Speed Records for NTRU [⋆]

Jens Hermans [⋆⋆], Frederik Vercauteren [⋆⋆⋆], and Bart Preneel

Katholieke Universiteit Leuven, ESAT/SCD-COSIC and IBBT
Kasteelpark Arenberg 10
B-3001 Leuven-Heverlee, Belgium
{jens.hermans,frederik.vercauteren,bart.preneel}@esat.kuleuven.be

Abstract. In this paper NTRUEncrypt is implemented for the first time on a GPU using the CUDA platform. As is shown, this operation lends itself perfectly for parallelization and performs extremely well compared to similar security levels for ECC and RSA giving speedups of around three to five orders of magnitude. The focus is on achieving a high throughput, in this case performing a large number of encryptions/decryptions in parallel. Using a modern GTX280 GPU a throughput of up to 200000 encryptions per second can be reached at a security level of 256 bits. This gives a theoretical data throughput of 47.8 MB/s. Comparing this to a symmetric cipher (not a very common comparison), this is only around 20 times slower than a recent AES implementation on a GPU.

Keywords: NTRU encryption, Graphical Processing Unit, Parallelization, CUDA.

1 Introduction

Graphical Processing Units (GPUs) have long been used only for the rendering of games and other graphical applications. More recent GPUs are also used for general purpose parallel programming, using new programming models. A General Purpose GPU (GPGPU) contains a large number of processor cores (240 for the GTX280 [23]) that run at frequencies that are mostly lower than CPUs (1.2 GHz for the GTX280 GPU compared to 3.8 GHz for a recent Intel Pentium 4 [18]). Compared to a CPU a GPU provides a much larger computing power (several GFlops, or even TFlops for multiple GPUs) for specific parallel applications, because of the large number of cores. The recent change towards general scalar processor cores, that support 32- or 64-bit integer and bitwise operations, offers a new opportunity to implement cryptographic applications on GPUs.

There are several cryptographic ciphers that have a high level of parallelism, making them suitable for implementation on GPU. For performing a single encryption/decryption GPUs might not be very well suited: there is a latency compared

[⋆] This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

[⋆⋆] Research assistant, sponsored by the Fund for Scientific Research - Flanders (FWO).

[⋆⋆⋆] Postdoctoral Fellow of the Fund for Scientific Research - Flanders (FWO).

to a CPU, because of the transfer of the data between main memory and GPU memory. In many applications the focus is not on the latency of a single cryptographic application, but on the throughput: one wants to perform a large number of encryptions/decryptions as fast as possible. In the case of a symmetric block cipher this will be the case when operating on a large block of data (using a suitable block cipher mode). Asymmetric ciphers are not often used in such a mode of operation, but more likely on servers that need to process many different secured connections where a large number of asymmetric cryptographic operations need to be performed. Currently cryptographic co-processors are used to speed up these operations, but a GPU might provide an alternative for these co-processors. An advantage is the fact that GPUs are almost by default present in modern computers and are also much underused. Another advantage is the flexibility: GPUs are easy to reprogram, making it an interesting co-processor to add to large computing farms. The large power consumption of the fastest GPUs is however a disadvantage, especially with a growing focus on the energy performance of data centers. One of the most likely uses of GPUs will be performing attacks on ciphers. GPUs have a very good computing power / price ratio, making them very economic for bulk computations. One can have around 200 GFlops (around 1 TFlops theoretically) for less than €500. In many attacks multiple cryptographic operations need to be performed, or at least part of these operations, so implementing and optimizing the original cryptographic operation on GPU is also of great use for attacks.

The choice for NTRUEncrypt (in short: NTRU) as the cryptographic cipher is less obvious: RSA [24] and ECC [7] are currently the respectively dominant and rising ciphers. NTRU has a large potential as a future cipher, given the very simple nature of it's core operation: the convolution (compared to a modular exponentiation for RSA and repeated squaring/doubling for ECC). This simple operation makes it very suitable for embedded devices with limited computing power, but also for parallelization, since a convolution can be split up over several processors. NTRU also has a good (asymptotic) performance of $O(N^2)$ (or even $O(N \log N)$ using FFT), compared to, for example, $O(N^3)$ for RSA. So, NTRU is expected to outperform RSA and ECC at similar security levels, and NTRU will also provide a good scalability for the future.

Because of these properties of NTRU, it was chosen as the cipher to be implemented on a GPU in this paper. For this paper the *ees1171ep1* parameter set is used, a high security ($k = 256$, the symmetric key size in bits) parameter set as claimed in [27]. Besides this parameter set a special version using product-form polynomials is also implemented. Product-form polynomials improve performance even further. Taking this high security level into account, NTRU performs very well when comparing with RSA (which would require a 15360 bit modulus) and ECC. For high througput applications a speedup of three to five orders of magnitude is reached compared to RSA and ECC. The GPU implementation reaches a throughput of up to 200 000 encryptions per second which is equivalent to a theoretical data throughput of 47.8 MB/s.

Organization In Section 2 previous work on cryptography on GPUs and NTRU implementations is discussed. Next, a brief introduction is given to the NTRU cryptosystem in Section 3, especially on the parameter sets that have been proposed in the literature. In Section 4 the basics of GPGPU programming are explained, with a focus on the CUDA platform. This knowledge of NTRU and CUDA is combined to make an optimized GPU implementation of NTRU in Section 5. Finally the performance of the implementation is evaluated and compared to other implementations and other ciphers in Section 6.

