Accelerating NTRU Encryption with Graphics Processing Units

Tianyu Bai , Spencer Davis , Juanjuan Li , Ying Gu and Hai Jiang

Department of Computer Science
Arkansas State University, USA
E-mail: {tianyu.bai, spencer.davis, juanjuan.li, ying.gu}@smail.astate.edu, hjiang@astate.edu

Abstract

Lattice based cryptography is attractive for its quantum computing resistance and efficient encryption/decryption process. However, the Big Data issue has perplexed most lattice based cryptographic systems since the overall processing is slowed down too much. This paper intends to analyze one of the major lattice-based cryptographic systems, Nth-degree truncated polynomial ring (NTRU), and accelerate its execution with Graphic Processing Unit (GPU) for acceptable processing speed. Three strategies, including single GPU with zero copy, single GPU with data transfer, and multi-GPU versions are proposed for performance comparison. GPU computing techniques such as stream and zero copy are applied to overlap computations and communications for possible speedup. Experimental results have demonstrated the effectiveness of GPU acceleration of NTRU. As the number of involved devices increases, better NTRU performance will be achieved.

Keywords: NTRU, Multi-GPU, CUDA, Acceleration.

1. Introduction

With the explosive growth of data and increasing demands on high performance computing in the modern scientific era, it does not seem that CPU will be able to handle the overwhelming computing burden alone. Graphic Processing Unit (GPU) is now turning into a forceful parallel computing co-processor for its innate hardware architecture. Because of the arising of Compute Unified Device Architecture (CUDA), achieving high performance computing by GPU is not only feasible, but also user friendly.

The complexity of lattice problems guarantees high theoretical security levels of lattice based cryptographic systems. Compact encryption and decryption schema for lattice based cryptography ensure its superiority on execution efficiency compared with other public key cryptographic systems such as RSA. Furthermore, the fact that lattice based cryptography is quantum secure enhances its probability to become a next generation mainstream cryptographic system.

However, for large data applications, traditional sequential lattice based cryptography is quite time-consuming and supports low system throughput. This paper is principally focused on applying both single GPU and multi-GPU acceleration strategies to a particular lattice based cryptographic system, NTRU, to ameliorate the computing capability bottleneck. This paper makes the following contributions:

- For NTRU key generation, a CPU implementation of was developed.
- For NTRU encryption, three GPU implementations (Single GPU, Single GPU-zero copy, and

Multi-GPU) are proposed to exploit three-level data parallelism (device, block, and thread levels).
- Stream technique is used to achieve device computation overlapping.
- Experiments have been conducted to demonstrate the effectiveness of GPU optimization.

The remainder of this paper is organized as follows. Section 2 briefly introduces basic number theory and the lattice concept for better understanding of lattice based cryptographic systems. NTRU background is included here as well. Section 3 contains the NTRU key generation process explanation and three GPU optimizations for execution acceleration. Section 4 demonstrates the experimental results and gives an analysis of the ideal scenario and bottleneck for each implementation. Section 5 describes related work. The conclusion and future work are given in Section 6.

2. NTRU: A Lattice-Based Cryptographic System

Programmability in heterogeneous distributed systems has become a critical issue as Clouds gains its popularity. It will determine the effectiveness of utilizing different system resources and easiness of writing code in distributed environments.

2.1. *Euclidean Vector Space*

Euclidean vector space E^n is a finite dimension space formed by lines or vectors. Each single vector in the space is represented by Cartesian coordinates [1], satisfying Euclidean space properties such as Euclidean translation [7] and Euclidean rotation [8].

Due to the complexity and abstractness of Euclidean vector space, real coordinate space R^n is defined with N real number coordinates for each vector. For instance, in dimension one, the vector space is actually the real number line in mathematic geometry.

2.2. *Lattice*

Lattice is the set of vectors such that every vector in the lattice could be represented as a linear combination of the lattice basis.

$$\mathscr{L}(b_1, b_2 \dots b_n) = (\lambda_1 b_1 + \lambda_2 b_2 \dots \lambda_n b_n) \lambda_i \in \mathbb{Z}^n$$

The basis of a lattice is a set of linear independent vectors ($b1$, $b2$, ... bn), which means scalar (λ_1, λ_2, ...λ_n) $\lambda \in \mathbb{Z}$ is not exist such that

$$\lambda_1 \begin{bmatrix} b_{11} \\ b_{21} \\ \vdots \\ b_{n1} \end{bmatrix} + \lambda_2 \begin{bmatrix} b_{12} \\ b_{22} \\ \vdots \\ b_{n2} \end{bmatrix} + \cdots \lambda_n \begin{bmatrix} b_{1n} \\ b_{2n} \\ \vdots \\ b_{nn} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

2.3. *Lattice-Based Hard Problems*

Most famous lattice-based hard problems include Shortest Vector Problem (SVP) and Closest Vector Problem (CVP).

