Accelerating NTRU based Homomorphic Encryption using GPUs

Wei Dai
Dept. of Electrical and Computer Engineering
Worcester Polytechnic Institute
Worcester, MA 01609
Email: wdai@wpi.edu

Yarkın Doröz
Dept. of Electrical and Computer Engineering
Worcester Polytechnic Institute
Worcester, MA 01609
Email: ydoroz@wpi.edu

Berk Sunar
Dept. of Electrical and Computer Engineering
Worcester Polytechnic Institute
Worcester, MA 01609
Email: sunar@wpi.edu

***Abstract*—**

We introduce a large polynomial arithmetic library optimized for Nvidia GPUs to support fully homomorphic encryption schemes. To realize the large polynomial arithmetic library we convert polynomials with large coefficients using the Chinese Remainder Theorem into many polynomials with small coefficients, and then carry out modular multiplications in the residue space using a custom developed discrete Fourier transform library. We further extend the library to support the homomorphic evaluation operations, i.e. addition, multiplication, and relinearization, in an NTRU based somewhat homomorphic encryption library. Finally, we put the library to use to evaluate homomorphic evaluation of two block ciphers: Prince and AES, which show 2.57 times and 7.6 times speedup, respectively, over an Intel Xeon software implementation.

I. Introduction

Since the first plausible construction was presented 5 years ago by Gentry [1] fully homomorphic encryption (FHE) has captured significant attention from academia, media and even industry. A FHE scheme is an encryption scheme which allows the efficient evaluation of an arbitrary depth circuit on data in encrypted form. Therefore, FHE research could ultimately lead to very exciting developments in a wide variety of areas enabling secure storage and more broadly private computations on the cloud [2]. Since its inception, three main branches of homomorphic encryption schemes emerged: lattice-based, integer-based [3]–[5] and learning-with-errors (LWE) or (ring) learning with errors ((R)LWE) based encryption [6]–[8].

Despite the rapid advances we have witnessed over the last 5 years, FHE still has not sufficiently matured to be used in real life applications. For example, an implementation by Gentry *et al.* [9] requires 36 hours for a homomorphic evaluation of AES using FHE. Another NTRU based proposal by Doröz [10] manages to evaluate AES nearly an order of magnitude faster than [9]. However, even this result is far from being used in practice. Another significant bottleneck of FHE schemes is memory use. In general, to support the evaluation of reasonably deep circuits while providing sufficient margin to counter lattice attacks, large ciphertext and public key sizes are required, e.g. see [9], [11]. Motivated by these two bottlenecks a significant number of works focused on improving the efficiency of the FHE schemes. For instance, techniques for eliminating the need for expensive bootstrapping evaluations [12] and techniques for batching multiple bits together for more effective parallel processing [13]–[15] have been developed. To tackle the efficiency bottleneck the use of alternative platforms, such as GPUs [16], reconfigurable logic [17]–[21] and custom ASIC [22], [23] has been proposed.

Initial research on FHE concentrated on lattice based schemes [11], [24], [25], involving relatively large public key and ciphertext sizes. Lattice based FHE schemes, and indeed the other FHE schemes, base their security on hard problems associated with lattices, such as the sparse subset sum problem (SSSP) or the shortest vector problem (SVP). SIMD techniques were proposed by Smart and Vercauteren [13] for these schemes to perform tasks in parallel and thus improve efficiency.

Our Contribution. In this work, we build a discrete Fourier transform based library for Nvidia GTX GPUs from the ground up to support somewhat and fully homomorphic encryption schemes based on large polynomial multiplications. We extend the library with homomorphic evaluation primitives including addition, multiplication and relinearization to handle NTRU based evaluation directly on the GPU. To demonstrate the efficiency of the library we implemented homomorphic evaluation of two block ciphers, i.e. AES and Prince, recently considered for homomorphic evaluation. We obtain significant speedups over the previously reported results. More importantly we determined that the efficiency is limited by the memory of the target GPU. Hence, even greater speedups may be achieved simply by moving to GPUs with more memory.

II. Related Work

Here we briefly summarize previous FHE/SWHE work which utilized GPUs to improve the performance. The first GPU implementation of a FHE scheme was presented by Wang *et al.* [16]. The authors implemented Gentry and Halevi's lattice-based FHE scheme [11] on an NVIDIA C2050 GPU using the Number Theoretical Transform (NTT) algorithm, achieving speed up factors of 7.68, 7.4 and 6.59 for encryption, decryption and the recryption operations, respectively, in the small parameter setting. The NTT algorithm was used to optimize the critical operation, i.e. modular multiplication of very large integers using the Schönhage Strassen algorithm followed by Barrett's reduction technique. For further performance gains the critical operations were parallelized and implemented on a GPU along with a number of precomputation optimizations.

