Enhanced Implementation of the NTRUEncrypt Algorithm Using Graphics Cards

Abdel Alim Kamal and Amr M. Youssef
Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada
{*a_kamala, youssef*}@ciise.concordia.ca

Abstract

The NTRU encryption algorithm, also known as NTRUEncrypt, is a parameterized family of lattice-based public key cryptosystems that has been accepted to the IEEE P1363 standards under the specifications for lattice-based public-key cryptography (IEEE P1363.1). The operations of the NTRU encryption algorithm show good characteristics for data parallel processing which makes the NTRU a good candidate to benefit from the high degree of parallelism available in modern graphics processing units (GPUs). In this paper, we investigate different GPU implementation options for the NTRU encryption algorithm. Our implementation, on the NVIDIA GTX275 GPU, using the CUDA framework, achieves about 77 MB/s for NTRU with the parameter set $(N, q, p) = (1171, 2048, 3)$.

1 Introduction

A Graphical Processing Unit (GPU) is a dedicated hardware for rendering graphics on devices such as personal computers, workstations, game consoles and embedded systems. Latest technologies made it possible to combine multiple GPUs in a single machine and hence achieving an affordable level of parallel processing. One of the main differences between GPUs and CPUs is that, in general, a CPU is optimized for executing high performance sequential code and hence the majority of its transistors are dedicated for flow control and branch prediction. On the other hand, because of the characteristics of its target applications, a GPU has a more parallel nature and the majority of its transistors are dedicated for computation. During the past few years, the power of GPUs has been increasing at a higher rate than that of CPUs. Consequently, general purpose computing on graphics processing units, i.e., the use of GPUs to accelerate the computations of different algorithms, has become a recent trend in parallel computing and played an important role in developing faster implementations for many algorithms in different areas. Because of their inherent computational complexity, encryption techniques are good candidates to benefit from the affordable high level of parallelism available on current GPUs especially given the fact that many cryptographic algorithms show good characteristics for data parallel processing such as high computation intensity and independent work loads. Furthermore, security is becoming increasingly important and there is a great demand to secure data in all phases of its life cycle from communication to active or archived storage. This trend requires increased processing power which can be met by a combination of standard CPUs and GPUs acting as cryptographic accelerators, especially since GPUs are now ubiquitous and, unless playing graphics intensive games, its computational power are under-utilized for the majority of the time.

Several researchers have explored the great potential to use this available GPU power in a similar role that existing hardware cryptographic accelerators play. In symmetric key cryptography, several works (e.g., [1], [2], [3], [4], and [5]) have reported different GPU implementations for the Advanced Encryption Standard (AES). A very fast GPU implementation for the Korean ARIA block cipher was achieved by Yung *et al.* [6] where they reached an encryption throughput of 4.8 Gbps. In asymmetric key cryptography, R. Szerwinski *et al.* [7], [8] exploited the power of GPUs to speed up the computations of the expensive operations such as modular exponentiation for RSA and DSA and point multiplication for ECC. They obtained 813 modular exponentiation per second for RSA and DSA-based systems with 1024 bit integers and 1412 point multiplications per second for ECC over P-224 NIST elliptic curves. A fast GPU implementation for one of the NIST SHA-3 hash function candidates, Blue Midnight Wish (BMW), was presented and analyzed in [9]. A GPU implementation for NTRU was also given in [10] where, using a modern 1.2 GHz GTX280 GPU, a throughput of up to 200,000 encryptions per second was reached at a security level of 256 bits. This gives a theoretical data throughput of 47.8 MB/s (See also [11] [12] for other software and hardware implementations of the NTRUEncrypt).

978-1-4244-7674-9/10/$26.00 ©2010 IEEE

In this work, we investigate different implementation options for the NTRU encryption on the NVIDIA GTX275 GPU. Using the Compute Unified Device Architecture (CUDA) framework, our implementation achieves more than 351,000 parallel encryption operations per second for NTRU with 256 bits security level (parameter set $(N, q, p) = (1171, 2048, 3)$) which corresponds to an encryption throughput of about 77.21 MB/s.

