Parallel Shortest Lattice Vector Enumeration on Graphics Cards*

Jens Hermans[1,**], Michael Schneider[2], Johannes Buchmann[2], Frederik Vercauteren[1,***], and Bart Preneel[1]

[1] Katholieke Universiteit Leuven - ESAT/SCD-COSIC and IBBT
{Jens.Hermans,Frederik.Vercauteren,Bart.Preneel}@esat.kuleuven.be
[2] Technische Universität Darmstadt
{mischnei,buchmann}@cdc.informatik.tu-darmstadt.de

Abstract. In this paper we present an algorithm for parallel exhaustive search for short vectors in lattices. This algorithm can be applied to a wide range of parallel computing systems. To illustrate the algorithm, it was implemented on graphics cards using CUDA, a programming framework for NVIDIA graphics cards. We gain large speedups compared to previous serial CPU implementations. Our implementation is almost 5 times faster in high lattice dimensions.

Exhaustive search is one of the main building blocks for lattice basis reduction in cryptanalysis. Our work results in an advance in practical lattice reduction.

Keywords: Lattice reduction, ENUM, parallelization, graphics cards, CUDA, exhaustive search.

1 Introduction

Lattice-based cryptosystems are assumed to be secure against quantum computer attacks. Therefore these systems are promising alternatives to factoring or discrete logarithm based systems. The security of lattice-based schemes is based on the hardness of special lattice problems. Lattice basis reduction helps to determine the actual hardness of those problems in practice. In the past few years there has been increased attention to exhaustive search algorithms for lattices, especially to implementation aspects. In this paper we consider parallelization and special hardware for the exhaustive search.

* The work described in this report has in part been supported by the Commission of the European Communities through the ICT program under contract ICT-2007-216676. The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability. 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).

D.J. Bernstein and T. Lange (Eds.): AFRICACRYPT 2010, LNCS 6055, pp. 52–68, 2010.

Lattice reduction is the search for short and orthogonal vectors in a lattice. The algorithm used for lattice reduction in practice today is the BKZ algorithm of Schnorr and Euchner [SE91]. It consists of two main parts, namely an exhaustive search ('enumeration') for shortest, non-zero vectors in lower dimensions and the LLL algorithm [LLL82] for the search for short (not *shortest*) vectors in high dimensions. The BKZ algorithm is parameterized by a blocksize parameter β, which determines the blocksize of the exhaustive search algorithm inside BKZ.

Algorithms for exhaustive search were presented by Kannan [Kan83] and by Fincke and Pohst [FP83]. Therefore, the enumeration is sometimes referred to as KFP-algorithm. Kannan's algorithm runs in $2^{\mathcal{O}(n \log n)}$ time, where n denotes the lattice dimension. Schnorr and Euchner presented a variant of the KFP exhaustive search, which is called ENUM [SE91]. Roughly speaking, enumeration algorithms perform a depth first search in a search tree that contains all lattice vectors in a certain search space, i.e., all vectors of Euclidean norm less than a specified bound. The main challenge is to determine which branches of the tree can be cut off to speed up the exhaustive search. Enumeration is always executed on lattice bases that are at least LLL reduced in a preprocessing step, as this reduces the runtime significantly compared to non-reduced bases.

The LLL algorithm runs in time polynomial in the lattice dimension and therefore can be applied in high lattice dimensions ($n > 1000$). The runtime of all known exhaustive search algorithms is exponential in the dimension, and therefore can only be applied in blocks of smaller dimension ($n \lessapprox 70$). With this, the runtime of BKZ increases exponentially in the blocksize β. As in BKZ, enumeration is executed very frequently, it is only practical to choose blocksizes up to 50. For high blocksize, our experience shows that ENUM takes 99% of the time of BKZ.

There are numerous works on parallelization of LLL [Vil92, HT98, RV92, Jou93] [Wet98, BW09]. Parallel versions of lattice enumeration were presented in the masters theses of Pujol [Puj08] and Dagdelen [Dag09] (in french and german language, respectively). Both approaches are not suitable for GPU, since they require dynamic creation of new threads, which is not possible for GPUs.

Being able to parallelize ENUM means to parallelize the second (more time consuming) building block of BKZ, which reduces the runtime of the most promising lattice reduction algorithm in total.

As a platform for our parallel implementation we have chosen graphical processing units (GPUs). Because of their design to perform identical operations on large amounts of graphical data, GPUs can run large numbers of threads in parallel, provided the threads execute similar instructions. We can take advantage of this design and split up the ENUM algorithm over several identical threads. The computation power of GPU rises faster than that of CPUs over the last years, with respect to floating point operations per second (GFlops). This trend is not supposed to stop, therefore using GPUs for computation will be a useful model also in the near future.

Our Contribution. In this paper we present a parallel version of the enumeration algorithm of [SE91] that finds a shortest, non-zero vector in a lattice. Since the

enumeration algorithm is tree-based, the main challenge is splitting the tree in some way and executing subtree enumerations in parallel. We use the CUDA framework of NVIDIA for implementing the algorithm on graphics cards. Because of the choice for GPUs, parallelization and splitting are more difficult than for a CPU parallelization. Firstly we explain the ideas of how to parallelize enumeration on GPU. Secondly we present some first experimental results. Using the GPU, we reduce the time required for enumeration of a random lattice in dimensions higher than 50 by a factor of almost 5. We are using random lattices in the sense of Goldstein and Mayer [GM03] for testing our implementation.

