An NVMe-based Secure Computing Platform with FPGA-based TFHE Accelerator

Yoshihiro Ohba[1], Tomoya Sanuki[1], Claude Gravel[2] and Kentaro Mihara[2]

[1] KIOXIA Corporation
[2] EAGLYS Inc.

Abstract.
In this paper, we introduce a new approach to secure computing by implementing a platform that utilizes an NVMe-based system with an FPGA-based Torus FHE accelerator, SSD, and middleware on the host-side. Our platform is the first of its kind to offer complete secure computing capabilities for TFHE using an FPGA-based accelerator. We have defined secure computing instructions to evaluate 14-bit to 14-bit functions using TFHE, and our middleware allows for communication of ciphertexts, keys, and secure computing programs while invoking secure computing programs through NVMe commands with metadata. Our CMux gate implementation features an optimized NTT/INTT circuit that eliminates pre-NTT and post-INTT operations by pre-scaling and pre-transforming constant polynomials such as the bootstrapping and private-functional key-switching keys. Our performance evaluation demonstrates that our secure computing platform outperforms CPU-based and GPU-based platforms by 15 to 120 times and by 2.5 to 3 times, respectively, in gate bootstrapping execution time. Additionally, our platform uses 7 to 12 times less electric energy consumption during the gate bootstrapping execution time compared to CPU-based platforms and 1.15 to 1.2 times less compared to GPU-based platforms.

Keywords: FHE · TFHE · FPGA · Accelerator · NVMe · SSD

1 Introduction

Securing data is vital as public and private organizations recognize it as an asset when collecting, using, and sharing information. For this reason, data protection regulations are growing worldwide, and the demand for global privacy and requirements is also increasing.

Along with these trends, there is a highly increasing demand for secure computation, also known as privacy-preserving methods. Gentry's seminal work introduced a class of cryptographic methods known as Fully Homomorphic Encryption (FHE) [18], and it is considered one of the most compelling technologies in secure computing. FHE-based privacy-preserving methods do not require computing nodes to decrypt encrypted data to perform secure computation, and therefore, the computing nodes are free from side-channel attacks.

Since its inception, FHE has sparked significant interest, leading to the emergence of novel constructions following Gentry's idea. This evolution has culminated in the development of four FHE schemes, namely, BGV [10], BFV [9, 16], CGGI (also known as TFHE (Torus FHE)) [13], and CKKS [8, 12], which are considered the most representative and are currently undergoing international standardization under ISO [30]. This progression showcases the growing interest in FHE and the continuous advancements in the field, making it an exciting area of research.

One of the critical components of FHE is bootstrapping, or a procedure to decrease the ciphertext error by homomorphically evaluating a decryption circuit [18]. The usual computation in bootstrapping is an inner product of two vectors encoding polynomials. One vector is a transformation of a ciphertext, and the other vector is an encryption of the secret key used for generating the ciphertext. The computational complexity of the polynomial multiplications is N times larger than the one for decrypting the ciphertext, where N is the degree of the ideal of the polynomial ring. Since N typically ranges from 2^{10} to 2^{16} depending on the FHE mechanism and security parameters [13, 20], there is a strong demand for speeding up the bootstrapping procedure. Some FHE accelerators were built for this purpose using Graphical Processing Units (GPU), Field Programmable Gate Arrays (FPGA), or Application Specific Integrated Circuit (ASIC). All FHE accelerators listed in [20] implement Number Theoretic Transform (NTT) into hardware. However, more work must be done on FHE-based secure computing platforms integrating TFHE accelerators.

We design and implement a Non-Volatile Memory express (NVMe)-based secure computing platform with an FPGA-based TFHE accelerator, Solid State Drive (SSD), and a host-sided middleware. We implement TFHE to speed up non-linear operations and bit-wise operations. Our secure computing platform uses NVMe commands for reading, writing, and executing secure computing programs containing a sequence of secure computing instructions. The NVMe commands also read and write both ciphertexts and keys. We define a set of secure computing instructions to evaluate any 14-bit to 14-bit function using TFHE.

Our accelerator has an optimized circuit for performing N-point NTT or INTT with $N = 16384$ operating at 200MHz using the method described in [28] to eliminate bit-reversal operations, precomputing twiddle factors, and to properly normalize. We develop an optimized CMux gate using a linear sum of the pre-scaled and pre-transformed constant polynomials as input parameters. We show that bootstrapping and private-functional key-switching keys are such polynomials. Section 6 shows that our secure computing platform outperforms CPU-based platforms and GPU-based platforms by 15 to 120 times and by 2.5 to 3 times, respectively, in gate bootstrapping execution time, and by 7 to 12 times and by 1.15 to 1.2 times, respectively, in electric energy consumption during the gate bootstrapping execution time.

The rest of this paper is as follows. Section 2 discusses existing work related to this paper and clarifies our contributions to FHE-based secure computing. Section 3 describes the basic architecture of our secure computing platform. Section 4 and Section 5 explain the design and implementation of our accelerator and middleware, respectively, in detail. Section 6 provides the performance evaluation of our secure computing platform implementation. Finally, Section 7 summarizes this paper and mentions our future work.

2 Related Work

A detailed survey on FHE is in [23]. There is also a survey on FHE accelerators in [20]. There are three works related to FHE-based secure computing platforms. In [34], an FPGA-based accelerator called SmartSSD [21] implements the basic operations needed for CKKS. In [15], a secure computing platform is implemented with an FPGA-based accelerator for NTT, focusing on accelerating the Chinese Remainder Transform (CRT) and using Direct Memory Access (DMA) for using host DRAM for communicating commands and data between CPU and FPGA. In contrast to [15], our platform focuses on speeding up NTT in a TFHE-specific way. However, a RISC-based secure computing platform for TFHE is developed from [25], mainly focusing on accelerating CMux-tree operations without accelerating the bootstrapping procedure.

Among FHE accelerators based on GPU [5, 6, 27, 35], ASIC [3, 32], and FPGA [2, 7, 15, 17, 29, 31, 33, 34], we have chosen FPGA as our hardware since bootstrapping is known as a memory-bandwidth-bound workload for CPU with its arithmetic intensity (the ratio between the number of executed operations and the number of bytes transferred between the CPU and the memory) being less than 1, see [11]. FPGA or ASIC is more suitable for such a workload than GPU because it allows multiple arithmetic logics to access the different internal memory blocks or caches simultaneously. FPGA is more suitable than ASIC for the initial pre-standard deployment phase.

There are several works on FPGA-based accelerators implementing NTT for TFHE [7, 17], CKKS [2, 29], and BGV/BFV [31, 33]. For instance, an FPGA-based TFHE accelerator [7] uses fixed-point Fast Fourier Transform (FFT), aiming for high throughput and low control overhead. Another FPGA-based TFHE accelerator [19] uses approximate multiplication-less integer FFT. Another FPGA-based TFHE accelerator [37] introduces an optimization technique called bootstrapping key unrolling designed on the tradeoff between the performance of bootstrapping and FPGA resource consumption. These three FPGA-based TFHE accelerators are implemented for smaller N ($N = 1024$). In contrast, our FPGA-based TFHE accelerator has a different design goal of speeding up the multiplication of polynomials without losing precision for a large value of N. In [17], an FPGA-based programmable vector engine that supports processing an application-specific instruction set is designed without accelerating the bootstrapping procedure. In [26], a TFHE accelerator on a commodity CPU-FPGA hybrid machine is designed for parallel execution of multiple homomorphic boolean gates to increase processing throughput, however, without reducing latency. We focus on reducing latency in the execution of bootstrapping.

The work in [19] has extensively evaluated the performance of an FPGA-based accelerator during bootstrapping, focusing on latency, throughput, and power consumption. However, a significant gap remains in the literature, as no study has yet comprehensively assessed an FHE-based secure computing platform, which includes an accelerator and a host CPU, in terms of these performance metrics.

Our major contributions are as follows.

- We are the first to provide a full-fledged secure computing platform for TFHE using an FPGA-based accelerator. The platform defines virtual registers for manipulating secure computing instructions designed for TFHE and a host-sided middleware to communicate ciphertexts, keys, and programs to compute. We can invoke a secure computing program over the accelerator through the middleware using NVMe commands with metadata.

- We are also the first to provide, instead of a processor-level comparison among different processors such as CPU, GPU, and FPGA, a platform-level comparison showing that a secure computing platform with an FPGA-based accelerator can outperform software-based and GPU-based secure computing platforms in terms of speed and energy consumption for executing bootstrapping operations for large values of N. We have shown especially that NTT and Inverse NTT (INTT) with $N = 16384$ are momory-I/O-bound workloads for GPU.

3 Basic Architecture

Figure 1 shows the basic architecture of our secure computing platform. This architecture has been chosen to process data close to its location. The architecture comprises a Host, an Accelerator, and NVMe SSDs (hereafter SSDs).

A secure computing application running on the host controls the behavior of the accelerator through middleware and an API by using NVMe Read/Write commands. Firstly, the accelerator intercepts each NVMe Read/Write command issued for the SSDs. Secondly, it holds a copy of the data carried in the command. Thirdly and finally, depending on the command type and the properties of the command data, referred to as secure computing metadata, it runs a program for homomorphic calculation and returns the result. The main reason for using NVMe as the carrier for secure computing data and metadata is to reduce overall latency for FHE-based secure computing by performing storage I/O and computing in a single I/O command, avoiding unnecessary data transfer from the host main memory to the accelerator after the host reads the data from the storage or the accelerator to the host main memory before the host writes the data to the storage. In other words, our accelerator is a computing storage accelerator. Also, unlike SmartSSD [21], in which an FPGA and an SSD are standalone PCIe devices connected under a PCIe switch, our architecture further avoids "double transfer" of the same data over the same PCIe segment between the FPGA and the PCIe switch, one for transferring the data between the host and FPGA and another for transferring the data between the FPGA and SSD, which halves the I/O throughput [22]. Instead, our architecture maintains the I/O throughput by physically separating the PCIe segment between the host and FPGA and the downstream PCIe segment between the FPGA and SSD. The basic architecture uses Peripheral Component Interconnect (PCIe) or NVMe over Fabrics (NVMe-oF) as an NVMe transport. In implementing the basic architecture, PCIe is the NVMe transport between the host and the accelerator and between the accelerator and the SSDs. Future work will extend the basic architecture to have multiple hosts and accelerators.

