GPU-Accelerated Number Theoretic Transform-Based Privacy Amplification for Quantum Key Distribution

Chenyu Wang[1], Kazuaki Doi[1], and Yutaro Ishigaki[1]

Corporate Laboratory, Toshiba Corporation, Japan [1]
chenyu.wang.t37@mail.toshiba
kazuaki.doi.f50@mail.toshiba
yutaro.ishigaki.h62@mail.toshiba

Abstract. Privacy Amplification (PA) constitutes a critical computational bottleneck in high-rate Quantum Key Distribution (QKD) systems, particularly when processing large data blocks required to mitigate finite-size security effects. In this work, we propose a high-performance GPU implementation based on the Number Theoretic Transform (NTT), enabling exact modular arithmetic under a suitable modulus and root of unity for large-scale privacy amplification. We optimize the NTT execution on GPUs by integrating a *hybrid butterfly computation scheme* that combines a warp-shuffle-based approach and a shared-memory-based approach, together with kernel fusion techniques and Barrett-based modular reduction to maximize memory bandwidth utilization and parallel efficiency. Experimental results on an NVIDIA L40 GPU demonstrate a throughput of **3.32 Gbps** for 2^{27}-bit input blocks. This result indicates that software-based NTT acceleration on commodity GPUs can support hundreds-of-Mbps-class Secret-Key-Rate (SKR) QKD systems.

Keywords: Quantum Key Distribution · Privacy Amplification · Number Theoretic Transform · GPU · Parallel Computing

1 Introduction

Quantum key distribution (QKD) has matured from theoretical protocols into a viable technology for guaranteeing information-theoretic security in critical infrastructure [18, 24]. QKD systems are evolving toward higher secret-key rates and longer transmission distances, which in turn demand efficient post-processing pipelines to handle an increasing volume of raw key data [7]. A critical component of this pipeline is privacy amplification (PA), which distills a shorter secure key from the corrected data by applying a universal hash function [12, 19]. In practice, PA compresses a corrected key of length n to a secret key of length r, where the ratio r/n is set by parameter estimation and finite-size security analysis. For example, in a high-rate discrete-variable QKD (DV-QKD) demonstration, Yuan *et al.* report a PA compression ratio of about $r/n \simeq 0.29$ (typical QBER $\approx 3\%$ and 10^8-bit PA blocks) [27]. Accordingly, sustaining a target secret-key rate

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

(SKR) typically requires PA throughput on the order of SKR/(r/n) (e.g., $\sim 3\times$ headroom when $r/n \approx 0.3$). As system-level raw-key generation capabilities grow through higher pulse/symbol rates as well as parallelization and multiplexing, this throughput requirement can make PA a computational bottleneck due to the large-scale linear operations involved.

Besides DV-QKD based on single-photon detection, continuous-variable QKD (CV-QKD) has also been actively studied as an alternative approach using coherent detection and Gaussian modulation, with comprehensive treatments of practical implementations and security analysis available in the literature [8]. While very long transmission distances typically lead to reduced key rates, recent short-reach, chip-scale demonstrations highlight that post-processing throughput can become critical even for CV-QKD: Ng *et al.* report a secret-key rate of 1.213 Gbit/s over 10 km using an integrated photonic-chip QKD system based on discrete-modulated CV-QKD [13]. Such gigabit-class operation implies that PA must sustain multi-Gbps-class throughput to keep pace with the target SKR under realistic compression ratios and protocol/channel-dependent overheads. In this work, we focus on DV-QKD privacy amplification, although the core acceleration techniques are directly applicable whenever PA is instantiated via large-block Toeplitz hashing, regardless of DV/CV QKD.

In many practical implementations, PA is instantiated by Toeplitz hashing, which reduces to a cyclic convolution suitable for transform-based evaluation [6, 9], with finite-size analyses often motivating the use of large blocks (e.g., $\sim 10^8$ bits) to approach asymptotic key rates [11]. To accelerate Toeplitz-based PA at such scales, transform methods based on the fast Fourier transform (FFT) have been explored on heterogeneous platforms, including programmable hardware such as FPGAs [9, 10]. However, floating-point arithmetic can complicate bit-exact coefficient recovery at large transform lengths and may require higher precision or correction steps [23]. Integer-domain alternatives, such as GMP-based schemes [26] and NTT-based constructions, avoid these rounding issues by performing the transform in $\mathbb{Z}_p$ with exact modular arithmetic [17], making NTT a natural choice for deterministic, bit-exact Toeplitz-hash PA.

Achieving high throughput for large-scale NTT on general-purpose hardware still requires careful attention to memory traffic and efficient modular arithmetic. While FPGA- and CPU-based PA accelerators have been actively studied [9, 5, 21], modern graphics processing units (GPUs) offer a compelling alternative as a programmable many-core platform with high memory bandwidth. When the data layout and access patterns are carefully co-designed with the GPU memory hierarchy, GPUs can deliver high effective throughput while avoiding the development cost and rigidity typically associated with FPGA designs.

In this paper, we present a GPU-accelerated PA implementation based on NTT, optimized for large-scale workloads motivated by finite-size considerations. Our design employs: (i) an NTT-friendly data layout to ensure coalesced global-memory access and minimize redundant transfers; (ii) a hybrid butterfly computation scheme that leverages warp-level communication and shared-memory reuse; (iii) kernel fusion strategies specifically designed to mitigate kernel launch

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

overhead and eliminate intermediate global-memory round-trips between NTT stages; and (iv) a Barrett-based modular reduction strategy to accelerate modular multiplication without expensive integer division. With these optimizations, our implementation achieves **3.32 Gbps** (40.382 ms) on an NVIDIA L40 GPU and **1.62 Gbps** (82.766 ms) on an NVIDIA RTX 3080. These results demonstrate that software-based NTT acceleration on modern GPUs can provide a practical high-throughput PA building block for high-rate QKD systems, and can support hundreds-of-Mbps-class SKR QKD systems under typical post-processing overheads.

