme, since the coefficients never outgrow signed 16-bit integers across the transform, reducing the cost from $0.1875n$ to $0.125n$ per layer. Larger polynomial sizes can be supported through multi-tile parallelism, and negative-wrapped transforms [LZ22] can be incorporated by folding pre- and post-scaling twist factors into the existing framework. Finally, integrating the NTT into complete PQC or FHE schemes would demonstrate end-to-end performance.

Acknowledgements. This work was supported in part by the Horizon 2020 ERC Advanced Grant (101020005 Belfort) and the CyberSecurity Research Program Flanders with reference number VOEWICS02. Henri Vermeersch is funded by Belfort Labs and Flanders Innovation & Entrepreneurship (VLAIO) through a Baekeland PhD mandate (project number HBC.2025.0934). Xander Pottier is funded by FWO (Research Foundation – Flanders) as Strategic Basic (SB) PhD fellow (project number 1S93126N). Thomas de Ruijter is funded in part through the Google PhD Fellowship program.

References

[AA24] Fahim Ahmed and Md Tanvir Arafin. SANTA: A Spatial Accelerator Design for Efficient Number Theoretic Transform (NTT) on Heterogeneous System-on-Chips. In *Proceedings of the 2024 Workshop on Attacks and Solutions in Hardware Security*, ASHES '24, page 2–10, New York, NY, USA, 2024. Association for Computing Machinery.

[AK01] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2001.

[AMDa] AMD (Advanced Micro Devices). AI Engine API User Guide (UG1529). https://download.amd.com/docnav/aiengine/xilinx2024_1/aiengine_api/aie_api/doc/index.html. Accessed: 2025-11-10.

[AMDb] AMD (Advanced Micro Devices). AI Engine Intrinsics User Guide (UG1078). https://www.xilinx.com/htmldocs/xilinx2024_1/aiengine_intrinsics/intrinsics/index.html. Accessed: 2025-11-10.

[AMDc] AMD (Advanced Micro Devices). AI Engine Kernel And Graph Programming Guide (UG1079). https://docs.amd.com/r/2024.1-English/ug1079-ai-engine-kernel-coding. Accessed: 2025-11-10.

[AMDd] AMD (Advanced Micro Devices). Versal Adaptive SoC AI Engine Architecture Manual (AM009). https://docs.amd.com/r/en-US/am009-versal-ai-engine. Accessed: 2025-11-10.

[ARS+20] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E.R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. Think fast: A tensor streaming processor (tsp) for accelerating deep learning workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 145–158, 2020.

[Bar87] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology — CRYPTO '86*, pages 311–323. Springer, 1987.

[BDL25] Nick Brown, Jake Davies, and Felix LeClair. Exploring fast fourier transforms on the tenstorrent wormhole, 2025.

[BDTV23] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Furkan Turan, and Ingrid Verbauwhede. FPT: A fixed-point accelerator for torus fully homomorphic encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 741–755. ACM, 2023.

[BHK+21] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. NeonNTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. Cryptology ePrint Archive, Report 2021/986, 2021.

[BNAMK21] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. High-speed NTT-based polynomial multiplication accelerator for CRYSTALS-Kyber post-quantum cryptography. Cryptology ePrint Archive, Report 2021/563, 2021.

[BZLZ25] Zhenshan Bao, Tianhao Zang, Yiqi Liu, and Wenbo Zhang. Efficient Number Theoretic Transform Accelerator on the Versal Platform Powered by the AI Engine. *Future Generation Computer Systems*, 166:107728, 2025.

[CGGI18] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Report 2018/421, 2018.

[CKKS16] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Report 2016/421, 2016.

[CYY+22] Xiangren Chen, Bohan Yang, Shouyi Yin, Shaojun Wei, and Leibo Liu. CFNTT: Scalable radix-2/4 NTT multiplication architecture with an efficient conflict-free memory mapping scheme. *IACR TCHES*, 2022(1):94–126, 2022.

[DBBS25] Dixit Dutt Bohra, Dip Sankar Banerjee, and Somitra Sanadhya. Sparse-aware ntt: Accelerating lattice-based cryptography on fpgas. In *2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, volume 1, pages 1–6, 2025.

[DCH+24] Sultan Durrani, Muhammad Saad Chughtai, Mert Hidayetoglu, Rashid Tahir, Abdul Dakkak, Lawrence Rauchwerger, Fareed Zaffar, and Wen-mei Hwu. Accelerating fourier and number theoretic transforms using tensor cores and warp shuffles. In *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques*, PACT '21, page 345–355. IEEE Press, 2024.

[Dil24] Module-lattice-based digital signature standard. Technical Report Federal Information Processing Standards Publications (FIPS) 204, National Institute of Standards and Technology, U.S. Department of Commerce, Washington, D.C., 2024.

