NTRU Modular Lattice Signature Scheme on CUDA GPUs

Wei Dai[1], John Schanck[2], Berk Sunar[1], William Whyte[2], and Zhenfei Zhang[2]

[1] Worcester Polytechnic Institute, Worcester, Massachusetts, USA
{wdai,sunar}@wpi.edu
[2] Security Innovation, Wilmington, Massachusetts, USA
{jschanck,wwhyte,zzhang}@securityinnovation.com

Abstract. In this work we show how to use Graphics Processing Units (GPUs) with Compute Unified Device Architecture (CUDA) to accelerate a lattice based signature scheme, namely, the NTRU modular lattice signature (NTRU-MLS) scheme. Lattice based schemes require operations on large vectors that are perfect candidates for GPU implementations. In addition, similar to most lattice based signature schemes, NTRU-MLS provides transcript security with a rejection sampling technique. With a GPU implementation, we are able to generate many candidates simultaneously, and hence mitigate the performance slowdown from rejection sampling. Our implementation results show that for the original NTRU-MLS parameter sets, we obtain a 2× improvement in the signing speed; for the revised parameter sets, where acceptance rate of rejection sampling is down to around 1%, our implementation can be as much as 47× faster than a CPU implementation.

Keywords: NTRU, digital signatures, lattice techniques, CUDA, GPU.

1 Introduction

Lattice based signature schemes have received a growing interest due to the threat of quantum computers [1]. Organizations and research groups are looking for candidate algorithms to replace RSA and ECC based schemes [2, 3]. However, lattice based signature schemes naturally have large signature sizes compared to RSA and ECC based solutions. Among existing candidates, schemes that are instantiated over NTRU lattices [4], such as BLISS [5] and NTRU-MLS [6], are the most practical. This is due to the special structure of NTRU lattices.

Lattice based signature schemes have a history of almost 20 years. Early lattice based signature schemes, such as GGHSign [7] and NTRUSign [8], leak private key information in a transcript of message/signature pairs. An attacker can produce a signing key from a long enough transcript using methods for "learning a parallelepiped" [9, 10].

In [11], Lyubashevsky proposed a rejection sampling method to thwart transcript leakage attacks. Using his technique, signatures are produced according to a fixed public distribution (typically either a Gaussian or a uniform distribution). A transcript reveals only this public distribution, and contains no information about the particular signing key that is used to generate the signatures.

This technique has become the de facto method for avoiding transcript leakage in lattice based signature schemes, such as [5, 6, 12]. However, to ensure that the output signature follows a given distribution, a large number of random candidate signatures may need to be generated for a single document before one is accepted. This slows down the total signing procedure significantly, but this slowdown can be mitigated by parallel computing.

Graphics Processing Units (GPUs) are a widely used source of computational power for highly parallelizable tasks. Commodity GPUs provide high performance for low cost, and are an attractive platform for cryptographic accelerators. Research such as [13, 14] have demonstrated that GPUs can provide significant throughput improvements for cryptographic operations.

Our contribution: In this paper, we investigate the impact of parallel computing on NTRU-MLS signature generation time on Compute Unified Device Architecture (CUDA) supported GPUs. The NTRU Modular Lattice Signature (NTRU-MLS) scheme was proposed at PQCrypto 2014 [6], and has several attractive features. First, the use of NTRU lattices enables fast and parallel signing operations. Second,

NTRU-MLS only requires samples from the uniform distribution. The discrete Gaussian sampling required by other schemes involves the use of look-up tables or other computation that can be difficult to implement efficiently on a GPU.

Since the debut of NTRU-MLS, there has been a major revision of the recommended parameter sets [15]. The new parameter sets offer higher security, smaller keys and smaller signatures, but require much more aggressive rejection sampling. As we can generate multiple candidate signatures in parallel with a GPU, our implementation performs surprisingly well with the revised parameter sets. With the original parameter sets, we see a two-fold speedup compared to a CPU implementation; while with the revised parameter sets, our implementation is 18-47 times faster than a CPU implementation.

2 Preliminaries

2.1 Notation

NTRU-MLS makes extensive use of operations in the ring $\mathcal{R} = \mathbb{Z}[x] / (x^N - 1)$. For a polynomial $\boldsymbol{f} = \sum_{i=0}^{N-1} a_i x^i$, its vector form is $\langle a_0, a_1, \ldots, a_{N-1} \rangle$. We abuse the notation $\boldsymbol{f}$ to denote either the vector or the polynomial where there is no ambiguity. We use $\|\boldsymbol{f}\|$ to denote the max-norm of $\boldsymbol{f}$, i.e., $\|\boldsymbol{f}\| = \max_{0 \le i < N} |a_i|$. Modulo operations are centered at 0. A vector/polynomial modulo q is carried out as coefficients modulo q and lifted to $[-q/2, q/2)$. For simplicity, we use $\mathcal{R}(k)$ to denote the set $\{\boldsymbol{f} \in \mathcal{R} : \|\boldsymbol{f}\| \le k\}$.

We work on dimension $n = 2N$ integral lattices $\mathcal{L} \subset \mathbb{Z}^n$. An NTRU lattice $\mathcal{L}_{\boldsymbol{h}}$ is determined by an integer modulus q and an element $\boldsymbol{h} \in \mathcal{R}$. Points in the lattice are in one-to-one correspondence with elements of the set $\{(\boldsymbol{s}, \boldsymbol{t}) \in \mathcal{R}^2 : \boldsymbol{t} \equiv \boldsymbol{h} * \boldsymbol{s} \pmod q\}$ by equating $\mathcal{R}^2$ and $\mathbb{Z}^n$ in the natural way. For NTRU-MLS one takes $\boldsymbol{h} \equiv \boldsymbol{f}^{-1} * \boldsymbol{g} \pmod q$ where $\boldsymbol{f}$ and $\boldsymbol{g}$ are short polynomials.