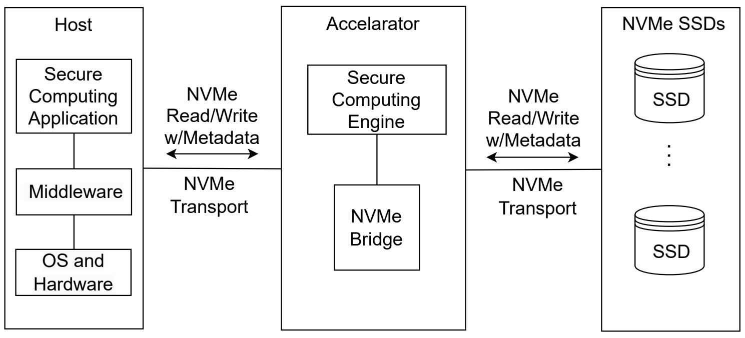


Figure 1: Basic architecture

4 Accelerator

The block diagram of our accelerator implementation is in Figure 2. Our accelerator is implemented on a HiTech Global HTG-937 board equipped with a Xilinx XCVU47P FPGA with three Super Logic Regions (SLRs) and 16GB of High Bandwidth Memory (HBM). The accelerator consists of a Secure Computing Engine (hereafter the computing engine) and an NVMe Bridge. The computing engine first inputs an NVMe Command or Completion with its associated data to the NVMe bridge; it extracts the secure computing data and metadata from the input and stores the secure computing data in a Virtual Register (VR). Secondly, the engine executes a program containing a sequence of secure computing instructions, depending on the type of secure computing data indicated in the secure computing metadata and the type of the NVMe Command. Thirdly, the engine outputs the NVMe Command or Completion and its associated data containing either the input or computed data to the NVMe bridge.

The computing engine has the following components.

- *Main Memory* stores VRs, a VR table, and page tables. It also has a stack region used by Push and Pop instructions, defined in Section 4.4. HBM is the memory device that provides sufficient memory access bandwidth. The main memory is partitioned into a persistent area for which paging is not applied and a non-persistent area for which paging is used. See Section 4.3 for details on paging.

- *Cache Memory* consists of many Block Random Access Memory (BRAM) blocks for high-speed distributed memory access.
- *Data Movers* move secure computing data and metadata among the main memory, cache memory, central processor, and module processor. Data movers have First-In First-Out (FIFO) buffers.
- *Module Processor* provides ring or vector operations. Multiple logic blocks in the module processor can access simultaneously different BRAM blocks in the cache memory. The module operations supported by the module processor are listed in Table 1. All module operations are implemented as High-Level Synthesis (HLS) modules, except for NTT and INTT. The NTT/INTT circuit is implemented as RTL (Register Transfer Level) modules and described in Section 4.5. Module operations are performed element-wise except for NTT, RING_ROT, VECTOR_ROT, and SAMPLE_EXT. The module processor has one MicroBlaze soft-core microprocessor for processing some module operations.
- *Central processor* executes microprograms to control the data movers and module processor and manage the main memory. One MicroBlaze soft-core microprocessor is used as the central processor.
- *Multiplexers and Demultiplexsors* exchange NVMe Commands and Completions with their associated data input from the NVMe bridge among the soft-core microprocessor, data movers, and NVMe bridge.

The NVMe bridge provides a bridging function of NVMe commands; that is, it forwards an NVMe command from the host to an SSD, either forwards Write Data from the host to the SSD or forwards Read Data from the SSD to the host, and forwards an NVMe Completion received from the SSD to the host. Before forwarding an NVMe Command or Completion, the NVMe bridge passes the NVMe Read/Write data to the computing engine for copying and data computation. The NVMe bridge has one MicroBlaze soft-core microprocessor.

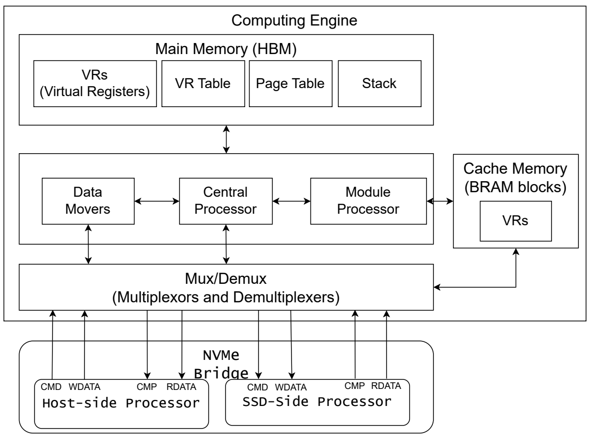


Figure 2: Accelerator block diagram

Table 1: Module processor operations. NTT operation is implemented as Register Transfer Level (RTL) modules. All other operations are implemented as High Level Synthesis (HLS) modules.

Name of Internal Operation	Module Type	Description
NTT	Ring	NTT and Inverse NTT
MULMOD64	Vector	64-bit element-wise multiplication modulo prime $p = 2^{64} - 2^{32} + 1$ for CMux
ADDMOD64	Vector	64-bit element-wise addition modulo prime $p = 2^{64} - 2^{32} + 1$ for CMux
KEY_SWITCH	Vector	Public functional KS
DECOMP	Vector	Gadget Decomposition
ADD32_ACC	Vector	32-bit element-wise addition to ACC
SUB32_ACC	Vector	32-bit element-wise subtraction to ACC
ADD32_VR	Vector	32-bit element-wise addition to VR
SUB32_VR	Vector	32-bit element-wise subtraction to VR
INT_MULT32	Vector	32-bit element-wise scalar multiplication to VR
RING_ROT	Ring	Circular rotation of ring coefficients
VECTOR_ROT	Vector	Circular rotation of Vector elements
SAMPLE_EXT	Ring	Sample Extract

ACC: Accumulator, KS: Key-Switching

4.1 Secure computing metadata

Each secure computing data (hereafter `sc_data`) accompanies secure computing metadata (hereafter `sc_metadata`) containing the Type of the `sc_data`, the Key Identifier identifying the set of keys associated with the `sc_data`, the Data Identifier of the `sc_data`, and the Size of the `sc_data`. The domain of the Type field is in Table 2.

Table 2: Secure computing metadata types

Type	Type Name	Description
0	PRG	Secure computing program
1	TV	Test vector
2	KEY	Key used for TFHE bootstrapping operations
3	TLWE-CoR	TLWE ciphertext invoking Compute-on-Read (CoR) operation
4	TLWE-CoW	TLWE ciphertext invoking Compute-on-Write (CoW) operation

Any `sc_data` carried in an NVMe Read/Write Command data is stored in the VR corresponding to the associated `sc_metadata`. In addition, a `sc_data` of Type 3 (TLWE-CoR) carried with an NVMe Read Command or Type 4 (TLWE-CoW) carried with an NVMe Write Command invokes execution of a secure computing program stored in the VR register of Type 0, or the program register (see Figure 3).

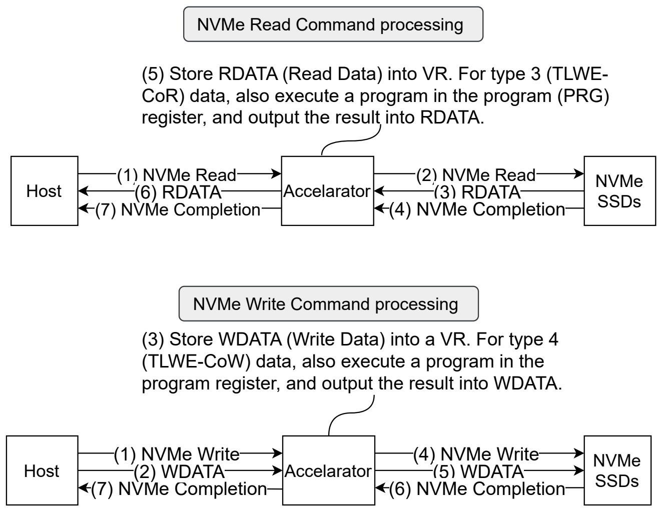

NVMe Read Command processing
(5) Store RDATA (Read Data) into VR. For type 3 (TLWE-CoR) data, also execute a program in the program (PRG) register, and output the result into RDATA.
Host
(1) NVMe Read
(6) RDATA
(7) NVMe Completion
Accelarator
(2) NVMe Read
(3) RDATA
(4) NVMe Completion
NVMe SSDs
NVMe Write Command processing
(3) Store WDATA (Write Data) into a VR. For type 4 (TLWE-CoW) data, also execute a program in the program register, and output the result into WDATA.
Host
(1) NVMe Write
(2) WDATA
(7) NVMe Completion
Accelarator
(4) NVMe Write
(5) WDATA
(6) NVMe Completion
NVMe SSDs

Figure 3: NVMe Read/Write command processing

4.2 Virtual registers and virtual addressing

Virtual Registers (VRs) are variable-length data structures maintained inside the accelerator and manipulated via NVMe Read/Write commands. VRs are distinguished from soft-core microprocessor registers in that the size of a VR can be so large that its entire part does not fit in FPGA logical elements. For example, a TFHE bootstrapping key can be a few gigabytes.

A VR number identifies each VR, uniquely calculated from the Type, Key Identifier, and Data Identifier contained in the `sc_metadata` associated with the corresponding `sc_data`. The VR table stores the pair of the VR number and the virtual address for each VR.

The type and data identifier specified in Table 3 determines the size of a VR. We use a 32-bit integer to encode an element in $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ for all data types other than the NTT-applied keys, which use 64-bit integers to multiply ring polynomials using NTT.

The accelerator uses a 38-bit virtual address with a 26-bit page number and a 12-bit offset. BKNTT, KSK, and PrvKSKNTT denote keys. BKNTT is the bootstrapping key transformed linearly; more specifically, it is NTT applied to the bootstrapping key. KSK is the key used in the public-functional key-switching mechanism of TFHE. PrvKSKNTT is the key used in the private-functional key-switching mechanism of TFHE and is transformed linearly in the same way as BKNTT. Parameters N, k, n, ℓ, and t denote the degree of the polynomial representing the ideal, the number of polynomials encoding a secret key in a Torus Ring Learning with Errors (TRLWE) sample, the bit length of the Torus LWE (TLWE) a secret key, the number of digits in the radix B_g of the Gadget Decomposition [13], and the number of digits in the binary-decomposed TLWE samples, respectively.