The rest of the paper is organized as follows. The relevant details of the NTRU encryption algorithm are reviewed in the next section. The thread organization and the memory model of the CUDA framework are described in section 3. The different implementation options are analyzed in section 4 and the implementation results are given in section 5.

2 Description of the NTRU algorithm

The NTRU encryption algorithm is a parameterized family of lattice-based public key cryptosystems. Both the encryption and decryption operations in NTRU are based on simple polynomial multiplication which makes it very fast compared to other alternatives such as RSA, and elliptic-curve-based systems. Recently, the NTRU system has been accepted to the IEEE P1363 standards under the specifications for lattice-based public-key cryptography [13].

NTRUEncrypt is parameterized by three integers: (N, p, q), where N is prime, gcd(p, q) = 1 and $p << q$. Let R, R_p, and R_q be the polynomial rings

$$R = \frac{\mathbb{Z}[x]}{x^N - 1}, R_p = \frac{\mathbb{Z}/p\mathbb{Z}[x]}{x^N - 1}, R_q = \frac{\mathbb{Z}/q\mathbb{Z}[x]}{x^N - 1}.$$

The product of two polynomials $a(x), b(x) \in R$ is given by

$$a(x) \star b(x) = c(x)$$

where

$$c_k = \sum_{i+j=k \pmod N} a_i b_{k-i}$$

For any positive integers d_1 and d_2, let $\tau(d_1, d_2)$ denote the set of ternary polynomials given by

$$\left\{ a(x) \in R_p \;\middle|\; \begin{array}{l} a(x) \text{ has } d_1 \text{ coefficients equal to 1,} \\ a(x) \text{ has } d_2 \text{ coefficients equal to -1,} \\ a(x) \text{ has all other coefficients equal to 0} \end{array} \right\}$$

In what follows, we briefly describe the key generation, encryption and decryption operations in the NTRU cryptosystem [14], [15]:

2.1 Key Generation

- Choose a private $f(x) \in \tau(d_f+1, d_f)$ that is invertible in R_q and R_p.
- Choose a private $g(x) \in \tau(d_g, d_g)$.
- Compute $F_q(x) = f^{-1}(x)$ in R_q and $F_p(x) = f^{-1}(x)$ in R_p.
- Compute $h(x) = F_q(x) \star g(x)$ mod q.

The polynomial $h(x)$ is the user's public key. The corresponding private key is the pair $(f(x), F_p(x))$. The following steps denote the encryption operations for plaintext $m(x) \in R_p$ and the decryption operations of the ciphertext $e(x) \in R_q$.

2.2 Encryption

- Choose a random ephemeral key $r(x) \in \tau(d_r, d_r)$.
- Compute the ciphertext $e(x) = pr(x) \star h(x) + m(x)$ mod q.

2.3 Decryption

- Compute $a(x) = f(x) \star e(x)$ mod q.
- Centerlift $a(x)$ to $a(x) \in R$
- Compute $m(x) = F_p(x) \star a(x)$ mod p.

By choosing $f(x) = 1 + pf_1(x)$, where $f_1(x) \in R$ in the key generation step, the polynomial multiplication in the last decryption step is eliminated since we will have $F_p(x) = 1$ mod p.

In this paper, we focus on the set of parameters $(N, q, p) = (1171, 2048, 3)$ which corresponds to a 256-bit security level. We also set $(d_f, d_g, d_r) = (106, 390, 106)$.

The most time consuming part of NTRU encryption and decryption is the convolution multiplication $r(x) \star h(x)$ mod q, and $f(x) \star e(x)$ mod q respectively. Hoffstein, and Silverman [14], [16] described a method for speeding up the encryption and decryption processes through the use of products of low Hamming weight polynomials. They use three low Hamming weight polynomials $r_1(x)$, $r_2(x)$, and $r_3(x)$ such that $r(x) = r_1(x) \star r_2(x) + r_3(x)$ in encryption and $f_1(x)$, $f_2(x)$, and $f_3(x)$ such that $f(x) = f_1(x) \star f_2(x) + f_3(x)$ in decryption. For $(N, q, p) = (1171, 2048, 3)$ the number of non-zero coefficients for each $r_i(x)$ and $f_i(x)$ is 5. The convolution multiplication for encryption $t(x) = r(x) \star h(x)$ mod q can be calculated as follows:

