The Journal of Supercomputing (2021) 77:1455–1474
https://doi.org/10.1007/s11227-020-03156-7

Accelerating number theoretic transform in GPU platform for fully homomorphic encryption

Jia-Zheng Goey[1] · Wai-Kong Lee[2] · Bok-Min Goi[1] · Wun-She Yap[1]

Published online: 18 May 2020

Abstract
In scientific computing and cryptography, there are many applications that involve large integer multiplication, which is a time-consuming operation. To reduce the computational complexity, number theoretic transform is widely used, wherein the multiplication can be performed in the frequency domain with reduced complexity. However, the speed performance of large integer multiplication is still not satisfactory if the operand size is very large (e.g., more than 100K-bit). In view of that, several researchers had proposed to accelerate the implementation of number theoretic transform using massively parallel GPU architecture. In this paper, we proposed several techniques to improve the performance of number theoretic transform implementation, which is faster than the state-of-the-art work by Dai et al. The proposed techniques include register-based twiddle factors storage and multi-stream asynchronous computation, which leverage on the features offered in new GPU architectures. The proposed number theoretic transform implementation was applied to CMNT fully homomorphic encryption scheme proposed by Coron et al. With the proposed implementation technique, homomorphic multiplications in CMNT take 0.27 ms on GTX1070 desktop GPU and 7.49 ms in Jetson TX1 embedded system, respectively. This shows that the proposed implementation is suitable for practical applications in server environment as well as embedded system.

Keywords Number theoretic transform · Homomorphic encryption · Graphics processing unit · Cryptography

1 Introduction

Large integer multiplication is widely used in scientific computing and cryptographic algorithms. For instance, it is used in estimating π to more than one million digits, investigating precise behavior of zeta function [1], etc. It is also required for

✉ Wai-Kong Lee
waikong.lee@gmail.com

Extended author information available on the last page of the article

computing some cryptography algorithms, wherein the modular integer multiplication involved is usually very large (100K-bit to 1 M-b) [2]. Hence, there are some prior works attempting to speed up large integer multiplication with hardware accelerator like GPU [2, 3] and FPGA [4].

The first fully homomorphic encryption (FHE) was introduced by Gentry [5] in his pioneering work in 2009. FHE allows arbitrary computation on the encrypted data without exposing the original content through decryption. This implies that many useful applications designed to protect the confidentiality of end users can be implemented in the cloud. These cryptosystems (public key encryption and FHE) require computation of very large integer modular multiplications. However, the first FHE scheme is still not ready for practical use, as the computation takes a very long time to complete. Later on, van Dijk et al. proposed an integer-based FHE [6], which is structurally simpler compared to Gentry FHE scheme [5]; however, it is time-consuming to encrypt one bit as it requires 320K-bit large integer modular multiplication. Moreover, DGHV scheme [6] has very large public key size for security reasons, resulting in slow encryption and decryption speed in implementation.

In 2011, Coron et al. [7] described a compression technique that reduces public key size of DGHV scheme from $O(\lambda^{10})$ to $O(\lambda^{7})$. This scheme (hereafter referred to as CMNT) [7] is more efficient compared to DGHV scheme [6], but it is still executing large integer multiplication of 160K-bit for encryption and homomorphic multiplication, which is time-consuming for real-world application. In view of that, we are motivated to improve the speed performance of large integer multiplication through state-of-the-art GPU architecture, which can be adopted for fast FHE computation.

Schönhage–Strassen multiplication algorithm (SSMA) [8] is widely used to improve the efficiency of large integer multiplication [2, 3]. It allows the integer multiplication to be performed in the frequency domain, which reduces the complexity from $O(n^2)$ to $O(nlog(n)log(log(n)))$, where n is the number of digits. The key operation in SSMA is the number theoretic transform (NTT), which is also the main target for optimization in this paper. To the best of our knowledge, there are two prior works that are closely related to this paper. Wang et al. [2] presented the first implementation of Gentry FHE scheme [5] in GPU, wherein the integer multiplication is performed through parallel implementation of SSMA and NTT. In their implementation, they adopted the optimization techniques proposed by Gentry and Halevi [9], which allow them to achieve impressive speedup on Tesla C2050 GPU platform. Later on, Dai et al. [3] presented cuHE library [10], which focuses on implementing the DHS [11]; somewhat homomorphic scheme in CUDA enabled GPUs with compute capability 3.0 and above. They adopted Chinese remainder theorem (CRT), NTT and Barrett reduction to optimize the implementation of DHS scheme [11]. These two works [2, 3] implemented lattice-based FHE, which is structurally more complicated compared to integer-based FHE [6, 7].

In this paper, we present an improved NTT implementation in GPU, which is faster than the state-of-the-art implementation by Dai et al. [3]. Leveraging this high speed NTT, we implemented the CMNT [7] FHE scheme in GPU with multi-stream computation, which can achieve impressive speedup against CPU implementation.

2 Background

2.1 CMNT homomorphic encryption scheme

CMNT scheme [7] is FHE that allows multiplication and addition to be performed on ciphertext (encrypted domain). This property is useful in secure computing environment, which allows the processing of data without revealing the content (decryption) of data. CMNT is an improvement from the DGHV scheme [6], which requires very large public key size. CMNT [7] is able to achieve shorter public key, because only a smaller subset of the public key is stored instead of the entire public key. One of the parameter sets of CMNT scheme is listed in Table 1; the algorithms in CMNT scheme [7] are described briefly in the following paragraphs.

KeyGen: This is the algorithm to generate public and private keys for encryption and decryption. The KeyGen algorithm is presented in Algorithm 1 and explained here. Generate random prime $p \in \mathbb{Z} \cap [2^{\eta-1}, 2^{\eta})$. Let $x_0 = q_0 \cdot p$ where q_0 is a random integer that its prime decomposition contains no repeated factors and it does not contain prime factors smaller than 2^{λ} in $[0, 2^{\lambda}/p]$. For $1 \leq i \leq \beta$ *and* $b \in 0, 1$, generate integer $x_{i,b}$.

$$x_{i,b} \;= x_{i,b} + p \cdot q_{i,b} + r_{i,b} \quad 1 \leq i \leq \beta \; and \; 0 \leq b \leq 1 \tag{1}$$

where $q_{i,b}$ are the random integers in $[0, q_0)$ and $r_{i,b}$ are the integers in $(-2^{\rho}, 2^{\rho})$.