The first part of this paper, namely the idea of parallelizing enumeration, can also be applied on multicore CPU. The idea of splitting the search tree into parts and search different subtrees independently in parallel is also applicable on CPU, or other parallel computing frameworks. As mentioned above, BKZ is only practical using blocksizes up to 50. As our GPU version of the enumeration performs best in dimensions n greater than 50, we would expect to speed up BKZ with high blocksizes only.

In contrast to our algorithm, Pujol's idea [Puj08] is to predict the number of enumeration steps in a subtree beforehand, using a volume heuristic. If the number of expected steps in a subtree exceeds some bound, the subtree is split recursively, and enumerated as different threads. Dagdelen [Dag09] bounds the height of subtrees that can be split recursively. Both ideas differ from our approach, as we use a real-time scheduling; when a subtree enumeration has exceeded a specified number of enumeration steps it is stopped, to balance the load of all GPU kernels. This fits best into the SIMD structure of GPUs, as both existing approaches lead to a huge number of diverging subthreads.

Structure of the Paper. In Section 2 we introduce the necessary preliminaries on lattices and GPUs. We discuss previous lattice reduction algorithms and the applications of lattices in cryptography. The GPU (CUDA) programming model is shortly introduced, explaining in more detail the memory model and data types which are important for our implementation. Section 3 explains our parallel enumeration algorithm, starting from the ENUM algorithm of Schnorr and Euchner and ending with the iterated GPU enumeration algorithm. Section 4 discusses the results obtained with our algorithm.

2 Preliminaries

A lattice is a discrete subgroup of $\mathbb{R}^d$. It can be represented by a basis matrix $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ $(n \leq d)$. We call $\mathcal{L}(\mathbf{B}) = \{\sum_{i=1}^{n} x_i \mathbf{b}_i, x_i \in \mathbb{Z}\}$ the lattice spanned by the column basis vectors $\mathbf{b}_i \in \mathbb{R}^d$ $(i = 1 \ldots n)$. The dimension n of a lattice is the number of linear independent vectors in the lattice, i.e. the number of basis vectors. When $n = d$ the lattice is called full dimensional.

The basis of a lattice is not unique. Every unimodular transformation $\mathbf{M}$, i.e. integer transformation with $\det \mathbf{M} = \pm 1$, turns a basis matrix $\mathbf{B}$ into a second basis $\mathbf{MB}$ of the same lattice.

The determinant of a lattice is defined as $\det(\mathcal{L}(\mathbf{B})) = \sqrt{\det(\mathbf{B}^T\mathbf{B})}$. For full dimensional lattices we have $\det(\mathcal{L}(\mathbf{B})) = |\det(\mathbf{B})|$. The determinant of a lattice is invariant of the choice of the lattice basis, which follows from the multiplicative property of the determinant and the fact that basis transformations have determinant ± 1.

The length of a shortest vector of a lattice $\mathcal{L}(\mathbf{B})$ is denoted $\lambda_1(\mathcal{L}(\mathbf{B}))$ or in short λ_1 if the lattice is uniquely determined.

The Gram-Schmidt algorithm computes an orthogonalization of a basis. It is an efficient algorithm that outputs $\mathbf{B}^* = [\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*]$ with $\mathbf{b}_i^*$ orthogonal and $\mu_{i,j}$ such that $\mathbf{B} = \mathbf{B}^* \cdot [\mu_{i,j}]$, where $[\mu_{i,j}]$ is an upper triangular matrix consisting of the Gram-Schmidt coefficients $\mu_{i,j}$ for $1 \leq j \leq i \leq n$. The orthogonalized matrix $\mathbf{B}^*$ is not necessarily a basis of the lattice.

2.1 Lattice Basis Reduction

Problems. Some lattice bases are more useful than others. The goal of lattice basis reduction (or in short lattice reduction) is to find a basis consisting of short and almost orthogonal lattice vectors. More exactly, we can define some (hard) problems on lattices. The most important one is the *shortest vector problem* (SVP), which consists of finding a vector $\mathbf{v} \in \mathcal{L} \setminus \{\mathbf{0}\}$ with $\|\mathbf{v}\| = \lambda_1(\mathcal{L}(\mathbf{B}))$. In most cases, the Euclidean norm $\|\cdot\|_2$ is considered. As the SVP is $\mathcal{NP}$-hard (at least under randomized reductions) [Din02, Kho05, RR06] people consider the approximate version γ-SVP, that tries to find a vector $\mathbf{v} \in \mathcal{L} \setminus \{\mathbf{0}\}$ with $\|\mathbf{v}\| \leq \gamma \cdot \lambda_1(\mathcal{L}(\mathbf{B}))$.

Other important problems like the *closest vector problem* (CVP) that searches for a nearest lattice vector to a given point in space, its approximation variant γ-CVP, or the *shortest basis problem* (SBP) are listed and described in detail in [MG02].

Algorithms. In 1982 Lenstra, Lenstra, and Lovász [LLL82] introduced the LLL algorithm, which was the first polynomial time algorithm to solve the approximate shortest vector problem in higher dimensions. Another algorithm is the BKZ block algorithm of Schnorr and Euchner [SE91]. In practice, this is the algorithm that gives the best solution to lattice reduction so far. Their paper [SE91] also introduces the enumeration algorithm (ENUM), a variant of the Fincke-Pohst [FP83] and Kannan [Kan83] algorithms. The ENUM algorithm is the fastest algorithm in practice to solve the exact shortest vector problem using complete enumeration of all lattice vectors in a suitable search space. It is used as a black box in the BKZ algorithm. The enumeration algorithm organizes linear combinations of the basis vectors in a search tree and performs a depth first search above the tree.

