High-Performance FV Somewhat Homomorphic Encryption on GPUs: An Implementation using CUDA

Ahmad Al Badawi[1,2], Bharadwaj Veeravalli[1], Chan Fook Mun[2] and Khin Mi Mi Aung[2]

[1] Faculty of Engineering, National University of Singapore, Singapore
ahmad@u.nus.edu, elebv@nus.edu.sg
[2] A * STAR, Data Storage Institute, Singapore
{chanfm,Mi_Mi_AUNG}@dsi.a-star.edu

Abstract. Homomorphic encryption (HE) offers great capabilities that can solve a wide range of privacy-preserving computing problems. This tool allows anyone to process encrypted data producing encrypted results that only the decryption key's owner can decrypt. Although HE has been realized in several public implementations, its performance is quite demanding. The reason for this is attributed to the huge amount of computation required by secure HE schemes. In this work, we present a CUDA-based implementation of the Fan and Vercauteren (FV) Somewhat Homomorphic Encryption (SHE) scheme. We demonstrate several algebraic tools such as the Chinese Remainder Theorem (CRT), Residual Number System (RNS) and Discrete Galois Transform (DGT) to accelerate and facilitate FV computation on GPUs. We also show how the entire FV computation can be done on GPU without multi-precision arithmetic. We compare our GPU implementation with two mature state-of-the-art implementations: 1) Microsoft SEAL v2.3.0-4 and 2) NFLlib-FV. Our implementation outperforms them and achieves on average 5.37x, 7.37x, 22.22x, 5.11x and 13.18x (resp. 2.03x, 2.94x, 27.86x, 8.53x and 18.69x) for key generation, encryption, decryption, homomorphic addition and homomorphic multiplication against SEAL-FV$_{\text{RNS}}$ (resp. NFLlib-FV).

Keywords: Homomorphic Encryption, FV, Parallel Processing, GPGPU, CUDA

1 Introduction

Nearing 8 years since their introduction, Fully Homomorphic Encryption (FHE) schemes are set to revolutionize privacy-preserving computing by allowing one to perform arbitrary operations on encrypted data without the need for the decryption key at any stage of computation. This scheme serves as a winning solution for cloud computing applications since it guarantees high level of privacy, confidentiality and serviceability at the same time. For instance, in private single-client cloud computing paradigm [VDJ10], thin clients with limited resources and computing power lease computation and storage facilities from powerful thick servers. Clients upload their data to the cloud which, aside from storage, is entitled to manipulate the data as per the owner's requests. However, once a client uploads her data to the cloud, she has no guaranteed global control over it. In particular, the client can not assert where the data will be stored, who can access it, or how it will be used raising up concerns about her privacy. FHE is considered a promising solution to overcome privacy issues by allowing clients to store their data encrypted on a non-necessarily trusted server, which can evaluate homomorphically arbitrary circuits (functions) on the encrypted data, and be able to process, search and retrieve their data without entrusting the server with their decryption keys.

Although the notion of FHE was introduced by Rivest, Adleman and Dertouzos in 1978 [RAD78], the first realization of FHE was only proposed in 2009 when Craig Gentry proposed a lattice based FHE scheme in his seminal PhD thesis [Gen09]. Gentry's ingenious idea can be simplified as follows: he starts with a noise (error)-based encryption scheme that hides the plaintext by some noise which can be completely filtered out by the decryption function. His scheme can homomorphically evaluate a limited set of circuits including its decryption circuit and one additional operation (NAND gate), in what is called Somewhat Homomorphic Encryption (SHE) scheme. Noise increases with each homomorphic evaluation until it blows up at some point and decryption cannot recover the plaintext anymore. To control the noise, he defines a Recrypt algorithm that takes as input: 1) an encryption of a worn out ciphertext[1], i.e., it contains large noise and 2) an encryption of the decryption key. Recrypt evaluates homomorphically the decryption circuit using the encrypted ciphertext and the encrypted decryption key to remove the inner encryption and outputs a substitute ciphertext (i.e., it encrypts the same plaintext) with less noise allowing for extra operations to be performed homomorphically. By repeating this process infinitely, he realized a FHE scheme that can evaluate arbitrary circuits of any size homomorphically. Gentry named this iterative procedure bootstrapping. The importance of Gentry's notion is twofold: 1) it proved theoretically that FHE is feasible in principle and 2) it formed as a basis blueprint for all subsequent FHE constructions.

Despite the fact that several FHE schemes have been realized in practice through a number of various implementations, they all require intensive amounts of computation and storage. This is attributed to the computations embodied in large number multiplications, large polynomial multiplications, large matrix multiplications or tensor products depending on the underlying scheme. To alleviate this problem, researchers proposed several techniques such as levelling the FHE scheme, which implies restricting the circuit multiplicative depth that can be evaluated on ciphertexts to a predefined threshold [BGV12]. Although this leaves us with SHE schemes, they can still be used for a set of specific applications. SHE schemes are in general more practical compared to FHE, nevertheless, they still require significant amount of computing power.

Another hot research track for improving HE schemes performance is to offload computations to hardware accelerators. FPGAs, ASICs and GPUs have proved to be powerful accelerators in various application domains [BFF12, MV08, Rup12]. They also have been used in accelerating some HE schemes [WHC$^+$12, PNPM15, DÖS15]. A common feature in the previous implementations is that only some parts of HE computation, usually polynomial multiplication, run on the hardware accelerator. They include high number of host-to-device and device-to-host data transfer rates aggravating the overall performance. In this work, we follow this track and explore the feasibility of accelerating a variant of the FV SHE [FV12] scheme using CUDA-enabled GPUs through GPGPU implementation. However, the main feature in our implementation is that all FV primitives: key generation, encryption, decryption and homomorphic addition and multiplication run entirely on GPU reducing host-to-device and device-to-host transfer rates. We also avoid multi-precision arithmetic (in the critical path of computation) by utilizing several algebraic tools, modular algorithms and adaptations of the original FV scheme proposed by Bajard et al. [BEHZ16]. We compare our implementation with Microsoft SEAL v2.3.0-4 [KLP16] and NFLlib-FV [AMBG$^+$16] implementations of the FV scheme. Implementation results show that our GPU solution outperforms them for the majority of parameter settings. The improvement is more evident with large system parameters that provide 80-bit security level and large multiplicative depth ($L \geq 5$). With small settings, our implementation outperforms them except in key generation and homomorphic addition due to data transfer between host

[1]At this point we have a doubly encrypted plaintext

and device and kernel launch overhead.

Precisely, the main contributions and scope of the paper can be summarized as follows:

- Accelerating the FV SHE scheme performance via a GPGPU-based implementation on CUDA-enabled GPUs.
- Assuring that all FV primitives are performed on GPU without multi-precision arithmetic on the critical path of computation by using several algebraic tools.
- Introducing a set of optimization techniques tailored for CUDA-enabled devices to boost up performance to optimum level.
- Comparing the proposed implementation against state-of-the-art serial and parallel implementations of the FV scheme.

1.1 Organization of the Paper

The rest of the paper is organized as follows: in Section 2, we touch on several software and hardware implementations of HE schemes with more focus on the FV scheme. Section 3 introduces briefly the general CUDA programming paradigm. Next, in Section 4 we introduce some algebraic tools such as the Chinese Remainder Theorem (CRT) and Discrete Galois Transform (DGT). We review the textbook FV scheme and its Residual Number System (RNS) variant in Section 5. In Section 6 we provide the layout of our implementation and optimization techniques used. Our experiments and comparison results are exposed in Section 7. Finally, Section 8 concludes the work and provides guidelines for potential future work.

2 Literature Review

Improving HE performance using hardware accelerators has recently started to gain greater attention from researchers. A few number of studies investigated the inclusion of different hardware platforms for that purpose. These platforms include FPGAs, ASICs and GPUs. FPGAs and GPUs may be considered more flexible and cheaper compared to ASICs making them more popular as hardware accelerators. To the best of our knowledge, the first work that employed FPGAs for HE is due to Cousins et al. [CGRS14]. Their work introduces elementary building blocks for implementing an NTRU-based FHE [SS11] using FPGA and Matlab Simulink. Unfortunately, the report does not include implementation results. A more recent study conducted by Doroz et al. [DÖS15] reports that encryption, decryption and recryption procedures of the Gentry-Halevi (GH) FHE [GH11] take 18.1 msec, 16.1 msec and 3.1 sec respectively, in a customized ASIC implementation.

More related to our work is Wang et al. [WHC$^+$15] GPU implementation of the GH-FHE. The authors managed to accelerate the scheme for small and medium parameter settings, however, for secure recommended large parameters, they failed as GPU memory was not sufficient to store all required precomputed constants. Nevertheless, their work shows that GPUs have potential to accommodate for HE computational requirements. Next, we touch on more related work where the FV scheme is implemented.

The FV SHE scheme was proposed by Fan and Vercauteren in [FV12]. The scheme can be viewed as an extension of Brakerski's Learning with Errors (LWE)-based FHE [Bra12] to the ring version RLWE. FV is a SHE scheme that can be bootstrapped and made FHE. The scheme has received widespread attention due to its simple construction and practical performance. One particular feature of the FV is known as scale invariance where modulus switching technique is not necessary for error control. Moreover, the scheme has been implemented by two different teams: 1) Microsoft [KLP16] and 2)

CryptoExperts [AMBG$^+$16]. We use their implementations to evaluate the performance of our GPU based implementation.

SEAL (v2.3.0-4) is a C++ CPU implementation that implements large number and polynomial arithmetic natively. Previous versions ($\leq$ v2.2) of SEAL implement the textbook FV scheme using multi-precision arithmetic. Starting from version 2.3, the library implements a variant of the scheme known as RNS FV proposed by Bajard et al. [BEHZ16]. As will be shown in Section 6, the performance has dramatically improved which highlights the importance of Bajard's et al. adaptations. Major properties of SEAL include a slight modification of the decryption and homomorphic multiplication procedures. In particular, decryption may accept ciphertexts of arbitrary size by using appropriate powers of the secret key. In addition, relinearization is made optional by allowing the ciphertext to grow in size after homomorphic multiplication. In addition, SEAL employs Nussbaumer's algorithm to compute the Number Theoretic Transform (NTT) for polynomial multiplication in $\mathbb{Z}_q[x]/\langle x^{2^k}+1\rangle$ and uses special primes as ciphertext coefficient modulus for efficient modulo operation. The library provides a user-friendly interface that includes several features such as: encoding functions for representing data in polynomials, which is a key requirement of FV, default parameters selector, noise monitoring and CRT batcher.