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

An extension of the authors' work [26] involves the modification of arithmetic operations to decrease costly back and forth NTT conversions. This modified method, when implemented on an NVIDIA GTX 690, achieves speed up factors of 174, 7.6 and 13.5 for encryption, decryption and the recryption, respectively, compared to results of the implementation of Gentry and Halevi's FHE scheme [11] running on an Intel Core i7 3770K machine. Wang *et al.* [21] propose an architecture for a 768K-bit FFT multiplier using a 64K-point finite field FFT as the key component. This component was implemented on both a Stratix V FPGA and a NVIDA Tesla C2050 GPU. The FPGA implementation is twice as fast as on the GPU [16].

A. Background

In 2012 Alt-López, Tromer and Vaikuntanathan proposed a leveled multi-key FHE scheme (ATV) [2]. The scheme was based on a variant of NTRU encryption scheme proposed by Stehlé and Steinfeld [27]. In the single key case of the scheme, operations for polynomial degree n and prime modulus q are performed in $\mathrm{R}_q = \mathbb{Z}_q[x]/\langle x^n+1\rangle$. The schme contains an error distribution function χ to sample random small B-bounded polynomials. The error distribution χ is a truncated discrete Gaussian distribution. The scheme consist of four primitive functions **KeyGen**, **Encrypt**, **Decrypt** and **Eval**:

KeyGen. We select decreasing sequence of primes $q_0 > q_1 > \cdots > q_d$ for each level. We sample $g^{(i)}$ and $u^{(i)}$ from χ, compute secret keys $f^{(i)} = 2u^{(i)}+1$ and public keys $h^{(i)} = 2g^{(i)}(f^{(i)})^{-1}$ for each level. Later we create evaluation keys for each level: $\zeta_\tau^{(i)}(x) = h^{(i)}s_\tau^{(i)} + 2e_\tau^{(i)} + 2^\tau(f^{(i-1)})^2$ where $\{s_\tau^{(i)}, e_\tau^{(i)}\} \in \chi$ and $\tau = [0, \lfloor \log q_i \rfloor]$.

Encrypt. To encrypt a bit b for the i^{th} level we compute: $c^{(i)} = h^{(i)}s + 2e + b$ where $\{s, e\} \in \chi$.

Decrypt. In order to compute the decryption of a value for specific level i we compute: $m = c^{(i)}f^{(i)} \pmod 2$.

Eval. The multiplication and addition of ciphertexts corresponds to XOR and AND operations respectively. The multiplication operation creates a significant noise which is coped by using relinearization and modulus switching. The relinearization computes $\tilde{c}^{(i)}(x) = \sum_\tau \zeta_\tau^{(i)}(x)\tilde{c}_\tau^{(i-1)}(x)$ where $\tilde{c}_\tau^{(i-1)}(x)$ are 1-bounded polynomials given as $\tilde{c}^{(i-1)}(x) = \sum_\tau 2^\tau \tilde{c}_\tau^{(i-1)}(x)$. By a modulus switching operation carried out as $\tilde{c}^{(i)}(x) = \lfloor \frac{q_i}{q_{i-1}}\tilde{c}^{(i)}(x) \rceil_2$ to cut the noise level by $\log(q_i/q_{i-1})$ bits. The operation $\lfloor\cdot\rceil_2$ rounds to the nearest integer in such a way as to match the parity bits to the original ciphertext.

We use a customized version of the ATV-FHE Scheme that is proposed in [10] by Doröz, Hu and Sunar (DHS). The code is written in C++ using NTL package that is compiled with GMP library. The library has the following costimizations:

- The operations are performed in $\mathrm{R}_{q_i} = \mathbb{Z}_{q_i}[x]/\langle\Psi_m(x)\rangle$, where $\Psi_m(x)$ is the m^{th} cyclotomic polynomial with degree $n = \varphi(m)$[1]. The decreasing moduli sequence q_i is selected as p^{d-i} where p is a prime that can handle $\log q$ bits of noise.
- The special form of $q_i = p^{d-i}$ is used to reduce the memory requirement significantly. We can recycle the evaluation key $\zeta_\tau^{(0)}(x)$ of the first level for all levels by evaluating $\zeta_\tau^{(i)}(x) = \zeta_\tau^{(0)}(x) \pmod{q_i}$.
- The special selected cyclotomic polynomial $\Psi_m(x)$ is used to batch multiple message bits into the same polynomial for parallel evaluations as proposed by Smart and Vercauteren [8], [13] (see also [9]). The polynomial $\Psi_m(x)$ is factorized over $\mathbb{F}_2$ into equal degree polynomials $F_i(x)$ which define the message slots in which message bits are embedded using the Chinese Remainder Theorem. We can batch $\ell = n/t$ number of messages where t is the smallest integer that satisfies $m|(2^t-1)$.

