The Journal of Supercomputing
https://doi.org/10.1007/s11227-021-03980-5

Efficient number theoretic transform implementation on GPU for homomorphic encryption

Özgün Özerk[1] · Can Elgezen[1] · Ahmet Can Mert[1] · Erdinç Öztürk[1] · Erkay Savaş[1]

Accepted: 28 June 2021

Abstract

Lattice-based cryptography forms the mathematical basis for current homomorphic encryption schemes, which allows computation directly on encrypted data. Homomorphic encryption enables privacy-preserving applications such as secure cloud computing; yet, its practical applications suffer from the high computational complexity of homomorphic operations. Fast implementations of the homomorphic encryption schemes heavily depend on efficient polynomial arithmetic, multiplication of very large degree polynomials over polynomial rings, in particular. Number theoretic transform (NTT) accelerates large polynomial multiplication significantly, and therefore, it is the core arithmetic operation in the majority of homomorphic encryption scheme implementations. Therefore, practical homomorphic applications require efficient and fast implementations of NTT in different computing platforms. In this work, we present an efficient and fast implementation of NTT, inverse NTT and NTT-based polynomial multiplication operations for GPU platforms. To demonstrate that our GPU implementation can be utilized as an actual accelerator, we experimented with the key generation, the encryption and the decryption operations of the Brakerski/Fan–Vercauteren (BFV) homomorphic encryption scheme implemented in Microsoft's SEAL homomorphic encryption library on GPU, all of which heavily depend on the NTT-based polynomial multiplication. Our GPU implementations improve the performance of these three BFV operations by up to 141.95×, 105.17× and 90.13×, respectively, on Tesla `v100` GPU compared to the highly optimized SEAL library running on an Intel i9-7900X CPU.

Keywords Lattice-based cryptography · Homomorphic encryption · SEAL · Number theoretic transform · Polynomial multiplication · GPU · CUDA

This work is supported by TÜBİTAK under Grant Number 118E725.

✉ Ahmet Can Mert
ahmetcanmert@sabanciuniv.edu

Extended author information available on the last page of the article

Published online: 13 July 2021

1 Introduction

Lattice-based cryptography is conjectured to be secure against attacks from quantum computers and thus supports post-quantum cryptography (PQC). Also, it provides the mathematical basis for fully homomorphic encryption (FHE) schemes, as demonstrated by Gentry in 2009 [21]. FHE allows computation on the encrypted data requiring neither decryption nor secret key and, therefore, enables secure processing of sensitive data. FHE offers a variety of applications ranging from private text classification to secure cloud computing [2].

Since Gentry's breakthrough, homomorphic encryption has gained tremendous amount of attention and different homomorphic encryption schemes are proposed in the literature such as Brakerski–Gentry–Vaikuntanathan (BGV) [13], Brakerski/Fan–Vercauteren (BFV) [19] and Cheon-Kim-Kim-Song (CKKS) [15]. There are also various efforts for developing their practical implementations. As such, there are different open-source and highly optimized software libraries such as SEAL [41], HElib [25] and PALISADE [35] for homomorphic encryption and computation. The SEAL library is developed by Microsoft Research and it supports the BFV and the CKKS schemes. HElib supports the BGV and the CKKS schemes while PALISADE supports the BGV, the BFV and the CKKS schemes.

Although the potential applications of FHE are of groundbreaking nature, its high algorithmic complexity is a standing impediment for efficient and practical implementations thereof. Among different core arithmetic operations in various FHE schemes, multiplication over polynomial rings is probably the most time-consuming. Therefore, there are different methods in the literature proposed for the efficient implementation of multiplication of two very large degree polynomials over polynomial ring $\mathbf{R}_{q,n}$, where n and q represent the degree of polynomials in the ring and coefficient modulus, respectively. The Toom-Cook [44] or Karatsuba [26] multiplications have been methods in use for a long time and generally utilized in schemes with polynomial rings, for which NTT is not suitable [32]. NTT-based polynomial multiplication is, on the other hand, highly utilized in lattice-based cryptosystems and it reduces the $\mathcal{O}(n^2)$ computational complexity of the schoolbook polynomial multiplication to $\mathcal{O}(n \cdot \log n)$ [16].

Although utilizing NTT improves the performance of polynomial multiplication operation, it is still inefficient for real life applications. Therefore, there are different NTT-based polynomial multiplication implementations proposed in the literature for efficient and practical lattice-based cryptosystems on different platforms: hardware architectures [33, 37, 38, 43], software implementations [1, 25, 41, 42], and implementations on GPUs [3–6, 17, 23, 24, 28, 29, 40, 46]. There are also hybrid approaches combining NTT-based and Karatsuba multiplication methods for the polynomial multiplication operation [7]. The NTT-based polynomial multiplication operation can be performed for a range of parameters n and q in different applications. For example, FHE applications require usually large n and q parameters while PQC utilizes smaller parameters. Therefore, an efficient implementation of NTT-based polynomial multiplication requires flexibility of supporting both FHE and PQC in addition to offering high-performance.

With a similar motivation of the works in the literature, we propose efficient NTT, INTT and NTT-based polynomial multiplication implementations on GPU in this work. The proposed implementations on GPU support a wide range of polynomial rings. The proposed implementations can perform a single NTT and INTT operations in $39\mu s$ and $23\mu s$, respectively, for the largest ring with $n = 32768$ and $\log_2(q) = 61$ in Tesla `V100` GPU including overhead of kernel calls.

In order to show that the proposed GPU implementations can be useful as actual accelerators in the homomorphic encryption schemes, for proof of concept, the proposed implementations are utilized to implement and accelerate the key generation, the encryption and the decryption operations of the BFV homomorphic encryption scheme on GPU.

For a quick recap, our contributions are listed as follows:

1. We present high-performance and efficient GPU implementations for NTT, INTT and NTT-based polynomial multiplication operations. The proposed GPU implementations support polynomials of degrees ranging from 2048 to 32768 with 30-bit and 61-bit coefficients[1]. We run the implementations on three different GPU platforms, Nvidia GTX `980`, Nvidia GTX `1080`, Nvidia Tesla `V100`, and a single NTT operation for polynomials of degree 32768 with 61-bit coefficients is performed in $73\mu s$, $36\mu s$, $39\mu s$ on Nvidia GTX `980`, Nvidia GTX `1080` and Nvidia Tesla `V100`, respectively.
2. We modified the NTT and INTT algorithms significantly for efficient *thread* usage on GPU and have come up with single-kernel and multi-kernel approaches which perform relatively better on small and large degree polynomials, respectively. The single-kernel approach allocates a single GPU *block* with 1024 GPU *threads* to process a single polynomial in a single kernel call while making use of fast *shared* memory. The multi-kernel approach, on the other hand, uses multiple *blocks* and $n/2$ *threads* and performs $\log_2(n)$ kernel calls for one NTT/INTT operation. As the main focus of our work, we introduced a novel hybrid design which combines single and multi-kernel approaches. It performs better than both of them for all polynomial degrees. The hybrid design first employs multi-kernel approach, then it switches to single-kernel approach as soon as the NTT/INTT block becomes sufficiently small. The hybrid approach achieves up to 9× speed up for different polynomial degrees compared to multi-kernel approach.
3. The key generation, encryption and decryption operations of the BFV scheme are fully implemented on GPU and compared to the BFV implementation on Microsoft's SEAL library running on an Intel i9-7900X CPU, and we observed up to 60.31×, 43.84×, 33.89× speed-up values on Nvidia GTX `980`; 56.07×, 40.85×, 25.05× speed-up values on Nvidia GTX `1080`; and 141.95×, 105.17×, 90.13× speed-up values on Nvidia Tesla `V100`, for key generation, encryption and decryption operations, respectively.

[1] A sample code is available at https://github.com/SU-CISEC/gpu-ntt.

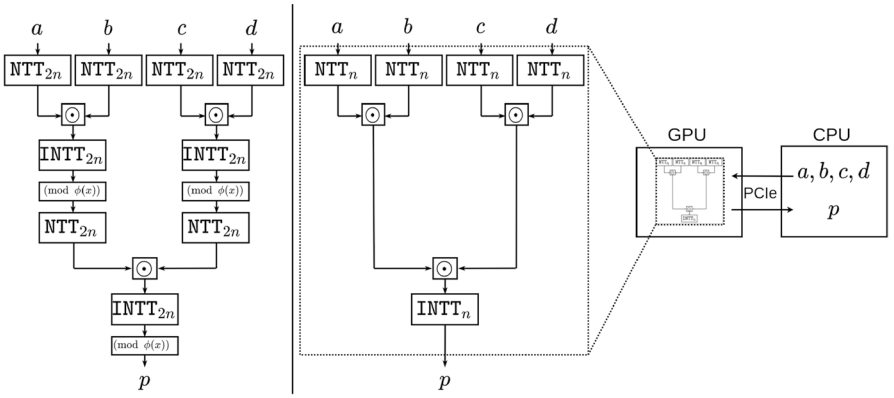


Fig. 1 Multiplication of four polynomials shown in the system architecture model of our accelerator infrastructure

4. Since key generation and encryption operations of the BFV scheme require random polynomials from uniform, ternary and discrete Gaussian distributions, we also introduced an implementation of random polynomial sampler for these distributions on GPU. We utilized Salsa20 implementation [22] for pseudo-random number generation for uniform distribution and inverse cumulative standard distribution function `normcdfinvf` in CUDA Math API.
5. Our implementation does not utilize a carrier prime unlike most of the other works in the literature [3, 17, 23, 27, 46]. Thus, instead of switching to NTT domain, then going back to polynomial domain using INTT for each multiplication operation, we can stay in the NTT domain and perform arbitrary number of operations without having the necessity of switching back to polynomial domain via INTT. This is illustrated in Fig. 1 where our approach shown on the right performs less operation.

The rest of the paper is organized as follows. Section 2 presents the notation, the background on NTT and polynomial multiplication operations and summarizes prior works in the literature. Section 3 introduces SEAL library. Section 4 presents the proposed GPU implementations. Section 5 presents the results and Sect. 6 concludes the paper.

2 Background

In this section, we present the notation used in the rest of the paper, brief descriptions of NTT, INTT, polynomial multiplication operations, structure of GPUs and prior works in the literature.

2.1 Notation

The ring $\mathbb{Z}_q$ consists of the set of integers $\{0, 1, \ldots, q-1\}$. Let the polynomial ring $\boldsymbol{R}_q = \mathbb{Z}_q[x]/\phi(x)$ represent all the polynomials reduced with the irreducible polynomial $\phi(x)$ with coefficients in $\mathbb{Z}_q$. When $\phi(x)$ is in the form $(x^n + 1)$, the polynomial ring $\mathbb{Z}_q[x]/(x^n + 1)$ is represented with $\boldsymbol{R}_{q,n}$, which consists of polynomials of degree at most $(n-1)$ with coefficients in $\mathbb{Z}_q$. For example, $\boldsymbol{R}_{17,32} = \mathbb{Z}_{17}[x]/(x^{32} + 1)$ represents the polynomials of degree at most 31 with coefficients in $\mathbb{Z}_{17}$.