2.2 Review of NTRU-MLS

Let $\mathcal{L}$ be a dimension n lattice. Let p be a small integer, and let $\boldsymbol{m}_p \in \mathbb{Z}_p^n$ represent the message to be signed. The goal of the signer in the modular lattice signature scheme is to find a vector $\boldsymbol{s}$, such that

1. $\boldsymbol{s} \in \mathcal{L}$;
2. $\|\boldsymbol{s}\|$ is small; and
3. $\boldsymbol{s} \equiv \boldsymbol{m}_p \mod p$.

Any lattice point $\boldsymbol{s}$ satisfying $\boldsymbol{s} \equiv \boldsymbol{m}_p \pmod p$ is a potential signature. With a short basis, it is easy to find such a vector $\boldsymbol{s}$ that satisfies all requirements. Anyone can use a public basis to verify the correctness of the signature. However, if one does not know a short basis, then it is hard to find a vector that satisfies all of the criteria simultaneously.

To enable transcript security, it is required that the signatures are distributed over a public region that is sufficiently large. A candidate will be rejected if it falls out of this region.

The NTRU-MLS scheme is an efficient instantiation of this idea using NTRU lattices. The scheme takes these parameters:

N	the degree of the polynomial ring
p	a small prime
q	an integer larger than and relatively prime to p
B_s, B_t	norm constraints of a signature $(\boldsymbol{s}, \boldsymbol{t})$.

The scheme follows these algorithms:

- Key generation:
  1. Pick two polynomials $\boldsymbol{F}, \boldsymbol{g} \in \mathbb{Z}_p^N$ that are both invertible modulo p and q; set $\boldsymbol{f} = p\boldsymbol{F}$.
  2. Compute $\boldsymbol{h} \equiv \boldsymbol{f}^{-1} * \boldsymbol{g} \pmod q$.
  3. Publish $\boldsymbol{h}$ as the public key; keep $(\boldsymbol{f}, \boldsymbol{g})$ as the secret key.

– Sign a document $\mu \in \{0,1\}^*$ with the secret key $(\boldsymbol{f}, \boldsymbol{g})$:
  1. $(\boldsymbol{s}_p, \boldsymbol{t}_p) \leftarrow \mathsf{Hash}(\boldsymbol{h}, \mu)$.
  2. $\boldsymbol{r} \leftarrow \mathcal{R}\left(\left\lfloor \frac{q}{2p} + \frac{1}{2} \right\rfloor\right)$.
  3. $\boldsymbol{s}_0 \leftarrow \boldsymbol{s}_p + p\boldsymbol{r}$; $\boldsymbol{t}_0 = \boldsymbol{s}_0 * \boldsymbol{h} \pmod q$.
  4. $\boldsymbol{a} \leftarrow (\boldsymbol{t}_p - \boldsymbol{t}_0) * \boldsymbol{g}^{-1} \pmod p$.
  5. $(\boldsymbol{s}, \boldsymbol{t}) \leftarrow (\boldsymbol{s}_0, \boldsymbol{t}_0) + \boldsymbol{a} * (\boldsymbol{f}, \boldsymbol{g})$
  6. If $\|\boldsymbol{s}\| > \frac{q}{2} - B_s$ or $\|\boldsymbol{t}\| > \frac{q}{2} - B_t$, goto Step 2.
  7. Output $\boldsymbol{s}$ as a signature of μ.
– Verify a signature $\boldsymbol{s}$ on document μ with respect to the public key $\boldsymbol{h}$:
  1. Compute $\boldsymbol{t} \equiv \boldsymbol{s} * \boldsymbol{h} \pmod q$
  2. $(\boldsymbol{s}_p, \boldsymbol{t}_p) \leftarrow \mathsf{Hash}(\boldsymbol{h}, \mu)$.
  3. If $(\boldsymbol{s}, \boldsymbol{t}) \not\equiv (\boldsymbol{s}_p, \boldsymbol{t}_p) \pmod p$, return false.
  4. If $\|\boldsymbol{s}\| > \frac{q}{2} - B_s$ or $\|\boldsymbol{t}\| > \frac{q}{2} - B_t$, return false.
  5. Return true.

Product Form Polynomials To enable fast ring convolutions, NTRU-MLS makes use of product form polynomials. The secret polynomial $\boldsymbol{F}$ is constructed as $\boldsymbol{F} = \boldsymbol{F}_1 * \boldsymbol{F}_2 + \boldsymbol{F}_3 + 1$, where $\boldsymbol{F}_i$ is a very sparse ternary polynomial with exactly d_i coefficients equal to $+1$ and d_i coefficients equal to -1, for $i = 1, 2, 3$ respectively. The same construction method applies to $\boldsymbol{g}$. See section 3.2 for more details.

Rejection sampling Rejection sampling is a technique to obtain a desired distribution χ_2 from an input distribution χ_1. In NTRU-MLS, the transcript security requires that a signature $(\boldsymbol{s}, \boldsymbol{t})$ is indistinguishable from a sample from a uniform distribution. However, as $(\boldsymbol{s}, \boldsymbol{t}) = (\boldsymbol{s}_0, \boldsymbol{t}_0) + \boldsymbol{a} * (\boldsymbol{f}, \boldsymbol{g})$, where $(\boldsymbol{s}_0, \boldsymbol{t}_0)$ is uniformly distributed[3] over $\mathcal{R}(q/2) \times \mathcal{R}(q/2)$, the raw output $(\boldsymbol{s}, \boldsymbol{t})$ does not follow a uniform distribution. Indeed, the distribution χ_1 of $(\boldsymbol{s}, \boldsymbol{t})$ is slightly thicker towards the edge of $\mathcal{R}(q/2) \times \mathcal{R}(q/2)$ due to the $\boldsymbol{a} * (\boldsymbol{f}, \boldsymbol{g})$ term. By rejecting the candidate signature that lies outside of $\mathcal{R}(q/2 - B_s) \times \mathcal{R}(q/2 - B_t)$ we obtain a uniform distribution χ_2 over $\mathcal{R}(q/2 - B_s) \times \mathcal{R}(q/2 - B_t)$.

As mentioned earlier, rejection sampling incurs a cost. Depending on the rejection rate, one need to regenerate candidate signatures (Steps 2-6) for many time. We use parallel computing to mitigate the slowdown due to rejection sampling.