On the other hand, NFLlib-FV is a parallel multi-core CPU implementation. It utilizes NFLlib, an NTT based fast lattice library that performs polynomial arithmetic in $\mathbb{Z}_q[x]/\langle x^{2^k}+1\rangle$. NFLlib utilizes vector instructions (SSE and AVX2) to perform arithmetic on multiprocessor architectures. The library compares favorably against well-known number theory libraries such as NTL [S$^+$01] and FLINT [Har13]. Unlike SEAL, NFLlib-FV has less features. For instance, it does not include encoding or CRT batching techniques.

3 CUDA Programming Paradigm

GPU, or any other hardware accelerator, can be viewed as an auxiliary set of compute elements that can aid the CPU in computing a specific task. The task at hand needs to be manually divided and assigned to compute elements through painstaking map and reduce procedures. Although some parallel computing frameworks, such as OpenACC [WSTaM12], reduce the complexity of this task, for specific problems map and reduce are often performed manually by the programmer to fully utilize the hardware. The most popular parallel computing frameworks are: OpenCL [Mun09] and CUDA [Nvi07]. Both frameworks extend the C language and provide a set of APIs for parallel programming on several platforms. While CUDA is supported only by NVIDIA devices such NVIDIA GPUs, OpenCL works with various platforms including NVIDIA GPUs. Unlike OpenACC, the programmer has to do the map and reduce procedures in these frameworks. In this work, the CUDA GPGPU programming framework is used. We provide here a basic overview of CUDA GPU hardware and programming paradigm.

CUDA GPU includes a number of Streaming Multiprocessors (SMs). An SM comprises several cores, also known as Streaming Processors (SPs), on chip that share computing resources such as shared memory, L1 cache, register file and special function unit. The SM can be viewed as a vector processor able to execute the same instruction on a number of operands, also known as Single Instruction Multiple Data (SIMD) instructions, by launching a number of threads, each running on a single core. CUDA provides a set of intrinsic variables providing meta-data to uniquely identify the threads.

The programmer determines the total number of threads executing a specific kernel (part of the code executed by CUDA threads) by organizing them in two structures: thread blocks and grids. A thread block is 1-3 dimensional structure specifying the number of threads in each dimension. Thread blocks are also grouped in grids of 1-3 dimensions specifying the number of blocks in each dimension. The sizes of thread blocks and grids are limited by the capabilities of the hardware. A thread block is assigned to run on a single SM by the GPU. These threads are also grouped into warps[2]. The GPU guarantees that threads in a warp run concurrently on one SM. The number of warps that can simultaneously run on an SM is hardware-specific depending on the number of warp schedulers. Although each thread gets its own set of registers, note that threads in a warp may access each other's registers through CUDA shuffle instructions.

Threads in one block can communicate directly through shared memory. In contrast, threads of different blocks can only communicate through global memory. In device code, synchronization points among threads in a block are possible, whereas global synchronization points among all threads (in different blocks) was not possible until CUDA 9 [Cud] through cooperative threads. Global synchronization can also be done by computing large tasks through several kernels launches.

CUDA GPUs include various types of memories differing in access speed, scope and read/write privileges. Global memory for instance is large, slow, persistent throughout the entire program execution and can be accessed by any thread for read/write. On the other hand, shared memory is on-chip, fast, but only visible by threads in a block for which it was declared. Another type of fast and globally accessible memory is the constant memory. It is limited to small size, 64 KB usually, and can only be used as a read-only memory optimized for temporal locality. Finally, there is another type of persistent read-only memory that utilizes the principle of spatial locality known as texture memory. Textures are suitable for non-coalescent access pattern. They are faster than global memory but slower than constant memory.

In order to achieve the best performance of GPUs, one should keep in mind the following guidelines:

1. Accessing the GPU global memory should be coalesced, i.e., consecutive threads access consecutive memory locations.

2. Avoiding to launch many kernels that perform simple tasks since data transfer and launch time may dominate any speedup gained.

3. Minimizing thread divergence in kernels by avoiding long branches. In the ideal scenario, threads in a warp execute the same path of instructions. With branches, threads satisfying the condition execute one path while the rest are delayed and evaluated in a second round.

4. Read-only data used heavily in kernels are better precomputed and stored in persistent constant, texture, or global memory on GPU. The choice of memory depends on data size and access pattern.

For a complete reference on CUDA programming, the reader is referred to [Nvi14, Wil13].

4 Mathematical Preliminaries

In this section, we review some essential concepts that we believe are key to understand the rest of the paper. We start by introducing the notations used throughout the paper. Next, we introduce different algebraic tools and representations that cater for efficient polynomial arithmetic.

[2]Currently, the number of threads in a warp is to 32.

4.1 Notation

We use capital letters to refer to sets and small letters for elements of a set. As usual, $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$, and $\mathbb{C}$ denote the integers, rationals, reals and complex numbers, respectively. We use $\lceil\cdot\rceil, \lfloor\cdot\rfloor, \lfloor\cdot\rceil$ to denote the round up, round down and round to nearest integer, respectively. We use $[\cdot]_q$ to denote the set of integers modulo q represented by the balanced set $\{\lceil -\frac{q}{2}\rceil, ..., \lfloor\frac{q-1}{2}\rfloor\}$, whereas $|\cdot|_q$ is represented by $\{0, \ldots, q-1\}$. We use $|\cdot|$ to denote the size in bits of a number or the cardinality of a set. R_q is the polynomial ring $\mathbb{Z}_q[x]/\langle x^n+1\rangle$, where n is a power of 2. Ring arithmetic is performed modulo both x^n+1 and q. Finally, uniform sampling of a from a set $\mathcal{S}$ is denoted as $a \overset{\$}{\leftarrow} \mathcal{S}$.

4.2 Residual Number System Representation of Polynomials

For secure settings and high multiplicative depth support[3], the ring $\mathbb{Z}_q[x]/\langle x^n+1\rangle$ contains long polynomials with large coefficients. In particular, n can be several thousand while $|q|$ can be several hundred of bits. Performing elementary arithmetic operations (addition and multiplication) with these polynomials can be done by multi-precision arithmetic. Although this approach is more intuitive and might be simpler to implement, it is not efficient especially with large parameters. Other modular techniques and algebraic tools can be used to achieve more efficient solutions. The basic idea is based on divide-and-conquer. A large computational problem is divided into several smaller and independent sub-problems which can be solved efficiently in parallel. The sub-solutions are then combined to form a final solution to the original problem. We show below how this can be applied to polynomial arithmetic.

The CRT/RNS can be used to decompose a polynomial $a(x)$ of large coefficients into a set of polynomials with smaller coefficients. First, we need to select a set of co-prime moduli, $\mathcal{M} = \{m_0, m_1, \ldots, m_{r-1}\}$. The number of moduli depends on two main factors: 1) m_i is chosen so that it fits in the underlying machine word and 2) $m = \prod_{i=0}^{r-1} m_i$ is greater than or equal to the coefficient modulus q. The map $a(x) \pmod{m} \mapsto (a(x) \pmod{m_0}, a(x) \pmod{m_1}, \ldots, a(x) \pmod{m_{r-1}})$ defines a ring isomorphism $\mathbb{Z}_m[x]/\langle x^n+1\rangle \cong \mathbb{Z}_{m_0}[x]/\langle x^n+1\rangle \times \mathbb{Z}_{m_1}[x]/\langle x^n+1\rangle \times \cdots \times \mathbb{Z}_{m_{r-1}}[x]/\langle x^n+1\rangle$. In other words, one can perform the computations in parallel in each ring $\mathbb{Z}_{m_i}[x]/\langle x^n+1\rangle$ and reconstruct the desired result in $\mathbb{Z}_m[x]/\langle x^n+1\rangle$ using a CRT reconstruction algorithm as follows. Let $\langle x_0, x_1, \ldots, x_{r-1}\rangle$ be the residues of integer x modulo $\mathcal{M}$, the integer x can be reconstructed via Equation 1.

$$x = \sum_{i=0}^{r-1} \frac{m}{m_i} \times \Big(\Big(\frac{m}{m_i}\Big)^{-1} x_i \pmod{m_i}\Big) \pmod{m} \tag{1}$$

A polynomial can be reconstructed from its residues by applying the CRT reconstruction on each coefficient residues. It can be clearly noticed that this operation is completely independent for each coefficient which makes it very suitable for GPU platforms. For further details on CRT and modular algorithms, the reader is referred to [Gar59, Knu97, VZGG13].

4.3 Number Theoretic Transform Representation of Polynomials

CRT representation can be used to add or subtract polynomials efficiently by adding/-subtracting polynomials residues coefficient-wise. Unfortunately, this is not the case with multiplication. In order to cater for efficient multiplication, polynomials can be represented by point-and-value instead of the canonical coefficient representation. This can be done by evaluating polynomials (of degree $< n$) on a set of points using a DFT-like transform known

[3]Multiplicative depth is defined as the maximal number of multiplications on any path in the circuit diagram.

as the Number Theoretic Transform (NTT) of length $2n$ [Pol71]. Point-wise multiplication of the NTT representation of two polynomials corresponds to the NTT representation of the product of the polynomials without reduction. A reduction algorithm modulo the ring polynomial modulus is still required. However, since we are working in $\mathbb{Z}_q[x]/\langle x^{2^k}+1\rangle$, the negative wrapped convolution can be used without reduction which can be computed efficiently via the Discrete Weighted Transform (DWT) [Win12]. Furthermore, with negative wrapped convolution, DWT transform of length n is only required.

For further improvement, one can use Crandall's Discrete Galois Transform (DGT) to compute an NTT equivalent representation, known as DGT representation in $GF(p^2)$, using $n/2$ transform data-path. Crandall initially proposed his algorithm assuming that p must be a Gaussian prime, i.e. $p \equiv 3 \pmod 4$ [Cra99]. However, in another work, we generalized his algorithm to work also with non-Gaussian primes, i.e. $p \equiv 1 \pmod 4$ [ABBA18]. We use our generalized DGT-based algorithm here to compute the DGT representation with $n/2$ FFT-like data-path. With our generalized algorithm, no further restrictions are imposed on the CRT moduli. For completeness, Algorithm 1 shows the DGT-based polynomial multiplication algorithm. For more details on its generalization and how to select its parameters, the reader is referred to [ABBA18].

