

StreamNTT: A High-Throughput, HLS-Based Streaming NTT Accelerator for HBM-Equipped FPGAs

Young-kyu Choi, Hyunwoo Park
Department of Computer Engineering
Inha University

Incheon, South Korea
ykc@inha.ac.kr, hwpark2000@gmail.com

Wei He, Sunwoong Kim

Department of Electrical and Microelectronic Engineering
Rochester Institute of Technology

Rochester, NY, USA
wh9297@rit.edu, sskeme@rit.edu

Abstract—Lattice-based post-quantum cryptography (PQC) relies extensively on the number-theoretic transform (NTT), and large-scale deployments require throughput-optimized accelerators capable of computing thousands of NTTs per session. To address this need, we present StreamNTT, a high-throughput NTT accelerator developed using high-level synthesis (HLS) for field-programmable gate arrays (FPGAs) equipped with high bandwidth memory (HBM). StreamNTT effectively leverages the parallelism inherent in the NTT by overcoming several obstacles that limit the scalability of streaming dataflow designs. Specifically, we introduce HLS-friendly butterfly units that integrate butterfly computation and reorder buffering into a single pipelined loop. In addition, we propose a butterfly merging strategy that reduces first-in, first-out channels and buffers. Finally, we present a placement-aware instance-level parallelism scheme for FPGA platforms with multiple HBM channels and super logic regions. On the Alveo U280 platform, the proposed optimizations improve throughput by a factor of 7.2 and achieve a speedup of more than 2.7 times compared to state-of-the-art FPGA-based NTT accelerators. As an open-source design, StreamNTT establishes a scalable foundation for advancing high-performance PQC acceleration.

Index Terms—number theoretic transform, post-quantum cryptography, field-programmable gate array, high-level synthesis, dataflow architecture, accelerator design.

I. INTRODUCTION

Modern post-quantum cryptography (PQC) schemes based on lattice algorithms rely heavily on polynomial multiplication accelerated by the number-theoretic transform (NTT). As PQC is deployed at scale, many applications may require processing hundreds to thousands of NTTs per session. For example, PQC-based transport layer security (TLS) handshakes must handle massive client fan-in within cloud data centers [1]. Similar demands arise when verifying large batches of PQC digital signatures during fleet-wide software updates or communication with numerous edge devices [2]. Service providers such as Google and Amazon, along with encrypted messaging platforms, are also expected to perform PQC-based authentication at high volumes [3]. Anticipating even greater computational demands from future PQC schemes, NVIDIA recently introduced cuPQC, a library featuring optimized NTT kernels [4].

Addressing these performance challenges requires high-throughput NTT acceleration. Field-programmable gate arrays

(FPGAs) are a strong fit for this workload as they support deep pipelining, large-scale parallelism, and customizable memory architectures. FPGAs also offer abundant computational resources for the intensive integer operations found in modular arithmetic. Furthermore, FPGAs enable rapid deployment in cloud or edge environments without ASIC lead times.

To design FPGA-based NTT accelerators productively, we adopt a high-level synthesis (HLS) tool. HLS enables designers to describe the accelerator architecture in C/C++ or OpenCL, significantly reducing the development effort compared to traditional RTL design. For high-throughput HLS designs, streaming dataflow architecture is widely used [5]. In this architecture, a large number of NTT butterfly computation modules operate concurrently and exchange data through first-in, first-out (FIFO) channels.

While many prior HLS-based NTT works focus on tuning compiler directives to improve performance [6]–[8], scaling the streaming dataflow architecture to support massive parallelism introduces several challenges.

First, the NTT exhibits varying coefficient access strides between stages, while Vitis HLS restricts inter-module communication to non-addressable FIFO channels [9]. Prior HLS-based NTT designs [7], [8] insert a reorder buffer within a dataflow module to support arbitrary access strides. However, this approach incurs serialized execution of buffer read, butterfly compute, and buffer write, since merging these operations into a single pipelined HLS loop is difficult.

Second, selecting the granularity of dataflow modules is a key design choice. A natural approach to maximize parallelism is to instantiate one module per butterfly stage [10], [11], since each module operates independently. But this incurs non-trivial resource overhead from the many inter-stage FIFOs. For example, FIFO streams between modules account for 23% of lookup table (LUT) usage for a baseline NTT design with a polynomial degree of 1024 and modulus 3221225473. A more efficient module mapping is necessary to lower this overhead and permit the instantiation of more butterfly units (BUs) on the FPGA.

Third, scaling to multiple NTT instances on recent FPGAs equipped with high bandwidth memory (HBM) introduces placement and timing challenges. When a single NTT instance is connected to multiple HBM pseudo-channels as in [12], placement and timing closure become difficult due to the need

This work is supported in part by Inha University Research Grant, National Research Foundation (NRF) Grant funded by Korea Ministry of Science and ICT (MSIT) (2022R1F1A1074521), and the National Science Foundation (under Grant No. 2347253).

for large multiplexers and complex data arbiters to scatter/gather coefficients. In addition, the resulting large dataflow modules often exceed super logic region (SLR) boundaries, further complicating placement.