In [PS08] Pujol and Stehlé analyze the stability of the enumeration when using floating point arithmetic. In [HS07], improved complexity bounds for Kannan's algorithm are presented. This paper also suggests some better preprocessing of lattice bases, i.e., the authors suggest to BKZ reduce a basis before running enumeration. This approach lowers the runtime of enumeration. In this paper we consider both LLL and BKZ pre-reduced bases. [AKS01] show how to solve SVP using a randomized algorithm in time $2^{\mathcal{O}(n)}$, but their algorithm requires

exponential space and is therefore impractical. The papers [NV08] and [MV10] present improved sieving variants, where the Gauss-sieving algorithm of [MV10] is shown to be really competitive to enumeration algorithms in practically interesting dimensions.

Several LLL variants were presented by Schnorr [Sch03], Nguyen and Stehlé [NS05], and Gama and Nguyen [GN08a]. The variant of [NS05] is implemented in the `fpLLL` library of [CPS], which is also the fastest public implementation of ENUM algorithms. Koy introduced the notion of a primal-dual reduction in [Koy04]. Schnorr [Sch03] and Ludwig [BL06] deal with random sampling reduction. Both are slightly different concepts of lattice reduction, where primal-dual reduction uses the dual of a lattice for reducing and random sampling combines LLL-like algorithms with an exhaustive point search in a set of lattice vectors that is likely to contain short vectors.

The papers [SE91, SH95] present a probabilistic improvement of ENUM, called tree pruning. The idea is to prune subtrees that are unlikely to contain shorter vectors. As it leads to a probabilistic variant of the enumeration algorithm, we do not consider pruning techniques here.

In [GN08b] Gama and Nguyen compare the `NTL` implementation [Sho] of floating point LLL, the deep insertion variant of LLL and the BKZ algorithm. It is the first comprehensive comparison of lattice basis reduction algorithms and helps understanding their practical behavior.

In [Vil92, HT93, RV92] the authors present parallel versions for n and n^2 processors, where n is the lattice dimension. In [Jou93] the parallel LLL of Villard [Vil92] is combined with the floating point ideas of [SE91]. In [Wet98] the authors present a blockwise generalization of Villards algorithm. Backes and Wetzel worked out a parallel variant of the LLL algorithm for multi-core CPU architectures [BW09]. For the parallelization of lattice reduction on GPU the authors are not aware of any previous work.

Applications. Lattice reduction has applications in cryptography as well as in cryptanalysis. The foundation of some cryptographic primitives is based on the hardness of lattice problems. Lattice reduction helps determining the practical hardness of those problems and is a basis for real world application of those hash functions, signatures, and encryption schemes. Well known examples are the SWIFFT hash functions of Lyubashevsky et al. [LMPR08], the signature schemes of [LM08, GPV08, Lyu09, Pei09a], or the encryption schemes of [AD97, Pei09b, SSTX09]. The NTRU [HPS98, otCC09] and GGH [GGH97] schemes do not provide a security proof, but the best attacks are also lattice based.

There are also attacks on RSA and similar systems, using lattice reduction to find small roots of polynomials [CNS99, DN00, May10]. Low density knapsack cryptosystems were successfully attacked with lattice reduction [LO85]. Other applications of lattice basis reduction are factoring numbers and computing discrete logarithms using diophantine approximations [Sch91]. In Operations Research, or generally speaking, discrete optimization, lattice reduction can be used to solve linear integer programs [Len83].

2.2 Programming Graphics Cards

A Graphical Processing Units (GPUs) is a piece of hardware that is specifically designed to perform a massive number of specific graphical operations in parallel. The introduction of platforms like CUDA by NVIDIA [Nvi07a] or CTM by ATI [AMD06], that make it easier to run custom programs instead of limited graphical operations on a GPU, has been the major breakthrough for the GPU as a general computing platform. The introduction of integer and bit arithmetic also broadened the scope to cryptographic applications.

Applications. Many general mathematical packages are available for GPU, like the BLAS library [NVI07b] that supports basic linear algebra operations.

An obvious application in the area of cryptography is brute force searching using multiple parallel threads on the GPU. There are also implementations of AES [CIKL05, Man07, HW07] and RSA [MPS07, SG08, Fle07] available. GPU implementations can also be used (partially) in cryptanalysis. In 2008, Bernstein et al. use parallelization techniques on graphics cards to solve integer factorization using elliptic curves [BCC$^+$09]. Using NVIDIA's CUDA parallelization framework, they gained a speed-up of up to 6 compared to computation on a four core CPU. However, to date, no applications based on lattices are available for GPU.

Programming Model. For the work in this paper the CUDA platform will be used. The GPUs from the Tesla range, which support CUDA, are composed of several multiprocessors, each containing a small number of scalar processors. For the programmer this underlying hardware model is hidden by the concept of SIMT-programming: Single Instruction, Multiple Thread. The basic idea is that the code for a single thread is written, which is then uploaded to the device and executed in parallel by multiple threads.