2.3 CUDA GPU Computing

CUDA is a parallel computing platform and application programming interface (API) model that allows the use of a CUDA-enabled GPU for general purpose processing. A GPU provides immense computational power, but has a highly specialized architecture. The programming model and memory hierarchy are much different from a CPU. Here we present a succinct review.

Programming Model A CUDA-enabled GPU consists of thousands of CUDA cores partitioned into an array of independent *streaming multiprocessors (MPs)*. Each MP is built with function units, private memory, data cache and instruction buffer. The CUDA programming model abstracts GPUs' hardware architecture for developers.

CUDA GPU computing involves two parties: CPUs and system memory are called "*host*"; GPUs and GPU memory are called "*device*". In a GPU-accelerated application, intensive computation are offloaded to the device, while the rest of the application remains on the host, with a parallel function invocation (a *kernel*). A kernel is supposed to be invoked by the host and executed by the device with an array of sequential

[3] To achieve so, the random vector $\boldsymbol{r}$ needs to be uniformly sampled from $\mathcal{R}\left(\left\lfloor \frac{q}{2p} + \frac{1}{2} \right\rfloor\right)$. If the random number generator is biased, one loses the transcript security. In our implementation, we use Salsa20 [16] which has been standardized by IETF [17], and has been adopted in several popular libraries, such as OpenSSH and IPsec.

threads. A MP is able to concurrently execute groups of 32 threads (*warps*) in a single instruction multiple data (SIMD) style. All threads, each having an thread index provide its own control flow, run the same code.

By the concept of CUDA, threads are further grouped into *blocks*. Every kernel is launched with *grid* configurations (the number of blocks and the number of threads per block). Each block is then assigned to one of the MPs. Only threads within a block can cooperate or synchronize. The dimension of a grid and that of each block determine how computing resource is assigned to program, i.e. the mapping of threads to CUDA cores and of blocks to MPs, which is mainly arranged by hardware controller components.

A basic model of computation offloading includes 3 steps: copy input data from the host to the device; launch a kernel (or kernels) with grid configurations; copy results from the device to the host when necessary. The computation complexity of a kernel and the amount of data transferred between host and device depend on the details of an implementation. A few criteria need to be considered to design an efficient GPU implementation, such as maximizing the use of processors and MPs (see 4.1), minimizing the latency caused by data accesses (see 2.3), etc..

Memory Management Memory management influences the performance of a GPU-accelerated program significantly. Several types of memory or cache are built within a GPU and each has an optimal access pattern. The GPU memory architecture is represented in Table 1 where memory types are listed from top to bottom by access speed from fast to slow.

Table 1. GPU memory organization

Memory	**Cached**	**Access**	**Scope**	**Lifetime**
Register	N/A	R/W	One thread	Thread
Constant	Yes	R	All thread + host	Application
Texture	Yes	R	All thread + host	Application
Shared	N/A	R/W	All threads in a block	Block
Local	No	R/W	One thread	Thread
Global	No	R/W	All thread + host	Application

An efficient kernel design should take advantage of these device memory properties:

- Global memory is off-chip, uncached, expensive to access. It is not a good option for data accessed repeatedly. However, it is adequate in size (e.g. 12 GB on a NVIDA GeForce Titan X graphic card), and it is the only memory type that can be modified by a kernel and be copied to the host. Therefore, it is used for data input and output. Global memory favors a coalesced access pattern: each thread in a warp accesses memory contiguously from the same 128 Bytes chunk. A non-coalesced (strided) access pattern could take hundreds of times more cycles.
- Constant memory is cached and has high speed with a limited size (usually 64 KB). It suits repeatedly requested data and outspeeds any other memory types if adjacent threads tend to access the same data.
- Texture memory is cached and read-only. It is bounded to global memory by the host. It works for this pattern: nearby thread are likely to read from data that have close addresses (non-coalesced). Texture memory is preferred for data that are read often, updated rarely, especially for data whose access pattern exhibits a spatial locality.
- Shared memory resides in each MP. It is allocated for and can be read or written by all threads in a block. It offers better performance than local or global memory, thus is often utilized as a buffer to hold intermediate data, or to re-order strided global memory accesses to a coalesced pattern. However, shared memory comes with a limited size per block (e.g. 96 KB on a NVIDA GeForce Titan X graphic card). Shared memory is divided into banks that each bank serves only one address per cycle. Multiple simultaneous accesses to a bank create a bank conflict. Shared memory can become as fast as registers, if all threads of a half-warp access a different bank (no bank conflict).

The data transfer between the host and the device, invoked to feed constant memory or global memory and to output results, may cause unwanted delay as well. They can be optimized in following methods: overlap data transfer and computation with CUDA streams; minimize the amount of data transferred and pack small transfers into one larger transfer when possible; use page-locked (or pinned) memory to achieve a higher bandwidth.

3 Our GPU Implementation

The idea is to offload the steps 2-7 of the signing algorithm in Section 2.2 to a GPU. Those steps (we may call them an "attempt") repeat until a valid signature is generated. Hence, the amount of computation grows when the probability of generating a valid signature, namely Prob[valid], decreases. For example, when choosing the parameter set $\left[N = 401, q = 2^{15}, p = 3, B_s = 138, B_t = 46\right]$, Prob[valid] is as low as 1.11%. On average it would take approximately 90 attempts to generate a valid signature. Since every attempt is independent, we may compute a number of them in parallel, which leads to the idea of exploiting a GPU.

When designing a GPU kernel that computes attempts, we considered several factors to achieve a higher performance, such as data types, device memory types, grid dimensions, etc.. We started with implementing polynomial operations and a random number generator based on the Salsa20 stream cipher. With the help of a profiling tool: the NVIDIA Visual Profiler, we improved kernels' performance, optimized the communication between the host and the device, and set optimal grid dimensions according to the device architecture and parameter selection.

3.1 The CPU-GPU Workflow

We designed a single CUDA kernel `GenSig` to compute Steps 2-7. As opposed to a design with several kernels, temporary results may utilize shared memory that has a faster accessing speed rather than global memory. We also have every CUDA block perform an independent attempt, due to the fact that shared memory is only accessible by threads within the same block. Let say that the host launches a kernel `GenSig` that creates β attempts with β blocks (the grid dimension/size) and τ threads (the block dimension/size) per block. The probability that a block generates a valid signature depends on the parameter set and is denoted by $\rho = \text{Prob[valid]}$.