To address these challenges, we present StreamNTT, an HLS-based, high-throughput, streaming NTT accelerator for HBM-equipped FPGAs. StreamNTT introduces an HLS-friendly integrated circular BU (ICBU) that unifies butterfly computation and circular buffering within a single pipelined loop, eliminating the need for addressable memory between dataflow modules. We further propose a butterfly merging strategy that reduces FIFO and buffer resource consumption while preserving module concurrency. Finally, we present a placement-aware instance-level parallelism scheme that partitions multiple NTT pipelines across an FPGA platform with multiple HBM channels and SLRs to improve scalability and timing.

Across representative PQC settings, StreamNTT effectively leverages the inherent parallelism of NTT and, to the best of our knowledge, achieves the highest throughput reported in the literature. To facilitate adoption, StreamNTT provides parameterized HLS templates and scripts for automated FPGA bitstream generation. StreamNTT is fully open source and is publicly available at <https://github.com/applesforme/StreamNTT>.

II. BACKGROUND AND RELATED WORK

A. Number Theoretic Transform (NTT)

The input of the NTT is a polynomial $a(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$, where n denotes the transform size (polynomial degree plus one). The NTT maps this polynomial to a vector $A = (A_0, \dots, A_{n-1})$, with

$$A_k = \sum_{j=0}^{n-1} a_j \psi^j \omega^{jk} \pmod{q}, \quad \text{for } k = 0, 1, \dots, n-1,$$

where the twiddle factor ω is a primitive n -th root of unity modulo q , and the adjustment factor ψ is a primitive $2n$ -th root of unity such that $\psi^2 \equiv \omega \pmod{q}$. A prime number is chosen for the modulus q .

The NTT proceeds through $\log_2 n$ stages of modular butterfly operations. At each stage s , the butterfly combines two coefficients $a_j^{(s)}$ and $a_{j+\text{str}(s)}^{(s)}$ to compute

$$\begin{aligned} a_j^{(s+1)} &= \left(a_j^{(s)} + \psi^{\delta(s)} \cdot \omega^{k \cdot \text{str}(s)} a_{j+\text{str}(s)}^{(s)} \right) \pmod{q}, \\ a_{j+\text{str}(s)}^{(s+1)} &= \left(a_j^{(s)} - \psi^{\delta(s)} \cdot \omega^{k \cdot \text{str}(s)} a_{j+\text{str}(s)}^{(s)} \right) \pmod{q}. \end{aligned} \quad (1)$$

The stride $\text{str}(s) = 2^{\log_2 n - s - 1}$ is the distance between paired coefficients and decreases as the transform progresses. The indicator $\delta(s) \in \{0, 1\}$ specifies whether ψ is applied at stage s , as required by the negacyclic formulation [13]. Leveraging the periodicity and symmetry of ψ and ω , the algorithm achieves $O(n \log n)$ complexity.

The NTT is the dominant computational kernel in lattice-based PQC schemes [2], [13], [14]. Core cryptographic operations such as encryption, decryption, key exchange, or digital

```

1 str = n >> (s + 1);
2 for (k = 0; k < n; k += 2*str) {
3   for (j = 0; j < str; j++) {
4     int idx = k + j;
5     tw = tw_rom[s][j]; // twiddle ROM lookup
6     t = reduce(tw * a[idx+str]); // mult reduce
7
8     // a[idx+str] = (a[idx] - t) mod q
9     x = a[idx] + q - t;
10    a[idx+str] = x - ((x >= q) ? q : 0);
11
12    // a[idx] = (a[idx] + t) mod q
13    y = a[idx] + t;
14    a[idx] = y - ((y >= q) ? q : 0);
15  }
16 }

```

Fig. 1. Loop structure for butterfly stage s

signature verification require multiple NTT computations. Consequently, constructing a high-throughput NTT design becomes a crucial factor for large-scale PQC deployment scenarios.

B. Related Work on High-Throughput NTT FPGA Designs

Among the many FPGA NTT papers in the literature, we focus on works that prioritize throughput optimization. PipeNTT [10] introduces a highly efficient pipelined architecture that leverages a block RAM (BRAM)-based reordering unit and a high-throughput BU with low hardware cost. Proteus [11] provides a design-time-flexible, parametric hardware generator for pipelined radix-2 single-path delay feedback and multi-path delay commutator architectures, offering several resource-efficient optimizations. NTTGen [15], a follow-up to [16], proposes a framework that automatically generates a range of NTT accelerators by exploiting data, pipeline, and batch parallelism, while using a streaming permutation network to avoid costly crossbars. The work in [2] discusses a batch-processing acceleration framework that mitigates CPU-FPGA communication overhead. The work in [17] achieves scalability for large-degree fully homomorphic encryption (FHE) workloads via a conflict-free memory mapping algorithm and balanced HBM-channel scheduling. AutoNTT [12] presents an open-source automatic design-space exploration tool that systematically explores iterative, dataflow, and hybrid architectures over a wide range of parameters and reduction methods. However, none of these works comprehensively exploit the full spectrum of NTT parallelism to maximize throughput on modern multi-SLR, multi-HBM FPGA platforms to the extent achieved by StreamNTT. A detailed comparison will be provided in subsequent sections.