2 Background and Related Work

This section reviews Toeplitz-hash-based privacy amplification and its acceleration via transform methods. We outline the formulation of Toeplitz hashing through a circulant embedding that enables fast cyclic-convolution evaluation, and we contrast FFT- and NTT-based approaches with emphasis on their arithmetic properties.

2.1 Toeplitz Hashing for Privacy Amplification

PA is commonly instantiated via a linear universal$_2$ hash family. Toeplitz hashing is widely adopted in practical QKD implementations due to its compact seed representation and efficient streaming capabilities. Let $\mathbf{x} \in \{0,1\}^n$ denote the reconciled key and $\mathbf{y} \in \{0,1\}^r$ the compressed key. A Toeplitz matrix $\mathbf{T} \in \{0,1\}^{r\times n}$ is defined by a seed $\mathbf{s} = (s_0, \ldots, s_{n+r-2}) \in \{0,1\}^{n+r-1}$ such that $T_{i,j} = s_{\,i-j+(n-1)}$ for $0 \le i < r$ and $0 \le j < n$. The PA output is computed as

$$\mathbf{y} = \mathbf{T}\mathbf{x} \bmod 2. \tag{1}$$

This matrix–vector multiplication is a structured linear map. By embedding the $r \times n$ Toeplitz operator into an $N \times N$ circulant matrix, where N denotes the transform length used for the circulant embedding and is chosen such that $N \ge n + r - 1$ (typically as a power of two), the computation in (1) can be evaluated via an N-point cyclic convolution [6].

This reduction lowers the complexity from $O(nr)$ to $O(N \log N)$, where N is the transform length introduced above, by utilizing fast transform algorithms.

2.2 Transform-Based Acceleration: FFT and NTT

To compute the cyclic convolution efficiently, the fast Fourier transform (FFT) is conventionally employed. FFT-based PA accelerates the computation but operates in complex floating-point arithmetic. Since the inputs are binary and the desired output is binary (mod 2) whereas the underlying convolution coefficients are integer-valued, floating-point rounding can complicate bit-exact integer recovery at large transform sizes. Ensuring correctness may require higher-precision data types and/or additional correction steps, introducing extra overhead [23].

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

The number theoretic transform (NTT) provides an integer-domain alternative by operating over the finite field $\mathbb{Z}_p$ for a prime modulus p. To ensure exact recovery of the linear convolution result, the modulus p must satisfy $p > N$. NTT shares the same staged structure and data permutations as FFT. Specifically, the core radix-2 butterfly operation for inputs u, v and twiddle factor W is defined as:

$$u' = (u + Wv) \bmod p, \qquad v' = (u - Wv) \bmod p, \tag{2}$$

replacing floating-point arithmetic with modular integer operations. Consequently, both FFT and NTT are bandwidth-sensitive for large N due to repeated data movement across $\log_2 N$ stages. However, high-throughput NTT implementations must additionally optimize modular multiplication and reduction (e.g., division-free reduction) to fully utilize general-purpose hardware such as GPUs [17].

Prior work has reported that FFT-based PA pipelines can suffer from floating-point rounding issues when recovering integer-valued convolution coefficients at large transform sizes. In particular, Wang *et al.* observed that single-precision FFT may fail to produce bit-exact results beyond a certain scale in their evaluated setting, and that ensuring correctness can require higher precision and/or additional correction steps, reducing throughput [23]. Motivated by these numerical limitations, we adopt an NTT-based approach that performs the transform entirely in modular integer arithmetic, thereby avoiding floating-point rounding in the transform and enabling deterministic, bit-exact computation under a suitably chosen modulus.

2.3 Prior Acceleration Works for Large-Scale PA and NTT

A substantial body of work has investigated accelerating Toeplitz-based PA on heterogeneous platforms. FFT-based PA has been implemented on FPGA using long-FFT strategies and dedicated FFT cores [9], and CPU implementations have demonstrated the feasibility of large-block processing with transform acceleration and careful engineering [21]. In continuous-variable QKD, Wang *et al.* proposed a high-speed implementation of length-compatible PA, emphasizing practical constraints and throughput-oriented design choices [23]. To address numerical reliability and exactness demands, other studies introduced multi-precision components, such as GMP-based schemes, to mitigate limitations of floating-point pipelines [26]. Recent large-scale FPGA work further discussed design trade-offs for large-block PA implementations, including numerical and resource considerations in transform-based Toeplitz multiplication, and reported the associated resource/design trade-offs [5].

Alongside PA-specific studies, the broader literature on fast NTT for GPUs provides implementation insights that are directly relevant to high-throughput modular transforms. Özcan and Sava present GPU-oriented NTT algorithms and discuss optimization considerations for performance on CUDA-capable devices [17]. These works collectively motivate NTT-based PA on GPUs: NTT preserves exact arithmetic while enabling aggressive parallelization via butterfly

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

stages, and performance depends critically on data layout, memory traffic, and efficient modular reduction.