The Host At the start, the host copies necessary input data to the device: the public key $\boldsymbol{h}$, the secret key $(\boldsymbol{f}, \boldsymbol{g})$ in product form and the precomputed $\boldsymbol{g}^{-1}$, the hash $(\boldsymbol{s}_p, \boldsymbol{t}_p)$ computed by the host, a 256-bit key, a 64-bit nonce and a 64-bit stream position required by the Salsa20 stream cipher.

The host also allocates memory space on both the device and the host to hold and to transfer output. Since `GenSig` generates β attempts, an array `Sig` large enough to hold β signatures and a β-bit flag `Pos` are allocated. A block that generates a valid signature marks the corresponding bit in `Pos` with atomic operations. After every kernel launch, the host copies `Pos` from the device. If the host then sees the k-th bit of `Pos` is marked, it copies the k-th signature in `Sig` from the device and finishes the signature generation.

When input is ready on the device, the host launches kernel `GenSig`. We may obtain a valid signature after $(\rho\beta)^{-1}$ kernel launches on average. The program needs to stop computing extra attempts as soon as a valid signature is generated. Such a termination instruction is issued by the host. The block(s) that generates a valid signature cannot terminate or inform the other blocks efficiently. Although an inter-block synchronization could be instantiated with repeated atomic accesses to global memory to perform a broadcast among blocks, it would introduce a significant overhead.

The Device The kernel `GenSig` assigns several CUDA blocks that each attempts to generate a signature. We set a flag `IsValid` with shared memory in each block to mark whether the signature generated by this block is valid or not. In our design, the validity of a signature can be diagnosed before the attempt finishes. In Fig. 1 we can see that three criteria have to be checked: the random number generation flag must not be

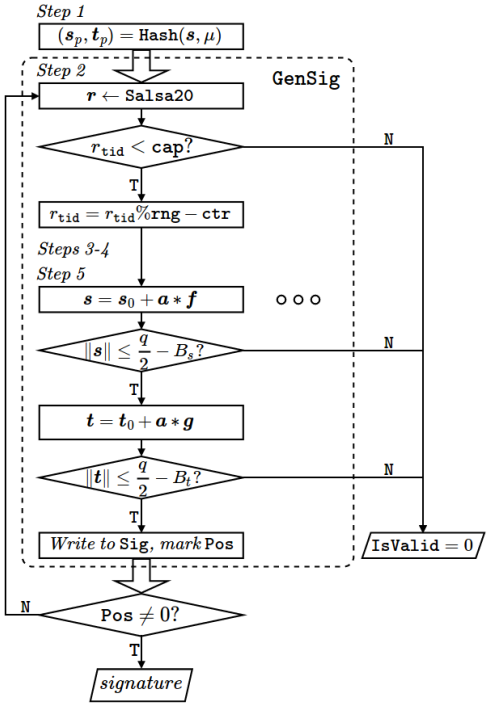

Fig. 1. Flowchart of The Signing Algorithm on GPU

set (see 3.3), the norm of $\boldsymbol{s}$ must be less than or equal to $q/2 - B_s$, and the norm of $\boldsymbol{t}$ must be less than or equal to $q/2 - B_t$ (Step 6 of the signing procedure).

A thread that has observed a failure among these criteria marks `IsValid` as false with an atomic operation and terminates itself right away. Each thread checks `IsValid` before consuming. However, a thread might mark `IsValid` as false under a criterion, while another thread have checked the same criterion and have started to compute the following steps. We use a final synchronization of all threads to ensure that a signature is stored to `Sig` and is marked in `Pos` only if it is valid. Such a strategy avoids unnecessary accesses to global memory. To be clear, one kernel could have generated more than one valid signatures and stored them separately. The host only retrieves one of them. The routines of each thread in `GenSig` are summarized in Fig. 1.

Summary The signing procedure works in the following way:

- Generate a hash of the document to be signed: $(\boldsymbol{s}_p, \boldsymbol{t}_p)$.
- Copy $\left(\boldsymbol{s}_p, \boldsymbol{t}_p, \boldsymbol{h}, \boldsymbol{g}^{-1}, \boldsymbol{F}_1, \boldsymbol{F}_2, \boldsymbol{F}_3, \boldsymbol{G}_1, \boldsymbol{G}_2, \boldsymbol{G}_3\right)$ to device memory.
- Repeatedly invoke the `GenSig` kernel until at least one valid signature is generated.
- Copy one valid signature to the host from the device.

Data type choices, device memory management, random number generation, polynomial operations and the selection of (β, τ) will be discussed later in this section.

3.2 Ring Operations

Here we explain how the algebraic operations are implemented. Steps 3, 4, and 5 in the signing algorithm require polynomial multiplication. Steps 3 and 4 require multiplication of arbitrary elements of $\mathcal{R}$ whereas

the multiplications in Step 5 involve product-form polynomials. In Section 3.2 we discuss our generic multiplication routine, and in Section 3.2 we discuss how we handle product-form polynomials. We also discuss how these routines determine the optimal grid dimensions (the number of blocks launched and the number of threads per block) for the CUDA kernel.

Data Types We carefully select these data types to avoid overflow and to reduce memory consumption as well:

- A polynomial in $\mathcal{R}(q/2)$, such as $\boldsymbol{h}$, is stored in an array of 32-bit signed integers;
- A polynomial in $\mathcal{R}(p/2)$, such as $\boldsymbol{s}_p, \boldsymbol{t}_p, \boldsymbol{g}^{-1}$, is stored in an array of 8-bit signed integers;
- Ternary polynomials $\boldsymbol{F}_i$ and $\boldsymbol{G}_i$ are stored in arrays of length $2d_i$. The entries of these arrays are 16-bit unsigned integers that represent the indices of $+1$ and -1 coefficients.