III. HLS-BASED NTT PARALLELIZATION AND STREAMNTT

We implement the NTT using the widely adopted iterative Cooley-Tukey radix-2 algorithm [18]. The loop structure for stage s is shown in Fig. 1—the outer loop indexes over stride windows, and the inner loop applies the butterfly computation to coefficient pairs. The constants $\psi^{\delta(s)} \cdot \omega^{k \cdot \text{str}(s)}$ in Eq. 1 are precomputed at compile time and stored in a twiddle ROM (denoted as `tw_rom` and utilize BRAMs). For modular reduction, we follow the method in [19]: the multiplication

TABLE I
TYPES AND DEGREES OF PARALLELISM IN STREAMNTT

Parallelism Type	Degree
Inter-stage parallelism	$\log_2 n$
Intra-stage parallelism	NBU
Butterfly pipeline parallelism	HLS-determined
Multi-instance parallelism	NNI

TABLE II
STREAMNTT DESIGN PARAMETERS

Parameter	Description
n	Transform size (polynomial degree + 1)
q	Prime modulus
w_{coeff}	Bitwidth of coefficients ($\lceil \log q \rceil$)
NCH	Number of HBM channels
EBW	Effective bandwidth per HBM channel
NBU	Number of BUs per stage
NNI	Number of NTT instances

result (wider range) uses a dedicated modular reduction, while additions/subtractions (narrower range) use lightweight conditional correction by q .

To accelerate NTT on FPGAs, we exploit four types of parallelism (Table I). First, *inter-stage parallelism* is realized by coarse-grained pipelining across all stages: each BU begins computation upon receiving its input from the preceding stage (Eq. 1) and sends its results to the next stage. For an NTT of size n , the degree is $\log_2 n$ stages. Second, *intra-stage parallelism* is achieved by instantiating NBU BUs per stage. Within each stage, butterfly operations are cyclically assigned to BUs. Specifically, in the loop indexed by j in Fig. 1, the i -th BU at stage s , denoted BU_s^i , processes indices $j = i, i + NBU, i + 2NBU, \dots$ (e.g., BU_s^1 handles $j = 1, NBU + 1, 2NBU + 1, \dots$). Third, *butterfly pipeline parallelism* exploits fine-grained pipelining within each BU. The modular multiplication, addition, subtraction, and reduction steps in Eq. 1 are pipelined with an initiation interval (II) of one. The pipeline depth is automatically determined by the HLS tool. Finally, *multi-instance parallelism* is supported by replicating the full NTT pipeline to process multiple independent input polynomials in parallel. We instantiate NNI complete NTT pipelines, each operating on a separate polynomial. The total number of deployed butterflies is $TBU = \log_2 n \cdot NBU \cdot NNI$.

To support these dimensions, we introduce StreamNTT, a high-throughput, streaming NTT accelerator developed with AMD/Xilinx Vitis HLS [9], TAPA [5], and RapidStream [20]. StreamNTT adopts a streaming dataflow architecture [5], where computational modules communicate via FIFO streams. Each module executes its own finite-state machine (FSM) independently, allowing a massive number of NTT butterflies to operate in parallel. The streaming dataflow architecture is programmed using the TAPA-RapidStream APIs [20].

To feed sufficient coefficients, StreamNTT targets HBM-equipped FPGA platforms. We primarily evaluate on the Alveo U280 platform [21], but the design can be easily adapted for other HBM-FPGA devices. StreamNTT incorporates a flexible, parameterized hardware template to automatically generate high-throughput designs—key design parameters are summarized in Table II.

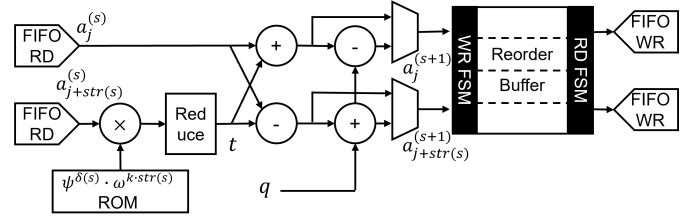


Fig. 2. BU dataflow module for stage s

```

1  str = n >> (s + 1);
2  for(;;){ // free-running infinite loop
3  #pragma HLS pipeline II=1
4  if( !aj_in/ajs_in.empty() && !rbuf[] .full() ){
5  a_j = aj_in.read(); // coefficients from
6  a_js = a_js_in.read(); // prev stage
7
8  ... // update tw_idx
9  tw = tw_rom[tw_idx];
10 t = reduce(tw * a_js); // mult reduction
11 x = a_j + q - t; y = a_j + t;
12
13 // write to coefficient reorder buffer
14 rbuf[wptr + str] = x - ((x >= q) ? q : 0);
15 rbuf[wptr] = y - ((y >= q) ? q : 0);
16
17 wptr++; // advance write pointer
18 } // write phase
19
20 if( !aj_out/ajs_out.full() && !rbuf[] .empty() ){
21 // read coefs from buffer and send to next
22 // stage at half stride
23 aj_out.write( rbuf[rptr] );
24 ajs_out.write( rbuf[rptr + str/2] );
25
26 rptr++; // advance read pointer
27 } // read phase