$$\begin{array}{lcl} t_1(x) & = & r_2(x) \star h(x), \\ t_2(x) & = & r_3(x) \star h(x), \\ t_3(x) & = & r_1(x) \star t_1(x), \\ t(x) & = & t_2(x) + t_3(x) \text{ mod } q. \end{array} \quad (1)$$

3 The CUDA framework

Early attempts to use GPUs in cryptography were not very encouraging due to its previously poor suitability to the problem space, especially, given the lack of integer processing support. Furthermore, GPU programming was challenging for those who are not familiar with graphics. Nowadays, high level language libraries supporting parallel operations on GPUs have been developed by the main manufacturers of GPUs [17] [18] [19] [20]. In what follows, we briefly summarize the thread organization and memory model of the Compute Unified Device Architecture (CUDA) [21] framework developed by NVIDIA.

3.1 Thread organization and memory model

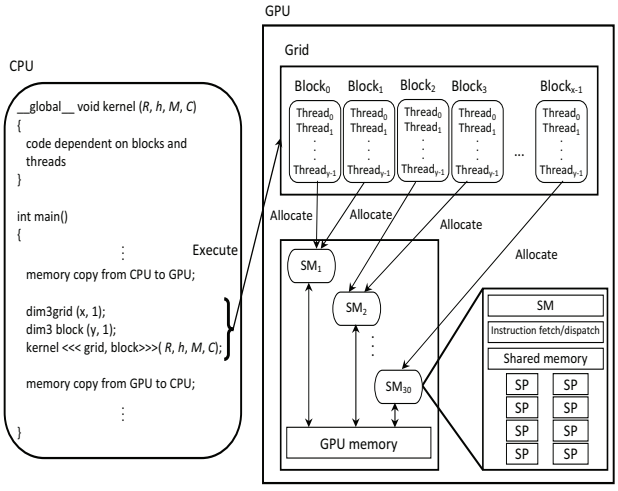

Figure 1. The CUDA programming model

The CUDA framework defines normal C functions called kernels. Typical candidates for a kernel are functions that are executed many times but on multiple independent data. An execution configuration is used to specify the number of times the function should be executed as threads on the GPU and how the threads are organized. To manage the large number of threads executed on the GPU, CUDA uses a thread hierarchy to identify and organize the threads. Unique coordinates are used to distinguish threads that execute the same function. As shown in Figure 1, for every launch, the threads are lined up in a grid which is divided into a two level hierarchy of thread blocks and threads identified by coordinates called *blockIdx* and *threadIdx* which are assigned to them by the CUDA runtime system. These coordinates are accessible in the kernel to identify the different threads. The threads within a thread block can be organized in a one, two or three-dimension. In one kernel launch, a dimension in the grid cannot exceed 65535 and the size of a thread block is limited to 512. When the grid of thread blocks are launched, each thread block is assigned to an arbitrary streaming multiprocessor (SM). Every thread within the thread block is computed on the same SM. To schedule the threads, the SMs use a technique that NVIDIA calls single-instruction multiple-thread (SIMT). It is similar to single instruction multiple data (SIMD) but SIMT specifies the execution and branching behavior of each thread. The SMs are allocated thread blocks, and the SIMT unit splits the threads into groups of 32 threads called warps. Each warp consists of threads with consecutive increasing thread IDs with the first warp of a thread block containing thread ID 0-31. For every clock cycle, the SM chooses a warp that is ready to execute for scheduling, and hands each streaming processor (SP) a thread.