Algorithm 1: CMNT KeyGen

```
1  **Output:** public key $pk$ and secret key $sk$.
2  Generate random prime $p \in \mathbb{Z} \cap [2^{\eta-1}, 2^{\eta})$;
3  Generate random square free $2^{\lambda}$-rough integer $q_0 \in [0, 2^{\gamma}/p)]$;
4  Let $x_0 = q_0 \cdot p$;
5  **for** $i = 1$ *to* $\beta$ **do**
6      **for** $b = 0$ *to* $1$ **do**
7          Generate random integers $q_{i,b} \in \; [0, q_0)$;
8          Generate random integers $r_{i,b} \in (-2^{\rho}, 2^{\rho})$;
9          Generate integers $x_{i,b} = p \cdot q_{i,b} + r_{i,b}$;
10     **end**
11 **end**
12 $pk = (x_0, x_{1,0}, x_{1,1}, \cdots, x_{\beta,0}, x_{\beta,1})$
13 $sk = p$
```

The authors [7] stated that the scheme is able to recover public key on the fly by combining elements in the small subset multiplicatively. The resulting public key $pk = (x_0, x_{1,0}, x_{1,1}, \ldots, x_{\beta,0}, x_{\beta,1})$; the private key $sk = p$. With this technique,

Table 1 CMNT scheme parameters

Parameters	λ	ρ	η	$\gamma * 10^5$	β	θ
Toy	42	16	1088	1.6	12	144

the scheme only stores smaller subset of public key instead of the entire public key, resulting in a shorter public key.

Encrypt Shorter public key in CMNT scheme leads to minor modification of the original DGHV encryption. The new encryption equation contains additional multiplication operation to include the public key restoration for recovering the actual public key. The steps for CMNT encryption are described in Algorithm 2. Firstly, generate a random vector $\boldsymbol{b} = (b_{i,j})$ of size $\tau = \beta^2$ and with components in $[0, 2^\alpha]$. Then, generate random integer $r \in (-2^{\rho'}, 2^{\rho'})$ to be used for encryption. The equation for encryption is:

$$c = m + 2r + 2 \sum_{i=1,j=1}^{\beta} b_{i,j} \cdot x_{i,0} \cdot x_{j,1} \; mod \; x_0 \tag{2}$$

Algorithm 2: CMNT Encryption

```
1  Input: $m$, one bit message $\in \{0,1\}$.
2  $x_i$ and $x_j$, short public keys.
3  Output: $c$, large integer ciphertext.
4  Generate random integer $r$ in $(-2^{p'}, 2^{p'})$;
5  for $i = 1$ to $\beta$ do
6      for $j = 1$ to $\beta$ do
7          Generate random $b_{i,j}$ of size $\tau = \beta^2$;
8          $temp += b_{i,j} \cdot x_{i,0} \cdot x_{j,1} \; mod \; x_0$
9      end
10 end
11 $c = m + 2r + 2 * temp$;
```

Note that the plaintext (m) is 1-bit in size, while the resulting ciphertext (c) is 160K-bit in size.

Evaluate: This is the execution of homomorphic addition or multiplication between two ciphertexts (160K-bit in size), returning the result in encrypted domain. Note that these operations involve large integer additions and multiplications.

$$c_r = c_1 + c_2 \; mod \; x_0 \tag{3}$$

$$c_r = c_1 * c_2 \; mod \; x_0 \tag{4}$$

Decrypt This algorithm produces plaintext based on the ciphertext and private key. Referring to Algorithm 3, the algorithm takes ciphertext and private key (sk) as inputs and then outputs 1-bit plaintext $m \in \{0,1\}$.

$$m' \leftarrow (c \; mod \; sk) \; mod \; x_0 \tag{5}$$

Algorithm 3: CMNT Decryption

```
1 Input: $c$, large integer ciphertext.
2 $sk$, private key.
3 Output: $m$, one bit message $\in \{0,1\}$.
4 $temp = c \ mod \ sk$;
5 $m' = temp \ mod \ x_0$;
```

From Algorithm 2, it is clear that the most time-consuming operation in encryption is the large integer multiplication (line 8). At the same time, the homomorphic multiplication also involves large integer multiplication. It is our aim in this paper to speed up the CMNT scheme by accelerating the most computational heavy part (large integer multiplication).

Referring to Table 1, $\lambda = 42$, $\tau = 144$, and $\gamma = 1.6 \cdot 10^5$, where τ is the number of x_i in public key and γ is the bit length of x_i. In the encryption process (Algorithm 2, line 8), each of the short public key with γ-bit multiplies with each other to recover the actual public key, which is very time-consuming. On top of that, the short public keys of γ-bit each perform multiplication for β^2 iterations (Table 1). Hence, it can be very slow to be implemented in normal computer system. We proposed several techniques to implement the 196K-bit SSMA in GPU to accommodate the 160K-bit (γ-bit) operands, with some buffer room for potential overflow. The multiplier implemented can be used to handle large integer multiplication (SSMA) and Barrett reduction described in next section.

2.2 Schönhage–Strassen multiplication algorithm (SSMA) with number theoretic transform (NTT)

SSMA [8] is a fast multiplication algorithm for large integer with low computational complexity of $O(n \ log \ n(log \ log \ (n)))$. In brief, SSMA consists of three steps.

1. **Step One:** Provide large integer x and y as input operands. These two large integer operands are broken into multiple limbs of smaller bit size (e.g., 24-bit). Operands are now transformed into frequency domain through NTT.
2. **Step Two:** Operands in frequency domain perform element-wise multiplication.
3. **Step Three:** Result of multiplication is transformed into time domain using inverse NTT.

Referring to Algorithm 4, SSMA takes two large operands, x and y. In line 3, these two operands (x and y) undergo NTT. Then, the operands in frequency domain perform element-wise multiplication (line 4–6). After the multiplication is completed, output z undergoes INTT, converting the results back to time domain (line 7).

