A Resource-Efficient Hardware Accelerator for Large-Size NTT via Algorithm–Architecture Co-Design

Kaixuan Wang
Shanghai Jiao Tong University
Shanghai, Shanghai, China
wangkaixuan@sjtu.edu.cn

Yifan Yanggong
Shanghai Jiao Tong University
Shanghai, Shanghai, China
wenzisdfz@sjtu.edu.cn

Xiaoyu Yang
Chipltech
Shanghai, Shanghai, China
huiniu@chipltech.com

Chenti Baixiao
Chipltech
Shanghai, Shanghai, China
baiyan@chipltech.com

Lei Wang*
Shanghai Jiao Tong University
Shanghai, Shanghai, China
wanglei_hb@sjtu.edu.cn

Abstract

Large-size Number Theoretic Transforms (NTTs) are key operations in modern Zero-Knowledge Proofs (ZKPs), where the NTT size often reaches millions and the arithmetic is over wide prime fields. To handle such NTTs on hardware, prior designs commonly rely on the decomposition algorithm, which makes large-size NTTs feasible by streaming sub-NTTs through limited on-chip buffers. However, in practical implementations, decomposition alone is insufficient to ensure high efficiency. Since coefficients and twiddle factors remain off-chip, performance tends to depend on data movement across the memory hierarchy and delivery to the processing elements (PEs).

To address these issues, we present RENTT, a resource-efficient accelerator for large-size NTTs with decomposition-oriented data movement schemes and memory hierarchy design. We implement RENTT on an FPGA and verify that RENTT supports NTT sizes up to $N = 2^{28}$ over 256-bit fields and completes a 2^{28}-point NTT in 1.52 seconds with 16 processing elements. Compared with the state-of-the-art FPGA baseline SAM, RENTT provides 2.64× speedup (1.52s vs. 4.02s), reduces 46.4% DSP usage (3629 vs. 6776 DSPs), and achieves a 3.58× lower area-time product.

CCS Concepts

• **Hardware → Hardware accelerators**.

Keywords

Number Theoretic Transform, FPGA Acceleration, Zero-knowledge Proofs

ACM Reference Format:
Kaixuan Wang, Yifan Yanggong, Xiaoyu Yang, Chenti Baixiao, and Lei Wang. 2026. A Resource-Efficient Hardware Accelerator for Large-Size NTT via Algorithm–Architecture Co-Design. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD '26).* ACM, New York, NY, USA, 9 pages. https://doi.org/3831252.3834217

*Corrsponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICCAD '26, San Jose, CA, USA
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2873-0/2026/11
https://doi.org/3831252.3834217

1 Introduction

Zero-Knowledge Proof (ZKP) [20] enables a prover to convince verifiers of a statement's validity without revealing any additional information. Recently, ZKPs have found urgent applications in various cutting-edge domains, such as blockchain [29, 34], verifiable fully homomorphic encryption [30], and machine learning [2, 12].

However, the proof generation is still computationally and memory intensive as the dominant performance bottleneck. This limits broader deployment of ZKP protocols in practice, and motivates extensive efforts to improve the proving efficiency [8, 10, 15, 18, 19]. In particular, hardware acceleration (e.g., ASIC-based accelerators [17, 28, 36], FPGA-based accelerators [1, 14, 23], and GPU-based accelerators [24, 25]) has emerged as a promising direction to reduce the latency of proof generation.

Following this direction, this paper focuses on *Number Theoretic Transform (NTT)* [26], one of the major bottlenecks in ZKP proof generation. Specifically, the proof of a target computation is typically reduced to checking a collection of polynomial relations. These relations frequently involve polynomial multiplications. Such polynomial multiplication is usually implemented via the NTT operation in practice. As a result, NTT executions dominate the proof generation performance. Hardware accelerators for ZKP protocols face the challenge of computing large-size NTTs under limited on-chip memory resources. Specifically, the NTT size often reaches 2^{20} to 2^{28} in practical ZKP protocols (e.g., Zcash [9] and Filecoin [11]) and the field bit-width is on the order of hundreds of bits (e.g., BN254 and BLS12-381). The resulting working set (such as the coefficients and twiddle factors) quickly grows to multiple gigabytes, far exceeding the on-chip memory capacity of modern FPGAs (typically tens of megabytes). Existing hardware accelerators for such large-size NTT commonly adopt the decomposition algorithm [6]. This algorithm views an N-point NTT as an $n_1 \times n_2$ matrix, streams intermediate data through off-chip memory, and reduces on-chip memory to $O(\max(n_1, n_2))$. It is widely adopted by hardware implementations [14, 17, 21, 23, 32, 35–37].

While decomposition makes large-size NTTs feasible, two practical issues remain underexplored. First, twiddle-factor (TF) management under parallel processing faces a trade-off: on-the-fly generation saves storage but consumes DSPs, whereas full precomputation

may cause bank conflicts when multiple PEs access TFs simultaneously. Second, the off-chip data layout that is burst-friendly in one decomposition phase often becomes stride-intensive in the next, leading to fragmented memory transactions. A detailed discussion is given in Sec. 2.3.

Overall, existing architectures should take into consideration an end-to-end co-design between TF management, burst-efficient off-chip memory access, and conflict-free on-chip parallelism. This fragmentation prevents the optimization of the Area-Time Product (ATP). Optimizing any single component in isolation typically shifts pressure to another (e.g., saving DSP slices via precompute increases memory cost; reducing the latency by using more PEs increases computation cost), leading to suboptimal latency–area trade-offs captured by ATP.

The main contributions of this paper are as follows.

- **Conflict-free twiddle factor management.** We propose a precomputation-based TF management scheme with a multi-bank address mapping and stage-aware access scheduling, guaranteeing that TF supply is conflict-free in every cycle while using only a single copy of the TF table shared across all PEs (Sec. 3).
- **Conflict-free coefficient transpose scheme.** We propose a transpose scheme that is integrated into Step 2 of the decomposition algorithm flow: while performing the mandatory element-wise TF multiplication, the accelerator applies an address permutation and writes coefficients back in a layout that restores contiguous, burst-friendly accesses for small-size NTTs in Step 3 (Sec. 4). This avoids burst fragmentation that would otherwise arise from strided column reads.
- **A decoupled memory-centric, parameterizable and configurable accelerator.** We present *RENTT*, a resource-efficient accelerator for large-size NTTs. RENTT adopts a decoupled multi-banked memory hierarchy to sustain high utilization under off-chip streaming. The accelerator is design-time parameterizable in both the PE parallelism T and the maximum supported NTT size $N_{\max}$, enabling implementations tailored to different FPGA resource budgets. Moreover, RENTT is runtime-configurable and supports any $N \le N_{\max}$ without modifying the circuit (Sec. 5).