3 Proposed GPU-Based Privacy Amplification Scheme

This section specifies how we realize large-block Toeplitz-hash PA on GPUs using an NTT-based pipeline and GPU-oriented kernels. We first summarize the end-to-end PA computation flow at the algorithm level, and then describe two core implementation ingredients: (i) a hybrid on-chip butterfly operation strategy (Section 3.2), and (ii) a kernel-fused 9-step (3D-decomposed) NTT/INTT design tailored to 2^{27}-point workloads (Section 3.3). Throughout, we keep arithmetic exact in $\mathbb{Z}_p$ and reduce global-memory traffic via fused scaling and implicit data reorders.

3.1 Overall PA Computation Flow

We consider a single PA block and start from the Toeplitz-hash definition in (1). Using the standard Toeplitz-to-circulant embedding viewpoint from structured linear algebra, the Toeplitz matrix–vector product can be mapped to a cyclic convolution after suitable padding and index reversal, enabling transform-domain acceleration in quasi-linear time [6].

In our implementation, the target large-block setting motivates a power-of-two transform length, and we focus on the 2^{27}-point regime. In this paper, we target PA blocks at the 2^{27}-bit scale and set the transform length to $N = 2^{27}$ points after Toeplitz-to-circulant embedding (with $N \geq n + r - 1$), where n denotes the length of the reconciled input block (in bits) and r denotes the length of the compressed output key (in bits). We pack the seed and input bits into two length-N sequences $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_p^N$ by mapping bits to $\{0, 1\} \subset \mathbb{Z}_p$, following standard transform-based Toeplitz-hash PA constructions [9, 21]. Concretely, we form $\mathbf{a}$ from the Toeplitz seed and $\mathbf{b}$ from the input block by applying the standard Toeplitz-to-circulant embedding (index reversal and zero-padding to length N), so that the resulting cyclic convolution corresponds to the desired Toeplitz multiplication on the extracted output segment. The cyclic convolution in $\mathbb{Z}_p$ is then evaluated via the NTT convolution theorem:

$$\mathbf{c} = \mathrm{INTT}_p\Big(\mathrm{NTT}_p(\mathbf{a}) \odot \mathrm{NTT}_p(\mathbf{b})\Big) \in \mathbb{Z}_p^N, \tag{3}$$

where $\odot$ denotes element-wise multiplication in $\mathbb{Z}_p$. We will refer to (3) as the *NTT-based PA core* in the remainder of this section. Finally, the PA output bits are obtained by extracting the prescribed entries of $\mathbf{c}$ implied by the embedding and applying the mod-2 reduction (and unpacking for word-level representations), following standard transform-based Toeplitz-hash PA constructions [9, 21].

Adaptation to other block lengths. Although we target the 2^{27}-point regime for throughput and finite-size motivation, the same flow extends to any

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

length by choosing a power-of-two transform size $N \geq n + r - 1$ and padding with zeros as required by the embedding. When N differs from 2^{27}, the GPU kernels can follow the same staged butterfly structure in (2); the only change is the decomposition/mapping strategy described in Section 3.3.

3.2 Hybrid butterfly operation on GPUs

This subsection describes how the staged radix-2 butterflies are executed in our GPU NTT kernel. Each stage applies the standard modular butterfly in (2) with a stage-dependent twiddle factor. Our goal is to realize the required pairwise exchange and modular arithmetic efficiently on GPU hardware while preserving exactness in $\mathbb{Z}_p$.

Schedule and ordering (DIT, with DIF also applicable). Our current implementation follows an iterative in-place *decimation-in-time (DIT)* schedule [22]. As is standard for iterative DIT realizations, the permutation required to obtain outputs in natural order (e.g., bit-reversal) is applied *when supplying inputs to the butterfly pipeline*: we read the data from global memory and perform the required index permutation in the global-memory load path before the first butterfly stage. After this permuted load, butterfly stages are executed in increasing stage order. Importantly, the *hybrid execution principle* below does not rely on DIT specifically: a decimation-in-frequency (DIF) realization preserves the same staged radix-2 dependency structure and can adopt the same *register-first / shared-memory-later* split, with appropriate indexing and permutation choices.

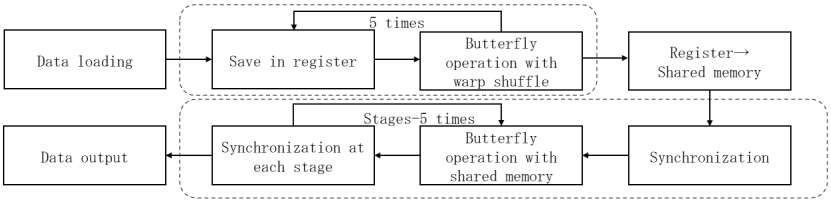

Fig. 1. Overview of the proposed hybrid butterfly operation on GPUs. Early stages are executed in a warp-synchronous manner with register-resident data and warp-shuffle exchanges, while later stages switch to shared-memory staging with stage-wise synchronization.

Stage-wise hybridization: warp-synchronous vs. shared-memory execution. Figure 1 illustrates the key observation: the butterfly partner distance grows exponentially with stage index. Early stages exchange data within a warp, whereas later stages require communication beyond warp scope. We therefore adopt a hybrid policy: (i) *early stages:* keep the *data coefficients* (i.e., the input/intermediate values u and v in (2)) in registers and use warp-level exchange to obtain butterfly partners, while applying the corresponding stage-dependent twiddle factor W in (2), thereby avoiding shared-memory traffic and block-wide

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

synchronization; and (ii) *later stages:* stage intermediate data in shared memory and execute remaining butterflies in-place with stage-wise synchronization.

