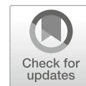
A FPGA-Based Heterogeneous Implementation of NTRUEncrypt



Hexuan Yu, Chaoyu Zhang, and Hai Jiang

1 Introduction

The tendency of quantum computing has been proved to imperil the majority of classical cryptography schemes. Under these circumstances, the post-quantum cryptography schemes that are resistant to quantum computing have attracted people's notice, which leads to the demand for innovations. Lattice-based cryptography is a promising quantum-safe cryptography family [1], both in terms of fundamental security properties and applicability to both traditional and emerging security problems such as digital signature, encryption/decryption, key exchange, and fully homomorphic encryption (FHE). NTRU (Nth-degree truncated polynomial ring unit) cryptosystem has been fully accepted into the IEEE P1363 standards as part of the specifications for lattice-based public-key cryptography (IEEE P1363.1) since 2009.

However, compared to classic number-theoretical cryptosystems, the computational complexity of NTRU is still relatively high, which may impede their large-scale application. The traditional implementations of NTRUEncrypt, as well as other cryptosystems, concentrate upon conventional processors such as CPUs, whereas, there were some fundamental problems which prevent those cryptosystems from increasing operations per watt including: (1) Memory wall. The incompatible frequency between memory and processor has become a bottleneck. It may take several clock cycles between initiating a request and retrieving data. (2) Data dependency between operations and software/hardware constraint prevent parallelization.

In the era of heterogeneous computing, the special-purpose computing device can be accessed by the CPU to offload some computations to achieve lower cost

H. Yu (✉) · C. Zhang · H. Jiang
Department of Computer Science, Arkansas State University, Jonesboro, AR, USA
e-mail: hexuan.yu@smail.astate.edu; chaoyu.zhang@smail.astate.edu; hjiang@astate.edu

© Springer Nature Switzerland AG 2021

H. R. Arabnia et al. (eds.), *Advances in Parallel & Distributed Processing, and Applications*, Transactions on Computational Science and Computational Intelligence, https://doi.org/10.1007/978-3-030-69984-0_34

or higher power efficiency. Some former GPU versions [2, 3] have clarified that NTRUEncrypt can benefit a lot from parallelization accelerators. FPGA (field-programmable gate array) has been widely explored as a hardware accelerator due to its reconfigurability and fast turnaround time, and it is one of the most cost-effective devices where tasks can be parallelized in significant measure.

This work explores the inherent parallelism of the NTRU-Encrypt cryptosystem on an FPGA-based heterogeneous system (CPU+FPGA) using OpenCL (open computing language) to maximize the throughput of NTRUEncrypt, considering the FPGA resource constraints such as on-chip memory, logic block, registers, and memory bandwidth. OpenCL is a parallel and cross-platform processing standard from the Khronos Group. Utilizing OpenCL on an FPGA can reduce the time-cost comparing with hardware description language (HDL) development and offer significantly higher performance than the one on other hardware devices such as CPUs, GPUs, and DSPs at much lower power consumption.

The rest of this paper is organized as follows.

Section 2 briefly describes the mathematics background and relevant procedures of NTRUEncrypt, including key generation and encryption/decryption. Section 3 introduces the framework of OpenCL-based NTRUEncrypt cryptosystem and our programming model. The implementation and results analysis, including the comparison between FPGA and GPU implementations, are given in Sects. 4, 5, respectively. The results show that NTRUEncrypt is computationally inexpensive within FPGA. The conclusion is drawn in Sect. 6.

2 NTRUEncrypt

In the past few decades, lattice-based cryptography has become one of the most intriguing areas of mathematical cryptography. The NTRUEncrypt is based on the shortest vector problem of lattice theory [4].

2.1 Notation

NTRU operations are based on objects in a truncated polynomial ring $R = Z[X]/(X^N - 1)$ with convolution multiplication, and all polynomials in the ring have integer coefficients and degree at most $N - 1$ with $X^N \equiv 1$:

$$a = a_0 + a_1X + a_2X^2 + \ldots + a_{N-2}X^{N-2} + a_{N-1}X^{N-1} \tag{1}$$

The product

$$C(X) = a(X) * b(X) \tag{2}$$

is given by

$$C_k = a_0 b_k + a_1 b_{k-1} + \ldots + a_{N-1} b_{k+1} = \sum_{i+j=k \bmod m} a_i b_j \tag{3}$$

In particular, if we write $a(X)$, $b(X)$, and $c(X)$ as vectors

$$a = [a_0, a_1, \ldots a_{N-1}], b = [b_0, b_1, \ldots b_{N-1}], c = [c_0, c_1, \ldots c_{N-1}] \tag{4}$$

then $c = a * b$ is the convolution product of two vectors having a size of N positions.

