Exploring the Feasibility of Fully Homomorphic Encryption

Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, Berk Sunar
Worcester Polytechnic Institute

Abstract—In a major breakthrough, Gentry introduced the first plausible construction of a fully homomorphic encryption (FHE) scheme in 2009. FHE allows the evaluation of arbitrary functions directly on encrypted data on untrusted servers. Later, in 2010 Gentry-Halevi presented the first FHE implementation. However, even for the small setting with 2,048 dimensions, the authors reported a performance of 1.8 seconds for a single bit encryption and 92 seconds for recryption on a high end server. Much of the latency is due to computationally intensive multi-million-bit modular multiplications.

In this paper, we adopt Strassen’s FFT-based algorithm for large number multiplication and employ a novel precomputation technique along with delayed modular reductions which allows us to carry out the bulk of computations in the frequency domain. We manage to eliminate all FFT conversion except the final inverse transformation drastically reducing the computation latency for all FHE primitives. In addition, we realize the FHE scheme on a GPU to further speed up the operations. Our experimental results with small parameter setting show speedups of 174, 7.6 and 13.5 times for encryption, decryption and recryption, respectively, when compared to the Gentry-Halevi implementation.

Index Terms—fully homomorphic encryption, GPU, large-number multiplication, modular reduction

1 INTRODUCTION

In the past decade, one of the most significant advances in cryptography has been the introduction of the first fully homomorphic encryption scheme (FHE) by Gentry [1]. This advance not only solved an open problem posed by Rivest [2], but also opened the door to many new applications. Indeed, using a FHE one may perform an arbitrary number of computations directly on the encrypted data without revealing of the secret key. Thus an untrusted party, such as a remotely hosted server, may perform computations on behalf of the owner on the data without compromising privacy. This property of FHE is precisely what makes it invaluable for the cloud computing industry today. For instance, if FHE is recognized early on in [1] that FHE is ideally suited to protect sensitive data on untrusted cloud servers. Considering the recent growth in the adoption of cloud services, it is foreseeable that FHE schemes will have a transforming effect on personal computing in the coming years.

Despite its promise, FHE is nowhere near ready for real-life deployment due to serious efficiency impediments. The first implementation of an FHE variant was proposed by Gentry and Halevi [3], who presented an impressive array of optimizations with the goals of reducing the size of the public-key and improving the performance of the primitives. Still, encryption of one bit takes more than a second on a high-end Intel Xeon based server, while recrypt primitive takes nearly half a minute for the lowest security setting. Furthermore, after every few bit-AND operations a recrypt operation must be applied to reduce the noise in the ciphertext to a manageable level. When application specific FHE hardware is considered, the situation becomes even worse. In [4], an FPGA implementation draft for improving the speed of FHE primitives was proposed. However, no implementation results were presented. Clearly, much work is needed before FHE becomes practically useful.

In this paper, we take another step in this direction. We present a GPU acceleration of the FHE variant introduced by Gentry and Halevi [3]. Our implementation shows significant improvement in speed over the Gentry-Halevi CPU implementation. Since GPU based cloud computing services are already available, e.g. on Amazon’s EC2 cluster GPU instances, our approach is well supported on existing cloud computing platforms. The GPU approach also makes sense when one considered the rapid progression in the processor industry. With continuous architectural improvements in recent years, GPUs have evolved into a massively parallel, multithreaded, many-core processor system with tremendous computational power. Owing to introduction of the Compute Unified Device Architecture (CUDA) programming paradigm, a vast of computation problems outside of the graphics domain have benefited from the superior performance of GPUs. Among the examples of the general purpose GPU (GPGPU) computing initiative are FFT [5], data processing [6] and many other science and engineering applications [7].