A polynomial $\boldsymbol{a}(x) = \sum_{i=0}^{n-1} a_i \cdot x^i$ in $\boldsymbol{R}_{q,n}$ can also be represented as a vector over $\mathbb{Z}_q$, $\boldsymbol{a} = [a_0, a_1, \ldots, a_{n-1}]$, where i represents the position of the coefficient such that $0 \leq i < n$. Similarly, we use $\boldsymbol{a}[i]$ to represent the coefficient of polynomial $\boldsymbol{a}$ at position i. Throughout the paper, we represent an integer and a polynomial with regular lowercase (e.g., a) and boldface lowercase (e.g., $\boldsymbol{a}$), respectively. Vectors in NTT domain are represented with a bar over their symbols. For example, $\bar{\boldsymbol{a}}$ represents the NTT domain representation of vector $\boldsymbol{a}$. Let $\cdot, \times$ and $\odot$ represent integer, polynomial and coefficient-wise vector multiplication, respectively. Let $(\boldsymbol{a} \cdot b)$ and $(\boldsymbol{a} + b)$ represent that coefficients of polynomial $\boldsymbol{a}$ are multiplied and added with integer b, respectively, if one of the operands is a single integer. Let $\boldsymbol{a} \leftarrow \boldsymbol{R}_{q,n}$ and $\boldsymbol{a} \xleftarrow{\$} \mathbf{S}$ represent that the polynomial $\boldsymbol{a}$ is sampled uniformly from $\boldsymbol{R}_{q,n}$ and from the set $\mathbf{S}$, respectively. Let $\lfloor . \rceil, \lceil . \rceil, \lfloor . \rfloor$ and $[.]_q$ represent round to nearest integer, rounding up, rounding down and the reduction by modulo q operations, respectively. For the rest of the paper, q and n represent the coefficient modulus and the degree of the polynomial ring, respectively.

2.2 Number theoretic transform

NTT is defined as discrete Fourier transform (DFT) over the ring $\mathbf{Z}_q$ and any efficient DFT algorithm can be adopted as an NTT algorithm. An n-point (pt) NTT operation transforms an n element vector $\boldsymbol{a}$ to another n element vector $\bar{\boldsymbol{a}}$ as defined in Eq. 1.

$$\bar{\boldsymbol{a}}_i = \sum_{j=0}^{n-1} a_j \cdot \omega^{ij} \pmod{q} \text{ for } i = 0, 1, \ldots, n-1. \tag{1}$$

The NTT calculations involve the constant called twiddle factor, $\omega \in \mathbb{Z}_q$, which is also defined as the n-th root of unity. The twiddle factor satisfies the conditions $\omega^n \equiv 1 \pmod{q}$ and $\omega^i \neq 1 \pmod{q}\ \forall i < n$, where $q \equiv 1 \pmod{n}$.

Similarly, the INTT operation uses almost the same formula as NTT operation as shown in Eq. 2 except that $\omega^{-1} \pmod{q}$, which is the modular inverse of ω in $\mathbb{Z}_q$, is used instead of ω and the resulting coefficients need to be multiplied with $n^{-1} \pmod{q}$ in $\mathbb{Z}_q$.

$$\boldsymbol{a}_i = \frac{1}{n} \sum_{j=0}^{n-1} \bar{a}_j \cdot \omega^{-ij} \pmod{q} \text{ for } i = 0, 1, \ldots, n-1. \tag{2}$$

Applying NTT and INTT operations as in Eqs. 1 and 2 leads to high computational complexity. Therefore, there are many efficient and fast implementations of NTT operation in the literature [17, 20, 30, 34], constructed around two very well-known approaches: *decimation in time* (DIT) and *decimation in frequency* (DIF) FFTs. The former and latter FFT operations utilize Cooley–Tukey (CT) and Gentleman–Sande (GS) butterfly structures, respectively [16].

2.3 Number theoretic transform based polynomial multiplication

The multiplication of polynomials $\boldsymbol{a}(x)$ and $\boldsymbol{b}(x)$ can be computed using schoolbook polynomial multiplication as shown in Eq. 3. When the polynomial multiplication is performed in $\boldsymbol{R}_q$, the resulting polynomial $\boldsymbol{c}(x)$ should be reduced by $\phi(x)$.

$$\boldsymbol{c}(x) = \boldsymbol{a}(x) \times \boldsymbol{b}(x) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} a_i \cdot b_j \cdot x^{i+j} \tag{3}$$

NTT and INTT operations enable efficient implementation of polynomial multiplication operation by converting the schoolbook polynomial multiplication operation into coefficient-wise multiplication operations as shown in Eq. 4 where $\texttt{NTT}_{2n}$ and $\texttt{INTT}_{2n}$ represent $2n$-pt NTT and $2n$-pt INTT operations, respectively. However, this requires doubling the sizes of input polynomials with zero-padding and there still should be a separate polynomial reduction operation by $\phi(x)$.

$$\boldsymbol{c}(x) = \texttt{INTT}_{2n}(\texttt{NTT}_{2n}(\boldsymbol{a}(x)) \odot \texttt{NTT}_{2n}(\boldsymbol{b}(x))) \bmod \phi(x) \tag{4}$$

When the polynomial ring is $\mathbf{R}_{q,n}$ where $q \equiv \pmod{2n}$, a technique called *negative wrapped convolution* is utilized, which eliminates the need for doubling the input sizes and the polynomial reduction operation as shown in Eqs. 5–8.

$$\hat{\boldsymbol{a}}(x) = [a_0, a_1, \ldots, a_{n-1}] \odot [\psi^0, \psi^1, \ldots, \psi^{(n-1)}] \tag{5}$$

$$\hat{\boldsymbol{b}}(x) = [b_0, b_1, \ldots, b_{n-1}] \odot [\psi^0, \psi^1, \ldots, \psi^{(n-1)}] \tag{6}$$

$$\hat{\boldsymbol{c}}(x) = \texttt{INTT}_n(\texttt{NTT}_n(\hat{\boldsymbol{a}}(x)) \odot \texttt{NTT}_n(\hat{\boldsymbol{b}}(x))) \tag{7}$$

$$\boldsymbol{c}(x) = [\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_{n-1}] \odot [\psi^0, \psi^{-1}, \ldots, \psi^{-(n-1)}] \tag{8}$$

However, this requires the coefficients of input and output polynomials to be multiplied with $[\psi^0, \psi^1, \ldots, \psi^{(n-1)}]$ and $[\psi^0, \psi^{-1}, \ldots, \psi^{-(n-1)}]$, which are usually referred as preprocessing and post-processing, respectively. The constant ψ is called $2n$-th root of unity satisfying the conditions $\psi^{2n} \equiv 1 \pmod{q}$ and $\psi^i \neq 1 \pmod{q}$ $\forall i < 2n$, where $q \equiv 1 \pmod{2n}$.

Roy et al. [39] merged preprocessing and NTT operations by employing DIT NTT operation utilizing CT butterfly structure, which takes the input in standard order and produces the output in bit-reversed order. The algorithm for merged preprocessing and NTT operations is shown in Algorithm 6, where $br(k)$ function performs bit-reversal on $\log_2(n)$-bit input k. We refer this operation as NTT for the rest of the paper.

Algorithm 1 Merged In-place NTT Algorithm

Input: $\boldsymbol{a}(x) \in \boldsymbol{R}_q^n$ in natural-order
Input: ψ_{rev} (powers of ψ stored in bit-reversed order where $\psi_{rev}[k] = \psi^{br(k)} \pmod q$)
Output: $\overline{\boldsymbol{a}}(x) \in \boldsymbol{R}_q^n$ in bit-reversed order

$t = n$
for $(m = 1;\ m < n;\ m = 2m)$ **do**
  $t = t/2$
  for $(i = 0;\ i < m;\ i++)$ **do**
    $j_1 = 2 \cdot i \cdot t$
    $j_2 = j_1 + t - 1$
    for $(j = j_1;\ j \leq j_2;\ j++)$ **do**
      $U = \boldsymbol{a}_j$
      $V = \boldsymbol{a}_{j+t} \cdot \psi_{rev}[m + i] \pmod q$
      $\boldsymbol{a}_j = U + V \pmod q$
      $\boldsymbol{a}_{j+t} = U - V \pmod q$
    end for
  end for
end for
return a

In [36], Pöppelmann et al. merged INTT and post-processing operations by employing DIF NTT operation utilizing the GS butterfly, which takes the input in bit-reversed order and produces the output in standard order. The algorithm for merged INTT and post-processing operations is shown in Algorithm 2, and we refer this operation as INTT for the rest of the paper. Using both techniques, preprocessing and post-processing operations can be eliminated at the expense of using two different butterfly structures as shown in Eq. 9, which formulates the NTT-based polynomial multiplication.

$$\boldsymbol{c} = \texttt{INTT}_n(\texttt{NTT}_n(\boldsymbol{a}(x)) \odot \texttt{NTT}_n(\boldsymbol{b}(x))) \tag{9}$$

Algorithm 2 Merged In-place INTT Algorithm

Input: $\overline{\boldsymbol{a}}(x)\boldsymbol{R}_q^n$ in bit-reversed order
Input: ψ_{rev}^{-1} (powers of ψ^{-1} stored in bit-reversed order where $\psi_{rev}^{-1}[k] = \psi^{-br(k)} \pmod q$)
Output: $\boldsymbol{a}(x) \in \mathbf{R}_q^n$ in natural-order

$t = 1$
for $(m = n;\ m > 1;\ m = m/2)$ **do**
  $j_1 = 0$
  $h = m/2$
  for $(i = 0;\ i < h;\ i++)$ **do**
    $j_2 = j_1 + t - 1$
    for $(j = j_1;\ j \leq j_2;\ j++)$ **do**
      $U = \overline{\boldsymbol{a}}_j$
      $V = \overline{\boldsymbol{a}}_{j+t}$
      $\overline{\boldsymbol{a}}_j = U + V \pmod q$
      $\overline{\boldsymbol{a}}_{j+t} = (U - V) \cdot \psi_{rev}^{-1}[h + i] \pmod q$
    end for
  end for
  $t = 2 \cdot t$
end for
return $\overline{a}$

For the rest of the paper, we use $\texttt{NTT}_n$ and $\texttt{INTT}_n$ for representing n-pt merged NTT and INTT operations, respectively.

2.4 Graphical processing unit (GPU)

Inner structures of NTT and INTT algorithms offer opportunities for parallelization. An in-depth insight of GPU organization and its fundamental working principals is of significant importance for understanding the techniques and algorithms suitable for GPU implementation introduced in the subsequent sections, which take advantage of inherently parallelizable nature of NTT and INTT algorithms.

While GPUs consist of many more cores than CPUs, GPU cores are significantly slower (less powerful) than CPU cores. Thus, GPUs provide us with a plausible alternative, when it comes to performing overly many, relatively simple operations. On the other hand, when the operation at hand is complex and not possible to be partitioned into smaller, concurrently executable parts, CPU stands always a much better alternative.

Kernel in the context of CUDA is a function that is called by the host (CPU) to execute on the device (GPU). Kernels run on streams, each of which can be considered as a single operation sequence. When run on different streams, kernels execute in parallel.

There are three important abstractions in kernels: *grid*, *block* and *thread*, which are illustrated in Fig. 2. Hierarchically organized, grids consist of blocks, which in turn consist of threads. Each grid can contain a certain maximum number of blocks (actual number varies depending on the GPU model), and these blocks can be organized inside the grid into 1- or 2-dimensional arrays. Every block has an ID

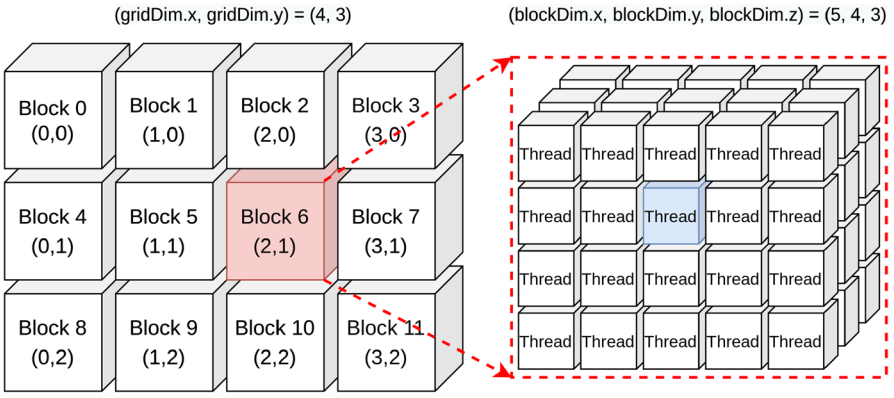


Fig. 2 Architectural overview of a GPU

determined by the number of dimensions. As shown in Fig. 2, a grid consisting of 12 blocks is organized as an array of 3 rows and 4 columns. The shaded block with ID (2, 1) in Fig. 2 indicates that it is in the first row and the second column of the grid (in other words, it's the 6^th block). Similar to grids, blocks can contain a certain maximum number of threads (again varies depending on the GPU model), and they can be organized into 1-, 2- or 3-dimensions arrays inside the block. Threads in a block also have indices, which are used to access a specific thread [e.g., the shaded thread in Fig. 2 has the ID derived from its indices (2, 1, 0)] within the block.