4.3 Paging

Like legacy computers, the accelerator invokes a paging function when its main memory is full. The paging algorithm implemented on the accelerator uses the SSDs as its swap area. It maintains the contents of received VRs to be stored in the main memory or the swap area in the following way. Each VR content sent initially to the accelerator

Table 3: Virtual register sizes (BKNTT: NTT-applied bootstrapping key, KSK: key-switching key, PrvKSKNTT: NTT-applied private-functional key-switching key)

Type	Data Identifier	VR Size in bytes
0 (PRG)	0	configurable
1 (TV)	any	$(k+1) \cdot N \cdot 4$
	1 (BKNTT)	$n \cdot N \cdot \ell \cdot (k+1)^2 \cdot 8$
2 (KEY)	2 (KSK)	$(n+1) \cdot t \cdot N \cdot 4$
	3 (PrvKSKNTT)	$(k+1) \cdot (n+1) \cdot t \cdot N \cdot 8$
3 (TLWE-CoR)	any	$(n+1) \cdot 4$
4 (TLWE-CoW)	any	$(n+1) \cdot 4$

through an NVMe command is temporarily held in a FIFO queue in the cache memory and then moved to the main memory. Suppose the central processor of the accelerator tries to access a VR, say x. Suppose neither x is in the main memory nor enough space is available to store x. In that case, the paging algorithm (i) selects some other VR, say y, stored in a page of the main memory and copies the page's content to the swap area, and then (ii) copies x to the page. Paging operations (i) and (ii) are *page-out* and *page-in*, respectively. The copy source of a page-in operation is either the swap area for previously received VR or the cache memory for newly received VR.

The accelerator includes a page table that is dedicated to VRs of Type 3 (TLWE-CoR) and Type 4 (TLWECoW), with a page size that matches the VR size. All other VRs are stored in the persistent area of the main memory, meaning that they do not require paging. Essentially, paging is only necessary for TLWE samples. Each entry in the page table contains a flag and a physical address for the corresponding page. If the flag is unset, the physical address field will contain the physical address of the main memory. If the flag is set, it will contain an LBA of the swap area. Page-in and page-out operations rely on NVMe Read/Write commands to transfer pages between the accelerator and SSDs. To avoid conflicts between host-issued and accelerator-issued NVMe commands, we have implemented a mechanism where the accelerator issues the NVMe commands for page-in and page-out operations. Then the accelerator:

1. Suspends forwarding of the NVMe Completion for the paging trigger command
2. Uses the Command Identifier of the paging trigger command for the accelerator-issued NVMe commands used for paging operations
3. Resumes forwarding of the suspended NVMe Completion once the paging operations terminate

Note that host-issued NVMe commands that are not paging trigger commands are not suspended during paging.

4.4 Secure computing instruction set

The accelerator supports the following secure computing instructions, summarized in Table 4.

Return sends the content of the VR register n containing a TLWE sample to the host or SSD.

Move moves the content of the VR register n_2 containing a TLWE sample to VR register n_1.

Push Moves the content of the VR register n containing a TLWE sample to the top of the stack and increments the stack pointer.

Pop moves the content of the stack top to VR register n and decrements the stack pointer.

Bootstrap uses the content of VR register tv containing a TFHE test vector, performs Gate Bootstrapping (GBS) [13] for the VR register n containing a TLWE sample, and stores the output to VR register n. Note that GBS also realizes Programmable Bootstrapping (PBS) [14] for a function $f(x)$ provided by the TFHE test vector, in which case the default TFHE test vector for the identity function is replaced with the provided one. Note that a pair of a bootstrapping key and a key-switching key used for GBS is identified by the Key ID field of the `sc_metadata` associated with VR register n.

HomAdd adds the content of VR register n_2 containing a TLWE sample and VR register n_1 containing another TLWE sample and stores the result to VR register n_1. HomAdd internally calls ADD32_VR module processor operation.

HomSub subtracts the content of VR register n_2 containing a TLWE sample from VR register n_1 containing another TLWE sample and stores the result to VR register n_1. HomSub internally calls SUB32_VR module processor operation.

HomIntMult multiplies the content of VR register n containing a TLWE sample by value v and stores the result to VR register n. HomIntMult internally calls INT_MULT32 module processor operation.

Table 4: Secure computing instruction set (VRs n, n_1, and n_2, each containing a TLWE sample. VR tv contains a TFHE test vector.)

Name	Type	Arg. 1	Arg. 2	Description
Return	0	n	none	return n
Move	1	n_1	n_2	$n_1 \leftarrow n_2$
Push	2	n	none	`++stackptr` $\leftarrow n$
Pop	3	n	none	$n \leftarrow$ `stackptr--`
Bootstrap	4	tv	n	perform GBS or PBS for n with tv
HomAdd	5	n_1	n_2	$n_1 \leftarrow n_1 + n_2$
HomSub	6	n_1	n_2	$n_1 \leftarrow n_1 - n_2$
HomIntMult	7	n	v	$n \leftarrow n \cdot v$

Table 5 shows the sequence of instructions of a secure computing program that homomorphically performs a multiplication of two integers, x and y. Also, Table 6 shows an example sequence of NVMe commands used for running the example secure computing program. Note that once VRs are loaded into the accelerator via NVMe Read/Write Commands, they only need to be reloaded once they need to be updated. For example, steps 1 through 6 in Table 6 are not required for the next run of another secure computing program using the same BK, KSK, and TV1. Also, for SSDs supporting the NVMe Metadata feature, the number of NVMe Commands in the sequence is reduced by half by having `sc_metadata` and its associated `sc_data` in the same NVMe Read/Write Command.

Table 5: An example of a secure computing program for computing $xy = \{(x+y)^2/4 - (x-y)^2/4\}$ homomorphically (r_i is the VR number for TLWE sample s_i. tv is the VR number for the test vector representing the function $f(z) = z^2/4$. TLWE samples s_2 and s_3 contain encrypted data for x and y, respectively. VR r_1 stores a temporal result and a final result to return.)

No.	Instruction	Argument(s)	No.	Instruction	Argument(s)
1	mov	r_1, r_2	5	bootstrap	tv, r_2
2	homadd	r_1, r_3	6	homsub	r_1, r_2
3	bootstrap	tv, r_1	7	return	r_1
4	homsub	r_2,r_3			

Table 6: An example sequence of NVMe commands (Write(d) represents an NVMe Write command with data d. Read(d) represents an NVMe Read command with data d. MD(x, y, z) represents the metadata of Type x, Key Identifier y, and Data Identifier z. An NVMe Completion (not shown in the figure) is returned for each NVMe command. Abbreviations: BK: bootstrapping key. KSK: key-switching key. PRG: secure computing program.)

No.	NVMe Command	Comment
1	Write(MD(2,0,1))	BKNTT
2	Write(k_0)	BKNTT data
3	Write(MD(2,0,2))	KSK
4	Write(k_1)	KSK data
5	Write(MD(1,0,0))	TV1
6	Write(v_1)	TV1 data
7	Write(MD(0,0,0))	PRG
8	Write(p)	PRG data
9	Write(MD(3,0,1))	TLWE-CoR
10	Write(s_1)	TLWE-CoR data for TLWE sample s_1 encrypting value x
11	Write(MD(4,0,2))	TLWE-CoW
12	Write(s_2)	TLWE-CoW data for TLWE sample s_2 to be used for encrypting $f(x)$. Secure computing program p is invoked here.
13	Read(MD(4,0,2))	TLWE-CoW
14	Read(s_2)	TLWE-CoW data for TLWE sample s_2 encrypting value $f(x)$

4.5 NTT implementation and optimized CMux

Since bootstrapping is the most time-consuming operation in TFHE, the NTT/INTT circuit in the module processor is implemented as an RTL module to optimize its circuit design. The NTT/INTT circuit supports N-point NTT/INTT with $N = 16384$. The NTT/INTT circuit implements an optimized scheme described in [28], which eliminates pre-FFT (Fast Fourier Transform) processing and post-IFFT (Inverse FFT) processing (including bit reversing) by merging NTT twiddle factors $\{\psi^i \mid 0 \leq i < N\}$ and FFT twiddle factors $\{\omega^i | 0 \leq i < N\}$ where ω is a primitive Nth primitive root of unity and ω is a primitive $2N$-th primitive root of unity, and thus $\psi = \omega^2$. For readers' convenience, the NTT and INTT constructions for $N = 8$ are shown in Figure 4. The constructions are similar to those in Figure 1 of [28], except we use Gentleman-Sande

(GS) butterfly for NTT and Cooley-Tukey (CT) butterfly for INTT, which eliminates bit-reversing FFT twiddle factors. In our implementation, $\omega = 10930245224889659871$ and $\psi = 3333600369887534767$. Appndix A gives mathematical derivations for the two types of optimized butterfly elements.

For a complete INTT operation, a normalization factor of $1/N$ is required for each output element of INTT in Figure 4. This scaling can be precomputed for the input of NTT depending on the arithmetic operation that uses NTT [28] in its implementation. In TFHE, CMux gate [13], defined as follows, is such an operation.

$$\mathrm{CMux}(C, d_0, d_1) = \sum_{i=1}^{(k+1)\ell} u_i \cdot C_i + d_0 = \langle u, C \rangle + d_0$$

where $C = (C_i)_{1 \le i < (k+1)\ell}$ is a Torus Ring GSW (TRGSW) sample, d_0 and d_1 are TRLWE samples, and $u = (u_1, u_2, ..., u_{(k+1)\ell}) \in (\mathbb{Z}[X]/(X^N + 1))^{(k+1)\ell}$ is the output of the Gadget Decomposition for $d_1 - d_0$ and $\langle x, y \rangle$ denotes the inner product of two vectors x and y.

In [13], CMux gates are used inside the Blind Rotate algorithm, which is invoked from Gate Bootstrapping (Case 1) or the Vertical Packing algorithm used together with Circuit Bootstrapping [13] (Case 2). In Case 1, C_i is the ith TRGSW sample of a bootstrapping key BK. In Case 2, C_i is the ith TRGSW sample of the output of Circuit Bootstrapping, calculated as a linear sum of the elements of a private-functional key-switching key PrvKSK. Since C_i is generally a linear sum of constant polynomials in both cases and taking advantage of the linearity property of NTT, our optimized CMux gate uses pre-scaled and pre-transformed key keyntt $= (\mathrm{keyntt}_i)_i = (\mathrm{NTT}^{(N)}(\mathrm{key}_i/N))_i$, where key_i is the ith element of BK or PrvKSK, as follows.

$$\begin{aligned}
\langle u, C \rangle &= \frac{1}{N} \sum_{i=1}^{(k+1)\ell} \mathrm{INTT}^{(N)} \left(\mathrm{NTT}^{(N)}(u_i) \odot \mathrm{NTT}^{(N)}(C_i) \right) \\
&= \mathrm{INTT}^{(N)} \left(\sum_{i=1}^{(k+1)\ell} \mathrm{NTT}^{(N)}(u_i) \odot \mathrm{NTT}^{(N)}(C_i/N) \right) \\
&= \mathrm{INTT}^{(N)} \left(\sum_{i=1}^{(k+1)\ell} \mathrm{NTT}^{(N)}(u_i) \odot \langle c'_i, \mathrm{keyntt} \rangle \right) \qquad (1)
\end{aligned}$$

Here, $\mathrm{NTT}^{(N)}(\cdot)$ and $\mathrm{INTT}^{(N)}(\cdot)$ are N-point NTT and INTT functions, respectively. $\odot$ denotes the Hadamard product. $c'_i = (c'_{i,j})_{1 \le j \le (n+1)t}$ is an integer vector with $c'_{i,j} = \delta_{ij}$ for Case 1 where δ_{ij} is Kronecker delta function, and $c'_{i,j} = -\tilde{c}_{i, j-1 \text{ div } t, j-1 \text{ mod } t}$ for Case 2 where $(\tilde{c}_{i,j,k})_{1 \le j \le n+1, 1 \le k \le t}$ are t bit-decomposed TLWE samples generated from the i-th TLWE sample in private-functional key-switching [13]. This optimization halves the number of NTT operations performed in CMux. For now, we only implemented Case 1.