Shortest Vector Problem (SVP): given basis of rank d, B^d and corresponding generated lattice $\mathscr{L}(B^d)$, a non-zero vector v is returned such that:

$$||v|| = min\{||v|| \mid v \in \mathscr{L}(B^d)\}$$

Closest Vector Problem (CVP):: given a randomized vector r, rank d basis B^d and corresponding generated lattice $\mathscr{L}(B^d)$, a vector v is returned such that:

$$distance(v,r) = min\{distance(v,r) \mid v \in \mathscr{L}(B^d)\}$$

2.4. *NTRU*

Nth-degree truncated polynomial ring (NTRU) lattice-based cryptographic system was devised by J. Hoffstein, J. Pipher and Joseph H. Silverman in 1996. For now, NTRU is a patented cryptographic system owned by Security Innovation [10]. Besides the outstanding security level, NTRU is also holding the eye for its scalability on platforms with limited resources due to its low power consumption and fast encryption speed.

2.4.1. NTRU Key Generation

The degree of truncated polynomial ring N is set to be a prime integer. Normally, number 503 is viewed as a relatively high security level degree number. Vectors f and g are randomly chosen from the truncated polynomial ring with small coefficients, whereas p and q are a co-prime integer pair for coefficient modular operations.

In practical implementations, the traditional value for p is 3 and q is a power of 2 although the values depend on the different implementation environments. f_q^{-1} and f_p^{-1} are the multiplicative inverses of f with the modular operation on each coefficient.The formula for key generation is expressed as the following, where the * operation is the convolution production defined above.

$H = p * f_q^{-1} * g \, mod \, q$

2.4.2. NTRU Encryption

Before encryption, data is first translated into the polynomial form M. There could be different approaches on heterogeneous platforms as the encryption procedure is not defined in NTRU algorithm, but implementation details such as execution time and memory space occupation should be handled deliberately. Additionally, a random polynomial r is applied to achieve better diffusion, obscuring the correlation between the plaintext and ciphertext. The formula for encryption is expressed as follows:

$E = r * H + M \, mod \, q$

2.4.3. NTRU Decryption

The receiver keeps polynomial f and its multiplicative inverse f_p^{-1} as the private key. Decryption occurs in the following three steps:

$D_1 = E * f \, mod \, q$

$D_2 = D_1 \, mod \, p$

$D_3 = f_p^{-1} * D_2 \, mod \, p$

To demonstrate that the decryption works, recall that in the encryption and key generation formulas, H is substituted by $p * f_q^{-1} * g$ into the encryption formula. A substituted E is taken into the formula of D_1, $D_1 = pr * g + f * M \, mod \, q$. The modular operation in the second step cancels the polynomial $pr * g$ and leaves $f * M \, mod \, q$ as the result of D_2. In the last step, $f_p^{-1} * f \, mod \, p$ is equalent to 1 mod p by the definition of multiplicative inverse, such that $D_3 \Rightarrow 1 * M \, mod \, p$.

2.4.4. NTRU Security Analysis

Cryptanalysis against NTRU could be treated as a momentous resource to assess the security level of NTRU cryptographic system. Three types of attacks against NTUR have been observed.

A brute force attack against NTRU is technically equalent to an exhaustive search of the small norm vectors f and g in a given lattice. The convolution production is calculated and compared with public key H for a possible match. As the degree of lattice increases to a certain level, the lattice volume will explode and the number of involved vectors will increase in a geometrical progression. Thus, brute force attack could not compromise the system in polynomial time even extraordinary computing power is provided.

Basis reduction [12] attack aims at the lattice structure of the NTRU cryptographic system. It regenerates the similar spatial structure lattice with better basis vectors, which is normally to have a small norm and orthogonal vectors. But, several researchers have pointed out that for some intentionally designed parameter combinations, basis reduction algorithm might not work in polynomial time.