The NTRU algorithm is defined by the following parameters:

- N. The degree parameter, defining the degree $N - 1$ of the polynomials in R.
- q. A large modulo. Polynomial coefficients are reduced modulo q.
- p. A small modulo. The coefficients of the message are reduced modulo p in decryption.
- $\mathcal{L}_f$. Private key space, fixing the polynomial form to define the number of positive ones for the private key f. The negative ones are fixed by $d_f - 1$.
- $\mathcal{L}_g$. Public key space, fixing the polynomial form to define the number of positive and negative ones for the random polynomial g used to calculate the public key.
- $\mathcal{L}_r$. Blinding value space, fixing the polynomial form to define the number of positive and negative ones of the random polynomial r used in the encryption process.
- $\mathcal{L}_m$. Plaintext space. NTRUEncrypt requires the message to be in a polynomial form, therefore the need for d_m to define the form of the message to be encrypted.

NTRU is specified by three public integer parameters (N, p, q) which represent the maximal degree $N - 1$ for all polynomials in the truncated ring R, a small modulus, and a large modulus, respectively, where it is assumed that N is prime, q is always larger than p, and p and q are co-prime. Note that p and q need not be prime, and four sets of polynomials $\mathcal{L}_f$, $\mathcal{L}_g$, $\mathcal{L}_r$, and $\mathcal{L}_m$ (a polynomial part of the private key, a polynomial for the generation of the public key, the message, and a blinding value, respectively) are all of the degree at most $N - 1$ [5].

2.2 Key Generation

The key generation includes the generation of the private key (f, f_p) and the public key h [6]. Choose random polynomials f and g from R with small coefficients, typically $\{-1, 0, 1\}$ for $p = 3$. Then compute f_p, i.e., the inverse of f mod p defined by

$$f * f_p = 1 \bmod p \tag{5}$$

Bob creates a public key h by choosing elements $f, g \epsilon R$, computing the mod q inverse f_q^{-1} of f , and setting

$$h \equiv f_q^{-1} * g \bmod p \tag{6}$$

Bob's private key is the element of f. Bob also precomputes and stores the mod p inverse f_q^{-1} of f.

2.3 Encryption

In order to encrypt a plaintext message $m \epsilon R$ using the public key h, Alice selects a random element $r \epsilon R$ and forms the ciphertext

$$e = r * h + m(\text{mod} q) \tag{7}$$

This ciphertext hides Alice's messages and can be sent safely to Bob.

2.4 Decryption

In order to decrypt the ciphertext e using the private key f, Bob first computes

$$a \equiv f * e(\text{mod} q) \tag{8}$$

He chooses $a \epsilon R$ to satisfy this congruence and to lie in a certain prespecified subset R_a of R. He next does the mod p calculation

$$f_p^{-1} * a(\text{mod} p) \tag{9}$$

and the value he computes is equal to m modulo p.

3 A Heterogeneous NTRUEncrypt Platform

The OpenCL standard is an open programming model for accelerating algorithms on a heterogeneous computing system. OpenCL extends the C-based programming language for developing portable codes on different platforms such as CPU, GPU, DSP, and FPGA. In this section, we describe our CPU-FPGA heterogeneous framework of NTRUEncrypt Cryptosystem. As shown in Fig. 1, our OpenCL system

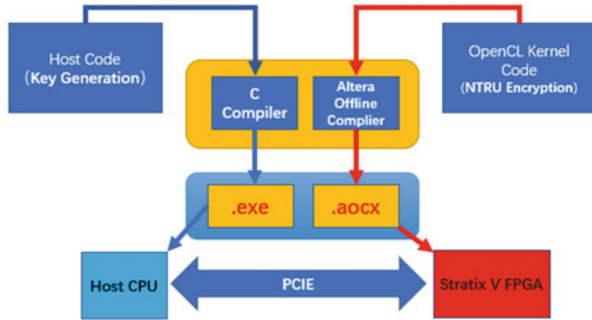


Fig. 1 FPGA Implementation of NTRU using OpenCL

implementation is divided into two components: the host PC and FPGA device. More details will be described as follows.