We implement RENTT and deploy it on an AMD Alveo U280 FPGA. RENTT supports NTT sizes up to $N = 2^{28}$ over 256-bit fields and completes a 2^{28}-point NTT in 1.52 seconds with 16 PEs (Sec. 6). Table 1 summarizes representative large-size NTT accelerators which also support wide bit-width arithmetic. Compared to state-of-the-art baselines, RENTT achieves a 3.58× lower ATP than SAM [14] at the same parallelism, primarily by eliminating twiddle-factor generation overhead while sustaining high PE utilization. Overall, to the best of our knowledge, RENTT achieves the lowest reported ATP among comparable FPGA-based large-size NTT accelerators, while remaining design-time parameterizable in $(T, N_{\max})$ and runtime-configurable for any $N \le N_{\max}$.

Table 1: Comparison of state-of-the-art hardware accelerators for large-size NTTs. TF Strategy: precomputed vs on-the-fly twiddle factor supply. Max Par.: maximum PE parallelism degree. Param.: design-time parameterizable. Conf.: runtime-configurable. Max Size: maximum supported NTT size. ATP: area-time product computed using the latency for computing a 2^{24}-point NTT with 256-bit width. "-": not directly comparable.

Work	TF Strategy	Max Par.	Max Size	Param.	Conf.	ATP (Improvement)
PipeZK [36]	Precomputed	4	2^{20}	✗	✓	-
SAM [14]	On-the-fly	16	2^{28}	✗	✓	131.34 (3.58 ×)
HiFA [23]	Precomputed	12	2^{24}	✓	✗	44.72 (1.22×)
This Work	**Precomputed**	**16**	2^{28}	✓	✓	**36.62 (1×)**

2 Preliminary

We denote the finite field modulo a prime number p as $\mathbb{F}_p$, where $W = \lceil \log_2 p \rceil$ represents the bit width. A polynomial $A(x) = \sum_{i=0}^{N-1} a_i x^i \in \mathbb{F}_p^{<N}[x]$ is identified by its coefficient vector $\mathbf{a} = (a_0, \cdots, a_{N-1}) \in \mathbb{F}_p^N$.

2.1 Number Theoretic Transform

The number theoretic transform (NTT) [26] is a variant of Discrete Fourier Transform (DFT) defined over $\mathbb{F}_p$. It maps a coefficient vector $\mathbf{a}$ to its evaluation form $\hat{\mathbf{a}} = (\hat{a}_0, \ldots, \hat{a}_{N-1}) \in \mathbb{F}_p^N$. Compared with the direct method for batch evaluation of a degree-$(N-1)$ polynomial at N distinct points, the NTT reduces the computational complexity from $O(N^2)$ to $O(N \log N)$. The forward NTT and its inverse (INTT) are defined as:

$$\hat{a}_i = \sum_{j=0}^{N-1} a_j \cdot \omega_N^{ij} \bmod p, \quad a_j = N^{-1} \cdot \sum_{i=0}^{N-1} \hat{a}_i \cdot \omega_N^{-ij} \bmod p \qquad (1)$$

where $\omega_N \in \mathbb{F}_p$ is a primitive N-th root of unity (satisfying $\omega_N^N \equiv 1$ and $\omega_N^{N/2} \equiv -1 \bmod p$). The powers of ω_N used in these computations are referred to as twiddle factors. Hardware implementations exploit the reduction property $\omega_{tN}^{tk} = \omega_N^k$ to derive sub-NTT twiddle factors from a master table. The inverse factors are obtained via Fermat's Little Theorem as $\omega_N^{-1} \equiv \omega_N^{p-2} \bmod p$.

The standard Cooley-Tukey radix-2 Decimation-in-Time (DIT) algorithm [16] computes the N-point NTT is $\log_2 N$ stages, each consisting of $N/2$ butterfly operations. The core computational unit is the *butterfly* (BF) operation, which processes two elements u, v and a twiddle factor ω per cycle.

The hardware efficiency of NTT is constrained by two factors. First, the arithmetic units (multipliers and adders over $\mathbb{F}_p$) are computation resource-intensive due to the large bit width W and modular reduction. Second, the data access pattern is highly irregular. In the i-th stage, the elements selected for a BF are spaced $N/2^{i+1}$ positions apart. As the stage index i increases, this stride decreases, creating complex memory access patterns that often result in bank conflicts and computational stalls in multi-PE (Processing Element) architectures.

2.2 Large-size NTT and Decomposition Algorithm

In zero-knowledge proof systems, N often reaches 2^{20} to 2^{28} with W over hundreds of bits (e.g., $W \approx 256$ bits) [14]. This creates a working set (coefficients and twiddle factors) of several gigabytes, far exceeding the on-chip memory capacity of modern FPGAs (typically tens of megabytes). In prior works [14, 23, 36], such NTT instances are regarded as a large-size NTT. Due to on-chip memory constraints, the entire working set must reside in off-chip memory, necessitating the iterative loading of data blocks into on-chip memory for processing.

To address this, the decomposition algorithm [6] is widely adopted by NTT accelerators [14, 23, 32, 35, 36]. It treats the N-point input as an $n_1 \times n_2$ matrix: **Step 1** performs n_1 instances of n_2-point NTTs on each row; **Step 2** multiplies each element $m_{i,j}$ by ω_N^{ij} and transposes the matrix; **Step 3** performs n_2 instances of n_1-point NTTs on the new rows. This decomposition reduces the on-chip memory requirement to only $O(\max(n_1, n_2))$. Our accelerator design is general and supports decompositions with $n_1 \neq n_2$. Specifically, when $n_1 > n_2$, our accelerator executes the n_2-point NTT and n_1-point NTTs independently in Step 1 and Step 3, respectively. In Step 2, we treat the data as a logical $n_1 \times n_1$ layout to simplify transpose scheduling, while masking the inactive columns ($j \geq n_2$) so the computation still corresponds to the original transform. For ease of presentation, we assume $n_1 = n_2 = n$ (thus $N = n^2$) throughout the rest of this paper.

2.3 Motivation

Our goal is to optimize end-to-end efficiency for decomposition-based large-size NTTs on FPGA, where both arithmetic execution and data movement contribute to the overall cost. While the decomposition algorithm makes large-size NTTs feasible under limited on-chip capacity, existing FPGA designs [14, 23] for large-size NTTs suffer from three recurring issues.

Twiddle-factor management under limited resource budgets. For large-size NTTs over wide bit-width fields, twiddle-factor (TF) handling can become an important design consideration: generating TFs on-the-fly reduces storage but consumes DSP resources that compete with the main datapath, whereas full precomputation increases storage and may push TF traffic to off-chip memory. A key observation is that TF consumption in decomposition-based NTTs is highly structured and stage-local, so only a small working subset is needed at any time if the access order is chosen carefully. This motivates a precomputation-based approach with compact table organization and address-aware access that matches the decomposition schedule, reducing DSP pressure without inflating the on-chip footprint or off-chip traffic.