Algorithm 1 Polynomial multiplication in R_{m_i} via the DGT

```
Let $R_{m_i}$ be the ring $\mathbb{Z}_{m_i}[x]/\langle x^n+1\rangle$ where $m_i$ is a prime and $n$ is a power of 2. Let $g$
be a primitive $n/2$-th root of unity in $GF(m_i^2)$ and $h \in \mathbb{Z}[i]$ be an $n/2$-th root of $i$ in
$GF(m_i^2)$. Let $a(x), b(x), c(x) \in R_{m_i}$ be polynomials each of degree less than $n$ with
integer coefficients $\{\lceil -\frac{m_i}{2} \rceil, ..., \lfloor \frac{m_i-1}{2} \rfloor\}$.

Input: $a(x), b(x), g, g^{-1}, h, h^{-1}, n, n^{-1}, m_i$
Output: $c(x) = a(x) \cdot b(x) \pmod{(m_i, x^n+1)}$

Precompute:
  $g^j, g^{-j}, h^j, h^{-j} \pmod{m_i}$, where $j = 0, \ldots, n/2-1$
Initialize:
  fold over input signals:
  $a'_j = a_j + i a_{j+n/2}$                                ▷ $i = \sqrt{-1}$
  $b'_j = b_j + i b_{j+n/2}$
Twist the folded signals:
  $\bar{a'}_j = a'_j h^j \pmod{m_i}$
  $\bar{b'}_j = b'_j h^j \pmod{m_i}$
Compute $n/2$ DGT:
  $A = DGT_g(\bar{a'})$
  $B = DGT_g(\bar{b'})$
Point-wise multiplication:
  $C_j = A_j \cdot B_j \pmod{m_i}$
Compute $n/2$ IDGT:
  $\bar{c'} = IDGT_{g^{-1}}(C)$
Remove twisting factors:
  $c'_j = \bar{c'}_j h^{-j} \pmod{m_i}$
Unfold output signal:
  $c_j = Re(c'_j)$                              ▷ $Re$ returns the real part
  $c_{j+n/2} = Im(c'_j)$                        ▷ $Im$ returns the imaginary part
return $c(x)$
```

Algorithm 1 shows that given two polynomials in DGT representation, their product can be computed in $\mathcal{O}(n)$ by coefficient-wise multiplication. We should remark here that every polynomial residue (in RNS/CRT domain) should be transformed to DGT before multiplication can start.

Note that DWT/DGT requires weighting the polynomials with some twisting factor h in Algorithm 1. These twisters should be distinguished from twiddle factors g used in the transform.

4.4 Limitations of Using the Ring $\mathbb{Z}[x]/\langle x^{2^k} + 1\rangle$

Although the polynomial ring used here offers fast ring multiplication, it suffers from a crucial problem related to plaintext batching. Plaintext batching, also known as CRT batching, is a common technique proposed by Brakerski et al. [BGH13] to reduce HE computational complexity. This technique enables packing several plaintexts in one ciphertext. Operations on the ciphertext affect the packed plaintexts in a SIMD manner. To highlight the limitation of using the ring $\mathbb{Z}[x]/\langle x^{2^k} + 1\rangle$, we need to provide a simple overview of plaintext batching.

As we shall see in subsequent sections, the plaintext space is a polynomial ring $R_t = \mathbb{Z}_t[x]/\langle x^{2^k} + 1\rangle$. For some (k, t), $x^{2^k} + 1$ can be factored modulo t into a set of mutually co-prime polynomials, say $f_i(x), 0 \le i < s$. Using the CRT, R_t can be partitioned by the following map:

$$R_t = \frac{\mathbb{Z}_t[x]}{x^{2^k} + 1} \cong \frac{\mathbb{Z}_t[x]}{f_0(x)} \times \cdots \times \frac{\mathbb{Z}_t[x]}{f_{s-1}(x)} \tag{2}$$

This means that we can encode s plaintext messages, apply CRT using the factors $f_i(x)$ as moduli, given that $f_i(x)$ are mutually co-prime, and produce a single polynomial that encodes s messages. The problem in R_t is that it only factors into a single factor considering $t = 2$. This is an important case since it allows handling Boolean gates XOR and AND using homomorphic addition and multiplication which makes constructing Boolean circuits very simple.

Nevertheless, the problem can be solved by choosing larger values of t at the expense that XOR becomes as costly as AND in noise growth. The reason for this can be illustrated by the following example:

Example 1 Let c_0, c_1 be encryptions of two ones. Suppose we want to evaluate XOR gate on c_0, c_1. Note that larger values of t means that polynomial coefficients can grow up to $t - 1$. Evaluating homomorphic addition on the inputs results in an encryption of 2. This can be corrected by subtracting $(2 \times c_0 \times c_1)$. Hence, the almost free homomorphic addition becomes as costly as homomorphic multiplication considering that multiplication by 2 can be done efficiently.

5 The Textbook FV and its RNS Variant

In this section, we review the textbook FV scheme and its RNS variant. We start by describing the plaintext and ciphertext spaces before delving into the main procedures of the schemes.

5.1 Plaintext and Ciphertext Spaces

The plaintext and ciphertext spaces are polynomial rings determined by the plaintext modulus t and ciphertext modulus q both $\in \mathbb{Z}$, where $2 \le t \ll q$. We refer to these rings by $R_t : \mathbb{Z}_t[x]/\langle x^n + 1\rangle$ and $R_q : \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. While plaintext is a single element $m \in R_t$, ciphertext is however, a pair of two elements $(ct[0], ct[1])$, where $ct[i] \in R_q$. The ratio $\Delta = \lfloor \frac{q}{t} \rfloor$ determines the multiplicative depth of the scheme. Choices of n, q affect the security level by defining the hardness of the underlying RLWE problem.

5.2 The Textbook FV Scheme

The textbook FV scheme described in [FV12] is a tuple of 5 procedures: key generation, encryption, decryption, homomorphic addition and homomorphic multiplication. In addition to plaintext and ciphertext spaces parameters, the scheme defines additional parameters described as follows:

- λ: a security parameter represented in unary notation.
- w: a decomposition base used to express a polynomial in R_q in terms of $l+1$ polynomials in base $w \in \mathbb{Z}$, where $l = \lfloor log_w^q \rfloor$.
- $\mathcal{X}_{err}$: a truncated zero-mean discrete Gaussian distribution used to sample error polynomials. The distribution is parameterized by the standard deviation σ and error bound β_{err}.

We review below the main procedures of the scheme.

- **KeyGen**(λ, w): The Secret Key (sk) is simply a binary polynomial $sk \overset{\$}{\leftarrow} R_2$. The Public Key (pk) is a pair of polynomials $(pk_0, pk_1) = (-[a \cdot sk + e]_q, a)$, where $a \overset{\$}{\leftarrow} R_q$ and $e \overset{\$}{\leftarrow} \mathcal{X}_{err}$. The Evaluation Key (evk) is a set of $(l+1)$ pairs of polynomials generated as follows: for $0 \leq i \leq l$, sample $a_i \overset{\$}{\leftarrow} R_q$ and $e_i \overset{\$}{\leftarrow} \mathcal{X}_{err}$. $evk[i] = ([w^i s^2 - (a_i \cdot sk + e_i)]_q, a_i)$. The procedure outputs: (sk, pk, evk).
- **Enc**(m, pk): encryption takes a plaintext message $m \in R_t$, samples $u \overset{\$}{\leftarrow} R_2$ and $e_1, e_2 \overset{\$}{\leftarrow} \mathcal{X}_{err}$. It produces the ciphertext $ct = ([\Delta m + pk_0 u + e_1]_q, [pk_1 u + e_2]_q)$.
- **Dec**(ct, sk): the procedure simply outputs $\left[\left\lfloor \frac{t}{q}[ct[0] + ct[1]sk]_q \right\rceil\right]_t$.
- **EvalAdd**(ct_0, ct_1): homomorphic addition takes two ciphertexts and produces: $ct_{add} = ([ct_0[0] + ct_1[0]]_q, [ct_0[1] + ct_1[1]]_q)$.
- **EvalMul**(ct_0, ct_1, evk): homomorphic multiplication takes two ciphertexts and computes

  1):
$$c_0 = \left[\left\lfloor \frac{t}{q} ct_0[0] ct_1[0] \right\rceil\right]_q$$
$$c_1 = \left[\left\lfloor \frac{t}{q} (ct_0[0] ct_1[1] + ct_0[1] ct_1[0]) \right\rceil\right]_q$$
$$c_2 = \left[\left\lfloor \frac{t}{q} ct_0[1] ct_1[1] \right\rceil\right]_q$$

  2) decomposes c_2 in base w as $c_2 = \sum_{i=0}^{l} c_2^{(i)} w^i$.

  3) returns ct_{mul}, such that:

$$ct_{mul}[0] = \left[c_0 + \sum_{i=0}^{l} evk[i][0] c_2^{(i)}\right]_q$$
$$ct_{mul}[1] = \left[c_1 + \sum_{i=0}^{l} evk[i][1] c_2^{(i)}\right]_q$$

It can be seen that all procedures incorporate arithmetic in polynomial ring. Polynomials are long and include large coefficients; hence either multi-precision or modular algorithms can be used to handle them. Modular algorithms offer better performance especially if they are executed in parallel [VZGG13]. One known modular algorithm is based on the CRT/RNS helps in decomposing long and large polynomials into a set of independent polynomials with small coefficients. Size of coefficients can be controlled by varying the size of CRT moduli. Usually, the CRT moduli size is chosen such that a modulus can fit in one machine word for efficient implementation.

The appealing property of the modular approach in handling polynomial arithmetic stems from the fact that the generated polynomials are independent and can be operated on in parallel making the problem suitable for parallel platforms. The only problem in CRT/RNS approach is that some operations become more difficult such as division-and-rounding and base decomposition. A close examination of the textbook FV shows that these operations are required for decryption and homomorphic multiplication. We should remark here that regarding division-and-rounding, the problem is not in division which can be converted to multiplication by a modular inverse, it is in rounding that requires comparison which is hardly compatible with RNS. As we mentioned in Section 4, polynomials are represented in two representations: 1) CRT/RNS and 2) NTT representations. After multiplication, polynomials are represented in NTT domain. It is not known if rounding is even possible to do in NTT representation which imposes a conversion to the CRT representation. Even in the CRT representation, rounding is not cheap and requires a complex implementation [OP07]. Hence, another conversion to the canonical coefficient representation of polynomials is required to perform multi-precision comparison. The same argument applies to base decomposition.

Fortunately, Bajard et al. [BEHZ16] proposed a clever way to perform approximate rounding in the RNS domain by utilizing the floor function. Since FV is a noisy/error based encryption scheme, introducing extra errors may not violate its functionality as long as the error is controlled. They adapted the textbook Dec and EvalMul to perform approximate rounding in CRT/RNS domain. Moreover, they avoided base decomposition by decomposing polynomials in the CRT moduli. The modified Dec and EvalMul procedures of Bajard et al. are more efficient and do not affect much the maximum multiplication depth supported by the scheme. In the following section, we review their procedures and introduce how to map them to GPU.