B. GPU NTRU Library

In this section we present an overview of our Nvidia GPU library. The library supports three primitive operations: large polynomial addition and multiplication, and relinearization. The operands are elements of $\mathrm{R}_q = \mathbb{Z}_q[x]/\langle\Psi_m(x)\rangle$. We instantiate the NTRU scheme with two parameter choices to satisfy our homomorphic evaluation needs: $(n, \log(q)) = (2^{14}, 575)$ or $(n, \log(q)) = (2^{15}, 1271)$. That is, in the largest parameter selection (or keys in the topmost circuit level in homomorphic evaluation), we need to perform in the order of 1,271 modular multiplications of polynomials of degree 2^{15} with 1,271-bit coefficients. Thus, we are computing with exceptionally *large* data objects. Furthermore, during homomorphic circuit evaluation, due to *modulus reduction* [6], the modulus q (which determines the size of the polynomial coefficients) gradually shrinks. Therefore, we need to design a flexible GPU library to optimally support the critical operations needed for homomorphic evaluation for the given architecture (number of cores and memory). We achieve this goal by carefully combining a number of common techniques as follows.

1) CRT Conversion: As an initial optimization we convert all operand polynomials with large coefficients into many polynomials with small coefficients by a direct application of the Chinese Remainder Theorem (CRT).

$$\texttt{CRT} : x \longrightarrow \{x \bmod p_0, x \bmod p_1, \cdots, x \bmod p_{l-1}\}$$

Through CRT conversion we obtain two sets of polynomials $\{A_0, A_1, \cdots, A_{l-1}\}$ and $\{B_0, B_1, \cdots, B_{l-1}\}$ where $A_i, B_i \in \mathrm{R}_{p_i} = \mathbb{Z}_{p_i}[x]/\langle\Psi_m(x)\rangle$. Complex operations such as polynomial multiplication may now be computed using the CRT representation with much smaller coefficients. Also large integer arithmetic operations are only needed in CRT and Inverse CRT (ICRT) computations. The CRT conversion creates parallel execution paths where we can explore tradeoffs. For instance, if we choose larger primes, then the number of prime numbers l will be smaller and vice versa. In our implementation, the size and number of prime numbers p_i's are decided automatically, mostly according to the degree and coefficient size of polynomial. The product of those prime numbers should be larger than the potentially largest coefficient of polynomial c that we will obtain as a result of a computation for accurate recovery through ICRT. We will briefly summarize

[1]The operation $\varphi(m)$ is the euler totient function.

the constrains on the primes later under ring multiplication and relinearization sections. A side benefit of using CRT is that it allows us to accommodate the change in the coefficient size during the levels of evaluation thereby yielding more flexibility. When the circuit evaluation level increases, since q gets smaller, we can simply decrease the number of primes l. The code automatically changes the size and number of prime numbers during the evaluation. Therefore, both multiplication and relinearization become faster as we proceed through the levels of evaluation.

At the end of the computations we compute the Inverse CRT. We modify the original ICRT routine slightly to obtain improved

$$\mathsf{ICRT}(x) = \sum_{i=0}^{l-1} \left(\frac{M}{p_i}\right) \cdot \left(\left(\frac{M}{p_i}\right)^{-1} \cdot x_i \mod p_i\right) \mod M$$

We pre-compute M (the product of prime numbers), $\frac{M}{p_i}$ and $\frac{M}{p_i} \mod p_i$, and move it to the GPU constant memory. In each iteration of the summation the term contributed is only as large as M. Therefore, we can perform the reduction with only a single conditional subtraction, $x = x - M$ if $x > M$ after each iteration.

2) Polynomial Multiplication: The key primitive for achieving decent performance in the NTRU GPU library is polynomial multiplication. Besides supporting the homomorphic evaluation of AND operations, polynomial modular multiplication is also the dominant operation in Relinearization. Since we are multiplying polynomials with very high degree, an NTT based polynomial multiplication algorithm similar to the Strassen's NTT based integer multiplication algorithm [28] is crucial to achieve reasonable performance. Emmart and Weems [29] present an NTT based very large integer multiplication algorithm that yields high performance on a GPU platform. The algorithm uses four-step Cooley-Tukey NTT iterations [30] and makes use of a prime modulus of special form $P =$ `0xffffffff00000001` to construct the NTT operation over the finite field $\mathbb{Z}/P\mathbb{Z}$. Since it supports the NTT lengths we use the same prime to construct the NTT. However, in contrast to [29] we achieve polynomial multiplication instead of integer multiplication. In the implementation we follow these steps:

- We first double the degree of the polynomials a, b to $2n$ by padding with 0 coefficients.
- Then we treat the coefficients of the polynomials as n-point sequences, and perform $2n$-point NTT on them to obtain A, B.
- We compute the component-wise product: $C = A \cdot B$.
- Finally, we compute the Inverse NTT: $c = \mathsf{INTT}(C)$, to recover the desired polynomial product $c = a \cdot b$.