The threads are organized in multidimensional arrays, called blocks. All blocks are again put in a multidimensional array, called the *grid*. When executing a program (a grid), threads are scheduled in groups of 32 threads, called *warps*. Within a warp threads should not diverge, as otherwise the execution of the warp is serialized.

Memory Model. The Tesla GPUs provide multiple levels of memory: registers, shared memory, global memory, texture and constant memory. Registers and shared memory are on chip and close to the multiprocessor and can be accessed with low latency. The number of registers and shared memory is limited, since the number available for one multiprocessor must be shared among all threads in a single block.

Global memory is off-chip and is not cached. As such, access to global memory can slow down the computations drastically, so several strategies for speeding up memory access should be considered (besides the general strategy of avoiding global memory access). By coalescing memory access, e.g. loading the same memory address or a consecutive block of memory from multiple threads, the delay is reduced, since a coalesced memory access has the same cost as a single random memory access. By launching a large number of blocks the latency

introduced by memory loading can also be hidden, since other blocks can be scheduled in the meantime.

The constant and texture memory are cached and can be used for specific types of data or special access patterns.

Instruction Set. Modern GPUs provide the full range of (32 and) 64 bit floating point, integer and bit operations. Addition and multiplication are fast, other operations can, depending on the type, be much slower. There is no point in using other than 32 or 64 bit numbers, since smaller types are always cast to larger types. Most GPUs have a specialized FMAD instruction, which performs a floating point multiplication followed by an addition at the cost of only a single operation. This instruction can be used during the BKZ enumeration.

One problem that occurs on GPUs is the fact that today GPUs are not able to deal with higher precision than 64 bit floating point numbers. For lattice reduction, sometimes higher bit sizes are required to guarantee the correct termination of the algorithms. For an n-dimensional lattice, using the floating point LLL algorithm of [LLL82], one requires a precision of $\mathcal{O}(n \log B)$ bits, where B is an upper bound for the length of the d-dimensional vectors [NS05]. For the L^2 algorithm of [NS05], the required bit size is $\mathcal{O}(n \log_2 3)$, which is independent of the norm of the input basis vectors. For more details on the floating point LLL analysis see [NS05] and [NS06].

In [PS08] the authors state that for enumeration algorithms double precision is suitable up to dimension 90, which is beyond the dimensions that are practical today. Therefore enumeration should be possible on actual graphics cards, whereas the implementation of LLL-like algorithms will be more complicated and require some multi-precision framework.

3 Parallel Enumeration on GPU

In this section we present our parallel algorithm for shortest vector enumeration in lattices. In Subsection 3.1 we briefly explain the ENUM algorithm of Schnorr and Euchner [SE91], which was used as a basis for our algorithm. Next, we present the basic idea for multi-thread enumeration in Subsection 3.2. Finally, in Subsection 3.3, we explain our parallel algorithm in detail.

The ENUM algorithm of Schnorr-Euchner is an improvement of the algorithms from [Kan83] and [FP83]. The ENUM algorithm is the fastest one today and also the one used in the `NTL` [Sho] and `fpLLL` [CPS] libraries. Therefore we have chosen this algorithm as basis for our parallel algorithm.

3.1 Original ENUM Algorithm

The ENUM algorithm enumerates over all linear combinations $[x_1, \ldots, x_n] \in \mathbb{Z}^n$ that generate a vector $\mathbf{v} = \sum_{i=1}^{n} x_i \mathbf{b}_i$ in the search space (i.e., all vectors $\mathbf{v}$ with $\|\mathbf{v}\|$ smaller than a specified bound). Those linear combinations are organized in a tree structure. Leafs of the tree contain full linear combinations, whereas inner nodes contain partly filled vectors. The search for the tree leaf that determines

the shortest lattice vector is performed in a depth first search order. The most important part of the enumeration is cutting off parts of the tree, i.e. the strategy which subtrees are explored and which ones cannot lead to a shorter vector.

Let i be the current level in the tree, $i = 1$ being at the bottom and $i = n$ at the top of the tree (c.f. Figure 1). Each step in the enumeration algorithm consists of computing an *intermediate* squared norm l_i, moving one level up or down the tree (to level $i' \in \{i-1, i+1\}$) and determining a new value for the coordinate $x_{i'}$.

Let $r_i = \|\mathbf{b}_i^*\|^2$. We define $l_i = l_{i+1} + y_i^2 r_i$ with $y_i = x_i - c_i$ and $c_i = -\sum_{j=i+1}^{n} \mu_{j,i} x_j$. So, for a certain choice of coordinates $x_i \dots x_n$ it holds that $l_k \geq l_i$ (with $k < i$) for all coordinate vectors $\mathbf{x}$ that end with the same coordinates $x_i \dots x_n$. This implies that the intermediate norm l_i can be used to cut off infeasible subtrees. If $l_i > A$, with A the squared norm of the shortest vector that has been found so far, the algorithm will increase i and move up inside the tree. Otherwise, the algorithm will lower i and move down in the tree. Usually, as initial bound A for the length of the shortest vector, one uses the norm of the first basis vector.

