Accelerating Fully Homomorphic Encryption Using GPU

Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang and Berk Sunar
Department of Electrial and Computer Engineering
Worcester Polytechnic Institute, Worcester, MA, USA
{weiwang, hhyy_best, lchen, xhuang, sunar}@wpi.edu

***Abstract*—As a major breakthrough, in 2009 Gentry introduced the first plausible construction of a fully homomorphic encryption (FHE) scheme. FHE allows the evaluation of arbitrary functions directly on encrypted data on untwisted servers. In 2010, Gentry and Halevi presented the first FHE implementation on an IBM x3500 server. However, this implementation remains impractical due to the high latency of encryption and recryption. The Gentry-Halevi (GH) FHE primitives utilize multi-million-bit modular multiplications and additions which are time-consuming tasks for a general purpose computer.**

In the GH-FHE implementation, the most computationally intensive arithmetic operation is modular multiplication. In this paper, the million-bit modular multiplication is computed in two steps. For large number multiplication, Strassen's FFT based algorithm is employed and accelerated on a graphics processing unit (GPU) through its massive parallelism. Subsequently, Barrett modular reduction algorithm is applied to implement modular reduction. As an experimental study, we implement the GH-FHE primitives for the small setting with a dimension of 2048 on NVIDIA C2050 GPU. The experimental results show the speedup factors of 7.68, 7.4 and 6.59 for encryption, decryption and recrypt respectively, when compared with the existing CPU implementation.

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

I. 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 resolved 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 platforms today. For instance, it was recognized early in [1] that the privacy of sensitive data on cloud computing platforms are ideally suited to be protected using FHE. Considering this model of savings in scale and the recent trend, it is safe to state that cloud computing will have a significant transforming effect on business and personal computing in the coming years. This presents a perfect application target for FHE schemes.

Despite its promising prospective, FHE is nowhere near real-life deployment due to serious efficiency impediments. The first implementation of a FHE variant was proposed by Gentry and Halevi [3], which presented an impressive array of optimizations in order to reduce the size of the public-key and to reduce the latencies 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 needs to 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 needs to be done before FHE becomes readily deployable for practical use.

In this paper, we take another step towards this direction. We present a GPU realization of the FHE variant introduced by Gentry and Halevi [3]. Our implementation shows significant improvement in speed over the existing 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 computing platforms. The GPU approach is also applicable from the hardware perspective. 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 computing (GPGPU) initiative are FFT [5], data processing [6] and many other science and engineering applications [7].

In this work, we present the first GPU implementation of the Gentry-Halevi FHE algorithm [3]. More specifically, we combine Strassen's FFT based integer multiplication algorithm with Barrett's modular reduction algorithm to implement an efficient modular multiplier that supports the operands in the size of million bits. We then utilize the modular multiplier and other operation units to implement the FHE primitives: encryption, decryption and recrypt. On the NVIDIA C2050, we obtain a factor of 8 speedup for decryption over the CPU implementation in [3]. We also present the efficient implementations of encryption and recryption, which both are optimized to take advantage of the GPU parallelism. Our GPU

978-1-4673-1576-0/12/$31.00 ©2012 IEEE

implementation yields a speedup factor of 8 for encryption and 7.6 for recrypt when compared with the CPU implementation in [3].

The rest of the paper is organized as follows. In Section 2, we briefly review Gentry's FHE scheme and the algorithms for modular multiplications. The Strassen FFT based multiplication scheme is also discussed in this section. Further optimizations are discussed in Section 3. In Section 4, we present the performance evaluation and experimental results.

II. Previous Work

A. The Gentry-Halevi FHE Scheme

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: $E(x_1) \star E(x_2) = E(x_1 \otimes x_2)$. If the scheme supports the homomorphic computation 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.

As mentioned before, the first FHE is proposed by Gentry in [1], [8]. However, this preliminary implementation is too inefficient to be used in any practical application. 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 reference [3].