```

Fig. 3. Proposed ICBU pseudocode

While the streaming dataflow architecture enables massive parallelism, several challenges arise when scaling for high throughput. As discussed in the introduction, the lack of addressable inter-stage memory in HLS dataflow complicates reorder-buffer design. A large number of FIFO streams between dataflow modules also incur notable LUT overhead. Finally, scaling across multiple HBM channels and SLRs introduces placement and timing-closure challenges. The solution to these challenges will be explained in Section IV.

IV. HIGH-THROUGHPUT HLS NTT ARCHITECTURE

A. HLS-Friendly Integrated Circular Butterfly Unit

We adapt the butterfly computation in Fig. 1 to a dataflow style (Fig. 2). Because the butterfly stride $\text{str}(s) = 2^{\log_2 n - s - 1}$ halves at each stage, we employ a coefficient reorder buffer [6], [7], [10], [11] to reorganize the coefficient data access pattern. The coefficients produced by the butterfly are written into the buffer when it is not full; in parallel, available data is read out and forwarded to the next stage at a different stride. This creates a logically decoupled pipeline in which read and write operate as independent FSMs. For stage s , each butterfly maintains a buffer of size $2 \cdot \text{str}(s) / NBU$, implemented as randomly

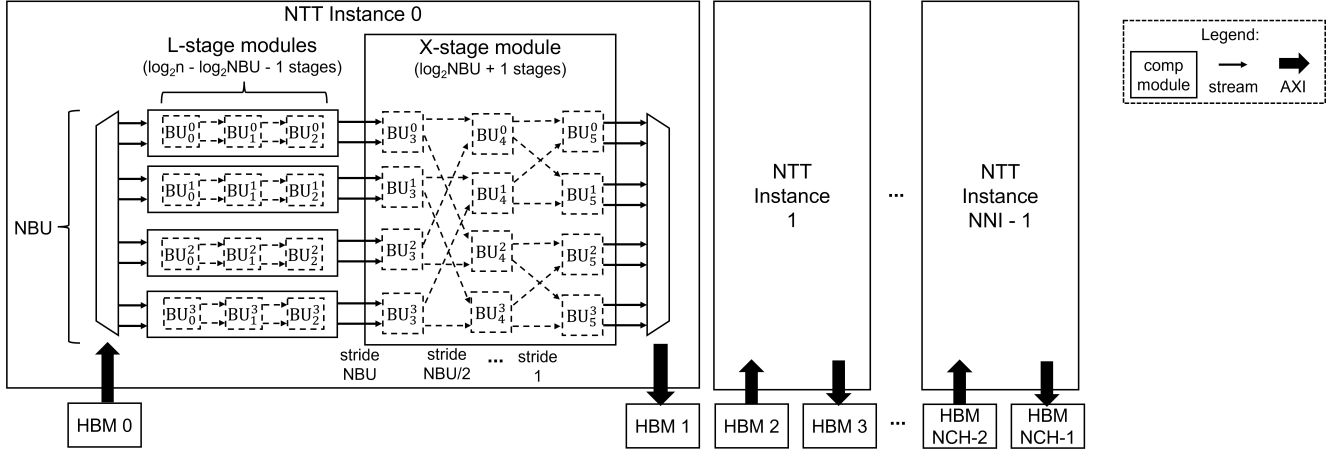


Fig. 4. Proposed streaming NTT architecture for HBM-FPGA platform (example shown for $NBU = 4$ and $\log_2 n = 6$)

addressable on-chip memory (e.g., LUTRAM) because writer and reader traverse the memory with different strides.

While such decoupled buffering is naturally expressible in Verilog RTL (e.g., with two parallel `always` blocks), implementation in HLS is considerably more challenging. Vitis HLS restricts inter-module communication among dataflow modules to non-addressable FIFOs [9], thereby preventing clean separation of the buffer reader and writer into distinct modules. Consequently, previous HLS works such as [7], [8] structure butterfly compute, buffer read, and buffer write as separate pipelined loops within one function, effectively serializing them and degrading performance.