The resulting large polynomial multiplication algorithm is summarized in Algorithm 1. The NTT conversion significantly improves the parallelism of the algorithm.

Recall that we have split the polynomial operands with large coefficients $a, b \in \mathrm{R}_q$ into l polynomials with small coefficients $A_i, B_i \in Z_{p_i}$ using CRT. To make the multiplication algorithm work with the CRT conversion and the NTT

Algorithm 1 Polynomial Multiplication

Input: Polynomials a, b with $(n, \log(q))$
Output: Polynomial c with $(2n, \log(nq^2))$
1: $\{a_i\} = \mathsf{CRT}(a), \{b_i\} = \mathsf{NTT}(b)$
2: $\{A_i\} = \mathsf{NTT}(\{a_i\}), \{B_i\} = \mathsf{NTT}(\{b_i\})$
3: $\{C_i\} = \{A_i\} \cdot \{B_i\}$
4: $\{c_i\} = \mathsf{INTT}(\{C_i\})$
5: $c = \mathsf{ICRT}(\{c_i\})$

operation w.r.t. the prime P, the following constraints need to be satisfied to prevent overflow:

- P needs to be larger than any largest possible coefficient in a polynomial product $P > n \cdot p_i^2$, and
- The range of the CRT should cover the largest possible product value, i.e. $\prod_{i=0}^{l-1} p_i > n \cdot q^2$.

This shows that, in the polynomial multiplication the size of the prime numbers is limited by the constant P and the polynomial degree n. The number of primes l is limited by both n and the coefficient size q. Thus, during homomorphic evaluation when we move to a new multiplicative level, since q is getting smaller, all we need to do is to decrease l to further speedup the multiplication to take advantage of the shrinking q.

We have implemented all of the required arithmetic operations over 64-bit or 32-bit integers implemented on an Nvidia GPU in Assembly code. All addition and multiplication operations are carried out in $\mathbb{Z}/P\mathbb{Z}$. Thanks to the special form of the prime number P, modular reduction can be realized using only shifts, additions and subtractions. Note that, the polynomial operands are too large for the constant memory to hold. Therefore, we have to store them in the slower global memory. The access speeds of GPU data from fastest (small) to slowest (large) is as follows register, shared memory, and global memory. We design the CUDA kernels as follows: Each thread loads data from global memory to registers. For NTT we actually only load half of the data, since half of the coefficients are zeros. A buffer using shared memory is created for blocks to store the data. We primarily use registers for computations. Therefore, we only access the global memory when we load data at the beginning of the computation and when we write back data once the computation is completed. Data addressing is arranged carefully in the way that the program always has *coalesced* global memory access. Using Bailey's NTT technique [31], for $2n > 4096$, the first two CUDA kernels deal with $(2n/4096)$ 4096-point NTT's. The third kernel computes 4096 $2n/4096$-point NTT's and transposes it when writing back the data. Furthermore, since the value of n varies from scheme to scheme (e.g. 2^{15} in AES, 2^{14} in Prince), we change the memory mapping code in all kernels and the entire third kernel according to the parameters. If we add coefficient permutation to the first kernel and multiply each coefficient with $(2n)^{-1} \mod P$, we just have the INNT functions.

3) Relinearization: The relinearization primitive has two inputs: the ciphertext (a polynomial) and a set of evaluation keys which consist of $\lceil \log(q) \rceil$ polynomials. Each of these polynomials has a high degree n and a large coefficient size q. The implemented relinearization operation is shown in Algorithm 2.