Warp-synchronous execution in early stages. In early stages, coefficients are kept register-resident and each thread obtains its butterfly partner via warp-level exchange, then applies (2). This avoids intermediate shared-memory stores and block-wide synchronization as long as the partner mapping remains intra-warp. Specifically, the register and shuffle approach applies while the butterfly partner mapping remains within a warp (i.e., stride ≤ 16 for a 32-thread warp). Once the mapping exceeds warp scope, we switch from register-based shuffles to shared memory to support inter-warp communication.

Modular arithmetic: division-free multiplication and bounded normalization. The dominant arithmetic cost in (2) is the modular product between a twiddle factor and a coefficient. We implement modular multiplication using a Barrett-style reduction with a precomputed constant

$$\mu \triangleq \left\lfloor \frac{2^{64}}{p} \right\rfloor, \tag{4}$$

so that for a 64-bit product $z = a \cdot b$, an approximate quotient can be obtained from the high half of $z \cdot \mu$, and the remainder is formed as

$$r = z - qp, \qquad q \approx \left\lfloor \frac{z}{p} \right\rfloor. \tag{5}$$

After a small fixed number of conditional subtractions, the result is normalized to $[0, p)$ and equals the exact modular product, while avoiding integer division [2, 4]. Similarly, the butterfly add/subtract updates are kept within a bounded range via lightweight conditional corrections so that values remain in $[0, p)$ throughout the stages.

Branchless butterfly selection and uniform control flow. Within the warp-synchronous phase, each butterfly operates on a partner pair (u, v) and produces two outputs (u', v') as in (2). When we implement the early stages using warp-level exchange, a given lane can obtain both input values (its own value and the partner value) and can therefore compute *both* candidates u' and v'. However, each lane must finally write back only one of them: lanes belonging to the "upper" half of a butterfly write u', whereas lanes in the "lower" half write v' according to the stage-dependent partner mapping. To keep control flow uniform, we avoid an explicit `if/else` branch per lane and instead select between u' and v' using predication (e.g., conditional moves or bit-mask selection). This branchless selection suppresses warp divergence and maintains a single, uniform instruction path across all lanes in the warp.

Register reuse via multi-coefficient processing (vectorized streams). As illustrated in Fig. 2, during the register-resident phase each thread holds multiple coefficients and advances these streams in parallel using warp-shuffle-based partner exchange in the early butterfly stages. In the register-resident phase, each thread holds a small vector of coefficients in registers and updates them in lockstep across stages. A single thread therefore advances multiple independent

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

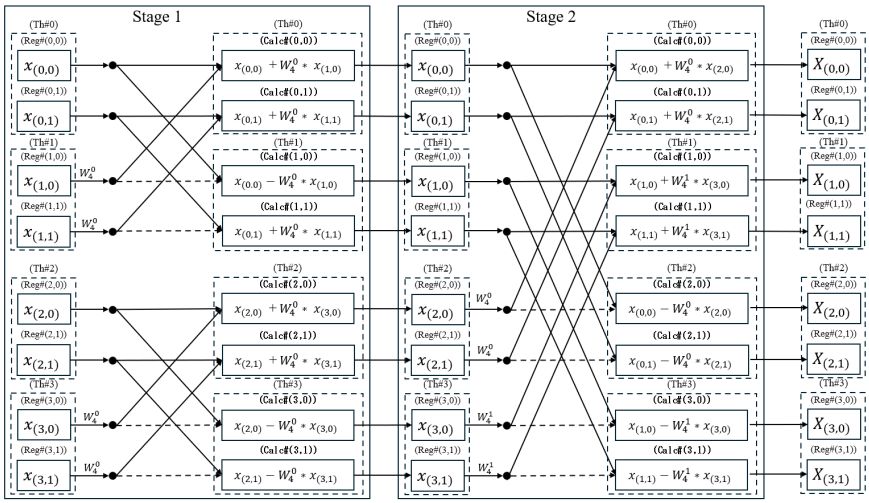

Fig. 2. Illustrative example for the register-resident phase with multi-coefficient processing per thread. Early-stage butterflies are evaluated warp-synchronously using warp shuffles.

coefficient streams simultaneously, rather than processing only one scalar coefficient at a time. In our implementation, we configure each thread to process a vector of four elements (four streams) in parallel, which naturally aligns with 128-bit global memory transactions. We note that this degree of vectorization is a tunable parameter; while four is optimal for the hardware tested, the vector width should be adjusted according to the available register file size and instruction throughput characteristics of the specific target GPU architecture.

At each early stage, the thread obtains the butterfly partner for each register value via a warp shuffle and applies the same radix-2 modular butterfly update in (2). The stage-dependent twiddle factor is reused across the per-thread vector elements at that stage, which amortizes twiddle access and control overhead while increasing instruction-level parallelism. When the data layout permits, these per-thread vector elements are also moved with vectorized global-memory transactions, improving effective bandwidth.

The vectorized multi-stream execution preserves correctness: it is equivalent to running the same butterfly schedule independently for each stream, but with multiple streams advanced together within a thread.