Generic Polynomial Multiplication Given two polynomials $\boldsymbol{a} = \sum_{i=0}^{N-1} a_i x^i$ and $\boldsymbol{b} = \sum_{i=0}^{N-1} b_i x^i$, and their product $\boldsymbol{c} = \sum_{i=0}^{N-1} c_i x^i$, where $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathcal{R}$, we have

$$\begin{aligned}\boldsymbol{c} &= \boldsymbol{a} * \boldsymbol{b} \pmod{x^N - 1} \\ &= \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a_i b_j x^{(i+j \bmod N)} \\ &= \sum_{i=0}^{N-1} \left(\sum_{j=0}^{N-1} a_j b_{(i-j \bmod N)} \right) x^i. \end{aligned} \tag{1}$$

We will hereafter drop the $(\bmod N)$ from superscripts and subscripts and will perform index arithmetic in $\mathbb{Z}_N$. The computation of $\boldsymbol{c}$ involves a total of N^2 integer multiplications and N^2 additions. As we want the computation to be evenly distributed among the threads, we use N threads and have each thread compute one coefficient of $\boldsymbol{c}$.

The computation is performed either modulo q (step 3) or modulo p (step 4). According to our parameter selections (see Table 4), q being a power of 2 provides fast modulo q operations. We also choose $p = 3$ which makes a polynomial in $\mathcal{R}(p/2)$ a ternary polynomial.

Product Form Polynomial Multiplication In step 5, $\boldsymbol{f}$ and $\boldsymbol{g}$ are computed from ternary polynomials $\boldsymbol{F}_i$ and $\boldsymbol{G}_i$:

$$\begin{aligned}\boldsymbol{f} &= p(\boldsymbol{F}_1 * \boldsymbol{F}_2 + \boldsymbol{F}_3 + 1) \\ \boldsymbol{g} &= \boldsymbol{G}_1 * \boldsymbol{G}_2 + \boldsymbol{G}_3 + 1.\end{aligned}$$

$\boldsymbol{F}_i$ and $\boldsymbol{G}_i$ have exactly d_i coefficients that are $+1$ and exactly d_i coefficients that are -1. In other words, $\boldsymbol{F}_i, \boldsymbol{G}_i \in \mathcal{T}(d_i)$ $(i = 1, 2, 3)$, where $\mathcal{T}(d_i) \subset \mathcal{R}(1)$ is defined as

$$\mathcal{T}(d_i) = \left\{ \boldsymbol{b} \subset \mathcal{R}(1) : \begin{array}{l} |\{j : b_j = +1\}| = d_i, \\ |\{j : b_j = -1\}| = d_i, \end{array} \right\}.$$

To represent elements of $\mathcal{T}(\cdot)$ we only have to record the positions of non-zero coefficients. We store a polynomial $\boldsymbol{b} \in \mathcal{T}(d)$ as $(\mathcal{B}^+, \mathcal{B}^-, d)$, where $\mathcal{B}^+ = \{j \in \mathbb{Z}_N : b_j = +1\}$, $\mathcal{B}^- = \{j \in \mathbb{Z}_N : b_j = -1\}$ and $|\mathcal{B}^+| = |\mathcal{B}^-| = d$.

Now we can express Step 5 of the signing routine as several multiplications involving polynomials in $\mathcal{T}(d)$. Given the polynomial $\boldsymbol{a}$ from Step 4, we compute

$$\begin{aligned}\boldsymbol{a} * \boldsymbol{f} &= p(\boldsymbol{a} * \boldsymbol{F}_1 * \boldsymbol{F}_2 + \boldsymbol{a} * \boldsymbol{F}_3) \\ \boldsymbol{a} * \boldsymbol{g} &= \boldsymbol{a} * \boldsymbol{G}_1 * \boldsymbol{G}_2 + \boldsymbol{a} * \boldsymbol{G}_3.\end{aligned}$$

Algorithm 1 Product Form Polynomial Multiplication with N Threads in A Block

Input: $\boldsymbol{a}$, $\boldsymbol{b}_i = \left(\mathcal{B}_i^+, \mathcal{B}_i^-, d_i\right)$, $i = 1, 2, 3$
Input: $\mathsf{tid} \in \mathbb{Z}_N$ ▷ thread ID that marks the current thread
Input: $\boldsymbol{t}$ ▷ temporary result shared by all threads

parfor all threads **do**
  procedure $\boldsymbol{t} = \boldsymbol{a} * \boldsymbol{b}_1$
    $t_{\mathsf{tid}} = 0$
    for $j \leftarrow 0$ **to** $d_1 - 1$ **do**
      $t_{\mathsf{tid}} += a_{\left(\mathsf{tid} - \mathcal{B}_1^+[j] \bmod N\right)}$
      $t_{\mathsf{tid}} -= a_{\left(\mathsf{tid} - \mathcal{B}_1^-[j] \bmod N\right)}$
    end for
  end procedure
  __syncthreads()__ ▷ Wait until $\boldsymbol{t}$ is ready
  procedure $\boldsymbol{c} = \boldsymbol{t} * \boldsymbol{b}_2$
    $c_{\mathsf{tid}} = 0$
    for $j \leftarrow 0$ **to** $d_2 - 1$ **do**
      $c_{\mathsf{tid}} += t_{\left(\mathsf{tid} - \mathcal{B}_2^+[j] \bmod N\right)}$
      $c_{\mathsf{tid}} -= t_{\left(\mathsf{tid} - \mathcal{B}_2^-[j] \bmod N\right)}$
    end for
  end procedure
  procedure $\boldsymbol{c} += \boldsymbol{a} * \boldsymbol{b}_3$
    for $j \leftarrow 0$ **to** $d_3 - 1$ **do**
      $c_{\mathsf{tid}} += a_{\left(\mathsf{tid} - \mathcal{B}_3^+[j] \bmod N\right)}$
      $c_{\mathsf{tid}} -= a_{\left(\mathsf{tid} - \mathcal{B}_3^-[j] \bmod N\right)}$
    end for
  end procedure
end parfor
Output: $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}_1 * \boldsymbol{b}_2 + \boldsymbol{a} * \boldsymbol{b}_3$