3.1 Overview

Where FPGAs shine in aspects of energy efficiency is at configurable logic and fixed precision computations. In lattice-based cryptography, it is exactly this property that makes FPGAs advantageous. The aim of this design is to mitigate the efficiency problem by providing an FPGA-based implementation that can run NTRUEncrypt cryptosystem on parallel hardware accelerators.

As shown in Fig. 1, the NTRUEncrypt Cryptosystem written in OpenCL consists of two parts: a host program for initialization and management and kernels that define the compute-intensive tasks. The host is responsible for device memory management, transferring data to devices, queuing work for devices, and error management. The host CPU communicates with the Stratix V FPGA via a PCI-E interface. The host can then get and utilize the encryption results, again via the PCI-E interface. OpenCL simplifies the transformation methodology of turning logic into FPGAs. As we have seen in Sect. 2, the operations consisted in this cryptosystem primitives are modular addition and multiplication operations; there contain neither logarithmic nor exponentiation operations. The polynomial multiplication and addition of NTRU encryption correspond to XOR and AND operations, respectively. Hence, they are easier to implement. However, this advantage is accompanied by the cost of intensive computation and storage requirements which may impede the adoption of NTRU over number-theoretic cryptosystems. In this case, the optimization of massive XOR and AND operations is of vital importance to a 256-bit level NTRU encryption.

3.2 Programming Model

OpenCL supports three programming models: data-parallel programming model, task-parallel programming model, and the hybrid of the two. Generally, NTRU encryption can be executed in both data-parallel and task-parallel programming models. This is because, firstly, the size of its input/output per loop is fixed and decided by chosen parameters and all the input plaintext will be used by the same encryption logic, which allows NTRU encryption to be executed in data-parallel form. Secondly, the XOR and AND operations toward its single bit are independent; thus it is also capable of being executed in a task-parallel fashion during each cycle.

Compared to CPU and GPU which execute the kernel on different cores, FPGA offers advantages by transforming the kernel into dedicated and pipelined hardware circuits. Data-parallel portions of the NTRUEncrypt algorithm are executed on FPGA as kernels, which are OpenCL functions.

Our design aims to minimize the number of resources, thus achieving higher energy efficiency while sustaining the same throughput. How much an algorithm uses the FPGA is an important aspect of FPGA programming. An algorithm only utilizes the logic it requires within the FPGA, leaving the remaining logic unused and consuming minimal power. Therefore, large power savings can be achieved by designing a kernel to meet only the needs of the target system, something that is not possible on CPU and GPU technologies.

In this heterogeneous system, the host launches encryption kernels across a 2D grid of work-items to be processed by the FPGA. Conceptually, work-items can be thought of as individual processing threads that each execute the same kernel function. Work-items have a unique index within the grid and typically compute different portions of the result. Work-items are grouped into work-groups, which are expected to execute independently from one another. The parallelism can be achieved since the kernel can be duplicated inside FPGA using pipelines.

Before executing the NTRUEncrypt kernel, Altera OpenCL offline compiler compiled the kernel code into *.aocx file. Altera SDK for OpenCL creates several I/O interfaces such as memory controller to read and write data to external DDR memory and internal memory. On top of that, it creates the PCIe communication link for data communication and kernel code invocation between host and FPGA. A high-level representation of the OpenCL system implemented on FPGA can be seen in Fig. 2.

3.3 Memory Hierarchy

The OpenCL kernels have access to four distinct memory regions distinguished by access type and scope [7]. Global memory allows read-and-write access from all work-items across all work-groups. Local memory also provides read-and-write access to work-items; however, it is only visible to other work-items within the