Shared-memory staging with conflict-mitigated layout. In later stages, butterfly partners reside in different warps, so intermediate values must be exchanged at block scope. We therefore stage a working set in shared memory as a 2-D tile that is repeatedly read and updated by the remaining butterfly stages. A direct row-major layout can lead to shared-memory bank conflicts when threads access strided partner locations. To mitigate this effect, we use a *padded* leading dimension, storing the tile as `tile[T][T+1]` rather than `tile[T][T]`. Here, `T` denotes the per-block tile extent (in elements) along each shared-memory dimension, i.e., the kernel stages a $T \times T$ workspace per thread block to hold the

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

intermediate NTT coefficients for a subproblem. This padding adds one dummy element per row, so that consecutive rows start at different bank alignments, and the common strided access patterns map more evenly across banks. Importantly, this padding changes only the physical placement in shared memory; the logical indices and the in-place butterfly updates remain exactly those of (2).

3.3 Kernel-Fused 9-Step NTT for 2^{27}-Point PA

This subsection describes our kernel-fused 9-step realization of the $N = 2^{27}$ NTT/INTT used in the NTT-based PA core (3). We first clarify the transform length and data layout, then outline the 9-step 3D factorization and its GPU-oriented advantages, followed by the fused-kernel mapping and the role of the inverse scaling.

Transform length and data layout. As introduced in Section 3.1, Toeplitz-hash PA maps a reconciled key $\mathbf{x} \in \{0,1\}^n$ to an r-bit output $\mathbf{y} \in \{0,1\}^r$ via a Toeplitz matrix, and the standard Toeplitz-to-circulant embedding requires a transform length N satisfying $N \geq n + r - 1$. In our NTT-based implementation we fix $N = 2^{27}$ so that a single 2^{27}-point transform covers our target PA block sizes. On the GPU, this 1D length-N array is stored as a logical matrix in global memory, which matches the row-major access pattern in our CUDA kernels while still representing one 2^{27}-point transform. The modulus p is a 32-bit NTT prime that admits a primitive 2^{27}-th root of unity, enabling an exact radix-2 NTT/INTT in $\mathbb{Z}_p$ [16]. Throughout, input bits are mapped to $\{0,1\} \subset \mathbb{Z}_p$ and padded to length N as required by the embedding, so the GPU kernels operate on a length-N field-valued vector.

9-step (3D) factorization for $N = 2^{27}$. We adopt a 3D factorization that is standard in large FFT/NTT implementations [1, 14, 15]. At the algorithmic level, the 2^{27}-point 1D vector is treated as a $512 \times 512 \times 512$ tensor, and the global transform is evaluated by three kernels of 512-point NTTs (one round per tensor dimension), with twiddle-factor multiplications between rounds. In our GPU implementation, we do not materialize an explicit 3D array in memory; instead, the data are stored as a 1D vector and accessed through a 2D view of shape $512 \times (512^2)$ with appropriate index mapping. This 2D representation is an implementation choice for address generation and memory coalescing, while the computation still follows the same 3D factorization schedule.

The first two kernels each apply a 512-point NTT to one tensor dimension, followed by a twiddle-factor multiplication, and then an axis permutation implemented as an implicit reorder in the same kernel. The third kernel applies the final 512-point NTT and writes the result back in 1D order. Figure 3 summarizes this 9-step schedule at a high level. Mathematically, it is a regrouping of the same radix-2 NTT computation as in (2); it preserves exactness in $\mathbb{Z}_p$ while improving locality and global-memory access regularity on the GPU.

On GPUs, this 3D factorization is particularly attractive: each 512-point subtransform is mapped to a thread block, enabling on-chip execution with the hybrid butterfly scheme from Section 3.2, while transpose-like steps are expressed as structured index re-mappings rather than irregular long-stride accesses.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

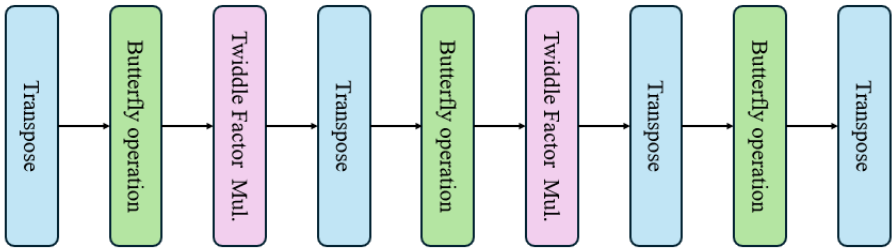

Fig. 3. Kernel-fused 9-step NTT pipeline for a 2^{27}-point transform. The input vector is reshaped into a 512×512^2 tensor. In our implementation, the nine logical operations are executed using three fused kernels: (Kernel 1) `Transpose`–`Butterfly`–`Twiddle`–`Transpose`, (Kernel 2) `Butterfly`–`Twiddle`–`Transpose`, and (Kernel 3) `Butterfly`–`Transpose`. The transpose-like steps are *implicit* reorders realized via output addressing (no standalone transpose kernels).

GPU kernels: fusion and implicit reorders. A naive implementation would realize each of the nine logical steps in Fig. 3 as a separate kernel, causing repeated global read/write passes and high kernel-launch overhead. We instead fuse the pipeline into three kernels per transform. Each fused kernel combines one 512-point subtransform with the required twiddle multiplication/Transpose step, and realizes the transpose-like reorder *implicitly* by writing outputs to the addresses of the next logical layout, thereby avoiding standalone transpose kernels and reducing intermediate global-memory traffic [17].

4 Experimental Evaluation

This section evaluates the proposed GPU-accelerated NTT-based PA implementation at the 2^{27}-bit scale (i.e., on the order of 10^8 bits). We first summarize the experimental setup and then compare throughput against representative large-block PA implementations reported in the literature.