Quantum computing [3] attack is a brand new form of attack, which combines a quantum computer and a quantum computing algorithm such as Shor's algorithm [13]. There is no theoretical proof that lattice based cryptographic systems such as NTRU could entirely defend against a quantum computing attack. However, several researchers' experiments have indicated that there seems to be no performance breakthrough in solving lattice problems with quantum computers. Thus, it is entirely possible that lattice structure has certain resistance against quantum computing attacks for now.

2.5. CUDA

CUDA (Compute Unified Device Architecture) is a programming paradigm developed by NVIDIA [9] to ease parallel programming in GPU. CUDA has abandoned the inflexible Graphics APIs to interact with GPU and allowed programmers to use the traditional C-style language to declare kernel functions, which will be executed directly on GPU CUDA cores. With the involvement of integer calculations, bitwise operations, and full utilization of hardware, CUDA has transcended former methods in dealing with GPU computing.

3. Acceleration of NTRU Cryptographic System

Optimizations of NTRU encryption algorithm are accomplished in three approaches: Single GPU, Single GPU with Zero Copy, and Multi-GPU optimizations.

To overcome NTRU's inefficient CPU implementation, Single GPU version uses one GPU as a coprocessor to achieve data parallel execution. Single GPU with Zero Copy version utilizes CUDA zero copy technique [11] to ameliorate the data transfer pressure. Dual-GPU version is an example of Multi-GPU optimization to achieve device level parallel processing.

3.1. NTRU's CPU Implementation

Traditionally, the NTRU encryption, denoted as $E = r * H + M \, mod \, q$, is processed by CPU. For the fixed public key H, the entire encryption process can be viewed analogously to a vector addition problem. Typically, a *for* loop is applied to encrypt each message unit into ciphertext. Although CPU-based implementation has adopted Instruction Level Parallelism (ILP), Single Instruction Multiple Data (SIMD) and Single Instruction Multiple Thread (SIMT) to ameliorate the inefficient hardware utilization and boost system throughput, further performance improvement is fatally restricted by the number of executing hardware pipelines, which will determine the maximum number of instructions that can be run simultaneously. To exploit the massive parallelism in NTRU encryption, GPU turns out to be a possible alternative.

3.2. NTRU Key Generation

Since NTRU key generation is a one-time process, normally it is done on CPU side. Major NTRU encryption parameters include the follows:

- Dimension size $N = 503$
- Modular operation integer $p = 3$, $q = 256$
- Private key: random polynomial f and its multiplicative inverse f_p^{-1}
- Public key: $H = pf_q * g \, mod \, q$

Basic operations are polynomial modular addition, denoted as $+$, and polynomial convolution production, denoted as $*$.

Polynomial modular addition is demonstrated in Algorithm 1, as shown in Fig. 1. The corresponding coefficients (with the same degrees) from two polynomials are added up together. Then the modular operation is performed.

```
Algorithm 1 polynomial modular addition
Input: { polynomial A, B and integer p}
1:      for i = 0 to N -1 do
2:          C[i] = (A[i] + B[i]) mod p // coefficient all in domain $Z_p$
3:      end for
Output: { Polynomial_addition returns sum of A and B}
```

Fig. 1. polynomial modular addition

Algorithm 2 shown in Fig. 2 defines the convolution production operation. The original polynomial multiplication is conducted in the first loop as the kth coefficient of polynomial $C[k]$ is equal to $\sum_{i=0}^{i=N-1} B_i A_{k-i}$. The truncated polynomial ring requires the most significant degree of every polynomial must be $N-1$. Thus, based on the observation of polynomial long division [14] consequence, the coefficients with degrees higher than $N-1$ should be mapped back to the lower degrees in the domain. This adjustment process is done by the second loop in Algorithm 2.

Algorithm 2 convolution production

```
Input: { polynomial A, B and integer p}
1:  for k = 0 to 2N-2 do
2:      for i = 0 to N-1 do
3:          if k - i >= 0 and k - i < N then
4:          C[k] = C[k] + B[i]A[k - i]
5:      end for
6:  end for
7:  for k = N to 2N-2 do
8:      C[k - N] = C[k - N] + C[k]
9:  end for
Output: { polynomial C, the convolution production of A and B}
```

Fig. 2. convolution production

Private key polynomial f is a randomly generated polynomial with small coefficients. The other part of the private key is the multiplicative inverse of polynomial f in domain $\mathbb{Z}_p$, and Extended Euclidian Algorithm is applied to calculate the multiplicative inverse [2].