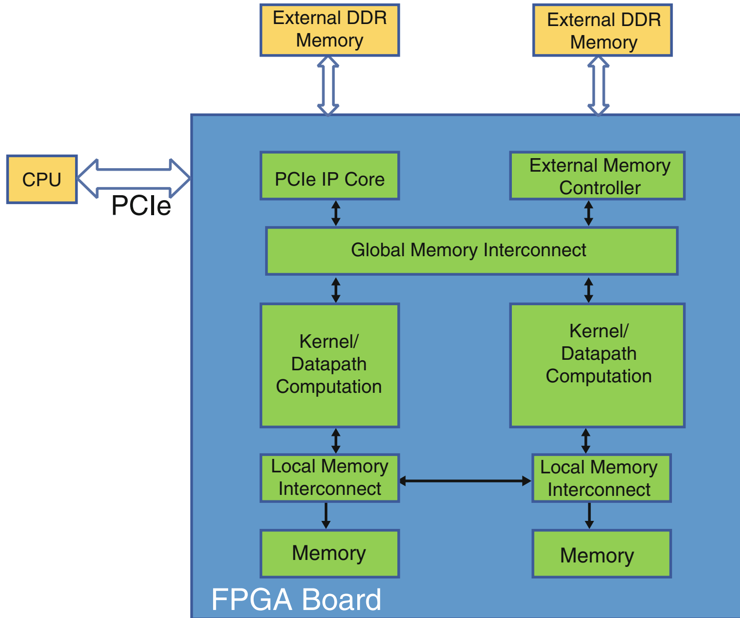


Fig. 2 OpenCL System on FPGA

same work-group. Constant memory is a region of read-only memory accessible to all work-items, thus immutable during kernel execution. Lastly, private memory provides read-and-write access visible to only individual work-items. Generally, its access speed is the fastest among the four memory regions.

NTRU encryption includes two kinds of data: plaintexts and key. In the first place, all the plaintexts and key are stored in the main memory of host. These data will be transferred into global or constant memory of the FPGA before execution. In order to maximize the performance, it is better to store these data in memory for the sake of speed. However, data features should be carefully considered during the memory distribution.

On FPGA, all local memory resources on FPGA are user-managed, and no explicit cache hierarchy exists, which is totally different from CPU and GPU. Offline compiler must generate complicated arbitration logic to deal with the memory access requests. One of the solutions to overcome memory bottleneck is to explicitly cache the elements of r and h in constant or local memory. Especially for r, this turns out to be a good strategy, since each *long* contains 32 coefficients, thereby reducing the number of accesses to global memory with a factor 32.

4 Implementation and Optimization

4.1 Key Generation

Before invoking NTRUEncrypt key generation or encryption, a deterministic random byte generator (DRBG) should be instantiated. The DRBG in this project implements the ANS X9.82 Part 3-2007 standard, using HMAC_DRBG. The security level of DRBG should be equal to or greater than the security level of NTRU parameters. Thus, 256-bit levels have been adopted for DRBG. The DRBG instantiation function returns a handle that we can pass to the key-generation and encryption functions.

The selection of the NTRU PKC parameters defines the different levels of security. Also, p and q must have no common factors. To provide security level as high as possible, we choose the parameter set NTRU_EES743EP1, which is an IEEE 1361.1 parameter set that gives 256 bits of security and is a trade-off between key size and encryption/decryption speed. The equivalent security level of RSA is 15,360 bits and ECC is 512 bits.

Major parameters and operation of this step are listed as follows:

- The degree parameter: $N = 743$
- Modular: $p = 3, q = 2048$
- Create private key: the key pair (f, f_p)
- Create public key: $h \equiv f_q^{-1} * g(\text{mod} q)$.

The algorithm assumes $q = 2^w$ so the reduction will be performed by extracting the lower w bits. In the mean time, f is a randomly generated polynomial with small coefficients, and $f^{(-1)}$ is the multiplicative inverse of f and computed using the extended Euclidean algorithm in this work. The key setup and the random keys generation are done offline by the host.

Algorithm 1 NTRU key generation

1: KeyGen $(N; g; q; p; h; f; f_p; f_q)$
   Require: p, q, N and random polynomials, f and g.
2: $Inverse_Poly_f_q(N; q; f; f_q)$
3: $Inverse_Poly_f_p(N; q; f; f_q)$
4: $PolyMultiply(f_q; g; h; N; q)$
5: **for** $i = 0$ to $N - 1$ **do**
6:     **if** $h[i] < 0$ **then** $h[i] = h[i] + q$
7:         Make sure all coefficients of h are positive.
8:     **end if**
9:     $h[i] = h[i] * p \bmod q$
10: **end for**
11: KeyGen returns the Public Key h and the polynomial inverse f_p through the argument list.

4.2 Encryption

As we mentioned in Sect. 2, the basic encryption operations of $e = r*h+m(\bmod q)$ are polynomial multiplication and addition. Benefited by the special form of the prime number p, modular reduction in this step can be realized using only shifts, additions, and subtractions. To encrypt n messages, each of size N elements, the polynomial multiplication between $r(x)$ and $h(x)$ is done first, and then the plaintext messages $m_i(x), i = 1 \dots n$ are added mod q to the multiplication output.