Avoiding stalls in decomposition-induced data reordering. NTTs require data reordering (e.g., matrix transpose) between phases, yet a straightforward mapping often mismatches the sequential, burst-friendly nature of off-chip memories (DDR/HBM) with the strided, parallel access demanded by on-chip PEs, leading to bank conflicts and pipeline bubbles. Prior work [14] adopts a more complex decomposition to mitigate burst-unfriendly column-wise accesses, which incurs additional control-logic overhead. Alternatively, other designs leverage tiled memory layouts [3, 13] to improve the burst locality of column-wise NTT accesses by processing the matrix in tiles/blocks, often at the cost of more complex address generation or additional buffering. The insight is that this overhead is largely a consequence of an uncoordinated address/banking design: with a permutation-aware mapping, reordering can be realized through conflict-free on-chip accesses and aligned burst transfers. This motivates a transpose/reordering scheme that co-designs on-chip banking with off-chip burst layout to sustain a continuous compute pipeline.

Scalable and flexible accelerator under varying FPGA budgets. Several prior large-size NTT accelerators employ tightly coupled control logic and fixed resource ratios, which limits portability across FPGA platforms and complicates scaling as routing and control overheads grow [23, 36]. Our observation is that large-size NTT execution admits a regular, iterative schedule: the same workload can be replayed across tiles/steps, while the control scheme can be parameterized independently of the physical instantiation scale. This motivates a decoupled and parameterized accelerator where scheduling is separated from the datapath scale, allowing the design to adapt different resource compositions and to scale performance with the number of instantiated PEs.

3 Conflict-Free Twiddle Factor Management Scheme

Building on the precomputed twiddle-factor (TF) approach, we propose a conflict-free TF management scheme to supply twiddle factors to parallel butterfly operations. We first describe the twiddle factor provisioning problem and then present the proposed memory-access scheme.

3.1 Problem Statement

In the decomposition algorithm (detailed in Sec. 2.2), the NTT computation is split into two types: small-size NTTs in Step 1 and Step 3, and element-wise multiplication with transpose in Step 2. These two types of computation impose distinct memory access patterns on TF management:

- **Element-wise multiplication with linear access pattern** requires N TFs ($\{\omega_N^k\}_{k\in[0,N-1]}$). Since access is sequential, streaming from off-chip memory is straightforward.
- **Small-size NTTs with butterfly access pattern** require n unique TFs ($\{\omega_n^k\}_{k\in[0,n-1]}$) for the small-size NTTs. Here, T PEs must access TFs in complex, non-linear patterns simultaneously.

The primary challenge lies in small-size NTTs. A precomputation approach used by the prior precomputation-based work [23] would require stage-wise table expansion to prevent memory access conflicts, leading to linear memory growth ($T \cdot \mathcal{M}$) where $\mathcal{M} = n/2 \cdot W$ denotes memory consumption for storing $\{\omega_n^k\}_{k\in[0,n/2-1]}$. Conversely, accessing a single table without optimization causes severe bank collisions, stalling the pipeline. For instance, consider a configuration with $T = 4$ PEs and a 4-bank memory utilizing standard linear interleaving, where the twiddle factor ω^k is stored in Bank (k mod 4). During a stage where the butterfly operations require PEs to access indices with a stride of 8 (e.g., requesting

$\{\omega^0, \omega^{16}, \omega^8, \omega^{24}\}$ in the last stage of a small NTT), all four access requests target Bank 0 simultaneously. Consequently, the memory controller must serialize these requests, forcing the pipeline to stall for 3 cycles and reducing the effective parallelism by 75%.

3.2 Twiddle Factor Memory Access Scheme

We design this scheme in three steps: precomputation and hierarchical selection, optimized storage mapping, and stage-aware conflict-free access.

3.2.1 The Data Preparation: Precomputation and hierarchical selection. In data preparation, we precompute a master table of twiddle factors $\{\omega_N^k\}_{k\in[0,N-1]}$ and store it in off-chip memory in a linear (contiguous) layout. The decomposition algorithm consumes twiddle factors in two distinct ways. In Step 1 and Step 3, the small-size NTTs require a stage-wise twiddle set for an n-point NTT. In Step 2, the element-wise multiplication requires a stream of ω_N^{ij} values associated with matrix coordinates. To avoid maintaining multiple redundant tables, we adopt a hierarchical selection from the master table. Using the reduction property, the twiddle factors needed by the n-point sub-NTTs can be obtained as a strided subset of the master table, and then staged into on-chip memory for conflict-free access during butterfly execution.

The off-chip loading frequency differs across steps. For Step 1 and Step 3, the required n-point twiddle table is reused across all sub-NTTs within the step; thus it is loaded once at the beginning of the step and then repeatedly accessed on-chip. In contrast, Step 2 processes the matrix in n tiles, and each tile uses a different segment of twiddle factors; therefore, the accelerator streams twiddle factors from off-chip memory n times over the course of Step 2.

3.2.2 The Storage Mechanism: Multi-bank Address Mapping. When loading TFs from off-chip to on-chip memory, we employ a bit-reversal permutation ($BI = \text{bitreverse}(i) \bmod T$) to distribute data across banks. This mapping breaks the linear correlation of standard indexing. By scattering logically adjacent TFs across physically distinct banks, we minimize the probability of collision during the strided access patterns characteristic of NTT butterfly operations. Here BI denotes the bank index into which the old address is mapped. BA denotes the offset of the storage location within each inner bank.

Algorithm 1: Twiddle Factor Address Mapping Algorithm

Input: NTT size n, PE number T.
Output: $\boldsymbol{Pos}$: $\boldsymbol{Pos}[i]$ is a tuple (BI, BA), where $BI \in [0, T-1]$ and $BA \in [0, n/T-1]$ are bank index and the offset of the storage location in each bank for twiddle factor ω^i

for $i \leftarrow 0$ **to** $n-1$ **do**
  $i' \leftarrow \text{bitreverse}(i)$
  $BI \leftarrow i' \bmod T$, $BA = \lfloor i'/T \rfloor$
  $\boldsymbol{Pos}[i] \leftarrow (BI, BA)$
end
return $\boldsymbol{Pos}$

3.2.3 The Retrieval Mechanism: Conflict-free Access Scheduling. To support the pipelined processing unit, the memory subsystem must supply T twiddle factors per cycle. However, the demand pattern of the NTT algorithm shifts dynamically across stages. We propose a stage-aware access scheduling mechanism (detailed in Algorithm 2) that adapts the addressing logic to three distinct phases:

Design Intuition. The scheduling logic exploits the hierarchical group structure of the NTT algorithm. In early stages, all T PEs require the same TF, so a single read is broadcast. In later stages, PEs require distinct TFs; the scheduler uses a base group index $Base_{GI}$ and PE offset to generate unique addresses that, combined with the bit-reversal storage mapping, land on different banks.