To address this, we propose an HLS-friendly integrated circular butterfly unit (ICBU) that merges butterfly compute and circular-buffer management into a single pipelined loop (Fig. 3). During the write phase (lines 4–18), the unit consumes coefficients from the preceding stage, performs the butterfly, and stores results in the buffer. In parallel, the read phase (lines 20–26) retrieves coefficients from the buffer and forwards them to the next stage, accessing the buffer at half the stride of the write phase (lines 14 and 23). Both phases are seamlessly integrated into a single pipelined HLS loop. For efficient memory utilization, the coefficient reorder buffer is implemented as a circular structure. The buffer is partitioned by a factor of four (omitted in code for brevity) to enable simultaneous access [19], [22]. The kernel runs continuously in a flattened infinite loop (`for(;;)`, line 2), eliminating the need for centralized module start/stop control signals [23]. Overall, the ICBU delivers a high-throughput, dataflow-compatible butterfly with integrated buffer management suitable for HLS.

B. Reducing Resource Consumption with Butterfly Merging

As explained in the introduction, assigning each butterfly to its own HLS dataflow module incurs LUT overhead due to inter-stage FIFO streams. Furthermore, this prevents sharing of the twiddle ROM (BRAMs provide two read ports) among BUs. To mitigate this inefficiency, we combine multiple butterflies into a single dataflow module. But this introduces two risks. First, because each dataflow module typically executes a single

pipelined loop, excessive merging can reduce concurrency. Second, overly large modules may exceed SLR boundaries, complicating placement and timing closure.

To achieve a balance between resource savings and performance, we selectively merge BUs when their concurrency is preserved and the expected FIFO reduction is substantial. To guide this process, we classify NTT stages into two types based on their communication patterns: line-mapped stages (L-stages) and cross-mapped stages (X-stages).

L-stages are the early stages where the butterfly stride $\text{str}(s)$ is larger than NBU . In these stages, each BU_s^i only feeds the BU with the same index i at the next stage (BU_{s+1}^i), forming line-like pipelines (Fig. 4). Accordingly, butterflies along each line are merged into a single dataflow module. Different lines are not merged together since no FIFOs exist between them. To preserve concurrent execution within each merged module, we unroll the ICBU from Section IV-A. Each unrolled instance maintains its own circular buffer and performs independent read/write access, while all instances execute in parallel within one pipelined loop.

X-stages appear in the later stages, where $\text{str}(s)$ becomes smaller than NBU (e.g., $\text{str}(s) = NBU/2, NBU/4, \dots$). Here, each BU's outputs are distributed to multiple BUs in the next stage (Fig. 4). Thus, we merge all NBU butterflies across the X-stages—combining $NBU \cdot (\log_2 NBU + 1)$ butterflies into a single dataflow module. Since coefficients are directly consumed by the next-stage BU, coefficient reorder buffers or ICBU are unnecessary in X-stages. The module is a straightforward deep computation pipeline of butterflies.

As shown later in Section V, the proposed butterfly merging scheme achieves a notable reduction in FIFO LUTs. Also, BRAM consumption is lowered because the optimization enables two BUs to share a single twiddle ROM.

C. Placement-Aware Instance-Level Parallelism

Prior HLS-based streaming NTT designs, such as [12], adopt an architecture where a *single* NTT instance exploits large intra- and inter-stage parallelism. While such an architecture may employ a total number of BUs comparable to our approach, we observe increasing placement and timing challenges

TABLE III
IMPACT OF PROPOSED OPTIMIZATIONS FOR $n = 1024, q = 3221225473$

Optimizations	LUT	FF	DSP	BRAM	NCH	NBU	NNI	TBU	Freq (MHz)	Thrpt (MPoly/s)	Speedup
Baseline	371K	346K	1,280	444	16	32	1	320	296	4.50	1.0×
+ICBU	357K	358K	1,280	444	16	32	1	320	283	12.9	2.9×
+ICBU+Merge	307K	270K	1,280	268	16	32	1	320	291	12.6	2.8×
+ICBU+Merge+ILP	648K	605K	2,560	536	32	4	16	640	306	32.4	7.2×

TABLE IV
FINAL NTT THROUGHPUT, EFFECTIVE HBM BANDWIDTH, AND ENERGY CONSUMPTION FOR VARIOUS PQC CONFIGURATIONS

Configurations (n, q)	LUT	FF	DSP	BRAM	NCH	NBU	NNI	TBU	Thrpt (MPoly/s)	BW (GB/s)	E (uJ/Poly)
(256, 7681)	540K	522K	2,048	376	32	16	16	2,048	195	200	0.28
(256, 8380417)	447K	632K	2,048	440	32	8	16	1,024	128	262	0.43
(1024, 12289)	411K	472K	1,280	568	32	8	16	1,024	62.5	256	0.94
(1024, 3221225473)	648K	605K	2,560	536	32	4	16	640	32.4	265	2.1

as resource utilization approaches device limits. This arises for two reasons. First, connecting a single NTT instance to multiple HBM channels introduces long datapaths and requires large multiplexers for aggregating data traffic. Second, X-stage merging (Section IV-B) can make a single NTT instance too large to fit within one SLR.