A practical note The relinearization procedure (steps 2 and 3 in EvalMul) is crucial to reduce ciphertext length and error growth during homomorphic evaluations. Furthermore, computing the product c_1 in step 1 can be performed via Karatsuba multiplication algorithm to reduce the number of multiplications from 4 to 3.

5.3 The RNS Variant of FV

The core component of the proposed RNS FV of Bajard et al. is an efficient method to convert between RNS bases called fast base conversion (FastBconv) provided in Equation 3. FastBconv converts an integer x represented in base q with r moduli to RNS representation in base $\mathcal{B}$. It can also be used for a polynomial where the operation is applied to each coefficient. FastBconv is similar to the CRT reconstruction except that no modular reduction modulo m is required. This means the conversion is approximate and introduces errors that can be removed by some correction techniques such as Shenoy-Kumaresan algorithm through a redundant modulus [SK89].

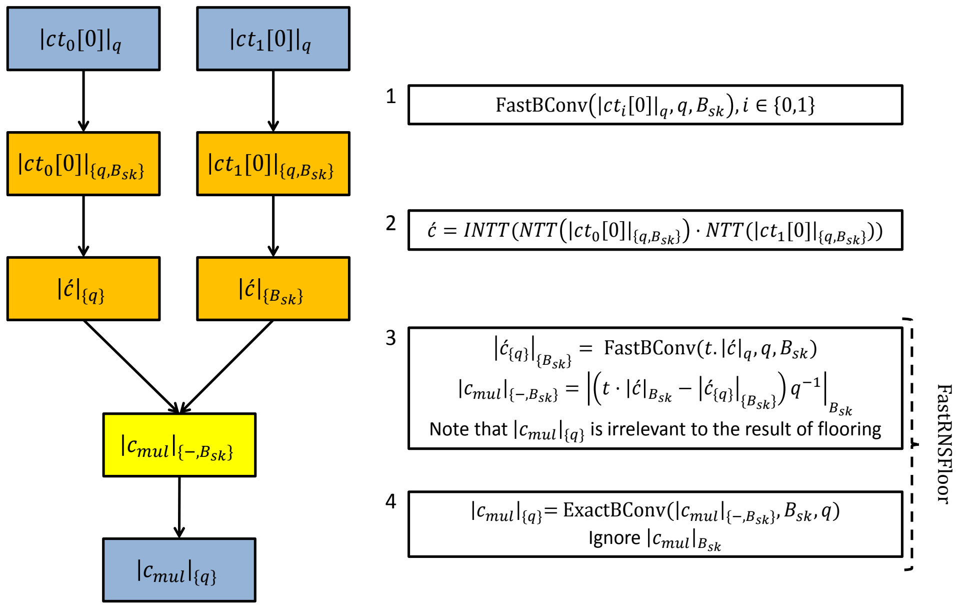
$$\text{FastBconv}(x, q, \mathcal{B}) = \Big(\sum_{i=1}^{r} \Big|x_i \frac{q_i}{q}\Big|_{q_i} \times \frac{q}{q_i} \pmod{b}\Big)_{b \in \mathcal{B}} \tag{3}$$

Another important aspect of the RNS FV is the replacement of round by floor. Rounding requires comparison which is hardly compatible with RNS, instead flooring can be done efficiently if the divisor is a product of a subset of the RNS moduli. For decryption, they use the basic definition of the floor function as in Equation 4 to calculate an approximate value of $\left[\left\lfloor \frac{t}{q}[ct[0] + ct[1]sk]_q \right\rceil\right]_t$. The redundant modulus γ is chosen co-prime to q and greater than or equal to r the number of CRT moduli in q. To correct the error due to fast base conversion and the usage of flooring instead of rounding, they apply Shenoy-Kumaresan algorithm. The new decryption procedure is shown in Algorithm 2.

$$\left\lfloor \frac{a}{q} \right\rfloor = \frac{a - (a \bmod q)}{q} \tag{4}$$

Algorithm 2 Dec$_{\text{RNS}}$ [BEHZ16]

```
Input: ciphertext $ct$, secret key $sk$ and a redundant modulus $\gamma \in \mathbb{Z}$.
Output: the plaintext message $[m]_t$.
for $j \in \{t, \gamma\}$ do
    $s^{(j)} \leftarrow \text{FastBconv}(|\gamma t \cdot (ct[0] + ct[1]sk)|_q, q, j) \times | - q^{-1}|_j \pmod j$
$\tilde{s}^{(\gamma)} \leftarrow [s^{(\gamma)}]_\gamma$
$m^{(t)} \leftarrow [(s^{(t)} - \tilde{s}^{(\gamma)}) \times |\gamma^{-1}|_t]_t$
return $m^{(t)}$
```



$|ct_0[0]|_q$
$|ct_1[0]|_q$
$|ct_0[0]|_{\{q,B_{sk}\}}$
$|ct_1[0]|_{\{q,B_{sk}\}}$
$|\acute{c}|_{\{q\}}$
$|\acute{c}|_{\{B_{sk}\}}$
$|c_{mul}|_{\{-,B_{sk}\}}$
$|c_{mul}|_{\{q\}}$
1
$\text{FastBConv}(|ct_i[0]|_q, q, B_{sk}), i \in \{0,1\}$
2
$\acute{c} = INTT(NTT(|ct_0[0]|_{\{q,B_{sk}\}}) \cdot NTT(|ct_1[0]|_{\{q,B_{sk}\}}))$
3
$\left|\acute{c}_{\{q\}}\right|_{\{B_{sk}\}} = \text{FastBConv}(t. |\acute{c}|_q, q, B_{sk})$
$|c_{mul}|_{\{-,B_{sk}\}} = \left|\left(t \cdot |\acute{c}|_{B_{sk}} - \left|\acute{c}_{\{q\}}\right|_{\{B_{sk}\}}\right) q^{-1}\right|_{B_{sk}}$
Note that $|c_{mul}|_{\{q\}}$ is irrelevant to the result of flooring
4
$|c_{mul}|_{\{q\}} = \text{ExactBConv}(|c_{mul}|_{\{-,B_{sk}\}}, B_{sk}, q)$
Ignore $|c_{mul}|_{B_{sk}}$
FastRNSFloor

Figure 1: Flow diagram of computing divide-and-round in RNS FV

On the other hand, homomorphic multiplication is more involved and carried out in four steps:

1. Extending polynomials in larger RNS moduli to contain polynomial product in $\mathbb{Z}[x]/\langle x^n + 1\rangle$, i.e. no modular reduction is applied to the coefficients.

2. Multiplying the extended polynomials in the NTT domain.

3. Performing approximate rounding by flooring.

4. Relinearization by decomposing c_2 in the RNS moduli of q instead of w.

Figure 1 shows how to carry out steps 1-3 to compute $c_0 = \left[\left\lfloor \frac{t}{q} ct_0[0] ct_1[0] \right\rceil\right]_q$. Initially, we have the RNS representation in base q of polynomials: $ct_0[0]$ and $ct_1[0]$. Each polynomial is extended modulo an auxiliary moduli $\mathcal{B}_{sk} = \mathcal{B} \cup m_{sk}$. The moduli in $\mathcal{B}$ are chosen co-prime to q such that $\prod_{i=1}^{l} m_i \geq 4(q-1)^2 n \cdot t, \forall m_i \in \{q \cup \mathcal{B}\}$. This is to contain the products c_0, c_1 and c_2 as if they were calculated in $\mathbb{Z}[x]/\langle x^n + 1 \rangle$, i.e. without a reduction modulo q. The redundant modulus m_{sk} must be co-prime to $\{q \cup \mathcal{B}\}$ and greater than $2(|\mathcal{B}| + \lceil \tau \rceil)$, where τ is the largest multiple of $\mathcal{B}$ the polynomial product may contain at any point of computation. It should be remarked that as we do fast conversion from base q to $\mathcal{B}_{sk}$, overflows of q may be generated and hence an extra moduli $\tilde{m} > |q|$ is used to remove these overflows using Shenoy-Kumaresan algorithm. Next, we multiply the extended polynomials in NTT domain. The product is converted back to the RNS base $\{q \cup \mathcal{B}_{sk}\}$. In step 3, flooring is carried out via division by q. Although division is hardly compatible with RNS, dividing by product of a subset of the moduli, also known as scaling, is much simpler. Bajard et al. used Barsi and Pinotti [BP95] scaling algorithm. While computing the floor function, the polynomials are multiplied by t as shown in step 3 in the figure. At this point, we have the floored polynomial in base $\mathcal{B}_{sk}$. Finally, we transform the product back to base q using Shenoy-Kumaresan algorithm and the redundant moduli m_{sk}. The resultant product contains an extra error e where $||e||_\infty \leq |q|$.

To bypass base decomposition in EvalMul, they adapt the relinearization step as follows:
1) $\xi_q(c_2) = \sum_{i=1}^{k} \left| c_2 \frac{q_i}{q} \right|_{q_i}$.
2) $\mathcal{P}_{\text{RNS},q}(sk^2) = \left(\left| sk^2 \frac{q}{q_1} \right|_q, \ldots, \left| sk^2 \frac{q}{q_k} \right|_q \right)$.
3) replace evk by $evk_{\text{RNS}}[i] = ([\mathcal{P}_{\text{RNS},q}(sk^2)[i] - (e_i + a_i \cdot sk)]_q, a_i)$
For completeness, the RNS EvalMul procedure is listed in Algorithm 3.

Algorithm 3 EvalMul$_{\text{RNS}}$ [BEHZ16]

```
Input: ciphertexts $ct_0, ct_1$, evaluation key $evk_{\text{RNS}}$ and two redundant moduli $m_{sk}, \tilde{m} \in \mathbb{Z}$.
Output: $ct_{mul}$ in $q$.
  S0: Convert fast $ct_0$ and $ct_1$ from $q$ to $\left\{\mathcal{B}_{sk} = (\{\mathcal{B} \cup m_{sk}\}) \cup \{\tilde{m}\}\right\}$
  S1: Convert $ct_0$ and $ct_1$ from $\mathcal{B}_{sk} \cup \tilde{m}$ to $\mathcal{B}_{sk}$ via Small Montgomery algorithm[4] to get $ct'_0$ and $ct'_1$.
  S2: Compute the product $ct'_\star = ct'_0 \times ct'_1$ in $q \cup \mathcal{B}_{sk}$.    ▷ $\times$ is ring multiplication
  S3: Convert fast $ct'_\star$ from $q$ to $\mathcal{B}_{sk}$ to get $\tilde{ct}_{mul} + e$ in $\mathcal{B}_{sk}$.
  S4: Convert from $\mathcal{B}_{sk}$ to $q$ to get $\tilde{ct}_{mul} + e$ in $q$.
  S5: Perform the adapted relinearization step to get $ct_{mul}$ in $q$.
```

To highlight the significant improvement in performance of the RNS variant over the textbook FV, Figure 2 shows the latency of decryption and homomorphic multiplication using SEAL (v2.1) which implements the textbook FV and SEAL (v2.3.0-4) which implements the RNS variant.

In the following section, we describe our GPU implementation of the RNS FV and an array of optimizations to achieve optimum performance on GPUs.

[4]An algorithm used to represent the polynomial coefficients in Montgomery representation with respect to $\tilde{m}$.

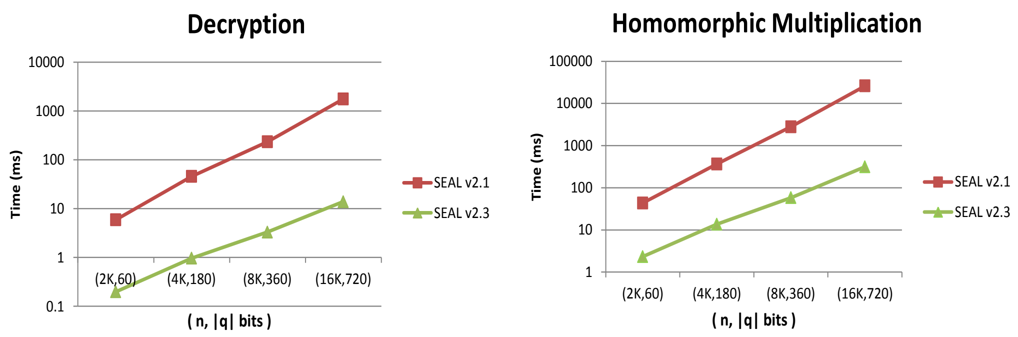


Figure 2: Performance of decryption and homomorphic multiplication in the textbook FV (implemented in SEAL (v2.1)) and its RNS variant (implemented by SEAL (v2.3.0-4)) averaged over 500 runs. Both versions were run on Intel(R) Xeon(R) E5-2620 at 2.40 GHz with 24 cores equipped with ArchLinux (4.8.13-1-ARCH). Note that the vertical axes in the figures are in log scale.

6 Implementation Layout

Our GPU implementation is highly inspired by cuHE, a CUDA based library for polynomial arithmetic [DS15]. cuHE was designed specifically to perform polynomial arithmetic in arbitrary rings on CUDA GPUs. The library utilizes NTL for CPU interface and employs the CRT, NTT and polynomial Barrett reduction. cuHE supports natively only three polynomial degrees: $\{8192, 16384, 32768\}$, i.e. polynomials with different degrees are padded with zeros to a higher supported degree. As the library deals with arbitrary polynomial rings, polynomial length is doubled and zero-padded before computing the NTT. After multiplication, Barrett reduction is used to reduce the product in the ring. Below we describe our implementation pointing out similarities and differences with cuHE.

6.1 Polynomial Representation

Polynomials are initialized using NTL on CPU. We use NTL mainly for initializing the encryption keys. Eventually, all the keys are converted to CRT and NTT domain and stored in GPU global memory.

On GPU, we store polynomials, represented in double CRT, in a Structure of Arrays (SOA) layout. This allows for coalescent access pattern allowing each thread to deal with one polynomial coefficient/residue[5] at a time.

6.2 The Chinese Remainder Theorem and its Reconstruction

The CRT operation takes a polynomial with large multi-word coefficients and produces a set of polynomial residues with smaller coefficients that can fit in one machine word. CRT reconstruction is the inverse of CRT, it applies the Chinese remainder theorem and reconstructs a polynomial from its residues.

The word size in our CUDA cards is 32 bit. The ideal solution in this scenario is to choose 32-bit CRT moduli to minimize the number of residues and to avoid using multi-precision arithmetic. Note that 2-word moduli can be used if the hardware/software

[5]A thread may process one whole coefficient in $\mathbb{Z}_q$ or a smaller residue of it. For instance, point-wise addition/multiplication is done at the residue level where each thread adds/multiplies two residues. In contrast, the CRT reconstruction is processed at the coefficient level such that each thread processes one coefficient at a time.

supports efficient 4-word arithmetic. Another factor that affects the choice of CRT moduli is the efficiency of modulo operation. A carefully chosen moduli offer more efficient modulo operation than arbitrary moduli. For instance, NFLlib [AMBG+16] uses a carefully chosen CRT moduli of size 30 bits and 62 bits to perform efficient modulo using lazy modular reduction procedure. After investigating NFLlib's approach, we found out that NFLlib's reduction algorithm is not ideally suited for GPUs. We performed a simple experiment to apply modulo reduction of N 62-bit integers modulo one of NFLlib 30-bit primes using NFLlib's reduction and the native modulo operation (%) in CUDA. The native modulo outperformed NFLlib's algorithm by a slight margin as illustrated in Table 1.

Table 1: Latency in (millisecond) of reducing N 62-bit integers modulo a 30-bit NFLlib prime $p = 984481793$ on Intel(R) Xeon(R) E5-2620 at 2.40 GHz with 24 cores and NVIDIA Tesla K80 card averaged over 10000 runs.

N	**CPU**			**GPU**		
	%	[AMBG+16] Algorithm 2	Speedup	%	[AMBG+16] Algorithm 2	Speedup
2^{11}	0.025	**0.007**	3.57x	**0.0089**	0.0093	0.96x
2^{12}	0.052	**0.014**	3.71x	**0.0091**	0.0094	0.97x
2^{13}	0.108	**0.031**	3.48x	**0.0097**	0.0099	0.98x
2^{14}	0.219	**0.054**	4.06x	**0.0100**	0.0102	0.98x

Although NFLlib reduction algorithm is very efficient on CPU, it does not perform well on GPU. This might be attributed to two reasons: 1) bit operations required in NFLlib's algorithm seem to be less efficient on CUDA and 2) thread divergence (or predicated execution overhead) due to a condition at the end of the algorithm. Hence, we use 32-bit primes to form our CRT moduli and perform modulo operation using the native CUDA modulo operation. In fact, using 32-bit primes over 30-bit primes may lead to reduced number of moduli to represent q and consequently less number of NTT procedures.

6.2.1 CRT

We launch n threads to reduce a polynomial of degree $< n$ modulo the CRT moduli. Each thread reduces a multi-precision integer modulo a single-precision CRT modulus. Let x be a d-word integer. Decomposing x in base 2^{32} results in $x_{d-1}(2^{32})^{d-1} + x_{d-2}(2^{32})^{d-2} + \cdots + x_0(2^{32})^0$, where x_i is a 32-bit number. Reducing x modulo, say m_j, can be applied to each term independently. Since CUDA supports double-precision[6] operations, we reduce each term of x independently and sum up the remainders modulo m_j. To obtain a more efficient implementation, one may choose to precompute and store the constants $(2^{32})^i \pmod{m_j}$ in GPU constant memory.

6.2.2 CRT-reconstruct

The CRT reconstruction can be performed by two methods: 1) using the Chinese remainder theorem as shown in Equation 1 and 2) using a mixed-radix system. Both methods can be implemented on GPUs, however we argue that the second method is more suited for GPU platforms. In order to prove this point, we need first to introduce Garner's algorithm that employs a mixed-radix system [Gar59]. We remark here that cuHE implements the classic CRT reconstruction.

Algorithm 4 shows two main steps: 1) precomputation of the constants C_i's and 2) reconstruction. Since we are dealing with fixed moduli $\mathcal{M}$, precomputation is done at

[6]In our scenario, double-precision operations refer to 64-bit operations.

program initialization and the constants C_i's are stored in GPU constant memory. Storage requirement for C_i's is $O(r)$, where r is the number of CRT moduli. The reconstruction step is however more involved and must be applied to each coefficient. The required operations for this step are as follows: 1) 2 multi-precision subtraction/addition modulo a single-precision integer m_i and 2) 1 multiplication of a multi-precision integer ($\prod_{j=0}^{i-1} m_j$) by a single-precision integer u. We perform these operations via CUDA PTX language to obtain access to the carry bits. Another optimization used here is storing the constants $K_i = \prod_{j=0}^{i-1} m_j$ in GPU constant memory. The product K_i requires i words of storage. Assuming the maximum number of CRT primes is z, we allocate constant memory of size $(z-1) \cdot z/2$. Figure 3 illustrates the storage and indexing of K_i's. We remark here that we use z instead of r as the constant memory size should be known at compile time.

Algorithm 4 Garner's CRT-reconstruct using mixed-radix system

```
Let $\mathcal{M} = \{m_0, m_1, \ldots, m_{r-1}\}$ denote the set of $r$ co-prime CRT moduli and $m = \prod_{i=0}^{r-1} m_i$.
**Input**: The RNS representation of $x \in \mathbb{Z}$: $RNS(x) = (x_0, x_1, \ldots, x_{r-1})$ modulo $\mathcal{M}$
**Output**: The unique integer $x$, such that $x < m$

**Precomputation**:
**for** $i = 1;\ i < r\ ;\ i{+}{+}$ **do**
    $C_i = 1$
    **for** $j = 0;\ j < i\ ;\ j{+}{+}$ **do**
        $t = m_j^{-1} \pmod{m_i}$
        $C_i = t \cdot C_i \pmod{m_i}$
**Reconstruction**:
  $u = x_0$
  $x = t$
**for** $i = 1;\ i < r\ ;\ i{+}{+}$ **do**
    $u = (x_i - x) \cdot C_i \pmod{m_i}$
    $x = x + u \cdot \prod_{j=0}^{i-1} m_j$
**return** $x$
```

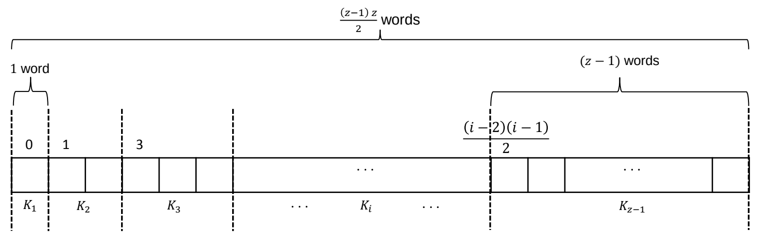


Figure 3: Memory layout of CRT products used in Garner's CRT-reconstruct algorithm

Efficiency of Garner's Algorithm on GPU Garner's CRT-reconstruct is more suited for GPU compared to classic CRT-reconstruct using Equation 1. First, classic CRT-reconstruct requires the reduction of multi-precision integer (the sum) modulo another multi-precision integer m. This operation is not only costly, but also requires a complex implementation. One may avoid this costly reduction by exploiting the fact that in each iteration, the sum is less than $2m$, i.e., reduction can be substituted by a check whether the sum is greater than or equal to m and subtracting m if the condition is true. Although this approach is less costly, it is not ideally suited for GPU platforms as the conditional check may create divergence in threads execution which in turn leads to sequential execution in a thread warp. In addition, the storage requirements for the constants used in the classic CRT-reconstruct is higher than that in Garner's algorithm. In classic CRT-reconstruct, one has to store r values of multi-precision numbers m/m_i and r of single-precision numbers, whereas in Garner's algorithm storage requirement is $(r(r-1))/2$ and r of single-precision numbers.

To back up our argument, we performed an experiment to reconstruct the coefficients of a randomly generated polynomial of fixed degree $< (n = 8192)$ with a predefined coefficient size. Table 2 shows the latency of three CRT-reconstruct algorithms: the classic algorithm employing Multi-Precision (MP) modulo m using Algorithm 14.20 in [MVOV96], enhanced classic with conditional MP subtraction using cuHE CRT-reconstruct and Garner's algorithm. It should be remarked that all algorithms were implemented in CUDA. In addition, the reported time does not include memory transfer. Pseudo-code for the implemented Garner's algorithm can be found in appendix A.1. For better performance, our pseudo-code stores negatives of the constants C_i.

Table 2: Latency in (millisecond) of reconstructing a polynomial of degree $< (n = 8192)$ with coefficients of size log_2^q on Tesla K80 NVIDIA GPU card averaged over 10000 runs.

$\log_2^q$	**Classic** MP-mod	**Classic** MP-sub	**Garner's** Algorithm
62	0.4142	0.0312	**0.0153**
186	0.8057	0.0624	**0.0506**
372	1.7040	0.1638	**0.1299**
744	3.8620	0.5660	**0.4252**

6.3 The Discrete Galois Transform and its Inverse

DGT and its inverse are used to compute the NTT representation of polynomials. As we mentioned previously, this transform allows computing the NTT (or more precisely the DGT) representation using $n/2$ FFT-like data-path. Thus, we only need to store $n/2$ twiddle roots. On the other hand, twisting factors are Gaussian integers each requires two integers for real and imaginary parts. We store twiddles in texture memory while the twisting factors in global memory due to different memory access patterns.

DGT and IDGT are computed via multi-radix Stockham algorithm. We adapted the algorithm to work in $GF(m_i^2)$. Details and pseudo-code of the algorithm can be found in [GLD⁺08, amd]. We should remark that cuHE computes the NTT representation using a single special 64-bit Solinas prime $\hat{p} = 2^{64} - 2^{32} + 1$ that provides efficient 128-bit integer reduction. In fact, this prime was originally used by Emmart and Weems [EW11] in their large integer GPU multiplier. Computing NTT in one prime has the following advantages: 1) one can optimize modulo operations for one modulus and 2) precomputed twiddles and twisting factors for a single prime are only required. However, this approach suffers from a significant drawback. The CRT moduli m_i must be $\leq \lfloor|\sqrt{\hat{p}/(2n)}|\rfloor$ so that the maximum convolution value is supported modulo $\hat{p}$. This bound covers only a single multiplication of two polynomials, if more than two polynomials are to be multiplied in the NTT domain, the bound will be further smaller. Smaller CRT moduli size increases the number of NTT transforms which deteriorates the performance.

For small polynomial degrees ($\leq 4k$) we launch a single kernel for computing DGT/IDGT. The entire computation is done by a single block which enables us to synchronize all the threads and process the polynomial in several passes. For larger degrees, 2-3 kernels might be required instead since several blocks are launched together. Global synchronization between threads is achieved by launching successive kernels on the same stream. We should remark that we perform the DGT/IDGT computation on all polynomial residues at the same time. We pack all polynomial residues together and launch the transform kernel only once. This reduces the total number of expensive kernel launches. Packing is simply done by launching 2D grid of thread blocks. Grid dimensions are x and y, where x is the number of thread blocks required to compute DGT/IDGT for a single residue and $y = r$ the number of primes in the CRT moduli.

In cuHE, NTT/INTT computation is carried out using 3 kernels hard-coded for each polynomial degree without batching. We noticed that kernel launch time takes more than kernel computation. By reducing the number of kernel launches, noticeable improvement can be achieved. In addition, one NTT/INTT conversion can be done at a time in cuHE. This is due to the usage of a single shared buffer as temporary storage for NTT computation. We provide a separate buffer for each DGT/IDGT launch allowing for concurrent DGT/IDGT invocations.

6.4 Fast Base Conversion

A close examination of fast base conversion (Equation 3) shows the similarity between it and the CRT reconstruction (Equation 1). However, since the summation is calculated modulo $b \in \mathcal{B}$, which is a single word modulus, the constants $\frac{q}{q_i} \pmod{b}$ can be pre-computed and stored in the constant memory. Storage requirement for them are $\mathcal{O}(z \cdot v)$ words, where z and v are the maximum number of moduli chosen for the bases q and $\mathcal{B}$. Similarly, the constants $\frac{q_i}{q} \pmod{q_i}$ are precomputed and stored in the constant memory requiring $\mathcal{O}(z)$ words.

On more constrained devices, one may choose to calculate the conversion without pre-computed constants similar to CRT-reconstruct by reconstructing the number from its residues followed by multi-precision modulo b. The only difference is that a multi-precision subtraction of the dynamic range is not required. This approach however requires costly multi-precision multiplication.

6.5 Random Number Generation

Random polynomials required for key generation and encryption are generated on GPU by means of CUDA cuRAND. Three distributions are identified: 1) $\mathcal{X}_2$, 2) $\mathcal{X}_q$, and 3) $\mathcal{X}_{err}$. The first two are uniform distributions while the third is discrete truncated Gaussian.

To sample a random polynomial from $\mathcal{X}_2$, we launch n threads. Each thread samples a number from cuRAND uniform random number generator modulo 2. For $\mathcal{X}_q$, each thread generates r random numbers and construct a random coefficient in CRT representation. Lastly, For $\mathcal{X}_{err}$ we launch $n/2$ threads, each thread uses its own uniform random number to generate a pair of independent standard, normally distributed random numbers using Box-Muller method [BM$^+$58].

6.6 Multiple CUDA Streams

In order to fully utilize the GPU and increase throughput, we launch independent kernels on multiple streams. This allows for concurrent and interleaved execution of independent operations. A clear example on this is the homomorphic multiplication procedure. Computing c_0, c_1, and c_2 are independent and can be done simultaneously. We also hide the latency of memory transfer behind kernel execution by using pinned memory allocation.

6.7 Memory Pool

Dealing with several representations of polynomial requires allocating and deallocating a large amount of memory during computation. In addition, the evaluation of homomorphic multiplication requires a large amount of temporary storage for base conversion. In order to reduce the cost of that, we employ a memory pool mechanism to pre-allocate memory on GPU at system initialization.

7 Experiments and Results

In this section, we compare our implementation with two other implementations. First, we describe our comparison methodology, the test environment and hardware configuration for both CPU and GPU. Next, we analyze the results and point out the pros and cons of each implementation.

7.1 Methodology

To evaluate the performance of our implementation, we compare it with Microsoft SEAL-FV$_{\text{RNS}}$ (v2.3.0-4) [KLP16] and NFLlib-FV[7] [AMBG$^+$16]. To highlight the efficiency of RNS techniques, we include timings of a GPU implementation that performs divide-and-round and base decomposition in coefficient representation on CPU via NTL version (10.5.0). We refer to these implementations as: GPU-FV$_{\text{RNS}}$, SEAL-FV$_{\text{RNS}}$, NFLlib-FV and G-CPU-FV$_{\text{K80}}$.

We report the running time of the FV primitives: key generation, encryption, decryption, homomorphic addition and homomorphic multiplication. Any initialization operation that would only be done once is not included in our measurements. Computing powers of roots, library objects initialization and creation were not included in the timing figures. On CPU, we measure the time via C++ library chrono [cpp]. On the other hand, CUDA events were used to record and measure kernels execution time. Each experiment was performed 500 times and the average execution time is always reported.

7.2 Testbed Environment

We developed our implementation via CUDA 8.0 on a lightly loaded 64-bit machine equipped with two 6-core CPUs (with hyper-threading enabled), NVIDIA Tesla K80 GPU of 3.7 compute capability and NVIDIA Tesla P100 of 6.0 compute capability. The operating system is ArchLinux (4.14.6-1-ARCH). The compilers used are GCC (7.2.1 20171128) and NVCC (8.0.61). We run NFLlib-FV and SEAL-FV$_{\text{RNS}}$ on the same machine enabling all CPU cores which also support AVX2 instructions. This means that NFLlib-FV can utilize 24 cores. Table 3 describes the hardware configuration of both CPU and GPU. We should remark that we only utilize one GPU in all our experiments. On the other hand, all CPU cores were enabled and made ready for use by SEAL-FV$_{\text{RNS}}$ and NFLlib-FV.

7.3 Choice of Parameters

The FV scheme is parametrized by the following parameters: n, q, t, σ and, β_{err}. There are other parameters related to RNS techniques such as $\gamma, m_{sk}, \tilde{m}$. In choosing FV's parameters, we followed Lepoint and Naehrig's security analysis [LN14]. For γ, m_{sk} and $\tilde{m}$, we consider the recommendations of Bajard et al. (see Table 1 in [BEHZ16]). Table 4 shows the range of values used in experiments and the maximum multiplicative depth L supported by the implementations. Note that these settings provide 80-bit security level.

[7]No version could be found for NFLlib-FV. The source code used had been cloned on Dec 30th 2017.

Table 3: Testbed environment hardware configurations.

Feature	**CPU**	**GPU**	
		K80	P100
Model	Intel(R) Xeon(R) E5-2620	K80	P100
# Cores	24	2496	3584
Frequency	2.40 GHz	0.82 GHz	1.328 GHz
RAM	62 GB	12 GB	16 GB

For NFLlib-FV, we choose the size of small CRT parameters as 62 bit. This actually gives better results compared to 30-bit moduli as shown in Table 5 in [BEHZ16]. For SEAL-FV$_{\mathrm{RNS}}$ (v2.3.0-4), the user may choose a combination of various sizes (30, 40, 50 and 60 bits) as small CRT moduli. We choose 60-bit moduli as it is the closest to NFLlib-FV and our settings. It should be remarked that SEAL-FV$_{\mathrm{RNS}}$ also provides a set of recommended settings for two levels of security: 128-bit and 192-bit. The user can use the "ChooserEvaluator" to select a set of parameters. For more details on SEAL-FV$_{\mathrm{RNS}}$'s default and recommended parameters, the reader is referred to Table 3 in [KLP16].