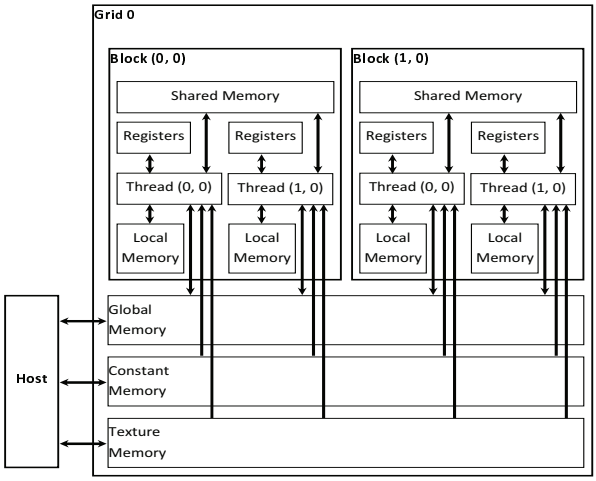

Figure 2. The CUDA memory model [21]

While Kernels run as threads on the GPU, the memory copying and the execution configuration is done by the CPU (the host PC). This separates the code into code executed on the CPU and code executed on the GPU. While both the host and the GPU manage their own memory space, data can be copied between them. Figure 2 shows the memory model in CUDA. Global memory (GM) is the most frequently used memory space since it is the only memory space that can be both read and written by both host and GPU. Constant and texture memory is read-only on the GPU, and is setup from the host for read-only data. The constant and texture memory is located in the same physical memory (DRAM) as global memory, but uses the texture unit in combination with a cache available to each SM to increase speed up reads from the memory spaces. Each SM has an 8 kB working set of constant and texture memory that is cached, which makes it useful for algorithms using data patterns that are read-only. Global memory, texture and constant mem-

ory can be accessed from any thread on the grid. Shared memory (SH) is the only memory space that is available on the actual GPU core and it is local to each SM, i.e., it can be accessed only within a thread block. Since the shared memory is very quick, it is optimal for sharing data across a thread block, or for performing computation before writing back to global memory. Thus to avoid the latency of global memory, a good way to compute data is to load data from global memory into shared memory, perform the computations in shared memory, synchronize each thread block and then write the result back to the global memory so that the host can read it when done.

4 Implementation Options for the NTRUEncrypt

The target platform for our implementation is the NVIDIA GTX275 GPU which has been installed on the PCI Express 2.0 bus of a PC running AMD Athlon 64 X2 5000+ dual-core processor at 2.6 GHz with 2 GB of RAM. CUDA enables us to use the GTX275 GPU card, which runs at 1.404 GHz, as a parallel machine which contains 30 SMs, each contains 8 SPs with a total of 240 processor cores. The total memory of the card is 896 MB and each SM has a 16 KB shared memory.

To exploit the advantages of the GPU, several plaintext messages $m_i(x), i = 1, \cdots n$, are encrypted in parallel. The GPU receives the plaintext messages as a matrix M where the i^{th} row in M corresponds to the plaintext message $m_i(x)$. Similarly, the corresponding random ephemeral keys are transferred to the GPU as a matrix R where the i^{th} row represents the random ephemeral key $r_i(x)$ used to encrypt the message $m_i(x)$. Finally, The GPU receives the public key $h(x)$ as a vector and compute the the ciphertext as a matrix C where the i^{th} row in C corresponds to the ciphertext $c_i(x)$. According to the method used to present the coefficients of $r_i(x)$, $h_i(x)$, $m_i(x)$ and $c_i(x)$, and the use of shared memory, we have the following implementation choices:

- *Representation of the polynomial* $r(x)$: The polynomial $r_i(x)$ can be represented in the naive form as a polynomial with N coefficients or it can be represented in the product form as shown by equation (1).

- *Representation of the polynomial coefficients:* Traditionally, each coefficient of the polynomials $r_i(x)$ and $c_i(x)$ is stored in one integer variable (normal coefficients representation). In order to reduce the memory copying time between the host and the GPU, we employed a technique referred to as bit-packing [22] where each coefficient in $r_i(x)$ is represented in 2-bits. Therefore, each $\lfloor \frac{32}{2} \rfloor = 16$ coefficients can be represented in one integer variable. Similarly, since $q = 2048 = 2^{11}$, $\lfloor \frac{32}{11} \rfloor = 2$ coefficient in $c_i(x)$ can be stored in one integer. This helps reduce the the time required to copy the data from/to the GPU at the expense of the added computational time required to pack/unpack the polynomial coefficients before using them in the actual computations.