The Algorithm 2 performs the polynomial multiplication of $r * h \bmod q$. As a reminder, the q in Step 4 is either p or q of Algorithm 1. It depends on which one is passed into the function. This algorithm only executes Step 4 if the current coefficients of $r[i]$ and $h[j]$ are both non-zero, which is in contrast with the guideline. This step approximately eliminates a third of unnecessary operations.

Algorithm 2 Polynomial multiplication in NTRUEncrypt

PolyMultiply $(N; q; r; h)$
Require: N, the coefficient modulus, q, and the two polynomials r and h.
Input: $h = h_0, \dots, h_{N-1}.r = r_0, \dots r_{N-1}$
Output: $e = e_0, \dots, e_{N-1}$
1: $e^{(0)} = 0$
2: **for** i=1 to N **do**
3: **for** all $i = 0$ to $N-1$ **do**
4: $e_i^{(j)} = e_{(i+1) \bmod N}^{(j-1)} + h_{(i+1) \bmod N} \times r_{(j-1)} \bmod q$
5: **end for**
6: **end for**
7: $e = e^{(N)} \bmod q$

4.3 Kernel Optimization

OpenCL is a strict framework that divides data into arrays and algorithms into kernel code that can manipulate this data. Therefore, the data permutation is a common operation used in various algorithms when implementing on FPGAs. Figure 3 illustrates the basic circuit-level structure for NTRU encryption. As a result, all operations of the encryption kernel are allocated dedicated hardware resources on the FPGA, prior to kernel execution. During execution, work-items step through each stage of the kernel one at a time. However, since each stage has dedicated hardware, multiple work-items may be passing through the circuit at any given moment, thus yielding pipeline parallelism. A key problem in designing this NTRU architecture is to permute streaming data. Permuting a long data sequence through hardware wiring leads to high area consumption and routing complexity. The kernel optimization techniques adopted in our implementation were introduced as follows:

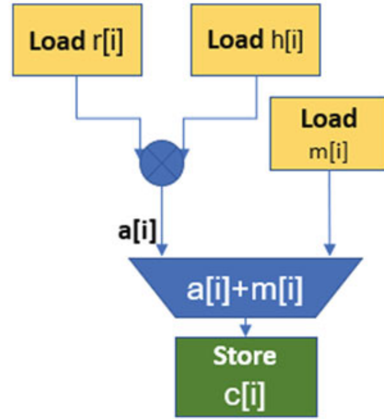


Fig. 3 Simplified kernel operations of NTRU encryption system on FPGA

(1) Multiple kernel copies. While OpenCL kernels are compiled to hardware logic circuits of fixed size, it is very common that a large portion of remaining FPGA resources are idling. To achieve the desired throughput, multiple copies of the NTRUEncrypt kernel were required. We create multiple copies of the NTRUEncrypt kernel pipelines to utilize the remaining resources of FPGA. These pipelines can execute independently from one another, and performance can scale linearly with the number of copies. Replication is handled in Altera OpenCL by setting the *num_compute_units* kernel attribute [8]. Fortunately, FPGAs are particularly efficient at integer arithmetic by allowing more than one work-group to fit within the FPGA. Targeting multiple work-groups is done using a simple kernel attribute *_attribute((num_copies(n)))*.

(2) Loop unrolling and pipelining loop. The removal of loop counter or loop testing logic is a benefit for the NTRUEncrypt algorithm on FPGAs. The nature of the NTRU encryption algorithm allows the entire code to be unrolled into a single very deep pipeline containing thousands of integer operations. The ACL compiler has #pragma directives that can be added to OpenCL code to instruct nested loops to be unrolled, allowing the full NTRUEncrypt code to be flattened. An optimally unrolled loop is a loop iteration that is launched every clock cycle. Launching one loop iteration per clock cycle maximizes pipeline efficiency and yields the best performance. As shown in the figure below, launching one loop per clock cycle allows a kernel to finish faster (Fig. 4).