Figure 5 shows two types of radix-2 butterfly calculation elements, one for NTT and the other for INTT. Our implementation integrates the two types of butterfly calculation elements in a single integrated butterfly circuit, as in Figure 6. The integrated butterfly circuit has the same construction as the NTT and INTT parts of the unified butterfly circuit described in [36].

Figure 7 shows the pipeline and parallel processing model for computing NTT and INTT in the module processor. The NTT/INTT circuit in the module processor has 32 integrated butterfly circuits operating in parallel at 200MHz, and is partitioned into two sub-circuit of 16 integrated butterfly circuits, where each sub-circuit processes one of the two polynomials in a sample $(a, b) \in \{\mathbb{Z}[X]/(X^N + 1)\}^2$. Data processing within each integrated butterfly circuit is pipelined so that BRAM reads for the subsequent coefficients and twiddles are done during butterfly calculation for the current coefficients and twiddles. Each integrated butterfly circuit performs 512 butterfly calculations in the pipeline to compute one row of NTT or INTT. Note that transferring coefficients between the HBM and BRAM is performed by the data mover in the background of the module processor pipeline, and it never causes a pipeline stall thanks to the high-bandwidth nature of HBM.

Figure 8 shows accelerator's die layout. Two design policies are applied to reduce data transfer among SLRs. First, each sub-circuit of the module processor is laid out in a different SLR (i.e., SLR#2 and SLR#3 in Figure 8). Second, the data movers are placed in SLR#1, the closest SLR to HBM, as the data movers are the interface between HBM and other FPGA logics.

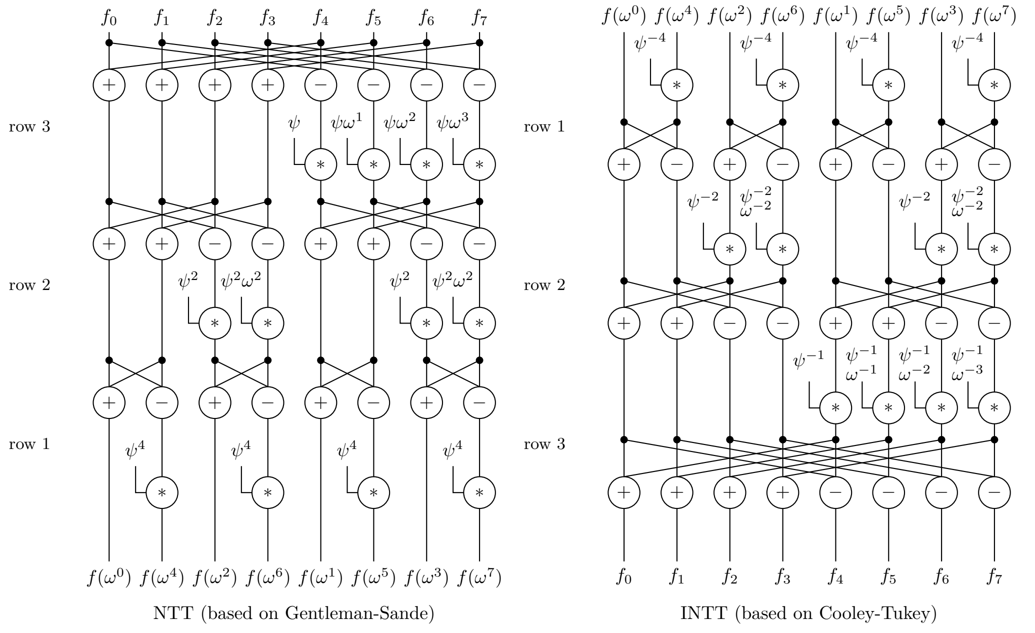

f_0 f_1 f_2 f_3 f_4 f_5 f_6 f_7
row 3
row 2
row 1
ψ $\psi\omega^1$ $\psi\omega^2$ $\psi\omega^3$
ψ^2 $\psi^2\omega^2$
ψ^4
$f(\omega^0)$ $f(\omega^4)$ $f(\omega^2)$ $f(\omega^6)$ $f(\omega^1)$ $f(\omega^5)$ $f(\omega^3)$ $f(\omega^7)$
NTT (based on Gentleman-Sande)
ψ^{-4}
ψ^{-2} $\psi^{-2}\omega^{-2}$
ψ^{-1} $\psi^{-1}\omega^{-1}$ $\psi^{-1}\omega^{-2}$ $\psi^{-1}\omega^{-3}$
INTT (based on Cooley-Tukey)

Figure 4: N-point NTT and INTT constructions for $N = 8$ (ω and ψ are Nth and $2N$th roots of unity, respectively. $\{\omega^i : 0 \leq i < N\}$ and $\{\psi^i : 0 \leq i < N\}$ are FFT twiddle and NTT twiddle factors, respectively. f_i is the ith input element in the time domain. $f(\omega^i)$ is the value in the frequency domain for f_i. INTT's row number is in reverse order of NTT's row number.)

4.6 Computing modulo prime $p = 2^{64} - 2^{32} + 1$

Although NTT for 32-bit Torus can use any prime number that is greater than 2^{32}, we choose a Proth prime $p = 2^{64} - 2^{32} + 1 = 18446744069414584321$ as used in existing open-source TFHE implementations because modulo calculation for a Proth prime requires no integer multiplication calculation. For two integers $x, y \in [0, p)$, $z = xy$ can be decomposed into three parameters $a, b \in [0, 2^{32})$ and $c \in [0, 2^{64})$ as $z = a \cdot 2^{96} + b \cdot 2^{64} + c$, $z \bmod p$ for z

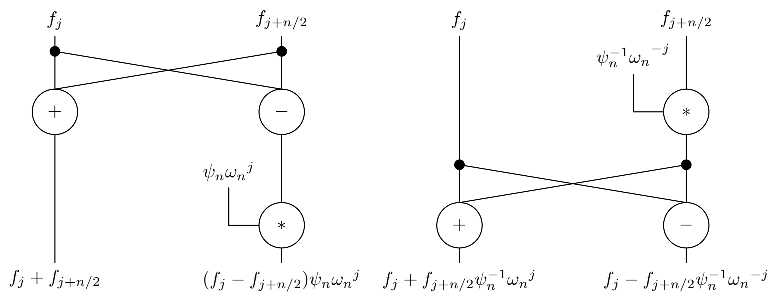

f_j
$f_{j+n/2}$
$\psi_n \omega_n{}^j$
$f_j + f_{j+n/2}$
$(f_j - f_{j+n/2})\psi_n \omega_n{}^j$
f_j
$f_{j+n/2}$
$\psi_n^{-1} \omega_n{}^{-j}$
$f_j + f_{j+n/2}\psi_n^{-1} \omega_n{}^j$
$f_j - f_{j+n/2}\psi_n^{-1} \omega_n{}^{-j}$

Figure 5: Two types of radix-2 butterfly circuits at row $\log_2 n$ (left: NTT butterfly, Right: INTT butterfly, $n = 2, 4, \ldots, 2^i, \ldots, N, 0 \le j < n/2$, $\omega_n = \omega^{N/n}, \psi_n = \psi^{N/n}$. N is a power of 2.)

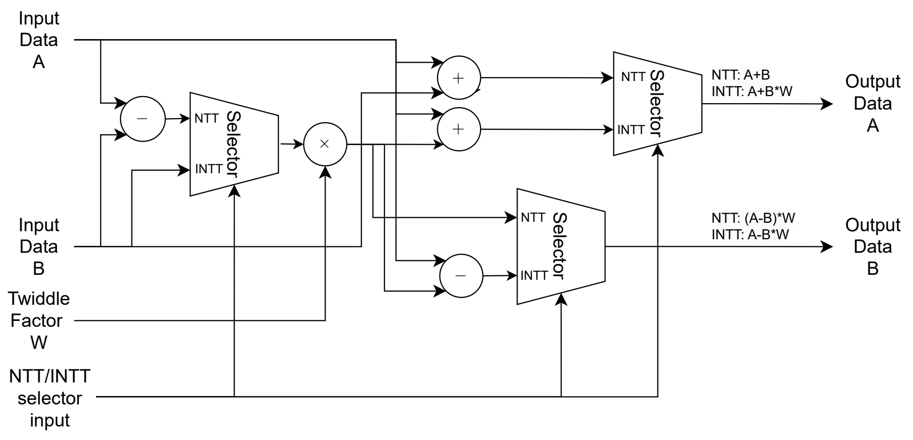


Figure 6: Integrated Butterfly Circuit

is calculated by using addition, subtraction, shift, and comparison operations as follows.

$$z \bmod p = \begin{cases} m(z), & \text{if } z < 2p \\ m\big(m(b \cdot 2^{32}) + \\ \quad m\big(m(c) + p - m(a+b)\big)\big), & \text{otherwise} \end{cases}$$

where

$$m(j) = \begin{cases} j, & \text{if } j < p \\ j + \texttt{unit32}(-1), & \text{if } j \in [p, 2p). \end{cases}$$

5 Middleware

To set/get `sc_metadata` and `sc_data` to/from the accelerator and let the accelerator execute secure computing programs via NVMe commands, the middleware of our platform uses the Blobstore feature of Storage Performance Development Kit (SPDK) (https://spdk.io/). Figure 9 shows the middleware architecture. The middleware API functions and the internal API functions from SPDK and SPDK Blobstore (hereafter the blobstore) are callback-based functions to achieve high-performance and nonblocking NVMe storage access; functions directly or indirectly interact with an abstraction thread library primarily based on the Portable Operating System Interface (POSIX).