Algorithm 4: Schönhage-Strassen Multiplication Algorithm

```
Input: $x_i$ and $y_i$, the series of input data of multiplier and multiplicand in time domain.
Output: $z_i$, large integer results of multiplication between $x_i$ and $y_i$.
$X \leftarrow NTT(x), Y \leftarrow NTT(y)$;
for $i = 0$ to $N - 1$ do
    $Z[i] \leftarrow X[i] * Y[i]$;
end
$z \leftarrow INTT(Z)$;
return $z$;
```

Discrete Fourier transform (DFT) is a technique widely used in signal processing applications, which works in the floating-point domain. NTT is a variant of DFT that computes transformation over finite field in integer domain. Forward NTT (refer to Eq. (6) and Algorithm 5) takes a series of inputs in time domain and then converts them into a series of outputs in frequency domain. In NTT domain, the multiplication can be performed in element-wise operation, which is very easy for parallel implementation. Once the multiplication completes, inverse NTT (INTT, refer to Eq. (7) and Algorithm 6) is executed to recover the results in time domain.

Forward transform

$$X_j = \sum_{i=0}^{N-1} x_i g^{ij} (mod\ m),\ 0 \le j < N \text{ , where } N \text{ is number of NTT points} \tag{6}$$

Inverse transform

$$x_i = N^{-1} \sum_{j=0}^{N-1} X_j g^{ij} (mod\ m),\ 0 \le i < N \text{ , where } N \text{ is number of NTT points} \tag{7}$$

Algorithm 5: Forward Number Theoretic Transform

```
Input: $x_i$ large integers in time domain.
Output: $X_j$ large integers in frequency domain.
for j=0 to N-1 do
    for i=0 to N-1 do
        $X_j = X_j + (x_i * g^{ij})$ mod m;
    end
end
```

Algorithm 6: Inverse Number Theoretic Transform

```
Input: $X_j$ large integers in frequency domain.
Output: $x_i$ large integers in time domain.
for i=0 to N-1 do
    for j=0 to N-1 do
        $x_i = x_i + (X_j * g^{ij})$ mod m;
    end
    $x_i = x_i * N^{-1}$;
end
```

NTT comes with $O(N^2)$ complexity, which is not efficient for implementation when the number of digit N is large. A more efficient way of computing NTT is by using Cooley–Tukey fast Fourier transform (CT-FFT) [12]. CT-FFT allows large FFT to be broken down and computed using multiple smaller-sized FFT in recursive manner, where the level of recursion can be determined based on the implementation requirement. All of these smaller FFT can then be computed in parallel. This technique is commonly used together with SSMA and was proven to be efficient by prior works [2, 3]. Equation (8) shows how CT-FFT can be computed.

Forward Cooley–Tukey fast Fourier transform

$$X_{(N_2 j_1 + j_2)} = \sum_{i_1=0}^{N_1-1} \left[\left(\sum_{i_2=0}^{N_2-1} x_{(N_1 i_2 + i_1)} g_{N_2(i_2 j_2)} \right) g_{N(i_1 j_2)} \right] g_{N_1(i_1 j_1)} (mod\ m) \tag{8}$$

Referring to Algorithm 7, the forward Cooley–Tukey fast Fourier transform, the inputs to CT-FFT are large integer in time domain (x), gN, gN_1 and gN_2. (These are the twiddle factors of N, N_1 and N_2, respectively.) CT-FFT algorithm can be divided into three steps: Step one starts from line 8 to line 16 in Algorithm 7, performing column FFT of size N_2. Following this is the step two (line 18 to 24 in Algorithm 7) that performs the twiddle factors' multiplications. Lastly, line 26 to 32 in Algorithm 7 calculates row FFT of size N_1.

Algorithm 7: Forward Cooley-Tukey Fast Fourier Transform

```
1  **Input:** $x$ large integers in time domain.
2  $t$, series of intermediate value.
3  $gN$, series of twiddle factors for $N$.
4  $gN_1$, series of twiddle factors for $N_1$.
5  $gN_2$, series of twiddle factors for $N_2$.
6  **Output:** $X$ large integers in frequency domain.
7  //Column-FFT
8  for $i_1 = 0$ to $N_1 - 1$ do
9      for $i_2 = 0$ to $N_2 - 1$ do
10         for $j_1 = 0$ to $N_1 - 1$ do
11             for $j_2 = 0$ to $N_2 - 1$ do
12                 $t_{(N_2 j_1 + j_2)} += x_{(N_1 i_2 + i_1)} * gN_{2(i_2 * j_2)}$ mod $m$;
13             end
14         end
15     end
16 end
17 //Twiddle Factor Multiplication
18 for $i_1 = 0$ to $N_1 - 1$ do
19     for $j_1 = 0$ to $N_1 - 1$ do
20         for $j_2 = 0$ to $N_2 - 1$ do
21             $t_{(N_2 j_1 + j_2)} *= gN_{(i_1 j_2)}$ mod $m$;
22         end
23     end
24 end
25 //Row-FFT
26 for $i_1 = 0$ to $N_1 - 1$ do
27     for $j_1 = 0$ to $N_1 - 1$ do
28         for $j_2 = 0$ to $N_2 - 1$ do
29             $X_{(N_2 j_1 + j_2)} += t_{(N_2 j_1 + j_2)} * gN_{1(i_1 j_1)}$ mod $m$;
30         end
31     end
32 end
```

Inverse Cooley–Tukey fast Fourier transform

$$x_{(N_2j_1+j_2)} = N^{-1} \sum_{i_1=0}^{N_1-1} \left[\left(\sum_{i_2=0}^{N_2-1} X_{(N_1i_2+i_1)} g_{N_2(i_2j_2)} \right) g_{N(i_1j_2)} \right] g_{N_1(i_1j_1)} (mod\ m) \tag{9}$$

The inverse CT-FFT (refer to Eq. (9) and Algorithm 8) is similar to forward CT-FFT, but it operates with inverse twiddle factors and X (inputs from frequency domain). CT-FFT is highly parallelizable by GPU, so we have adopted this technique to compute NTT in SSMA. Figure 1 summarizes the SSMA algorithm flow in graphical way.