To simplify, the problem is to compute $\boldsymbol{a} * \boldsymbol{F}_1$, $(\boldsymbol{a} * \boldsymbol{F_1}) * \boldsymbol{F_2}$, $\boldsymbol{a} * \boldsymbol{F_3}$, and likewise for $\boldsymbol{g}$. Each of these multiplications is between one dense and one sparse polynomial in $\mathcal{T}(d)$ for some d. Given a dense polynomial $\boldsymbol{a}$ and a sparse polynomial $\boldsymbol{b} \in \mathcal{T}(d)$ represented as $(\mathcal{B}^+, \mathcal{B}^-, d)$, their product $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$ can be computed as

$$
\begin{aligned}
\boldsymbol{c} &= \boldsymbol{a} * \boldsymbol{b} \pmod{x^N - 1} \\
&= \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a_i b_j x^{i+j} \\
&= \sum_{i=0}^{N-1} \left(\sum_{j \in \mathcal{B}^+} a_i x^{i+j} - \sum_{j \in \mathcal{B}^-} a_i x^{i+j} \right) \\
&= \sum_{i=0}^{N-1} \left(\sum_{j \in \mathcal{B}^+} a_{i-j} x^{i} - \sum_{j \in \mathcal{B}^-} a_{i-j} x^{i} \right) \\
&= \sum_{i=0}^{N-1} \left(\sum_{j \in \mathcal{B}^+} a_{i-j} - \sum_{j \in \mathcal{B}^-} a_{i-j} \right) x^i. \qquad (2)
\end{aligned}
$$

Now when implemented, (2) requires only a total of Nd integer additions and subtractions which offers a much better performance than (1), since d_i is selected much smaller than N. Furthermore, (2) is completely parallelizable. As we want the computation to be evenly distributed among threads, we use N threads, one for each coefficient of $\boldsymbol{c}$.

Algorithm 2 Random Polynomial Generation with N Threads in A Block

Input: $\mathsf{ctr}, \mathsf{rng}, \mathsf{cap}$
Input: $\mathsf{tid} \in \mathbb{Z}_N$ ▷ thread ID that marks the current thread
Input: $\boldsymbol{t}$ ▷ a buffer to hold $\lceil N/16 \rceil$ integers (32-bit)

parfor $\lceil N/16 \rceil$ threads **do**
  procedure THE SALSA20 BLOCK CIPHER
    Write output to $t_{16\mathsf{tid}} \sim t_{16\mathsf{tid}+15}$
  end procedure
end parfor
parfor N threads **do**
  procedure SCALE t_{tid} TO r_{tid}
    if $t_{\mathsf{tid}} < \mathsf{cap}$ **then**
      $r_{\mathsf{tid}} = (t_{\mathsf{tid}} \bmod \mathsf{rng}) - \mathsf{ctr}$
    else
      Mark this signature as invalid
    end if
  end procedure
end parfor
Output: $\boldsymbol{r} \in \mathcal{R}\left(\left\lfloor \frac{q}{2p} + \frac{1}{2} \right\rfloor\right)$

Remark: We considered (2) as an alternative method to perform generic polynomial multiplications. However, when implemented in CUDA program, (2) introduces a large number of conditional branches that make it slower than (1).

Algorithm 1 explains how product form multiplication is implemented in parallel. All threads execute the same code; tid marks the thread ID and indexes the coefficient of the result that the thread computes. We create a buffer $\boldsymbol{t}$ shared by all threads to hold a temporary polynomial result. In the procedure $\boldsymbol{c} = \boldsymbol{t} * \boldsymbol{b}_2$, a thread reads from a location in $\boldsymbol{t}$ that is previously modified by another thread. To avoid a write-after-read hazard that would lead to an incorrect result, we call a barrier function (*__syncthreads()__*) that synchronizes all threads after the writes to $\boldsymbol{t}$.

We need to determine the types of device memory to use and block dimensions. In terms of performance, $\boldsymbol{t}$ requires fast read and write by all threads, and therefore utilizes shared memory. For $\mathcal{B}_i^+$ and $\mathcal{B}_i^-$, constant memory is obviously the right type to use, since all threads read the same locations of $\mathcal{B}_i^+$ and $\mathcal{B}_i^-$ at the same time. All threads should belong to the same thread block to synchronize. Hence, it takes at least N threads on each block to handle a polynomial multiplication. $\boldsymbol{a}$ has a random access pattern by the algorithm, hence adopts shared memory.

Summary Polynomial multiplications requires at least N threads to execute. And all these threads belong to the same block. The usage of device memory types are listed in Table 2.

3.3 Random Number Generator

Step 2 of the signing algorithm requires a fast random number generator on the device. We adopted the Salsa20 stream cipher [16] which expands a 256-bit key, a 64-bit nonce that increases by every attempt, and a 64-bit stream position to a 512-bit output. The random polynomial $\boldsymbol{r}$ is uniformly sampled from $\mathcal{R}\left(\left\lfloor \frac{q}{2p} + \frac{1}{2} \right\rfloor\right)$, i.e. the coefficients of $\boldsymbol{r}$ are uniformly random over the interval

$$\left[-\left\lfloor \frac{q+p-1}{2p} \right\rfloor + 1, \ \left\lfloor \frac{q+p-1}{2p} \right\rfloor - 1 \right]. \tag{3}$$

Let

$$\begin{aligned}\mathsf{ctr} &= \left\lfloor \tfrac{q+p-1}{2p} \right\rfloor - 1 \\ \mathsf{rng} &= 2 * \mathsf{ctr} + 1 \\ \mathsf{cap} &= 2^{32} - 1 - \left(2^{32} - 1 \bmod \mathsf{rng}\right).\end{aligned}$$

To achieve an uniform distribution, we generate uniform integers in $\mathbb{Z}_{\mathsf{cap}}$ and map them into interval (3) via:

$$r \mapsto (r \bmod \mathsf{rng}) - \mathsf{ctr}.$$

Taking 32 bits of the output of Salsa20 yields a random integer $r \in \mathbb{Z}_{2^{32}}$. However, applying the above map to elements of $\mathbb{Z}_{2^{32}}$ does not yield the uniform distribution on the interval (3). To fix this, we mark the signature invalid if it ever happens that $r_{\mathsf{tid}} \geq \mathsf{cap}$. This reduces the probability of generating a valid signature by a factor of $\left(1 - \left(1 - \frac{\mathsf{cap}}{2^{32}}\right)^N\right)$. Since $\mathsf{rng} << 2^{32}$, the loss in acceptance rate is tolerable.