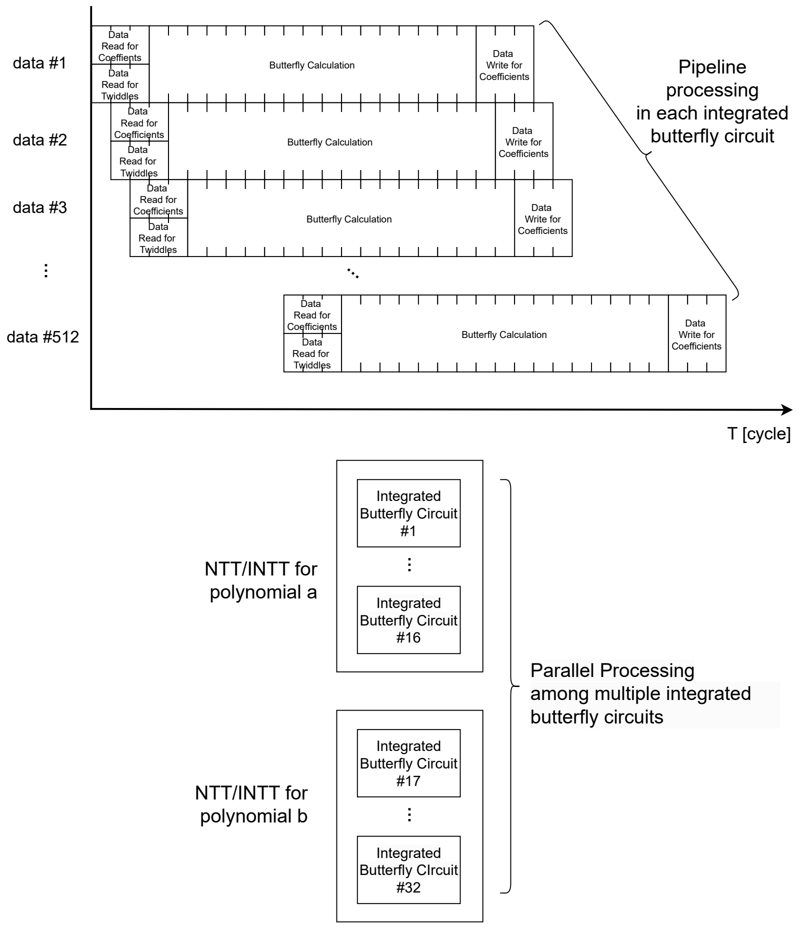

data #1
data #2
data #3
data #512
Data Read for Coeffients
Data Read for Twiddles
Data Read for Coefficients
Butterfly Calculation
Data Write for Coefficients
Pipeline processing in each integrated butterfly circuit
T [cycle]
NTT/INTT for polynomial a
Integrated Butterfly Circuit #1
Integrated Butterfly Circuit #16
NTT/INTT for polynomial b
Integrated Butterfly Circuit #17
Integrated Butterfly CIrcuit #32
Parallel Processing among multiple integrated butterfly circuits

Figure 7: Pipeline and Parallel Processing Model for NTT and INTT (upper: pipelining within each integrated butterfly circuit, lower: parallel processing among multiple integrated butterfly circuits)

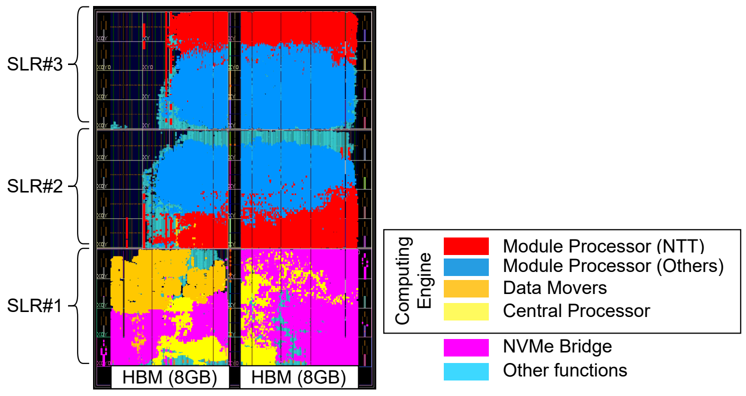


Figure 8: Accelerator die layout

The blobstore manages user data as blobs. A blob consists of blob data containing the user data and blob metadata describing the attributes, such as the size of the blob data. The blob data and metadata are stored as clusters, each consisting of one or more pages stored in consecutive logical blocks. The first cluster stores pieces of blob metadata in its corresponding region and the remaining clusters store pieces of blob data. The host's RAM keeps a copy of the blob metadata region.

Our accelerator can assemble VRs without overhead due to accessing the blob metadata region of the disk or maintaining a copy of the blob metadata region in its HBM or BRAM (1) by using the extended portion of blob metadata for passing secure computing metadata between the middleware API and the Blobstore API, and (2) by mapping between the blob metadata and `sc_metadata` in one of the following ways.

The mapping is straightforward for SSDs supporting NVMe Metadata; the middleware places the `sc_metadata` into the NVMe Metadata part of an NVMe Read/Write Command for reading or writing a page or pages of a cluster. For other SSDs not supporting NVMe Metadata, the middleware partitions the entire NVMe LBA space into two equally-sized LBA subspaces, using the first LBA subspace to store all clusters, and the second LBA subspace to store `sc_metadata`. Let $L = \log_2(S_{\max})$ where $S_{\max}$ is the maximum size of the blob storage, $p(i, j, k)$ be the kth page of the jth cluster of the ith blob, $a(i, j, k)$ be the LBA of $p(i, j, k)$, respectively. Then, the LBA of the secure metadata for the jth cluster of the ith blob is calculated as $a(i, j, 0) + 2^{L-1}$, as shown in Figure 10. Our current middleware and accelerator implementation is based on the latter scheme. Note that more space-efficient subspace management is possible for the latter scheme by packing pieces of `sc_metadata` into consecutive logical blocks in the second LBA subspace to give more room for the first LBA subspace.

The middleware API functions are written in Rust (`https://www.rust-lang.org/`) and listed in Table 7. Since the blob sizes for some VRs, such as BKNTT, can be large, two methods are defined for blob read and write commands. One method specifies a file name as the source and destination of the blob data in `write_blob1` and `read_blob1`, respectively. The other method specifies a memory address as the source and destination of the blob data in `write_blob2` and `write_blob2`, respectively.

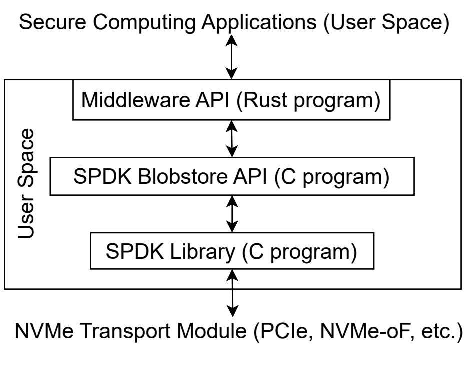

Secure Computing Applications (User Space)
User Space
Middleware API (Rust program)
SPDK Blobstore API (C program)
SPDK Library (C program)
NVMe Transport Module (PCIe, NVMe-oF, etc.)

Figure 9: Middleware architecture

6 Performance Evaluation

Along with our contribution to developing a full-fledged secure computing platform for TFHE using an FPGA-based accelerator, we provide a system-level comparison among FPGA-based, GPU-based and CPU-based secure computing platforms. This section

Table 7: Middleware API functions (`md` is `sc_metadata`, `blobid` is blob identifier, `spdk_blob_op_complete` and `spdk_blob_op_with_id_complete` are callback functions, and `mut* c_void` is mutable pointer to arguments of callback functions.)

Function	Arguments
create_blob	- `md: Metadata` - `cb_fn: spdk_blob_op_with_id_complete` - `cb_arg: *mut c_void`
delete_blob	- `blobid: BlobId` - `cb_fn: spdk_blob_op_complete` - `cb_arg: *mut c_void`
write_blob1	- `blobid: BlobId` - `data_file: String` - `cb_fn: spdk_blob_op_complete` - `cb_arg: *mut c_void`
write_blob2	- `blobid: BlobId` - `data: &mut Vec<u8>` - `cb_fn: spdk_blob_op_complete` - `cb_arg: *mut c_void`
read_blob1	- `blobid: BlobId` - `data_file: String` - `cb_fn: spdk_blob_op_complete` - `cb_arg: *mut c_void`
read_blob2	- `blobid: BlobId` - `data: &mut Vec<u8>` - `cb_fn: spdk_blob_op_complete` - `cb_arg: *mut c_void`

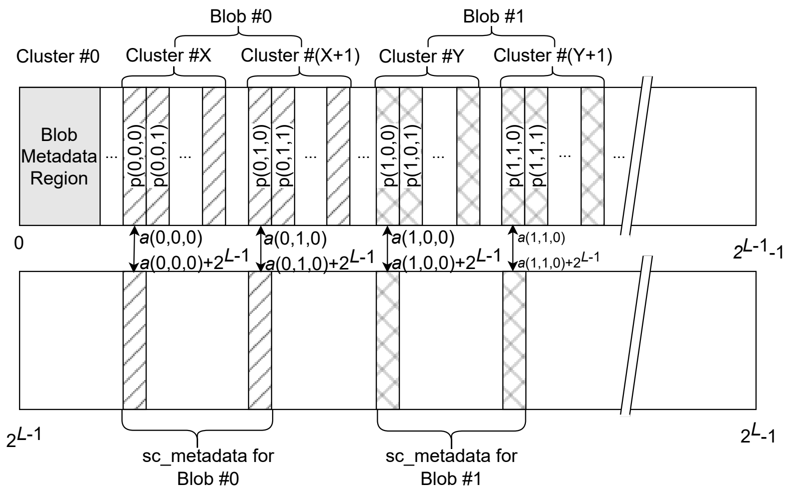


Figure 10: NVMe logical address space map

uses the following TFHE parameters for evaluating our secure computing platform with an FPGA-based accelerator: $n = 800$, $\alpha = 2^{-19}$, $N = 16384$, $k = 1$, $B_g = 2^6$, $l = 5$, $t = 7$ where α is the standard deviation of the noise. This parameter set provides 128-bit classical security [4] using a lattice parameter estimator tool (`https://github.com/malb/lattice-estimator`). Table 8 shows the sizes of fixed-length VRs with this parameter set.

Table 8: Evaluated virtual register sizes (BK: bootstrapping key, BKNTT: NTT-applied bootstrapping key, KSK: key-switching key)

Type	Data Identifier	VR Size in bytes
2 (KEY)	1 (BKNTT)	2.10GB
	2 (KSK)	367MB
3 (TLWE-CoR)	any	3.2KB
4 (TLWE-CoW)	any	3.2KB

6.1 Amount of FPGA resources

Table 9 shows the amount of FPGA resources. BRAM resources are the most utilized resource in the FPGA. Table 10 shows the amount of FPGA logic resources per function. The computing engine uses over triple as many resources as the NVMe bridge.