Algorithm 2: Twiddle Factor Memory Access Algorithm

Input: NTT size $n = 2^k$, number of processing elements $T = 2^t$ with $n \geq T$; a stage index $r \in [0, k-1]$ and a cycle index $c \in [0, \frac{n}{2T} - 1]$.
Output: $\boldsymbol{tf}$, where $tf[i]$ denotes the position of the twiddle factor accessed by i-th PE for a given stage index r and cycle index c.

/* 1. Compute Structural Parameters */
$n_{tfpg} = n/2^{r+1}$; $n_g = 2^r$
$n_{bps} \leftarrow n/(2T)$; $n_{bpg} \leftarrow n/(2n_gT)$; $n_{gpc} \leftarrow T/n_{tfpg}$
/* 2. Compute Positional Parameters */
$GI \leftarrow \lfloor c/n_{bpg} \rfloor$
$Base_{GI} \leftarrow c \times n_{gpc}$
/* 3. Determine Bank and Address for each PE */
for $i \leftarrow 0$ **to** $T-1$ **do**
  if $r < t$ **then**
    /* Stage 1: Direct Broadcast */
    $BI \leftarrow GI$, $BA \leftarrow 0$
  end
  else if $r < k - t$ **then**
    /* Stage 2: Modulo Broadcast */
    $BI \leftarrow GI \bmod T$, $BA \leftarrow \lfloor GI/T \rfloor$
  end
  else
    /* Stage 3: Parallel Access */
    $BI \leftarrow (Base_{GI} + \lfloor i/n_{tfpg} \rfloor) \bmod T$
    $BA \leftarrow \lfloor (Base_{GI} + \lfloor i/n_{tfpg} \rfloor)/T \rfloor$
  end
  $tf[i] \leftarrow (BI, BA)$
end
return $\boldsymbol{tf}$

Algorithm 2 translates the linear execution time (cycle c) into the hierarchical coordinates of the NTT. This mapping relies on a set of derived control parameters:

- **Structural Parameters:** The variable n_{tfpg} (twiddle factors per group) defines the *periodicity* or *reusable length* of the twiddle factors in the current stage. In the r-th stage of a radix-2 CT NTT, this run length decays as $n_{tfpg} = n/2^{r+1}$. A large n_{tfpg} implies strong TF reuse and broadcast delivery,

whereas a small n_{tfpg} requires more distinct TFs per cycle and parallel delivery. n_{bps} denotes the cycle number for T PEs to execute all butterfly operations in each stage, which is equal to $n/(2T)$. Butterfly operations are processed in groups to match the PE parallelism. Let $n_g = 2^r$ denote the number of groups in r-th stage and n_{bpg} denote the number of butterfly operations per group, and let n_{gpc} denote the number of groups processed per cycle.

- **Positioning Parameters:** To locate the current operation within the NTT structure, *GI* serves as the group index, identifying which logic block the current cycle belongs to. For the final stages, $Base_{GI}$ provides the precise offset required to generate the parallel permutation pattern.

4 Conflict-free Coefficient Matrix Transpose Scheme

In this section, we propose a coefficient-matrix transposition scheme which introduces a mathematically derived address permutation coupled with a three-stage scheduling loop. This design achieves fully pipelined, conflict-free memory access during matrix transpose, hiding the transpose latency within the data transfer time.

4.1 Problem Statement

Large-size NTT decomposition alternates between two computation phases that access the coefficient matrix along different dimensions (row-wise vs. column-wise). A coefficient layout that is burst-friendly for one phase tends to become burst-unfriendly for the other, which degrades off-chip efficiency due to fragmented bursts and increased memory transactions. To restore burst-contiguous off-chip streaming for the subsequent phase, we must transpose the coefficient matrix stored in off-chip memory.

A straightforward full-matrix transpose in off-chip memory is expensive, as it introduces an additional pass of irregular reads/writes and is difficult to overlap with computation. Instead, we adopt a tiled off-chip layout [3] and decompose the global transpose into two parts: (i) *inter-tile placement*, which rearranges the positions of tiles in the global matrix, and (ii) *intra-tile transpose*, which transposes coefficients within each tile. Under a tiled layout, inter-tile placement can be naturally realized by choosing appropriate off-chip write-back addresses (thus maintaining burst-contiguous streaming). The remaining challenge is how to perform the intra-tile transpose efficiently on-chip without bank conflicts.

4.2 On-chip Transpose Scheme

RENTT provides a dedicated Secondary Memory to serve twiddle factors during Step 2 (EWM). After Step 2 completes, the TF table is no longer on the critical path. We exploit this slack by using the Secondary Memory as an *auxiliary buffer* to hold the post-EWM coefficients for transposition. The transpose datapath therefore reads coefficients from the Secondary Memory and writes the permuted results back into the Primary Memory.

Let $T = 2^t$ be the PE count and $n = 2^k$ be the number of coefficients in a tile (a $2^{k/2} \times 2^{k/2}$ block) which is equal to the size of sub-NTTs. Each on-chip banked memory (Primary Memory or Secondary Memory) has T banks and depth 2^{k-t}. We denote a physical location by (BI, BA), where $BI \in [0, T-1]$ is the bank index and $BA \in [0, 2^{k-t}-1]$ is the intra-bank address. The corresponding linear index within the tile is $I_p = BA \times T + BI$. The coordinate of I_p within the tile is $(x, y) = \left(I_p \bmod 2^{k/2}, \left\lfloor I_p/2^{k/2} \right\rfloor\right)$. Tile transpose swaps the two coordinates:

$$(x', y') = (y, x), \qquad I'_p = x' \times 2^{k/2} + y' = y \times 2^{k/2} + x. \quad (2)$$

Permutation-to-bank mapping. We implement the intra-tile transpose via a deterministic address permutation that maps (BI, BA) to a destination (BI', BA'). In particular, the destination (BI', BA') of linear index I'_p can be written in the following form:

$$BI' = I'_p \bmod T = \left(\left\lfloor BA/2^{k/2-t} \right\rfloor\right) \bmod T, \quad (3)$$

$$BA' = \left\lfloor \frac{I'_p}{T} \right\rfloor = \left\lfloor \frac{BA}{2^{k/2}} \right\rfloor + \left(T \times (BA \bmod 2^{k/2-t}) + BI\right) \times 2^{k/2-t}. \quad (4)$$

Note that the Primary Memory is physically organized as $2T$ banks for the subsequent compute pipeline; during transpose, we only exercise T bank writes per cycle (one per PE), while the $2T$-bank organization is preserved for paired accesses in later stages.