Public key is generated in two separate steps. First, convolution production of g and f_q is calculated and g is a randomly chosen polynomial as well. Second, the result is multiplied with integer p and the modular operation is applied to coefficients of all degrees in domain $\mathbb{Z}_q$.

3.3. Single GPU Version with Zero Copy

CUDA's zero copy technique is designed to ameliorate the overwhelming pressure on data transfer by overlapping two different directional memory copies and kernel executions.

Zero copy apportions page locked memory space in host memory rather than GPU global memory. Since such memory space is also mapped to GPU memory space, GPU programs can directly access it without explicit data movement operations.

Algorithm 3, shown in Fig. 3, has briefly described the implementation layout where cudaHostAlloc is called to allocate pinned memory on host and map it into the device's address space through cudaHostGetDevicePointer. Then, the kernel function can be executed directly since data will be fetched from memory as needed. Additionally, CUDA threads need to be synchronized to make sure that all asynchronous memory access requests have been finished.

Algorithm 3 single gpu-zero copy implementation

```
Input: { NTRU parameters N, p, q, public key H}
1:  Allocate pinned memory on host, size = data_size
2:  Map Host page locked memory to device address space
3:  Do kernel execution  // device pointer direct access host memory
4:  Synchronize threads on device
Output: { encrypted data E is directly written to host memory}
```

Fig. 3. single gpu-zero copy implementation

Traditionally, CPU memory is allocated as pageable memory such that the operating system could create an illusion of a much larger virtual memory. However, when GPU DMA engine executes cudaMemcpy operation, it will first allocate a temporary page locked memory and then copy the required data from pageable memory into a temporary page locked buffer. Finally, the DMA engine carries data through the PCIe bus to the device and frees the temporary buffer. With zero copy technique, this extra data copy process is avoided and thereby data transfer time decreases.

Furthermore, zero copy automatically overlaps data transfer and kernel execution. Although memory copy still exists, it happens only if any threads request memory access in host page locked memory. Such asynchronous memory access could be processed while other threads maintain kernel execution. Consequently, such overlapping might reduce overall communication latency.

3.4. Single GPU Version with Data Transfer

Other than the strategy of leaving data in host pinned memory, the traditional approach is to move data into GPU memory for processing. GPU global memory is selected due to its relatively large capacity and direct usage, unlike shared memory which requires extra data movements.

Coalesced memory access could lessen latency to a certain degree. When each thread accesses global memory, it retrieves a data block instead of one data item. If other threads can reutilize other items in the same data block, several memory accesses could be avoided and the total latency is reduced.

Single GPU version starts with the CudaMalloc() function call to dynamically allocate memory space in GPU global memory. Then CudaMemcpy() copies data from CPU memory to the just allocated GPU global memory. The kernel function will not start until all required data has been transferred, i.e., the memory copy operation is synchronized.

Hardware restricts the maximum number of threads inside one block and registers that can be allocated for each thread. For a complicated application involving many operations for each single thread, larger register sizes may benefit performance. Otherwise, simpler applications such as NTRU encryption prefer to use a large number of threads and blocks, reducing the frequency of execution repetition for better performance.

```
Algorithm 4 single gpu implementation
Input: { NTRU parameters N, p, q, public key H}
1:    Allocate device global memory
2:    Copy data to be encrypted from host to device
3:    Do kernel execution
4:    Copy encrypted data back to host memory
Output: { encrypted data E}
```

Fig. 4. single gpu implementation

In NTRU encryption, data for encryption has been laid out in adjacent area and memory coalescing is exploited thoroughly. Since data movement and execution are scheduled in separate phases, no overlapping is achieved. The single-GPU version design is depicted in Algorithm 4, shown in Fig. 4.