(3) Using I/O channels. Another optimization on our FPGA version not currently present on GPUs is to take advantage of I/O channels and kernel channels (OpenCL 2.0 pipes) [9]. As Fig. 5 shows, kernel channels allow encryption kernels to transfer data to one another via a first-in-first-out (FIFO) buffer, without the need for host interaction. Implementation of channels decouples data movement between concurrently executing encryption kernels from the host processor. Data written to a channel remains in a channel as long as the

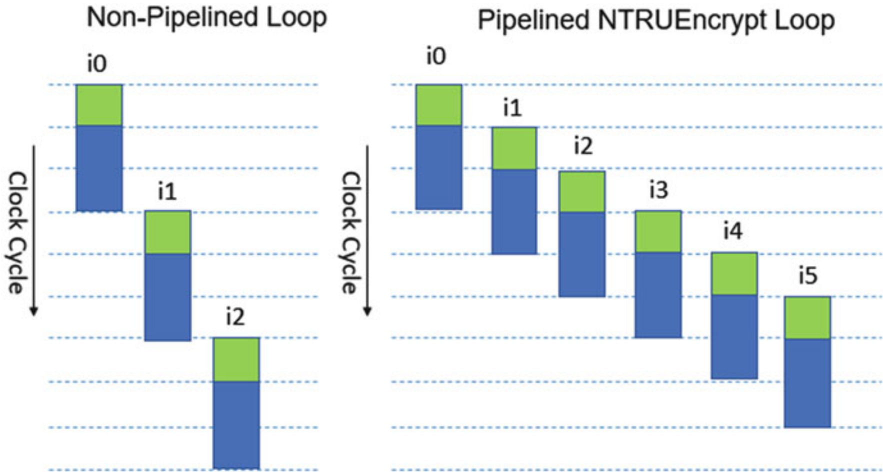


Fig. 4 The launch frequency of a loop iteration between a non-pipelined loop and a pipelined loop

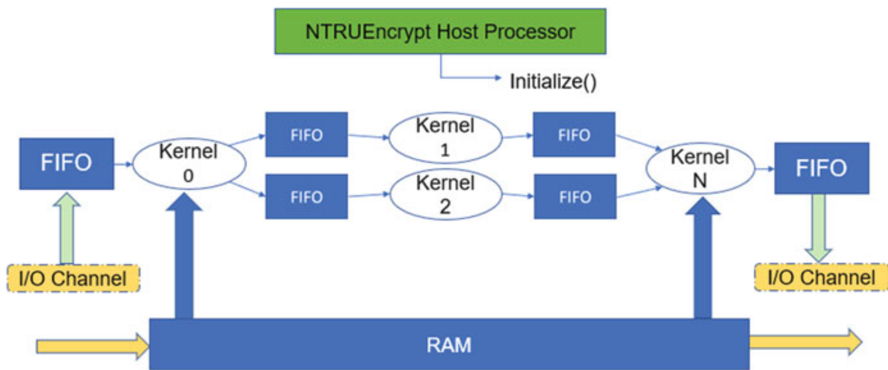


Fig. 5 An overview of channels' implementation

encryption kernel program remains loaded on the FPGA device. In other words, data written to a channel persists across multiple work-group invocations.

(4) Kernel vectorization. Whereas replication makes extra copies of the encryption kernel pipeline, kernel vectorization maintains a single pipeline where each work-item then does N times as much work. By dealing with larger units of work, kernel vectorization can even reduce the number of loads and stores. In our encryption step, all kernels' arguments are uniform, and there are no return value while each computation is dedicated to its corresponding m.

(5) Changing the memory access pattern. As we mentioned in Sect. 3, multiple memory interfaces have been configured on a single FPGA board. The NTRU-Encrypt kernel performs a large number of memory accesses. Therefore it is

important to direct which interface should be used for individual buffers, which can be done via attributes.

Our experimental broad uses SDRAM as global memory, which was configured as burst-interleaved in default. In most circumstances, the default burst-interleaved configuration leads to the best load balancing between the memory banks. However, in our case, we partition the banks manually as two non-interleaved and contiguous memory regions to achieve better load balancing [7]. Contiguous memory access optimizations analyze statically the access patterns of global load and store operations in a kernel. By basing the array index on the work-item global ID, the offline compiler can direct contiguous load operations. As shown in Fig. 6, load operations retrieve the data sequentially from the input array and send the read data to the pipeline as required. Contiguous store operations then store elements of the result that exits the computation pipeline in sequential locations within global memory.

Our another optimization of memory access efficiency is to minimize the number of global memory accesses by preloading the data from a group of computations from global memory to constant memory. Constant memory resides in global memory, but the kernel loads it into an on-chip cache shared by all work-groups at runtime. However, unlike global memory accesses that have extra hardware to tolerate long memory latencies, the constant cache suffers large performance penalties for cache misses. As the security parameter grows, the preloaded data r and h in the constant buffer cannot fit into the constant cache. We achieve better performance with *_global const* arguments instead of *_constant*. In this way, if the host application writes to constant memory that is already loaded into the constant cache, the cached data is discarded (i.e., invalidated) from the constant cache.