2 Related work

There is already much software available for GPUs, ranging from simple linear algebra packages to complex physical simulations. There has not been much development of cryptographic applications for the GPU, until recently when GPUs started supporting integer and bitwise operations. For example, AES was implemented on GPU [20] [14] [8], offering a maximum throughput of 831 MBytes/s (128 bit key, [20]).

RSA [24] has been implemented before the introduction of recent GPGPU platforms, using the OpenGL API [21] and more recently using modern platforms [26] [11], reaching up to 813 modular exponentiations (1024-bit integers) per second [26]. GPUs are also used to launch attacks, for example elliptic curve integer factoring [5] and brute force attacks, like for wireless networks [25].

There are no GPU implementations for NTRU. NTRU has however been implemented on a variety of platforms, like embedded devices, FPGAs [3] and custom hardware [1]. NTRU turns out to perform very well on devices with limited computing capabilities, given the simple nature of the convolution that is the central encryption/decryption operation. Compared to other modern cryptosystems like ECC, NTRU turns out to be very fast [19].

3 NTRUEncrypt

In this section the basics of NTRU are briefly introduced, based upon [2], to which we refer for further, more complete, information.

Let $\mathbb{Z}$ denote the ring of integers and $\mathbb{Z}_q$ the integers modulo q. NTRUEncrypt is a public-key cryptosystem that works with the polynomial ring $P(N) = \mathbb{Z}[X]/(X^N - 1)$ (and $P_q(N) = \mathbb{Z}_q[X]/(X^N - 1)$), where N is a positive prime. A vector from $\mathbb{Z}^N$ (resp. $\mathbb{Z}_q^N$) can be associated with a polynomial by $f = (f_0, f_1, \ldots, f_{N-1}) = \sum_{i=0}^{N-1} f_i X^i$

The multiplication of two polynomials $h = f \star g$ is defined as the cyclic convolution of their coefficients:

$$h_k = (f \star g)_k = \sum_{i+j \equiv k \mod N} f_i \cdot g_j \; (0 \leq k < N) \tag{1}$$

which is the ordinary polynomial multiplication modulo $X^N - 1$.

The polynomials used in NTRU are selected from several polynomial sets $\mathcal{L}_f, \mathcal{L}_g, \mathcal{L}_r$ and $\mathcal{L}_m$. First the basic operations (key creation, encryption and decryption) of NTRU are introduced and afterwards, in Section 3.1, the structure of the polynomials and the parameter sets are discussed.

Key creation The private key is a polynomial f, chosen at random from the set $\mathcal{L}_f$. Another polynomial $g \in \mathcal{L}_g$ is also chosen at random, but is not needed anymore after key generation. From these polynomials the public key h can be computed as

$$h = p \star f_q^{-1} \star g \mod q \tag{2}$$

where f_q^{-1} is the inverse of f in $P_q(N)$ and p is a polynomial (usually 3 or $X + 2$).

The polynomials f and g generally have small coefficients, while h has large coefficients.

Encryption The message $m \in \mathcal{L}_m$ can be encrypted by choosing a random polynomial $r \in \mathcal{L}_r$ as a blinding factor and computing the ciphertext as

$$e = r \star h + m \mod q. \tag{3}$$

In practical schemes the message is padded with random bits and masked. For this paper, these steps are ignored, and only the computation of $r \star h + m \mod q$ is considered.

Decryption Decryption can be done by convolving the ciphertext e with the private key f

$$a \equiv e \star f \equiv p \star r \star g + m \star f \mod q \tag{4}$$

and next convolving by $f_p^{-1} \mod p$. By a careful choice of f it can be assured that $f_p^{-1} = 1$, so only a reduction mod p is needed.

One of the problems NTRU faces are decryption failures: the first step of the decryption only computes $a \mod q$ and not a. The problem is that knowing $a \mod q$ is not enough to know $a \mod p$. The problem of decryption failures has been studied extensively in [17]. In this paper it suffices to pick the coefficients of a from $(-q/2, q/2]$ and assume the probability of decryption failures is negligibly low.

3.1 Parameter sets

The parameter N must always be chosen to be prime, since composites allows the problem to be decomposed [13]. The parameter q is mostly chosen as a power of 2, to ease the computations modulo q. The parameter p must be relatively prime to q, but it is not necessary that p is an integer, it can be a polynomial. Popular choices for p are 3 and $X + 2$.

Besides the parameters N, p, q there are the sets of polynomials $\mathcal{L}_f, \mathcal{L}_g, \mathcal{L}_m, \mathcal{L}_r$ that have to be defined. The message space $\mathcal{L}_m$ is defined as $P_p(N)$, since the message is obtained during the decryption after reducing modulo p.

The other sets of polynomials are chosen as ternary (for $p = 3$) or binary (for $p = X + 2$) polynomials.

Ternary polynomials Define $\mathcal{L}(d_x, d_y)$ as the set of all ternary polynomials that have d_x coefficients set to 1 and d_y coefficients set to -1 (all other coefficients are 0).

One of the most natural choices for the polynomial sets is

$$\mathcal{L}_f = \{1 + p \star F : F \in \mathcal{L}(d_f, d_f)\}\ ,\ \mathcal{L}_r = \mathcal{L}(d_r, d_r)\ ,\ \mathcal{L}_g = \mathcal{L}(d_g, d_g)$$

which is also used in the most recent standards draft [27]. The choice of $\mathcal{L}_f$ as $1+p\star F$ guarantees that $f_p^{-1} = 1$.