Conflict-free schedule construction. To guarantee bank-disjoint writes (i.e., $BI'_i \neq BI'_j, \forall i, j \in [0, T-1]$ and $i \neq j$) each cycle, we decompose

$$BA = 2^{k/2-t} \times s + a, \quad s \in [0, 2^{k/2}-1],\ a \in [0, 2^{k/2-t}-1]. \quad (5)$$

Then Eq. (3) reduces to $BI' = s \bmod T$. For a given cycle, we process T PEs indexed by $\tau \in [0, T-1]$ and set the source bank index as $BI = \tau$ (hence the read path is trivially conflict-free: one read each bank of Secondary Memory). We choose s as

$$s = \left(b \cdot T + c + BI\right) \bmod 2^{k/2}, \quad b \in [0, 2^{k/2-t}-1],\ c \in [0, T-1]. \quad (6)$$

Combining Eq. (6) with $BI' = s \bmod T$ yields $BI' = (c + BI) \bmod T$. Therefore, within the same cycle (fixed c), the set $\{BI'\}$ over $BI = \tau$ spans all T residues exactly once, guaranteeing conflict-free writes to the Primary Memory. By iterating loops in the order (b, a, c) and executing all T PEs per cycle, the scheduler covers all (BI, BA) locations and completes the intra-tile transpose with sustained throughput.

4.3 End-to-end Transpose Scheme

We now summarize the complete transpose flow in three stages under the tiled off-chip layout. Figure 1 illustrates the transpose scheme with a simplified example ($N = 2^{12}, T = 4$).

In Stage 1 (*burst-contiguous stream-in*), coefficients are streamed tile by tile from off-chip memory using long contiguous bursts and placed into the coefficient RAM in row-major order with the standard linear mapping $(BI, BA) = (i \bmod T,\ \lfloor i/T \rfloor), i \in [0, n-1]$. In parallel, the required TF table is streamed into the Secondary Memory to serve Step 2.

In Stage 2 (*intra-tile transpose*), after EWM finishes, the Secondary Memory holds the results. The transpose scheduler streams T coefficients per cycle from the T TF banks, and each PE writes its coefficient to the permuted coordinate (BI', BA') in the Primary Memory. Although the Primary Memory is physically organized as $2T$ banks, we pair every two banks into a logical T-bank view with doubled depth during transposition. By ensuring that the T read

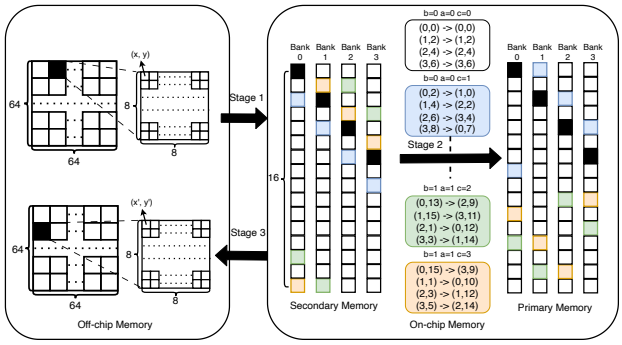


Figure 1: The coefficient matrix transpose scheme.

bank indices $\{BI\}$ and the T write bank indices $\{BI'\}$ are each pairwise distinct, the on-chip transpose achieves conflict-free accesses and sustains full PE throughput.

In Stage 3 (*burst-contiguous stream-out with inter-tile placement*), the transposed tile is streamed back to off-chip memory using long contiguous bursts. The off-chip *write-back addresses* follow the tiled layout of the transposed matrix, so the global inter-tile placement is realized implicitly by the off-chip addressing without an explicit global transpose pass.

In each cycle, the scheduler makes the destination bank indices $\{BI'\}$ span all T residues exactly once, so no two PEs write to the same bank simultaneously, ensuring the transpose schedule is conflict-free. Moreover, both stream-in and stream-out traverse tiles sequentially, while the reordering is performed on-chip within each tile, preserving burst-contiguous off-chip accesses. Finally, the scheme reuses the Secondary Memory after Step 2 as the transpose buffer, avoiding additional transpose-specific on-chip storage.

5 RENTT Architecture

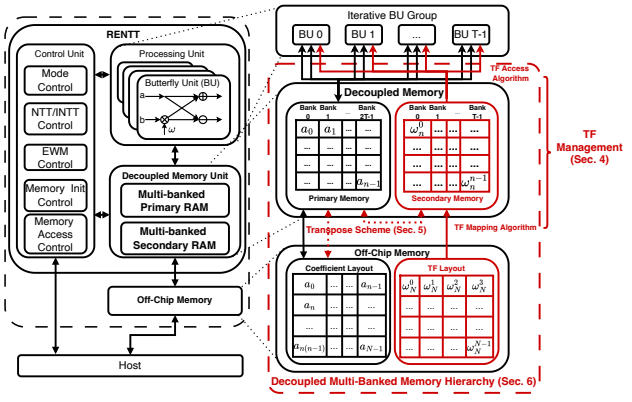


Figure 2: Overview of the RENTT.

As illustrated in Fig. 2, RENTT comprises three subsystems: a Control Unit, a Decoupled Multi-Banked Memory Unit, and a Processing Unit. Red highlights the proposed twiddle factor management (Sec. 3), coefficient transpose scheme (Sec. 4), and the decoupled multi-banked memory hierarchy in this section. By accessing the precomputed twiddle factors and using matrix transpose scheme (as detailed in Sec. 3 and 4), the hardware core remains resource-efficient while supporting high-parallelism computations.

5.1 Decoupled Multi-banked Memory Hierarchy

To eliminate structural hazards between polynomial coefficient access and twiddle factor retrieval, we design a decoupled multi-banked memory hierarchy. Unlike unified memory architectures that suffer from port contention, our design physically segregates the storage of dynamic data and static constants into two independent units: the *Primary Memory* and the *Secondary Memory*.

The *Primary Memory* is a coefficient buffer with a capacity of $n \times W$ bits, which is decoupled from twiddle-factor storage. It is implemented as a multi-banked memory with $2T$ banks to provide effective two-read and two-write ports for the compute pipeline. This banking factor is chosen to sustain $2T$ coefficient reads and $2T$ coefficient writes per cycle, matching the access demand of T parallel butterfly units (two inputs and two outputs per butterfly) without bank conflicts. In the EWM mode, this module also acts as the staging ground for the coefficient matrix transpose (Sec. 4).