Each block is scheduled on a computational unit, referred as *streaming multiprocessor* (SM). After the scheduling, the blocks are then sliced into *warps*, each of which consists of 32 threads. A SM then runs the warps in an arbitrary order determined by the warp-scheduler in the SM.

Global memory is the largest memory partition in GPU, accessible by all threads. Among other memory alternatives on GPU, communication with the *global* memory is the most time-consuming; therefore, access to it should be substituted with other memory alternatives, if possible. The data stored in the *constant* memory are cached and cannot be changed during the execution of operations. The constant memory is significantly faster than the *global* memory. The assignment of each block to SM allows another alternative in the memory hierarchy, the *shared* memory. Unlike *constant* memory, *shared* memory can be modified during execution. It is again faster than the *global* memory, but only accessible by the threads within the same block. In other words, a thread cannot access other blocks' *shared* memories. *Registers* are the fastest, yet smallest storage units, which are private to the threads. A thread can neither read or write other thread's *registers*. The only exception is that threads can read each others' registers only if they are in the same warp. All these details pertaining to memory access rights are summarized in Table 1.

Table 1 Summary of memory hierarchy inside GPU and their relationships with threads

Memory Type	A thread can	
	Read	Write
Global Memory	✓	✓
Texture Memory	✓	–
Constant Memory	✓	–
Shared Memory (of its block)	✓	✓
Shared Memory (of another block)	–	–
Registers of other threads (in another warp)	–	–
Registers of other threads (in the same warp)	✓	–
Registers of itself	✓	✓

2.5 Prior works

Efficient implementations of NTT and NTT-based polynomial multiplication operations for various platforms (GPU, FPGA, CPU) and applications (PQC, FHE) have been studied in the literature. GPUs offer a wide variety of optimizations for efficient implementations and they are profitably utilized for accelerating intrinsically parallelizable operations. Moreover, some works in the literature employ more than one GPU by adapting their designs to take advantage of larger degree of parallelism offered by multiple GPUs. For instance, Badawi et al. [5] provided the performance evaluation results of their implementation for a variant of the BFV scheme on such multi-GPU design. Another strategy is to take advantage of the memory hierarchy in GPUs. Overall speed of execution can be significantly improved by utilizing faster memory types that are located on the higher part of memory hierarchy (i.e., *shared*) and reducing the communication with global memory [17, 46]. Goey et al. [23] proposed a GPU accelerator for NTT operations utilized in FHE applications. Besides very-well known strategies for optimizing memory access in GPUs, they also utilized *warp shuffle*, which allows threads in the same warp to read from each others registers. This enables faster data access than both global and shared memories [23]. Lee et al. [28] proposed a method to improve the performance of NTT by eliminating the recursive part of the Nussbaumer algorithm with nega-cyclic convolution, along with the optimizations on the non-coalesced memory access pattern. Lee et al. focused on mitigating the warp divergence problem of the NTT operation and they utilized NTT operation in qTESLA, which is a lattice-based post-quantum digital signature scheme [29]. In [24], Gupta et al. proposed a GPU accelerator for PQC schemes NewHope and Kyber with an efficient implementation of NTT. In [27], Kim et al. analyzed and improved the performance of NTT operation by introducing on-the-fly twiddle factor generation for FHE schemes. Sahu's work [40] focuses on a high-performance implementation of proxy re-encryption schemes on Nvidia GPUs by designing a parallel NTT procedure. Dai et al. [17] proposed cuHE, which is a complete homomorphic encryption library written with CUDA for GPU platforms, creating a solid reference point for future works. Our proposed GPU implementations are compared with these works in Sect. 5.

Last but not least, Alves' [8] work combines discrete Galois transform (DGT) with double CRT, thus representing an alternative approach compared to concentrating on NTT. Since they are not focusing NTT on their work, it is not possible to perform a comparison between our work and theirs.

3 Microsoft's SEAL homomorphic encryption library [41]

The SEAL homomorphic encryption library is developed by Cryptography Research Group at Microsoft Research, which enables fast and efficient homomorphic applications for a variety of applications ranging from private information retrieval to secure neural network inference [9, 14]. SEAL supports two homomorphic encryption schemes, BFV and the CKKS, for implementing homomorphic operations, where the former works with integers while the latter enables homomorphic arithmetic using real numbers. Since, in this work, we utilize the proposed GPU implementation for accelerating the operations of the BFV scheme, details of the CKKS scheme will not be presented in this section.

3.1 BFV homomorphic scheme

BFV [19] is a homomorphic encryption scheme proposed by Fan *et al.*, which extends Brakerski's construction [12]. It is based on ring learning with errors (RLWE) problem [31], and it involves intensive polynomial arithmetic. Let the plaintext and ciphertext spaces be $\boldsymbol{R}_{t,n}$ and $\boldsymbol{R}_{q,n}$, respectively, for some integer $t > 1$, where neither q nor t has to be prime. Suppose $\Delta = \lfloor q/t \rfloor$ and let χ represent discrete Gaussian distribution. Also, while the symbol $\leftarrow$ represents random sampling from uniform distribution, $\overset{\$}{\leftarrow}\chi$ stands for sampling from the distribution χ. Key generation, encryption and decryption operations described in the textbook-BFV scheme are shown below.

– Key Generation: $\boldsymbol{s} \leftarrow \boldsymbol{R}_{2,n}$, $\boldsymbol{a} \leftarrow \boldsymbol{R}_{q,n}$ and $\boldsymbol{e} \overset{\$}{\leftarrow}\chi$,

$sk = \boldsymbol{s}$, $pk = (p_0, p_1) = ([-(a \times s + e)]_q, a)$.

– Encryption: $\boldsymbol{m} \in \boldsymbol{R}_{t,n}$, $\boldsymbol{p}_0, \boldsymbol{p}_1 \in \boldsymbol{R}_{q,n}$, $\boldsymbol{u} \leftarrow \boldsymbol{R}_{2,n}$ and $\boldsymbol{e}_1, \boldsymbol{e}_2 \overset{\$}{\leftarrow}\chi$,

$ct = (\boldsymbol{c}_0, \boldsymbol{c}_1) = ([\boldsymbol{m} \cdot \Delta + \boldsymbol{p}_0 \times \boldsymbol{u} + \boldsymbol{e}_1]_q, [\boldsymbol{p}_1 \times \boldsymbol{u} + \boldsymbol{e}_2]_q)$.

– Decryption: $\boldsymbol{c}_0, \boldsymbol{c}_1 \in \boldsymbol{R}_{q,n}$ and $sk \in \boldsymbol{R}_{2,n}$,

$\boldsymbol{m} = [\lfloor \frac{t}{q}[\boldsymbol{c}_0 + \boldsymbol{c}_1 \times \boldsymbol{s}]_q \rceil]_t$.

3.2 Residue number system (RNS)

For homomorphic computation, an operation is first expressed as a logic or arithmetic circuit. Homomorphic encryption applications are practical only if the multiplicative depth of the circuit, which is to be homomorphically evaluated, is not very high. For example, private information retrieval from a table of 65536 entries [38], which requires a multiplicative depth of at least 4, is reasonably fast when implemented using homomorphic encryption. More complicated homomorphic operations result in larger values of n and q due to the increased depth of the circuit.

Efficient arithmetic with large values of the modulus q is very challenging. Therefore, residue number system (RNS) enabling parallelism at algorithmic level for modular integer arithmetic is frequently utilized in the implementations of homomorphic encryption schemes. In RNS, a set of coprime moduli q_i is used such that

$$q = \prod_{i=0}^{r-1} q_i,$$

where r is the number of moduli used. Using RNS, arithmetic operations modulo q (and thus on $\boldsymbol{R}_q$) can be mapped into operations on smaller q_i values (or $\boldsymbol{R}_{q_i}$), which can be performed in parallel. For example, a large 109-bit modulus can be constructed using 3 smaller moduli of sizes 36-bit, 36-bit and 37-bit. The RNS arithmetic requires conversion of a larger integer a in q to smaller integers, $a_i = a \bmod q_i$, in moduli q_i. Reconstruction of a from integers a_i in moduli q_is via Chinese Remainder Theorem (CRT) can be performed as

$$a = \sum_{i=0}^{r-1} a_i \cdot M_i \cdot m_i \pmod{q},$$

where $M_i = (q/q_i)$ and $m_i = M_i^{-1} \pmod{q_i}$ for $i = 0, \ldots, r-1$.

The SEAL library utilizes RNS and implements its arithmetic operations slightly different than the textbook-BFV, where polynomial arithmetic can be performed in parallel for each modulo q_i [10]. We also use the same approach in our GPU implementation since RNS enables leveraging the parallelism supported in GPU architecture. For the rest of the paper, a polynomial $\boldsymbol{a}$ in $\boldsymbol{R}_{q,n}$ with $q = \prod_{i=0}^{r-1} q_i$ will be represented as the array of $\boldsymbol{a}^i$ in $\boldsymbol{R}_{q_i,n}$ for $i = 0, 1, \ldots, r-1$.

3.3 Implementation of homomorphic operations in SEAL library

The full RNS implementations of key generation, encryption and decryption operations in SEAL are based on Algorithms 3, 4 and 5, respectively. Key generation and encryption operations of SEAL require random polynomials. To this end, SEAL utilizes three different distributions, (*i*) ternary distribution in $\boldsymbol{R}_{2,n}$, (*ii*) uniform distribution in $\boldsymbol{R}_{q,n}$, (*iii*) discrete Gaussian distribution χ with 0 mean and relatively small standard deviation σ. Besides, all three operations utilize polynomial arithmetic. The key generation operation requires $2r$ NTT and r coefficient-wise multiplications of two vectors. The encryption operation requires r NTT, $2r$ INTT and

$(6r-3)$ coefficient-wise multiplications of two vectors. The decryption operation requires $(r-1)$ NTT, $(r-1)$ INTT and $(3r-1)$ coefficient-wise multiplications of two vectors.