Details are provided in Algorithm 2. Every group of 16 threads generates a 512-bit random seed and each thread generates a single 32-bit element of interval (3) from a unique portion seed. The Salsa20 rounds cannot be easily parallelized on 16 threads. However, the performance of the whole kernel is not much affected if we compute all the rounds on each thread with registers.

3.4 Memory Management

Table 2 lists all the inputs and buffers needed by the CUDA kernel. As explained in algorithm 1, all threads access to the same location in $\boldsymbol{F}_i$ and $\boldsymbol{G}_i$ at the same time, where constant memory is chosen. However, the other inputs, although are read only, have different access patterns that are more efficient with shared memory. To minimize the latency cause by memory accesses, we first copy those data from the host to device texture memory, and then load them into shared memory. Together with other buffers that require read and write accesses, the total amount of shared memory is $(10N + 68)$ bytes, which is less than the limit of our GPU (96 KB per block).

Table 2. Elements That Use GPU Memory

Input / Buffer	Memory Type	Integer Type	Size in Bytes
$\boldsymbol{F}_i$, $\boldsymbol{G}_i$	constant	16-bit unsigned	$4(d_1 + d_2 + d_3)$
$\boldsymbol{s}_p$, $\boldsymbol{t}_p$, $\boldsymbol{g}^{-1}$	shared, texture	8-bit signed	N
$\boldsymbol{h}$	shared, texture	32-bit signed	$4N$
Salsa20	shared, texture	32-bit unsigned	64
$\boldsymbol{a}$, $(\boldsymbol{t}_p - \boldsymbol{t}_0)$	shared	8-bit signed	N
$\boldsymbol{s}_0$, $\boldsymbol{t}_0$	shared	32-bit signed	$4N$
`IsValid`	shared	32-bit unsigned	4

The kernel avoids access to global memory as much as possible, since it is expensive. When at least one valid signature is generated, the kernel will write the signature to a global memory location from which it will eventually be read into system memory. Each valid signature ($4N$ bytes) is written to a specified position in the result buffer. If more than one valid signature is generated, more global memory accesses are required. Although these global memory accesses are unnecessary, the only way to avoid them is with global memory reads to check whether a valid signature has been generated. In practice, this solution yields no performance improvement.

4 Implementation Results

In this section we discuss performance influential factors such as CUDA grid dimensions, the GPU architecture and the parameter selection. We then present implementation results showcasing the performance of our NTRU-MLS on GPU, and compare them against the previous C instantiation proposed by Hoffstein et al. in [6].

4.1 Tuning CUDA Implementation

Parameter sets No. 1-4 in Table 4 were described in in [6]. We have listed them with their reduced bit security estimates due to the attack described in [18]. The revised parameter sets No.5-9 provide a higher level of security and a reduced signature (and key) size. The improvements come at a cost of a more aggressive rejection rate (i.e. a lower Prob[valid]) which implies a huge increase of signing time. It is clearly observed when we compare signing speeds in Table 4 of revised parameters to those of original parameters.

As in 3.1, let $\rho = \text{Prob[valid]}$, and let β be the grid size launched by the kernel `GenSig`. In average, after $(\rho\beta)^{-1}$ kernel launches, a valid signature is generated. Unfortunately, those kernels are launched sequentially because of the validity check on the host after every launch. The choice of β affects performance greatly.

The NVIDIA GTX Titan X graphics card based on the GM200 GPU (see Table 3) has 24 multiprocessors (MPs) that are capable of computing multiple blocks. If we select $\beta \leq 24$, at least one MP is idle during computation. That is to say, choosing $\beta = 24$ does not increase the latency of the kernel `GenSig` much, comparing to choosing a smaller grid size. Although for parameters No.1-4, No.6 and No.7 whose acceptance rates are higher than $1/24$, setting $b = 24$ might generate more than one valid signatures, which is unnecessary and introduces extra accesses to device global memory when writing back signatures. However, the affect is negligible, as is observed in tests.

Table 3. NVIDIA GeForce Titan X (GM200) Technical Specifications

Item	Spec	Item	Spec
CUDA Cores	$3,072$	Multiprocessors (MPs)	24
Device Memory	12 GB	CUDA Cores Per MP	128
GPU Clock	1.2 GHz	Blocks Per MP	≤ 16
Memory Clock	3.5 GHz	Warps Per MP	≤ 64
Warp Size	32	Threads Per MP	$\leq 2,048$
Threads Per Block	$\leq 1,024$	Registers Per MP	$65,536$
Wraps Per Block	≤ 32	Shared Memory Per MP	≤ 96 KB
Registers Per Thread	≤ 63	Computing Capability	5.2

For other parameters with high rejection rates, we set a larger grid size, usually $\beta = 48$. The NVIDIA Visual Profiler shows that setting $\beta = 48$ achieves a full utilization of MPs (all MPs running similar amounts of computation task), based on the amount of registers, the size of shared memory and the block dimension. In average, a valid signature is generated every two kernels, which makes $\beta = 48$ an optimal choice.

The grid size choice is highly determined by the GPU architectures, of which the number of MPs and profiler results can be different.

4.2 Performance

The benchmarking results of our implementation are shown in the column "This Work" of Table 4. These results are obtained by running the implementations on a machine with an NVIDIA GTX Titan X graphics card (see 3) and a 3.5GHz Intel Core i7-3770K processor in single-thread mode. We reproduce the results of the C implementation by Hoffstein et al. [6] on the same machine. The results are shown in the column "[6]" and the speedup of CUDA implementation over C implementation is provided in the last column. For compilation we used gcc v4.8.4 for the C implementation and the NVIDIA CUDA Compiler v7.5.17 for the CUDA implementation.

The results shown are given by an average of $10,000$ signature generations with different keys and messages. To ease the comparison, we reuse the same routines for Salsa20 and message hash function in signing algorithm, and also reuse the same implementation in Hoffstein et al.'s software for everything other than signing.