Table 9: FPGA resources (LUT: Look-Up Table, FF: Flip Flop, BRAM: Block RAM, URAM: Ultra RAM, DSP: Digital Signal Processor)

Resource	Utilization	Available	Utilization (%)
LUTs	625520	1303680	47.98
FFs	763718	2607360	29.29
BRAM Blocks	1265.50	2016	62.72
URAM Blocks	96	960	10.00
DSP Slices	1564	9024	17.33

Table 10: Breakdown of FPGA resources (NB: NVMe Bridge, CE: Computing Engine. Registers are constructed from FFs.)

Name	LUTs	Registers	BRAM Blocks	URAM Blocks	DSP Slices
NB	134504	115600	233	0	9
CE	461123	615303	1023.5	64	1549
Other	29893	32818	8	32	6

6.2 Secure computing instruction execution time

Table 11 shows the average, minimum, and maximum execution times of each secure computing instruction by our accelerator. Table 12 shows the average execution times of GBS for comparing among a software-based platform, a GPU-based platform and our FPGA-based platform.

Our FPGA-based platform uses AMD Ryzen 9 5950X (3.4-4.9GHz/16-core/32-thread/64MB cache) CPU with 128GB RAM as its host-side CPU. We took 10 runs for each secure computing instruction on our accelerator.

The software-based platform uses TFHEpp, an open-source TFHE implementation [24], running on two CPU architectures; one is AMD Ryzen 9 5950X (the same CPU as the host-side CPU of our FPGA-based platform), and the other is Apple M1 (an ARM-based system-on-a-chip (SoC) processor) with 16GB RAM. As the pen-source software, we use the *gatebootstrappingntt* test suite from the TFHPpp [24] with commit c6c5a38, using the same parameter set as our accelerator. We took 10 of measurements for the *gatebootstrappingntt* test suite on the CPU.

As for the GPU, we use Tesla T4 on AWS EC2 g4dn.2xlarge running a modified version of cuFHE library [1] to add support for $N = 16384$. Note that the original cuFHE library only supports GBS for N = 1024 and contains several flaws in their NTT implementation, such as a lack of multiplication by a twiddle factor inside radix-2 butterfly. We also fixed the flaws for fair comparison. We implemented two different schemes for GPU, namely Scheme 1 and Scheme 2. In Scheme 1, each thread performs 1 butterfly calculation at each butterfly stage of NTT/INTT by allocating 8 Streaming Multiprocessors (SMs) for each NTT and INTT, and at most $8(k+1)(= 16)$ NTT or INTT operations run in parallel on 16 SMs. In Scheme 2, each thread sequentially performs 8 butterfly calculations at each stage of NTT/INTT by allocating 1 SM for each NTT, and at most $(k + 1)\ell(= 10)$ NTT or INTT operations run in parallel on 10 SMs. Scheme 1 achieves higher parallelism than Scheme 2, whereas Scheme 2 avoids device-level thread synchronization during GBS processing, including NTT and INTT. In both schemes, there are 1024 threads per SM. Both schemes are implemented to generate less than $1024 \cdot M$ instantaneous threads where M is the maximum number of SMs and $M = 40$ for Tesla T4. For the GPU, we took 100 measurements for GBS.

We note that device-level thread synchronization is required among threads across multiple SMs. We also note that the entire NTT or INTT input or output data for $N = 16384$ coefficients of a polynomial fits into the L2 cache of the GPU device while the data does not fit into the L1 cache of a single SM. In terms of GBS processing time for $N = 16384$, our accelerator outperforms CPU-based platforms and GPU-based platforms by 15 to 120 times and by 2.5 to 3 times, respectively.

Figure 11 shows the GPU and FPGA processing breakdowns of GBS. The GPU is 3 to 4 times slower than the FPGA for processing NTT and INTT, while there is no significant difference for non-NTT/INTT operations. Figure 12 shows the GPU processing breakdown of NTT and INTT. Comparing Scheme 1 and 2 in Figure 12 shows a tradeoff between parallelism and synchronization in GPU. Figure 12 also reveals that NTT and INTT are memory-bandwidth-bound workloads for GPU.

Our FPGA outperforms the GPU in terms of GBS processing time for large degree (such as $N = 16384$) polynomials because (i) our FPGA allows multiple integrated butterfly circuits to access different BRAM blocks in parallel, (ii) our FPGA pipelines butterfly calculation and memory access, and (iii) our FPGA does not require device-level thread synchronization.

6.3 Secure computing program execution time

Table 13 shows the execution time of a secure computation program on our platform. We use the secure computing program listed in Table 5. According to Table 13, since the program contains two Bootstrap instructions, each taking 249.96ms, Bootstrap dominates the overall performance of the execution time of a secure computing program compared

Table 11: Secure computation instruction execution time on our accelerator. The minimum and maximum values for Bootstrap execution time are within ±10us of the average value.

Instruction	Average	Minimum	Maximum
Bootstrap	249.96ms	249.96ms	249.97ms
HomAdd	124us	124us	125us
HomSub	124us	124us	125us
HomIntMult	90us	90us	90us

Table 12: GBS execution time comparison

Software-based Platform		GPU-based Platform		Our FPGA-based Platform
Ryzen 9	Apple M1	Scheme 1	Scheme 2	
3.97s	30.8s	617ms	754ms	250ms

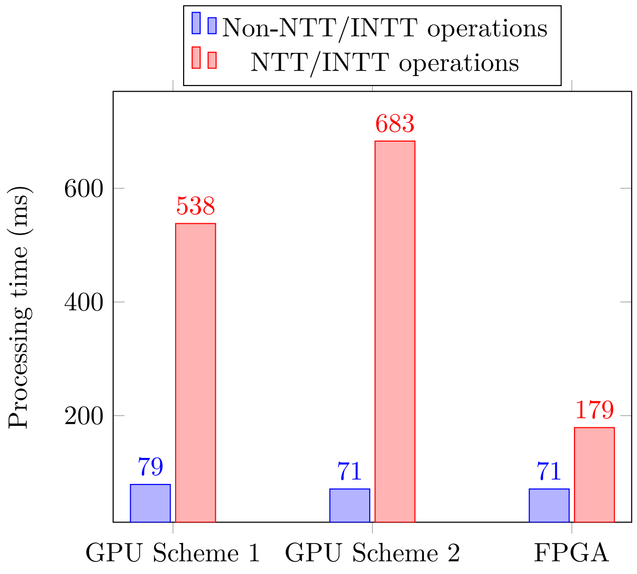


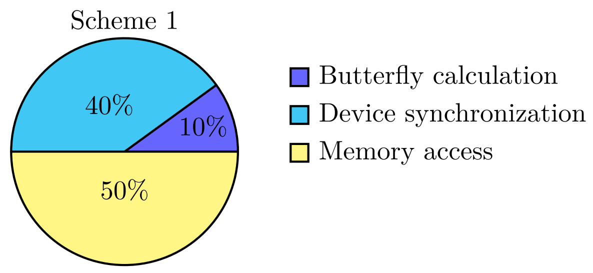
Figure 11: GPU and FPGA Processing Breakdown for GBS

to the execution time of other instructions and the processing time of the NVMe Write Command for invoking the program and writing the program's output to the SSD.

6.4 Power and energy consumption

Tables 14 and 15 show the electric power and energy consumption of the software-based, GPS-based, and FPGA-based platforms, respectively.

The power consumption of the Ryzen 9 CPU is measured using the AMD μProf tool (`https://www.amd.com/en/developer/uprof.html`). The power consumption of the Apple M1 CPU is measured using the Mx Power Gadget tool (`https://www.seense.com/menubarstats/mxpg/`). The power consumption of the GPU is measured using *nvidia-smi*



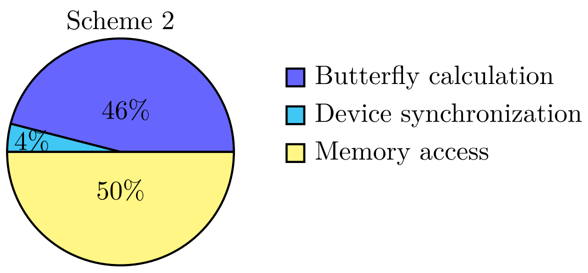


Figure 12: GPU Processing Breakdown for NTT and INTT

Table 13: Secure computation execution time on our platform for a program homomorphically computing xy. The total includes the computing time and the processing time of the NVMe Write Command for invoking the program and writing the TLWE sample carrying the return value to the SSD.

Computing time	Total
500.39ms	502.28ms

command. Since the virtual Performance Monitoring Unit (vPMU) feature is disabled in the AWS hypervisor, the CPU's power data hosting the GPU is unavailable. The power consumption of our FPGA accelerator board is measured using a Tektronics A622 current probe and a PicoScope 3206A oscilloscope. The energy consumption is calculated using the power consumption and the GBS execution time shown in Section 6.2.

Table 16 shows the breakdown of the power consumption on the FPGA chip (XCVU47P) of our accelerator estimated using the Xilinx Vivado tool (https://www.xilinx.com/products/design-tools/vivado.html), with a default toggle rate of 12.5%, where a toggle rate reflects how often outputs of gates change per clock cycle on average. The estimated total on-chip power in Table 16(a) is less than the measured power of the FPGA board during GBS in Table 14 because Table 16(a) is calculated based on the reference clock frequency of 150MHz, whereas we use the feature of dynamic clock reconfiguration of Phase-Locked Loop (PLL) to increase the operating frequency to 200MHz.

Our FPGA-based platform may consume more power when idle or processing GBS than software-based and GPU-based platforms. However, during GBS execution, our platform consumes less energy than the other platforms. Our platform uses 12 times less energy than Ryzen 9 and 7 times less energy than Apple M1. It also uses 1.15 to 1.2 times less energy than GPU. Our platform offers higher GBS throughput per watt than any other platform. As illustrated in Table 16, HBM consumes over 50% of the dynamic power of the FPGA chip. Since NTT and INTT are memory-bandwidth-bound workloads, our accelerator is optimally designed to utilize power where it's most needed.

In future work, we plan to explore implementing a power-saving scheme to reduce energy consumption during idle states.