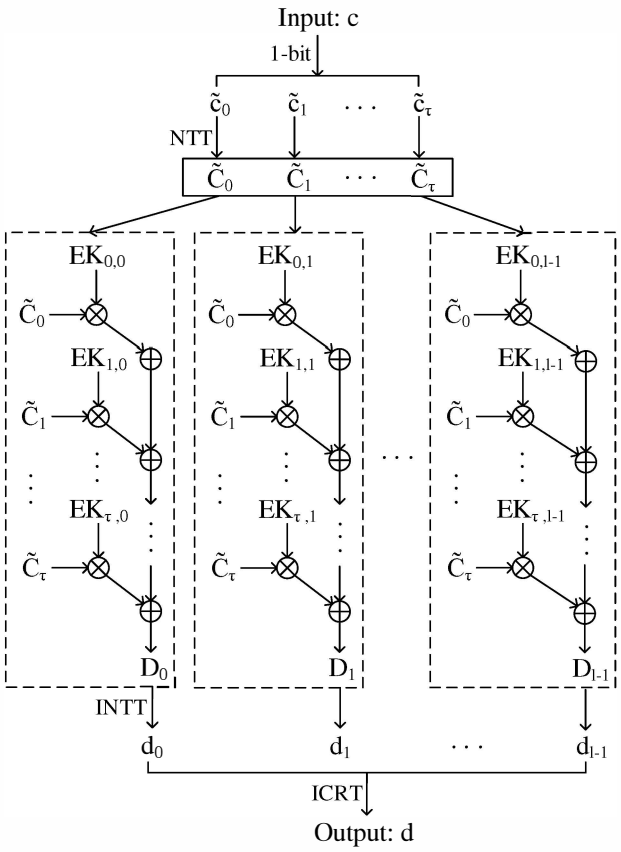

Fig. 1. Flowchart showing processing steps during relinearization

Algorithm 2 Relinearization

Input: Polynomial c with $(n, \log(q))$
Output: Polynomial d with $(2n, \log(nq\log(q)))$
1: $\{\tilde{C}_\tau\} = \mathsf{NTT}(\{\tilde{c}_\tau\})$
2: **for** $i = 0 \ldots, l-1$ **do**
3: load $EK_{i,0}, EK_{i,1}, \cdots, EK_{i,\lceil \log(q) \rceil - 1}$
4: $\{D_i\} = \{\sum_{\tau=0}^{\lceil \log(q) \rceil - 1} \tilde{C}_\tau \cdot EK_{i,\tau}\}$
5: **end for**
6: $\{d_i\} = \mathsf{INTT}(\{D_i\})$
7: $d = \mathsf{ICRT}(\{d_i\})$

As in polynomial multiplication, the relinearization operation imposes restrictions on the CRT parameter selection as follows:

- The range of CRT should cover the largest possible result of relinearization: $\prod_{i=0}^{l-1} p_i > \lceil \log(q) \rceil \cdot n \cdot q$ and
- $P > \lceil \log(q) \rceil \cdot n \cdot p_i$.

Relinearization involves $\lceil \log(q) \rceil$ polynomial multiplications and additions. Additions are coefficient wise, which means $2n$ parallel threads can solve it efficiently. In a level of homomorphic evaluation since we will be using the evaluation keys multiple times, we perform a CRT conversion on each obtaining $(\lceil \log(q) \rceil \cdot l)$ polynomials with small coefficients. Then we perform NTT on each and store the result in memory. In each level, we have a new set of keys. If we were to convert the evaluation keys for each relinearization, it would slow down the computation. Usually we do not increase the level until we finish all relinearizations in the current level. Therefore, it is better to generate evaluation keys in CRT and NTT domains when we move to a new level and store them for later use. Even if we do switch a few levels at a time, we can store the converted evaluation keys of all those levels when we have enough space (as in our implementation of PRINCE), or we can use the setup from the topmost level (as in our implementation of AES). The converted keys take a huge amount of space. In first level of AES evaluation, with $(n, \log(q)) = (32768, 1271)$, converted keys consume approximately 23 GBytes memory. Since we are switching four levels at a time, we use the converted keys of the lowest level. In first level of the Prince circuit evaluation, with $(n, \log(q)) = (16384, 575)$, the converted keys consume only 2 GBytes of memory. We can keep converted keys for several levels in memory, and switch between them when needed. As for the ciphertext representation, since we are converting it into a set of $\lceil \log(q) \rceil$ polynomials with 1-bit coefficients, CRT is not needed. This saves time and space. We perform NTT on the set of polynomials and keep them in GPU global memory.

Next the component-wise product is computed. Since evaluation keys are too large to fit into GPU memory, we copy a part of them from host memory to GPU memory and compute the sum of their coefficient-wise product with the ciphertext. Except for the last levels that use small keys, in relinearization most of the time is spent in copying the converted evaluation keys to GPU memory. We allocate page-locked host memory, which has the fastest transfer speed (according to our tests) to store converted evaluation keys. INTT is performed on the computation result of each CRT prime residue. We also test an alternative method: performing INTT after every component-wise multiplication and then add them up. The latter method supports larger sizes and requires fewer primes than the former one. In this way the key size and loops in the code are reduced, but in a quite limited scale. However, it needs $\lceil \log(q) \rceil \cdot l$ INTT's while the former one needs only l INTT's. As a result of test, the former method has better performance. Then after ICRT we obtain the relinearization result with polynomial degree and coefficient size $(2n, \log(q))$.