Designed to implement the twiddle factor management scheme (Sec. 3), the *Secondary Memory* is a precomputed twiddle factor buffer with a capacity of $n \times W$ bits and is physically separated from the coefficient memory to eliminate port contention. This capacity matches the coefficient buffer because the element-wise multiplication (EWM) stage consumes one twiddle factor per coefficient; hence, a full pass over n coefficients requires n twiddle factors to be buffered. It is provisioned as a memory organized into T banks with one-read and one-write ports, enabling the Processing Unit to fetch T twiddle factors per cycle in parallel with coefficient accesses. This separation sustains a conflict-free pipeline without replicating per-PE twiddle tables or performing on-the-fly twiddle generation.

5.2 Processing Unit

The processing unit (PU) supports three operating modes: the n-point NTT, the n-point INTT, and element-wise multiplication (EWM). The PU consists of T processing elements (PEs) organized as an iterative PE architecture, following prior work [27]. Each PE implements a butterfly unit (BU) and performs one butterfly operation per cycle. Therefore, the PU sustains up to T parallel butterfly operations per clock cycle. Each BU integrates a modular adder, a modular subtractor, and a modular multiplier. The PU exploits intra-stage parallelism via time-multiplexing. The array of T PEs in the PU iteratively computes all butterfly operations within the current stage before advancing to the next stage. Each BU implements a modular multiplier (Karatsuba-based large-integer multiplication [33] followed by Barrett reduction [7]) alongside modular add/subtract units, dominating per-PE resource utilization.

5.3 Hierarchical Control and Computation Flow

The Control Unit (CU) in the RENTT functions as the address generation engine, translating the logical indices defined in our algorithms into physical bank addresses. It orchestrates the workflow shown in Fig. 3.

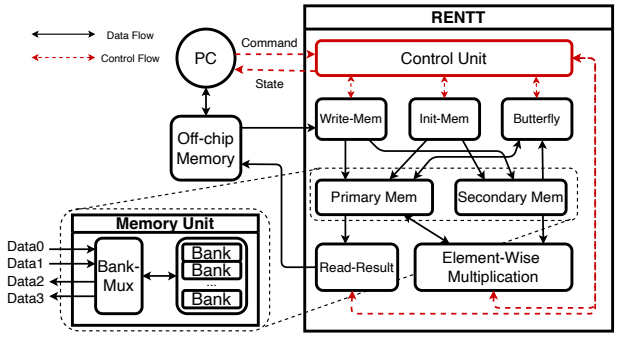


Figure 3: The dataflow and control signals of RENTT.

Orchestration of large-size NTTs. To execute a large-size NTT of size N, the CU decomposes the operation according to the decomposition algorithm. It dynamically reconfigures the pipeline into two distinct modes: Small-NTT Mode and EWM Mode. In Small-NTT mode, the CU streams coefficients between external DDR and the on-chip memories via AXI, and orchestrates the PU to perform small-size NTTs. Meanwhile, it applies the proposed TF management scheme (Alg. 1 and Alg. 2) to generate conflict-free TF addresses in the Secondary Memory, ensuring that each PE receives the correct twiddle factor every cycle. In EWM Mode with transpose, the CU coordinates the PEs to perform element-wise multiplication and then write the resulting coefficients to permuted locations. Specifically, it uses the conflict-free matrix transpose to map each output coefficient to (BI', BA') across the banked Primary Memory, thereby completing the coefficient transposition (reordering) on the fly and preparing a burst-friendly layout for the subsequent small-size NTTs.

6 Evaluation

6.1 Experiment Setup and Methodology

We implement RENTT in SystemVerilog and deploy it on an AMD Alveo U280 FPGA card [5], featuring 9024 DSP slices, 1304K LUTs, 2607K registers, 2016 instances of 36-kbit BRAM, and 960 instances of 288-kbit URAM. The design is synthesized using Synplify Premier 2024 and placed-and-routed using Vivado 2023, targeting 200 MHz for the core logic and 300 MHz for the I/O and DDR memory interface. We evaluate NTT sizes from 2^{20} to 2^{28} over a 256-bit prime field, matching the target ZKP setting. All results are obtained from on-board execution. Input coefficients and precomputed twiddle factors are preloaded into the DDR memory. We measure end-to-end latency from the time the input resides in off-chip memory to the time the output is written back to off-chip memory. Reported results exclude host-to-board transfers over PCIe.

We compare RENTT with three representative large-size NTT accelerators: PipeZK (ASIC), one of the earliest NTT accelerators targeting ZKP systems [36]; SAM (FPGA), which supports NTT sizes up to 2^{28} [14]; and HiFA (FPGA), which demonstrates strong resource efficiency [23]. We report (i) end-to-end latency in milliseconds and (ii) the area-time product (ATP), which mitigates the effect of device resource differences. Following prior FPGA studies [22, 31], we adopt weighting factors to aggregate heterogeneous resources into an equivalent area metric: $(\frac{\mathrm{NUM_{LUT}}}{16} + \frac{\mathrm{NUM_{FF}}}{8} + 102.4\times \mathrm{NUM_{DSP}} + 56 \times \mathrm{NUM_{BRAM}})/1000$. For the FPGA baselines, we take the reported latency and resource utilization from the corresponding papers [14, 23]. In particular, both our measurements and HiFA's baseline results are reported on AMD Alveo U280, while SAM reports results on AMD Alveo U250 [4]. Since U250 and U280 share identical on-chip architectural specifications (LUT/FF fabric, DSP, BRAM/URAM granularity), we treat them as comparable and focus on ATP to account for residual platform and frequency differences.

6.2 Comparison of Area-Time Product

Table 2 summarizes end-to-end latency and ATP across designs. At the largest setting ($N = 2^{28}$, 256-bit), RENTT completes one NTT in 1.52 s, while SAM requires 4.02 s, providing a 2.64× lower latency. More importantly, RENTT achieves a 4.21× lower ATP (682.88 vs. 2876.27), indicating substantially better performance density. Compared with SAM, RENTT uses significantly fewer DSPs on FPGA (3629 vs. 6776), consistent with our goal of eliminating on-the-fly twiddle generation overhead while maintaining scalable throughput under DDR-only bandwidth constraints. Compared with both SAM and HiFA, RENTT consumes more LUTs and FFs. We emphasize that LUTs and FFs are typically less scarce than DSPs on modern FPGAs, and the overall efficiency is better reflected by ATP: despite higher LUT and FF cost, RENTT achieves a lower ATP.

6.3 Cycle Counts and Latency Analysis

Table 3 breaks down RENTT latency into three steps. Increasing the PE count from 4 to 16 yields near-linear speedup. For $N = 2^{28}$, the total latency reduces from 6.07 s (4 PEs) to 3.03 s (8 PEs) and further to 1.52 s (16 PEs), closely tracking the ideal $1/T$ scaling. This scaling behavior indicates that RENTT sustains high PE utilization without stalling from twiddle-factor (TF) bank conflicts or coefficient-bank conflicts, matching the intent of our conflict-free TF management and banking-aware scheduling. We also observe that Step 1 and Step 3 contribute similarly (each 638 ms at 16 PEs for $N = 2^{28}$), while Step 2 is smaller (241 ms). The balanced Step 1/3 cost suggests that the column-wise phase does not suffer from burst-fragmented off-chip accesses; instead, the reordering fused into Step 2 effectively restores burst-contiguous streaming for Step 3, preventing the strided-column penalty.