$\mathsf{Encrypt}$: To encrypt a bit $b \in \{0,1\}$ with a public key (d,r). $\mathsf{Encrypt}$ first generates a random "noise vector" $u = \langle u_0, u_1, \ldots, u_{n-1} \rangle$, with each entry chosen as 0 with the probability 0.5 and as ± 1 with probability 0.25 each. Then the message bit b is encrypted by computing

$$c = [u(r)]_d = \left[b + 2\sum_{i=1}^{n-1} u_i r^i \right]_d \tag{1}$$

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 $c_1 = \mathsf{Encrypt}(m_1)$ and $c_2 = \mathsf{Encrypt}(m_2)$, we have:

$$\mathsf{Encrypt}(m_1 + m_2) = (c_1 + c_2) \bmod d$$
$$\mathsf{Encrypt}(m_1 \cdot m_2) = (c_1 \cdot c_2) \bmod d\ .$$

$\mathsf{Decrypt}$: The encrypted bit c can be recovered by computing

$$m = [c \cdot w]_d \bmod 2 \tag{2}$$

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

$\mathsf{Recrypt}$: Briefly, the $\mathsf{Recrypt}$ process is simply the homomorphic decryption of the ciphertext. However, due to the fact that we can only encrypt a single bit and only a limited number of arithmetic operations can be evaluated, 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 $\sum^s w_i = w$. Each w_i is further expressed as $w_i = x_i R^{l_i} \bmod d$ where R is some constant, x_i is random and $l_i \in \{1, 2, \ldots, S\}$ is also random. The recrypt process can then be expressed as:

$$\begin{aligned} m &= [c \cdot w]_d \bmod 2 \\ &= \left[\sum^S c x_i R^{l_i} \right]_d \bmod 2 \\ &= \left[\sum^S c x_i R^{l_i} \right]_2 - \left[\left\lfloor (\sum^S c x_i R^{l_i})/d \right\rceil \cdot d \right]_2 \\ &= \left[\sum^S c x_i R^{l_i} \right]_2 - \left[\left\lfloor \sum^S (c x_i R^{l_i}/d) \right\rceil \right]_2 . \end{aligned}$$

The $\mathsf{Recrypt}$ process can then be divided into two parts. First 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 $\{\eta_1^{(i)}, \eta_2^{(i)}, \ldots, \eta_n^{(i)}\}$ where only two elements are "1" and all other elements are "0"s. Suppose the two positions are labelled 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

$$c x_i R^{l_i} = \sum_a \eta_a^{(i)} \sum_b \eta_b^{(i)} c x_i R^{l(a,b)}\ .$$

Obviously, only when $\eta_a^{(i)}$ and $\eta_b^{(i)}$ are both "1", the corresponding $c x_i R^{l(a,b)}$ is selected out. In addition, if we encode the l in the way that each iteration will increase it by one, 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 η_i will be stored in encrypted form and incorporated into the public key.

B. Fast Multiplications on GPUs

The Strassen FFT Multiplication Algorithm: Large integer multiplication is by far the most time consuming operation in the FHE scheme. Therefore, it becomes the first target for optimization. As mentioned earlier, the key feature a GPU provides is parallelism. Therefore, a good parallel algorithm will be well matched with GPU hardware. In [9], Strassen described such a multiplication algorithm based on FFT, which offers a good solution for effectively parallel computation of the large-number multiplication. Briefly, the Strassen FFT algorithm can be summarized as follows:

1) Given a base b, compute the Fast Fourier Transform of the digits (with respect to the base) of A and B, treating each digit as an FFT sample.

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 $C' = invFFT(C)$.
4) Resolve the carries: when $C'[i] \geq B$:set $C'[i+1] = C'[i+1] + (C'[i] \operatorname{div} b)$, and $C'[i] = C'[i] \bmod b$.

Emmart and Weems' Approach: In [10], Emmart and Weems implemented the Strassen FFT based multiplication algorithm on GPUs with computational optimizations. 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 [11]. 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 [12] is employed to compute the large size FFT. Assembly language level optimization and better arrangement of shared memory for GPU cores are also introduced.

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 19 when comparison with multiplication on the CPUs of the same technology generation. We follow their implementation in [10] and test it on the NVIDIA Tesla C2050. As we can see from Table I, the actual speedup factors are slightly different from [10]. Nevertheless, it is a significant speedup over CPU. Therefore, we employ this particular instance of the Strassen FFT based multiplication algorithm in our FHE implementation.