We instead adopt a placement-aware strategy based on instance-level parallelism. Rather than enlarging a single NTT instance, we deploy multiple smaller NTT pipelines, each operating independently and interfacing with a dedicated pair of HBM channels. As illustrated in Fig. 4, we allocate HBM channels in consecutive pairs: HBM #0 and #1 to instance 0, HBM #2 and #3 to instance 1, and so forth. This mapping encourages the Vitis tool to localize the placement of an NTT instance. In addition, the data arbitration logic is simplified, since a single instance no longer needs to aggregate traffic across many HBM channels.

To ensure efficient utilization of HBM bandwidth, we size the intra-stage parallelism (NBU) of each instance to reflect the effective bandwidth (EBW) of its assigned HBM channel. Let w_{coeff} denote the coefficient bitwidth (set to $\lceil \log q \rceil$) and f_{clk} the kernel clock frequency. Then,

$$NBU = EBW / (w_{\text{coeff}} \cdot f_{\text{clk}}).$$

The proposed instance-level parallelism, combined with bandwidth-aware NBU , simpler arbitration, and HBM channel mapping, improves overall design scalability, simplifies floorplanning, and enables higher clock frequencies. Empirical results validating this optimization are provided in Section V.

V. EXPERIMENTAL RESULTS

A. Experimental Setup

All experiments are conducted on the AMD/Xilinx Alveo U280 FPGA platform, equipped with 32 HBM channels [21]. FPGA board energy consumption is measured using the XRT power profiler [24]. StreamNTT designs are synthesized using TAPA [5], RapidStream [20], and AMD/Xilinx Vitis 2023.2 [24] tools.

We evaluate StreamNTT using four PQC-related NTT configurations. $n = 256, q = 7681$ originates from early Ring-LWE cryptography. $n = 256, q = 8380417$ is the configuration standardized in CRYSTALS-Dilithium, a NIST PQC digital

signature scheme. $n = 1024, q = 12289$ corresponds to the reference parameter used in the NewHope key encapsulation mechanism. $n = 1024, q = 3221225473$ is an experimental high-security lattice scheme introduced in [22].

B. Optimization Analysis

We analyze the effect of incrementally applying the proposed optimizations to the configuration of $n = 1024, q = 3221225473$. The resulting impact on throughput, area utilization, and clock frequency is summarized in Table III. The baseline reflects the conventional dataflow implementation characterized by the following: (i) each BU is realized as a separate dataflow module [10], [11]; (ii) within a module, coefficient reorder buffer read, butterfly compute, and buffer write are implemented as separate pipelined loops [7], [8]; and (iii) all butterflies are combined into a single NTT instance [12]. Throughput is reported as the averaged latency of processing 10,000 polynomials and is measured in mega-polynomials per second (MPoly/s) [12].

Introducing the ICBU enables concurrent execution of coefficient buffer read, butterfly compute, and buffer write operations. The clock frequency remains nearly unchanged, while throughput improves by a factor of 2.9. Next, merging L-stage and X-stage butterflies reduces the resources allocated to FIFOs and allows butterflies to share the twiddle ROMs via two-port BRAMs. This optimization decreases LUT, flip-flop (FF), and BRAM consumption by 14%, 25%, and 39%, respectively. However, despite the reduced resource utilization (24% LUT utilization of the Alveo U280 platform in the ICBU+Merge variant), the design could not scale beyond 320 BUs because the Vitis tool struggled to place the large merged X-stage module within a single SLR. By introducing instance-level parallelism (ILP) and consecutive HBM channel mapping, we were able to double the total number of BUs (TBUs) to 640. Furthermore, assigning a different HBM channel pair to each instance allows independent execution of each instance and simplifies data arbitration among HBM channels. This increases the effective throughput of each butterfly by 29% ($= 32.4/12.6 \cdot 320/640$). Overall, the cumulative speedup achieved after applying all proposed optimizations is $7.2\times$.

Table IV reports the final NTT throughput, effective HBM bandwidth, and energy consumption for the four PQC con-

TABLE V
FPGA PLATFORM RESOURCES USED FOR NORMALIZATION IN TABLE VI

Platform	LUT	FF	DSP	BRAM18K	BW (GB/s)
XC7VX485T	304K	607K	2,800	2,060	12.8
XC7VX690T	433K	866K	3,600	2,940	25.6
Alveo U280	1,304K	2,607K	9,024	4,032	460

figurations. We obtain higher throughput and lower energy consumption for configurations with $n = 256$ compared to $n = 1,024$ because the NTT has a computational complexity of $O(n \log n)$. We observe a similar trend for smaller q because reducing the coefficient bitwidth $w_{\text{coeff}} (= \lceil \log q \rceil)$ from 32 bits ($q = 3221225473$) to 14 bits ($q = 12289$) allows more coefficients to be transferred per HBM access.