RENTT exhibits near-linear scaling up in latency to 16 PEs for N ranging from 2^{20} to 2^{28}, suggesting that the design is not bottlenecked by on-chip bank conflicts in either coefficient access or TF supply. Under the DDR-only setting, the remaining limiting factor is the sustainable off-chip bandwidth and memory capacity: as N increases, the design inevitably performs full off-chip streaming of coefficients and (for Step 2) twiddle factors, making the end-to-end time increasingly sensitive to external memory throughput.

6.4 Resource Utilization Analysis

Table 4 reports FPGA resource utilization under different configurations of the PE parallelism T and the maximum supported NTT size $N_{\max}$. For a fixed $N_{\max} = 2^{28}$, the BRAM usage remains constant across T, since BRAM is dominated by the on-chip storage capacity for the target problem size (i.e., the coefficient and twiddle factor memory sized for $N_{\max}$) rather than the compute parallelism. This is

Design	Platform Type	Freq (MHz)	LUT (k)	FF (k)	BRAM	DSP	N	Latency (ms)	ATP (Improvement)
PipeZK [36]	UMC 28nm ASIC	300	-	-	-	-	2^{20}	11.00 (1.69×)	-
SAM [14]	Alveo U250 FPGA	100	593k	533k	2269	6776	2^{20}	12.61 (1.93×)	9.02 (3.31×)
							2^{24}	183.56 (2.14×)	131.34 (3.58 ×)
							2^{28}	4020 (2.65×)	2876.27 (4.21×)
HiFA [23]	Alveo U280 FPGA	220	300k	351k	737	2912	2^{20}	6.48 (0.99×)	1.95 (0.72×)
		165	380k	399k	986	3640	2^{24}	118.18 (1.38×)	44.72 (1.22×)
RENTT	Alveo U280 FPGA	200	895k	957k	384	3629	2^{20}	6.52 (1×)	2.73 (1×)
			942k	1007k	384	3629	2^{24}	85.71 (1×)	36.62 (1×)
			992k	1060k	512	3629	2^{28}	1517 (1×)	682.88 (1×)

Table 2: Latency comparison for 256-bit modulus across varying polynomial sizes. "–": not applicable.

#PE	N	*Step 1*		*Step 2*		*Step 3*		*Total*	
		Cycle	Latency	Cycle	Latency	Cycle	Latency	Cycle	Latency
4	2^{20}	6.69M	11.14	2.28M	3.80	6.69M	11.14	15.65M	26.09
	2^{24}	84.74M	141.24	36.21M	60.35	84.74M	141.24	205.70M	342.83
	2^{28}	1.53G	2552.00	579.14M	965.24	1.53G	2552.01	3.64G	6069.25
8	2^{20}	3.34M	5.57	1.14M	1.90	3.34M	5.57	7.83M	13.04
	2^{24}	42.37M	70.62	18.11M	30.18	42.37M	70.62	102.85M	171.42
	2^{28}	765.60M	1276.00	289.57M	482.62	765.60M	1276.00	1.82G	3034.62
16	2^{20}	1.67M	2.79	570K	0.95	1.67M	2.79	3.91M	6.52
	2^{24}	21.19M	35.31	9.05M	15.09	21.19M	35.31	51.42M	85.71
	2^{28}	382.80M	638.00	144.79M	241.31	382.80M	638.00	910.39M	1517.31

Table 3: Cycle counts and latencies (in milliseconds) of RENTT.

T	$N_{\max}$	LUT	FF	BRAM	DSP
4	2^{28}	250236	273302	512	917
8	2^{28}	493303	530063	512	1819
16	2^{28}	991675	1059591	512	3629
16	2^{24}	942091	1006611	384	3629
16	2^{20}	894986	956280	384	3629

Table 4: Comparison of resource utilization.

because our memory hierarchy decouples on-chip storage capacity from compute parallelism: the total capacity of coefficient and TF is fixed for $N_{\max}$, while increasing T is supported by banked access and scheduling without replicating the full storage.

In contrast, LUT, FF and DSP scale with T, because they are primarily consumed by the replicated compute datapaths (e.g., modular multipliers and per-PE control). As T increases from 4 to 16, LUT, FF and DSP rise accordingly, reflecting the additional parallel compute resources. When T is fixed (e.g., $T = 16$) and $N_{\max}$ decreases from 2^{28} to 2^{20}, BRAM reduces due to the smaller on-chip buffering requirement, while LUT, FF and DSP approximately remain unchanged because the computation modules are unchanged.

7 Conclusion

This paper presented RENTT, a resource-efficient accelerator for large-size NTTs. RENTT introduces three techniques: a conflict-free twiddle-factor management scheme, a conflict-free coefficient transpose scheme, and a configurable, decoupled memory-centric architecture. RENTT supports large-size NTTs for ZKP requirement and completes a 2^{28}-point NTT in 1.52 s with 16 PEs, while significantly reducing DSP usage compared with representative FPGA baselines. These results demonstrate that RENTT delivers configurable and parameterizable large-size NTT acceleration, achieving scalable performance with improved resource efficiency.

References

[1] Kaveh Aasaraai, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, Javier Varela, and Kevin Bowers. 2023. Cyclone-NTT: An NTT/FFT Architecture Using Quasi-Streaming of Large Datasets on DDR- and HBM-based FPGA Platforms. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable*

Gate Arrays, FPGA 2023, Monterey, CA, USA, February 12-14, 2023, Paolo Ienne and Zhiru Zhang (Eds.). ACM, 47. doi:10.1145/3543622.3573143

[2] Kasra Abbaszadeh, Christodoulos Pappas, Jonathan Katz, and Dimitrios Papadopoulos. 2024. Zero-Knowledge Proofs of Training for Deep Neural Networks. In *ACM CCS 2024*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM Press, 4316–4330. doi:10.1145/3658644.3670316

[3] Berkin Akin, Franz Franchetti, and James C Hoe. 2016. FFTs with near-optimal memory access through block data layouts: Algorithm, architecture and design automation. *Journal of Signal Processing Systems* 85, 1 (2016), 67–82.

[4] AMD, Inc. 2023. Alveo U250 Data Center Accelerator Card Data Sheet. https://docs.amd.com/r/en-US/ds962-u200-u250. DS962.

[5] AMD, Inc. 2023. Alveo U280 Data Center Accelerator Card Data Sheet. https://docs.amd.com/r/en-US/ds963-u280/Summary. DS963.