- *Storage options:* Data can be stored in global memory or in the shared memory. The advantage of using shared memory over global memory is the low latency in calculation. However, the size of the GPU shared memory is limited (16 KB for each SM of the GPU).

We assume that the key setup and the ephemeral keys generation are done off-line by the host (i.e., the PC). To encrypt n messages, each of size N elements, the convolution multiplication between $r_i(x)$ and $h_i(x)$ is done first and then the plaintext messages $m_i(x), i = 1 \cdots n$ are added (mod q) to the convolution output.

Figure 3 illustrates the convolution operation using the normal polynomial form (NP), the normal coefficients representation (NC) and the global memory as a storage. In Figure 3, the $N \times n$ matrix R corresponds to the n random ephemeral keys, each of size N elements. The i^{th} row of the $N \times N$ matrix H corresponds to a cyclically shifted version of the public key $h(x)$ by i elements. It should be noted that the matrix H shown in this Figure is a virtual matrix used for illustration purpose only, i.e., it is not physically stored on the GPU or the CPU memory. In other words, only $h(x)$ is physically stored and the different rows of H are created by the way that different threads use to access $h(x)$ using different values for $blockIdx.x$ and $blockIdx.y$ as shown in Algorithm 1. Each thread reads one row of the matrix R and one column of the matrix H and then computes the ciphertext by adding the result of the convolution to the corresponding element of the plaintext message $m_i(x)$. One kernel is used to calculate the encryption operation with $32 \times 16 = 512$ threads (to exploit the full capabilities of the SMs) which corresponds to $\lceil N/32 \rceil \times \lceil n/16 \rceil$ thread blocks.

As shown in Figure 4, when the bit-packing approach (BP) is used to present the coefficients of R and C, the size of the R matrix is reduced from $n \times N$ to $n \times \lceil N/16 \rceil$. Similarly, the size of the corresponding ciphertext matrix is reduced from $n \times N$ to $n \times \lceil N/2 \rceil$. Consequently, R can be transferred from the host to the GPU in a shorter time. Also copying C from the GPU to the host will require less time.

The computation efficiency on the GPU can be further improved when a thread block can load a block of data into the on-chip shared memory, process it there, and then write the final results back to external memory. Figure 5 shows the case where the shared memory is used as a storage when $r_i(x)$ is presented in the naive polynomial form

Algorithm 1 Pseudo code for Naive implementation of NTRUEncrypt on GPU

1: INPUT: Plaintext matrix $M_{n\times N}$, random ephemeral key matrix $R_{n\times N}$, the public key $h_{1\times N}$.
2: OUTPUT: The ciphertext matrix $C_{n\times N}$
3: $tx \longleftarrow blockIdx.x * blockDim.x + threadIdx.x$
4: $ty \longleftarrow blockIdx.y * blockDim.y + threadIdx.y$
5: **for** $i = 1$ to N **do**
6: **if** $(R[ty * N + i] = 1)$ **then**
7: $sum \longleftarrow sum + h[(tx + N - i) \bmod N]$
8: **else if** $(R[ty * N + i] = -1)$ **then**
9: $sum \longleftarrow sum - h[(tx + N - i) \bmod N]$
10: **else**
11: $sum \longleftarrow sum$
12: **end if**
13: **end for**
14: $C[ty * N + tx] \longleftarrow (sum + M[ty * N + tx]) \bmod q$