The next value for $x_{i'}$ is selected in an interval of length $\sqrt{\frac{A - l_{i'+1}}{r_{i'}}}$ centered at $c_{i'}$. The interval is enumerated according to the zig-zag pattern described in [SE91]. Starting from a central value $\lfloor c_{i'} \rceil$, ENUM will generate a sequence $\lfloor c_{i'} \rceil + 1, \lfloor c_{i'} \rceil - 1, \lfloor c_{i'} \rceil + 2, \lfloor c_{i'} \rceil - 2, \dots$ for the coordinate $x_{i'}$. To be able to generate such a pattern, helper vectors $\mathbf{\Delta x} \in \mathbb{Z}^n$ are used. We do not require to store $\mathbf{\Delta^2 x}$ as in the orginal algorithm [SE91, PS08], as the computation of the zigzag pattern is done in a slightly different way as in the original algorithm. For a more detailed description of the ENUM algorithm we refer to [PS08].

3.2 Multi-thread Enumeration

Roughly speaking, the parallel enumeration works as follows. The search tree of combinations that is explored in the enumeration algorithm can be split at a high level, distributing subtrees among several threads. Each thread then runs an enumeration algorithm, keeping the first coefficients fixed. These fixed coefficients are called *start vectors*. The subtree enumerations can run independently, which limits communication between threads. The top level enumeration is performed on CPU and outputs start vectors for the GPU threads.

When the number of postponed subtrees is higher than the number of threads that we can start in parallel, then we copy the start vectors to the GPU and let it enumerate the subtrees. After all threads have finished enumerating their subtrees we proceed in the same manner: caching start vectors on CPU and starting a batch of subtree enumerations on GPU. Figure 1 illustrates this approach. The variable α defines the region where the initial enumeration is performed. The subtrees where GPU threads work are also depicted in Figure 1.

If a GPU subtree enumeration finds a new optimal vector, it writes back the coordinates $\mathbf{x}$ and the squared norm A of this vector to the main memory. The other GPU threads will directly receive the new value for A, which will allow them to cut away more parts of the subtree.

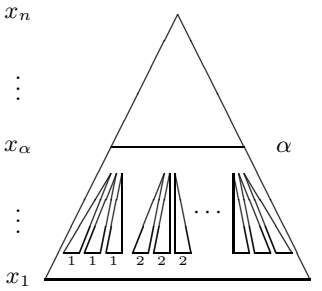

Fig. 1. Illustration of the algorithm flow. The top part is enumerated on CPU, the lower subtrees are explored in parallel on GPU. The tiny numbers illustrate which subtrees are enumerated in the same iteration.

Early Termination. The computation power of the GPU is used best when as many threads as possible are working at the same time. Recall that the GPU uses warps as the basic execution units: all threads in a warp are running the same instructions (or some of the threads in the warp are stalled in the case of branching).

In general, more starting vectors than there are GPU threads are uploaded in each run of the GPU kernel. This allows us to do some load balancing on the GPU, to make sure all threads are busy. To avoid the GPU being stalled by a few long running subtree enumerations, the GPU stops when just a few subtrees are left. We call this process, by which the GPU stops some subtrees even though they are not finished, *early termination*.

At the end of Section 3.3 details are included on the exact way early termination and our load balancing algorithm works. For now it suffices to know that, because of early termination, some of the subtree enumerations are not finished after a single launch of the GPU kernel. This is the main reason why the entire algorithm is iterated several times. At each iteration the GPU launches a mix of enumerations: new subtrees (start vectors) from the top enumeration and subtrees that were not finished in one of the previous GPU launches.

3.3 The Iterated Parallel ENUM Algorithm

Algorithm 1 shows the high-level layout of the GPU enumeration algorithm. Details concerning the updating of the bound A, as well as the write-back of newly discovered optimal vectors have been omitted. The actual enumeration is also not shown: it is part of several subroutines which are called from the main algorithm.

The whole process of launching a grid of GPU threads is iterated several times (line 2), until the whole search tree has been enumerated either on GPU or CPU.

In line 3, the top of the search tree is enumerated, to generate a set S of starting vectors $\mathbf{x}_k$ for which enumeration should be started at level α. More detailed, the top enumeration in the region between α and n outputs distinct vectors

$$\mathbf{x}_k = [0, \ldots, 0, x_{k,\alpha}, \ldots, x_{k,n}] \qquad \text{for } k = 1 \ldots \text{NUMSTARTPOINTS} - \#T\,.$$

Algorithm 1. High-level Iterated Parallel ENUM Algorithm

Input: $\mathbf{b}_i (i = 1, \ldots, n)$, A, α, n

```
1  Compute the Gram-Schmidt orthogonalization of b_i
2  while true do
3  |  S = {(x_k, Δx_k, L_k = α, s_k = 0)}_k ← Top enum: generate at most
   |  NUMSTARTPOINTS − #T vectors
4  |  R = {(x̄_k, Δx_k, L_k, s_k)}_k ← GPU enumeration, starting from S ∪ T
5  |  T ← {R_k : subtree k was not finished}
6  |  if #T < CPUTHRESHOLD then
7  |  |  Enumerate the starting points in T on the CPU.
8  |  |  Stop
9  |  end
10 end
```

Output: $(x_1, \ldots, x_n)$ with $\left\|\sum_{i=1}^{n} x_i \mathbf{b}_i\right\| = \lambda_1(\mathcal{L})$

The top enumeration will stop automatically if a sufficient number of vectors from the top of the tree have been enumerated. The rest of the top of the tree is enumerated in the following iterations of the algorithm.