Algorithm 3 Implementation of Key Generation Operation in SEAL

Output: $\overline{s}^i \in \boldsymbol{R}_{q,n}, \overline{p}_0^i, \overline{p}_1^i \in \boldsymbol{R}_{q_i,n}$ for $0 \le i < r$

$\boldsymbol{s} \leftarrow \mathbf{R}_{2,n}$ ▷ Secret key generation
for $(i = 0;\ i < r;\ i = i + 1)$ **do**
  $\overline{\boldsymbol{s}}^i = \mathtt{NTT}_n(\boldsymbol{s})$ ▷ Operations in $\boldsymbol{R}_{q_i,n}$
end for
$\boldsymbol{e} \xleftarrow{\$} \chi$ ▷ Public key generation
for $(i = 0;\ i < r;\ i = i + 1)$ **do**
  $\overline{\boldsymbol{a}}^i \leftarrow \boldsymbol{R}_{q_i,n}$ ▷ Already in NTT domain
  $\overline{\boldsymbol{p}}_0^i = [-(\overline{\boldsymbol{a}}^i \odot \overline{\boldsymbol{s}}^i + \mathtt{NTT}_n(\boldsymbol{e}))]_{q_i}$ ▷ Operations in $\boldsymbol{R}_{q_i,n}$
  $\overline{\boldsymbol{p}}_1^i = \overline{\boldsymbol{a}}^i$
end for
return $\overline{\boldsymbol{s}}, \overline{\boldsymbol{p}}_0, \overline{\boldsymbol{p}}_1$

Algorithm 4 Implementation of Encryption Operation in SEAL

Input: $\boldsymbol{m} \in \boldsymbol{R}_{t,n}, \overline{\boldsymbol{p}}_0^i, \overline{\boldsymbol{p}}_1^i \in \boldsymbol{R}_{q_i}^n$ for $0 \le i < r$
Output: $\boldsymbol{c}_0^i, \boldsymbol{c}_1^i \in \boldsymbol{R}_{q_i,n}$ for $0 \le i < (r-1)$

$\boldsymbol{e}_1, \boldsymbol{e}_2 \xleftarrow{\$} \chi$
$\boldsymbol{u} \leftarrow \boldsymbol{R}_{q,n}$
for $(i = 0;\ i < r;\ i = i + 1)$ **do**
  $\overline{\boldsymbol{u}}^i = \mathtt{NTT}_n(\boldsymbol{u})$ ▷ Operations in $\boldsymbol{R}_{q_i}^n$
  $\boldsymbol{c}_0^i = \mathtt{INTT}_n([\overline{\boldsymbol{p}}_0^i \odot \overline{\boldsymbol{u}}^i + \boldsymbol{e}_1]_{q_i})$ ▷ Operations in $\boldsymbol{R}_{q_i}^n$
  $\boldsymbol{c}_1^i = \mathtt{INTT}_n([\overline{\boldsymbol{p}}_1^i \odot \overline{\boldsymbol{u}}^i + \boldsymbol{e}_2]_{q_i})$ ▷ Operations in $\boldsymbol{R}_{q_i}^n$
end for
for $(i = 0;\ i < (r-1);\ i = i + 1)$ **do**
  $\boldsymbol{t}_0 = [\overline{\boldsymbol{c}}_0^{r-1} \cdot \lfloor \frac{q_{r-1}}{2} \rfloor]_{q_i}$
  $\boldsymbol{t}_1 = [\overline{\boldsymbol{c}}_1^{r-1} \cdot \lfloor \frac{q_{r-1}}{2} \rfloor]_{q_i}$
  $\boldsymbol{c}_0^i = [(\boldsymbol{c}_0^i - \boldsymbol{t}_0 + \lfloor \frac{q_{r-1}}{2} \rfloor) \cdot q_{r-1}^{-1}]_{q_i}$
  $\boldsymbol{c}_1^i = [(\boldsymbol{c}_1^i - \boldsymbol{t}_1 + \lfloor \frac{q_{r-1}}{2} \rfloor) \cdot q_{r-1}^{-1}]_{q_i}$
end for
for $(j = 0;\ j < n;\ j = j + 1)$ **do** ▷ Add $\boldsymbol{m}$ to $\boldsymbol{c}_0$
  $f = \frac{\boldsymbol{m}[j] \cdot (q \pmod t) + \lfloor \frac{t+1}{2} \rfloor}{t}$
  for $(i = 0;\ i < r-1;\ i = i + 1)$ **do**
    $\boldsymbol{c}_0^i[j] = [\boldsymbol{c}_0^i[j] + (\boldsymbol{m}[j] \cdot \frac{q_i}{t} + f)]_{q_i}$ ▷ Operations in $\boldsymbol{R}_{q_i}^n$
  end for
end for
return $\boldsymbol{c}_0, \boldsymbol{c}_1$

Algorithm 5 Implementation of Decryption Operation in SEAL

Input: $\overline{s}^i \in \boldsymbol{R}_{q_i,n}$, $\boldsymbol{c}_0^i, \boldsymbol{c}_1^i \in \boldsymbol{R}_{q_i,n}$ for $0 \leq i < (r-1)$, γ where $\gamma > q$ and $\texttt{gcd}(\gamma, q) = 1$
Output: $\boldsymbol{m} \in \boldsymbol{R}_{t,n}$

for $(i = 0;\ i < (r-1);\ i = +1)$ **do**
  $\boldsymbol{mt}^i = \texttt{INTT}_n(\texttt{NTT}_n(\boldsymbol{c}_1^i) \odot \overline{s}^i)$ ▷ Operations in $\boldsymbol{R}_{q_i,n}$
  $\boldsymbol{mt}^i = [\boldsymbol{mt}^i + \boldsymbol{c}_0^i]_{q_i}$ ▷ Operations in $\boldsymbol{R}_{q_i,n}$
end for
for $(i = 0;\ i < (r-1);\ i = i+1)$ **do** ▷ Convert from base q to base (t, γ)
  $\boldsymbol{mt}^i = [\boldsymbol{mt}^i \cdot (t \cdot \gamma)]_{q_i} \cdot q^{-1}$
end for
for $(j = 0;\ j < 2;\ j = i+1)$ **do** ▷ $\text{mod}[t, \gamma]_0 = t$ and $\text{mod}[t, \gamma]_1 = \gamma$
  for $(k = 0;\ k < n;\ k = k+1)$ **do**
    $acc = 0$
    for $(i = 0;\ i < (r-1);\ i = i+1)$ **do**
      $acc = (acc + \boldsymbol{mt}^i[k] \cdot q) \bmod [t, \gamma]_j$
    end for
    $\boldsymbol{mt\gamma}_j[k] = acc \bmod [t, \gamma]_j$
  end for
end for
$\boldsymbol{mt\gamma}_0 = [\boldsymbol{mt\gamma}_0 \cdot (-q)^{-1}]_t$ ▷ Divide polynomial by $(-q)$ in $\mathbb{Z}_t$
$\boldsymbol{mt\gamma}_1 = [\boldsymbol{mt\gamma}_1 \cdot (-q)^{-1}]_\gamma$ ▷ Divide polynomial by $(-q)$ in $\mathbb{Z}_\gamma$
for $(i = 0;\ i < n;\ i = i+1)$ **do**
  if $\boldsymbol{mt\gamma}_1[i] > \frac{\gamma}{2}$ **then**
    $\boldsymbol{m}[i] = [\boldsymbol{mt\gamma}_1[i] + \gamma - \boldsymbol{mt\gamma}_0[i]]_t$
  else
    $\boldsymbol{m}[i] = [\boldsymbol{mt\gamma}_1[i] - \boldsymbol{mt\gamma}_0[i]]_t$
  end if
end for
$\boldsymbol{m} = [\boldsymbol{m} \cdot \gamma^{-1}]_t$
return $\boldsymbol{m}$

In order to find out the most time-consuming operations in key generation, encryption and decryption operations of the SEAL implementation, their timing breakdowns for different parameter sets are obtained on an Intel i9-7900X CPU running at 3.30 GHz × 20 with 32 GB RAM using GCC version `7.5.0` in Ubuntu `16.04.6` LTS, using SEAL version `3.5` and shown in Table 2. The results are obtained for three parameter sets targeting 128-bit security $(n, \log_2(q), r) \in \{(2048, 54, 1), (8192, 218, 5), (32768, 881, 16)\}$ with $t = 1024$. As shown in Table 2, the percentage of polynomial arithmetic in overall execution times increases as polynomial degree n and ciphertext modulus q are increased. For $n = 32768$ and $\log_2(q) = 881$, polynomial arithmetic constitutes 32.2%, 59.1% and 74.2% of the execution time for key generation, encryption and decryption operations, respectively.

4 Proposed implementation

In this section, we will discuss the design of the proposed GPU implementations and the optimizations thereof, in a bottom-up fashion. We will start from efficient modular reduction operation implementation and then describe the NTT implementation

Table 2 Timing breakdown of arithmetic operations in key generation, encryption and decryption operations in terms of percentage (%)

Operation	(2048,54,1)			(8192,218,5)			(32768,881,16)		
	Key	Enc	Dec	Key	Enc	Dec	Key	Enc	Dec
PA	10.5	13.8	49.8	25.7	43.2	73.6	32.2	59.1	74.2
$\leftarrow \boldsymbol{R}_{2,n}$	19.1	12.5	–	6.2	6.4	–	3.9	4.7	–
$\overset{\$}{\leftarrow} \chi$	35.8	67.0	–	16.4	37.8	–	7.5	18.8	–
$\leftarrow \boldsymbol{R}_{q,n}$	23.9	–	–	49.2	–	–	53.9	–	–
Other	10.7	6.7	50.2	2.5	12.6	26,4	2.5	17.4	25.8

PA Polynomial Arithmetic, *Key* Key Generation, *Enc* Encryption, *Dec* Decryption

and the optimizations. Finally, we will present GPU implementations of key generation, encryption and decryption operations of the BFV scheme.

Our proposed GPU implementations comes in two versions. The first version supports modular arithmetic with modulus up to 30-bit while the second does up to 61-bit modulus. For the rest of the paper, the implementations with 30-bit and 61-bit coefficient modulus q are referred as 30-bit and 61-bit designs, respectively. The proposed GPU implementations support polynomials of degrees ranging from 2048 to 32768 where polynomial degree is power of two.

4.1 Barrett reduction algorithm and 128-bit unsigned integer library

Algorithm 6 Barrett Reduction Algorithm for Modular Multiplication

Input: a, $b < q$, $w = K = \lceil \log_2(q) \rceil$, $\mu = \lfloor \frac{2^{2K}}{q} \rfloor$
Output: $c = a \cdot b \bmod q$
1: $t \leftarrow a \cdot b$
2: $s \leftarrow \lfloor \frac{t \cdot \mu}{2^{2w}} \rfloor$ ▷ $s \leftarrow (t \cdot \mu) \gg 2w$ (i.e. right shift by $2w$ bits)
3: $r \leftarrow s \cdot q$
4: $c \leftarrow t - r$
5: **if** $c \geq q$ **then**
6: **return** $c - q$
7: **else**
8: **return** c
9: **end if**

In NTT, INTT and polynomial multiplication operations, there are a substantial number of multiplications in modulo q, which take a considerable amount of time. Therefore, a lightweight and efficient modular reduction algorithm is essential to have high-performance implementation. As our modular reduction algorithm, we chose Barrett reduction [11].

The Barrett reduction algorithm replaces the division in reduction operation with multiplication, shift and subtraction operations, which are all significantly faster than the division operation, as shown in Algorithm 6. Barrett reduction calculates $c = a \cdot b$ modulo q by approximation. For a modulus of bit-length K, when

the number to be reduced has bit-length of L, where $0 \leq L \leq 2K$, this approximation becomes exact. Keeping that property in mind, it can be observed that the coefficients of the polynomials in $\boldsymbol{R}_{q,n}$ are all smaller than the coefficient modulus, q. When two coefficients are multiplied, the result is guaranteed to have a bit-length that is smaller than or equal to $2K$. Thus, the Barrett reduction algorithm always returns the correct results in our case. For a modulus q, the algorithm requires a precomputed value, μ defined as

$$\mu = \lfloor \frac{2^{2w}}{q} \rfloor, \tag{10}$$

where w is takes as bit-length of modulus, K.

For the Barrett algorithm to fit better into GPU design, here, we tweak the original Barrett reduction in Algorithm 6. The original Barrett reduction algorithm multiplies the result of $a \cdot b$ by μ, where a, b and μ are K bits long (see Step 2 of Algorithm 6). This can lead to integer variables as long as $3K$ bits long during the computations. In our 30-bit and 61-bit designs, this generates integers as large as 90 bits and 183 bits, respectively. Storing and performing operations on these integers are demanding, especially on GPU resources. Therefore, to reduce the size of the integers obtained after these operations down to a maximum of $2K$-bit, we divide the right shift operation by $2w$ as shown in Step 2 of Algorithm 6 into two separate $(w-2)$ and $(w+2)$ right shift operations as shown in Steps 2 and 4 of Algorithm 7, respectively. The first right shift operation is performed after the multiplication $a \cdot b$, where the result gets right shifted by $(w-2)$. The second right shift operation is performed after the multiplication with μ, where the result gets right shifted by $(w+2)$. Using a modulus q of at most 61 bits, by this tweak it is guaranteed that the intermediate values will be below or equal to $2K = 122$ bits, which saves memory and improves the performance. In Algorithm 7, we present the modified Barrett reduction algorithm optimized for GPU implementation.