In this work, we present an array of algorithmic optimizations to the Gentry-Halevi FHE algorithm [3]. More specifically, we reformulate the iterations to delay the modular reductions in the implementation of the encryption and recryption primitives to eliminate costly back and forth conversions normally experienced during Strassen’s FFT based integer multiplication algorithm. Thus, we are performing almost all of the computations...
in the FFT domain. Modular reductions are still required in the final step of the implementations. For this we utilize Barrett’s modular reduction algorithm. We achieve further improvements by choosing a GPU platform that matches the computational needs of the Gentry-Halevi FHE. For the small parameter setting, we achieve a 13.5-fold speedup for the most critical primitive, i.e., recryption, and 174-fold and 7.6-fold speedup in the cases of encryption and decryption, respectively, on the NVIDIA GTX690 over the CPU implementation on Intel i7 3770K at 3.5 GHz.

The rest of the paper is organized as follows: Section II briefly introduces the Gentry-Halevi FHE; Section III provides the design and performance of large number adder and subtraction algorithm; a large-number modular multiplier design is presented in Section IV; the complete GH-FHE GPU implementation is discussed in Section V which is followed by the experiment results in Section VI and conclusions in Section VII.

2 GENTRY’S FULLY HOMOMORPHIC ENCRYPTION

Informally a homomorphic encryption scheme refers to an encryption function that allows one to induce a binary operation on the plaintexts while only manipulating the ciphertexts without the knowledge of the encryption key: $E(x_1) \star E(x_2) = E(x_1 \oplus x_2)$. If the scheme supports the efficient homomorphic evaluation of any efficiently computable function, it is called a fully homomorphic encryption scheme (FHE). With FHE, an honest but curious party can perform any computation directly with encrypted result without gaining access to the plaintext.

The first FHE was proposed by Gentry in [1], [8]. However, this preliminary implementation is far too inefficient to be used in any practical applications. The Gentry-Halevi FHE variant with a number of optimizations and the results of a reference implementation were presented in [3]. Here we only present a high-level overview of the primitives and the details can be referred to the original work in [3].

Encrypt: To encrypt a bit $b \in \{0, 1\}$ with a public key $(d, r)$, Encrypt first generates a random “noise vector” $u = \langle u_0, u_1, \ldots, u_{n-1} \rangle$, with each entry chosen as 0 with some probability $p$ and as ±1 with probability $(1-p)/2$ each. Clearly, $p$ will determine the hamming weight of the random noise $u$. Gentry showed in [3] that $u$ can contain a large number of zeros without impact the security level, i.e., $p$ could be very large.

Then the message bit $b$ is encrypted by computing

$$c = [b + u \cdot r]_d = \left[ b + 2 \sum_{i=1}^{n-1} w_i r^i \right]_d$$

(1)

where $d$ and $r$ is part of the public key.

Eval: When encrypted, arithmetic operations can be performed directly on the ciphertext with corresponding modular operations. Suppose $c_1 = \text{Encrypt}(m_1)$ and $c_2 = \text{Encrypt}(m_2)$, we have:

$$\begin{align*}
\text{Encrypt}(m_1 + m_2) &= (c_1 + c_2) \mod d \\
\text{Encrypt}(m_1 \cdot m_2) &= (c_1 \cdot c_2) \mod d .
\end{align*}$$

(2)

Decrypt: An encrypted bit can be recovered from a ciphertext $c$ by computing

$$m = [c \cdot w]_d \mod 2$$

(3)

where $w$ is the private key and $d$ is part of the public key.

Recrypt: The Recrypt process is realized by homomorphically evaluating the decryption circuit on the ciphertext. However, due to the fact that we can only encrypt a single bit and that we can only evaluate a limited number of arithmetic operations, we need an extremely shallow decryption method. In [3], the authors discussed a practical way to re-organize the decryption process to make this possible. Informally, the private key is divided into $s$ pieces that satisfy $\sum s_i w_i = w$. Each $w_i$ is further expressed as $w_i = x_i R^l_i \mod d$ where $R$ is constant, $x_i$ is random and $l_i \in \{1, 2, \ldots, S\}$ is also random. The recryption process can then be expressed as:

$$m = [c \cdot w]_d \mod 2 = \left( \sum_i S_i c_{x_i} R^{l_i} \right)_d \mod 2$$

$$= \left[ \sum_i S_i c_{x_i} R^{l_i} \right]_2 \cdot \left[ \left( \sum_i S_i c_{x_i} R^{l_i} \right) / d \right]_2 \cdot d \mod 2$$