Line 4 performs the actual GPU enumeration. In each iteration, a set of starting vectors and starting levels $\{\mathbf{x}_k, L_k\}$ is uploaded to the GPU. These starting vectors can be either vectors generated by the top enumeration in the region between α and n (in which case $L_k = \alpha$) or the vectors (and levels) written back by the GPU because of early termination, so that the enumeration will continue. In total NUMSTARTPOINTS vectors (a mix of new and old vectors) are uploaded at each iteration. For each starting vector $\mathbf{x}_k$ (with associated starting level L_k) the GPU outputs a vector

$$\bar{\mathbf{x}}_k = [\bar{x}_{k,1}, \ldots, \bar{x}_{k,\alpha-1}, x_{k,\alpha}, \ldots, x_{k,n}] \qquad \text{for } k = 1 \ldots \text{NUMSTARTPOINTS}$$

(which describes the current position in the search tree), the current level L_k, the number of enumeration steps s_k performed and also part of the internal state of the enumeration. This state $\{\bar{\mathbf{x}}_k, \mathbf{\Delta x}_k, L_k\}$ can be used to continue the enumeration later on. The vectors $\mathbf{\Delta x}_k$ are used in the enumeration to generate the zig-zag pattern and are part of the internal state of the enumeration [SE91]. This state is added to the output to be able to efficiently restart the enumeration at the point it was terminated.

Line 5 will select the resulting vectors from the GPU enumeration that were terminated early. These will be added to the set T of *leftover* vectors, which will be relaunched in the next iteration of the algorithm. If the set of leftover vectors is too small to get an efficient GPU enumeration, the CPU takes over and finishes off the last part of the enumeration. This final part only takes limited time.

GPU Threads and Load Balancing. In Section 3.2 the need for a load balancing algorithm was introduced: all threads should remain active and to ensure this, each thread in the same warp should run the same instruction. One of the

problems in achieving this, is the length difference of each subtree enumeration. Some very long subtree enumeration can cause all the other threads in the warp to become idle after they finish their subtree enumeration.

Therefore the number of enumeration steps that each thread can perform on a subtree is limited by $\mathbf{M}$. When $\mathbf{M}$ is exceeded, a subtree enumeration is forced to stop. After this, all threads in the same warp will reinitialise: they will either continue the previous subtree enumeration (that was terminated by reaching $\mathbf{M}$) or they will pick a new starting vector of the list $S \cup T$ delivered by the CPU. Then the enumeration starts again, limited to $\mathbf{M}$ enumeration steps.

In our experiments, NUMSTARTPOINTS was around 20-30 times higher than NUMTHREADS, which means that on average every GPU thread enumerated 20-30 subtrees in each iteration. $\mathbf{M}$ was chosen to be around 50-200.

4 Experimental Results

In this section we present some results of the CUDA implementation of our algorithm. For comparison we used the highly optimized ENUM algorithm of the `fpLLL` library in version 3.0.11 from [CPS]. `NTL` does not allow to run ENUM as a standalone SVP solver, but [Puj08] and the ENUM timings of [GN08b] show that `fpLLL`'s ENUM runs faster than `NTL`'s (the bit size of the lattice bases used in [GN08b] is higher than what we used, therefore a comparison with those timings is to be drawn carefully).

The CUDA program was compiled using `nvcc`, for the CPU programs we used `g++` with compiler flag `-O2`. The tests were run on an Intel Core2 Extreme CPU X9650 (using one single core) running at 3 GHz, and an NVIDIA GTX 280 graphics card. We run up to 100000 threads in parallel on the GPU. The code of our program can be found online.[1]

We chose random lattices following the construction principle of [GM03] with bit size of the entries of $10 \cdot n$. This type of lattices was also used in [GN08b] and [NS06]. We start with the basis in Hermite normal form and LLL-reduce them with $\delta = 0.99$. At the end of this section, we present some timings using BKZ-20 reduced bases, to show the capabilities of stronger pre-reduction.

Both algorithms, the enum of `fpLLL` (run with parameter *-a svp*) and our CUDA version, always output the same coefficient vectors and therefore a lattice vector with shortest possible length. We compare now the throughput of GPU and CPU concerning enumerations steps. Section 3.1 gives the explanation what is computed in each enumeration step. On the GPU, up to 200 million enumeration steps per second can be computed, while similar experiments on CPU only yielded 25 million steps per second. We choose $\alpha = n - 11$ for our experiments, this shapes up to be a good choice in practice. Table 1 and Figure 2 illustrate the experimental results. The figure shows the runtimes of both algorithms when applied to five different lattices of each dimension. One can notice that in dimension above 44, our CUDA implementation always outperforms the `fpLLL` implementation.

[1] `http://homes.esat.kuleuven.be/~jhermans/gpuenum/index.html`

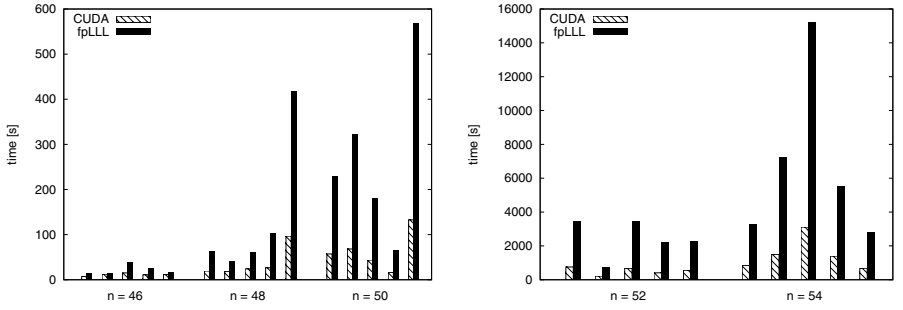

Fig. 2. Timings for enumeration. The graph shows the time needed for enumerating five different random lattices in each dimension n. It compares the ENUM algorithm of the `fpLLL`-library with our parallel CUDA version.