Algorithm 7 Barrett Modular Multiplication Optimized for GPU

Input: a, $b < q$, $w = K = \lceil \log_2(q) \rceil$, $\mu = \lfloor \frac{2^{2K}}{q} \rfloor$
Output: $c = a \cdot b \bmod p$
1: $t \leftarrow a \cdot b$
2: $x_1 \leftarrow t \gg (w-2)$
3: $x_2 \leftarrow x_1 \cdot \mu$
4: $s \leftarrow x_2 \gg (w+2)$
5: $r \leftarrow s \cdot q$
6: $c \leftarrow t - r$
7: **if** $c \geq q$ **then**
8: **return** $c - q$
9: **else**
10: **return** c
11: **end if**

Next, for our 61-bit design, we needed a way to store unsigned integers up to 128 bits. Since CUDA does not have intrinsic support for 128-bit integers, we developed

a highly optimized 128-bit unsigned integer library. A 128-bit unsigned integer is represented by two 64-bit unsigned integers, which are naturally supported by CUDA and C/C++. One integer represents the least significant 64 bits while the other integer represents most significant 64 bits. We overloaded all the arithmetic, binary and logic operators. The operators which are not required during GPU computations are coded using C/C++. During the computations performed on the GPU, the multiplication of two 64-bit unsigned integers and the subtraction of two 128-bit unsigned integers are required. These operations are coded using the *PTX* inline assembly feature of CUDA for achieving the maximum possible performance. *PTX* enables inserting pseudoassembly code in a higher-level language such as C/C++. It provides features that are unavailable otherwise such as setting and using the carry bits. Therefore, multiplication and subtraction operations written in *PTX* compile into fewer instructions than those written in C/C++. Indeed, the multiplication operation written using *PTX* shows slightly better performance than the multiplication performed using Nvidia's intrinsic function, `umul`, and its derivatives.

4.2 NTT implementation on GPU

As shown in Algorithm 1, the NTT operation is performed with 3 nested *for* loops. If *for* loops are unrolled, it can be seen that an n-pt NTT operation actually consists of $\log_2(n)$ iterations and there are $(n/2)$ butterfly operations in each iteration. Iterations of the NTT operation for a toy example (i.e., $n = 8$) are depicted in Fig. 3, where the box with the letter `B` illustrates the butterfly operation and the box with powers of ψ shows the particular power of the $2n$-th root of unity used in the corresponding iteration. During execution, results are read from and written back to the same input array a, which is called *in-place* computation. Since $n = 8$ in Fig. 3, there are $\log_2(8) = 3$ iterations and 4 butterfly operations in each iteration. Note that the NTT blocks are halved after each iteration. Our INTT algorithm can be visualized to work in a similar way, but the order of the butterfly operations is somewhat reversed.

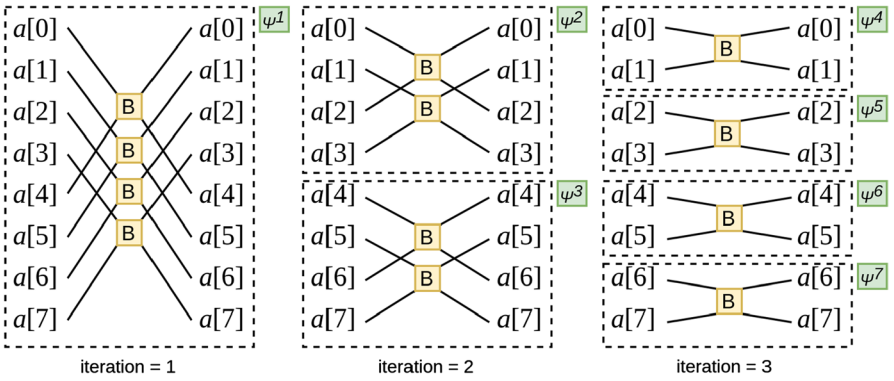


Fig. 3 Visualization of NTT operation for $n = 8$

Algorithm 8 Scheduling of Array Elements to Threads in GPU for NTT Algorithm

1: **for** (length $= 1$; length $< n$; length $= 2 \cdot$ length) **do**
2: tid $=$ Global index of thread in GPU
3: step $= (n/\texttt{length})/2$
4: psi_step $=$ tid/step
5: target_idx $= (\texttt{psi\_step} \cdot \texttt{step} \cdot 2) + (\texttt{tid} \bmod \texttt{step})$
6: step_group $=$ length $+$ psi_step
7: psi $=$ psis[step_group]
8: U $=$ a[target_idx]
9: V $=$ a[target_idx $+$ step]
10: ▷ Thread assignment is completed at this point
11: ▷ Every thread has its corresponding U, V and psi values
12: **end for**

The challenge in implementing the NTT/INTT algorithms in GPU is to assign the threads efficiently to achieve high utilization. For the best performance, all the threads should be busy, and the workload of each thread should be equivalent. From Fig. 3, for every butterfly operation, B, in each iteration, two elements of the array a are used together (see also Steps 10 and 11 of Algorithm 1). Thereby, we elect to use only one GPU thread for two elements of array a. Algorithm 8 shows how one thread is scheduled for each butterfly operation with two array elements. From Steps 2-8 of Algorithm 8, we can observe that each thread obtains the necessary values for performing one butterfly operation, namely two elements of the array a and the corresponding power of ψ.

Figure 4 illustrates how the array elements are assigned to threads in Algorithm 8 for the example when $n = 8$. Also, it shows the role of each variable defined in Algorithm 8. In Fig. 4, the array psis, which holds the powers of ψ, and its indices psi_step defined in Algorithm 8 are shown with boxes with solid edges. The boxes with dashed edges indicate the array elements using the same power of ψ. The array elements in the same dashed box form the *step group*, whose index is kept in the variable step_group. After each iteration, the number of step groups doubles, and the variable length keeps track of the number of such boxes. In the step groups, the numbers on the left and on the right stand for the indices of the array elements in Step 8 and Step 9 of Algorithm 8, respectively. The variable step is used to determine the second element of the array a that is assigned to the same thread. In summary, using the variables step and length, a thread can easily find out the array elements to process in every iteration independently.

For example, in the second iteration, where both length and step are 2, the first two array elements are assigned to the first two threads, and we use the variable step to find the two consecutive elements of the array a (here a[2] and a[3]) that will be assigned to the first two threads. The elements in the dashed box below are processed in an identical way.

Also, each thread uses the step_group variable to keep track of its step group in each iteration and use it to access the correct element of the array psis. For

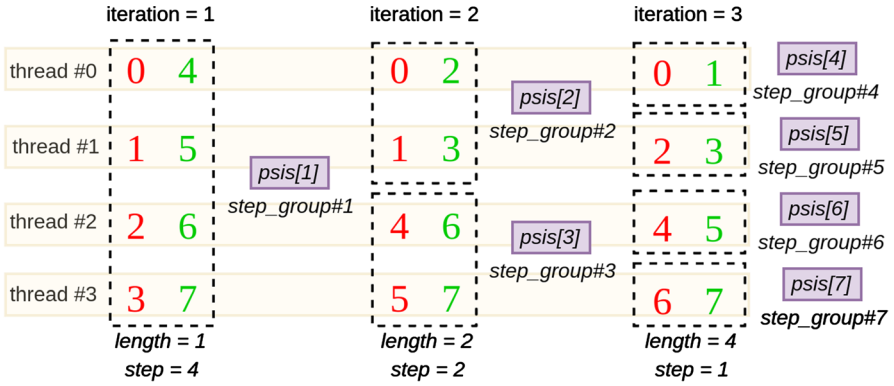


Fig. 4 GPU thread assignment for $n = 8$

example, in the second iteration in Fig. 4, the first and second step groups consists of array elements with indices 0,1,2,3 and those with indices 4,5,6,7, respectively.

The `target_idx` variable is used by a thread to compute the index of the first array element to be processed by that thread in each iteration (see Step 5 of Algorithm 8 for its computation). The thread first computes the number of array elements in the previous step groups for that iteration (`psi_step` · `step` · 2), which yields the first array element in its step group. Then, the thread uses its own id to find out the array element that it will process in that iteration.

Note that Algorithm 8 illustrates thread scheduling when the dimension of the input array is at most 2048 as we use one GPU thread per two elements of the array `a` and majority of the current GPU technologies do not support having more than 1024 threads per block to the best of our knowledge. Therefore, all threads in a GPU block performs the same steps in Algorithm 8. Nevertheless, handling an array with more than 2048 elements can be challenging and different approaches are possible. Here, we partition the array into groups of 2048 elements and use multiple GPU blocks to process them in different iterations of NTT computation.

4.2.1 Optimizations of memory usage and kernel calls

Efficient use of GPU memory hierarchy plays a significant role in the overall performance of our implementations. Coefficients of the input polynomial (array `a`) and different elements of the array `psis` are being constantly reached, rendering the memory access patterns a crucial part of the optimization process. The most basic and direct solution would be utilizing the global memory; however, this also would possibly lead to the worst performance.

Utilizing Shared Memory: Since we process the elements of the array `a` in GPU blocks, shared memory offers the best alternative for the threads that are located in the same block. The critical part here is that every element of the array `psis` is being accessed only once per block. Thereby, copying its elements into shared memory would carry no benefit. On the other hand, the elements of `a` are accessed

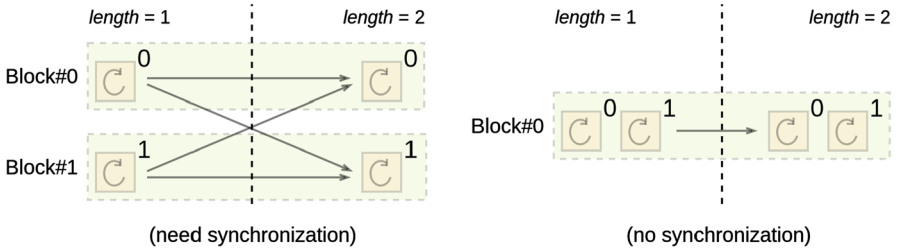
more than once during the computation of NTT/INTT; in fact, $\log_2(n)$ times, to be precise. Instead of making $\log_2(n)$ requests to global memory for a single element, we can make only one request to global memory for copying an element of `a` to shared memory and then access shared memory $\log_2(n)$ times for it.

4.2.2 Hybrid approach for kernel calls

We also investigate alternative methods for kernel calls. Essentially, we consider two alternatives: (i) in **single-kernel approach** we call a kernel once for the computation of NTT/INTT of the whole array (resulting in a total of $\log_2(n)$ iterations in a single-kernel call) and (ii) in **multi-kernel approach** we call a kernel for every iteration for the computation of NTT/INTT operation (resulting in a total of $\log_2(n)$ kernel calls). Both approaches have their advantages and disadvantages. For smaller-sized arrays, single-kernel approach provides better results. However, for larger array sizes (from $n = 2^{14}$ to $n = 2^{16}$, which is determined experimentally on the GPUs used in this study), a multi-kernel approach is advantageous.

In Fig 5, the principle of the our single-kernel approach is explained. The boxes with circular arrow symbols represent the operation, in which a GPU block of 1024 threads processes 2048 elements of the array `a`. We have two alternatives for scheduling of array elements to GPU blocks. In the first alternative, every GPU block processes 2048 elements of `a` concurrently as shown in Fig 5a. However, due to the dependencies shown with arrows connecting the boxes in Fig. 5a, a synchronization mechanism is needed for GPU blocks to wait for each other before progressing into the next iteration. Otherwise, race conditions may occur and lead to wrong results. But, if we assign a group of consecutive `step` $\times 2$ elements of `a` to the same GPU block as shown in Fig. 5b, then no GPU block needs to wait for other GPU blocks processing the other parts of the array `a`. This solution however exploits limited parallelism offered by the GPU block itself. Multi-kernel approach, on the other hand, can benefit from parallelism beyond the limitation of the single-kernel approach.