[6] David H Bailey. 1990. FFTs in external or hierarchical memory. *The journal of Supercomputing* 4 (1990), 23–35.

[7] Paul Barrett. 1987. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *CRYPTO'86 (LNCS, Vol. 263)*, Andrew M. Odlyzko (Ed.). Springer, Berlin, Heidelberg, 311–323. doi:10.1007/3-540-47721-7_24

[8] Eli Ben-Sasson, Iddo Bentov, Alessandro Chiesa, Ariel Gabizon, Daniel Genkin, Matan Hamilis, Evgenya Pergament, Michael Riabzev, Mark Silberstein, Eran Tromer, and Madars Virza. 2017. Computational Integrity with a Public Random String from Quasi-Linear PCPs. In *EUROCRYPT 2017, Part III (LNCS, Vol. 10212)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.). Springer, Cham, 551–579. doi:10.1007/978-3-319-56617-7_19

[9] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 459–474. doi:10.1109/SP.2014.36

[10] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. 2016. Interactive Oracle Proofs. In *TCC 2016-B, Part II (LNCS, Vol. 9986)*, Martin Hirt and Adam D. Smith (Eds.). Springer, Berlin, Heidelberg, 31–60. doi:10.1007/978-3-662-53644-5_2

[11] Juan Benet and Protocol Labs. 2017. Filecoin: A Decentralized Storage Network. White paper. https://filecoin.io/filecoin.pdf Version dated July 19, 2017.

[12] Bing-Jyue Chen, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang. 2024. Zkml: An optimizing system for ml inference in zero-knowledge proofs. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 560–574.

[13] Xiangren Chen, Bohan Yang, Wenping Zhu, Hanning Wang, Qichao Tao, Shuying Yin, Min Zhu, Shaojun Wei, and Leibo Liu. 2025. A High-performance NTT/MSM Accelerator for Zero-knowledge Proof Using Load-balanced Fully-pipelined Montgomery Multiplier. *IACR TCHES* 2025, 1 (2025), 275–313. doi:10.46586/tches.v2025.i1.275-313

[14] Wang Cheng and Gao Mingyu. 2023. SAM: A Scalable Accelerator for Number Theoretic Transform Using Multi-Dimensional Decomposition. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[15] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *EUROCRYPT 2020, Part I (LNCS, Vol. 12105)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Cham, 738–768. doi:10.1007/978-3-030-45721-1_26

[16] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.

[17] Alhad Daftardar, Brandon Reagen, and Siddharth Garg. 2024. SZKP: A Scalable Accelerator Architecture for Zero-Knowledge Proofs. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques* (Long Beach, CA, USA) *(PACT '24)*. Association for Computing Machinery, New York, NY, USA, 271–283. doi:10.1145/3656019.3676898

[18] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Report 2019/953. https://eprint.iacr.org/2019/953

[19] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In *EUROCRYPT 2013 (LNCS, Vol. 7881)*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer, Berlin, Heidelberg, 626–645. doi:10.1007/978-3-642-38348-9_37

[20] S Goldwasser, S Micali, and C Rackoff. 1985. The knowledge complexity of interactive proof-systems. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. 291–304.

[21] Zhenyu Guan, Yongqing Zhu, Yicheng Huang, Luchang Lei, Xueyan Wang, Hongyang Jia, Yi Chen, Bo Zhang, Jin Dong, and Song Bian. 2024. Esc-ntt: An elastic, seamless and compact architecture for multi-parameter ntt acceleration. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[22] Pengzhou He, Samira Carolina Oliva Madrigal, Çetin Kaya Koç, Tianyou Bao, and Jiafeng Xie. 2024. CASA: A Compact and Scalable Accelerator for Approximate Homomorphic Encryption. *IACR TCHES* 2024, 2 (2024), 451–480. doi:10.46586/tches.v2024.i2.451-480

[23] Qilin Hu, Haotian Wang, Chubo Liu, Keqin Li, and Kenli Li. 2025. HiFA: A High-Performance and Flexible Acceleration Framework for Large-Size Number Theoretic Transform. *ACM Transactions on Reconfigurable Technology and Systems* 18, 4 (2025), 1–32.

[24] Tao Lu, Yuxun Chen, Zonghui Wang, Xiaohang Wang, Wenzhi Chen, and Jiaheng Zhang. 2025. BatchZK: A Fully Pipelined GPU-Accelerated System for Batch Generation of Zero-Knowledge Proofs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 100–115.

[25] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 2023. Gzkp: A gpu accelerated zero-knowledge proof system. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 340–353.

[26] John M Pollard. 1971. The fast Fourier transform in a finite field. *Mathematics of computation* 25, 114 (1971), 365–374.

[27] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. 2014. Compact Ring-LWE Cryptoprocessor. In *CHES 2014 (LNCS, Vol. 8731)*, Lejla Batina and Matthew Robshaw (Eds.). Springer, Berlin, Heidelberg, 371–391. doi:10.1007/978-3-662-44709-3_21

[28] Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. 2024. Accelerating zero-knowledge proofs through hardware-algorithm co-design. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 366–379.

[29] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy*. IEEE, 459–474.

[30] Louis Tremblay Thibault and Michael Walter. 2024. Towards Verifiable FHE in Practice: Proving Correct Execution of TFHE's Bootstrapping using plonky2. Cryptology ePrint Archive, Report 2024/451. https://eprint.iacr.org/2024/451

[31] Yazheng Tu, Pengzhou He, Çetin Kaya Koç, and Jiafeng Xie. 2023. LEAP: Lightweight and efficient accelerator for sparse polynomial multiplication of HQC. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 31, 6 (2023), 892–896.

[32] Cheng Wang and Mingyu Gao. 2025. UniZK: Accelerating Zero-Knowledge Proof with Unified Hardware and Flexible Kernel Mapping. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1101–1117.

[33] André Weimerskirch and Christof Paar. 2006. Generalizations of the Karatsuba Algorithm for Efficient Implementations. Cryptology ePrint Archive, Report 2006/224. https://eprint.iacr.org/2006/224

[34] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. 2022. zkbridge: Trustless cross-chain bridges made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 3003–3017.

[35] Zhengbang Yang, Lutan Zhao, Peinan Li, Han Liu, Kai Li, Boyan Zhao, Dan Meng, and Rui Hou. 2025. LegoZK: A Dynamically Reconfigurable Accelerator for Zero-Knowledge Proof. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 113–126.

[36] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. 2021. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 416–428.

[37] Hao Zhou, Changxu Liu, Lan Yang, Li Shang, and Fan Yang. 2025. ReZK: A Highly Reconfigurable Accelerator for Zero-Knowledge Proof. *IEEE Trans. Circuits Syst. I Regul. Pap.* 72, 2 (2025), 802–815. doi:10.1109/TCSI.2024.3468033