For ternary polynomials p is set to 3.

Binary polynomials Binary polynomials offer an alternative for ternary polynomials and are much easier to implement in hardware and software. A disadvantage is that binary polynomials are by definition unbalanced, so $f(1) \neq 0$. As a consequence information on m, namely $m(1)$, leaks.

In [12] the following parameters are used:

$$\mathcal{L}_f = \{1 + p \star F : F \in \mathcal{L}(d_f, d_f)\}\ ,\ \mathcal{L}_r = \mathcal{L}(d_r, 0)\ ,\ \mathcal{L}_g = \mathcal{L}(d_g, 0)$$

Product-form polynomials The central operation when encrypting is a convolution with a binary/ternary polynomial. The number of non-zero elements in $r \in \mathcal{L}_r$ is crucial for the performance of the encryption operation. A smaller number of non-zero elements will make the convolution faster (and lower memory usage, depending on the storage strategy) but will also degrade the security. By taking

$$\mathcal{L}_r = \{r_1 \star r_2 + r_3 : r_1 \in \mathcal{L}_{r_1}, r_2 \in \mathcal{L}_{r_2}, r_3 \in \mathcal{L}_{r_3}\}$$

with $d_{r_1}, d_{r_2}, d_{r_3} \ll d_r$ the convolution is still secure, since $r_1 \star r_2 + r_3$ still contains roughly the same amount of randomness as a single random r [15]. For our implementation $d_{r_1} = d_{r_2} = d_{r_3} = 5$, so each polynomial r_i has 10 non-zero coordinates. The performance is however increased drastically. The convolution $t = r \star h \mod q$ can be computed in several steps as in [3]:

$$t_1 \leftarrow r_2 \star h \quad ; \quad t_2 \leftarrow r_1 \star t_1 \quad ; \quad t_3 \leftarrow r_3 \star h \quad ; \quad t \leftarrow t_2 + t_3 \mod q \qquad (5)$$

Since each of r_1, r_2, r_3 have a low number of non-zero elements, the convolutions in (5) are much faster, requiring less additions than $r \star h$. Another advantage is the lower storage requirement.

4 GPU programming

4.1 The CUDA platform

The CUDA programming guide [22] explains in detail all aspects of the platform and programming model and was used as a basis for the following sections. The GTX280 that was used for this paper is a GPU that belongs to the range of Tesla Architecture GPUs from Nvidia. The Tesla architecture is based upon an array of multiprocessors (30 for the GTX280) that each contain 8 scalar processors. A multiprocessor is operated as a SIMT-processor (Single-Instruction, Multiple-Thread): a single instruction uploaded to the GPU causes multiple threads to perform the same scalar operation (on different data). The CUDA programming model from Nvidia, that is used to program their GPUs, provides a C-like language to program the GPU.

Programming model As stated above, all programming is done using scalar operations: one needs to program a single thread which will then be executed in multitude on the GPU. Threads are grouped into blocks. All blocks together form a 'grid' of blocks. Threads within the same block can use shared memory. Both threads and blocks can be addressed in a multi-dimensional way. All scheduling of instructions (threads) on the multiprocessors is hidden from the programmer and is done on-chip. Threads are scheduled in *warps* of 32 threads. For optimal performance divergent branching inside the same half-warp (16 threads) must be avoided: each thread in a half-warp must execute the same instruction, otherwise the execution will be serialized. If divergent branching occurs, one possible strategy is to ensure that the thread ID for which divergence occurs coincides with a change of half-warp.

Memory A multiprocessor contains fast on-chip memory in the form of registers, shared memory and caches. Off-chip memory is also available in the form of global memory and specialized texture and constant memory. The global memory is not cached. The GTX280 provides 1GB of off-chip memory.

Each of the memory types has specific features and caveats [1]:

- Global memory: as the global memory is off-chip there is a large performance hit (hundreds of cycles). Another issue is that multiple threads might access different global memory addresses at the same time, which creates a bottle-neck and forces the scheduler to stop the execution of the block until all memory is loaded. It is recommended to run a large number of blocks, to ensure the scheduler can keep the multiprocessors busy, while memory loading takes place. One way to avoid such large performance penalties are coalesced memory reads, in which all threads from a half-warp access either the same address or a block of consecutive addresses. In the case of loading a single address the total cost is only one memory load.
- Registers: care has to be taken to limit the number of registers per thread as the registers are shared among all threads and blocks running on the same multiprocessor.

[1] Texture memory is not used in this paper, so details have been omitted.

- Shared memory: shared memory is stored in banks, such that consecutive 32 bits are stored in consecutive banks. When accessing shared memory one needs to ensure that threads within the same warp access different banks, to avoid 'bank conflicts'. Bank conflicts result in serialization of the execution.
- Constant memory: the advantage of using constant memory is the presence of a special read-only cache, which allows for fast access times.

Instructions Almost all operations that are available in C can be used in CUDA. CUDA only uses 32-bit (`int`, `float`) and 64-bit variables (`long`, `double`) for arithmetic, other types are converted first. In this paper, we will refer to 32-bit integers as '`int`' (or just 'integer') and to 64-bit integers as a '`long`'. Integer addition and bitwise operations take 4 clock cycles. 32-bit integer multiplication takes 16 cycles. Integer division and modulo are expensive and should be avoided.