[FZZ+25] Guang Fan, Mingzhe Zhang, Fangyu Zheng, Shengyu Fan, Tian Zhou, Xianglong Deng, Wenxu Tang, Liang Kong, Yixuan Song, and Shoumeng Yan. Warpdrive: Gpu-based fully homomorphic encryption acceleration leveraging tensor and cuda cores. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1187–1200, 2025.

[Har14] David Harvey. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation*, 60:113–119, January 2014.

[JYP+17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit, 2017.

[JZG+25] Zhuoran Ji, Jianyu Zhao, Peimin Gao, Xiangkai Yin, and Lei Ju. Accelerating number theoretic transform with multi-gpu systems for efficient zero knowledge proof. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 1–14, New York, NY, USA, 2025. Association for Computing Machinery.

[KHF25] Dilshan Kumarathunga, Qilin Hu, and Zhenman Fang. Autontt: Automatic architecture design and exploration for number theoretic transform acceleration on fpgas. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–9, 2025.

[KN11] Arun Kejariwal and Alexandru Nicolau. Modulo Scheduling and Loop Pipelining. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1158–1173. Springer US, Boston, MA, 2011.

[Kyb24] Module-lattice-based key-encapsulation mechanism standard. Technical Report Federal Information Processing Standards Publications (FIPS) 203, National Institute of Standards and Technology, U.S. Department of Commerce, Washington, D.C., 2024.

[LBRS25] Chetan Harsha Lekkala, Sitadevi Bharatula, B. Naresh Kumar Reddy, and K Sarangam. Optimized fpga architectures for high-speed ntt computation. In *2025 Fifth International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, pages 1–5, 2025.

[LM07] Anton Lokhmotov and Alan Mycroft. Optimal bit-reversal using vector permutations. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, page 198–199, New York, NY, USA, 2007. Association for Computing Machinery.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. Cryptology ePrint Archive, Report 2016/504, 2016.

[LNP22] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. Lattice-based zero-knowledge proofs and applications: Shorter, simpler, and more general. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 71–101, Cham, 2022. Springer Nature Switzerland.

[LZ22] Zhichuang Liang and Yunlei Zhao. Number theoretic transform and its applications in lattice-based cryptosystems: A survey. *CoRR*, abs/2211.13546, 2022.

[NKNT24] Ai Nozaki, Takuya Kojima, Hiroshi Nakamura, and Hideki Takase. A study on number theoretic transform acceleration on amd ai engine. In *2024 IEEE 17th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 325–331, 2024.

[OJS25] Alisah Ozcan, Arsalan Javeed, and Erkay Savas. High-performance number theoretic transform on gpu through radix2-ct and 4-step algorithms. *IEEE Access*, 13:87862–87883, 2025.

[Pea68] Marshall C. Pease. An adaptation of the fast fourier transform for parallel processing. *J. ACM*, 15(2):252–264, April 1968.

[Rau95] Bantwal Ramakrishna Rau. Iterative modulo scheduling. *Hewlett-Packard Laboratories*, November 1995.

[RL24] John Reuben and Steve Leibson. A survey of neural processing units: From cpus and gpus to specialized ai accelerators. In *IEEE Open Journal of the Computer Society*, volume 5, pages 78–94, 2024.

[SIF+25] Cristina Silvano, Daniele Ielmini, Fabrizio Ferrandi, Leandro Fiorin, Serena Curzel, Luca Benini, Francesco Conti, Angelo Garofalo, Cristian Zambelli, Enrico Calore, Sebastiano Schifano, Maurizio Palesi, Giuseppe Ascia, Davide Patti, Nicola Petra, Davide De Caro, Luciano Lavagno, Teodoro Urso, Valeria Cardellini, Gian Carlo Cardarilli, Robert Birke, and Stefania Perri. A survey on deep learning hardware accelerators for heterogeneous hpc platforms. *ACM Comput. Surv.*, 57(11), June 2025.

[SKP24] Mahesh Subramony, David Kramer, and Indrani Paul. AMD Ryzen 7040 Series. *IEEE Micro*, 44(3):18–24, 2024.

[VL92] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform.* Frontiers in Applied Mathematics. Society for Industrial and Applied Mathematics, 1992.

[Xil18] Xilinx Inc. Versal, The First Adaptive Compute Acceleration Platform (ACAP). White Paper WP505, Xilinx, 2018. Version 1.0, October 2, 2018.

[ZKA+25] Moonis Amir Zaman, Muhammad Bilal Khan, Waqar Ahmad, Ayesha Khalid, and Máire O'Neill. A lightweight and hardware-efficient ntt fpga accelerator for fhe applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, pages 1–1, 2025.