Table 4. Benchmarking results (in terms of μs) of our CUDA C implementation of a valid signature generation, with an NVIDIA GTX Titan X graphics card and a 3.5GHz Intel Core i7-3770K CPU in single-thread mode, are provided in the column "This Work". Results are compared with Hoffstein et al.'s C implementation with the same CPU. The speedup achieved is given in the last column.

No.	N	p	q	B_s	B_t	d_1, d_2, d_3	Prob[valid]	Security (bits)	Signature Size (bytes)	Signing Time (μs)		Grid Size	Speedup
										[6]	This Work		
1	401	3	2^{18}	240	80	8, 8, 6	37.57%	65	1,706	475	238	24	×2.00
2	439	3	2^{19}	264	88	9, 8, 5	55.46%	70	1,976	367	250	24	×1.47
3	593	3	2^{19}	300	100	10, 10, 8	40.46%	110	2,670	870	311	24	×2.80
4	743	3	2^{20}	336	112	11, 11, 15	53.00%	146	3,530	852	379	24	×2.25
5	401	3	2^{15}	138	46	8, 8, 6	1.11%	82	1,404	25,147	**533**	48	**×47**
6	443	3	2^{16}	138	46	9, 8, 5	8.31%	88	1,662	4,975	**272**	24	**×18**
7	563	3	2^{16}	174	58	10, 9, 8	1.86%	126	2,112	20,097	**639**	48	**×31**
8	743	3	2^{17}	186	62	11, 11, 6	6.01%	179	2,972	13,894	**447**	24	**×31**
9	907	3	2^{17}	225	75	13, 12, 7	1.57%	269	3,628	72,719	**1,617**	48	**×45**

The results in Table 4 only show a slight speedup (1.47-2.80 times) with parameters No.1-4. One reason is that [6] makes use of product-form polynomial as well and performs other polynomial multiplications with the Karatsuba multiplication. That is to say, Hoffstein et al.'s software is highly optimized in terms of efficiency, with lower complexity than the CUDA implementation. Another reason is that the Prob[valid] is fairly high. The CUDA implementation actually by probability generates 9-13 valid signatures of which only one is retrieved and counted.

Results with parameters No.5-9 show an improvement of performance of up to 47 times. Therefore, the NTRU-MLS scheme can adopt new parameters with improved security and reduced signature size, and yet keeps similar latencies. A more compact signature size, although the benefit of which is not observable in our experiments, should save noticeable network delay.

Acknowledgment

Sunar and Dai's work was in part provided by the US National Science Foundation CNS Award #1319130. The CUDA GPU implementation in this work is published at `https://github.com/vernamlab/NTRUMLS`.

References

1. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: FOCS. pp. 124–134 (1994)
2. Chen, L., Jordan, S., Liu, Y.K., Moody, D., Peralta, R., Perlner, R., Smith-Tone, D.: Report on post-quantum cryptography. National Institute of Standards and Technology Internal Report 8105 (February 2016)
3. NSA Suite B Cryptography - NSA/CSS, `https://www.nsa.gov/ia/programs/suiteb_cryptography/`
4. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings. pp. 267–288 (1998), `http://dx.doi.org/10.1007/BFb0054868`
5. Ducas, L., Durmus, A., Lepoint, T., Lyubashevsky, V.: Lattice signatures and bimodal gaussians. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, LNCS, vol. 8042, pp. 40–56. Springer (2013)
6. Hoffstein, J., Pipher, J., Schanck, J., Silverman, J., Whyte, W.: Transcript secure signatures based on modular lattices. In: Mosca, M. (ed.) Post-Quantum Cryptography, Lecture Notes in Computer Science, vol. 8772, pp. 142–159. Springer International Publishing (2014), `http://dx.doi.org/10.1007/978-3-319-11659-4_9`
7. Goldreich, O., Goldwasser, S., Halevi, S.: Public-key cryptosystems from lattice reduction problems. In: Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings. pp. 112–131 (1997), `http://dx.doi.org/10.1007/BFb0052231`

8. Hoffstein, J., Howgrave-Graham, N., Pipher, J., Silverman, J.H., Whyte, W.: NTRUSIGN: digital signatures using the NTRU lattice. In: Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings. pp. 122–140 (2003), http://dx.doi.org/10.1007/3-540-36563-X_9
9. Nguyen, P.Q., Regev, O.: Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures. J. Cryptology 22(2), 139–160 (2009), http://dx.doi.org/10.1007/s00145-008-9031-0
10. Ducas, L., Nguyen, P.Q.: Learning a zonotope and more: Cryptanalysis of ntrusign countermeasures. In: Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings. pp. 433–450 (2012), http://dx.doi.org/10.1007/978-3-642-34961-4_27
11. Lyubashevsky, V.: Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In: ASIACRYPT 2009, pp. 598–616. Springer (2009)
12. Lyubashevsky, V.: Lattice signatures without trapdoors. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012, LNCS, vol. 7237, pp. 738–755. Springer (2012)
13. Harrison, O., Waldron, J.: Efficient acceleration of asymmetric cryptography on graphics hardware. In: Progress in Cryptology–AFRICACRYPT 2009, pp. 350–367. Springer (2009)
14. Manavski, S.A.: Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In: Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on. pp. 65–68. IEEE (2007)
15. Hoffstein, J., Pipher, J., Schanck, J.M., Silverman, J.H., Whyte, W.: Transcript secure signatures based on modular lattices. Cryptology ePrint Archive, Report 2014/457 (2014), http://eprint.iacr.org/
16. Bernstein, D.: The salsa20 family of stream ciphers. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs, Lecture Notes in Computer Science, vol. 4986, pp. 84–97. Springer Berlin Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-68351-3_8
17. Nir, Y., Langley, A.: ChaCha20 and Poly1305 for IETF Protocols, https://tools.ietf.org/html/rfc7539
18. Howgrave-Graham, N.: Advances in Cryptology - CRYPTO 2007: 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007. Proceedings, chap. A Hybrid Lattice-Reduction and Meet-in-the-Middle Attack Against NTRU, pp. 150–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-74143-5_9