Table 14: Comparison of power consumption

State	Software-based Platform		GPU		Our FPGA-based Platform
	Ryzen 9	Apple M1	Scheme 1	Scheme 2	CPU+Acclerator (Accelerator)
Idle	18.61W	0.082W	14.36W	14.36W	60.84W (42.23W)
GBS	63.08W	4.64W	43.56W	41.7W	77.79W (59.18W)

Table 15: Comparison on energy consumption per GBS

Software-based Platform		GPU		Our FPGA-based Platform
Ryzen 9	Apple M1	Scheme 1	Scheme 2	CPU+Acclerator (Accelerator)
250.42J	143J	23.44J	22.46J	19.44J (14.80J)

Table 16: Breakdown of the power consumption on FPGA chip

(a) On-Chip Power

Element	Power	%
Hard IP	0.59W	1%
Dynamic	45.63W	88%
Static	5.89W	11%
Total	52.11W	100%

(b) Dynamic Power

Element	Power	%
Clocks	3.71W	8%
Signals	5.75W	13%
Logic	4.47W	10%
BRAM	2.07W	5%
URAM	0.24W	1%
DSP	0.73W	2%
I/O	0.03W	$< 1\%$
HBM	25.27W	52%
Other	3.36W	9%
Total	45.63W	100%

7 Summary and Future Work

We have successfully developed and implemented an exceptionally secure computing platform that utilizes NVMe technology, an FPGA-based TFHE accelerator, SSD, and a middleware on the host side. Our platform stands out from the crowd as it supports a set of secure computing instructions that enable the evaluation of any 14-bit to 14-bit function using TFHE and virtual registers. Our performance evaluations have demonstrated that our platform outperforms CPU and GPU-based platforms by 15 to 120 times and by 2.5 to 3 times, respectively, in gate bootstrapping execution time. Furthermore, our platform has lower electric energy consumption during the gate bootstrapping execution time, outperforming CPU and GPU-based platforms by 7 to 12 times and by 1.15 to 1.2 times, respectively.

Moving forward, we are confident in our ability to develop a compiler and assembler that will convert applications into instructions that can be executed on our secure computing platform via our middleware API. We are also planning to enhance the platform's architecture to include clusters of FPGA-based accelerators and SSDs interconnected by a high-speed network. By using NVMe-oF, we aim to increase our platform's scalability, while Kubernetes will support partial or complete reconfiguration of FPGA and dynamic scheduling of secure computing tasks. Finally, we are confident in our capacity to extend the platform's capabilities to support secure computing for 16-bit to 16-bit functions.

References

[1] V. Group, *Cuda-accelerated fully homomorphic encryption library*, 2019.

[2] Agrawal, R., de Castro, L., Yang, G., Juvekar, C., Yazicigil, R., Chandrakasan, A., Vaikuntanathan, V., and Joshi, A., *FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption*, 2023 IEEE international symposium on high-performance computer architecture (HPCA), 2023, pp. 882–895.

[3] Aikata, A., Mert, C. A., Kwon, S., Deryabin, M., and Roy, S. S., *Reed: Chiplet-based scalable hardware accelerator for fully homomorphic encryption*, 2023.

[4] Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Halevi, S., Hoffstein, J., Laine, K., Lauter, K., Lokam, S., Micciancio, D., Moody. D, Morrison, T, Sahai, A, and Vaikuntanathan, V., *Homomorphic encryption security standard*, HomomorphicEncryption.org, 2018.

[5] Al Badawi, A., Veeravalli, B., Lin, J., Xiao, N., Kazuaki, M., and Mi, A. K. M., *Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters*, IEEE Transactions on Parallel and Distributed Systems **32** (2020), no. 2, 379–391.

[6] Al Badawi, A., Veeravalli, B., Mun, C. F., and Aung, K. M. M., *High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA*, IACR Transactions on Cryptographic Hardware and Embedded Systems (2018), 70–95.

[7] Beirendonck M. V., D'Anvers, J., and Verbauwhede, I., *FPT: a fixed-point accelerator for torus fully homomorphic encryption*, Cryptology ePrint Archive **2022** (2022), 1635.

[8] Bossuat,J.-P., Mouchet, C., Troncoso-Pastoriza, J., and Hubaux, J.-P., *Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys*, Cryptology ePrint Archive **2020** (2020), 1203.

[9] Brakerski, Z., *Fully homomorphic encryption without modulus switching from classical gapsvp*, Advances in cryptology-crypto 2012, 2012, pp. 868–886.

[10] Brakerski, Z., Gentry, C., and Vaikuntanathan, V., *(leveled) fully homomorphic encryption without bootstrapping*, ACM Transactions on Computation Theory (TOCT) **6** (2014), no. 3, 1–36.

[11] de Castro, L., Agrawal, R., Yazicigil, R., Chandrakasan, A., Vaikuntanathan, V., Juvekar, C., and Joshi, A., *Does fully homomorphic encryption need compute acceleration?*, arXiv preprint arXiv:2112.06396 (2021).

[12] Cheon, J. H., Kim, A., Kim, M., and Song, Y., *Homomorphic encryption for arithmetic of approximate numbers*, Advances in cryptology – asiacrypt 2017, 2017, pp. 409–437.

[13] Chillotti, I, Gama N., Georgieva M., and Izabachène, M., *TFHE: Fast fully homomorphic encryption over the torus*, Journal of cryptology, 2020, pp. 34–91.

[14] Chillotti, I., Ligier, D., Orfila, J.-B., and Tap, S., *Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe*, IACR Cryptology ePrint Archive **2021** (2021), 315.

[15] Cousins, D. B., Rohloff, K., and Sumorok, D., *Designing an FPGA-accelerated homomorphic encryption co-processor*, IEEE Transactions on Emerging Topics in Computing **5** (2017), no. 2, 193–206.

[16] Fan, J. and Vercauteren, F., *Somewhat practical fully homomorphic encryption*, IACR Cryptol. ePrint Arch. **2022** (2012), 144.

[17] Gener, S., Newton, P., Tan, D., Richelson, S., Lemieux, G., and Brisk, P., *An FPGA-based programmable vector engine for fast fully homomorphic encryption over the torus*, SPSL: Secure and private systems for machine learning (ISCA workshop), 2021.

[18] Gentry, C., *Fully homomorphic encryption using ideal lattices*, Proceedings of the forty-first annual acm symposium on theory of computing, 2009, pp. 169–178.

[19] Jiang, L., Lou, Q., and Joshi, N., *MATCHA: A fast and energy-efficient accelerator for fully homomorphic encryption over the Torus*, Proceedings of the 59th ACM/IEEE design automation conference, 2022, pp. 235–240.

[20] Latibari, B. S., Gubbi, K. I., Homayoun, H., and Sasan, A., *A survey on FHE acceleration*, 2023 IEEE 16th Dallas circuits and systems conference (DCAS), 2023, pp. 1–6.

[21] Lee, J. H., Zhang, H., Lagrange, V., Krishnamoorthy, P., Zhao, X., and Ki, Y. S., *SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD*, IEEE Computer Architecture Letters **19** (2020), no. 2, 110–113.

[22] Lee, Y., Chung, J., and Rhu, M., *SmartSAGE: Training large-scale graph neural networks using in-storage processing architectures*, arXiv preprint arXiv:2205.04711 (2022).

[23] Marcolla, C., Sucasas, V., Manzano, M., Bassoli, R., Fitzek, F. H. P, and Aaraj, N., *Survey on fully homomorphic encryption, theory, and applications*, Cryptology ePrint Archive **2022** (2022), 1602.

[24] Matsuoka, K., *TFHEpp: pure C++ implementation of TFHE cryptosystem*, 2020.

[25] Matsuoka, K., Banno, R., Matsumoto, N., Sato, T., and Bian, S., *Virtual secure platform: A Five-Stage pipeline processor over TFHE*, 30th USENIX security symposium (USENIX Security 21), 2021, pp. 4007–4024.

[26] Nam, K., Oh, H., Moon, H., and Paek, Y., *Accelerating N-bit Operations over TFHE on Commodity CPU-FPGA*, 2022 IEEE/ACM international conference on computer aided design (ICCAD), 2022, pp. 1–9.

[27] Özerk, Ö., Elgezen, C., Mert, A. C., Öztürk, E., and Savaş, E., *Efficient number theoretic transform implementation on GPU for homomorphic encryption*, The Journal of Supercomputing **78** (2022), no. 2, 2840–2872.

[28] Pöppelmann, T., Oder, T., and Güneysu, T., *High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers*, Progress in cryptology – latincrypt 2015, 2015, pp. 346–365.

[29] Riazi, M. S., Laine, K., Pelton, B., and Dai, W., *HEAX: An architecture for computing on encrypted data*, Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 1295–1309.

[30] Rohloff, K., *The HomomorphicEncryption.org Community and the Applied Fully Homomorphic Encryption Standardization Efforts*, 2023.

[31] Roy, S. S., Turan, F., Jarvinen, K., Vercauteren, F., and Verbauwhede, I., *FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data*, 2019 IEEE International symposium on high performance computer architecture (HPCA), 2019, pp. 387–398.

[32] Samardzic, N., Feldmann, A., Krastev, A., Devadas, S., Dreslinski, R., Peikert, C., and Sanchez, D., *F1: A fast and programmable accelerator for fully homomorphic encryption*, Micro-54: 54th annual ieee/acm international symposium on microarchitecture, 2021, pp. 238–252.

[33] Turan, F., Roy, S. S., and Verbauwhede, I., *HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA*, IEEE Transactions on Computers **69** (2020), no. 8, 1185–1196.

[34] Yang, Y., Lu, H., and Li, X., *Poseidon-NDP: Practical Fully Homomorphic Encryption Accelerator Based on Near Data Processing Architecture*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2023), 1–1.

[35] Zhai, Y., Ibrahim, M., Qiu, Y., Boemer, F., Chen, Z., Titov, A., and Lyashevsky, A., *Accelerating encrypted computing on Intel GPUs*, 2022 IEEE international parallel and distributed processing symposium (IPDPS), 2022, pp. 705–716.

[36] Yufei Xing and Shuguo Li, *A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2021** (2021Feb.), no. 2, 328–356.

[37] Tian Ye, Rajgopal Kannan, and Viktor K. Prasanna, *FPGA acceleration of fully homomorphic encryption over the Torus*, 2022 IEEE high performance extreme computing conference (HPEC), 2022, pp. 1–7.

A Number Theoretic Transform

This section provides equations and algorithms that lead to the NTT and INTT constructions described in 4.5.

A.1 Equations

We derive equations used as the basis for NTT and INTT butterflies shown in Figures 4, 5, and 6. Let $\omega_N = \omega$ and $\psi_N = \psi$. We use the following equations in this section: $\omega_N^2 = \omega_{N/2}$, $\psi_N^2 = \psi_{N/2}$, $\omega_N^N = 1$, and $\omega_N^{N/2} = -1$.