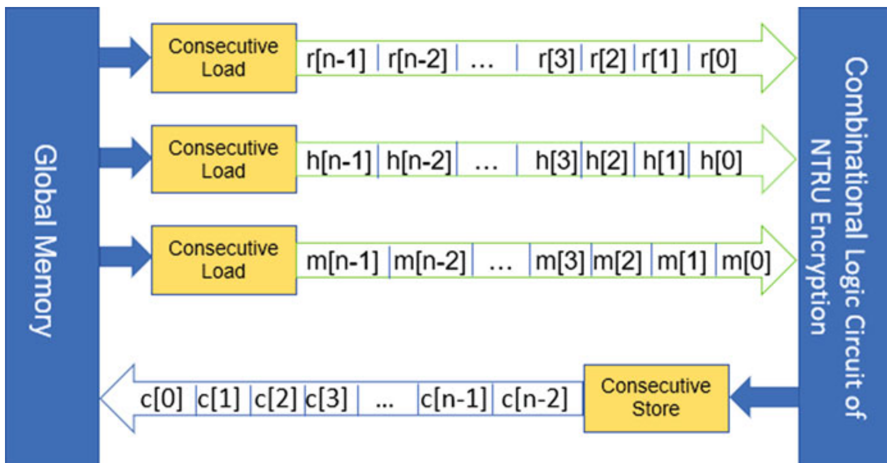


Fig. 6 Contiguous memory access

5 Experimental Results And Analysis

5.1 *Experimental Setup*

In this part, we detail comparison between FPGA and GPU implementation versions in terms of throughput and power consumption. We accelerate the running time of these cryptosystems by exploiting the inherent parallelism in computations through an FPGA-based parallelism implementation.

The NTRUEncrypt system is implemented on Intel i7-9700K CPU with OpenCL-enabled Altera Stratix V FPGA with 16 GB DDR3, 50 MHz oscillator, and 2 PCI Express hard IP blocks that support for PCIe Gen1/2/3. Intel OpenCL 2.0 development kit and Altera Quartus 18.0 are also used. The parallel implementation is based on the OpenCL framework and can run on arbitrary hardware platform accelerators with minor changes. To measure the performance improvement, the same OpenCL source code with minor change was compiled and ran on an NVidia 2080 GPU card with 1515 MHz clock rate.

5.2 *Comparisons and Analysis*

Our experiments illustrate the power consumption and FLOP throughput for the GPU-CPU and FPGA-CPU implementations (Table 1). The FPGA parallelism version performs as expected: In GPU, the throughput for the same security level is 72 MB/s. In contrast, FPGA modules speed up encryption by a factor of nearly 1.5. The average encryption throughput of FPGA is 113 MB/s. The power consumption of the FPGA accelerator is also significantly lower, requiring approximately 55 watts compared to several hundred watts on the GPU. To achieve 113 MB/s throughput for the NTRU encryption described here, only 54% of the Stratix V FPGA device was utilized. The remainder could be left unused for power savings, or extra kernels could be further optimized and placed in parallel to the encryption core.

By focusing hardware resources only on the algorithm to be executed, FPGAs can provide better performance per watt than GPUs for this work. The key difference between kernel execution on GPUs and FPGAs is how parallelism is handled. GPUs are "single-instruction, multiple-data" (SIMD) devices – groups of processing elements perform the same operation on their own individual work-items. On the

Table 1 Performance comparison of NTRUEncrypt in FPGA and GPU

	Encryption throughput	Power consumption
Stratix V FPGA	113 MB/s	55 watts
GTX 2080 GPU	72 MB/s	∼230 watts