In the multi-kernel approach, we schedule $n/2048$ GPU blocks for every kernel call, all of which operate concurrently. Consequently, the multi-kernel approach can perform better when the array size is sufficiently large. But, every call to kernel



(a) with single iteration **(b)** with multiple iterations

Fig. 5 Single-kernel synchronization for $n = 2^{12}$

incurs certain overhead that results in worse performance than the single-kernel approach for smaller-sized arrays. Fully utilizing the potential of parallelism due to multiple kernel calls while reducing the call overhead in an optimal way, we develop the so-called hybrid approach, in which we start with the multi-kernel approach, but switch to the single-kernel approach when the step groups have fewer number of array elements.

We illustrate the hybrid approach for an array of size $n = 2^{15}$ in Fig. 6 for 61-bit design. In the leftmost column, we make a kernel call, whereby each of 16 block operates on separate 2048 elements of array a, in parallel. We make two more kernel calls in the same manner for the second and third iterations of the NTT computation. Until this point, we adopt the multi-kernel approach. However, when the number of

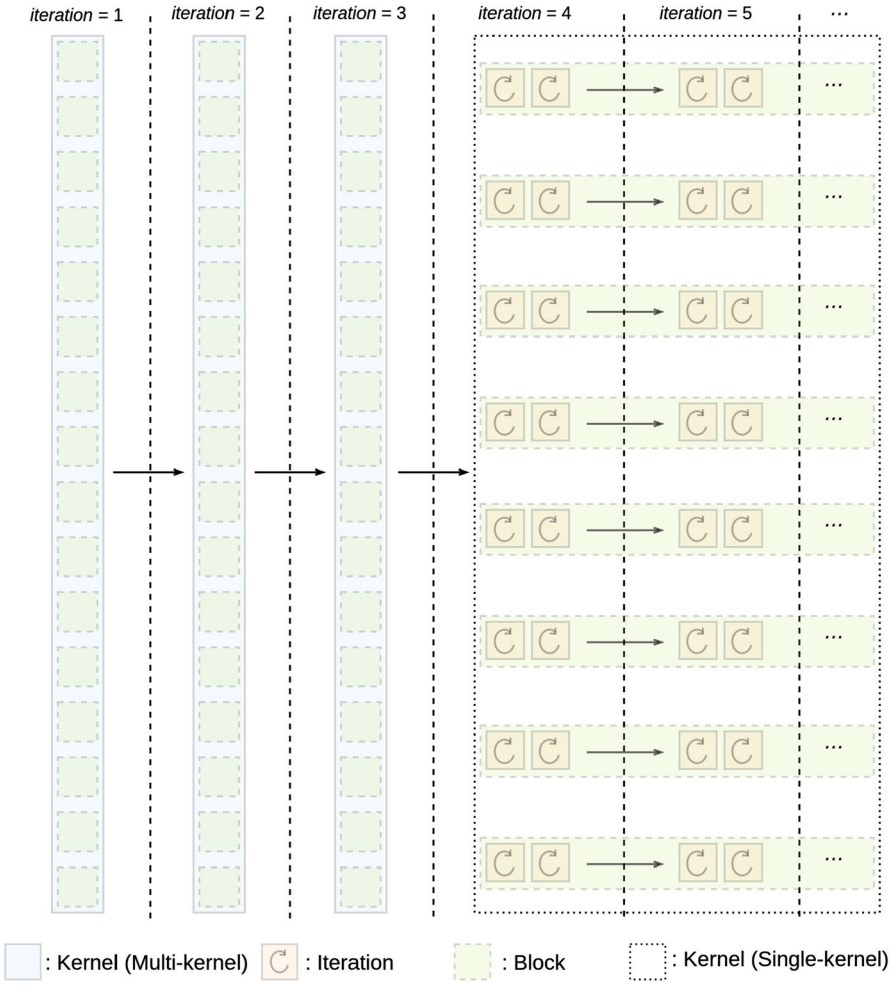


Fig. 6 Hybrid approach internals for $n = 2^{15}$ and 61-bit moduli

elements in a step group is 2^{12}, we switch to the single-kernel approach. Here, a total of eight GPU blocks process the rest of the iterations on their part of the array in a single kernel call. Note that when the step group size is 2^{12}, the lower and the upper 2048 elements of the array are processed sequentially. At first, this may seem to be rather inefficient. Nevertheless, the alternative would be making an additional kernel call, which brings in the kernel overhead. Then, we can immediately see there is a trade-off between the overhead and the time incurred due to sequential processing of two 2048 elements. We experimentally find out that the kernel call overhead is higher in terms of execution time when the step group size is 2^{12}. In other words, the step group size of 2^{12} is the optimal point to switch from the multi-kernel approach to the single-kernel approach. Note that the switching point can be different for the INTT operation and for the 30-bit design.

In the multi-kernel phase of the hybrid approach, the input array is accessed from the slow global memory, instead of copying it to the faster shared memory. This is due to the fact that the shared memory does not survive across the kernel calls. However, in the single-kernel phase, array elements are copied to the shared memory once, and they are accessed from there for the rest of the iterations as they are in the same block within the same kernel. Table 3 lists the number of NTT iterations that can be performed using shared memory.

The INTT algorithm is slightly different from NTT as it starts from small step groups and merges them after each iteration. Consequently, the pattern of kernel calls is the exact opposite of that of the NTT operation; we switch from single to multi-kernel approach. We find the optimal switching point when the step size is 2048 for 61-bit design in INTT computation.

Finally, when they are needed, all CPU-GPU memory transfer operations (`DeviceToHost`, `HostToDevice`) are performed asynchronously to achieve further optimization.

4.2.3 Other optimizations

Zhang et al. propose a technique to eliminate the final multiplication of coefficients with $\frac{1}{n}$ in $\mathbb{Z}_q$ after INTT operation as shown in Eq. 2 [45]. Instead, outputs of the GS butterfly operation utilized in INTT can be multiplied with $\frac{1}{2}$ in $\mathbb{Z}_q$, which generates

Table 3 Shared memory utilization with hybrid approach

n	TI	30-bit				61-bit			
		NTT		INTT		NTT		INTT	
		I(S-)	I(S+)	I(S-)	I(S+)	I(S-)	I(S+)	I(S-)	I(S+)
2048	11	0	11	0	11	0	11	0	11
4096	12	0	12	0	12	0	12	1	11
8192	13	0	13	1	12	1	12	2	11
16384	14	1	13	2	12	2	12	3	11
32768	15	2	13	3	12	3	12	4	11

TI Total number of iterations, *I(S-)* Number of iterations performed without shared memory, *I(S+)* Number of iterations performed with shared memory

$\frac{U+V}{2} \pmod q$ and $\frac{(U-V)\psi}{2} \pmod q$. For an odd coefficient modulus q, division by 2, namely $\frac{a}{2}$, can be performed in $\mathbb{Z}_q$ as shown in Eq. 11.

$$\frac{a}{2} \pmod q = a \cdot \left(\frac{q+1}{2}\right) \tag{11}$$

If a is an even integer, expression in Eq. 11 becomes ($a \gg 1$), and if a is an odd integer, the expression becomes $(a \gg 1) + \frac{q+1}{2}$ where $\gg$ represents right shift operation. This optimization replaces modular multiplication operations after INTT with right shift and addition operations, both of which are faster.

4.2.4 Time complexity

The time complexity of Algorithm 1 is $\mathcal{O}(n \cdot \log_2(n))$ since the outer `for` loop is executed for $\log_2(n)$ times and the inner two loops are executed $n/2$ times combined. The proposed algorithm, Algorithm 8, distributes the inner two loops to GPU threads and executes them in parallel since these operations are independent of each other. This reduces the time complexity of Algorithm 8 to just $\mathcal{O}(\log_2(n))$, which is also the complexity of the multi-kernel approach.

The single-kernel approach, on the other hand, is utilized only in specific cases (where `step_group` ≤ 4096 for the 61-bit and `step_group` ≤ 8192 for the 30-bit). For the 61-bit design, time complexity of single kernel design is doubled in comparison with the multi-kernel approach, because 1024 threads can handle 4096 elements in the `step_group`, in 2 sequential iterations in the worst case. For the 30-bit design, time complexity of single kernel design is quadrupled, because at maximum, we would have 8192 elements in the `step_group`, requiring 4 sequential iterations. Both of the single kernel designs (30-bit and 61-bit) are of $\mathcal{O}(\log_2(n))$ complexity.

In the hybrid approach, the inner two `for` loops of Algorithm 1 are executed in parallel in batches of 2 and 4 for 61-bit and 30-bit designs, respectively. Since these numbers are constant, hybrid approach also has $\mathcal{O}(\log_2(n))$ time complexity. The time complexities of NTT and INTT algorithms are identical.

4.3 Implementation of the BFV scheme on GPU

To demonstrate the performance of the proposed NTT and INTT implementations in a practical setting, we implemented the key generation, the encryption and the decryption operations of the BFV scheme on GPU. For these operations, all necessary parameters and powers of ψ are generated on CPU of the host computer nd then sent to GPU prior to any computation. GPU uses the same parameters for the key generation, the encryption and the decryption operations until a new parameter set is sent to GPU. We adopt an approach, referred as *kernel merge strategy*, whereby we find the optimal number of kernel calls that yields the best performance.

4.3.1 Key generation

Key generation of the BFV scheme is implemented as shown in Algorithm 3. The key generation algorithm takes n, q_i and χ as inputs and generates r secret key polynomials and $2r$ public key polynomials in $\boldsymbol{R}_{q_i,n}$ for $0 \leq i < r$. The key generation operation requires randomly generated polynomials from uniform, ternary and discrete Gaussian distributions, which require a random polynomial sampler. Also, it performs NTT and coefficient-wise multiplication operations.

For the generation of random polynomials, we first generate a sequence of cryptographically secure random bytes and then converted them into random integers from desired distributions (uniform, ternary, discrete Gaussian). We use the Salsa20 stream cipher [18] for generating random bytes by utilizing an existing implementation [22].

For discrete Gaussian distribution, consecutive 4 byte outputs from Salsa20 are interpreted as an unsigned 32-bit integer and converted to a number between 0 and 1. If the converted value is either 0 or 1, we add or subtract the smallest floating point number, defined in CUDA to get a value strictly between 0 and 1. Then, we applied inverse cumulative standard distribution function `normcdfinvf` in CUDA Math API to a value, which results in a random number from discrete Gaussian distribution with mean 0 and standard deviation 1. Since the SEAL library uses a discrete Gaussian distribution with standard deviation 3.2, we multiply the number with 3.2.

For uniform distribution in $\mathbb{Z}_{q_i}$, consecutive 8 byte outputs from Salsa20 are interpreted as an unsigned 64-bit integer, converted to a value between 0 and 1, and then multiplied by $(q_i - 1)$. For ternary distribution, each byte output from Salsa20 is interpreted as an unsigned 8-bit integer, mapped to a value between 0 and 2, decremented by 1, and finally, its fractional part is discarded to obtain either -1, 0 or 1.

As shown in Algorithm 1, random polynomial generation from uniform distribution, NTT and coefficient-wise multiplication operations are performed for different q_i values in every iteration of the *for* loops. By following the kernel merge strategy, instead of invoking a separate kernel call for every iteration of the for loops, we invoke a single kernel call to handle all the iterations, avoiding the overhead of calling multiple kernels.

4.3.2 Encryption

The encryption of the BFV scheme is implemented as shown in Algorithm 4. The encryption algorithm takes n, q_i, χ, one plaintext polynomial in $\boldsymbol{R}_{t,n}$ and $2r$ public key polynomials in $\boldsymbol{R}_{q_i,n}$ as inputs and generates $2r$ ciphertext polynomials in $\boldsymbol{R}_{q_i,n}$ for $0 \leq i < r$. The encryption operation requires randomly generated polynomials from ternary and discrete Gaussian distributions. Also, it performs polynomial arithmetic including mainly NTT, INTT and coefficient-wise multiplication operations. As in the key generation implementation, we use the kernel merge strategy. Here, we invoke a single kernel call for generating random polynomials from discrete Gaussian and uniform distributions. Also, we use a single kernel call for each for loop.