5 The implementation

For the implementation the *ees1171ep1* parameter set from [27] is used. This parameter set (with ternary polynomials and $N = 1171, p = 3, q = 2048 = 2^{11}, d_r = 106$) is one of the three strongest from the draft standard. Considering the relatively young age of NTRU and recent attacks (e.g. [16]), it is better to be rather conservative in the parameter choices and take one of the strong parameter sets.

Two implementations were made: one using the default ternary polynomials, the other using product-form ternary polynomials. In the last case $d_{r_1} = d_{r_2} = d_{r_3} = 5$.

The generation of random data (needed for encryption) is performed by the CPU, although parallel implementations exist for CUDA. There are several reasons for this choice: first of all it is the goal of this paper to compare the central NTRU operation, the convolution, and not to compare choices of random number generators. By computing the random numbers beforehand on CPU, any influence of the choice of the random generator is excluded. Second, one might consider an attack strategy in which the opponent would explicitly choose r, instead of using random numbers. Another advantage of performing the generation of r on CPU is exploiting the parallel computation by using both CPU and GPU.

5.1 Operations

Both parallel encryption (two variants) and parallel decryption are implemented on CUDA. The superscript i in m^i denotes the i-th message that is used in the parallel computation. The operations are defined as follows:

- **Encryption:** given $r^i \in \mathcal{L}_r$, h^i and $m^i \in \mathcal{L}_m$ (for $i \in [0, P)$, with P the number of parallel encryption operations) the kernel computes $e^i = r^i \star h^i + m^i \mod q$. Two strategies for the public key are considered: one which uses the same public key for all encryptions ($\forall i : h_i = h$) and one with different public keys for every operation.

– **Decryption:** given e^i and f, compute m^i. The private key is the same for all decryptions.

Key generation was not implemented, although situations exist where one would like to generate multiple keys in parallel.

For encryption both ordinary and product-form ternary polynomials are used as r.

The decryption operation can be written as

$$e \star f \equiv e \star (1 + p \star F) \equiv e + (e \star F) + (e \star F) \ll 1 \mod q \tag{6}$$

where "$\ll$" is a left bit shift. Besides some extra scalar operations for each coefficient, one can reuse the encryption algorithm. In the next sections only encryption is discussed. The results section only includes results for the case that F is an ordinary ternary polynomial. Because there was no performance difference compared to encryption, decryption was not implemented for product-form polynomials.

5.2 Memory usage - Bit packing

Since all data must be transferred from the main computer memory to the GPU (device) memory, it is in the best interest to limit the amount of memory used.

One standard technique is bit packing. The ternary coefficients of r can be encoded as 2 bits, of which 32 can be packed into a 64-bit `long`. The coefficients of h are each 11 bit long, allowing for up to 5 coefficients to be stored in a `long`. We however pack only 4 elements of h in a long. The extra unused bits come in handy when performing an addition on the entire long, so that the overflow does not corrupt one of the packed values stored higher in the bit array. Note that although the polynomial m also has ternary coefficients we choose to store it using 11 bits per element. This way, the result of the encryption e (which is mod q) can be written in the same space as m, which results in a smaller memory usage. In total 623 `long`'s are required to store h, m and r.

For the implementation with product-form polynomials the values of r_1, r_2 and r_3 can be stored in a different way. Instead of encoding each ternary coefficient as two bits, the indices of the non-zero coefficients are stored, as in [3]. Since each index is in $[0, N-1]$, $\lceil \log_2(N) \rceil = 11$ bits are needed to store each index. These indices are again packed, but not aligned to 16 bit multiples, since the access is sequential (see further). The memory consumption is only lowered moderately to 592 `long`s, but the new structure of the convolution has a large impact on the construction of the loops and thus the performance.

Since multiple encryption/decryption operations are performed, multiple messages m and blinds r need to be uploaded to the device. All variables are stored in one large array of `long`'s, e.g. a single m^b is packed to 293 `long`s, with the total array being $293 \times P$ `long`'s. Note that the time for bit packing the data on CPU is not included in the timing results and that all host-memory is page-locked.

In the next sections and the algorithms in Appendix A, we use the notation $x_{\text{packed},i}$ to refer to the `long` containing the i-th element of the x polynomial (which is denoted as x_i). $P(i)$ is used to denote the index of the long that contains x_i. When there is a reference to x_i in the pseudo-code, the index calculation and decoding are implicit.

5.3 Encoding

The coefficients of h are encoded as 11 bit integers, in the range $[0, q-1]$. The blind r, consisting of ternary coefficients, is encoded by mapping $\{0, 1, -1\}$ to 2-bit values (which can be chosen arbitrarily). The message m also consists of ternary coefficients, but for efficient computation, these are loaded in the memory space that will contain the result e. Because of this, the ternary coefficients are stored as 11-bit values in two's complement (e.g. $(-1)_3 = 2^{11} - 1$).

5.4 Blocks, threads and loop nesting

Parallelism is present at two levels: at the level of a single encryption, which is split over multiple threads, and at the level of the parallel encryptions, which are split over the blocks. When performing a single encryption, one needs to access all elements r_i^b, h_j^b and e_k^b. Each block (block index denoted with the superscript b) is responsible for doing a single encryption/decryption operation. To make storing e_k as efficient as possible, each thread is responsible of computing 4 coefficients of e, which implies that each thread writes only one long.