4) CUDA Library and NTL Library: Space for polynomials are allocated by the NTL Library. When we pass them to GPU, we use a 1-D array with $n \cdot \frac{\lceil \log(q) \rceil}{32}$ unsigned integers. CUDA handles arrays efficiently, and with regular indexing memory copy operations and accesses are very fast. Data type conversion between the NTL Library and unsigned integer takes time but does not slow the program by much. Also the output of polynomial multiplication or relinearization is not in $\mathrm{R}_q = \mathbb{Z}_q[x]/\langle \Psi_m(x) \rangle$ since modular reduction is delayed to take place after the result has been transferred to the CPU. While in relinearization, even though we are multiplying polynomials, one of the multipliers has 1-bit coefficients.

III. Homomorphic Evaluation

Here we put our GPU library to use to homomorphically evaluate two example block ciphers: Prince and AES using the NTRU based FHE cipher [2], [10], [32].

A. Evaluation of the Prince Block Cipher

Prince is a lightweight block cipher that has been optimized for small hardware footprint [33]. The homomorphic encryption is done by using a 128-bit key on a 64-bit message that are encrypted in homomorphic domain. The message is transformed into a 4×4 matrix where each element is a nibble. Prince has 12 rounds of substitution-permutation network with 4-bit S–Boxes, shift rows and mix columns operations. Each round uses the same key with a different 64-bit round constant to create variations among round keys. Prince has the following four operations:

1) Key Schedule: The key is split into two halves $\{k_0, k_1\}$. The first half of the key k_0 is XORed for only one time right before starting the round operations. In each round operation, the second half of the key k_1 is XORed at the end. After the round operations are completed, k_0 is transformed into $k_0' = (k_0 \gg 1) \oplus (k_0 \gg 63)$ and XORed.

2) Round Constant Addition: The round constants RC_i are XORed at each round, where i is the round number. The round constants hold the following property $RC_i \oplus RC_{11-i} =$ `0xc0ac29b7c97c50dd`.

3) S–Box: The S–Box is the only operation with nonlinearity in Prince. It computes a 4-bit to 4-bit mapping. We use the original S–Box parameters as in [33] and implement a 2-depth logic circuit to compute the mapping.

4) Linear Layer: The linear layer consists of two steps: shift rows and mix column. The shift row operation reorders the rows of the input message matrix by swapping them. In mix column, each output bit is computed by XORing three input bits.

We use all the operations explained above in all rounds except the S–Box and Linear Layer. They have their own inverse versions that are used in the last six rounds instead of the S–Box and Linear Layer. However this does not effect the circuit depth in each round, thus we can evaluate Prince in 24 levels.

B. Evaluating AES

In the homomorphic evaluation of AES, we take the AES keys and the individually encrypted messages as inputs. As in standard AES with 128-bit keys, homomorphic AES is evaluated in 10 rounds, where the message is transformed into a 4×4 matrix and the operations are divided into four steps:

1) AddRoundKey: The round keys are precomputed, encrypted and given alongside the encrypted AES keys and message. Since a round operation has depth 4 circuit, the round key for level i is computed in $\mathrm{R}_{q_{4i}}$ for $0 \leq i \leq 10$. The first round key is XORed as the computation starts and the rest are XORed at the end of each round operation.

2) ShiftRows: The shifting of rows is a simple operation that only requires swapping of indices trivially handled in the code. This operation has no effect on the noise.

3) MixColumns: In the Mix Column operation a byte in the message matrix is multiplied with one of the constant terms of $\{x+1, x, 1\}$ in modulo $(x^8 + x^4 + x^3 + x + 1)$. These products are evaluated by simple XORs and shifts. Once the products of the rows are finished, the four values are added to each other. The addition operations add a few bits of noise.

4) SubBytes: The SubBytes step (S–Box) is the only place where homomorphic multiplications and Relinearization takes place. The S–Box conversion is evaluated for each byte b of the $e \times 4$ matrix as $s = Mb^{-1} \oplus B$. In [34], the authors introduce an efficient inversion implementation by converting the input from $GF(2^8)$ into a tower field representation $GF(((2^2)^2)^2)$ using an isomorphism. This is the S-box circuit we implement in this work. The full 10 round 128 bit-AES block homomorphic evaluation requires the evaluation of a depth 40 circuit.

IV. Implementation Results

In our experiments, we use a server equipped with Intel Xeon E5-2609 running @2.5 Ghz, 64 GBytes of memory, and a NVIDIA GeForce GTX 690 running @915 Mhz with 3072 stream processors and 4 GBytes of memory[2]. We ran the experiments using a single thread of the Intel Xeon processor and a single GTX 680 graphics processor. In terms of software setup, we used Ubuntu 12.04, Cuda platform 6.0 and compiled our code using Shoup's NTL 6.1 [35] library linked with GMP 6.0 .