4.1 Experimental Setup

All kernels were implemented in CUDA 12.2 and compiled with aggressive optimization flags (e.g., `-O3`) while preserving exact modular arithmetic in $\mathbb{Z}_p$. GPU execution time is measured by `cudaEvent` and reported as *GPU kernel time* for the PA core (two forward NTTs, pointwise multiplication, and one inverse NTT). Unless otherwise stated, host–device transfers are excluded, as practical QKD pipelines can overlap communication and computation.

We evaluate two NVIDIA GPUs listed in Table 1. The RTX 3080 represents a high-end Ampere GPU, while the L40 is a data-center Ada GPU with substantially larger memory capacity and on-chip resources.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

Table 1. GPU platforms used in our experiments.

GPU	Arch.	CUDA cores	Mem. [GB]	Mem. type	BW [GB/s]
RTX 3080	Ampere (GA102)	8,960	12	GDDR6X	912
L40	Ada (AD102)	18,176	48	GDDR6	864

Table 2. Forward NTT latency under the BLS12-377 prime modulus. Baselines (Merge-NTT) are taken from [17]; values are in μs. Our time reports the total GPU kernel time of one forward NTT (sum of fused kernels when applicable), averaged over 100 runs after warm-up and measured by `cudaEvent`.

$\log_2 N$	This work [μs]		Ref. [17] [μs]	
	RTX 3080	L40	RTX 4090	A100
18	62.848	**49.888**	64.040	120.060
21	243.456	**119.680**	424.250	854.360
24	2305.908	**927.968**	4178.860	7294.050
27	19431.763	**10204.768**	36780.500	65097.40

4.2 NTT Kernel Result Comparison

We compare the latency of one *forward NTT* computation under the BLS12-377 prime modulus, measured as GPU kernel time. This kernel-level comparison uses the BLS12-377 modulus to match the setting of [17] for microbenchmarking; our end-to-end PA pipeline uses a 32-bit NTT prime as described in Section 3.3. Our measurements on RTX 3080 and L40 are contrasted with the Merge-NTT baselines reported for A100 and RTX 4090 in [17]. We compare latencies at the same transform lengths under the same prime modulus. Table 2 shows that latency increases with $\log_2 N$ for all platforms. This is consistent with staged radix-2 NTT execution, where larger N implies a greater number of butterfly stages and a larger working set.

Comparison against Merge-NTT baselines. Table 2 compares our forward-NTT latency under the same modulus setting with the Merge-NTT baselines reported in [17]. Across all tested transform sizes, our L40 implementation achieves the lowest latency among the compared platforms. This observation is notable because overall GPU capability is not determined by a single peak metric: different devices (RTX 3080, L40, RTX 4090, A100) exhibit distinct trade-offs in memory bandwidth, cache hierarchy, and on-chip resource limits, which can dominate the performance of bandwidth-sensitive NTT kernels.

For large N, an iterative radix-2 NTT becomes increasingly constrained by *data movement* rather than raw integer throughput. Each stage requires reading and writing a large working set and performing regular partner exchanges; as N grows, the working set quickly exceeds on-chip storage, making effective global-memory bandwidth and synchronization overhead key determinants of end-to-end latency. In this regime, optimizations that reduce global-memory round-trips and avoid unnecessary barriers can outweigh differences in peak compute throughput. In addition, larger on-chip caches and higher sustained memory bandwidth can improve effective reuse of twiddles and the working set, which may further benefit large-scale NTT execution.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

Our kernel is explicitly designed for this large-N behavior. First, the hybrid butterfly execution keeps early stages warp-synchronous with register-resident data and warp shuffles, reducing shared-memory staging and block-wide synchronization until inter-warp exchanges become unavoidable (Section 3.2). Second, in the 2^{27}-scale pipeline, the 9-step decomposition and kernel fusion reduce global-memory traffic by merging local NTT computation with the required scaling and reorder steps, where transpose-like reorders are realized by address remapping at store time rather than by separate transpose kernels. These choices directly target the dominant costs at large N and improve effective bandwidth utilization.

We therefore interpret the results in Table 2 as evidence that, under a matched modulus setting, the proposed implementation achieves higher *kernel-level efficiency* than the Merge-NTT baselines in [17], with the advantage becoming more pronounced at larger transform sizes where memory traffic and synchronization costs dominate. In this sense, the efficiency of the proposed approach is closely related to the GPU type considered. Because the implementation is dominated by large-scale staged data movement, its realized efficiency depends primarily on architectural features such as sustained memory bandwidth, cache capacity, shared-memory/register availability, and the efficiency of warp-synchronous data exchange, rather than on peak arithmetic throughput alone. GPUs with stronger support for these data-movement-critical aspects can better exploit the hybrid butterfly schedule and the fused 9-step pipeline, especially at large transform sizes.

4.3 PA Kernel Result Comparison

Table 3 summarizes throughput results for large-block PA around the 10^8-bit regime. In this table, 10^8 and $2^{27} \approx 1.34 \times 10^8$ are treated as the same order-of-magnitude scale for comparison. We also explicitly mark the primary acceleration/arithmetic approach of each work. Specifically, the listed approaches include FFT-based Toeplitz hashing, NTT-based PA designs, and other non-transform constructions such as GMP-based modular hashing and hybrid-hash families. Note that different works may instantiate PA with different universal hash constructions.