Table I
MULTIPLICATION TIME CPU VS GPU

Size in K bits	On CPU	On GPU	Speedup
1024 x 1024	8.1 ms	0.765 ms	10.6
2048 x 2048	18.8 ms	1.483 ms	12.7
4096 x 4096	42.0 ms	3.201 ms	13.1
8192 x 8192	97.0 ms	6.383 ms	15.2
16384 x 16384	221.5 ms	12.718 ms	17.4

C. Modular Multiplication

An efficient modular multiplication is crucial for the FHE implementation. Many cryptographic software implementations employ the Montgomery multiplication algorithm, c.f. [13], [14]. 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 take advantage of the parallelism feature of GPUs. In [15], for example, a Montgomery multiplication implementation on NVIDIA Geforce 9800GX2 card was presented. The speedup factor of GPU decreased from 2.6 for 160-bit modular multiplication to 0.6 for 384-bit modular multiplication, which showed a negative trend with growing operand sizes. In addition, the underlying large integer multiplication algorithm we consider here 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.

```
1: procedure BARRETT($t, M$)                ▷ Output: $r = t \bmod M$
2:     $q \leftarrow \lceil \log_2(M) \rceil$                ▷ Precomputation
3:     $\mu \leftarrow \lfloor \frac{2^q}{M} \rfloor$                ▷ 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 $r$                ▷ $r = t \bmod M$
9: end procedure
```

Figure 1. Barrett reduction algorithm

Montgomery reduction [16] and the Barrett reduction algorithms [17] are among the most popular modular reduction algorithms. For the same reason as stated above, the interleaved Montgomery reduction algorithm cannot exploit the full power of the GPU. If we use large residue so that no long carry chains there, the Montgomery reduction will have the similar complexity as the Barrett reduction. However, the Barrett approach has a simpler structure and thus is easier to apply further optimizations. Therefore, we choose the Barrett method for modular reductions.

Given two positive integers t and M, the Barrett modular reduction approach computes $r = t \bmod M$. The algorithm as shown in Fig. 1 requires precomputation of $\mu = \lfloor \frac{2^q}{M} \rfloor (q = \lceil \log_2(M) \rceil)$. If multiple reductions are to be computed with the same modulo M, then this number can be reused for all reductions, which is exactly the case we have.

Note that the step 2 of the reduction is a loop. However, in our FHE implementation, as t is normally the result of the multiplication between two integers which are smaller than M. The loop can always finish very fast.

III. GPU IMPLEMENTATION OF FHE

The FHE algorithm consists of four functions: 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 on the other three primitives.

For the Decrypt process, we perform the computation as in 2. Obviously, the most time consuming computation is a single operation of large number modular multiplication . Directly applying the FFT based Strassen algorithm and Barrett reduction will speed up the Decrypt operation significantly. Given that Decrypt is already sufficiently fast, we turn our attention to Encrypt and Recrypt.

A. Implementing Encrypt

For the Encrypt process, the most expensive operation is the evaluation of the degree-$(n-1)$ polynomial u at the 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" $u^{bot}(r) = \sum_{i=0}^{n/2-1} u_i r^i$ and a "top half" $u^{top}(r) = \sum_{i=0}^{n/2-1} u_{i+d/2} r^i$. Then $y = r^{n/2} u^{top}(r) + u^{bot}(r)$ can be calculated. The degree can be repeatedly cut in half and once the degree is small enough then the "trivial implementation" can be used to compute all powers of r.

In our implementation, as the GPU does not support recursive calls, we use a more 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:

$$\begin{aligned} \sum (u_i r^i) &= \sum_{j=0}^{t-1} [r^{w \cdot j} \cdot \sum_{i=0}^{w-1} (u_{i+wj} r^i)] \\ &= ((a_{t-1} r^w + a_{t-2}) r^w + a_{t-3}) r^w + \\ &\quad \ldots + a_1) r^w + a_0, \\ a_j &= \sum_{i=0}^{w-1} (u_{i+wj}), \end{aligned}$$

where additions and multiplications are evaluated with modulo d. After organizing the computation as described above, we can introduce pre-computation to further speed up the process. As r is a known constant for the encryption, the $r^i, i = 0, 1, \ldots, w$ can be pre-computed. In addition, to reduce the overhead caused by the relatively slow communication between the CPU and the GPU, these pre-computed values can be pre-loaded into GPU memory before the Encrypt process starts. Clearly, larger window size w leads to less multiplications, which yields better performance. However, it also means higher storage requirement for more pre-computed values. Hence, it is trade-off between speed and memory use.