Table 1. Average time needed for enumeration of LLL pre-reduced lattices in each dimension n. The table presents the percentage of time that the GPU version takes compared to the `fpLLL` version.

n	40	42	44	46	48	50	52	54	56	60
`fpLLL` - ENUM	0.96s	2.41s	17.7s	22.0s	136s	273s	2434s	6821s	137489s	-
CUDA - ENUM	2.75s	4.29s	11.7s	11.4s	37.0s	63.5s	520s	1504s	30752s	274268s
	286%	178%	66%	52%	27%	23%	21%	22%	22%	-

Table 1 shows the average value over all five lattices in each dimension. Again one notices that the GPU algorithm demonstrates its strength in dimensions above 44, where the time goes down to 22% in dimensions 54 and 56 and down to 21% in dimension 52. Therefore we state that the GPU algorithm gains big speedups in dimensions higher than 45, which are the interesting ones in practice. In dimension 60, `fpLLL` did not finish the experiments in time, therefore only the average time of the CUDA version is presented in the table.

Table 2 presents the timing of the same bases, pre-reduced using BKZ algorithm with blocksize 20. The time of the BKZ-20 reduction is not included in the timings shown in the table. For dimension 64 we changed α (the subtree dimension) from the usual $n - 11$ to $\alpha = n - 14$, as this leads to lower timings in high dimensions. First, one can notice that both algorithms run much faster

Table 2. Average time needed for enumeration of BKZ-20 pre-reduced lattices in each dimension n. The time for pre-reduction is omitted in both cases.

n	48	50	52	54	56	58	60	62	64
`fpLLL` - ENUM	2.96s	7.30s	36.5s	79.2s	190s	601s	1293s	7395s	15069s
CUDA - ENUM	3.88s	5.42s	16.9s	27.3s	56.8s	119s	336s	986s	4884s
	131%	74%	46%	34%	30%	20%	26%	13%	32%

when using stronger pre-processing, a fact that was already mentioned in [HS07]. Second, we see that the speedup of the GPU version goes down to 13% in the best case (dimension 62).

As pruning would speed up both the serial and the parallel enumeration, we expect the same speedups with pruning.

It is hard to give an estimate of the achieved speedup compared to the number of threads used: since GPUs have hardware-based scheduling, it is not possible to know the number of active threads exactly. Other properties, like memory access and divergent warps, have a much greater influence on the performance and cannot be measured in thread counts or similar figures. When comparing only the number of double fmadds, the GTX 280 should be able to do 13 times more fmadd's than a single Core2 Extreme X9650.[2] Based on our results we fill only 30 to 40% of the GPUs ALUs. Using the CUDA Profiler, we determine that in our experiments around 12% of branches was divergent, which implies a loss of parallelism and also some ALUs being left idle. There is also a high number of warp serializations due to conflicting shared and constant memory access. The ratio warp serializations/instructions is around 35%.

To compare CPUs and GPUs, we can have a look at the cost of both platforms in dollardays, similar to the comparison in [BCC+09]. We assume a cost of around $2200 for our CPU (quad core) + 2x GTX295 setup. For a CPU-only system, the cost is only around $900. Given a speedup of 5 for a GPU compared to a CPU, we get a total speedup of 24 (4 CPU cores + 4 GPUs) in the $2200 machines and only a speedup of 4 in the CPU-only machine, assuming we can use all cores. This gives $225 \cdot t$ dollardays for the CPU-only system and only $91 \cdot t$ dollardays for the CPU+GPU system, where t is the time. This shows that even in this model of expense, the GPU implementation gains an advantage of around 2.4.

5 Further Work

Further improvements are possible using multiple CPU cores. Our implementation only uses one CPU core for the top enumeration and the rest of the outer loop of the enumeration. During the subtree enumerations on the GPU, the main part of the algorithm, the CPU is not used. When the GPU starts a batch of subtree enumerations it would be possible to start threads on the CPU cores as well. We expect a speedup of two compared to our actual implementation using this idea.

It is possible to start enumeration using a shorter starting value than the first basis vectors norm. The Gaussian heuristic can be used to predict the norm of the shortest basis vector λ_1. This can lead to enormous speedups in the algorithm. We did not include this improvement into our algorithm so far to get comparable results to `fpLLL`.

[2] A GTX280 can do 30 double fmadds in a 1.3GHz cycle, a single Core2 core can do 2 double fmadds in every two 3GHz cycle, which gives us a speedup of 13 for the GTX280.

Acknowledgments

We thank the anonymous referees for their valuable comments. We thank Özgür Dagdelen for creating some of the initial ideas of parallelizing lattice enumeration and Benjamin Milde, Chen-Mou Cheng, and Bo-Yin Yang for the nice discussions and helpful ideas.

We would like to thank EcryptII[3] and CASED[4] for providing the funding for the visits during which this work was prepared. Part of the work was done during the authors' visit to Center for Information and Electronics Technologies, National Taiwan University.