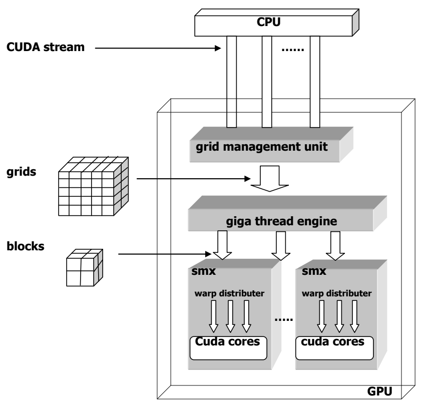

Fig. 5. Task scheduling in Nvidia Kepler GPU

Logically there are three steps to finish the encryption process: data transfer from CPU to GPU memory, GPU kernel execution, and data transfer from GPU back to CPU memory. As shown in Fig. 5, all operations are packed into a CUDA stream, a logical single direction channel from host to device. On GPU side, data transfer operations are passed to DMA engines and kernels are taken over by the Grid Management Unit, which launches kernels and generates job grids, partitioned into blocks. These blocks depend on the parameters in the kernel function call. Giga Thread Engine assigns blocks to different SMXs. Several blocks from same or different kernels can be assigned to one SMX where four Warp Schedulers (in Kepler) inside SMX are in charge of the thread level scheduling. The basic scheduling unit is warp with 32 bound threads which execute the same instructions.

GPU code is first compiled by the NVCC compiler into a virtual assembly language PTX [5], and then translated into machine code. The Warp Scheduler analyzes each instruction and assigns a warp to execute on a particular hardware pipeline, such as a stream processor (CUDA core), LD/ST unit or Special Function unit, based on the instruction types.

3.5. Multi-GPU Version

Multi-GPU version dispatches workload across multiple GPUs to reduce execution time through computation overlapping. Key generation process still remains on CPU side and only data processing is ported to multiple GPUs. Intricate data dependency must be taken into consideration during data division. For NTRU encryption, all elements for encryption are independent to each other. Then, equal division is acceptable here.

One crucial practical misapprehension is intuitively executing the encryption algorithm on two devices. Recall the CUDA stream concept, such a manipulation will lead to a sequential execution of two GPUs, because all of the operations are packed into a single logical path to Fermi or earlier GPUs and each operation in a CUDA stream is handled sequentially.

Algorithm 5 multi-gpu implementation

```
Input: {NTRU parameters N, p, q, public key H}
1:   create CUDA streams for different devices
2:   for i=0 to device_number-1
3:       memory copy from host to device on stream i
4:   end for
5:   for i=0 to device_number-1
6:       do kernel execution on stream i
7:   end for
8:   for i=0 to device_number-1
9:       memory copy from device to host on stream i
10:  end for
Output: {encrypted data E}
```

Fig. 6. multi-gpu implementation

To avoid sequential execution, two separate CUDA streams are generated and data transfer and kernel operations for different devices are packed into their corresponding streams. Algorithm 5, shown in Fig. 6, demonstrates multi-stream strategy for Fermi GPU. Operations are dispatched into different streams without blocking each other since Fermi GPU will merge all streams in one logical channel. However, Kepler's Hyper-Q technique eliminates this three-loop handling since scheduler will scan all streams simultaneously.

Performance improvement relies on platform hardware support, which may increase the cost of this optimization; however, as the number of co-processors grows, even better performance could be achieved.

4. Experimental Results

Performance analysis and comparison have been conducted for four NTRU implementations: single CPU, single GPU with zero copy, single GPU with data transfer and multi-GPU versions.

CPU version is implemented in C language, while the other three optimizations are programmed in CUDA. The testing platform is equipped with an Intel Xeon E5-2620 processor and two GeForce GTX 680 GPUs. Execution time is acquired by the *gettimeofday* function call and *cudaEvent* API, respectively.

The execution time comparison only involves NTRU encryption while the public key generation is omitted. Then, the execution platform will be identical in all four cases for fair comparison. Moreover, the time consumed for data transfer is included in the GPU implementations, since GPU optimizations should pay for the penalty of extra data movement. The comparison result is shown in Fig. 7.

4.1. CPU Version

The CPU version does not adopt any parallelization techniques such as multithreading. It merely relies on CPU hardware and scheduling algorithms to actualize instruction-level execution acceleration.

As Fig. 7 implies, the CPU implementation achieves the best performance when the encrypted message data size is smaller than eighty million bytes due to optimized scheduling techniques such as ILP and the lack of extra data movement. However, as the data size reaches one billion byes, the execution time of CPU version begins to grow due to the limited number of CPU computing pipelines. CPU hardware fails to exploit the further parallelism in big data and achieves almost linear execution time increasing as data expands.