In our implementation, we have the dimension $n = 2048$ and $|d|$ is approximately 128KB. We can estimate the performance and storage requirement for different window sizes from that. The estimated value is listed in Table II. We choose the case of window size $w = 64$ for our implementation.

Table II
COMPARISON BETWEEN DIFFERENT WINDOW SIZES

Window Size	Number of Multiplications	Size of Pre-computed values
16	127	2 MB
32	63	4 MB
64	31	8 MB
128	15	16 MB

B. Implementing Recrypt

The Recrypt process is more complicated. As mentioned in previous section, Recrypt process can be divided into two steps: process S blocks separately and then sum them up. For the process separate blocks, the the most time-consuming computation is in the form of

$$cx_i R^{l_i} = \sum_a \eta_a^{(i)} \sum_b \eta_b^{(i)} cx_i R^{l(a,b)} .$$

where η_i is part of the public key. As we mentioned in previous sections, if we encode the l in a proper way such that each iteration it only increases by one, the next factor $cx_i R^{l(a,b)}$ can be easily computed by multiplying R with the result of the previous iteration. Here we refer $cx_i R^{l(a,b)}$ for each iteration as a *factor*. In each iteration, we compute *factor* $=$ *factor* $\cdot R \bmod d$ and determine whether we should sum η_b 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 harness the the computational power in the overall system.

The constants used in Recrypt are part of the public key. They can be "pre-computed" to further speed up the process. Similar to the Encrypt, the public keys can be pre-loaded into the GPU memory to eliminate the costly CPU-GPU communication step. Taking our implementation as an example, the public key size is about 70MB. It can perfectly fit into the 3GB GPU memory of the latest graphic cards. In fact, 3GB is enough even for the large setting in FHE [3], whose public key size is about 2.25GB.

Upon completion of processing all the "blocks", we can sum these partial results. In practice, retaining only 4 most significant bits for each number is sufficient for correctness, i.e. to make decryption work. Note that during the whole Recrypt process, all of the operations are evaluated homomorphically. All the numbers which are summed together are encrypted bit by bit. Therefore, we follow the design of binary adders and substitute bit operations with corresponding Eval operations - modular evaluation operations. The addition algorithm used here is called the grade-school addition. It takes about $O(s^2)$ multiplications to compute the sum of s numbers. Hence, we need $O(s^2)$ modular multiplications to perform the grade-school addition homomorphically. Clearly the efficiency of the Strassen-FFT and Barrett reduction based modular multiplication algorithm directly translates into an efficient homomorphic addition computation.

IV. EXPERIMENTAL RESULTS

An a case study, the Encrypt, Decrypt and Recrypt of the Gentry-Halevi FHE scheme are evaluated on a server with Intel Xeon X5650 processor running at 2.67GHz, 14 GB RAM and two NVIDIA Tesla C2050s, each of which has 448 cores, 3GB memory running at 1150MHz. However, only one GPU is used in this implementation. Shoup's NTL library [18] is used for high-level numeric operations and GNU's GMP library [?] for the underlying integer arithmetic operations. A modified version of the code from [10] is used to perform the Strassen FFT multiplication on GPU.

For an experimental study, we employed the smallest setting with a lattice-dimension of 2,048. In this setting, the determinant d has about 790,000 bits. In practical applications, the key generation can usually be processed offline and we do not need to accelerate this part. Gentry-Halevi's implementation code [3] is also executed on the the same platform for comparisons. The main results of our experiments are summarized in Table III. We see that our GPU implementation is about 8.2, 8 and 7.6 faster for encryption, decryption and recryption, respectively, when compared to the Gentry-Halevi implementation on the CPU [3].