(4)

The Encrypt process can then be divided into two parts. First we compute the sum of $c_{x_i} R^{l_i}$ for each “block” $i$. To further optimize this process, encode $l_i$ to a 0 – 1 vector $\langle \eta^{(i)}_1, \eta^{(i)}_2, \ldots, \eta^{(i)}_{m_i} \rangle$ where only two elements are “1” and all other elements are “0”s. Suppose the two positions are labeled as $a$ and $b$. We write $l(a, b)$ to refer to the corresponding value of $l$. Alternatively we can obtain $c_{x_i} R^{l_i}$ from

$$c_{x_i} R^{l_i} = \sum_a \eta^{(i)}_a \sum_b \eta^{(i)}_b c_{x_i} R^{l(a, b)}.$$  (5)

Obviously, only when $\eta^{(i)}_a$ and $\eta^{(i)}_b$ are both “1”, the corresponding $c_{x_i} R^{l(a, b)}$ is selected. In addition, if we encode $l_i$ in a way that each iteration only increases it by 1, the next factor $c_{x_i} R^{l(a, b)}$ can be easily computed by multiplying $R$ to the result of the previous computation.

After applying these modifications, all operations involved in this formulation of decryption become bit operations realizable by sufficiently shallow circuits. Thus we can evaluate this process homomorphically. The parameters $\eta_i$ are stored in encrypted form and incorporated into the public key.


3 Achieving Efficient Large Number Additions

3.1 Addition and Modular Addition

Large number addition has low arithmetic intensity, however, a traditional carry-ripple adder would result a very large carry-chain that is not efficient for parallel processing. Instead, carry-lookahead addition is often used in for high-speed addition [9]. For parallel processing on a GPU, large number addition is implemented using a carry-lookahead scheme similar to the one presented in [10]. A large number is first broken into \( m \) warps, each warp contains \( n \) words, and each word is a 32-bit integer. The addition is computed using two GPU kernels. In the first kernel, each warp performs an addition and all warps run in parallel threads. Each warp outputs 3 values: the sum \( z_i \), a carry out bit \( c_i \) and a “critical flag” bit \( f_i \). The critical flag is set that if \( z_i \) are all 1’s, which indicates it will generate a carry out bit if there is a carry in from the adjacent lower warp. The second kernel resolves the whole carry chain for all warps. The second kernel does the hierarchical carry computation to determine the carry in for each chunk. If there is a carry in, then the chunk is incremented, rippling the carry through the chunk as needed.

<table>
<thead>
<tr>
<th>Size in K bits</th>
<th>Modular ADD on CPU</th>
<th>Modular ADD on GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.032 ms</td>
<td>0.016 ms</td>
</tr>
<tr>
<td>2048</td>
<td>0.069 ms</td>
<td>0.022 ms</td>
</tr>
<tr>
<td>4096</td>
<td>0.139 ms</td>
<td>0.051 ms</td>
</tr>
<tr>
<td>8192</td>
<td>0.526 ms</td>
<td>0.048 ms</td>
</tr>
<tr>
<td>16384</td>
<td>1.240 ms</td>
<td>0.082 ms</td>
</tr>
</tbody>
</table>

The modular addition operation, i.e. \( x + y \mod m \), is also implemented as a pair of kernels on a GPU. The first kernel computes \( x + y + m \) of the most significant 64 words of \( x, y \) and \( m \). If the result is positive, then we only need compute \( x + y + m \) of the full value. If the result is strongly negative, then we need only compute \( x - y + m \) of the full value. In the case that the sign of \( x - y \) cannot be determined by the top 64 words only, both \( x - y \) and \( x - y + m \) are computed and it leaves the second kernel to choose one of them. In the first kernel, each warp in the computation handles a chunk of the computation, ignoring the carries from less significant blocks. The second kernel handles all the inter-block carries to resolve the global carries. If the first kernel cannot determine whether \( x - y \) or \( x - y + m \) is to be calculated, the second kernel will choose the correct value after it resolves the global carries, which is similar to the method used in the modular addition. Table 1 lists the computing time of modular addition on CPU and GPU, respectively.