In Table I we give the timing results for relinearization and multiplication operations for each block cipher settings. The timings are given for the functions and do not contain the data type conversion from NTL. In multiplication we have two CRT and NTT conversions, so they are given in $\times 2$ form. In terms of relinearization we only use one input and in CRT step the inputs are in binary polynomial form which takes negligible time to compute. In comparison of the timings; in Prince, where n and $\log q$ are half of AES, the multiplication is $7.2\times$ and the relinearization is $9.6\times$ faster than AES.

TABLE I. Performance of Multiplication and Relinearization in Prince and AES implementations.

	Multiplication	
	Prince	AES
CRT$\times$2 (msec)	5.70	49.6
NTT$\times$2 (msec)	9.00	39.8
MULT (msec)	0.31	1.2
I-NTT (msec)	5.20	22.4
I-CRT (msec)	12.2	121
Total (sec)	**0.0325**	**0.2340**

	Relinearization	
	Prince	AES
CRT$\times$1 (msec)	n/a	n/a
NTT$\times$1 (msec)	53	256
MULT (msec)	833	8300
I-NTT (msec)	1.5	7.2
I-CRT (msec)	3.5	40.7
Total (sec)	**0.89**	**8.60**

We compare our GPU performance with the CPU performance of our server in Table II. The results clearly shows that in a simpler arithmetic operation such as multiplication, the gain of GPU is only a factor of 2.8. However for relinearization, which requires much heavier computation, the gain is more than an order of magnitude, i.e. ~ 12 times.

Finally, we compare our results with Prince in [32] and AES in [9], [10] in Table III. In AES, we reach a runtime of 4.15 hours which gives us 7.3 seconds batched per block AES evaluation time. Compared to the Byte and SIMD techniques of [9] we achieve a speedup factors of $328\times$ and $41\times$, respectively. When we compare our result with [10], where the authors use the same scheme and AES circuit as ours,

[2]The NVIDIA GeForce GTX 690 is formed of dual GTX 680's. Therefore the given resources are split into two equal halves for each GTX 680.

TABLE II. TIMING COMPARISON BETWEEN THE CPU AND GPU IMPLEMENTATIONS FOR THE OPERATIONS.

	Prince			AES		
	GPU	CPU	SPEEDUP	GPU	CPU	SPEEDUP
MULTIPLICATION	0.063	0.18	$\times$ **2.8**	0.34	0.97	$\times$**2.8**
RELINEARIZATION	0.89	10.9	$\times$**12.2**	8.97	103.37	$\times$**11.5**

we are $7.6\times$ faster. Unfortunately for Prince the speedup is limited to $2.57\times$ since Prince has a higher ratio of arithmetic operations such as additions, and swaps than Multiplications and Relinearizations. Thus, it does not benefit as much by GPU acceleration as AES does.

TABLE III. PERFORMANCE COMPARISON OF PRINCE AND AES IMPLEMENTATIONS.

		TOTAL TIME	#BLOCKS	PER BLOCK	Speedup
AES	SIMD Xeon [9]	36 h	54	2400 sec	$\times$1
	Byte Xeon [9]	65 h	720	300 sec	$\times$8
	NTRU Xeon [10]	31 h	2048	55 sec	$\times$43
	Ours (GPU)	**4.15 h**	**2048**	**7.3 sec**	$\times$**328**
Prince	Prince [32]	57 min	1024	3.3 sec	$\times$1
	Ours (GPU)	**22 min**	**1024**	**1.28 sec**	$\times$**2.57**

ACKNOWLEDGMENT

This work was supported by NSF Awards #1117590 and #1319130.

REFERENCES

[1] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[2] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *STOC*, 2012.

[3] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *EUROCRYPT*, 2010, pp. 24–43.

[4] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi, "Fully homomorphic encryption over the integers with shorter public keys," in *CRYPTO*, 2011, pp. 487–504.

[5] J.-S. Coron, D. Naccache, and M. Tibouchi, "Public key compression and modulus switching for fully homomorphic encryption over the integers," in *EUROCRYPT*, 2012, pp. 446–464.

[6] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," in *FOCS*, 2011, pp. 97–106.

[7] C. Gentry, S. Halevi, and N. P. Smart, "Better bootstrapping in fully homomorphic encryption," *IACR Cryptology ePrint Archive 2011/680*, vol. 2011, 2011.

[8] ——, "Fully homomorphic encryption with polylog overhead," *IACR Cryptology ePrint Archive Report 2011/566*, 2011, http://eprint.iacr.org/.