Table III
FHE ON DIFFERENT PLATFORMS

	CPU	GPU
Encrypt	1.69 sec	0.22 sec
Decrypt	18.5 msec	2.5 msec
Recrypt	27.68 sec	4.2 sec

If we look into the entire 4.2 seconds of the time it takes to compute Recrypt, we discover that it takes about 3.56 seconds to process these "blocks" and about 0.68 seconds to perform the grade-school addition. Further dissection of the block processing on GPU, about 2.66 seconds are dedicated for the multiplications and 0.24 seconds for the additions. At the same time, the CPU spends 0.9 seconds computing *factor*. Clearly, the sum of the time is more than 3.56 seconds. This indicates the fact that the CPU and the GPU are actually performing computing tasks concurrently.

V. CONCLUSION

In this paper, we present the first GPU implementation of a fully homomorphic encryption scheme. To optimally support the higher level primitives of the Gentry-Halevi FHE, we develop efficient techniques for large integer arithmetic operations. At the lower level, we pair Emmart and Weems' implementation of Strassen's FFT multiplication with Barrett reduction to realize a high-performance modular multiplication on a GPU. Using this basic operation along with pure Barrett reduction and modular addition, we implement the FHE primitives. In addition, we tailor the encryption and recrypt functions to make optimal use of GPU features as well as to avoid obstacles, such as lack of support for recursive operations. We also develop a pre-computation strategy to further enhance the efficiency of the encryption primitive.

The performance results of the FHE primitives are obtained from the executions on a server equipped with a NVIDIA Tesla C2050 GPU. Experimental results show the speedup factors of 7.68, 7.4 and 6.59 for Encrypt, Decrypt and Recrypt, respectively, when compared with the CPU reference implementation in [3]. Although further advance are still heavily sought before FHE becomes deployable in real-world applications, this work shows that the performance of FHEs can be significantly improved by carefully choosing the target platform and by tailoring the algorithms.

VI. ACKNOWLEDGMENTS

We would like to thank Nial Emmart for his recommendations and for sharing his GPU FFT multiplication code with us. We also would like to thanks the National Science Foundation for supporting this work in part through NSF-CNS Award #1117590.

REFERENCES

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st annual ACM symposium on Theory of computing*. ACM, 2009, pp. 169–178.

[2] R. Rivest, L. Adleman, and M. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 32, no. 4, pp. 169–178, 1978.

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

[4] D. Cousins, K. Rohloff, C. Peikert, and R. Schantz, "SIPHER: Scalable implementation of primitives for homomorphic encryption–FPGA implementation using Simulink," *High Performance Extreme Computing Conference*, 2011.

[5] X. Cui, Y. Chen, and H. Mei, "Improving performance of matrix multiplication and fft on gpu," in *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*. IEEE, 2009, pp. 42–48.

[6] N. Govindaraju, N. Raghuvanshi, and D. Manocha, "Fast and approximate stream mining of quantiles and frequencies using graphics processors," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 2005, pp. 611–622.

[7] J. Baladron Pezoa, D. Fasoli, and O. Faugeras, "Three applications of gpu computing in neuroscience," *Computing in Science & Engineering*, no. 99, pp. 1–1, 2011.

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

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

[10] N. Emmart and C. Weems, "High precision integer multiplication with a gpu using strassen's algorithm with multiple fft sizes," *Parallel Processing Letters*, vol. 21, no. 3, p. 359, 2011.

[11] J. Solinas, "Generalized mersenne numbers," *Technical Reports*, 1999.

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

[13] C. McIvor, M. McLoone, and J. McCanny, "Fast montgomery modular multiplication and rsa cryptographic processor architectures," in *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, vol. 1. IEEE, 2003, pp. 379–384.

[14] A. Daly and W. Marnane, "Efficient architectures for implementing montgomery modular multiplication and rsa modular exponentiation on reconfigurable logic," in *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*. ACM, 2002, pp. 40–49.

[15] P. Giorgi, T. Izard, A. Tisserand *et al.*, "Comparison of modular arithmetic algorithms on gpus," 2009.

[16] P. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[17] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology: CRYPTO 1986*. Springer, 1987, pp. 311–323.

[18] V. Shoup, "NTL: A library for doing number theory," 2001.