C. Quantitative Comparison with Related Works

Next, we provide a quantitative comparison with state-of-the-art throughput-optimized FPGA NTT accelerators. Since our objective is to maximize throughput by fully utilizing the available resources of the target FPGA platform, we report normalized throughput values that account for differences in available LUTs, digital signal processing (DSP) slices, and external memory bandwidth across platforms (Table V). For example, for a prior work that targets VX690T, the throughput normalized by LUT is defined as

$$\text{Thrpt}_{\text{LUTnm,VX690T}} = \frac{\text{Thrpt}_{\text{prior_work}}}{\text{Thrpt}_{\text{ours}}} \times \frac{\text{LUT}_{\text{U280}}}{\text{LUT}_{\text{VX690T}}}.$$

But this comparison should be regarded as a reference only, since differences in target applications, Vitis toolchain versions, FPGA technologies, moduli, and modular reduction algorithms can also affect the results.

The comparison is shown in Table VI. For $n = 1024$ and $\log q$ in the 28–32b range, our design achieves substantially higher throughput than PipeNTT [10] and Proteus [11], which rely only on butterfly pipeline parallelism and inter-stage parallelism (limited to 10 BUs for $n = 1024$). In contrast, NTTGen [15] deploys many more butterflies (320 in total), yet its throughput remains relatively low (0.91 MPoly/s versus our 32.4 MPoly/s). This limitation arises because its instance-level parallelism is coupled to the number of processed polynomials, preventing full exploitation of inter-stage parallelism through coarse-grained pipelining across multiple polynomials.

The table also demonstrates that our design achieves $2.7\times$ ($= 32.4 / 12.1$) higher throughput than AutoNTT on the same Alveo U280 board for an NTT with $n = 1024$ and $\log q = 28$ -32 bits. This improvement arises from our proposed optimizations. By merging dataflow modules, we reduced the number of

FIFOs and twiddle ROMs (Section IV-B), which enabled the placement of more BUs (640 versus 320). In addition, AutoNTT does not utilize instance-level parallelism as our design (Section IV-C), and the Vitis tool likely encountered greater placement and routing challenges when scaling to many BUs on a multi-SLR platform such as Alveo U280. Moreover, each BU attains 34% higher throughput with our ICBU optimization (Section IV-A), which operates at a higher clock frequency.

VI. CONCLUSION

In this work, we presented StreamNTT, a high-throughput, HLS-based NTT accelerator for HBM-equipped FPGAs. StreamNTT fully exploits the inherent parallelism of the NTT by addressing key obstacles that limit the scalability of streaming dataflow designs. The proposed HLS-friendly ICBU improves throughput by integrating butterfly computation and reorder buffer management into a single pipelined loop. Merging L- and X-stage BUs significantly reduces LUT and BRAM utilization, while placement-aware instance-level parallelism enables the deployment of more BUs and simultaneously enhances per-butterfly throughput. Collectively, these optimizations yield a $7.2\times$ improvement in throughput. StreamNTT delivers $2.7\times$ or higher throughput compared to state-of-the-art FPGA-based NTT accelerators under PQC configurations. As an open-source implementation, StreamNTT provides a practical and scalable foundation for advancing high-performance PQC accelerator design.

For future work, we plan to extend StreamNTT to support FHE schemes that operate with much larger polynomial degrees and a greater number of moduli (e.g., $n = 2^{17}$ and more than 40 moduli [25]) compared to typical PQC schemes. Such configurations substantially increase throughput demands and introduce new architectural challenges. Possible directions include: 1) adopting a more general modular reduction algorithm capable of handling diverse arbitrary moduli, 2) efficiently utilizing UltraRAMs (URAMs) as coefficient reorder buffers, as buffer sizes scale proportionally with the polynomial degree and the number of moduli, and 3) generating twiddle factors on the fly from a compact set of base values to avoid the cost of storing all twiddle factors for multiple moduli in the FPGA internal memory.

ACKNOWLEDGMENT

We thank Yuze Chi, Linfeng Du, Licheng Guo, Jason Lau, Yuanlong Xiao, Yutong Xie (Rapidstream), Hanho Lee, and Yongwoo Lee (Inha University) for the discussion and the help with the implementation.

TABLE VI
QUANTITATIVE COMPARISON WITH RELATED WORKS ($n = 1024$, $\log q = 28\text{B}$ – 32B)

Work	$\log q$	Reduction Scheme	FPGA Platform	LUT	FF	DSP	BRAM / URAM	TBU	Freq (MHz)	Thrpt (MPoly/s)	Thrpt LUTnm	Thrpt DSPnm	Thrpt BWnm
PipeNTT [10]	28b	k-RED	VX485T	3.4K	3.1K	63	6 / 0	10	175	0.17	0.023	0.017	0.19
Proteus [11]	28b	Mntgmry	VX485T	9.8K	4.7K	40	3 / 0	10	150	0.29	0.038	0.029	0.32
NTTGen [15]	28b	Mersenne	VX690T	206K	159K	640	80 / 0	320	210	0.91	0.084	0.070	0.50
AutoNTT [12]	28b	Mntgmry	AU280	499K	441K	1,920	32 / 32	320	242	12.1	0.37	0.37	0.37
StreamNTT	32b	Zhang [19]	AU280	648K	605K	2,560	536 / 0	640	306	32.4	1	1	1