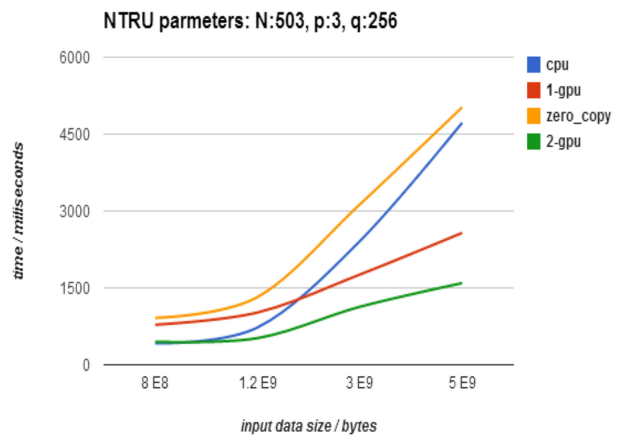

Fig. 7. Performance Comparison with Four Schemes

4.2. Single GPU with Zero Copy

The performance of single GPU version with zero copy is thwarted since its execution time is even longer than the CPU one.

Logically the entire amount of data transferred from CPU to GPU should be the same no matter if zero copy technique is applied or not. For GPU kernel execution, thread operations are arranged by warp schedulers dynamically. Zero copy intends to incur overlapping by allowing data transfer on PCIe buses and kernel execution on GPU at the same time.

However, frequently accessing host page locked memory might reduce memory access efficiency. In frequent data transfers, PCIe bus might not be fully utilized since threads issue memory access arbitrarily. Such under-utilization of data results in more data access trips than the traditional *cudaMemcpy* approach. Additionally, laconic operation of NTRU encryption results in tiny kernel execution time, which reduces the benefit from overlapping dramatically. As result, single GPU with zero copy fails to achieve any speedup.

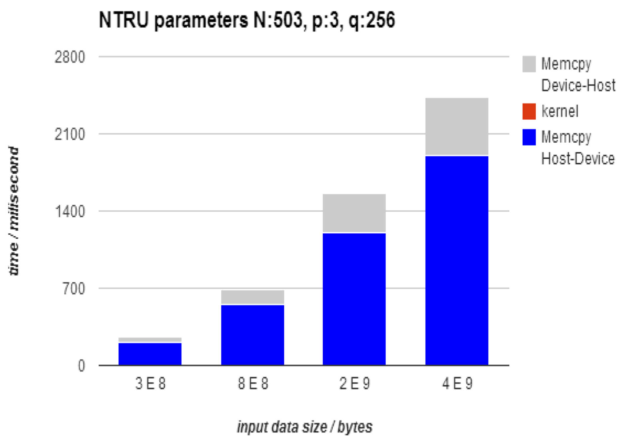

Fig. 8. Execution Breakdown of Single GPU Version

4.3. *Single GPU with Data Transfer*

When input data size is small, the execution speedup achieved by the parallelism exploitation on GPU fails to compensate the loss in GPU memory allocation and data transfer. As in Fig. 7, when data is smaller than 1.5 billion bytes, CPU version outperforms the single GPU one with data transfer. However, when data size is larger than two billion, the time saved by the parallel execution on GPU exceeds the extra data transfer time. Therefore, overall speedup becomes quite obvious. When the data size reaches five billion, the single GPU version only uses half of the time spent with CPU implementation.

Even though the single GPU version beats the CPU implementation for large data size, it is hard to achieve further performance gains. Data transfer overhead dominates the overall execution time. Overhead breakdowns are shown in Fig. 8 for different data sizes. The time of data movement from CPU to GPU is longer than the one from GPU to CPU due to the GPU global memory allocation operations. For NTRU encryption, the kernel execution time is almost ignorable.

4.4. *Multi GPU Performance*

Since zero copy technique does not bring in any performance gain for NTRU in single GPU version, the Multi-GPU one adopts the traditional data transfer approach. As shown in Fig. 7, a dual-GPU version achieves 3 times speedup over the CPU version and 1.25 times speedup over single GPU (with data transfer) in a case with 5 billion bytes input data.

The overall NTRU encryption time includes communication and computation since they are not overlapped. The dual-GPU version can cut the computation time in half, not the overall encryption time which is still dominated by data transfer time. Two times speedup cannot be achieved in data transfer since the time for GPU memory allocation, execution environment preparation, and combination of partial results will not increase linearly as the number of GPUs increases.