Algorithm 8: Inverse Cooley-Tukey Fast Fourier Transform

```
Input: $x$ large integers in time domain.
$t$, series of intermediate value.
$gN$, series of twiddle factors for $N$.
$gN_1$, series of twiddle factors for $N_1$.
$gN_2$, series of twiddle factors for $N_2$.
Output: $X$ large integers in frequency domain.
for $i_1 = 0$ to $N_1 - 1$ do
    for $i_2 = 0$ to $N_2 - 1$ do
        for $j_1 = 0$ to $N_1 - 1$ do
            for $j_2 = 0$ to $N_2 - 1$ do
                $t_{(N_2j_1+j_2)} += x_{(N_1i_2+i_1)} * gN_{2(i_2*j_2)}$ mod m;
            end
        end
    end
end
for $i_1 = 0$ to $N_1 - 1$ do
    for $j_1 = 0$ to $N_1 - 1$ do
        for $j_2 = 0$ to $N_2 - 1$ do
            $t_{(N_2j_1+j_2)} *= gN_{(i_1j_2)}$ mod m;
        end
    end
end
for $i_1 = 0$ to $N_1 - 1$ do
    for $j_1 = 0$ to $N_1 - 1$ do
        for $j_2 = 0$ to $N_2 - 1$ do
            $X_{(N_2j_1+j_2)} += t_{(N_2j_1+j_2)} * gN_{1(i_1j_1)}$ mod m;
        end
    end
end
$X_{N_2j_1+j_2} *= N^{-1}$;
```

2.3 Barrett reduction

In order to deal with modular reduction, Barrett reduction [13] comes in handy. The algorithm computes $r = x\ mod\ m$, where x and m are the inputs. Barrett reduction

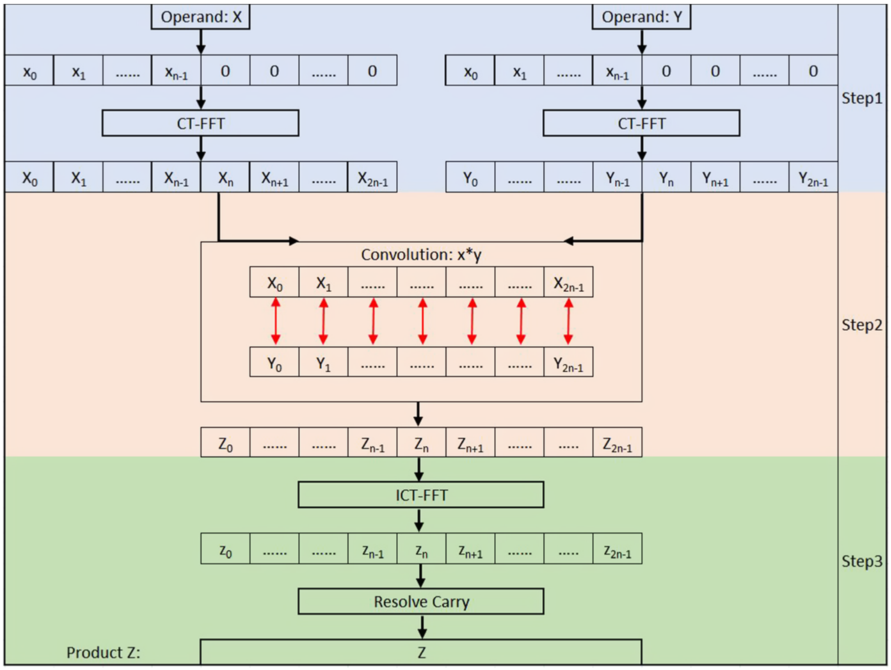
Operand: X
Operand: Y
x_0 x_1 ...... x_{n-1} 0 0 0
CT-FFT
X_0 X_1 ...... X_{n-1} X_n X_{n+1} ...... X_{2n-1}
Y_0 .... Y_{n-1} Y_n Y_{n+1} ...... Y_{2n-1}
Step1
Convolution: x*y
Step2
Z_0 .... Z_{n-1} Z_n Z_{n+1} .... Z_{2n-1}
ICT-FFT
z_0 .... z_{n-1} z_n z_{n+1} .... z_{2n-1}
Resolve Carry
Product Z:
Z
Step3

Fig. 1 SSMA algorithm flow

requires additional precomputation of $\mu = \lfloor b^{2k}/m \rfloor$ (line 4 and 5 in Algorithm 9). However, this precomputation is beneficial to repeated modulus, because μ will remain the same when modulus m is unchanged; we can reuse the precomputed values to improve the speed performance. Besides precomputation, time-consuming division in Algorithm 9 can be implemented using right shifting, allowing a faster and more efficient computation. Lastly, multiplication by 2 (line 4 Algorithm 9) can be implemented as left shifting.

Conventional Barrett reduction only supports single precision modular reduction; we had converted it to support multiple precision modular reduction. In particular, the multiplication in line 6 of Algorithm 9 is replaced by the SSMA to support multi-precision operands, while Barrett reduction is used to compute the remainder of this large integer. Since most of the CMNT function requires the same x_0 as modulus (e.g., Eq. (2, 3, 4 and 5), precomputed μ can be used repeatedly to save computational time without generating a new μ. It is also advantageous to implement Barrett reduction in CMNT because there are many modulus operations that involve the public key x_0 .

Algorithm 9: Barrett reduction

```
Input: $x$, large integer input
$m$, modulus.
Output: $r$, remainder.
$q = 2 * size\_of(x)$;                              ▷ Precomputation
$\mu = \lfloor 2^q/m \rfloor$;                      ▷ Precomputation
$r = x - m * [x * \mu/2^q]$ ;   ▷ Multiplication is replaced with proposed SSMA
while $r \geq m$ do
    r -= m;
end
return $r$;
```

2.4 Overview of GPU architecture

GPU consists of multiple streaming processors (SMs), and each of them is capable of processing tasks individually. Programmers can offload parallel part of their algorithm to be executed in GPU and at the same time process the serial part in CPU. Application can run significantly faster compared to implementation that only utilizes CPU. In 2007, NVIDIA developed CUDA, the programming model for general purpose computing on GPU. Since then, programmers are able to leverage the sheer amount of parallelism in GPU architecture to speed up their programs, which results in many innovative applications adopting GPU in their implementation [14, 15].