Comparison with CPU and FPGA Baselines. As detailed in Table 3, implementations on CPUs and FPGAs face a significant performance bottleneck at the 10^8-bit block size, with throughputs consistently plateauing around the 0.1 Gbps mark. This limitation appears platform-bound rather than algorithm-bound: whether employing FFT [21], NTT [20], or hybrid modular hashing [26, 25], CPU-based solutions yield results below 0.15 Gbps. Similarly, the FPGA-based DM3H design [5] operates in the same regime (0.125 Gbps). In sharp contrast, our GPU-based approach leverages massive parallelism to bridge this gap, providing substantially higher throughput than the representative CPU/FPGA results listed in Table 3, noting that platforms and implementation settings differ across studies.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

Table 3. Large-block PA implementations around the 10^8-bit regime. Throughput is given in Gbps. The method labels "DM3H" [5] and "MMH-MH" [25] follow the terminology used in the respective works and denote NTT-accelerated *modular-hashing*-based PA constructions.

Work	Plat.	Method	Block size [bits]	Thr. [Gbps]
Takahashi et al. [20]	CPU	NTT	10^8	0.109
Yan et al. [26]	CPU	GMP	10^8	0.140
Yan et al. [25]	CPU	MMH-MH	10^8	0.140
Tang et al. [21]	CPU	FFT	10^8	0.071
Cheng et al. [5]	FPGA	DM3H	10^8	0.125
Wang et al. [23]	GPU	FFT	10^8	1.35
Nico et al. [3]	GPU	FFT	10^8	3.45
This work (RTX 3080)	GPU	**NTT**	10^8	**1.62**
This work (L40)	GPU	**NTT**	10^8	**3.32**

Comparison with GPU-based PA. Table 3 compares our work with existing large-block PA implementations. Regarding GPU-based approaches, Wang *et al.* reported 1.35 Gbps on an NVIDIA K80 using FFT [23], and more recently, Nico *et al.* demonstrated 3.45 Gbps on an NVIDIA RTX 3080, also utilizing an FFT-based pipeline [3]. While these FFT-based schemes achieve high raw throughput, they rely on floating-point arithmetic. As discussed in Section 2.2 and noted in [23], ensuring bit-exact recovery of discrete coefficients becomes increasingly precarious as transform sizes scale to the 10^8-bit regime, which requires careful numerical safeguards to guarantee bit-exact recovery at large transform lengths (e.g., higher precision and/or correction steps as discussed in [23]). In contrast, our pipeline uses an NTT-based construction that performs the transform entirely in $\mathbb{Z}_p$ and therefore avoids floating-point rounding in the transform by design. To the best of our knowledge, our work achieves the highest reported throughput among NTT-based large-block PA implementations under comparable block sizes and exact-arithmetic requirements. We demonstrate that the numerical stability of NTT can be achieved with multi-Gbps performance suitable for real-time systems, effectively bridging the gap between exact arithmetic and high-speed processing.

5 Conclusion

We presented a GPU-accelerated PA design for QKD based on NTTs, targeting the large-block regime motivated by finite-size considerations. By executing all transform-domain operations in $\mathbb{Z}_p$, the proposed pipeline provides bit-exact modular arithmetic by construction, avoiding the numerical rounding risks that can arise in floating-point FFT-based PA at large transform lengths. On the implementation side, we combined (i) a hybrid butterfly operation strategy that uses warp-synchronous register-resident updates for early stages and shared-memory staging for later stages, with (ii) a kernel-fused 9-step (3D-decomposed) NTT/INTT design that reduces global-memory round-trips via fused scaling and implicit data reorders. Experimental evaluation on two modern GPUs shows that

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

the proposed approach achieves **3.32 Gbps** (40.382 ms) on an NVIDIA L40 and **1.62 Gbps** (82.766 ms) on an NVIDIA RTX 3080 for 2^{27}-bit PA blocks. Consequently, the achieved performance offers sufficient processing capacity to support real-time QKD systems aiming for secret-key rates exceeding several hundred Mbps, accommodating varying compression ratios and protocol overheads.

Acknowledgment

This work is supported by the Ministry of Internal Affairs and Communications, Japan, via the project of R&D of ICT Priority Technology (JPMI00316) 'Research and development for early social implementation of quantum cryptography communication networks' (JPJ013328).