4.3.3 Decryption

The decryption of the BFV scheme is implemented as shown in Algorithm 3. The decryption algorithm takes n, q_i, γ parameters and r secret key polynomials in $\boldsymbol{R}_{q_i,n}$ and $2r$ ciphertext polynomials in $\boldsymbol{R}_{q_i,n}$ as inputs and generates one plaintext polynomials in $\boldsymbol{R}_{t,n}$ for $0 \leq i < r$. The decryption operation does not utilize randomly generated polynomials. It performs polynomial arithmetic including mainly NTT, INTT and coefficient-wise multiplication operations.

In addition to kernel merging strategy, different streams are also utilized here. When not specified explicitly, a kernel is scheduled on the default stream; and thus all operations are being executed sequentially. For concurrent operations, we use multiple streams, which allows to run them in parallel.

5 Implementation results

Performance results of the proposed NTT and INTT implementations are obtained on three different GPUs: Nvidia GTX `980`, Nvidia GTX `1080` and Nvidia Tesla `V100` with test environments shown in Table 4. The proposed implementations are verified using the real data generated using the SEAL library.

Performance results of the SEAL are obtained on an Intel i9-7900X CPU running at 3.30 GHz × 20 with 32 GB RAM using GCC version `7.5.0` in Ubuntu `16.04.6` LTS, using SEAL version `3.5`. Performance results of the key generation, the encryption and the decryption implementations of BFV scheme on GPU are obtained only for Nvidia Tesla `V100` with test environment shown in Table 4. For obtaining the performance results of GPU implementations, we used `nvvp` profiler utilizing `nvprof`. Note that the time required for initialization steps such as sending necessary parameters and powers of ψ to GPU from CPU is not included in performance results of NTT/INTT operations.

Table 4 Test environments

Specifications	GPUs		
	GTX `980`	GTX `1080`	Tesla `V100`
# of cores	2048	2560	5120
Memory	4GB	8GB	32GB
Frequency (MHz)	1127	1607	1230
Bandwidth	224.4 GB/s	320.3 GB/s	897 GB/s
FP64 performance	155.6 GFLOPS (1:32)	277.3 GFLOPS (1:32)	7.066 TFLOPS (1:2)
Compute capability	5.2	6.1	7.0
CPU	Intel E5-2620 v4	Intel i9 7900X	Intel Gold 5122
RAM	20GB	32GB	62GB

5.1 Performance results and comparison with SEAL

In order to evaluate the effects of individual optimizations, we experimented with five different designs for 30-bit implementation with different n values and obtained performance results on Nvidia GTX 980 GPU: (*i*) the design **D1** uses single kernel for one NTT/INTT operation, (*ii*) the design **D2** uses multiple kernels for one NTT/INTT operation, (*iii*) the design **D3** uses single kernel for one NTT/INTT operation and utilizes shared memory, (*iv*) the design **D4** uses single kernel for one NTT/INTT operation and utilizes warp shuffle (as utilized in [23]), (*v*) the design **D5** utilizes the proposed hybrid approach for one NTT/INTT operation presented in Sect. 4. The performance results of all designs are shown in Table 5.

As shown in Table 5, utilizing shared memory (**D3**) improves the performance of the NTT/INTT implementation with single kernel up to 10%, 25% and 17% for n values 2048, 4096 and 8192, respectively, compared to **D1**. Due to limited capacity of shared memory on GPU, this optimization can only be applied to the polynomials of degree 8192 or less ($n \leq 8192$) for 30-bit implementation. Applying warp shuffle to the implementation with single kernel (**D4**) also shows some improvements, but not as significant as the shared memory utilization. As shown in Table 5, the warp shuffle optimization improves the performance of NTT/INTT up to 10% for different n values . This is expected since the warp shuffle mechanism can only be used for the first five iterations of NTT/INTT operation.

As shown in Table 5, the design with single kernel utilizing shared memory (**D3**) shows 9.4×, 6.1× and 3.2× better performance than the design with multiple kernels (**D2**) for n values 2048, 4096 and 8192, respectively. On the other hand, utilizing multiple kernels per NTT/INTT operation shows better performance in case of $n > 8192$. The design with multiple kernels (**D2**) shows 1.5×, 1.5× and 2.4× better performance than the design with single kernel without shared memory utilization (**D1**) for n values 16384, 32768 and 65536, respectively. Therefore, as explained in Sect. 4, we combined both approaches and utilized the hybrid approach with shared

Table 5 Performance results with optimizations on GTX 980 in μs

Operation	Design	$\log_2(n)$					
		11	12	13	14	15	16
NTT	D1	12	21	38	73	149	309
	D2*	95	98	103	109	117	127
	D3	10	16	32	–	–	–
	D4	11	21	37	72	145	303
	D5*	10	16	32	42	49	61
INTT	D1	14	24	40	75	151	315
	D2*	88	102	108	111	114	118
	D3	13	17	33	–	–	–
	D4	13	24	41	75	148	305
	D5*	13	17	21	30	36	46

*Including kernel overhead for multiple kernels

memory (**D5**). The design with hybrid approach shows 1.2-5× better performance for different n values compared to the design **D1**. Similar comparison results are applicable to the INTT operation as well. Since the design with hybrid approach (**D5**) provides the best result, we obtained the performance results of 30-bit and 61-bit implementations for NTT and INTT with hybrid approach for three different GPU platforms as shown in Table 6.

The key generation, the encryption and the decryption operations of the BFV scheme are implemented on GPU as explained in Sect. 4. The operations are implemented for five different parameter sets $(n,\log_2(q), r) \in \{(4096,109,3), (8192,152,4), (16384,237,5), (32768,496,9), (32768,880,16)\}$ with $t = 1024$ being used in the SEAL library and targeting various security levels. GPU implementations of the key generation, the encryption and the decryption utilize the proposed efficient NTT and INTT implementations. The performance results of the SEAL library and the proposed GPU implementations for the given set of parameters are presented in Table 7.

As shown in Table 7, the speedup values by GPU implementation compared to the SEAL library become larger as the the ring parameters are increased. For $(n, \log_2(q), r) = (32768,880,16)$, the GPU implementations of the key generation, the encryption and the decryption operations are improved by 141.95×, 94.39× and 90.13×, respectively, compared to those of the SEAL library running on CPU. There are mainly two reasons for this improvement: (*i*) the percentage of polynomial arithmetic increases as the ring parameters are increased as shown in Table 2, (*ii*) as r is increased, GPU platform leverages parallelism better compared to the CPU platform. For the sake of simplicity, we only present best improvement results for GPU platforms Nvidia GTX `980` and Nvidia GTX `1080`. We observe up to 60.31×, 43.85×, 33.89× speed up on Nvidia GTX `980`; and 53.08×, 40.85×, 25.06× speed up on Nvidia GTX `1080`, for the key generation, the encryption and the decryption operations, respectively.

Table 6 Performance results of NTT and INTT implementations in μs

Operation	Platform	$\log_2(q)$	$\log_2(n)$					
			11	12	13	14	15	16
NTT	GTX `980`	30	10	16	32	42	49	61
		55	20	36	43	51	73	–
	GTX `1080`	30	7	15	16	23	24	28
		55	11	21	27	33	36	–
	Tesla `V100`	30	7	11.5	22.5	25.5	27.7	39
		55	12.5	22.5	27	29	39	–
INTT	GTX `980`	30	13	17	21	30	36	46
		55	25	31	35	41	52	–
	GTX `1080`	30	7	9	12	14	16	22
		55	12	15	18	20	24	–
	Tesla `V100`	30	7.5	13	14.5	16.3	18.3	20.7
		55	12.5	15.5	18	21	23	–

Table 7 Performance results of the key generation, the encryption and the decryption operations of the BFV scheme in μs

$(n,\log_2(q),r)$	Operation	SEAL	TW	Speedup
(4096,109,3)	Key generation	2020.00	123.86	16.31×
	Encryption	1744.20	85.82	20.32×
	Decryption	363.37	79.46	4.57×
(8192,152,4)	Key leneration	4480.00	135.81	32.99×
	Encryption	4009.76	99.93	40.13×
	Decryption	1012.74	87.46	11.58×
(16384,237,5)	Key generation	10541.00	176.64	59.68×
	Encryption	9286.84	119.26	77.87×
	Decryption	2688.53	104.13	25.82×
(32768,496,9)	Key generation	35125.00	273.73	128.32×
	Encryption	29038.60	276.10	105.17×
	Decryption	11365.70	160.05	71.01×
(32768,880,16)	Key generation	60729.00	427.81	141.95×
	Encryption	48586.70	514.73	94.39×
	Decryption	22215.50	246.48	90.13×

TW This work

5.2 Comparison with prior works

There are plenty of works in the literature targeting acceleration of FHE and PQC schemes on GPU. In [4], Badawi et al. accelerate polynomial multiplication operation implemented on NFLlib [1] using GPU with a variant of Stockham NTT algorithm. They present performance results only up to $n = 8192$ with q around 400-bit for polynomial multiplication operation. However, their implementation does not utilize RNS; therefore, it is not easy to compare our work with [4]. In [5, 6], Badawi et al. implement and accelerate some arithmetic operations of the BFV scheme on GPU. However, their work use discrete Galois transform (DGT) instead of NTT. In [6], their GPU implementation for the key generation, the encryption and the decryption operations of the BFV scheme achieves 5×, 7× and 22× speedup on average, respectively, on Tesla `P100` for different parameter sets compared to the SEAL library (version `2.3`). Our work, on the other hand, improves the performance of the key generation, the encryption and the decryption operations of the BFV scheme by 75.8×, 67.6× and 40.6× on average, respectively, on Tesla `V100` compared to the SEAL library (version `3.5`).

Some of the works [24, 28, 29] focus on PQC schemes utilizing small n (namely 1024 or less) values and fixed q. Therefore, they are not applicable to FHE schemes and they are not included in the comparison. The works in [3, 17, 23, 27, 40, 46] focus on accelerating FHE schemes and they also present separate performance results for NTT/INTT operations. We compare our results with these works in Table 8. It should be noted that the comparison presented in this section may not be ideal but an estimate due to platform differences. Compared to the works in [17, 23, 27, 46], our implementations on GTX `1080` and Tesla `V100` show better timing performance. Our implementations on all GPU platforms outperform the work

presented in [40]. As shown in Table 8, GTX 1080 shows better performance than Tesla V100 in some cases. The base clock of Tesla V100 is 1230 MHz while GTX 1080 has 1607 MHz base clock. Therefore, for a single operation, GTX 1080 can outperform Tesla V100 by roughly 1.3×. On the other hand, Tesla V100 has 80 streaming multiprocessors which is four times more than GTX 1080 has. When the number of operations to be processed is higher than that GTX 1080's streaming multiprocessors can handle simultaneously, Tesla V100 can show better performance. Thus, GTX 1080 can perform better for some parameter sets, due to its higher base clock frequency.