For the normal ternary polynomials, the algorithm executed by each thread is presented in Algorithm 3. There is an implicit 'outer loop' that iterates over k (the parallel threads). The middle loop (over i) selects the element from r^b and then uses simple branching and addition (no multiplications).

Algorithm 1 shows the algorithm for the product-form ternary polynomials. The implicit outer loop is the same, but the computation inside is completely different. The computation of $r_2 \star h$ is split over all threads and the results are stored (packed) in shared memory. Unlike the other convolutions in Algorithm 1, all threads need all indices of $r_2 \star h$ and not just the $k \ldots k+3$-th coefficients.

Since r_1, r_2 and r_3 are stored using indices, the convolution algorithm is different from that used for ordinary polynomials. Algorithm 2 describes part of such a convolution. Again, only 4 coefficients of the result are computed, which matches the division of the convolution among the threads.

5.5 Memory access

Since the convolutions are very simple operations, using only addition and some index-keeping operations, the memory access will be the bottleneck. One of the solutions is to explicitly cache the elements of r and h in registers (the GPU does not have a cache for global 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. For h no significant benefits were observed, so the caching was omitted. The main reason is that the packed coefficients of h are less often accessed (many of the r_i are zero) and they are accessed in a more or less random pattern, so caching them for more than one iteration (over i in Algorithm 3) makes no sense. There is however a benefit from executing multiple threads in parallel: when thread t accesses h_j, thread $t+1$ will at the same time access h_{j+4}, which is always stored in the next `long`. This means that memory access is coalesced, although bad alignment of the memory blocks will prevent the full speedup.

For product-form polynomials the number of memory accesses is much lower: the space used to store r is smaller. As r_1,r_2 and r_3 are accessed only once, this means a drop in memory access from 296 to 48 bytes per block. The number of accesses to h also goes down: only the convolutions $r_2 \star h$ and $r_3 \star h$ need access to h. r_2 and r_3 each have 10 non-zero coefficients, giving a total of 20 accesses to h for each element in the result, so $20 \times 1171 = 23420$ `long`s per block, compared to $2Nd_r = 248252$ `long`s per block for ordinary polynomials.

Note that the access to e is coalesced, since each thread accesses a consecutive `long`.

5.6 Branching

Almost no divergent branching occurs during the execution of the algorithms. In the case of normal polynomials branching on r_i is not divergent, as each thread has the same value for i. The only divergent branches are for the modulo computation. There is one aspect when using product-form polynomials in Algorithm 1 that might cause a performance hit: the thread synchronization. Since the intermediate result t_{shared} is shared among all threads, all threads should wait for the completion of that computation.

6 Results

In this section the results of the GPU implementations are compared to a simple CPU implementation in C and other implementations found in the literature. The CPU tests were performed on an Intel Core2 Extreme CPU, running at 3.00GHz. This processor has four cores, but only one of these cores is used as the CPU implementation is not parallel. The GPU simulations were performed on a GTX280. To verify that all implementations were correct, the output was verified (with success) against a reference implementation in Magma [6].

Table 1 shows the results expressed as milliseconds per operation (or operations per second). Results for different h_i are, obviously, only available for the GPU when doing multiple (20000) operations in parallel. The times in Table 1 are the minimal times over 10 identical experiments. All results are expressed as wall clock time, since this is the only way to be able to compare CPU and GPU. Taking the minimum time ensures that clearing of the cache or context switches do not bias the results. Clearing of the cache and context switches depend heavily on the environment in which the program is used, so it would not be fair to include these in the measurements. Overall, the GPU times had a small variance, so the difference between average time and minimal time was negligible. The time for copying data from main to GPU memory is included in the GPU performance figures.

The CPU implementation does not use any optimizations like bit packing and just consists of a few nested loops. The CPU implementation only performs one single encryption/decryption. Despite the fact that the CPU implementation is not optimized, we use it as a rough basis for comparison for the GPU version. The available performance results for previous implementations are for different (less secure) parameter sets, which makes it very hard to compare.

From Table 1 it is clear that encryption and decryption have roughly the same performance: the extra element-wise operations for decryption do not take much time. This is also the reason that decryption was not implemented separately for product-form ternary polynomials, since it would show the same performance. Encryption with the same h is slightly faster than using different h^i, although an explanation for this has not been found [2].

Figure 1 shows the subsequent gain in performance when increasing the number of parallel encryptions (for ordinary polynomials). Around 2^{11} encryptions the GPU approaches its maximum performance, larger numbers of parallel encryptions yield only a slight improvement in the number of operations per second.

Table 1 shows that for all implementations, product-form polynomials are much faster, as expected by the lower number of memory accesses in Section 5.5. The performance increases by almost a factor 10 compared to ordinary polynomials. Again a small difference is observed between encryption with the same and different h^i.