Table 2

<table>
<thead>
<tr>
<th>Size in K bits</th>
<th>Modular SUB on CPU</th>
<th>Modular SUB on GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.020 ms</td>
<td>0.0165 ms</td>
</tr>
<tr>
<td>2048</td>
<td>0.042 ms</td>
<td>0.023 ms</td>
</tr>
<tr>
<td>4096</td>
<td>0.111 ms</td>
<td>0.034 ms</td>
</tr>
<tr>
<td>8192</td>
<td>0.281 ms</td>
<td>0.054 ms</td>
</tr>
<tr>
<td>16384</td>
<td>0.579 ms</td>
<td>0.094 ms</td>
</tr>
</tbody>
</table>

4 Modular Multiplication of Large Numbers

4.1 The Schönhage-Strassen FFT Multiplication Algorithm

Large integer multiplication is by far the most time consuming operation in the FHE primitives. Therefore, it becomes the main target for acceleration. In [11], Strassen described a multiplication algorithm based on Fast Fourier Transform (FFT), which offers a good solution for effectively parallel computation of the large-number multiplication as shown in Fig. 1. The algorithm uses Fast Fourier transforms in rings with \( 2^k + 1 \) elements, i.e. a specialized number theoretic transform.
Briefly, the Strassen FFT algorithm can be summarized as follows:

1) Break large numbers $A$ and $B$ into a series of words $a(n)$ and $b(n)$ given a base $b$, and compute the FFT of the $A$ and $B$ series by treating each word as an sample in the time domain.

2) Multiply the FFT results, component by component: set $C[i] = FFT(A)[i] * FFT(B)[i]$.

3) Compute the inverse fast Fourier transform: set $c(n) = IFFT(C)$.

4) Resolve the carries: when $c[i] \geq b$, set $c[i+1] = c[i+1] + (c[i] \text{ div } b)$, and $c[i] = c[i] \mod b$.

### 4.2 Emmart and Weems’ Approach

In [12], Emmart and Weems implemented the Strassen FFT based multiplication algorithm on GPUs. Specifically, they performed the FFT operation in finite field $Z/pZ$ with a prime $p$ to make the FFT exact. In fact, they chose the $p = 0xFFFFFFFF00000001$ from a special family of prime numbers which are called Solinas Primes [13]. Solinas Primes support high efficiency modulo computations and this $p$ especially is ideal for 32-bit processors, which has also been incorporated into the latest GPUs. In addition, an improved version of Bailey’s FFT technique [14] is employed to compute the large size FFT. The performance of the final implementation is very promising. For the operands up to 16,320K bits, it shows a speedup factor of up to 16.7 when comparison with multiplication on the CPUs of the same technology generation.

We follow the implementation in [12] and test it on the GPU. As we can see from Table 3, the actual speedup factors are slightly different from [12]. Nevertheless, it is a significant speedup over the implementations achieved on CPUs. Therefore, we employ this specific instance of the Strassen FFT based multiplication algorithm in the FHE implementation.

We note however, the optimizations we later introduce virtually eliminate the need for FFT conversions except at the very beginning or the very end of the computation chains. The only exception is the decryption primitive which is implemented using a single modular multiplication. Therefore, while still necessary, the efficiency of the FFT operation has a negligible impact on the overall performance of encryption and recryption.