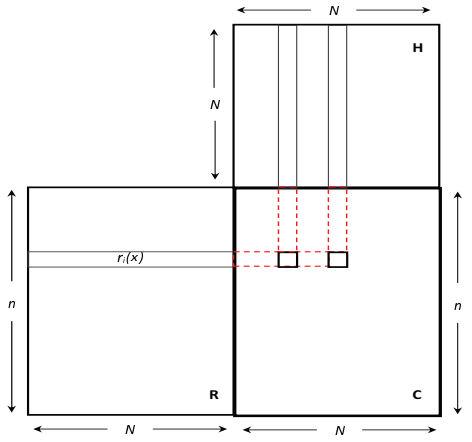

Figure 3. The convolution operation using NP, NC, and GM

and the normal representation is used to encode the polynomial coefficients. Each thread block is responsible for computing one block of the convolution matrix and each thread within the block is responsible for computing one element of the block. Two kernels are used to calculate the encryption operation. The first one calculates the matrix H from the vector $h(x)$ and then loads it into the shared memory. The second kernel completes the encryption operation via the shared memory. It also performs the necessary padding to ensure that the length of all blocks in R and H is divisible by 16. The CUDA $\_\_syncthreads()$ function is used to ensure that the data is copied to the shared memory before performing any calculations on it. In the first kernel, $32 \times 16 = 512$ threads and $\lceil N/32 \rceil \times \lceil n/16 \rceil$ blocks are used. The second kernel uses $16 \times 16 = 256$ threads and

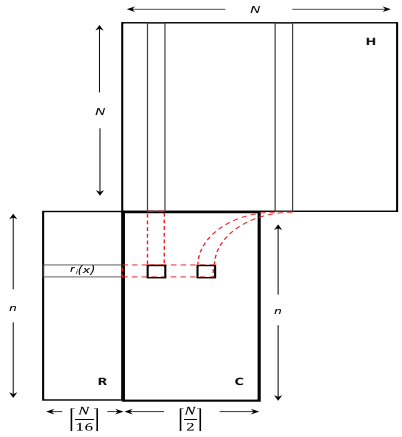

Figure 4. The convolution operation using NP, BP, and GM

$\lceil N/16 \rceil \times \lceil n/16 \rceil$ blocks.

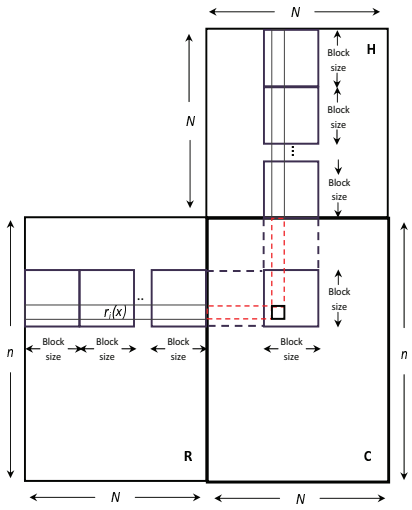

Figure 5. The convolution operation using NP, NC, and SH

Figure 6 shows the corresponding scenario where the bit-packing method is used to represent the polynomial coefficients and the shared memory is used to store the matrices R and H. Again, one kernel calculates the matrix H from the vector $h(x)$ and then loads it into the shared memory. The second kernel completes the encryption operation via the shared memory with $16 \times 16 = 256$ threads and $\lceil \lceil N/2 \rceil /16 \rceil \times \lceil n/16 \rceil$ blocks. The dimension of the matrix R is reduced to $\lceil N/16 \rceil \times n$ and divided into blocks of size 16×16 KB. Also, the matrix H is divided into blocks of size 16×16 KB. The blocks in R and the correspond-

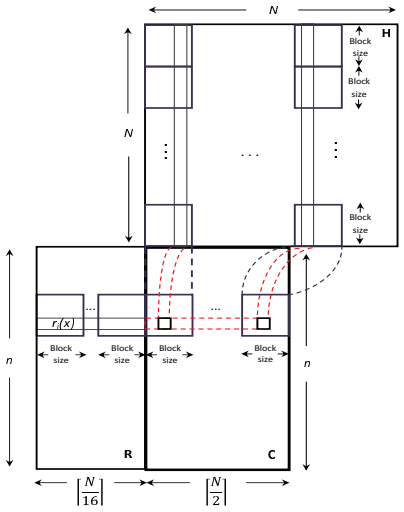

Figure 6. The convolution operation using NP, BP, and SH

ing blocks in H are loaded into the shared memory. After performing the computation, the results are written into the global memory.