Table 2 compares the CPU and GPU implementations with previous work on NTRU and to some RSA and ECC implementations. A note of caution is due, since the previous NTRU implementations use much lower security parameters and because the platforms that are used are totally different. Also note that the amount of data encrypted per operation is different. As a very rough extrapolation to convert the results for the other NTRU implementations to the security level of our implementation one can use the $O(N^2)$ asymptotic performance of NTRU. This drastically lowers the performance measures for the other NTRU implementations, ignoring even the increase of q and d_r. For applications with a focus on high throughput (many op/s), the CUDA implementation for product-form polynomials outperforms all other NTRU implementations (taking the higher security parameters and amount of data into account). The implementation with product-form polynomials gives a speed of more than 200 000 encryptions per second or 41.8 MByte/s. For applications that need to perform a small number of encryptions with low latency, the parallelization of CUDA does not give much speedup compared to the CPU. However, when comparing NTRU with RSA and ECC, the speedup is large: up to 1300 times faster than 2048-bit RSA and 117 times faster than ECC NIST-224 when comparing the number of encryptions per second (or up to 1113 times faster than 2048-bit RSA when comparing the data throughput). In addition, the security level of NTRU is much higher: when extrapolating to RSA and ECC with $k = 256$ bit

[2] The opposite result was expected. As h was not stored in constant memory, there should be no benefit from caching.

security, this would add an extra factor of around 10 for ECC and around 400 for RSA (assuming $O(N^3)$ operations for RSA and ECC, where N is the length of a message block). So, in this extrapolation, NTRU has a speedup of five orders of magnitude compared to RSA and three orders of magnitude compared to ECC. The results listed for RSA encryption on CPU are operations with a small public key ($e = 17$), which allows for further optimization that has not been done for the RSA GPU implementation.

	Encryption (different h^i)		Encryption (same h^i)		Decryption	
	μs/op	op/s	μs/op	op/s	μs/op	op/s
Ordinary						
CPU	-	-	$10.5 \cdot 10^3$	(95)	$10.5 \cdot 10^3$	(95)
GPU, 1 op.	-	-	$1.75 \cdot 10^3$	-	$1.87 \cdot 10^3$	-
GPU, 20000 ops	41.3	24 213	40.0	25 025	41.1	24 331
Product-form						
CPU	-	-	$0.31 \cdot 10^3$	(3225.8)	-	-
GPU, 1 op.	-	-	$0.16 \cdot 10^3$	-	-	-
GPU, $\sim 2^{16}$ ops	4.58	218 204	4.51	221 845	-	-

Table 1. Performance comparison of NTRU on an Intel Core2 CPU and a Nvidia GTX280 GPU using ordinary and product-form ternary polynomials ($N = 1171, q = 2048, p = 3$).

7 Conclusion

In this paper NTRU encryption/decryption was implemented for the first time on GPU. Several design choices, such as the NTRU parameters sets, are compared. The exact implementation is analysed in detail against the CUDA platform, explaining the impact of every choice by looking at the underlying effects on branching, memory access, blocks & threads... Although the programming is done in C, the CUDA model has its own specific ins and outs that take some time to learn, making a good implementation not very straightforward.

Many external factors, like power consumption, cost, reprogrammability, context (latency vs throughput), space... besides the speed of the cipher influence the choice of platform. In areas in which reprogrammability, cost and throughput are important and power consumption is of lesser importance, a GPU implementation is a very good option.

For 2^{16} encryptions a peak performance of around 218 000 encryptions/s (or 4.58×10^{-6} s/encryption) is reached, using product-form polynomials. This corresponds to a theoretical data throughput of 47.8 MB/s. The GPU performs at its best when performing a large number of parallel NTRU operations. Parallel NTRU implementations could serve well on servers processing many secured connections or in various attack strategies in which many (partial) encryption operations are

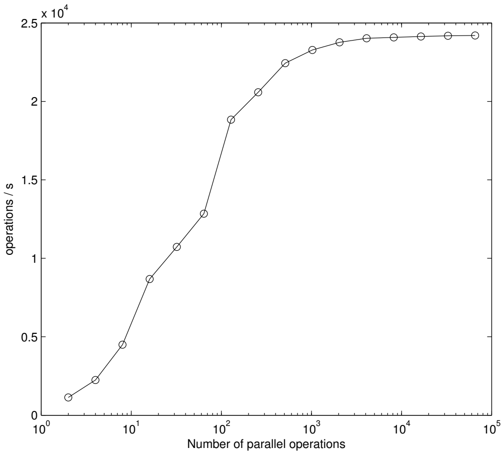

Fig. 1. NTRU encryption operations per second using ordinary polynomials and the same h $(N = 1171, q = 2048, p = 3)$.

Platform		(N, q, p)	**Enc/s**	**Dec/s**	bit/op
FPGA [3]	Xilinx Virtex 1000EFG860 @ 50 MHz	$(251, 128, X + 2)$	$193 \cdot 10^3$	-	251
Palm [3]	Dragonball @ 20 MHz (C)	Product form	21	11	
Palm [3]	Dragonball @ 20 MHz (ASM)	$(k < 80)$	30	16	
ARM C [3]	ARM7TDMI @ 37 MHz		307	148	
FPGA [1]	Xilinx Virtex 1000EFG860 @ 500kHz	$(167, 128, 3)$	18	8.4	250
		$(k \ll 80)$			
C	Intel Core2 Extreme @ 3.00GHz	$(1171, 2048, 3)$	95	95	1756
CUDA	GTX280 (1 op)	$(k = 256$ [27]$)$	571	546	
CUDA	GTX280 (20000 ops)		$24 \cdot 10^3$	$24 \cdot 10^3$	
C	Intel Core2 Extreme @ 3.00GHz	$(1171, 2048, 3)$	$3.22 \cdot 10^3$	-	1756
CUDA	GTX280 (1 op)	Product form	$6.25 \cdot 10^3$	-	
CUDA	GTX280 ($\sim 2^{16}$ ops)	$(k = 256$ [27]$)$	$218 \cdot 10^3$	-	
RSA comparison					
CUDA [26]	Nvidia 8800GTS	1024 bit		813	1024
C++ [9]	Intel Core2 @ 1.83GHz	$(k = 80$ [4]$)$	$(14 \cdot 10^3)$	657	1024
CUDA [26]	Nvidia 8800GTS	2048 bit		104	2048
C++ [9]	Intel Core2 @ 1.83GHz	$(k = 112$ [4]$)$	$(6.66 \cdot 10^3)$	168	2048
ECC comparison					
CUDA [26]	Nvidia 8800GTS (PointMul)	ECC NIST-224		$1.41 \cdot 10^3$	
C [10]	Intel Core2 @ 1.83 GHz (ECDSA)	$(k = 112$ [4]$)$		$1.86 \cdot 10^3$	