5. Related Work

Lattice basis reduction algorithm is serving as a theoretical foundation in resolving the hard lattice problem. Thijs Laarhoven [6] classified lattice reduction algorithms into different categories and analyzed their practical performance in solving particular lattice problems. Connections between the lattice problem and logical conversions of basis reduction algorithms were thoroughly explored. Moreover, improved enumeration and sieving techniques helped analyze hard lattice problems in different viewpoints. Finally, the long execution time in practically solving lattice problems has proved the security of lattice based cryptosystems from the other side.

Efficient NTRU Implementations [4] by Colleen consist of both software and hardware designs. In the software design, involved operations such as addition, convolution production, random polynomial generation, and multiplicative inverse calculation have all been explained. Additionally, an improved version of the convolution production was proposed for better scalability. The hardware pipeline design was developed base on the analysis of NTRU algorithm features. NTRU multiplier and adder were proposed for higher CPI (Cycles per Instruction) and more efficient hardware utilization over the traditional CPU pipeline.

6. Conclusions and Future Work

As Nth-degree truncated polynomial ring (NTRU) becomes a major quantum-resistant candidate in lattice based cryptographic systems, its processing speed turns in to a major concern. This paper analyzes NTRU structure and utilizes Graphics Processing Unit (GPU) to accelerate NTRU encryption. The single GPU version parallelizes NTRU encryption on GPU to overcome the CPU hardware restriction. Although considerable execution speedup has been achieved, relatively narrow data transfer bandwidth between the host and device impedes further performance gains. Multi-GPU version implants device-level parallelism to promote both data transfer efficiency and computation capability.

GPU zero copy technique has been used to overlap the communication and computation in CPU/GPU bus (PCIe) and GPU itself. However, the factual performance is undermined by high frequent data transfer requests and low bus utilization. Obviously, NTRU is not a good candidate for such optimization.

The future work includes pushing NTRU to distributed systems in Big Data cases and supporting NTRU with OpenCL across heterogeneous platforms for flexibility and scalability.

References

1. D. Cremer and J. A. Pople, General definition of ring puckering coordinates, Journal of the American Mathematic Society, pp. 1354–1358, 1975.
2. Andreas Enge, The extended euclidian algorithm on polynomials and the computational efficiency of hyperelloptic cryptosystems, Designs Codes and Cryptograhpy, 2001
3. Jozef Gruska and Czech Republik, Quantum Computing, 2004
4. Fei Hu, Kyle Wilhelm, Michael Schab, Marcin Lukowiak, Stanislaw Radziszouske, and Yang Xiao, Ntru-based sensor network security: a low-power hardware implementation perspective, Security and Communication Networks, pp. 71–81, 2009.
5. Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili, A characterization and analysis of ptx kernels, In Workload Charaterization, 2009
6. Thijs Laarhoven, Joop van de Pol, and Benne de Weger, Solving hard lattice problems and the security of lattice-based cryptosystems, IACR Cryptology ePrint Archive, 2012
7. Peter D. Lax and Ralph S. Phillips, Translation representations for automorphic solutions of the wave equation in non-euclidean spaces, In Communications on pure and applied mathematics, pp. 303–328, 1994.
8. Shahn Majid, q-euclidean space and quantum wick rotation by twisting, Journal of Mathematical Physics, pp. 5025–5034, 1994.
9. CUDA Nvidia, Programming guide, 2008
10. Kazuhiko Ohashi, Security innovation on several assets under asymmetric information, Japanese Economic Review, pp. 75–95, 1999.
11. Jason Sanders and Edward Kandrot, Cuda by example, An Introduction to General-Purpose GPU Programming/J. Sanders, E. Kandrot-Addision Wesley Professional, 2010.
12. Claus-Peter Schnorr and Martin Euchner, Lattice basis reduction: improved practical algorithms and sloving subset sum problems, Mathematical programming, pp. 181–199, 1994.
13. Lieven M. K. Vandersypen, Matthias Steffen, Gregory Breyta, Costantino S. Yannoni, Mark H. Sherwood, and Isaac L. Chuang, Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance, Nature, pp. 883–887, 2001.
14. Stanislaw H. Zak and Kai Hwang, Polynomial division on systolic arrays, IEEE Trasnsactions on Computers, pp. 577–578, 1985.