The work in [3] outperforms our implementations; however, it uses a fixed special prime moduli as in the works [3, 17, 23, 46]. These works utilize a special prime, $q = 2^{64} - 2^{32} + 1$, called Solinas prime, which is used as a *carrier* modulus for NTT/INTT operations in $\boldsymbol{R}_{q_i,n}$. This special prime enables following two very useful properties: (*i*) efficient modular reduction operation using only addition and subtraction, (*ii*) 64th root of unity of q is 8, which enables converting multiplication

Table 8 Comparison table

Work	Platform	n	$\lceil \log_2(q) \rceil$	NTT (μs)	INTT (μs)
[17]*	GTX 690	16384	64[c]	56	65.3
		32768	64[c]	71.2	83.6
[17]*[,a]	Tesla K80	16384	64[c]	12.9	12.5
		32768	64[c]	19	21.6
[17]*[,b]	GTX 1070	16384	64[c]	66.8	–
[3]*	Tesla K80	16384	64[c]	9.6	9.7
		32768	64[c]	15.3	16.2
[23]*	GTX 1070	16384	64[c]	57.8	–
[46]*	RTX 2080 Ti	32768	64[c]	83.3	96
[27]	Titan V	16384	60	44.1	–
		32768	60	84.2	–
[40]	GTX 1050	16384	NA	255	–
		32768	NA	470	–
	Titan RTX 1080	16384	NA	375	–
		32768	NA	425	–
This work	GTX 980	16384	55	51	41
		32768	55	73	52
	GTX 1080	16384	55	33	20
		32768	55	36	24
	Tesla V100	16384	55	29	21
		32768	55	39	23

[a]Results are from [3]

[b]Results are from [23]

[c]Actual q_i is restricted by $q_i^2 n < 2^{64} - 2^{32} + 1$

*Uses constant q=0xFFFFFFFF00000001

operation with twiddle factor into simple shift operation. Although using this prime as carrier modulus reduces the computational complexity and improves the performance, it comes with the following limitations: (*i*) size of the input polynomials needs to be doubled (from n to $2n$) and upper n coefficients should be zero-padded for NTT-based polynomial multiplication (i.e., 32768-pt NTT operation should be used for a ring with $n = 16384$), (*ii*) although NTT operation is performed in $\boldsymbol{R}_{q,n}$ where q is the special prime, q_i of encryption scheme is constrained by $q_i^2 \cdot n < q$ (i.e., q_i can be a 24-bit modulus at most for $q = 2^{64} - 2^{32} + 1$ and $n = 65536$), (*iii*) the resulting polynomial after INTT operation requires polynomial reduction with $(x^n + 1)$ for bringing its $2n$ size back to n, (*iv*) multiplying more than two polynomials requires going back-and-forth between NTT and polynomial domains.

In our work, we do not employ a carrier modulus and, thus we are not restricted by the aforementioned limitations. We do not have to double the size of input polynomials, apply polynomial reduction operation or go back-and-forth between NTT and polynomial domains for multiplying more than two polynomials.

6 Conclusion and future work

In this work, we proposed efficient GPU implementations of NTT and INTT operations to be used in NTT-based polynomial multiplication. The proposed implementations can perform a single NTT and INTT operations in $39\mu s$ and $23\mu s$, respectively, including kernel overhead for the largest ring with $n = 32768$ and $\log_2(q) = 61$ on Tesla `V100` GPU. The proposed GPU implementations are utilized for accelerating the key generation, the encryption and the decryption operations of the BFV homomorphic encryption scheme, which use NTT and INTT operations frequently. The proposed GPU implementations improve the performance of the key generation, the encryption and the decryption operations by up to 141.95×, 105.17× and 90.13×, respectively, on Tesla `V100` compared to the implementations on SEAL library running on an Intel i9-7900X CPU. The proposed GPU implementations prove to be useful as accelerators for computation-demanding FHE schemes and we also showed that utilizing GPUs as accelerators for FHE libraries such as SEAL is very promising.

As future work, we target accelerating homomorphic multiplication and generating a complete GPU accelerator library for BFV and CKKS schemes, which are required for practical FHE applications. Finally, with minor modifications the proposed NTT and INTT implementations can be used to accelerate PQC schemes with smaller polynomial rings or NTT-unfriendly lattice-based cryptosystems.

References

1. Aguilar-Melchor C, Barrier J, Guelton S, Guinet A, Killijian MO, Lepoint T (2016) Nfllib: Ntt-based fast lattice library. Topics in Cryptology. In: Cryptographers' Track at the RSA Conference. San Francisco, CA, USA, pp. 341–356
2. Al Badawi A, Hoang L, Mun CF, Laine K, Aung KMM (2020) Privft: Private and fast text classification with homomorphic encryption. IEEE Access 8:226544–226556
3. Al Badawi A, Veeravalli B, Aung KMM (2018) Faster number theoretic transform on graphics processors for ring learning with errors based cryptography. In: 2018 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI). IEEE, pp. 26–31
4. Al Badawi A, Veeravalli B, Aung KMM, Hamadicharef B (2018) Accelerating subset sum and lattice based public-key cryptosystems with multi-core cpus and gpus. J Parallel Distrib Comput 119:179–190
5. Al Badawi A, Veeravalli B, Lin J, Xiao N, Kazuaki M, Mi AKM (2021) Multi-gpu design and performance evaluation of homomorphic encryption on gpu clusters. IEEE Trans Parallel Distrib Syst 32(2):379–391
6. Al Badawi A, Veeravalli B, Mun CF, Aung KMM (2018) High-performance FV somewhat homomorphic encryption on gpus: an implementation using cuda. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 70–95
7. Alkım E, Bilgin YA, Cenk M (2019) Compact and simple RLWE based key encapsulation mechanism. In: International Conference on Cryptology and Information Security in Latin America. Springer, pp. 237–256
8. Alves PGMR, Ortiz JN, Aranha DF (2020) Faster homomorphic encryption over gpgpus via hierarchical DGT. Cryptology ePrint Archive, Report 2020/861
9. Angel S, Chen H, Laine K, Setty S (2018) PIR with compressed queries and amortized query processing. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 962–979. IEEE
10. Bajard JC, Eynard J, Hasan MA, Zucca V (2016) A full RNS variant of FV like somewhat homomorphic encryption schemes. In: International Conference on Selected Areas in Cryptography. NL, Canada, pp. 423–442
11. Barrett P (1986) Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. Adv Cryptol CRYPTO-86 263:311–323
12. Brakerski Z (2012) Fully homomorphic encryption without modulus switching from classical gapsvp. In: Annual Cryptology Conference. Springer, pp. 868–886
13. Brakerski Z, Gentry C, Vaikuntanathan V (2014) (Leveled) fully homomorphic encryption without bootstrapping. ACM Trans Comput Theory (TOCT) 6(3):1–36
14. Brutzkus A, Elisha O (2019) Gilad-Bachrach, R.: Low latency privacy preserving inference. In: International Conference on Machine Learning
15. Cheon JH, Kim A, Kim M, Song Y (2017) Homomorphic encryption for arithmetic of approximate numbers. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 409–437. Springer
16. Chu E, George A (1999) Inside the FFT black box: serial and parallel fast Fourier transform algorithms. CRC Press, Boca Raton
17. Dai W, Sunar B (2015) cuHE: a homomorphic encryption accelerator library. In: International Conference on Cryptography and Information Security in the Balkans. Springer, pp. 169–186
18. Bernstein DJ (2008) The salsa20 family of stream ciphers. Lect Notes Comput Sci 4986:84–97. https://doi.org/10.1007/978-3-540-68351-3_8
19. Fan J, Vercauteren F (2012) Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/14
20. Feng X, Li S, Xu S (2019) RLWE-oriented high-speed polynomial multiplier utilizing multi-lane stockham NTT algorithm. IEEE Transactions on Circuits and Systems II: Express Briefs. p. 1 . https://doi.org/10.1109/TCSII.2019.2917621
21. Gentry C, Boneh D (2009) A Fully Homomorphic Encryption Scheme, vol 20. Stanford university, Stanford
22. Ghosh M. Salsa20 cuda. https://github.com/moinakg/salsa20_core_cuda
23. Goey JZ, Lee WK, Goi BM et al (2021) Accelerating number theoretic transform in GPU platform for fully homomorphic encryption. J Supercomput 77:1455–1474. https://doi.org/10.1007/s11227-020-03156-7

24. Gupta N, Jati A, Chauhan AK, Chattopadhyay A (2020) PQC acceleration using gpus: FrodoKEM, NewHope and Kyber. IEEE Transactions on Parallel and Distributed Systems, p. 1
25. Halevi S, Shoup V (2014) Algorithms in Helib. Advances in Cryptology-CRYPTO 2014. Santa Barbara, CA, USA, pp 554–571
26. Karatsuba AA, Ofman YP (1962) Multiplication of many-digital numbers by automatic computers. In: Doklady Akademii Nauk, vol. 145, pp. 293–294. Russian Academy of Sciences
27. Kim S, Jung W, Park J, Ahn JH (2020) Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In: 2020 IEEE International Symposium on Workload Characterization (IISWC). https://doi.org/10.1109/iiswc50251.2020.00033
28. Lee WK, Akleylek S, Wong DCK et al (2021) Parallel implementation of nussbaumer algorithm and number theoretic transform on a GPU platform: application to qTESLA. J Supercomput 77:3289–3314. https://doi.org/10.1007/s11227-020-03392-x
29. Lee WK, Akleylek S, Yap WS, Goi BM (2019) Accelerating number theoretic transform in gpu platform for qtesla scheme. In: International Conference on Information Security Practice and Experience. Springer, pp. 41–55
30. Longa P, Naehrig M (2016) Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In: Cryptology and Network Security. Milan, Italy, pp. 124–139
31. Lyubashevsky V, Peikert C, Regev O (2010) On ideal lattices and learning with errors over rings. In: Advances in Cryptology-EUROCRYPT. French Riviera, pp. 1–23
32. Mera JMB, Karmakar A, Verbauwhede I (2020) Time-memory trade-off in toom-cook multiplication: an application to module-lattice based cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 222–244
33. Mert AC, Öztürk E, Savaş E (2019) Design and implementation of encryption/decryption architectures for BFV homomorphic encryption scheme. IEEE Trans Very Large Scale Integr (VLSI) Syst 28(2):353–362
34. Pollard JM (1971) The fast Fourier transform in a finite field. Math Comput 25(114):365–374
35. Polyakov Y, Rohloff K, Ryan GW (2017) Palisade lattice cryptography library user manual. Cybersecurity Research Center, New Jersey Institute of Technology (NJIT), Tech. Rep
36. Pöppelmann T, Oder T, Güneysu T (2015) High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In: International Conference on Cryptology and Information Security in Latin America. Springer, pp. 346–365
37. Riazi MS, Laine K, Pelton B, Dai W (2020) Heax: an architecture for computing on encrypted data. In: Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS' 20, pp. 1295-1309. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3373376.3378523
38. Roy SS, Turan F, Jarvinen K, Vercauteren F, Verbauwhede I (2019) Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. Cryptology ePrint Archive, Report 2019/160
39. Roy SS, Vercauteren F, Mentens N, Chen DD, Verbauwhede I (2014) Compact ring-lwe cryptoprocessor. In: Batina L, Robshaw M (eds) Cryptographic Hardware and Embedded Systems-CHES 2014. Springer, Berlin, pp 371–391
40. Sahu G, Rohloff K (2020) Accelerating lattice based proxy re-encryption schemes on gpus. In: Krenn S, Shulman H, Vaudenay S (eds) Cryptology and Network Security. Springer International Publishing, Cham, pp 613–632
41. Microsoft, SEAL, (2020) Microsoft Research. Redmond, Microsoft SEAL, (release 3.6). https://github.com/Microsoft/SEAL
42. Seiler G (2018) Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. IACR Cryptol ePr Arch 2018:39
43. Sinha Roy S, Järvinen K, Vliegen J, Vercauteren F, Verbauwhede I (2018) Hepcloud: an fpga-based multicore processor for FV somewhat homomorphic function evaluation. IEEE Trans Comp 67(11):1637–1650. https://doi.org/10.1109/TC.2018.2816640
44. Toom AL (1963) The complexity of a scheme of functional elements realizing the multiplication of integers. Sov Math Dokl 3:714–716
45. Zhang N, Yang B, Chen C, Yin S, Wei S, Liu L (2020) Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT. IACR Trans on CHES 2:49–72. https://doi.org/10.13154/tches.v2020.i2.49-72

46. Zheng, Z (2020) Encrypted Cloud using GPUs, (Master's Thesis, KU Leuven, Leuven, Belgium). Retrieved from https://www.esat.kuleuven.be/cosic/publications/thesis-394.pdf

Authors and Affiliations

Özgün Özerk[1] · Can Elgezen[1] · Ahmet Can Mert[1] · Erdinç Öztürk[1] · Erkay Savaş[1]

Özgün Özerk
ozgunozerk@sabanciuniv.edu

Can Elgezen
celgezen@sabanciuniv.edu

Erdinç Öztürk
erdinco@sabanciuniv.edu

Erkay Savaş
erkays@sabanciuniv.edu

[1] Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey