cuHE: A Homomorphic Encryption Accelerator Library

Wei Dai and Berk Sunar

Worcester Polytechnic Institute,
Worcester, Massachusetts, USA

Abstract. We introduce a CUDA GPU library to accelerate evaluations with homomorphic schemes defined over polynomial rings enabled with a number of optimizations including algebraic techniques for efficient evaluation, memory minimization techniques, memory and thread scheduling and low level CUDA hand-tuned assembly optimizations to take full advantage of the mass parallelism and high memory bandwidth GPUs offer. The arithmetic functions constructed to handle very large polynomial operands using number-theoretic transform (NTT) and Chinese remainder theorem (CRT) based methods are then extended to implement the primitives of the leveled homomorphic encryption scheme proposed by López-Alt, Tromer and Vaikuntanathan. To compare the performance of the proposed CUDA library we implemented two applications: the Prince block cipher and homomorphic sorting algorithms on two GPU platforms in single GPU and multiple GPU configurations. We observed a speedup of 25 times and 51 times over the best previous GPU implementation for Prince with single and triple GPUs, respectively. Similarly for homomorphic sorting we obtained 12-41 times speedup depending on the number and size of the sorted elements.

Keywords: Homomorphic evaluation, GPU acceleration, large polynomial arithmetic.

1 Introduction

Fully homomorphic encryption (FHE) has gained increasing attention from cryptographers ever since its first plausible secure construction was introduced by Gentry [17] in 2009. FHE allows one to perform arbitrary computation on encrypted data without the need of a secret key, hence without knowledge of original data. That feature would have invaluable implications for the way we utilize computing services. For instance, FHE is capable of protecting the privacy of sensitive data on cloud computing platforms. We have witnessed amazing number of improvements in fully and somewhat homomorphic encryption schemes (SWHE) over the past few years [20, 34, 3, 4, 18, 2]. In [19] Gentry, Halevi and Smart (GHS) proposed the first homomorphic evaluation of a complex circuit, i.e. a full AES block. The implementation makes use of batching [31, 32], key switching [2] and modulus switching techniques to efficiently evaluate a leveled circuit. In [28] a leveled NTRU [24, 33] based FHE scheme was introduced by López-Alt, Tromer and Vaikuntanathan (LTV), featuring much slower growth of noise during homomorphic computation. Later Doröz, Hu and Sunar (DHS) [11] used an LTV SWHE variant to evaluate AES more efficiently. More recently, Ducas and Micciancio [15] presented an efficient implementation of the bootstrapping algorithm.

At the same time researchers have also started investigating how to best put these new homomorphic evaluation tools to use in privatizing applications. In particular, in [25] Lauter et al. analyzed the problems of evaluating averages, standard deviations, and logistical regressions which provide basic tools for a number of real-world applications in the medical, financial, and advertising domains. Later in [26], Lauter et al. demonstrated the viability of privatized computation of genomic data. In [14], Doröz et al. used an NTRU based SWHE scheme to construct a bandwidth efficient private information retrieval scheme. Bos et al. in [1] showed how to privately perform predictive analysis tasks on encrypted medical data. Graepel et al. in [21] showed that it is possible to execute machine learning algorithms on privatized data. Cheon et al. [7] presented an implementation to homomorphically evaluate dynamic programming algorithms such as Hamming distance, edit distance, and the Smith-Waterman algorithm on encrypted genomic data. Çetin et al. [6] analyzed the complexity and provided implementation results for homomorphic sorting.

Despite the rapid advances, HE evaluation efficiency remains as one of the obstacles preventing it from deployment in real-life applications. Given the computation and bandwidth complexity of HE schemes, alternative platforms such as FPGAs, application-specific integrated circuits (ASIC) and graphics processor units (GPU) need to be employed. Over the last decade GPUs have evolved to highly parallel, multi-threaded, many-core processor systems with tremendous computing power. Compared to FPGA and ASIC platforms, general-purpose computing on GPUs (GPGPU) yields higher efficiency when normalized by price. For example, in [12] an NTT conversion costs 0.05 msec on a $\$5,000$ FPGA, whereas takes only 0.15 msec on a \$200 GPU (NVIDIA GTX 770). The results of [35, 9, 10] demonstrate the power of GPU-accelerated HE evaluations. Another critical advantage of GPUs is the strong memory architecture and high communication bandwidth. Bandwidth is crucial for HE evaluation due to very large evaluation keys and ciphertexts. In contrast, FPGAs feature much simpler and more limited memory architectures and unless supplied with a custom memory I/O interface design the bandwidth suffers greatly.

For CPU platforms Halevi and Shoup published the HElib [23], a C++ library for HE that is based on Brakerski-Gentry-Vaikuntanathan (BGV) cryptosystem [2]. More recently, Ducas and Micciancio [15] published FHEW, another CPU library which features encryption and bootstrapping. In this paper, we propose cuHE: a GPU-accelerated optimized SWHE library in CUDA/C++. The library is designed to boost polynomial based HE schemes such as LTV, BGV and DHS. Our aim is to accelerate homomorphic circuit evaluations of leveled circuits via CUDA GPUs. To demonstrate the performance gain achieved with cuHE, by employing the DHS scheme [11], along with many optimizations, to implement the Prince block cipher and a homomorphic sorting algorithm on integers.

Our Contributions

- The cuHE library offers the feasibility of accelerating polynomial based homomorphic encryption and various circuit evaluations with CUDA GPUs.
- We incorporated various optimizations and design alternative methods to exploit the memory organization and bandwidth of CUDA GPUs. In particular, we adapted our parameter selection process to optimally map HE evaluation keys and precomputed values into the right storage type from fastest and more frequently used to slowest least accessed. Moreover, we also utilized OpenMP and CUDA hybrid programming for simultaneous computation on multiple GPUs.
- We attain the fastest homomorphic block cipher implementation, i.e. Prince at 51 msec (1 GPU), using the cuHE library which is 25 times faster than the previously reported fastest implementation [9]. Further, our library is able to evaluate homomorphic sorting on an integer array of various sizes 12-41 times faster compared to a CPU implementation [6].

2 Background

2.1 The LTV SWHE Scheme

In this section we briefly explain the LTV SWHE [28] with specializations introduced in [11]. We work with polynomials in ring $R = \mathbb{Z}[x]/(m(x))$ where $\deg m(x) = n$. All operations are performed in $R_q = R/qR$ where q is an odd modulus. Elements of $\mathbb{Z}_q$ are associated with elements of $\{\lfloor\frac{-q}{2}\rfloor,\ldots,\lfloor\frac{q}{2}\rfloor\}$. A truncated discrete Gaussian distribution χ is used as an error distribution from which we can sample random small B-bounded polynomials. The primitives of the public key encryption scheme are Keygen, Enc, Dec and Eval.

Keygen We generate a decreasing sequence of odd moduli $q_0 > q_1 > \cdots > q_{d-1}$ where d denotes the circuit depth and a monic polynomial $m(x)$. They define the ring for each level. $m(x)$ is the product of l monic polynomials each of which defines a message slot. In [11] more details about batching are explained. Keys are generated for the 0-th level, and are updated for every other level.

The 0*-th level* Sample $\alpha, \beta \leftarrow \chi$, and set $sk^{(0)} = 2\alpha + 1$ and $pk^{(0)} = 2\beta(sk^{(0)})^{-1}$ in ring $R_{q_0} = \mathbb{Z}_{q_0}[x]/\langle m\rangle$ (re-sample if $sk^{(0)}$ is not invertible in the this ring). Then for $\tau \in \mathbb{Z}_{\lceil\frac{\log q_0}{w}\rceil}$ where $w \leqslant \log q_{d-1}$ is a preset value, sample $s_\tau^{(0)}, e_\tau^{(0)} \leftarrow \chi$ and publish evaluation key $\{ek_\tau^{(0)} \mid \tau \in \mathbb{Z}_{\lceil\frac{\log q_0}{w}\rceil}\}$ where $ek_\tau^{(0)} = pk^{(0)}s_\tau^{(0)} + 2e_\tau^{(0)} + 2^{w\tau}sk^{(0)}$. *The i-th level* Compute $sk^{(i)} = sk^{(0)} \pmod{q_i}$ and $pk^{(i)} = pk^{(0)} \pmod{q_i}$ in ring $R_{q_i} = \mathbb{Z}_{q_i}[x]/\langle m\rangle$. Then compute evaluation key $\{ek_\tau^{(i)} \mid \tau \in \mathbb{Z}_{\lceil\frac{\log q_i}{w}\rceil}\}$ where $ek_\tau^{(i)} = ek_\tau^{(0)} \pmod{q_i}$.

Enc To encrypt a bit $b \in \{0,1\}$ with public key $(pk^{(0)}, q_0)$, sample $s, e \leftarrow \chi$, and set $c^{(0)} = pk^{(0)}s + 2e + b$ in R_{q_0}.

Dec To decrypt a ciphertext $c^{(i)}$, multiply the ciphertext with the corresponding private key $sk^{(i)}$ in R_{q_i} and then compute the message by modulo two: $b = c^{(i)}sk^{(i)} \pmod 2$.

Eval We perform arithmetic operations directly on ciphertexts. Suppose $c_1^{(i)} = \mathsf{Enc}(b_1) \pmod{q_i}$ and $c_2^{(i)} = \mathsf{Enc}(b_2) \pmod{q_i}$. The XOR gate is realized by adding ciphertexts: $b_1 + b_2 = \mathsf{Dec}(c_1^{(i)} + c_2^{(i)})$. The AND gate is realized by multiplying ciphertexts. However, polynomial multiplication incurs a much greater growth in the noise. So each multiplication step is followed by relinearization and modulus switching. First we compute $\tilde{c}^{(i)} = c_1^{(i)} \times c_2^{(i)}$ in R_{q_i}. To obtain $\tilde{c}^{(i+1)}$ from $\tilde{c}^{(i)}$, we perform relinearization on $\tilde{c}^{(i)}$. We expand it as $\tilde{c}^{(i)} = \sum_{\tau=0}^{\lceil\frac{\log q_i}{w}\rceil} 2^\tau \tilde{c}_\tau^{(i)}$ where $\tilde{c}_\tau^{(i)}$ takes its coefficients from $\mathbb{Z}_{2^w}$. Then set $\tilde{c}^{(i+1)} = \sum_{\tau=0}^{\lceil\frac{\log q_i}{w}\rceil} ek_\tau^{(i)}\tilde{c}_\tau^{(i)}$ in R_{q_i}. To obtain $c^{(i+1)}$ in $R_{q_{i+1}}$, we perform modulus switching: $c^{(i+1)} = \lfloor\frac{q_{i+1}}{q_i}c^{(i)}\rceil_2$ and then we have $m_1 \times m_2 = \mathsf{Dec}(c^{(i+1)})$.

2.2 Arithmetic Tools

Schönhage-Strassen's Multiplication Here we very briefly introduce Schönhage-Strassen's polynomial multiplication scheme [29]. Given $f = \sum_{k=0}^{n-1} a_k x^k$ and $g = \sum_{k=0}^{n-1} b_k x^k$, we compute $\hat{f} = \sum_{k=0}^{2n-1} \hat{a}_k x^k$, where $[\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{2n-1}] = \mathsf{NTT}([a_0, a_1, \ldots, a_{n-1}, 0, \ldots, 0])$. One multiplication of two degree n polynomials consists of two $2n$-point NTTs, one coefficient-wise multiplication and one $2n$-point inverse transform (INTT):

- Inputs: $f = \sum_{k=0}^{n-1} a_k x^k$, $g = \sum_{k=0}^{n-1} b_k x^k$;
- NTT Conversion: $f \to \hat{f} = \sum_{k=0}^{2n-1} \hat{a}_k x^k$, $g \to \hat{g} = \sum_{k=0}^{2n-1} \hat{b}_k x^k$;
- Output: $f \times g = \mathsf{INTT}(\sum_{k=0}^{2n-1} \hat{a}_k \hat{b}_k x^k)$.

CRT We introduce the CRT to handle large integer computation. We generate t prime numbers $\{p_0, p_1, \ldots, p_{t-1}\}$ with $B_p < 32$ bits. We further compute, for each level, $q_i = p_0 p_1 \cdots p_{t_i}$ where $0 < t_i < t_{i-1} < t$ as in [19]. Then, we have $R_{q_i} \cong R_{p_0} \times \cdots \times R_{p_{t_i}}$. Given a polynomial $f =$

$\sum_{k=0}^{n-1} a_k x^k$ in ring R_{q_i}, we compute a vector of polynomials $\mathcal{F} = [f_{(0)},\ f_{(1)},\ \ldots,\ f_{(t_i-1)}]$ as its CRT representation: $f_{(j)} = \sum_{k=0}^{n-1} a_{k(j)} x^k \ \in R_{p_j}$, where $a_{k(j)} = a_k \pmod{p_j}$, $j \in \mathbb{Z}_{t_i}$. For all $f, g \in R_{q_i}$ where $i \in \mathbb{Z}_d$, and $\mathcal{F} = \mathsf{CRT}(f)$, $\mathcal{G} = \mathsf{CRT}(g)$, we have $f \circ g = \mathsf{ICRT}(\mathcal{F} \circ \mathcal{G})$, where $\mathcal{F} \circ \mathcal{G} = [f_{(0)} \circ g_{(0)}, \ldots, f_{(t_i-1)} \circ g_{(t_i-1)}]$. Given a polynomial modulus m and $\mathcal{M} = \mathsf{CRT}(\mathsf{m})$, for all f, we have $f \pmod{m} = \mathsf{ICRT}(\mathcal{F} \pmod{\mathcal{M}})$. Other than CRT and ICRT, no large integer operation is needed.

2.3 CRT, NTT

In our implementations, the degree of modulus m is 8192, 16384 or 32768. And q_0 has more than 256, 512 or 1024 bits, respectively. Coefficient independent operations, e.g. polynomial addition, can provide sufficient parallelism for a GPU realization. Still, two problems remain to be solved: how to compute large integers on CUDA GPUs; and how to efficiently implement operations that are not coefficient independent, e.g. polynomial multiplication. Those problems are handled by using CRT and NTT together.

3 GPU Basics

GPUs are powerful but highly specialized devices that require careful coding to take full advantage of the massive parallelism offered. Specifically, the programming model and memory organization is much different from in CPUs. Here we present a concise overview.

3.1 Programming Model

In general, a GPU-accelerated scheme offloads compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. A GPU has its own on-chip memory. We call the CPU and memory "host", while GPU and its on-chip memory "device". A normal GPU computational task includes 3 operations: copying essential data from host to device (memcpy_h2d), initializing computation (a kernel) on device, copying result from device to host (memcpy_d2h) when necessary. A CUDA kernel is executed by an array of sequential threads. All threads run the same code, with an ID to compute memory addresses and make control decisions. On a GPU with warp size of 32, the kernel is executed in groups of 32 threads. Threads are further grouped into blocks. Only threads within a block can cooperate or synchronize. A kernel launch defines a grid of thread blocks. The dimension of a grid and that of each block determine how computing resource is assigned to program. The computation complexity of a kernel and the amount of data transferred between host and device depend on the details of an implementation.

3.2 Stream Management

A stream is a sequence of operations that execute in issue-order on the device. A default stream is created by CUDA on each device when no stream is specified. On a single stream, any operation will wait until the previous one completes. However, some operations, e.g. a kernel and a memcpy (without data dependency), are not necessarily sequential. We create extra streams so that operations on different streams can run concurrently and be interleaved. This not only allows a more flexible execution order, but also improves performance. Figure 3 (see Appendix) gives an example of how using multiple streams makes a difference. We can hide the latency of memcpy behind a kernel execution. Alternatively, we may further break down one kernel launch into several parts in order to create concurrency. Every stream belongs to its own device. To have streams on different devices run concurrently and synchronize as needed is multi-GPU computing. However, merely using streams to launch tasks on different devices creates expensive latency. Using OpenMP along with streams is a better solution. The goal of stream management is to achieve the best possible utilization of computing resources.

3.3 Memory Management

A significant ingredient to the performance of a program is memory management. The effect is particularly strong on GPUs since there are many different types of memory to store data and since the GPU-CPU interface tends to be slow. The GPU memory architecture is represented in Table 5 (see Appendix). Memory types are listed from top to bottom by access speed from fast to slow. Before executing a kernel, we need to feed constant memory and global memory with data, and bind texture memory if needed. Other than using streams to overlap data transfer and computation, we optimize these data transfers in following methods: minimizing the amount of data transferred between host and device when possible, batching many small transfers into one larger transfer, using page-locked (or pinned) memory to achieve a higher bandwidth. Towards an efficient application, kernels should be designed to take advantage of the memory hierarchy properties:

– Constant memory is cached and fast. Due to its limited size, e.g. 64 KB, it is only suitable for repeatedly requested data.
– Global memory is not cached, expensive to access, and huge in size, e.g. 2 GB. Data that is only read once, or is updated by kernels is better allocated in global memory. The pattern of memory access in kernels also matters. If each thread in a warp accesses memory contiguously from the same 128 B chunk, it is called coalesced memory access. A non-coalesced (strided) memory access could make a kernel hundreds of times slower.
– Texture memory is designed for this scenario: a thread is likely to read from an address near the ones that nearby threads read (non-coalesced). It is better to use texture memory when the data is updated rarely but read often, especially when the read access pattern exhibits a spatial locality.
– Shared memory is allocated for all threads in a block. It offers better performance than local or global memory and allows all threads in a block to communicate. Thus it is often used as a buffer to hold intermediate data, or to re-order strided global memory accesses to a coalesced pattern. However, only a limited size of shared memory can be allocated per block. Typically one configures the number of threads per block according to shared memory size. The shared memory is accessed by many threads, so that it is divided into banks. Since each bank can serve only one address per cycle, multiple simultaneous accesses to a bank result in a bank conflict. If all threads of a half-warp access a different bank (no bank conflict), the shared memory may become as fast as the registers.

4 Our Contribution: A CUDA Polynomial Arithmetic Library

In this section we explain how the basic polynomial arithmetic operations, such as multiplication, addition and polynomial modular reduction are implemented efficiently on CUDA GPUs. Our design is optimized for the device NVIDIA Geforce GTX 680, one of the Kepler architecture GPUs.

It has 1536 CUDA cores, 2 GB memory, 64 KB constant memory, 48 KB shared memory per block, CUDA Capability 3.0 and warp size of 32. On a device with better specifications, the program is believed to provide a better performance, yet has room to improve if configured and customized for the device.

4.1 Overview

Interfacing with NTL. We build our library to interface with the NTL library by Shoup [30]. Most implementations of polynomial based HE schemes are built on NTL. We provide an interface to NTL data types, in particular to the polynomial class ZZX so that GPU acceleration can be achieved with very little modification to a program. Another reason is that we only support very limited types of polynomial operations. Therefore, until non performance critical operations, e.g. a polynomial inversion in a polynomial ring, are implemented we may still utilize the NTL library.

Polynomial Representation. Suppose we work within the i-th level of a circuit. To store an

Table 1: Polynomial Representation on a GPU.

Domain	Word Type	# of Elements
RAW	32-bit unsigned int	$n\lceil\frac{\log q_i}{32}\rceil$
CRT	32-bit unsigned int	$t_i n$
NTT	64-bit unsigned int	$t_i n$

n-degree polynomial in R_{q_i} in GPU memory, we use an array of $n\lceil\frac{\log q_i}{32}\rceil$ 32-bit unsigned integers, where every $\lceil\frac{\log q_i}{32}\rceil$ integers denote a polynomial coefficient. We call a polynomial of this form in RAW domain. In the background section, we introduced two techniques: CRT and NTT, which give a polynomial CRT and NTT domain representations. Table 1 lists the structure and storage size of a polynomial in each domain. Figure 1 illustrates basic routines of operations on polynomials. Due to mathematical properties, each domain supports certain operations more efficiently as shown in Figure 1. In other words, to perform a certain operation, the polynomial should first be converted to a specific domain unless it is already in the desired domain. As shown in Table 1, the CRT domain representation requires more space than the RAW domain. However, having polynomials stay in the CRT domain saves one CRT conversion in every polynomial operation and one ICRT conversion in every operations except relinearization. Moreover, the sequence of operations might also create unnecessary latency if there are conversions that could have been spared.

4.2 CRT/ICRT

CRT prime numbers are precomputed based on the application settings and are stored in constant memory. ICRT conversion for a coefficient x is $x = \sum_{j=0}^{t_i-1} q_i/p_j \cdot ((q_i/p_j)^{-1} \cdot x_{(j)} \pmod{p_j}) \pmod{q_i}$ where $q_i = \prod_{j=0}^{t_i-1} p_j$. To efficiently compute ICRT, constants: q_i, $\{\frac{q_i}{p_j}\}$ and $\{(\frac{q_i}{p_j})^{-1} \pmod{p_j}\}$, where $j \in \mathbb{Z}_{t_i}$, are also precomputed and stored in constant memory. Given the coefficients of a RAW domain polynomial $[\mathbf{a_0}, \dots, \mathbf{a_{n-1}}]$, the number of 32-bit unsigned integers we use to represent each coefficient $\mathbf{a_k} \in \mathbb{Z}_{q_i}$ is $|\mathbf{a_k}| = \lceil\frac{\log q_i}{32}\rceil$. Its CRT domain representation is $\{[\mathbf{a_0}_{(j)}, \dots, \mathbf{a_{n-1}}_{(j)}] \mid j \in \mathbb{Z}_{t_i}\}$. A straightforward CRT_kernel design is to have every thread handle the CRT of one coefficient

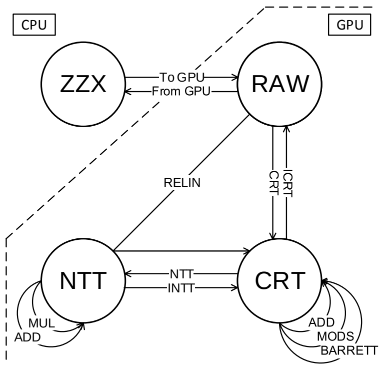


Fig. 1: Four representation domains for polynomials

$\mathbf{a_k}$, as in Figure 4a (see Appendix). However, that exhibits strided global memory access. Instead, we use shared memory to build a buffer as in Figure 4b (see Appendix). Not only do we reorder all accesses to global memory as coalesced, but also we avoid bank conflicts when reading or writing to shared memory. The ICRT_kernel operation is designed similarly. Moreover, we make wide use of registers in ICRT_kernel with assembly code for a better performance.

4.3 NTT/INTT

NTT is performed on a polynomial in R_{p_j}. We take an array $A = [a_0, \ldots, a_{n-1}, 0, \ldots, 0]$, i.e. n coefficients appended with n zeros, as input. We obtain a new array $\hat{A} = [\hat{a}_0, \ldots, \hat{a}_{2n-1}]$ by performing a $2n$-point NTT on A. Given t_i CRT prime numbers, to convert a CRT domain polynomial to NTT domain, we need t_i NTTs. We follow the approach of Dai et al. [9] to build an NTT scheme on GPU. According to FHE scheme settings, we only support NTTs of 16384, 32768 and 65536 points. Let $N = 2n$ be the size of NTT. We construct three CUDA kernels to adopt the four-step Cooley-Tukey algorithm [8]. As shown in Algorithm 4 (see Appendix), an N-point NTT is computed with several smaller size NTTs. What is not shown in Algorithm 4 is that a 64-point NTT is computed with 8-point NTTs. In [16] the benefit of working in finite field $\mathbb{F}_P$ is demonstrated where $P =$ `0xFFFFFFFF00000001`. In such a field, modulo P operations may be computed efficiently. Besides, 8 is a 64-th primitive root of P. By using $\langle 8 \rangle \subset \mathbb{F}_P$, 64-point NTTs can be done with shifts rather than requiring 64-bit by 64-bit multiplications. We build inline device functions for arithmetic operations in $\mathbb{F}_P$ in assembly code. In kernels we use shared memory to store those points. That ensures coalesced global memory accesses and fast transpose computation. We precompute $2N$ twiddle factors and bind them to texture memory since they are constant and are too large for constant memory. INTT is basically an NTT with extra steps. Given $\hat{A} = [\hat{a}_0, \ldots, \hat{a}_{2n-1}]$, we first re-order the array as $\hat{A}' = [\hat{a}_0, \hat{a}_{2n-1}, \hat{a}_{2n-2}, \ldots, \hat{a}_1]$. Then we compute $A = \frac{1}{N}\mathsf{NTT}(\hat{A}') \pmod{p_j}$.

4.4 Polynomial Multiplication

Polynomial multiplication takes NTT domain inputs or first converts inputs to NTT domain. Algorithm 1 shows the four steps needed to compute a multiplication. The coefficient-wise multiplication DOTMUL has high parallelism, which is very suitable for GPU computing. Compared to NTTs and

Algorithm 1 Polynomial Multiplication

```
1: Input NTT domain polynomials $\hat{\mathcal{F}}$ and $\hat{\mathcal{G}}$
2: $\hat{\mathcal{H}} = \hat{\mathcal{F}} \cdot \hat{\mathcal{G}}$                ▷ coefficient-wise multiplication
3: $\mathcal{H} \leftarrow \mathsf{INTT}(\hat{\mathcal{H}})$                ▷ convert to CRT domain
4: Output $\mathcal{H} \pmod{M}$                ▷ polynomial modular reduction
```

INTTs, DOTMUL is almost negligible in terms of overhead. Since the product is a $(2n-2)$-degree polynomial in CRT domain, it is followed by modular reductions over R_{p_j}, for all $j \in \mathbb{Z}_t$.

4.5 Polynomial Addition

Polynomial addition is essential for two functions in homomorphic circuit evaluation. One is in the homomorphic evaluation of an XOR gate which is simply implemented as a polynomial addition. For this the addition operation is carried out in the CRT domain. It provides sufficient parallelism for a GPU to process and also yields a result in the ring R_{q_i} without the need of coefficient modular reduction. The other computation that needs polynomial addition is in the accumulation part of relinearization. We will discuss this in detail later.

4.6 Polynomial Barrett Reduction

Polynomial computation is in ring $R_{q_i} = \mathbb{Z}_{q_i}/m$, where $\deg m = n$. Given a computation result f with $\deg f \geqslant n$, a polynomial reduction modulo m is needed. In fact, $\deg f \leq 2n-2$ always holds in our construction. We implement a customized Barrett reduction on polynomials by using our polynomial multiplication schemes as in Algorithm 2. We precomputed all constant polynomials generated from the modulus, and stored them in the GPU memory as described in the procedure "Precomputation". The goal of Barrett reduction is to compute $r = f \pmod{m}$. We take the CRT domain polynomial as input and return CRT domain polynomial as output.

4.7 Supporting HE Operations

To evaluate a leveled circuit, besides operations introduced above, we need other processes to reduce the introduced noise, e.g. by multiplication. An AND gate is followed by a relinearization. All ciphertexts are processed with modulus switching to be ready for next level. In our implementation Keygen is modified for a faster relinearization and parameters are selected to accommodate our GPU implementation.

Relinearization. A relinearization computes products of ciphertexts and evaluation keys. It then accumulates the products. By operating additions in the NTT domain we reduce the overhead of INTT in each multiplication. Given a polynomial $c^{(i)}$ in RAW domain we first expand it to $\tilde{c}_\tau^{(i)}$ where $\tau \in \mathbb{Z}_{\eta_i}$ and $\eta_i = \lceil \frac{\log q_i}{w} \rceil$. We call w the size of relinearization window. Then we need to compute $\tilde{c}^{(i+1)} = \sum_{\tau=0}^{\eta_i} ek_\tau^{(i)} \tilde{c}_\tau^{(i)}$ in R_{q_i} which is equivalent to computing $\tilde{\mathcal{C}}^{(i+1)} = \mathsf{INTT}\left(\sum_{\tau=0}^{\eta_i} \hat{\mathcal{EK}}_\tau^{(i)} \hat{\tilde{\mathcal{C}}}_\tau^{(i)}\right)$. We find a way to precompute and store evaluation keys for all levels. In Keygen, we convert evaluation keys of the 0-th level to NTT domain and store them. For every $\tau \in \mathbb{Z}_{\eta_0}$ compute

$$ek_\tau^{(0)} \xrightarrow{CRT} \mathcal{EK}_\tau^{(0)} \xrightarrow{NTT} \hat{\mathcal{EK}}_\tau^{(0)}.$$

Algorithm 2 Polynomial Barrett Reduction

```
1: procedure PRECOMPUTATION($m$)
2:     $u = \lfloor x^{2n-1}/m \rfloor$
3:     Store $\mathcal{M} = \mathsf{CRT}(m)$
4:     Store $\hat{\mathcal{M}} = \mathsf{NTT}(\mathcal{M})$
5:     Store $\hat{\mathcal{U}} = \mathsf{NTT}(CRT(u))$
6: end procedure
7: procedure BARRETTREDUCTION($\mathcal{F}$)
8:     $\mathcal{Q} = \mathsf{trunc}(\mathcal{F},\ n-1)$                ▷ input in CRT domain
9:     $\hat{\mathcal{Q}} = \mathsf{NTT}(\mathcal{Q})$                ▷ 1st multiplication
10:    $\hat{\mathcal{Q}} = \hat{\mathcal{Q}} * \hat{\mathcal{U}}$
11:    $\mathcal{Q} = \mathsf{INTT}(\hat{\mathcal{Q}})$
12:    $\mathcal{Q} = \mathsf{trunc}(\mathcal{Q},\ n)$
13:    $\hat{\mathcal{Q}} = \mathsf{NTT}(\mathcal{Q})$                ▷ 2nd multiplication
14:    $\hat{\mathcal{Q}} = \hat{\mathcal{Q}} * \hat{\mathcal{M}}$
15:    $\mathcal{Q} = \mathsf{INTT}(\hat{\mathcal{Q}})$
16:    $\mathcal{R} = \mathcal{F} - \mathcal{Q}$                ▷ subtraction
17:    if $\deg \mathcal{R} \geqslant \deg \mathcal{M}$ then
18:        $\mathcal{R} = \mathcal{R} - \mathcal{M}$
19:    end if
20:    Return $\mathcal{R}$                ▷ output in CRT domain
21: end procedure
```

Then $\{\hat{\mathcal{EK}}_\tau^{(0)} \mid \tau \in \mathbb{Z}_{\eta_0}\}$ is stored in GPU global memory. We no longer need to update the evaluation keys for any other level, observing that $\hat{\mathcal{EK}}_\tau^{(i)} \subseteq \hat{\mathcal{EK}}_\tau^{(0)}$, for all $i \in \mathbb{Z}_d$ and $\tau \in \mathbb{Z}_{\eta_i}$. Here what matters the most is the overhead of expanding and converting the ciphertexts. To convert $\tilde{c}_\tau^{(i)}$ to $\hat{\tilde{\mathcal{C}}}_\tau^{(i)}$ for all $\tau \in \mathbb{Z}_{\eta_i}$, we need η_i CRTs and $\eta_i t_i$ NTTs. However, if we set $w < \log p_j$, then for all $j \in \mathbb{Z}_{t_0}$ we have $\tilde{c}_\tau^{(i)} \in R_{2^w} \subset R_{p_j}$, i.e. $\tilde{\mathcal{C}}_\tau^{(i)} = \{\tilde{c}_\tau^{(i)}\}$. In such a setting, we only need η_i NTTs to convert $\tilde{c}_\tau^{(i)}$ to the NTT domain.

Based on these optimizations, we build a *multiplier and accumulator* for NTT domain polynomials. Suppose we have sufficient memory on GPU to hold all $\hat{\mathcal{EK}}_\tau^{(0)}$. Only one kernel that uses the shared memory to load all $\hat{\tilde{\mathcal{C}}}_\tau^{(i)}$ will suffice. We also provide solutions when the evaluation keys are too large for the GPU memory to hold. On a multi-GPU system, we evenly distribute the keys on devices. When the keys on another device is requested, copy them from that device to the current device. This is the best solution for two reasons: the bandwidth between devices is much larger than that between the device and host; accesses to memory on another device in a kernel yield roughly 3 times less overhead, compared to accessing the current device's memory.

Double-CRT Setting. According to [11], to correctly evaluate a circuit of depth d and to reach a desired security level, we can determine the lower bounds of n and $\log q_0$, and that $\delta_q \leq \log \frac{q_i}{q_{i+1}} = \prod_{j=t_i}^{t_i-1} p_j$ where $i \in \mathbb{Z}_d$. Let B_p be the size of CRT prime numbers, i.e. $B_p = \log p_j$. Then we know that $2^{B_p} < \sqrt{P/n} < 2^{32}$. To simplify, we set $t_i = d - i - 1$, $B_p \geq \delta_q$. Then we have

$$\delta_q \leq B_p < \log \sqrt{P/n}.$$

We select $B_p = \delta_q$. Then we select the relinearization window size such as $w < B_p$. In these settings, we reach the desired security level with minimal computation.

Algorithm 3 Modulus Switching

```
1: $\mathcal{A} = \{a_{(0)}, \ldots, a_{(t_i-1)}\} \leftarrow \mathsf{CRT}(a^{(i)})$
2: for $k \leftarrow 1,$ do
3:     $a^* \leftarrow a_{(t_i-k)}$
4:     if $a^* = 1 \pmod 2$ then
5:         if $a^* > (p_{t_i-k} - 1)/2$ then
6:             $a^* = a^* - p_{t_i-k}$
7:         else
8:             $a^* = a^* + p_{t_i-k}$
9:         end if
10:    end if
11:    $\mathcal{A} = (\mathcal{A} - a^*)/p_{t_i-k} \pmod{p_{t_i-k}}$
12: end for
13: $a^{(i+1)} = \mathsf{ICRT}(\mathcal{A}) = \mathsf{ICRT}\big(\{a_{(0)}, \ldots, a_{(t_{i+1}-1)}\}\big)$
```

Modulus Switching. In [19] Gentry et al. proposed a method to perform modulus switching on ciphertexts in CRT domain (double-CRT), by generating $q_i = p_0 p_1 \cdots p_{t_i-1}$ where $i \in \mathbb{Z}_d$. Since modulus switching is a coefficient independent operation, to simplify, we represent it on a single coefficient. Given a coefficient $a^{(i)} \in \mathbb{Z}_{q_i}$ where $i \in \mathbb{Z}_d$, modulus switching is designed as in Algorithm 3 to obtain $a^{(i+1)} \in \mathbb{Z}_{q_{i+1}}$ such that $a^{(i+1)} = a^{(i)} \pmod 2$ and $\epsilon = |a^{(i+1)} - \frac{q_{i+1}}{q_i} a^{(i)}|$ where $-1 \leq \epsilon \leq 1$ always holds. We precompute $p_j^{-1} \pmod{p_k}$ for all $k \in \mathbb{Z}_{t_0} \setminus \mathbb{Z}_{t_{d-1}}$ and $j \in \mathbb{Z}_k$. These values are stored as a lookup table in constant memory.

Precomputation Routine. For a circuit with depth d, we select parameters with a sequence of constraints: $d \rightarrow n \rightarrow \delta_q \rightarrow B_p \rightarrow w$. We generate a set of d prime numbers with B_p bits $\mathcal{P} = \{p_0, \ldots, p_{d-1}\}$ as CRT constants. For each level $i \in \mathbb{Z}_d$ of the circuit we generate ICRT constants: $q_i = \prod_{j=0}^{i-1} p_j$, $\mathcal{Q}_i^* = \{\frac{q_i}{p_j} \mid j \in \mathbb{Z}_i\}$ and $\mathcal{Q}_i^\dagger = \{(\frac{q_i}{p_j})^{-1} \pmod{p_j} \mid j \in \mathbb{Z}_i\}$. We also generate Modulus Switching constants for all levels: $\mathcal{P}^{-1} = \{p_{j,k}^{-1} = p_k^{-1} \pmod{p_j} \mid j \in \mathbb{Z}_i \setminus \{0\}, k \in \mathbb{Z}_j\}$. $\mathcal{P}$ and $\mathcal{P}^{-1}$ are stored in GPU constant memory. However, we store $\mathcal{Q} = \{q_i \mid i \in \mathbb{Z}_d\}$, $\mathcal{Q}^* = \{\mathcal{Q}_i^* \mid i \in \mathbb{Z}_d\}$ and $\mathcal{Q}^\dagger = \{\mathcal{Q}_i^\dagger \mid i \in \mathbb{Z}_d\}$ in CPU memory at first. We update ICRT constants for ICRT conversions in a new level by copying q_i, $\mathcal{Q}_i^*$ and $\mathcal{Q}_i^\dagger$ to GPU constant memory. We generate an n degree monic polynomial m as polynomial modulus and compute $u = x^{2n-1}/m \in R_{q_0}$ for Barrett reduction. Their CRT and NTT domain representations $\mathcal{M}$, $\hat{\mathcal{M}}$ and $\hat{\mathcal{U}}$ are computed and bound to GPU texture memory. Table 2 is a summary of precomputed data, showing storage memory types and sizes. Besides general expressions of size in bytes, we also list the memory usage of three target circuits: Prince stands for the Prince block cipher that has $[d = 25,\ n = 16384,\ B_p = 25,\ w = 16]$. Sorting(8) is a sorting circuit of 8 unsigned 32-bit integers, with parameters set to $[13,\ 8192,\ 20,\ 16]$. Similarly, Sorting(32) sorts 32 unsigned integers and has parameters $[15,\ 8192,\ 22,\ 16]$.

Keygen As explained in background, for all levels, we generate secret keys $sk^{(i)}$ and public keys $pk^{(i)}$. Based on those, we compute $ek_{\tau^{(i)}}$ as evaluation keys. We then convert and store their NTT domain representations $\hat{\mathcal{EK}}_{\tau^{(i)}}$ in GPU memory.

5 Implementation Results

We implemented the proposed algorithms on two target GPU platforms: NVIDIA GeForce GTX770 and GTX690. Note that the GTX690 consists of two GTX680 GPUs. We programmed the GTX690

Table 2: Precomputation

Content	Memory Type	Size (Bytes)			
		Equation	Prince	Sorting(8)	Sorting(32)
$\mathcal{P}$	constant	$4d$	100	52	60
q_i	constant	$4\lceil (d-i)B_p/32 \rceil$	≤ 80	≤ 36	≤ 44
$\mathcal{Q}_i^*$	constant	$4(d-i)\lceil (d-i-1)B_p/32 \rceil$	$\leq 1,900$	≤ 416	≤ 600
$\mathcal{Q}_i^\dagger$	constant	$4(d-i)$	≤ 100	≤ 52	≤ 60
$\mathcal{P}^{-1}$	constant	$2d(d-1)$	$1,200$	312	420
$\mathcal{M}$	texture	$4dn$	$1,638,400$	$425,984$	$491,520$
$\hat{\mathcal{M}}$	texture	$16dn$	$6,553,600$	$1,703,936$	$1,966,080$
$\hat{\mathcal{U}}$	texture	$16dn$	$6,553,600$	$1,703,936$	$1,966,080$
$\hat{\mathcal{EK}}_\tau^{(i)}$	global	$16dn\lceil dB_p/w \rceil$	$262,144,000$	$28,966,912$	$41,287,680$

in both single GPU, i.e. GTX 680, and in multi-GPU modes. The testing environment is summarized in Table 3. We show performance of our library and compare it to CPU implementations using the NTL library (v9.2.0) which is adopted by DHS-HE [11] and HELib [22].

Table 3: Testing Environment

Item	Specification	Item	Specification
CPU	Intel Core i7-3770K	GPU	NVIDIA GeForce GTX690
# of Cores	4	# of CUDA Cores	1536×2
# of Threads	8	GPU Core Frequency	1020 MHz
CPU Frequency	3.50 GHz	GPU Memory	2 GB $\times$ 2
Cache	8 MB	GPU	NVIDIA GeForce GTX770
System Memory	32 GB DDR3	# of CUDA Cores	1536
NTL (single-threaded)	9.2.0	GPU Core Frequency	1163 MHz
GMP	6.0.0a	GPU Memory	2 GB

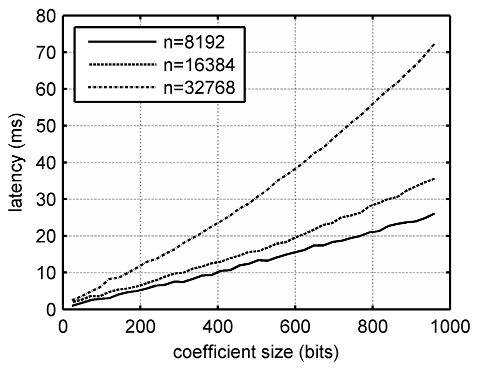
5.1 Performance of GPU Library Primitives

Table 6 (see Appendix) shows the latency of the basic polynomial operations. MULADD stands for the *multiplier and accumulator* for NTT domain polynomials. ADD denotes polynomial addition in CRT domain. The latencies in the table of NTT conversions, whose speed is solely affected by n, consist of d iterations.

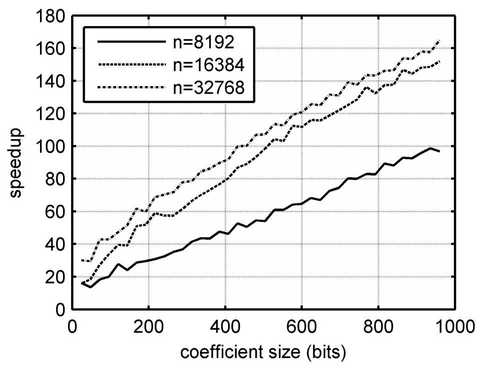
Figure 2a shows the performance of relinearization. Doröz et al. [11] use the NTL library with an optimized polynomial reduction method. As shown in Figure 2b, the speedup is at least 20 times, and increases as the coefficient size increases, up to 160 times for 960 bit coefficients. Note that Dai et al. [9] did not fully implement relinearization on the GPU but rather relies on NTL/CPU for coefficient and polynomial reduction. For instance, for Prince parameters with coefficient size of 575 bits, our relinearization takes only 18.3 msec whereas Dai et al.'s takes 890 msec on GPU plus an additional 363 msec for reduction on the CPU. This yields a speedup of 68 times.

5.2 Performance of Sample Algorithms

To demonstrate the performance gain obtained by the cuHE library we implemented the Prince block cipher, and homomorphic sorting algorithms with array sizes $4, 8, 16, 32$. The homomorphic



(a)



(b)

Fig. 2: Performance of relinearization (a) latency with growing coefficient size (b) speedup over [11]

Table 4: Performance of Implemented Algorithms

Platform	Prince		Sorting 8		Sorting 16		Sorting 32	
	Amortized 1/1024	**Speedup**	**Amortized 1/630**	**Speedup**	**Amortized 1/630**	**Speedup**	**Amortized 1/630**	**Speedup**
CPU (1-bit) [13]	3.3 sec	1	n/a	n/a	n/a	n/a	n/a	n/a
GTX 680 (1-bit) [9]	1.28 sec	2.6	n/a	n/a	n/a	n/a	n/a	n/a
CPU (16-bit) [13, 5]	1.98 sec	1.7	944 ms	1	4.28 sec	1	18.60 sec	1
GTX 680 (1 GPU)	51 ms	64	62 ms	15	291 ms	15	1.52 sec	12
GTX 770 (1 GPU)	45 ms	72	55 ms	17	256 ms	17	1.35 sec	14
GTX 690 (2 GPUs)	32 ms	103	34 ms	27	162 ms	26	864 ms	22
GTX 690/770 (3 GPUs)	24 ms	134	23 ms	41	108 ms	39	678 ms	27

evaluation performance is summarized in Table 4. We updated and reran Doröz et al.'s homomorphic Prince [13] with a 16-bit relinearization window. With cuHE library, we achieve 40 times speedup on a single GPU, 135 times on three GPUs simultaneously, over the Doröz et al. CPU implementation. Also compared to Dai et al.'s [9] the speedup is 25 times on the same GPU device.

Finally we would like to note that the proposed Prince implementation is the *fastest homomorphic block cipher implementation* currently available. For instance, Lepoint and Naehrig evaluated homomorphic SIMON-64/128 in 2.04 sec with 4 cores on Intel Core i7-2600 at 3.4 GHz [27] for the $n = 32,768$ setting. Our homomorphic Prince is 40 times faster for $n = 16,384$, and 20 times for $n = 32,768$.

Acknowledgment

Funding for this research was in part provided by the US National Science Foundation CNS Award #1319130.

Appendix

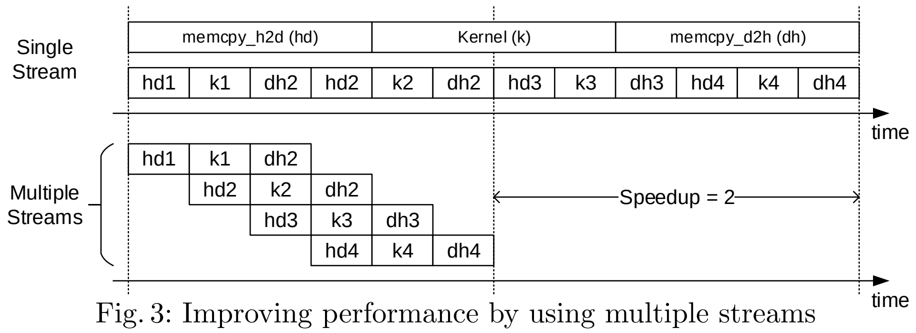


Fig. 3: Improving performance by using multiple streams

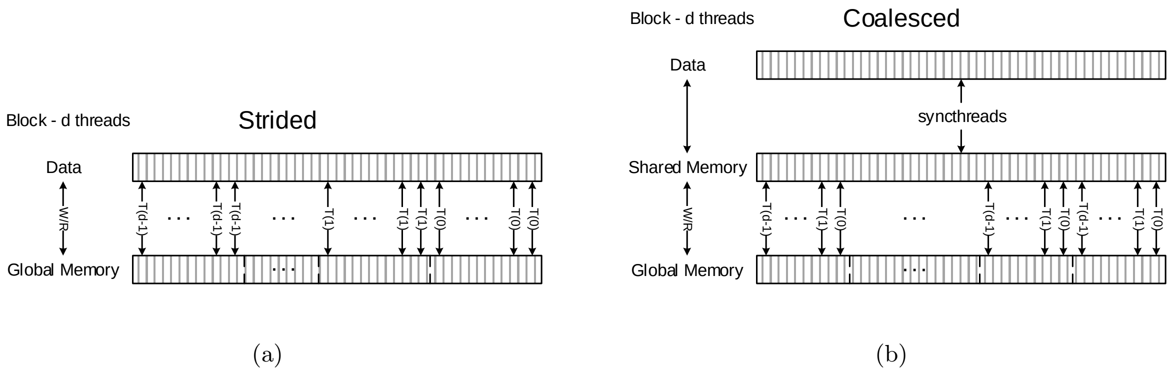


Fig. 4: Using shared memory to avoid strided access to global memory.

Table 5: GPU memory organization

Memory	**Cached**	**Access**	**Scope**	**Lifetime**
Register	N/A	R/W	One thread	Thread
Constant	Yes	R	All thread + host	Application
Texture	Yes	R	All thread + host	Application
Shared	N/A	R/W	All threads in a block	Block
Local	No	R/W	One thread	Thread
Global	No	R/W	All thread + host	Application

Table 6: Performance of Basic Operations on Polynomials (d, n, dB_p) where $B_p = 24$

Functions	**Latency (ms)**		
	(15, 8192, 360)	**(25, 16384, 600)**	**(40, 32768, 960)**
CRT	0.7	4	21.31
ICRT	0.54	3.73	17.94
NTT	0.84	1.78	6.24
INTT	0.98	2.09	6.86
MULADD	0.06	0.11	0.19
BARRETT	5.1	10	32.63
ADD	0.1	0.67	0.92

Algorithm 4 N-point NTT

```
 1: $N$ samples: 4096 rows (consecutive) by $N/4096$ columns
 2: for $N/4096$ columns do                                  ▷ 1st kernel
 3:     4096 samples: 64 rows by 64 columns
 4:     for 64 columns do
 5:         64-point NTT
 6:     end for
 7:     Transpose
 8:     Multiply twiddle factors of 4096-point NTT
 9:     for 64 columns do                                    ▷ 2nd kernel
10:         64-point NTT
11:     end for
12: end for
13: Transpose
14: Multiply twiddle factors of $N$-point NTT
15: for 4096 columns do                                      ▷ 3rd kernel
16:     $N/4096$-point NTT
17: end for
```

References

1. Bos, J.W., Lauter, K., Naehrig, M.: Private predictive analysis on encrypted medical data. Tech. Rep. MSR-TR-2013-81 (September 2013), http://research.microsoft.com/apps/pubs/default.aspx?id=200652
2. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. pp. 309–325. ACM (2012)
3. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-lwe and security for key dependent messages. In: Advances in Cryptology–CRYPTO 2011, pp. 505–524. Springer (2011)
4. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) lwe. SIAM Journal on Computing 43(2), 831–871 (2014)
5. Çetin, G.S., Doröz, Y., Sunar, B., Savaş, E.: Low Depth Circuits for Efficient Homomorphic Sorting (To appear in LATINCRYPT 2015)
6. Chatterjee, A., Kaushal, M., Sengupta, I.: Accelerating sorting of fully homomorphic encrypted data. In: Paul, G., Vaudenay, S. (eds.) Progress in Cryptology INDOCRYPT 2013, Lecture Notes in Computer Science, vol. 8250, pp. 262–273. Springer International Publishing (2013)
7. Cheon, Jung, H., Miran, K., Kristin, L.: Secure dna-sequence analysis on encrypted dna nucleotides. (2014), http://media.eurekalert.org/aaasnewsroom/MCM/FIL_000000001439/EncryptedSW.pdf
8. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Mathematics of computation 19(90), 297–301 (1965)
9. Dai, W., Doröz, Y., Sunar, B.: Accelerating NTRU based Homomorphic Encryption using GPUs. 2014 IEEE High Performance Extreme Computing Conference (HPEC'14) (2014)
10. Dai, W., Doröz, Y., Sunar, B.: Accelerating SWHE based PIRs using GPUs (To appear in WAHC'2015)
11. Doröz, Y., Hu, Y., Sunar, B.: Homomorphic AES Evaluation using NTRU. IACR Cryptology ePrint Archive 2014, 39 (2014)

12. Doröz, Y., Öztürk, E., Savaş, E., Sunar, B.: Accelerating ltv based homomorphic encryption in reconfigurable hardware (To appear in Cryptographic Hardware and Embedded Systems–CHES 2015)
13. Doröz, Y., Shahverdi, A., Eisenbarth, T., Sunar, B.: Toward practical homomorphic evaluation of block ciphers using prince. In: Financial Cryptography and Data Security, pp. 208–220. Springer (2014)
14. Doröz, Y., Sunar, B., Hammouri, G.: Bandwidth efficient pir from ntru. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) Financial Cryptography and Data Security, Lecture Notes in Computer Science, vol. 8438, pp. 195–207. Springer Berlin Heidelberg (2014)
15. Ducas, L., Micciancio, D.: Fhew: Bootstrapping homomorphic encryption in less than a second. In: Advances in Cryptology–EUROCRYPT 2015, pp. 617–640. Springer (2015)
16. Emmart, N., Weems, C.C.: High precision integer multiplication with a GPU using Strassen's algorithm with multiple FFT sizes. Parallel Processing Letters 21(03), 359–375 (2011)
17. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009)
18. Gentry, C., Halevi, S.: Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. In: Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on. pp. 107–109. IEEE (2011)
19. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Advances in Cryptology–CRYPTO 2012, pp. 850–867. Springer (2012)
20. Gentry, C., et al.: Fully homomorphic encryption using ideal lattices. In: STOC. vol. 9, pp. 169–178 (2009)
21. Graepel, T., Lauter, K., Naehrig, M.: Ml confidential: Machine learning on encrypted data. In: Kwon, T., Lee, M.K., Kwon, D. (eds.) Information Security and Cryptology ICISC 2012, Lecture Notes in Computer Science, vol. 7839, pp. 1–21. Springer Berlin Heidelberg (2013)
22. Halevi, S., Shoup, V.: HElib - An Implementation of homomorphic encryption, https://github.com/shaih/HElib/
23. Halevi, S., Shoup, V.: Design and implementation of a homomorphic-encryption library (2013)
24. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: Algorithmic number theory, pp. 267–288. Springer (1998)
25. Lauter, K., Naehrig, M., Vaikuntanathan, V.: Can homomorphic encryption be practical. Cloud Computing Security Workshop pp. 113–124 (2011)
26. Lauter, K., Lopez-Alt, A., Naehrig, M.: Private computation on encrypted genomic data. Tech. Rep. MSR-TR-2014-93 (June 2014), http://research.microsoft.com/apps/pubs/default.aspx?id=219979
27. Lepoint, T., Naehrig, M.: A comparison of the homomorphic encryption schemes fv and yashe. In: Progress in Cryptology–AFRICACRYPT 2014, pp. 318–335. Springer (2014)
28. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: Proceedings of the forty-fourth annual ACM symposium on Theory of computing. pp. 1219–1234. ACM (2012)
29. Schönhage, D.D.A., Strassen, V.: Schnelle multiplikation grosser zahlen. Computing 7(3-4), 281–292 (1971)
30. Shoup, V.: NTL: A Library for doing Number Theory, http://www.shoup.net/ntl/
31. Smart, N.P., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Public Key Cryptography–PKC 2010, pp. 420–443. Springer (2010)
32. Smart, N.P., Vercauteren, F.: Fully homomorphic simd operations. Designs, codes and cryptography 71(1), 57–81 (2014)
33. Stehlé, D., Steinfeld, R.: Making NTRU as secure as worst-case problems over ideal lattices. In: Advances in Cryptology–EUROCRYPT 2011, pp. 27–47. Springer (2011)
34. Van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Advances in cryptology–EUROCRYPT 2010, pp. 24–43. Springer (2010)
35. Wang, W., Hu, Y., Chen, L., Huang, X., Sunar, B.: Accelerating fully homomorphic encryption using GPU. In: High Performance Extreme Computing (HPEC), 2012 IEEE Conference on. pp. 1–5. IEEE (2012)