Similar analysis applies when the product form (PF) is used except that the convolution operation is performed for three times per each encryption operation as shown in Eq. (1).

5 Implementation Results

As mentioned in the previous section, the key setup and the ephemeral keys generation are assumed to be generated off-line by the host. Thus, the total encryption time is equal to the time required to transfer the data from/to GPU and the time required to perform the necessary computational operations inside the GPU. The use of shared memory requires a padding of $h(x)$, $m_i(x)$, and $r_i(x)$ in order to obtain a polynomial size dividable by the block size in the shared memory. Since the decryption performs merely the same as encryption, we refer to the results for encryption.

According to the discussion in the previous section, there are $2^3 = 8$ combinations of implementation options for the NTRUEncrypt on the GPU. Figure 7 shows how the total encryption time required to encrypt n messages in parallel varies with n (i.e., as n messages are loaded in one time from the host to the GPU and then encrypted in parallel on the GPU). Figure 8 shows the corresponding encryption throughput. Due to the limited memory of the GPU, the maximum number of possible parallel operations is limited and varies depending on the memory requirements associated with the eight different implementation choices.

It is clear that the use of product form, bit packing and shared memory allows us to achieve the best throughput where, for example at $n = 32768$, the achieved number of parallel encryption operations per second is 351,635. Since each message block has 1171 tuples and each tuple can assume the value of $+1, -1, 0$, i.e., each tuple has $2 \times \frac{3}{4}$ bits of information, then encrypting 351,635 message blocks per second corresponds to an encryption throughput of $351,635 \times 1171 \times 1.5 \times \frac{1}{8} \approx 77.21$ MB/sec.

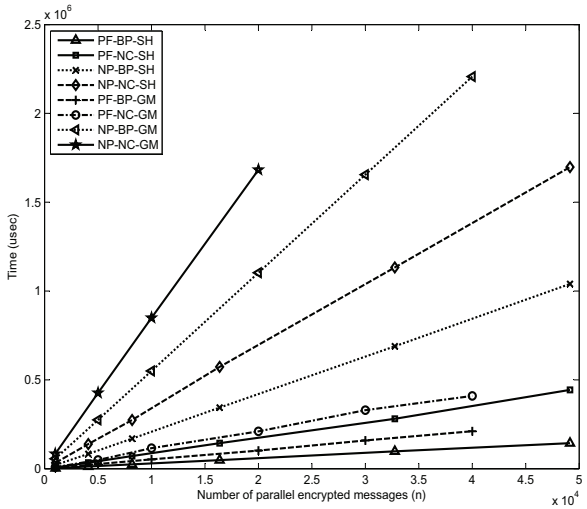

Figure 7. Total time required to encrypt n messages in parallel

6 Conclusions

In this paper we have explored different implementation options for the NTRUEncrypt algorithm with parameter set $(N, q, p) = (1171, 2048, 3)$ on the GTX275 GPU using the CUDA framework. From our implementation results, it is clear that utilizing the shared memory, using the product form for the polynomial r and representing the polynomial coefficients in the bit-packing format result in the most efficient implementation option where the achieved encryption throughput is about 77.21 MB/s.

References

[1] D. Cook, J. Ioannidis, A. Keromytis, and J. Luck, "CryptoGraphics: Secret key cryptography using graphics cards", *in proc. of CT-RSA*, LNCS vol. 3376, pp. 334-350, Springer, Heidelberg, 2005.