[9] ——, "Homomorphic evaluation of the AES circuit," *IACR Cryptology ePrint Archive*, vol. 2012, 2012.

[10] Y. Doröz, Y. Hu, and B. Sunar, "Homomorphic AES evaluation using NTRU," Cryptology ePrint Archive, Report 2014/039, 2014, http://eprint.iacr.org/.

[11] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," in *EUROCRYPT*, 2011, pp. 129–148.

[12] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping," *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 18, p. 111, 2011.

[13] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *IACR Cryptology ePrint Archive*, vol. 2011, p. 133, 2011.

[14] Z. Brakerski, C. Gentry, and S. Halevi, "Packed ciphertexts in LWE-based homomorphic encryption," *IACR Cryptology ePrint Archive*, vol. 2012, p. 565, 2012.

[15] J.-S. Coron, T. Lepoint, and M. Tibouchi, "Batch fully homomorphic encryption over the integers," *IACR Cryptology ePrint Archive*, vol. 2013, p. 36, 2013.

[16] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using GPU," in *HPEC*, 2012, pp. 1–5.

[17] D. Cousins, K. Rohloff, R. Schantz, and C. Peikert, "SIPHER: Scalable implementation of primitives for homomorphic encrytion," Internet Source, September 2011.

[18] D. Cousins, K. Rohloff, C. Peikert, and R. E. Schantz, "An update on SIPHER (scalable implementation of primitives for homomorphic encRyption) - FPGA implementation using simulink," in *HPEC*, 2012, pp. 1–5.

[19] C. Moore, N. Hanley, J. McAllister, M. O'Neill, E. O'Sullivan, and X. Cao, "Targeting FPGA DSP slices for a large integer multiplier for integer based FHE," *Workshop on Applied Homomorphic Cryptography*, vol. 7862, 2013.

[20] X. Cao, C. Moore, M. O'Neill, N. Hanley, and E. O'Sullivan, "Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction," *Under Review*, 2013.

[21] W. Wang and X. Huang, "FPGA implementation of a large-number multiplier for fully homomorphic encryption," in *ISCAS*, 2013, pp. 2589–2592.

[22] Y. Doröz, E. Öztürk, and B. Sunar, "Evaluating the hardware performance of a million-bit multiplier," in *Digital System Design (DSD), 2013 16th Euromicro Conference on*, 2013.

[23] ——, "Accelerating fully homomorphic encryption in hardware," 2013, draft, Under Review. [Online]. Available: http://ecewp.ece.wpi.edu/wordpress/vernam/files/2013/09/Accelerating-Fully-Homomorphic-Encryption-in-Hardware.pdf

[24] C. Gentry and S. Halevi, "Fully homomorphic encryption without squashing using depth-3 arithmetic circuits," *IACR Cryptology ePrint Archive*, vol. 2011, p. 279, 2011.

[25] N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *Public Key Cryptography*, 2010, pp. 420–443.

[26] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 99, no. PrePrints, p. 1, 2013.

[27] D. Stehlé and R. Steinfeld, "Making ntru as secure as worst-case problems over ideal lattices," *Advances in Cryptology – EUROCRYPT '11*, pp. 27–4, 2011.

[28] D. D. A. Schönhage and V. Strassen, "Schnelle multiplikation grosser zahlen," *Computing*, vol. 7, no. 3-4, pp. 281–292, 1971.

[29] N. Emmart and C. C. Weems, "High precision integer multiplication with a gpu using strassen's algorithm with multiple fft sizes," *Parallel Processing Letters*, vol. 21, no. 03, pp. 359–375, 2011.

[30] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Math. comput*, vol. 19, no. 90, pp. 297–301, 1965.

[31] D. H. Bailey, "Ffts in external of hierarchical memory," in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. ACM, 1989, pp. 234–242.

[32] Y. Doröz, A. Shahverdi, T. Eisenbarth, and B. Sunar, "Toward practical homomorphic evaluation of block ciphers using prince," Cryptology ePrint Archive, Report 2014/233, 2014, http://eprint.iacr.org/.

[33] J. Borghoff and et al., "PRINCE, a low-latency block cipher for pervasive computing applications," in *ASIACRYPT 2012*, ser. LNCS, X. Wang and K. Sako, Eds. Springer Berlin Heidelberg, 2012, vol. 7658.

[34] D. Canright, "A very compact S-Box for AES," in *Cryptographic Hardware and Embedded Systems CHES 2005*, ser. Lecture Notes in Computer Science, J. R. Rao and B. Sunar, Eds. Springer Berlin Heidelberg, 2005, vol. 3659, pp. 441–455.

[35] V. Shoup, NTL: A Library for doing Number Theory. [Online]. Available: http://www.shoup.net/ntl/