GPU contains multiple types of memory, namely global memory, constant memory, texture memory, shared memory and registers. Global and constant memories are the off-chip memory (DRAM) with slow execution speed and cached by L1 and L2 cache. Global memory is large in size (several gigabytes), so it is usually used for storing input data from the CPU or buffering the results from the GPU before sending it to the CPU. Constant memory is used for storing read-only data. Texture memory is also a read-only memory, but it is optimized for 2D spatial locality, which can be more advantageous to global and constant memory if the access pattern does not coalesce.

On the other hand, shared memory is user managed cache with high accessing speed compared to global memory. Register is the fastest memory in GPU architecture, but it has very limited size, and it is only accessible within a thread. To share data across different threads, one has to rely on shared memory (same block) or global memory (different blocks), which results in high memory latency.

Furthermore, NVIDIA GPU also supports dynamic parallelism and warp shuffle. Dynamic parallelism is a feature that allows CUDA kernel to create new threads and synchronize new tasks directly inside GPU, reducing the need to transfer execution control and data between CPU and GPU. Warp shuffle is an instruction that allows GPU threads to share register data in a group of 32 (which is known as a *warp*), thus reducing time to fetch data from shared memory or global memory.

3 Proposed techniques for CMNT implementation in GPU

This section presents the fast NTT implementation in GPU based on our proposed methods, together with the acceleration of CMNT scheme with GPU.

3.1 Proposed Cooley–Tukey Fast Fourier Transform implementation

For SSMA to be implemented correctly, there are some restrictions to follow, which is also described by Emmart et al. [16]. The maximum convolution value must fit in the field, $(k/2)(b-1)^2 < p$, where k is the size of FFT, b is the bit size per element, and p is the prime number. Prime number p is set to be 0xFFFFFFFF00000001 due to a number of useful properties. With p being 0xFFFFFFFF00000001, the field $\mathbb{Z}/p\mathbb{Z}$ supports power-of-2 NTT sizes up to 2^{32}. Besides that, 64-point NTT twiddle factor multiplication (Refer to Sect. 3.1 second Kernel) can be done with shifting rather than performing 64-bit by 64-bit multiplications, because 8 is the root of unity (g) for 64-point NTT. As 64-point NTT twiddle factor can be computed using g^N where $N : \{0, 1, 2, \ldots, 63\}$, we get $8^N = (2^3)^N$; therefore, multiplication of 64-point twiddle factor ($X_j = \sum x_i * g^{ij}$) can be done with left shifting.

The state-of-the-art GPU implementation of NTT is presented by Dai et al. [3], and the codes are released as open source (cuHE) [10]. In the cuHE implementation, the twiddle factors are stored in texture memory and loaded into shared memory during the NTT computation, with the aim of reducing memory access latency. However, shared memory may suffer from severe bank conflict issues if the amount of data to be process is huge. In view of that, we proposed to buffer the twiddle factors in GPU registers and then access these data across different threads during the NTT computation through *warp shuffle* instruction. Warp shuffle is a relatively new instruction that improves data fetching latency of GPU, released in year 2013 through Kepler architecture GPU [17]. It speeds up the data access by sharing data between different threads through the fast registers, instead of fetching data from shared memory that can be slower due to bank conflicts. In cuHE implementation, there are three kernels responsible for forward NTT conversion, which are described in detail in the following paragraphs.

CT-FFT decomposition first breaks N elements NTT into smaller NTT (N_1 and N_2), where $N = N_1 \times N_2$, in which N_1 represents the number of row FFT and N_2 is the number of column FFT.

First kernel Referring to Fig. 2, column FFT is implemented with N_2 number of blocks with 64 threads each; each thread computes 8-point NTT simultaneously. Therefore, each block is capable of computing 512-point NTT (64 threads $\times$ 8-point NTT) concurrently; this is actually corresponds to line 8 to 16 in Algorithm 7. The number of blocks launched in parallel to compute the column-NTT for both input operands (x and y) is $2 \times N_2$.

Second kernel Twiddle factors multiplication is implemented in this kernel. It multiplies the output from first kernel with the corresponding twiddle factors. Unlike the first kernel, every twiddle factor is only used once; therefore, there is no advantage to buffer all of them in registers or shared memory. This corresponds to line 18–24 in Algorithm 7.

Third kernel Referring to Fig. 3, row FFT is implemented with N_1 blocks, 64 threads each; each thread computes 8-point NTT simultaneously. Therefore, each block is capable of computing 512-point NTT (64 threads × 8-point NTT) concurrently. This corresponds to line 26–32 in Algorithm 7. Total blocks launched to compute row-NTT for both operands (x and y) are $2 \times N_1$.

cuHE [10] implementation stores the twiddle factors in shared memory to reduce the memory access latency. The indexing pattern can be described through Eq. (10):

$$roots = tf[(tidx >> 3)|(i << 3)]),\ 0 \leq i < 8 \tag{10}$$

Each thread with unique ID ($tidx$) accesses one twiddle factor, and it was repeated for eight times. Our proposed technique requires more sophisticated indexing compared to using shared memory.

However, we found that storing twiddle factors in GPU shared memory may inflict bank conflicts, which decreases the performance. We propose to store the twiddle factors in registers and access it through warp shuffle instruction. Warp shuffle technique is applied to twiddle factors multiplication of the first kernel for faster data access. Prior to the implementation of warp shuffle in cuHE NTT kernel, the access pattern of each thread is observed. Once the access pattern is determined, a generalized equation is formulated to assign twiddle factors for registers in each thread within a block. For the same 64K-point NTT, one twiddle factor is stored in the register by using the generalized equation (11).

$$reg0 = tf[((tidx\%8) * (bidx >> 1) * 128 + (16 * (bidx >> 1) * (tidx >> 3)))] \tag{11}$$

reg0 of Eq. (11) stores twiddle factor (*tf*) of the specific index, according to their thread number (*tidx*) and block number (*bidx*). After the indexing is completed, warp shuffle instruction (*__shfl*) in CUDA allows each thread to access the register of other threads within the same warp (32 threads). With this method, the twiddle factors can be shared among threads from the same warp and do not suffer any memory bank conflicts that the shared memory does. Table 2 shows a small example (for illustration purpose) of our warp shuffle implementation with eight threads per block; note that the block size is larger in actual implementation.