### 4.3 Modular Multiplication

Efficient modular multiplication is crucial for the decryption primitive. The other primitives only use modular reduction at the very end of the computations only once. Many cryptographic software implementations employ the Montgomery multiplication algorithm, cf. [15], [16]. Montgomery multiplication replaces costly trial divisions with additional multiplications. Unfortunately, the interleaved versions of the Montgomery multiplication algorithm generates long carry chains with little instruction-level parallelism. For the same reason, it is hard to realize Montgomery’s algorithm on parallel computing friendly GPUs. For example, a Montgomery multiplication implementation on GeForce 9800GX2 card was presented in [17]. The speedup factor of GPU decreased from 2.6 to 0.6 when the operand size increases from 160-bit to 384-bit, which showed little speedup if any can be achieved with large operand sizes. In addition, the underlying large integer multiplication algorithm we use is FFT based and optimized for very large numbers. Therefore, there does not seem to be any easy way to break it into smaller pieces. In conclusion, we implement modular multiplications without integrating the multiplication and reduction steps, but instead by executing them in sequence.

#### 4.3.1 Modular Reduction

The most popular algorithms for modular reduction are the Montgomery reduction [18] and the Barrett reduction algorithms [19]. As mentioned earlier, the interleaved Montgomery reduction algorithm cannot exploit the parallel processing on GPUs. The Barrett approach has a simpler structure and thus lends itself better for further
Algorithm 1. Barret reduction algorithm

1: procedure BARRETT(t, M) ▷ Output: $r = t \mod M$
2: $q \leftarrow 2 \lceil \log_2(M) \rceil$ ▷ Precomputation
3: $\mu \leftarrow \frac{M}{2^q}$ ▷ Precomputation
4: $r \leftarrow t - M \lfloor t \mu / 2^q \rfloor$
5: while $r \geq M$ do
6: $r \leftarrow r - M$
7: end while
8: return $r$ ▷ $r = t \mod M$
9: end procedure

Note that code from line 5 to line 7 is a loop. However, it can be shown that the initial $r$ for this loop is smaller than $3M - 1$. Therefore, this loop can finish quickly. In addition, the value $\mu = \frac{M}{2^q}$ can be precomputed to speed up the process. If multiple reductions are to be computed with the same modulus $M$, then this value can be reused for all reductions, which is exactly the case we have.

In addition, it would be advantageous to apply truncations only at multiples of the word size $w$ of the multiplier hardware (usually 32 bits) rather than at the original bit positions. In this case, we require $q$ to be a multiple of the word size $w$. With this approach, the division by $2^q$ can be easily implemented by discarding the least significant $q/w$ words.

5 Optimization of FHE Primitives

The FHE algorithm consists of four primitives: KeyGen, Encrypt, Decrypt and Recrypt. The KeyGen is only called once during the setup phase. Since keys are generated once and then preloaded to the GPU, the speed of KeyGen is not as important. Therefore we focus our attention to optimizing the other three primitives.

For the Decrypt primitive, we perform the computation as in 3. The flow is shown in Fig. 2. Obviously, the time spent in the primitive is equivalent to the time it takes to compute a single modular multiplication with large operands. Applying the FFT based Strassen algorithm and Barret reduction, which we discussed earlier, yields significant speedup for the Decrypt operation.

5.1 Optimizing Encrypt

To realize the Encrypt primitive, we need to evaluate a degree-$(n-1)$ polynomial $u$ at point $r$. In [3], a recursive approach for evaluating the 0-1 polynomial $u$ of degree $(n-1)$ at root $r$ modulo $d$. The polynomial $u(x) = \sum_{i=0}^{n-1} u_i r^i$ is split into a “bottom half” $u_{\text{bot}}(r) = \sum_{i=0}^{n/2-1} u_i r^i$ and a “top half” $u_{\text{top}}(r) = \sum_{i=0}^{n/2-1} u_i d/2^r$.

Then $y = r^{n/2} u_{\text{top}}(r) + u_{\text{bot}}(r)$ can be computed. The same procedure repeats until the remaining degree is small enough to be computed directly.