Note that the choice of $\tilde{m}$ impacts the noise growth in homomorphic multiplication. A lower value of $\tilde{m}$ increases noise growth. In fact, one may choose $\tilde{m} = 0$ and skip (S1) in Algorithm 3. Although that improves the performance, it lowers the maximum multiplicative depth supported.

To avoid the complexity of calculating RNS parameters, we choose constant values for $\gamma, \tilde{m}$ and m_{sk} as shown in Table 4.

Table 4: Parameters for NFLlib-FV, GPU-FV$_{\mathrm{RNS}}$, G-CPU-FV$_{\mathrm{K80}}$ and SEAL-FV$_{\mathrm{RNS}}$ (v2.3.0-4).

n	$\lceil log_2^q \rceil$		t	σ	β_{err}	γ	$\tilde{m}$	m_{sk}	L
	SEAL	rest							
2^{11}	60	62	2	8	48	7	(no need)	4211212259	2
2^{12}	180	186	2	8	48	13	(no need)	4211212259	5
2^{13}	360	372	2	8	48	37	2^{16}	4211212259	13
2^{14}	720	744	2	8	48	53	2^{16}	4211212259	25

7.4 Timing Results

Table 5 lists the timing and speedup gains for the four implementations. Speedup is computed with reference to P100 except for G-CPU-FV$_{\mathrm{K80}}$ whose speedup is calculated with reference to K80. It can be noticed that GPU-FV$_{\mathrm{RNS}}$ performance on P100 compares favorably against the rest. On average, it achieves 5.37x, 7.37x, 22.22x, 5.11x and 13.18x (resp. 2.03x, 2.94x, 27.86x, 8.53x and 18.69x) for key generation, encryption, decryption, homomorphic addition and homomorphic multiplication against SEAL-FV$_{\mathrm{RNS}}$ (resp. NFLlib-FV). On K80, GPU-FV$_{\mathrm{RNS}}$ still outperforms SEAL-FV$_{\mathrm{RNS}}$ and NFLlib-FV but with less speedup gains.

A common feature in all primitives is as parameter sizes increase, differences in performance become more evident. On the other hand, less improvement is noticed with small parameters. As can be seen in Figure 4, GPU utilization rate increases as problem grows in size. When the amount of offloaded work is small, data transfer time between host and device and kernel launches overhead dominates computation time. Another noticeable feature is that amongst the five procedures, key generation and encryption have the lowest speedup when compared to NFLlib-FV, this is due to the expensive uniform and Gaussian sampling required for generating error polynomials. Obviously, NFLlib random generation algorithm is more efficient (in small settings) than our simple Box-Muller based random generator.

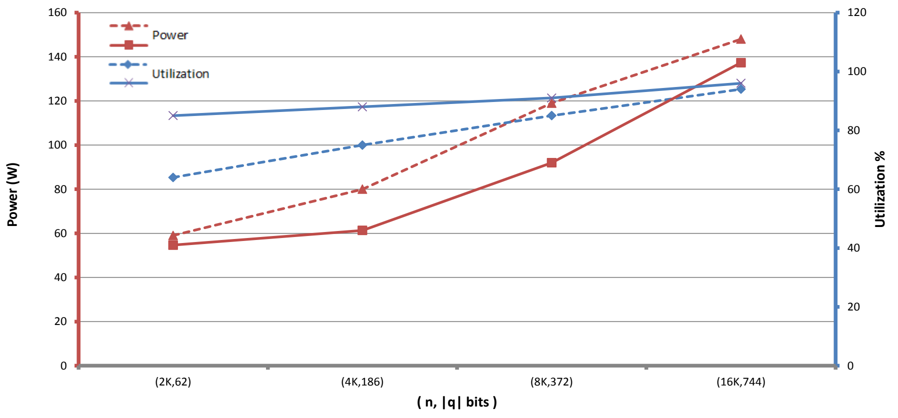


Figure 4: P100 (solid) and K80 (dashed) power consumption (W) and utilization rate for homomorphic multiplication of polynomials of degree $< n$ with coefficients size $|q|$ bits averaged over 10000 samples.

Focusing on SEAL-FV$_{\mathrm{RNS}}$ and NFLlib-FV, although NFLlib-FV is a multi-core library employing AVX2 and SSE instructions, it only outperforms SEAL-FV$_{\mathrm{RNS}}$ (which only employs software based multi-threading) in key generation and encryption. In decryption and homomorphic addition, SEAL-FV$_{\mathrm{RNS}}$ outperforms NFLlib-FV by a slight margin. The gap is more evident with homomorphic multiplication since SEAL-FV$_{\mathrm{RNS}}$ employs Bajard's et al. RNS FV. It would be interesting to find out the performance of NFLlib-FV with the RNS techniques adopted[8].

RNS variant performance can also be noticed from GPU-FV$_{\mathrm{RNS}}$ and G-CPU-FV$_{\mathrm{K80}}$ results. Performing divide-and-round and base decomposition on coefficient representation is certainly not a desired approach especially for homomorphic multiplication.

A question that may rise is how much speedup we can get from GPU and whether we were able to reach that. Although there seems no simple and accurate model to evaluate GPU performance, some approximated results from the literature can help. A team of Intel corporation provided a thorough benchmark study to evaluate the performance of NVIDIA GTX280 and Intel Core i7 960 processors [LKC^{+}10] using 14 compute/memory-bound workloads. They found that the gap between the two processors narrows down to only 2.5x (in favor of GPU) when software optimizations are applied appropriately to CPU and GPU codes. Although the processors used in our experiments are not the same, one can infer that GPU would not provide massive improvement compared to CPU. In our experiments, we could gain speedups from 0.32x to 60.95x against highly optimized CPU implementations on a single GPU card.

We present one more experiment to provide insights into hardware utilization and power consumption. In this experiment, only homomorphic multiplication (the most performance-critical operation) is performed. Unsurprisingly, utilization rate increases as polynomial degree and coefficient size increase. Although no changes were made on

[8]The reader should note that we could only run NFLlib-FV for $(n, |q|) = (2^{14}, 744)$ after increasing the system stack size. With default stack size (8 MB) the library crashes on multiplication since it allocates large arrays on the stack. The problem is resolved after increasing the stack size to (64 MB).

Table 5: Latency T (millisecond) averaged over 500 runs of the FV primitives and speedup $\mathcal{S}$ for the full GPU implementation(GPU-FV$_{\text{RNS}}$) on P100 and K80, cooperative CPU-GPU FV(G-CPU-FV$_{\text{K80}}$) on K80, SEAL-FV$_{\text{RNS}}$ (v2.3.0-4) and NFLlib-FV.

Procedure	n	$\lvert q\rvert$		**GPU-FV$_{\text{RNS}}$**			**G-CPU-FV$_{\text{K80}}$**		**SEAL$_{\text{RNS}}$**		**NFLlib-FV**	
		SEAL	rest	T$_{\text{P100}}$	T$_{\text{K80}}$	$\mathcal{S}$	T	$\mathcal{S}$	T	$\mathcal{S}$	T	$\mathcal{S}$
KeyGen	2^{11}	60	62	2.786	5.943	2.13x	-	-	4.981	1.79x	**0.903**	0.32x
	2^{12}	180	186	**8.147**	17.218	2.11x	-	-	35.624	4.37x	12.131	1.49x
	2^{13}	360	372	**30.943**	86.095	2.78x	-	-	212.719	6.87x	83.366	2.69x
	2^{14}	720	744	**174.508**	356.224	2.04x	-	-	1474.903	8.45x	633.42	3.63x
Enc	2^{11}	60	62	**0.271**	0.541	2.00x	-	-	1.642	6.06x	0.276	1.02x
	2^{12}	180	186	**0.730**	1.440	1.97x	-	-	4.095	5.61x	1.081	1.48x
	2^{13}	360	372	**1.081**	2.645	2.45x	-	-	9.792	9.06x	4.321	4.00x
	2^{14}	720	744	**3.296**	6.657	2.02x	-	-	28.834	8.75x	17.353	5.26x
Dec	2^{11}	60	62	**0.065**	0.151	2.32x	1.484	9.83x	0.198	3.05x	0.396	6.09x
	2^{12}	180	186	**0.108**	0.194	1.80x	3.444	17.75x	0.967	8.95x	1.494	13.83x
	2^{13}	360	372	**0.152**	0.252	1.66x	9.615	38.15x	3.308	21.76x	4.645	30.56x
	2^{14}	720	744	**0.252**	0.610	2.42x	19.55	32.05x	13.888	55.11x	15.36	60.95x
Add	2^{11}	60	62	0.016	0.037	2.31x	-	-	**0.007**	0.44x	**0.007**	0.44x
	2^{12}	180	186	**0.021**	0.052	2.48x	-	-	0.054	2.57x	0.065	3.10x
	2^{13}	360	372	**0.030**	0.068	2.27x	-	-	0.177	5.90x	0.309	10.30x
	2^{14}	720	744	**0.053**	0.127	2.40x	-	-	0.612	11.55x	1.075	20.28x
Mul	2^{11}	60	62	**1.072**	3.343	3.12x	203.622	60.91x	2.327	2.17x	3.231	3.01x
	2^{12}	180	186	**1.833**	3.873	2.11x	817.958	211.19x	13.655	7.45x	16.722	9.12x
	2^{13}	360	372	**3.538**	7.700	2.18x	3351.660	435.28x	57.857	16.35x	81.363	23.00x
	2^{14}	720	744	**11.747**	28.953	2.46x	12674.51	437.76x	314.029	26.73x	465.587	39.63x

code, P100 utilization rate increases at lower rate compared to K80. Similarly, power consumption increase from 62 W to 148 W for K80. Power on P100 has a similar behavior but with less magnitude. Note that in this experiment GPU-FV$_{\text{RNS}}$ was set to run on a non-heated GPU card in each $(n, |q|)$ setting to limit the effect of previous runs on current measurements.

Lastly, we provide the latency of core compute kernels of our GPU implementation shown in Tables 6 and 7. It can be seen that the RNS kernels (Fast Base Conversion and Fast RNS Floor) contribute largely to the total execution time. It should be remarked that although CRT functions are also expensive, they are not on the critical path of computation as they are only used in encoding and decoding.