Referring to Table 2, the thread IDs are labeled as T0–T7 respectively, and each thread requires eight register values in order to compute. The first row of the table is twiddle factors that each thread stores in their respective register, and the row “Required value” is the thread Tx’s register content, where x = 0,1,2,3,4,5,6,7. For thread T0, it requires register content from T1, T2, T3, T4, T5, T6 and T7, which can be read through warp shuffle. For thread T1, it requires T0, T2, T3, T4, T5, T6 and T7; this pattern is similar for the remaining threads.

With warp shuffle implemented, each thread holds only one twiddle factor and shares twiddle factor among threads within a warp. As compared to original implementation, all 64 twiddle factors are fetched from shared memory even though only

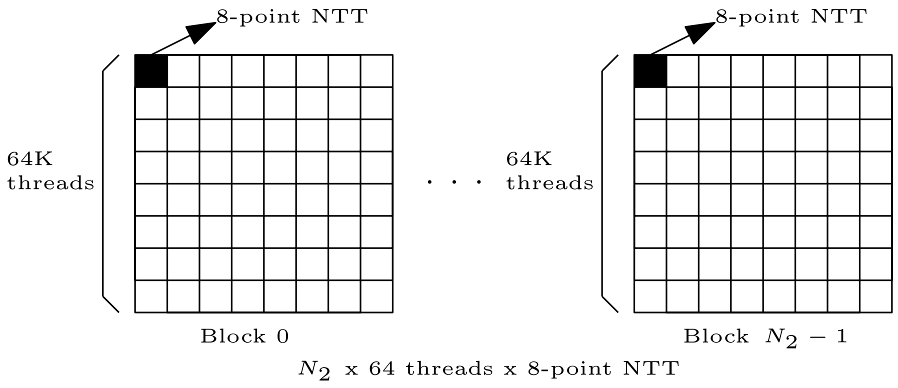

Fig. 2 CT-FFT kernel one

8 are used in each thread. Our warp shuffle implementation is more efficient, albeit more complicated. We are able to obtain a speedup compared to the original cuHE shared memory storage method.

3.2 Proposed SSMA with Cooley–Tukey Fast Fourier transform

SSMA (Algorithm 4) is being modified to integrate with CT-FFT (Algorithm 7) and implemented in GPU for further speed improvement. Emmart et al. [16] found that the element samples should be byte-aligned to keep load and store routines simple and efficient. For $b = 2^{24}$ and p = 0xFFFFFFFF00000001, k must be less than or equal to 2^{17}, where $(k/2)(b-1)^2 < p$. Therefore, we end up using 24-bit per element for 16K-point and 64K-point FFT, while the modulo prime p is 0xFFFFFFFF00000001. The maximum operand size for our SSMA implementation is 196K-bit for 16K-point FFT and 786K-bit for 64K-point FFT, calculated using the formula $(k/2) \times b$. 16K-point FFT SSMA is implemented to accommodate 160K-bit

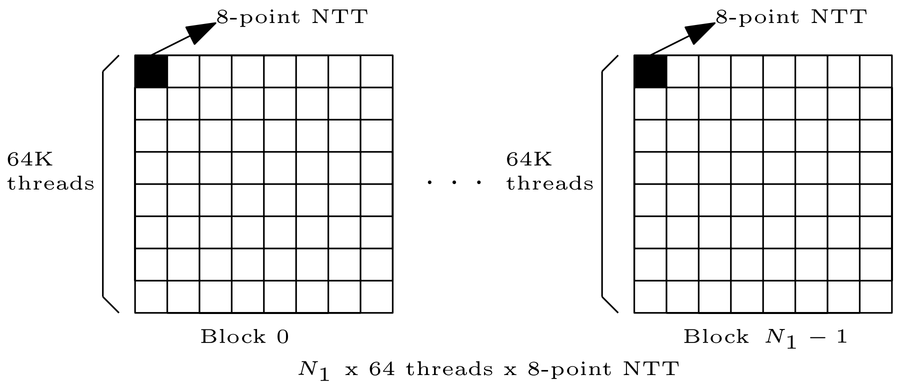

Fig. 3 CT-FFT kernel three

operands, as needed in CMNT encryption and homomorphic multiplication. On the other hand, 64K-point FFT SSMA is implemented for larger operand size.

Besides that, we also proposed multi-stream technique to further accelerate the execution of SSMA kernel. Multi-stream technique enables multiple different CUDA operations to run in parallel, so that both CT-FFT kernels are able to execute concurrently. For instance, inputs *x* and *y* are able to be transformed into NTT domain in parallel, provided that there are available resources in GPU. Through experiments, we found that the optimum number of streams is four; increasing the number of streams beyond four does not give any performance advantage. This is due to the fact that the GPU hardware resources are already fully occupied, so giving more streams of workloads does not improve the performance further. Our proposed SSMA is able to compute four sets of operands multiplication at the same instance. Each stream will handle one set of operands as illustrated in Figs. 4 and 5.

Figure 5 shows the overlapping effect of four streams kernel execution in GPU. Firstly, section S1 shows overlapping of memory copy from CPU to GPU memory. Followed by section S2, execution of four sets of operands CT-FFT forwards transformation. Then, section S3 indicates element-wise multiplication. Lastly, section S4 shows the results of four SSMA being transformed back to the time domain. Most operations in GPU are performed in parallel as shown in Fig. 5. However, kernels in section S3 complete their executions in a short duration, so the overlapping effect does not present. With multi-stream technique, our implementation ensures that the GPU is almost fully utilized for memory copy and computation, thus achieving high performance.