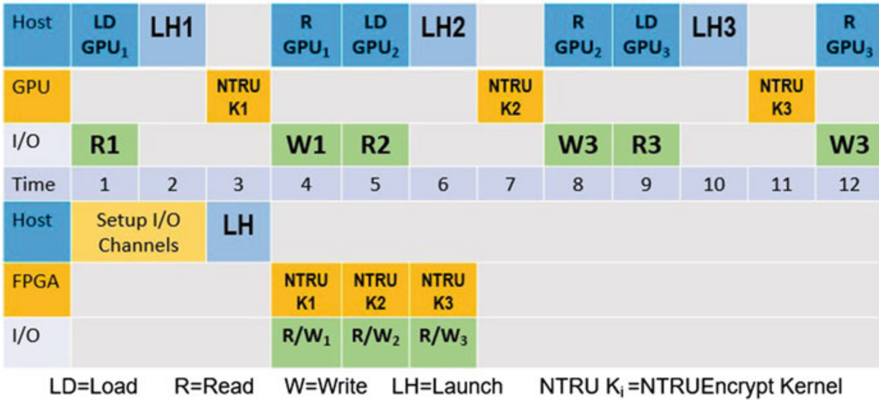


Fig. 7 FPGA vs GPU: I/O channel benefits

other hand, FPGAs exploit pipeline parallelism – different stages of the instructions are applied to different work-items concurrently.

GPUs consist of hundreds of simple processing "cores," which can each handle their own work-items. Several GPU cores execute the same instruction in lock-step with one another as a SIMD unit of fixed size (sometimes called a warp). On an FPGA, each NTRUEncrypt kernel is compiled to a custom circuit.

As we mentioned in Sect. 4, the I/O channels and kernel channels make FPGA stronger than GPU. Traditionally, GPU kernels that want to pass data to one another may do so by issuing reads and writes to global memory combined with synchronization. Performance and power efficiency gains are achieved by the removal of these intermediate reads and writes. FPGAs extend the idea of kernel channels even further to allow I/O pipes, which allow kernels to access directly from a streaming interface without host interactions, known as I/O channels. Effectively the NTRUEncrypt host configures the data pipeline and then steps out of the data path. Figure 7 illustrates an NTRU encryption kernel being executed on three sets of data coming from an I/O source. Significant time savings are achieved because the FPGA communicates directly with the I/O source and no longer needs the host to serve as a middleman.

This FPGA implementation takes advantage of computing unit replication, kernel SIMD vectorization, and non-blocking I/O channels to achieve higher throughput and lower kernel time. The SIMD vectorization duplicates only the data path of the compute unit without generating additional memory interfaces. When the kernel is vectorized, the static memory coalescing is performed automatically by the compiler to generate a memory interface that can coalesce the multiple memory loads into a single wide load.

6 Summary

The general purpose of the implementation presented herein was to provide a reference FPGA implementation of the NTRU encryption scheme and to assess its real-world properties. The operations of the NTRU encryption algorithm show good characteristics of parallel processing which makes NTRU a good candidate to benefit from the high degree of parallelism available in FPGA. FPGAs offer a middle ware among the platforms with high programmability and energy efficiency without sacrificing the throughput of the NTRUEncrypt. It is also observed that the FPGA performs better than GPU as a pipeline complexity grows. NTRUEncrypt gives incredible performance gains at no loss in security on an FPGA-based heterogeneous system. There are areas where significant improvements can be made and fine-tuned. Furthermore, the OpenCL specification can make FPGAs even more useful.

References

1. D. Micciancio, O. Regev, Lattice-based cryptography, in *Post-quantum Cryptography* (Springer, Berlin/Heidelberg, 2009), pp. 147–191
2. J. Hermans, F. Vercauteren, B. Preneel, Speed records for NTRU, in *Cryptographers' Track at the RSA Conference* (Springer, Berlin/Heidelberg, 2010)
3. T. Bai et al., Analysis and acceleration of NTRU lattice-based cryptographic system, in *15th IEEE/ACIS SNPD* (IEEE, 2014)
4. J. Hoffstein, J. Pipher, J.H. Silverman, NTRU: a ring-based public key cryptosystem, in *International Algorithmic Number Theory Symposium* (Springer, Berlin/Heidelberg, 1998)
5. J. Hoffstein, J. Silverman, Optimizations for NTRU, in *Proceedings of the Conference of Public-Key Cryptography and Computational Number Theory* (2001)
6. J. Hoffstein et al., Practical lattice-based cryptography: NTRUEncrypt and NTRUSign, in *The LLL Algorithm* (Springer, Berlin/Heidelberg, 2009), pp. 349–390
7. Intel FPGA SDK for OpenCL Pro Edition: Programming Guide. 2019.12
8. H.R. Zohouri, High performance computing with FPGAs and OpenCL. arXiv preprint arXiv:1810.09773 2018
9. Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide. 2019.9