A.1.1 Equations for NTT butterfly

We denote $\mathrm{NTT}_i^{(N)}(f)$ and $\mathrm{DFT}_i^{(N)}(f)$ the ith output of N-point NTT and Discrete Fourier Transform (DFT) for time-domain input vector $f = (f_i)_{0\le i<N}$, respectively.

An NTT butterfly is composed using Gentleman-Sande (GS) butterfly by partitioning the output vector into two subvectors, one containing even elements and the other containing odd elements. Equations used for NTT butterfly are derived as follows.

$$\mathrm{NTT}_i^{(N)}(f) = \psi_N^i \mathrm{DFT}_i^{(N)}(f) = \psi_N^i \sum_{j=0}^{N-1} f_j \omega_N^{ij}$$

To compose an NTT butterfly, we partition the output vector $\mathrm{NTT}^{(N)}(f)$ into two subvectors, one containing even elements of $\mathrm{NTT}^{(N)}(f)$ and the other containing odd elements of $\mathrm{NTT}^{(N)}(f)$. Let $g = (g_j)_{0\le j<N/2} = (f_j + f_{j+N/2})_{0\le j<N/2}$ and $h = (h_j)_{0\le j<N/2} = ((f_j - f_{j+N/2})\psi_N \omega_N^j)_{0\le j<N/2}$.

(i) For $i = 2r$ such that $0 \le r < N/2$,

$$\begin{aligned}
\mathrm{NTT}_{2r}^{(N)}(f) &= \psi_N^{2r} \left(\sum_{j=0}^{N/2-1} f_j \omega_N^{2rj} \right. \\
&\qquad \left. + \sum_{j=0}^{N/2-1} f_{j+N/2} \omega_N^{2r(j+N/2)} \right) \\
&= \psi_{N/2}^r \left(\sum_{j=0}^{N/2-1} f_j \omega_{N/2}^{rj} \right. \\
&\quad \left. + \sum_{j=0}^{N/2-1} f_{j+N/2} \omega_{N/2}^{rj} \right) \\
&= \psi_{N/2}^r \sum_{j=0}^{N/2-1} \left(f_j + f_{j+N/2}\right) \omega_{N/2}^{rj} \\
&= \mathrm{NTT}_r^{(N/2)}(g)
\end{aligned}$$

(ii) For $i = 2r + 1$ such that $0 \le r < N/2$,

$$
\begin{aligned}
\mathrm{NTT}^{(N)}_{2r+1}(f) &= \psi_N^{2r+1}\left(\sum_{j=0}^{N/2-1} f_j \omega_N^{(2r+1)j}\right.\\
&\qquad \left. + \sum_{j=0}^{N/2-1} f_{j+N/2}\omega_N^{(2r+1)(j+N/2)}\right)\\
&= \psi_{N/2}^{r}\psi_N\left(\sum_{j=0}^{N/2-1} f_j \omega_N^{j}\omega_{N/2}^{rj}\right.\\
&\qquad \left. - \sum_{j=0}^{N/2-1} f_{j+N/2}\omega_N^{j}\omega_{N/2}^{rj}\right)\\
&= \psi_{N/2}^{r}\sum_{j=0}^{N/2-1}\left((f_j - f_{j+N/2})\psi_N\omega_N^{j}\right)\omega_{N/2}^{rj}\\
&= \mathrm{NTT}^{(N/2)}_{r}(h)
\end{aligned}
$$

Each pair of f_j and $f_{j+N/2}$ $(0 \le j < N/2)$ are the inputs of an NTT butterfly with a pair of $(f_j + f_{j+N/2})$ and $(f_j - f_{j+N/2})\psi_N\omega_N^j$ as its outputs.

A.1.2 Equations for INTT butterfly

We denote $\mathrm{INTT}^{(N)}_i(F)$ and $\mathrm{uIDFT}^{(N)}_i(F)$ the ith output of N-point, unnormalized Inverse NTT and unnormalized Inverse DFT for frequency-domain input vector $F = (F_i)_{0\le i<N}$, respectively.

Equations used for INTT butterfly are derived as follows.

$$\mathrm{INTT}^{(N)}_i(F) = \mathrm{IDFT}^{(N)}_i(F \odot \Psi^{-1}) = \sum_{j=0}^{N-1}(F_j\psi_N^{-j})\omega_N^{-ij}$$

An INTT butterfly is composed using Cooley-Tukey (CT) butterfly by partitioning the input vector F into two subvectors F_{ev} and F_{od} where F_{ev} contains even elements of F and F_{od} contains odd elements of F. Let $A_i = \mathrm{INTT}^{(N/2)}_i(F_{\mathrm{ev}})$ and $B_i = \mathrm{INTT}^{(N/2)}_i(F_{\mathrm{od}})$ $(0 \le i < N/2)$. The INTT butterfly is composed as follows.

(i) For $0 \leq i < N/2$,

$$\begin{aligned}
\mathrm{INTT}_i^{(N)}(F) &= \sum_{j=0}^{N/2-1} (F_{2j}\psi_N^{-2j})\omega_N^{i(-2j)} \\
&\quad + \sum_{j=0}^{N/2-1} F_{2j+1}\psi_N^{-(2j+1)}\omega_N^{-i(2j+1)} \\
&= \sum_{j=0}^{N/2-1} F_{2j}\psi_{N/2}^{-j}\omega_{N/2}^{-ij} \\
&\quad + \psi_N^{-1}\omega_N^{i} \sum_{j=0}^{N/2-1} F_{2j+1}\psi_{N/2}^{-j}\omega_{N/2}^{-ij} \\
&= \mathrm{INTT}_i^{(N/2)}(F_{\mathrm{ev}}) + \psi_N^{-1}\omega_N^{i}\mathrm{INTT}_i^{(N/2)}(F_{\mathrm{od}}) \\
&= A_i + \psi_N^{-1}\omega_N^{i}B_i
\end{aligned}$$

(ii) For $N/2 \leq i < N$,

$$\begin{aligned}
\mathrm{INTT}_i^{(N)}(F) &= \sum_{j=0}^{N/2-1} (f_{2j}\psi_N^{-2j})\omega_N^{i(-2j)} \\
&\quad + \sum_{j=0}^{N/2-1} F_{2j+1}\psi_N^{-(2j+1)}\omega_N^{-(N/2+i)(2j+1)} \\
&= \sum_{j=0}^{N/2-1} F_{2j}\psi_{N/2}^{-j}\omega_{N/2}^{-ij} \\
&\quad - \psi_N^{-1}\omega_N^{-i} \sum_{j=0}^{N/2-1} F_{2j+1}\psi_{N/2}^{-j}\omega_{N/2}^{-ij} \\
&= \mathrm{INTT}_i^{(N/2)}(F_{\mathrm{ev}}) - \psi_N^{-1}\omega_N^{-i}\mathrm{INTT}_i^{(N/2)}(F_{\mathrm{od}}) \\
&= A_i - \psi_N^{-1}\omega_N^{-i}B_i
\end{aligned}$$

Each pair of A_i and B_i are the inputs of an INTT butterfly having a pair of $\left(A_i + \psi_N^{-1}\omega_N^{-i}B_i\right)$ and $\left(A_i - \psi_N^{-1}\omega_N^{-i}B_i\right)$ as its outputs for $0 \leq i < N/2$ and $N/2 \leq i < N$, respectively.

A.2 Algorithms

Algorithm 1 below computes the NTT of a vector of length N. Algorithm 1 computes in place. Outputs from Algorithm 1 remain in bit-reversed order; we remind the reader that our goal is to compute the convolution of polynomials represented as vectors. Readers can refer to Equation (1) from Section 4.5 for the definition of convolution. If the bits of an output from Algorithm 1 are in canonical order, it would incur an additional cost when computing convolutions. Algorithms 1 and 2 do not output normalized transforms to save time when calculating convolutions.

Algorithm 1 Number Theoretic Transform

```
Input: $N$, length of transform ($N$ is a power of 2.)
Input: $\Phi = (\psi^{N/2^{r+1}}\omega^{jN/2^{r+1}})_{0 \leq r < \log_2 N, 0 \leq j \leq r}$, two-dimensional list of pre-computed
    twiddles with the second dimension listed in bit-canonical order.
```

Input: **a**, data vector of length N in bit-canonical order
Output: NTT(a) in bit-reversed order

```
1:  $m \leftarrow N/2$
2:  $r \leftarrow \log_2 N - 1$                          // NTT Row number minus 1
3:  $k \leftarrow 1$
4:  while $m \geq 1$ do
5:      for $0 \leq i < m$ do
6:          $j_1 \leftarrow 2ik$
7:          $j_2 \leftarrow j_1 + k - 1$               // Interval length is $k - 1$
8:          for $j_1 \leq j \leq j_2$ do              // Butterfly operations here
9:              $t \leftarrow a_j$
10:             $u \leftarrow a_{j+k}$
11:             $a_j \leftarrow (t + u) \bmod p$
12:             $a_{j+k} \leftarrow (t - u)\Phi_{r,j} \bmod p$
13:         end for
14:     end for
15:     $m \leftarrow m/2$
16:     $r \leftarrow r - 1$
17:     $k \leftarrow 2k$
18: end while
```

Algorithm 2 Inverse Number Theoretic Transform

Input: N, length of transform (N is a power of 2.)
Input: $\Phi^* = (\psi^{-N/2^{r+1}}\omega^{-jN/2^{r+1}})_{0 \leq r < \log_2 N, 0 \leq j \leq r}$, two-dimensional list of pre-computed twiddles with the second dimension listed in bit-canonical order.
Input: **a**, data vector of length N in bit-reversed order
Output: INTT(a) in bit-canonical order

```
1:  $m \leftarrow 1$
2:  $r \leftarrow 0$                                  // INTT Row number minus 1
3:  $k \leftarrow N/2$
4:  while $m < N$ do
5:      for $0 \leq i < m$ do
6:          $j_1 \leftarrow 2ik$
7:          $j_2 \leftarrow j_1 + k - 1$               // Interval length is $k - 1$
8:          for $j_1 \leq j \leq j_2$ do              // Butterfly operations here
9:              $t \leftarrow a_j$
10:             $u \leftarrow a_{j+k}\Phi^*_{r,j}$
11:             $a_j \leftarrow t + u \bmod p$
12:             $a_{j+k} \leftarrow t - u \bmod p$
13:         end for
14:     end for
15:     $m \leftarrow 2m$
16:     $r \leftarrow r + 1$
17:     $k \leftarrow k/2$
18: end while
```

Algorithms 1 and 2 are similar to Algorithms 8 and 7, respectively, from [28]. However, we point out that Algorithms 7 and 8 from [28] contain errors corrected here.