Table 2. Comparison of several NTRU, RSA and ECC implementations. The chosen parameter set and claimed security level (k) is listed for all ciphers. The number of operations per second is listed, together with the amount of data encrypted/decrypted per operation (excluding all padding, headers...)

needed. A single NTRU operation on GPU is still faster than a (simple) CPU implementation, but the speedup is limited. Even then a GPU might be interesting to simply move load off the CPU.

Comparing NTRU to other cryptosystems like RSA and ECC shows that NTRU, at a high security level, is much faster than RSA (around five orders of magnitude) and ECC (around three orders of magnitude). Even when only performing a single operation NTRU is still faster by around a factor of 35 for 2048 bit RSA and 3 for ECC NIST-244. Because of the ways NTRU can be parallelized, NTRU also clearly outperforms RSA and ECC for high-throughput applications. So, both for low-latency (single operation) and high-throughput (multiple operations) applications NTRU on GPU outperforms RSA and ECC.

References

1. A.C. Atıcı, L. Batina, J. Fan, I. Verbauwhede, and S.B.O. Yalçın. Low-cost implementations of NTRU for pervasive security. In *ASAP 2008*, pages 79–84. IEEE Computer Society, 2008.
2. ECRYPT AZTEC. Lightweight Asymmetric Cryptography and Alternatives to RSA, 2005.
3. D.V. Bailey, D. Coffin, A.J. Elbirt, J.H. Silverman, and A.D. Woodbury. NTRU in Constrained Devices. In *CHES 2001*, volume 2162 of *LNCS*, pages 262–272. Springer, 2001.
4. E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Recommendation for Key Management. *NIST special publication*, 800:57, 2007.
5. D.J. Bernstein, H.C. Chen, M.S. Chen, C.M. Cheng, C.H. Hsiao, T. Lange, Z.C. Lin, and B.Y. Yang. The Billion-Mulmod-Per-Second PC. In *SHARCS 2009*, pages 131–144, 2009.
6. W. Bosma, J. Cannon, and C. Playoust. The Magma Algebra System I: The User Language. *Journal of Symbolic Computation*, 24(3-4):235–265, 1997.
7. H. Cohen, G. Frey, and R. Avanzi. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* CRC Press, 2006.
8. D. Cook, J. Ioannidis, A.D. Keromytis, and J. Luck. Cryptographics: Secret key cryptography using graphics cards. In *CT-RSA 2005*, volume 3376 of *LNCS*, pages 334–350. Springer, 2005.
9. W. Dai. Crypto++: benchmarks. `http://www.cryptopp.com/benchmarks.html`.
10. Ecrypt Ebats. ECRYPT benchmarking of asymmetric systems. `http://www.ecrypt.eu.org/ebats/`, 2007.
11. S. Fleissner. GPU-Accelerated Montgomery Exponentiation. In *International Conference on Computational Science (1)*, volume 4487 of *LNCS*, pages 213–220. Springer, 2007.
12. Consortium for Efficient Embedded Security. Efficient embedded security standards #1: Implementation aspects of NTRU and NSS, Version 1. 2002.
13. C. Gentry. Key Recovery and Message Attacks on NTRU-Composite. In *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 182–194. Springer, 2001.
14. O. Harrison and J. Waldron. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In *CHES 2007*, volume 4727 of *LNCS*, pages 209–226. Springer, 2007.
15. J. Hoffstein and J.H. Silverman. Random small Hamming weight products with applications to cryptography. *Discrete Applied Mathematics*, 130(1):37–49, 2003.

16. N. Howgrave-Graham. A Hybrid Lattice-Reduction and Meet-in-the-Middle Attack Against NTRU. In *CRYPTO 2007*, volume 4622 of *LNCS*, pages 150–169. Springer, 2007.

17. N. Howgrave-Graham, P.Q. Nguyen, D. Pointcheval, J. Proos, J.H. Silverman, A. Singer, and W. Whyte. The impact of decryption failures on the security of ntru encryption. In *CRYPTO 2003*, volume 2729 of *LNCS*, pages 226–246. Springer, 2003.

18. Intel. Intel Pentium 4 - SL8Q9 Datasheet, 2008.

19. P. Karu and J. Loikkanen. Practical Comparison of Fast Public-key Cryptosystems. `http://www.tml.tkk.fi/Opinnot/Tik-110.501/2000/papers/`, 2001.