In our implementation, to fully exploit the power of pre-computation, we use a direct approach for polynomial evaluations. Specifically, we apply the sliding window technique to compute the polynomial. Suppose the window size is $w$ and we need $t = n/w$ windows, we compute:

$$\sum (u_i r^j) = \sum_{j=0}^{t-1} [r^{w-j} \cdot \sum_{i=0}^{w-1} (u_i w^j r^i)].$$

Here all additions and multiplications are evaluated modulo $d$. After organizing the computation as described above, we can introduce pre-computation to speed up the process. As $r$ is determined during KeyGen and therefore known apriori, the $r^i$, $i = 0, 1, \ldots, w$ values can be precomputed. In order to further reduce the overhead caused by the relatively slow communication between the CPU and the GPU, these precomputed values can be preloaded into GPU memory before the Encrypt process starts. Clearly, larger window size $w$ leads to fewer multiplications but an increased memory requirement. Hence, we have a trade-off between speed and memory use. In addition, as mentioned in previous sections, the majority of the coefficients of $u$ are zeros. It is possible that all the coefficients a window are zeros. In this case, we can skip the multiplication to further speed up the process.

In addition, as we use the FFT based algorithm to compute the multiplications, these pre-computed values can also be saved in FFT form. Since the FFT form is linear, we can directly evaluate additions in FFT domain. Therefore, the whole computation before the
final reduction can be performed in FFT domain:

$$\sum (u_r r^i) = IFFT(\sum_{j=0}^{31} [R^{64j} \cdot \sum_{i=0}^{63} (u_{i+64j} R^i)]) \mod d,$$

where $R$ is the precomputed FFT form of corresponding $r^i$. With this reformulation we eliminated almost all of the costly FFTs and IFFTs and modular reductions. Also as a side-benefit of staying in the FFT domain, carry propagations among words normally performed during addition operations are eliminated, which also contributes to the speed up.

In our implementation with dimension $n = 2048$, we choose the window size as $w = 64$. With this parameters, the CPU implementation of Encrypt runs in 1.08 seconds while our implementation on GPU the run time is significantly reduced to only 6.2 ms.

5.2 Implementing Recrypt

The Recrypt primitive is significantly more complicated. As mentioned earlier, Recrypt can be divided into two steps: processing of $S$ blocks and the computation of their sum. In the first step, the most time-consuming computation is as follows

$$cx_i R^i = \sum_a \eta^{(i)} \sum_{b} \eta^{(i)} cx_i R^{(a,b)}.$$

Here $\eta$ is part of the public key. If we encode the $l$ in a proper way such that each iteration it only increases by one, the next factor $cx_i R^{(a,b)}$ can be easily computed by multiplying $R$ with the result of the previous iteration. Here we refer to $cx_i R^{(a,b)}$ as the factor for each iteration. In each iteration, we update factor $\cdot R \mod d$ and determine whether we should sum $\eta$ or not. Since in this process $R$ is a small constant, the computation may even be performed on the CPU without any noticeable loss of efficiency in the overall scheme. Therefore, the CPU is used to compute the new factor value while the GPU is busy computing the additions from previous iteration. This approach allows us to run the CPU and the GPU concurrently and therefore harnessing the full computational power of the overall system.

The constants used in Recrypt are part of the public key. They can be precomputed to further speed up the computation. Similar to Encrypt, the public keys can be pre-loaded into the GPU memory to eliminate the latency incurred in CPU-GPU communications. In our implementation we targeted the small (security) setting, where the public key is about 140MB. The public key can perfectly fit into the GPU memory of the latest graphic cards. In fact, the public key will even fit into the GPU memory in the large setting, whose public key is about 2.25GB [3].

Furthermore, the majority of the computation in the Recrypt can also be represented as some “add-mul-add” chain as in the Encrypt. Therefore, the similar optimizations can be applied. The computation before the reduction can be performed in the FFT domain, reducing the number of expensive FFT and IFFT operations significantly. However, this optimization will also cause growth in public key size and make it impossible to store the whole public key in the GPU memory in the high dimension case. Fortunately, with such high dimension settings, the computation time is long enough to dwarf this extra communication overhead.