3.3 Implementation of CMNT

In this section, we present the implementation of the encryption algorithm (see Algorithm 2) in CMNT scheme, through the GPU-based SSMA discussed in Sect. 3.2. Public key multiplication is being computed in line 8 of Algorithm 2, using our proposed multi-stream SSMA and CT-FFT with warp shuffle. Multiplication can be performed in parallel thus resulting in faster encryption. Besides the proposed SSMA implementation, we also adopted Barrett reduction [13] to support

Table 2 Thread register warp shuffle

Reg value	Thread ID							
	T0	T1	T2	T3	T4	T5	T6	T7
	0	1	2	3	4	5	6	7
Required value	T1	T0	T0	T0	T0	T0	T0	T0
	T2	T2	T1	T1	T1	T1	T1	T1
	T3	T3	T3	T2	T2	T2	T2	T2
	T4	T4	T4	T4	T3	T3	T3	T3
	T5	T5	T5	T5	T5	T4	T4	T4
	T6	T6	T6	T6	T6	T6	T5	T5
	T7	T7	T7	T7	T7	T7	T7	T6

Fig. 4 Multi-stream parallel SSMA

$$Stream1 \longrightarrow SSMA(X_1 \times Y_1)$$
$$Stream2 \longrightarrow SSMA(X_2 \times Y_2)$$
$$Stream3 \longrightarrow SSMA(X_3 \times Y_3)$$
$$Stream4 \longrightarrow SSMA(X_4 \times Y_4)$$

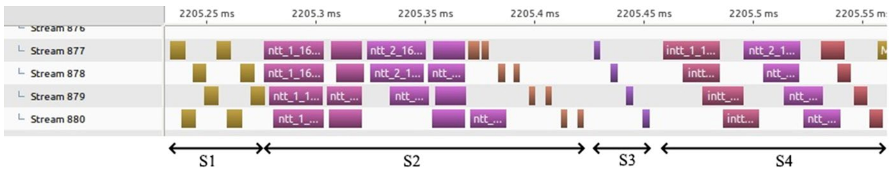

Fig. 5 SSMA GPU profiler

large integer modular reduction needed in CMNT scheme. Similarly, we replaced large integer multiplication operation in Barrett reduction with our proposed SSMA.

Furthermore, we analyzed CMNT scheme encryption and noticed that all the message bit encryption uses the same public key. Hence, we optimized the encryption process by precomputing the public key multiplication in line 8 of Algorithm 2 and stored it for all subsequent encryptions. Lastly, we also replaced the multiplication by two (in line 11 of Algorithm 2) with a more efficient left shift operation.

On the other hand, homomorphic multiplication is accelerated in GPU using our proposed SSMA. The large integer multiplication is performed using 16K-point CT-FFT which supports up to 196K-bit operands. Multi-stream SSMA is able to compute up to four sets of homomorphic operations in parallel. The result is being averaged by four and tabulated in next section.

4 Experimental results

The techniques described in this paper are implemented on both desktop PC (equipped with GPU) and NVIDIA Jetson TX1 embedded system. The source code for cuHE is obtained from the public domain (published by the authors) [10]. The specifications of desktop PC and NVIDIA Jetson TX1 are listed in Table 3.

Table 4 compares our CT-FFT with warp shuffle over cuHE [10] for both 16K-point FFT and 64K-point FFT. Experiments were carried out using randomly generated input operands to perform NTT, and the average timing is taken over ten iterations. The NTT operations (16K-point FFT and 64K-point FFT) are performed entirely in the respective GPU device. Compared to cuHE, our proposed implementation is able to achieve 1.16 times and 1.44 times speedup for 16K-point and 64K-point FFT, respectively, on desktop PC.

Table 5 shows the results of our SSMA implementation, which incorporate the proposed CT-FFT with warp shuffle. The results also show improvement against cuHE [10], when multiple streams are being used. Warp shuffle version of 16K

CT-FFT SSMA is slower than cuHE implementation when there is only one stream being used. On the other hand, the same technique applied to 64K CT-FFT SSMA shows improvement (ranging from 0.9–35.9%) over cuHE implementation. This is due to the fact that cuHE [10] stores the twiddle factors in shared memory, which may not have serious bank conflict issues when the NTT size is small and only one stream is being used. For larger NTT size (64K-point) and multiple streams involved, the bank conflicts are more serious as there are more operations being computed in parallel; hence, the proposed warp shuffle technique is more advantageous under such scenario.

CMNT ciphertext homomorphic multiplication experiments are carried out on both desktop PC and embedded device (TX1), where the results are tabulated in Table 6. Note that homomorphic multiplication is essentially one SSMA operation with 16K-FFT (160K-bit operands). The comparison is made between average of four parallel SSMA in GPU and single SSMA in CPU. Referring to the SSMA described in Algorithm 4, the execution time is measured from line 3 to line 7, which includes forward CT-FFT, convolution (element-wise multiplication) and inverse CT-FFT. Our proposed solution (warp shuffle-based CT-FFT and multi-stream computation) is the fastest among all SSMA implementations that runs on desktop PC. It shows 725 times speedup compared to CPU-only implementation.

Table 7 shows the CMNT encryption time (Algorithm 2, line 4–line 11) for $\lambda = 42$. The slowest timing performance is from CPU-only implementation, and it takes around 30 s to encrypt one bit message with security parameter $\lambda = 42$. Our proposed implementation is able to speed up encryption process up to 55 times compared to CPU-only implementation. On Jetson TX1 embedded system, the speedup for our proposed implementation is lesser, due to the fact that TX1 has lesser GPU cores compared to GTX1070.

5 Conclusions and discussions

In this paper, we present the implementation of CT-FFT, SSMA and Barrett reduction in GPU platform, together with the acceleration of CMNT encryption and homomorphic multiplication. Our proposed CT-FFT with warp shuffle shows improvement of 16 and 44% over cuHE [3] for 16K-FFT (support 192K-bit operands) and 64K-FFT (support 768K-bit operands). Furthermore, the proposed implementation of CT-FFT with warp shuffle shows even more improvement when multiple streams are being used. We are able to achieve up to 41% improvement with multi-stream SSMA compared to single-stream SSMA in cuHE. With our proposed SSMA implementation, it only takes 0.27 ms to compute one homomorphic multiplication, which is 725.93 times faster than the CPU-only implementation. Similarly, speedup of 55.74 times compared to CPU-only implementation is also observed for CMNT encryption with our proposed techniques. Our proposed techniques can be adopted in desktop CPU (for cloud computing) and embedded system (for edge computing). For example, under IoT communication architecture, cloud server equipped with GPU can employ our implementation techniques to perform data analysis on sensor data, in encrypted domain. On the other hand, the homomorphic