20. S.A. Manavski. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In *ICSPC 2007*, pages 65–68. IEEE, Nov. 2007.

21. A. Moss, D. Page, and N.P. Smart. Toward Acceleration of RSA Using 3D Graphics Hardware. In *Cryptography and Coding, IMA Int. Conf. 2007*, volume 4887 of *LNCS*, pages 364–383. Springer, 2007.

22. Nvidia. Compute Unified Device Architecture Programming Guide, 2007.

23. Nvidia. GeForce GTX280 - GeForce GTX 200 GPU Datasheet, 2008.

24. R.L. Rivest, A. Shamir, and L.M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

25. M. Settings. Password crackers see bigger picture. *Network Security*, 2007(12):20–20, 2007.

26. R. Szerwinski and T. Güneysu. Exploiting the Power of GPUs for Asymmetric Cryptography. In *CHES 2008*, volume 5154 of *LNCS*, pages 79–99. Springer, 2008.

27. W. Whyte, N. Howgrave-Graham, J. Hoffstein, J. Pipher, J.H. Silverman, and P. Hirschhorn. IEEE P1363.1: Public Key Cryptographic Techniques Based on Hard Problems over Lattices.

A Code listings

Algorithm 1 Pseudo-code for a single NTRU encryption (product-form polynomials)

```
$b \leftarrow$ blockID
$k \leftarrow 4 *$ threadID
Allocate $e_{\text{temp}}[0 \ldots 3] \leftarrow 0$
Allocate $t_{\text{shared}}[0 \ldots N-1]$
$t_{\text{shared}}[k \ldots k+3] \leftarrow Convolve(h^b, r^b_{2,+}, r^b_{2,-}, k, t_{\text{shared}}[k \ldots k+3])$
Synchronize threads
$e_{\text{temp}}[0 \ldots 3] \leftarrow Convolve(t_{\text{shared}}, r^b_{1,+}, r^b_{1,-}, k, e_{\text{temp}}[0 \ldots 3])$
$e_{\text{temp}}[0 \ldots 3] \leftarrow Convolve(h^b, r^b_{3,+}, r^b_{3,-}, k, e_{\text{temp}}[0 \ldots 3])$
for $l = 0$ to $3$ do
    $e^b_{k+l} \leftarrow m^b_{k+l} + e_{\text{temp}}[l] \mod q$
end for
```

Algorithm 2 Pseudo-code for a single product-form convolution. $Convolve(h, r^+, r^-, k, t)$

```
Require: $h$: an ordinary polynomial,
    $r^+$, $r^-$: the positions of the $+1$ and $-1$ elements in the polynomial $r$,
    $t$: result of the convolution,
    $k$: offset of the results that need to be calculated.
Ensure: $t[k \dots k+3] = \{h \star r\}_{k \dots k+3}$
$k \leftarrow 4 * \text{threadID}$
for $l = 0$ to $d_{r-1} - 1$ do
    $i \leftarrow r_l^+$
    for $\delta_k = 0$ to $3$ do
        $t[k + \delta_k] \leftarrow t[k + \delta_k] + h_{(k+\delta_k - i \mod N)}$
    end for
end for
for $l = 0$ to $d_{r-1} - 1$ do
    $i \leftarrow r_l^-$
    for $\delta_k = 0$ to $3$ do
        $t[k + \delta_k] \leftarrow t[k + \delta_k] - h_{(k+\delta_k - i \mod N)}$
    end for
end for
return $t[k \dots k+3] \mod q$
```

Algorithm 3 Pseudo-code for a single NTRU encryption (ordinary polynomials)

```
$b \leftarrow \text{blockID}$
$k \leftarrow 4 * \text{threadID}$
Allocate $e_{\text{temp}}[0 \dots 3] \leftarrow 0$
for $i = 0$ to $10$ do
    for $l = 0$ to $3$ do
        if $P(i) \neq P(i-1)$ then
            $r_{\text{cache}} \leftarrow r^b_{\text{packed},i}$
        end if
        $r_{\text{elem}} \leftarrow r_i$ (from $r_{\text{cache}}$)
        $j \leftarrow k + l - i \mod N$
        if $r_{\text{elem}} = 1$ then
            $e_{\text{temp}}[l] \leftarrow e_{\text{temp}}[l] + h^b_j$
        end if
        if $r_{\text{elem}} = -1$ then
            $e_{\text{temp}}[l] \leftarrow e_{\text{temp}}[l] - h^b_j$
        end if
    end for
end for
for $l = 0$ to $3$ do
    $e^b_{k+l} \leftarrow m^b_{k+l} + e_{\text{temp}}[l] \mod q$
end for
```

Algorithm 4 Pseudo-code for a single NTRU Decryption

```
Require: $F$: the private key ($f = 1 + p \star F$)
    $e$: the encrypted message
$k \leftarrow 4 * \text{threadID}$
Execute Algorithm 3, taking $m = 0$, $r = F$ and $h = e$ and obtaining $t[0 \dots 3]$.

for $l = 0$ to $3$ do
    $t[l] \leftarrow t[l] + (t[l] \ll 1) + e_{k+l}$
    $tmp \leftarrow t[l] - p * ((p^{-1} \mod q) * t[l] \gg \log_2 q)$
    $(t[l] > q) \Rightarrow (tmp \leftarrow tmp + 1)$
    $t[l] \leftarrow tmp$
end for
```