References

1. Bailey, D.H.: FFTs in external or hierarchical memory. The Journal of Supercomputing **4**(1), 23–35 (1990). https://doi.org/10.1007/BF00162341
2. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Advances in Cryptology — CRYPTO '86. Lecture Notes in Computer Science, vol. 263, pp. 311–323. Springer (1987). https://doi.org/10.1007/3-540-47721-7_24
3. Bosshard, N., Christen, R., Hänggi, E., Hofstetter, J.: Fast privacy amplification on gpus. Poster at QIP 2021 (2021), available at `https://doi.org/10.5281/zenodo.4551775`
4. Brent, R.P., Zimmermann, P.: Modern Computer Arithmetic. Cambridge University Press (2010). https://doi.org/10.1017/CBO9780511921698
5. Cheng, X., Mao, H., Xu, H., Li, Q.: Large-scale FPGA-based Privacy Amplification exceeding 10^8 bits for Quantum Key Distribution. arXiv:2503.09331 (2025). https://doi.org/10.48550/arXiv.2503.09331
6. Golub, G.H., Van Loan, C.F.: Matrix Computations. Johns Hopkins University Press, 4 edn. (2013)
7. Grünenfelder, F., Boaron, A., Resta, G.V., Perrenoud, M., Rusca, D., Barreiro, C., Houlmann, R., Sax, R., Stasi, L., El-Khoury, S., et al.: Fast single-photon detectors and real-time key distillation enable high secret-key-rate quantum key distribution systems. Nature Photonics **17**(5), 422–426 (2023)
8. Laudenbach, F., Pacher, C., Fung, C.H.F., Poppe, A., Peev, M., Schrenk, B., Hentschel, M., Walther, P., Hübel, H.: Continuous-variable quantum key distribution with gaussian modulationthe theory of practical implementations. Advanced Quantum Technologies **1**(1), 1800011 (2018)
9. Li, Q., Yan, B., Mao, H., Xue, X.: High-speed implementation of FFT-based privacy amplification on FPGA in Quantum Key Distribution. arXiv:1809.07592 (2018)
10. Li, Q., Yan, B., Mao, H., Xue, X., Han, Q., Guo, H.: High-speed and adaptive FPGA-based privacy amplification in Quantum Key Distribution. IEEE Access **7**, 21482–21490 (2019). https://doi.org/10.1109/ACCESS.2019.2897940
11. Lucamarini, M., Patel, K.A., Dynes, J.F., Fröhlich, B., Sharpe, A.W., Dixon, A.R., Yuan, Z.L., Penty, R.V., Shields, A.J.: Efficient decoy-state Quantum Key Distribution with quantified security. Optics Express **21**(21), 24550–24565 (2013). https://doi.org/10.1364/OE.21.24550

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45

12. Luo, Y., Cheng, X., Mao, H.K., Li, Q.: An overview of postprocessing in Quantum Key Distribution. Mathematics **12**(14), 2243 (2024). https://doi.org/10.3390/math12142243
13. Ng, S.Q., Kanitschar, F., Zhang, G., Wang, C.: Gigabit-rate quantum key distribution on integrated photonic chips (2025), accepted for presentation at QCrypt 2025 (per conference accepted-papers list)
14. Nukada, A., Matsuoka, S.: Auto-tuning 3-d FFT library for CUDA GPUs. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09). pp. 1–10 (2009). https://doi.org/10.1145/1654059.1654090
15. Nukada, A., Sato, K., Matsuoka, S.: Scalable multi-GPU 3-d FFT for TSUBAME 2.0 supercomputer. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12). pp. 1–10 (2012). https://doi.org/10.1109/SC.2012.100
16. Nussbaumer, H.J.: Number Theoretic Transforms. In: Fast Fourier Transform and Convolution Algorithms, Springer Series in Information Sciences, vol. 2. Springer, Berlin, Heidelberg (1982). https://doi.org/10.1007/978-3-642-81897-4_8
17. Özcan, A.Ş., Savaş, E.: Two algorithms for fast GPU implementation of NTT. Cryptology ePrint Archive, Paper 2023/1410 (2023), `https://eprint.iacr.org/2023/1410`
18. Pirandola, S., Andersen, U.L., Banchi, L., Berta, M., Bunandar, D., Colbeck, R., Englund, D., et al.: Advances in Quantum Cryptography. Advances in Optics and Photonics **12**(4), 1012–1236 (2020). https://doi.org/10.1364/AOP.361502
19. Renner, R., König, R.: Universally composable privacy amplification against Quantum adversaries. In: Theory of Cryptography Conference (TCC 2005). Lecture Notes in Computer Science, vol. 3378, pp. 407–425. Springer (2005). https://doi.org/10.1007/978-3-540-30576-7_22
20. Takahashi, R., Tanizawa, Y., Dixon, A.R.: High-speed implementation of privacy amplification in Quantum Key Distribution. In: QCrypt 2016 Extended Abstracts (2016), `https://obj.umiacs.umd.edu/extended_abstracts/QCrypt_2016_paper_160.pdf`
21. Tang, B., Liu, B., Zhai, Y., Wu, C., Yu, W.: High-speed and large-scale privacy amplification scheme for Quantum Key Distribution. Scientific Reports **9**, 15733 (2019). https://doi.org/10.1038/s41598-019-50290-1
22. Van Loan, C.F.: Computational Frameworks for the Fast Fourier Transform. SIAM (1992)
23. Wang, X., Zhang, Y., Li, Z., Xu, B., Yu, S., Guo, H.: High-speed implementation of length-compatible privacy amplification in Continuous-Variable Quantum Key Distribution. IEEE Photonics Journal **10**(3), 1–9 (2018). https://doi.org/10.1109/JPHOT.2018.2824316, art. no. 7600309
24. Xu, F., Ma, X., Zhang, Q., Lo, H.K., Pan, J.W.: Secure Quantum Key Distribution with realistic devices. Reviews of Modern Physics **92**(2), 025002 (2020). https://doi.org/10.1103/RevModPhys.92.025002
25. Yan, B., Li, Q., Mao, H., Chen, N.: An efficient hybrid hash based privacy amplification algorithm for Quantum Key Distribution. Quantum Information Processing **21**(4), 130 (2022). https://doi.org/10.1007/s11128-022-03452-8
26. Yan, B., Li, Q., Mao, H., Xue, X.: High-speed privacy amplification scheme using gmp in quantum key distribution. IEEE Photonics Journal **12**(3), 1–13 (2020)
27. Yuan, Z., Plews, A., Takahashi, R., Doi, K., Tam, W., Sharpe, A., Dixon, A., Lavelle, E., Dynes, J., Murakami, A., et al.: 10-mb/s quantum key distribution. Journal of Lightwave Technology **36**(16), 3427–3433 (2018)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_45