Table 3 Experimental setup specification

	Desktop PC	NVIDIA Jetson TX1
CPU	Intel i7-6700K	ARM Cortex-A57
RAM	16GB	4GB
GPU	GTX 1070 Pascal	Maxwell
CUDA cores	1920	256

Table 4 Time taken for forward CT-FFT on desktop PC

N-point FFT	Time taken for CT-FFT(ms)		Speedup
	cuHE	warp shuffle (proposed technique)	
16 K (196K-bit operands)	0.06678	0.05781	1.15515
64 K (768K-bit operands)	0.11582	0.08018	1.44448

Table 5 Comparison of SSMA timing performance between cuHE and the proposed multi-stream SSMA

No of stream	SSMA with 16K CT-FFT(ms)			SSMA with 64K CT-FFT(ms)		
	cuHE	Warp shuffle	Speedup against cuHE	cuHE	Warp shuffle	Speedup against cuHE
1	0.380	0.382	0.995	0.454	0.450	1.009
2	–	0.292	1.301	–	0.378	1.201
4	–	0.269	1.413	–	0.334	1.359

computation (addition and multiplication) can also be performed on gateway device equipped with GPU, which is considered as edge computing.

Our implementation contains several limitations, which can be improved in future. Relinearization [7] can also be implemented in GPU, so that the CMNT scheme can be used to compute infinite homomorphic operations, which is useful in certain applications. Besides, the existing work can be extended to support multiple bits homomorphic operations, instead of single-bit operation. These are the two directions that we plan to pursue as our future work.

Table 6 Time taken for homomorphic multiplication

Device	Homo. multiplication	Time taken (ms)	Speedup
Desktop PC	CPU only	196	1
	cuHE (GPU)	0.38	515.79
	Proposed method(GPU)	0.27	725.93
Jetson TX1	CPU only	799	1
	Proposed method(GPU)	7.49	106.68

Table 7 Time taken for CMNT encryption, $\lambda = 42$

Device	Encryption implementation	Time taken (s)	Speedup
Desktop PC	CPU only	29.54	1
	Proposed method(CPU + GPU)	0.53	55.74
Jetson TX1	CPU only	230.81	1
	Proposed method(CPU + GPU)	21.76	10.61

Acknowledgements This project is funded by Universiti Tunku Abdul Rahman Research Fund (UTARRF) under the grant number IPSR/RMC/UTARRF/2016-C1/G1. Part of the research work is funded by Fundamental Research Grant Scheme (FRGS), Malaysia, under project number FRGS/1/2018/STG06/UTAR/03/1.

References

1. Harvey D (2015) Computing zeta functions of arithmetic schemes. Proc Lond Math Soc 111(6):1379–1401
2. Wang W, Hu Y, Chen L, Huang X, Sunar B (2015) Exploring the feasibility of fully homomorphic encryption. IEEE Trans Comput 64(3):698–706
3. Wei D, Berk S (2015) cuHE: a homomorphic encryption accelerator library. International conference on cryptography and information security in the balkans. Springer, Berlin
4. Öztürk E, Doröz Y, Savas E, Sunar B (2016) A custom accelerator for homomorphic encryption applications. IEEE Trans Comput 66(1):3–16
5. Gentry C (2009) A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University
6. Van Dijk M, Gentry C, Halevi S, Vaikuntanathan V (2010) Fully homomorphic encryption over the integers. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp 24–43
7. Coron J-S, Naccache D, Tibouchi M (2012) Fully homomorphic encryption over the integers with shorter public keys. Advances in cryptology—CRYPTO 2011. CRYPTO. Lecture Notes in Computer Science, vol 6841. Springer, Berlin, Heidelberg, 2011
8. Schönhage A, Strassen V (1971) Schnelle Multiplikation grosser Zahlen. Computing 7:281–292
9. Gentry C, Halevi S (2011) Implementing Gentry's fully-homomorphic encryption scheme. In: Proceedings advances in cryptology-EUROCRYPT, pp 129–148
10. CUDA Homomorphic Encryption Library. https://github.com/vernamlab/cuHE. Accessed 1 Apr 2019
11. Doröz Y, Shalverdi A, Eisenbarth T, Sunar B (2014) Toward practical homomorphic evaluation of block ciphers using prince. In: 2nd workshop on applied homomorphic cryptography and encrypted computing
12. Cooley JW, Tukey JW (1965) An algorithm for the machine calculation of complex Fourier series. Math Comput 19(90):297–301

13. Barrett P (2006) Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Advances in cryptology—CRYPTO' 86. Lecture Notes in Computer Science, vol 263, pp 311–323
14. Li F, Ye Y, Tian Z, Zhang X (2018) CPU versus GPU: which can perform matrix computation faster–performance comparison for basic linear algebra subprograms. Neural Comput Appl 31:4353–4365
15. Hernández DE, Olague G, Hernández B, Clemente E (2018) CUDA-based parallelization of a bio-inspired model for fast object classification. Neural Comput Appl 30(10):3007–3018
16. Emmart N, Weems CC (2011) High precision integer multiplication with a GPU using Strassen's algorithm with multiple FFT sizes. Parallel Process Lett 21(3):359–375
17. Kepler—The World's fastest, most efficient hpc architecture. https://www.NVIDIA.in/object/NVIDIA-kepler-in.html. Accessed 21 Apr 2019
18. Fully-Homomorphic-DGHV-and-Variants. https://github.com/deevashwer/Fully-Homomorphic-DGHV-and-Variants. Accessed 1 Apr 2019

Affiliations

Jia-Zheng Goey[1] · Wai-Kong Lee[2] · Bok-Min Goi[1] · Wun-She Yap[1]

Jia-Zheng Goey
goey.jz93@1utar.my

Bok-Min Goi
goibm@utar.edu.my

Wun-She Yap
yapws@utar.edu.my

[1] Universiti Tunku Abdul Rahman, Jalan Sungai Long, Bandar Sungai Long, 43000 Kajang, Malaysia

[2] Universiti Tunku Abdul Rahman, Jalan Universiti, Bandar Barat, 31900 Kampar, Malaysia