Table 6: Latency in (millisecond) of core compute kernels on NVIDIA Tesla K80.

n	$\lceil log_2^q \rceil$	**CRT**	**CRT Reconstruct**	**DGT**	**DGT Inverse**	**Fast Base Conversion**	**Fast RNS Floor**
2^{11}	62	0.007	0.024	0.031	0.033	0.023	0.050
2^{12}	186	0.022	0.056	0.039	0.041	0.064	0.140
2^{13}	372	0.091	0.131	0.048	0.050	0.183	0.443
2^{14}	744	1.057	1.409	0.193	0.202	0.858	2.090

Table 7: Latency in (millisecond) of core compute kernels on NVIDIA Tesla P100.

n	$\lceil log_2^q \rceil$	**CRT**	**CRT Reconstruct**	**DGT**	**DGT Inverse**	**Fast Base Conversion**	**Fast RNS Floor**
2^{11}	62	0.004	0.014	0.027	0.030	0.012	0.028
2^{12}	186	0.012	0.037	0.036	0.040	0.030	0.069
2^{13}	372	0.040	0.100	0.040	0.048	0.062	0.164
2^{14}	744	0.218	0.323	0.087	0.093	0.318	0.790

8 Conclusion

In this work, we presented the details of a GPU implementation of the FV SHE scheme. We developed a high performance modular arithmetic library for the polynomial ring $\mathbb{Z}_q[x]/\langle x^{2^k} + 1\rangle$. In this library, we employed several algebraic tools such as CRT, RNS and DGT/IDGT to perform arithmetic on long polynomials with large coefficients.

We showed that division-and-round and base decomposition in the textbook FV scheme require multi-precision arithmetic due to the need for positional numbering system. This is even more difficult to deal with on GPU. To overcome this problem, we employed the fast base conversion techniques of Bajard et al. to do computation in the CRT/RNS domain on GPU without the need to offload the computation to CPU.

The RNS techniques used here require conversion form NTT to RNS domain. It would lead to a more efficient solution if division-and-round and base decomposition were performed in the NTT domain directly. Moreover, other CUDA programming models may apply to improve our implementation further such as handling polynomial coefficient by a warp or half warp of threads instead of a single thread.

We compared our implementation with mature state-of-the-art implementations of the textbook FV and provided the speedups achieved.

As an immediate extension of this work, we believe our implementation can be ported to run on other hardware accelerators such FPGAs and ASICs. These platforms are known to offer even higher computational power than GPUs.

Acknowledgments

This work was supported by A * STAR, Data Storage Institute and the National University of Singapore. We thank Benjamin Hong Meng Tan and Nan Xiao who provided insight and expertise that greatly assisted the research. We would also like to thank the four anonymous reviewers for their comments and insights which helped in improving this work.

A CUDA code of main Algorithms

A.1 CRT Reconstruction via Garner's Algorithm

```
__global__ void ICRT_Garner(uint32 *out, uint32 *in,
                uint32 r, uint32 m_w32_cnt, uint32 n) {
        int t_id = blockIdx.x*blockDim.x+threadIdx.x;
        if (t_id >= n) return;
        uint32 x[z+1];
        for (int i = 0; i < m_w32_cnt; i++)
           x[i] = 0;
        uint32 u = in[t_id];
        x[0] = u;
        for (int m = 1; m < r; m++) {
           uint32 p = tab_p[m];
           uint32 t0 = ((uint64)tab_c[m]*in[m*n+t_id]) % p;
           uint32 x_w32_cnt = 0;
           for (int i = 0; i < m_w32_cnt; i++)
              if (x[i] != 0) x_w32_cnt++;
           int64 t1 = (int64)x[0] - in[m*n+t_id];
           if (t1<=p)
              t1 += p;
           if (t1>p)
              t1 -= p;
           uint32 old_x0 = x[0];
           x[0] = t1;

           u = MP_mod_SP(x, x_w32_cnt, p);
           u = ((uint64)tab_minus_c[m]*u) % p;
           u = t2;
           uint32 pj_index = ((m-1)*m)>>1;
           uint32 pj_w32_cnt = m;
           x[0] = old_x0;
           // Mul MP with SP and Add MP number
           MP_mul_SP_add_MP(tab_pj+pj_index, pj_w32_cnt,
                         u, x, x_w32_cnt);
        }
        for (int i = 0; i < m_w32_cnt; i++)
           out[t_id+i*n] = x[i];
}
```

References

[ABBA18] Ahmad Al Badawi, Veeravalli Bharadwaj, and Khin Mi Mi Aung. Efficient polynomial multiplication via modified discrete galois transform and negacyclic convolution. In *Future of Information and Communications Conference (FICC)*. IEEE, 2018.

[AMBG$^+$16] Carlos Aguilar-Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrede Lepoint. Nfllib: Ntt-based fast lattice library. In *Cryptographers? Track at the RSA Conference*, pages 341–356. Springer, 2016.

[amd] Opencl: Optimization case study fast fourier transform - part 1, 2. Available at: http://developer.amd.com/resources/articles-whitepapers/opencl-optimization-case-study-fast-fourier-transform-part-1/. Accessed 2017.

[BEHZ16] Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In *Selected Areas in Cryptography-SAC*, 2016.

[BFF12] Javier Baladron, Diego Fasoli, and Olivier Faugeras. Three applications of gpu computing in neuroscience. *Computing in Science & Engineering*, 14(3):40–47, 2012.

[BGH13] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *Public Key Cryptography*, volume 7778, pages 1–13. Springer, 2013.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

[BM$^+$58] George EP Box, Mervin E Muller, et al. A note on the generation of random normal deviates. *The annals of mathematical statistics*, 29(2):610–611, 1958.

[BP95] Ferruccio Barsi and Maria Cristina Pinotti. Fast base extension and precise scaling in rns for look-up table implementations. *IEEE transactions on signal processing*, 43(10):2427–2430, 1995.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *CRYPTO*, volume 7417, pages 868–886. Springer, 2012.

[CGRS14] David Bruce Cousins, John Golusky, Kurt Rohloff, and Daniel Sumorok. An fpga co-processor implementation of homomorphic encryption. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6. IEEE, 2014.

[cpp] C++ chrono time library. Available at: http://en.cppreference.com/w/cpp/chrono. Accessed 2018.

[Cra99] Richard E Crandall. Integer convolution via split-radix fast galois transform. *Center for Advanced Computation Reed College*, 1999.

[Cud] Cooperative groups: Flexible cuda thread programming. Available at: https://devblogs.nvidia.com/parallelforall/cooperative-groups/. Accessed 2017.

[DÖS15] Yarkın Doröz, Erdinç Öztürk, and Berk Sunar. Accelerating fully homomorphic encryption in hardware. *IEEE Transactions on Computers*, 64(6):1509–1521, 2015.

[DS15] Wei Dai and Berk Sunar. cuhe: A homomorphic encryption accelerator library. In *International Conference on Cryptography and Information Security in the Balkans*, pages 169–186. Springer, 2015.

[EW11] Niall Emmart and Charles C Weems. High precision integer multiplication with a gpu using strassen's algorithm with multiple fft sizes. *Parallel Processing Letters*, 21(03):359–375, 2011.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[Gar59] Harvey L Garner. The residue number system. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 146–153. ACM, IEEE, 1959.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme.* PhD thesis, Stanford University, 2009.

[GH11] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT*, volume 6632, pages 129–148. Springer, 2011.

[GLD$^+$08] Naga K Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 2. IEEE Press, 2008.

[Har13] William B Hart. Flint: Fast library for number theory. *Computeralgebra Rundbrief*, 2013.

[KLP16] Hao Chen Kim Laine and Rachel Player. Simple encrypted arithmetic library-seal (v2. 1). Technical report, Technical report, Microsoft Research, 2016.

[Knu97] Donald E Knuth. The art of computer programming, 3rd edn. seminumerical algorithms, vol. 2, 1997.

[LKC$^+$10] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH computer architecture news*, 38(3):451–460, 2010.

[LN14] Tancrede Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes fv and yashe. In *International Conference on Cryptology in Africa*, pages 318–335. Springer, 2014.

[Mun09] Aaftab Munshi. The opencl specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–314. IEEE, 2009.

[MV08] John Michalakes and Manish Vachharajani. Gpu acceleration of numerical weather prediction. *Parallel Processing Letters*, 18(04):531–548, 2008.

[MVOV96] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography.* CRC press, 1996.

[Nvi07] CUDA Nvidia. Compute unified device architecture programming guide, 2007.

[Nvi14] CUDA Nvidia. Toolkit documentation. *NVIDIA CUDA Getting Started Guide for Linux*, 2014.

[OP07] Amos R Omondi and Benjamin Premkumar. *Residue number systems: theory and implementation*, volume 2. World Scientific, 2007.

[PNPM15] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrian Macias. Accelerating homomorphic evaluation on reconfigurable hardware. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 143–163. Springer, 2015.

[Pol71] John M Pollard. The fast fourier transform in a finite field. *Mathematics of computation*, 25(114):365–374, 1971.

[RAD78] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[Rup12] K. Rupp. GPU-Accelerated Non-negative Matrix Factorization for Text Mining. In *NVIDIA GPU Technology Conference 2012*, page 77, 2012.

[S$^+$01] Victor Shoup et al. Ntl: A library for doing number theory, 2001.

[SK89] AP Shenoy and Ramdas Kumaresan. Fast base extension using a redundant modulus in rns. *IEEE Transactions on Computers*, 38(2):292–297, 1989.

[SS11] Damien Stehlé and Ron Steinfeld. Making ntru as secure as worst-case problems over ideal lattices. In *Eurocrypt*, volume 6632, pages 27–47. Springer, 2011.

[VDJ10] Marten Van Dijk and Ari Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. *HotSec*, 10:1–8, 2010.

[VZGG13] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra.* Cambridge university press, 2013.

[WHC$^+$12] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Accelerating fully homomorphic encryption using gpu. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5. IEEE, 2012.

[WHC$^+$15] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Exploring the feasibility of fully homomorphic encryption. *IEEE Transactions on Computers*, 64(3):698–706, 2015.

[Wil13] Nicholas Wilt. *The cuda handbook: A comprehensive guide to gpu programming.* Pearson Education, 2013.

[Win12] Franz Winkler. *Polynomial algorithms in computer algebra.* Springer Science & Business Media, 2012.

[WSTaM12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc?first experiences with real-world applications. *Euro-Par 2012 Parallel Processing*, pages 859–870, 2012.