![Figure 3. Proposed Encrypt and Decrypt in the FFT domain](image)

6 IMPLEMENTATION RESULTS

We realized the Encrypt, Decrypt and Recrypt primitives of the Gentry-Halevi FHE scheme with the proposed optimizations on a machine with Intel Core i7 3770K running at 3.5 GHz with 8 GB RAM and a NVIDIA GTX 690 running at 1.02 GHz with 4GB memory. Only one GPU is used in this implementation. Shoup’s NTL library [20] is used for high-level numeric operations and GNU’s GMP library [21] for the underlying integer arithmetic operations. A modified version of the code from [12] is used to perform the Strassen FFT multiplication on GPU.

We implemented the scheme with small and medium parameter setting, respectively dimension 2,048 and 8192. We also recompiled the code provided by Gentry and Halevi for the CPU implementation [3] on the same computer for comparison. The performance results are summarized in Table 4.

As we can see clearly from the table, our implementation for the small case is about 174, 7.6 and 13.5 times
faster than the original Gentry-Halevi code for encryption, decryption and recryption, respectively [3]. The impressive speedup of encryption is due to the fact that encryption benefits significantly from pre-computation. For the medium case with dimension 8192, we also achieved a speed up of 442, 9.7 and 11.7. Note that the encryption process enjoys even more speedup as the dimension grows.

To explore the effect of our optimizations, we also broke down and evaluated the time consumption for the Recrypt primitive. Use the small case as example, if we look into the 1.32 seconds of time it takes to compute the Recrypt, we discover that it takes about 0.87 seconds for processing blocks and 0.46 second for grade-school addition. Further inspection of the block processing part reveals that the CPU multiplications and additions take about 0.54 second. In the meantime, it takes the CPU about 0.6 second to compute the factor. Clearly, the sum of this two latencies amounts to more than 0.87 seconds. This is due to the fact that the CPU and the GPU are working in parallel. In addition, the time consumed on multiplication is minimized due to the reduction of the costly FFT/IFFT computations and modular reductions.

For the medium case, the GPU memory is not large enough to hold the whole public key. Therefore, the keys are loaded when required, which introduces an overhead of about 0.6 seconds for Recrypt. However, as the computation for the medium case consumes much more time, the impact of this overhead to the overall performance is limited.

### Table 4

Performance of FHE primitives with proposed optimizations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Small Setting: Dimension 2048</th>
<th>Speedup</th>
<th>Medium Setting: Dimension 8192</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypt</td>
<td>CPU: 14.56 sec, GPU: 2.4 m sec</td>
<td>442</td>
<td>CPU: 11.38 sec, GPU: 2.4 m sec</td>
<td>442</td>
</tr>
<tr>
<td>Decrypt</td>
<td>CPU: 14 m sec, GPU: 1.84 m sec</td>
<td>7.6</td>
<td>CPU: 17.8 sec, GPU: 1.32 m sec</td>
<td>13.5</td>
</tr>
<tr>
<td>Recrypt</td>
<td>CPU: 17.8 sec, GPU: 1.32 m sec</td>
<td>7.6</td>
<td>CPU: 97.6 sec, GPU: 1.08 m sec</td>
<td>1.08</td>
</tr>
</tbody>
</table>

7 Conclusion

In this paper, we applied a number of algorithmic optimizations to the Gentry-Halevi FHE. The modular reduction operations are delayed in the implementation of the encryption and recryption primitives to reduce the number of costly large integer multiplications. Barrett's modular reduction algorithm is used to realize the remaining few modular reductions. In addition, we re-organized the encryption and recryption process so that we are performing the majority of the computations in FFT domain. Thus we managed to eliminate most costly back and forth conversions in Strassen's FFT based integer multiplication algorithm.

A GPU platform is used to further speed up the implementation. The performance results of the FHE primitives are obtained from the experiment on a machine with Intel Core i7 3770K and NVIDIA GTX 690. Experimental results with small parameter setting (2048 dimension) yields speedup factors of 174, 7.6 and 13.5 for Encrypt, Decrypt and Recrypt, respectively, when compared with the CPU reference implementation by Gentry and Halevi. Although the presented GPU implementation is not quite fast enough for cloud computing use yet, this is a solid step towards achieving that goal.

### References