REFERENCES

- [1] M. Sosnowski *et al.*, “The Performance of Post-Quantum TLS 1.3,” in *Proc. Int. Conf. Emerging Networking EXperiments and Technologies (CoNEXT)*, 2023, pp. 19–27.
- [2] X. Carril *et al.*, “Hardware acceleration for high-volume operations of CRYSTALS-kyber and CRYSTALS-dilithium,” *ACM Trans. Reconfigurable Technology and Systems*, vol. 17, no. 3, 2024.
- [3] D. Sikeridis, P. Kampanakis, and M. Devetsikiotis, “Post-quantum authentication in TLS 1.3: A performance study,” *Cryptology ePrint Archive*, 2020.
- [4] Y. Doroz, S. Reeves, and M. Wolf, “Introducing NVIDIA cuPQC for GPU-Accelerated Post-Quantum Cryptography,” 2024. [Online]. Available: <https://developer.nvidia.com/blog/introducing-nvidia-cupqc-for-gpu-accelerated-post-quantum-cryptography/>
- [5] Y. Chi *et al.*, “Extending high-level synthesis for task-parallel programs,” in *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 204–213.
- [6] D. T. Nguyen, V. B. Dang, and K. Gaj, “High-level synthesis in implementing and benchmarking number theoretic transform in lattice-based post-quantum cryptography using software/hardware codesign,” in *Int. Symp. Applied Reconfigurable Computing*, 2020, pp. 247–257.
- [7] A. C. Mert *et al.*, “An extensive study of flexible design methods for the Number Theoretic Transform,” *IEEE Trans. Computers*, vol. 71, no. 11, pp. 2829–2843, 2022.
- [8] A. El-Kady *et al.*, “High-level synthesis design approach for number-theoretic multiplier,” in *IFIP/IEEE Int. Conf. Very Large Scale Integration (VLSI-SoC)*, 2022.
- [9] Xilinx, “Vitis High-Level Synthesis User Guide (UG1399),” 2025. [Online]. Available: <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls>
- [10] Z. Ye, R. C. Cheung, and K. Huang, “PipeNTT: A pipelined number theoretic transform architecture,” *IEEE Trans. Circuits and Systems II: Express Briefs*, vol. 69, no. 10, pp. 4068–4072, 2022.
- [11] F. Hirner, A. C. Mert, and S. S. Roy, “Proteus: A pipelined NTT architecture generator,” *IEEE Trans. Very Large Scale Integration Systems*, 2024.
- [12] D. Kumarathunga, Q. Hu, and Z. Fang, “AutoNTT: Automatic Architecture Design and Exploration for Number Theoretic Transform Acceleration on FPGAs,” in *Proc. IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machine (FCCM)*, 2025.
- [13] E. Alkim *et al.*, “Post-quantum Key Exchange—A New Hope,” in *USENIX Security Symposium*, 2016, pp. 327–343.
- [14] L. Beckwith, D. T. Nguyen, and K. Gaj, “High-performance hardware implementation of lattice-based digital signatures,” *Cryptology ePrint Archive*, 2022.
- [15] Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, “NTTGen: a framework for generating low latency ntt implementations on FPGA,” in *Proc. ACM Int. Conf. Computing Frontiers*, 2022, pp. 30–39.
- [16] T. Ye *et al.*, “FPGA acceleration of Number Theoretic Transform,” in *Int. Conf. High Performance Computing*, 2021, pp. 98–117.
- [17] Z. Lu *et al.*, “An NTT/INTT Accelerator with Ultra-High Throughput and Area Efficiency for FHE,” in *Proc. ACM/IEEE Design Automation Conference (DAC)*, 2024.
- [18] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series,” *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.
- [19] N. Zhang *et al.*, “Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT,” *IACR Trans. Cryptographic Hardware and Embedded Systems*, pp. 49–72, 2020.
- [20] RapidStream, “RapidStream TAPA Documentation,” 2025. [Online]. Available: <https://tapa.readthedocs.io/en/main/>
- [21] AMD, *AMD Alveo U280 Data Center Accelerator Card: Product Brief*, 2025. [Online]. Available: <https://www.xilinx.com/publications/product-briefs/alveo-u280-product-brief.pdf>
- [22] B. Li, Y. Yan, Y. Wei, and H. Han, “Scalable and parallel optimization of the number theoretic transform based on FPGA,” *IEEE Trans. Very Large Scale Integration Systems*, vol. 32, no. 2, pp. 291–304, 2024.
- [23] Y.-k. Choi, Y. Chi, J. Lau, and J. Cong, “TARO: Automatic optimization for free-running kernels in FPGA high-level synthesis,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 7, pp. 2423–2427, 2022.
- [24] Xilinx, “Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393),” 2025. [Online]. Available: <https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration>
- [25] S. Kim *et al.*, “Hardware architecture of a number theoretic transform for a bootstrappable RNS-based homomorphic encryption scheme,” in *Proc. IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machine (FCCM)*, 2020, pp. 56–64.