[2] O. Harrison, J. Waldron, "AES encryption implementation and analysis on commodity graphics processing

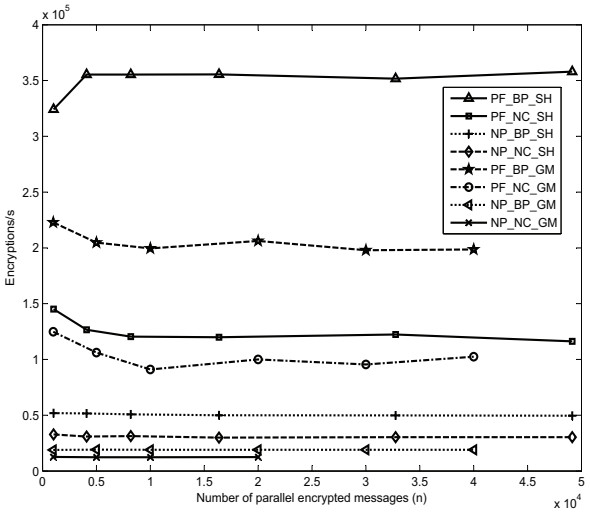

Figure 8. The number of parallel encrypted messages per second.

unit", *in proc. of CHES*, LNCS vol. 4727, pp. 209226, Springer, Heidelberg, 2007.

[3] S. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography", *in proc. of ICSPC*, pp. 65-68, 2007.

[4] U. Rosenberg, *Using graphic processing unit in block cipher calculations,* Masters thesis, University of Tartu, Tartu, Estonia, 2007.

[5] A. Ottesen, *Efficient parallelisation techniques for applications running on GPUs using the CUDA framework,* Masters thesis, University of Oslo, Oslo, Norway, 2009.

[6] Y. Yeom, Y. Cho, and Moti Yung, "High-Speed Implementations of Block Cipher ARIA Using Graphics processing Units", *in proc. of MUE*, pp. 271-275, 2008.

[7] R. Szerwinski, and T. Guneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography", *in proc. of CHES*, LNCS vol. 5154, pp. 79-99, Springer, Heidelberg, 2007.

[8] R. Szerwinski, *Efficient Cryptography on Graphics Hardware,* Diplomas thesis, Ruhr-University of Bochum, Bochum, Germany, 2008.

[9] G. Osa, *Fast Implementation of Two Hash Algorithms on nVidia CUDA GPU,* Masters thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2009.

[10] J. Hermans, F. Vercauteren, and B. Preneel, "Speed Records for NTRU", *in proc. of CT-RSA*, LNCS vol. 5985, pp. 73-88, Springer, Heidelberg, 2010.

[11] D. V. Bailey, D. Coffin, A. Elbirt, J. H. Silverman, and A. D. Woodbury, "NTRU in constrained devices", *in proc. of CHES*, LNCS vol. 2162, pp. 262-272, 2001.

[12] A. Kamal, and A. Youssef, "An FPGA Implementation of the NTRUEncrypt Cryptosystem", *in proc. of ICM*, pp. 209-212 , 2009.

[13] J. Hoffstein, J. Pipher, J. H. Silverman, and P. Hirschhorn, *IEEE P1363.1/D10: draft standard for public kry cryptographic techniques based on hard problems over lattices*, 2008.

[14] J. Hoffstein and J. H. Silverman, "Optimizations for NTRU", *in proc. of Public Key Cryptography and Computational Number Theory*, de Gruyter, Warsaw, September 2000.

[15] J. Hoffstein, J. Pipher and J. H. Silverman, *An Introduction to Mathematical Cryptography*: Undergraduate Texts in Mathematics, Springer, 2008.

[16] J. Hoffstein and J.H. Silverman, "Random Small Hamming Weight Products with applications to cryptography", *Discrete Applied Mathematics*, vol. 130, no. 1, pp. 3749, 2003.

[17] AMD "Close to Metal" Technology Unleashes the Power of Stream Computing: AMD Press Release, November 14, 2006.

[18] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a manycore x86 architecture for visual computing", *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 115, 2008.

[19] A. Munshi, "OpenCL Parallel Computing on the GPU and CPU", *in proc. of SIGGRAPH*, 2008.

[20] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware", *ACM Transactions on Graphics*, vol. 23, no. 3, p. 777-786, USA, 2004.

[21] NVIDIA, Nvidia Developer Zone, $http://www.nvidia.com/object/cuda_home_new.html$.

[22] M. Monteverde, *NTRU software implementation for constrained devices,* Master's thesis, Katholieke Universiteit Leuven, 2008.