References

[AD97] Ajtai, M., Dwork, C.: A public-key cryptosystem with worst-case/average-case equivalence. In: Proceedings of the Annual Symposium on the Theory of Computing — STOC 1997, pp. 284–293 (1997)

[AKS01] Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: Proceedings of the Annual Symposium on the Theory of Computing — STOC 2001, pp. 601–610. ACM Press, New York (2001)

[AMD06] Advanced Micro Devices. ATI CTM Guide. Technical report (2006)

[BCC+09] Bernstein, D.J., Chen, T.-R., Cheng, C.-M., Lange, T., Yang, B.-Y.: ECM on graphics cards. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 483–501. Springer, Heidelberg (2009)

[BL06] Buchmann, J., Ludwig, C.: Practical lattice basis sampling reduction. In: Hess, F., Pauli, S., Pohst, M. (eds.) ANTS 2006. LNCS, vol. 4076, pp. 222–237. Springer, Heidelberg (2006)

[BW09] Backes, W., Wetzel, S.: Parallel lattice basis reduction using a multi-threaded Schnorr-Euchner LLL algorithm. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009 Parallel Processing. LNCS, vol. 5704, pp. 960–973. Springer, Heidelberg (2009)

[CIKL05] Cook, D.L., Ioannidis, J., Keromytis, A.D., Luck, J.: Cryptographics: Secret key cryptography using graphics cards. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 334–350. Springer, Heidelberg (2005)

[CNS99] Coupé, C., Nguyen, P.Q., Stern, J.: The effectiveness of lattice attacks against low exponent RSA. In: Imai, H., Zheng, Y. (eds.) PKC 1999. LNCS, vol. 1560, pp. 204–218. Springer, Heidelberg (1999)

[CPS] Cadé, D., Pujol, X., Stehlé, D.: fpLLL - a floating point LLL implementation. Available at Damien Stehlé's homepage at école normale supérieure de Lyon, `http://perso.ens-lyon.fr/damien.stehle/english.html`

[Dag09] Dagdelen, Ö.: Parallelisierung von Gitterbasisreduktionen. Masters thesis, TU Darmstadt (2009)

[Din02] Dinur, I.: Approximating SVP_∞ to within almost-polynomial factors is NP-hard. Theoretical Computer Science 285(1), 55–71 (2002)

[3] `http://www.ecrypt.eu.org/`

[4] `http://www.cased.de`

[DN00] Durfee, G., Nguyen, P.Q.: Cryptanalysis of the RSA schemes with short secret exponent from Asiacrypt 1999. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 14–29. Springer, Heidelberg (2000)

[Fle07] Fleissner, S.: GPU-Accelerated Montgomery Exponentiation. In: Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2007. LNCS, vol. 4487, pp. 213–220. Springer, Heidelberg (2007)

[FP83] Fincke, U., Pohst, M.: A procedure for determining algebraic integers of given norm. In: van Hulzen, J.A. (ed.) ISSAC 1983 and EUROCAL 1983. LNCS, vol. 162, pp. 194–202. Springer, Heidelberg (1983)

[GGH97] Goldreich, O., Goldwasser, S., Halevi, S.: Public-key cryptosystems from lattice reduction problems. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 112–131. Springer, Heidelberg (1997)

[GM03] Goldstein, D., Mayer, A.: On the equidistribution of hecke points. Forum Mathematicum 2003 15(2), 165–189 (2003)

[GN08a] Gama, N., Nguyen, P.Q.: Finding short lattice vectors within Mordell's inequality. In: Proceedings of the Annual Symposium on the Theory of Computing — STOC 2008, pp. 207–216. ACM Press, New York (2008)

[GN08b] Gama, N., Nguyen, P.Q.: Predicting lattice reduction. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 31–51. Springer, Heidelberg (2008)

[GPV08] Gentry, C., Peikert, C., Vaikuntanathan, V.: Trapdoors for hard lattices and new cryptographic constructions. In: Proceedings of the Annual Symposium on the Theory of Computing — STOC 2008, pp. 197–206. ACM Press, New York (2008)

[HPS98] Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: Buhler, J.P. (ed.) ANTS 1998. LNCS, vol. 1423, pp. 267–288. Springer, Heidelberg (1998)

[HS07] Hanrot, G., Stehlé, D.: Improved analysis of kannan's shortest lattice vector algorithm. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 170–186. Springer, Heidelberg (2007)

[HT93] Heckler, C., Thiele, L.: A parallel lattice basis reduction for mesh-connected processor arrays and parallel complexity. In: IEEE Symposium on Parallel and Distributed Processing — SPDP, pp. 400–407. IEEE Computer Society Press, Los Alamitos (1993)

[HT98] Heckler, C., Thiele, L.: Complexity analysis of a parallel lattice basis reduction algorithm. SIAM J. Comput. 27(5), 1295–1302 (1998)

[HW07] Harrison, O., Waldron, J.: AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 209–226. Springer, Heidelberg (2007)

[Jou93] Joux, A.: A fast parallel lattice reduction algorithm. In: Proceedings of the Second Gauss Symposium, pp. 1–15 (1993)

[Kan83] Kannan, R.: Improved algorithms for integer programming and related lattice problems. In: Proceedings of the Annual Symposium on the Theory of Computing — STOC 1983, pp. 193–206. ACM Press, New York (1983)

[Kho05] Khot, S.: Hardness of approximating the shortest vector problem in lattices. J. ACM 52(5), 789–808 (2005)

[Koy04] Koy, H.: Primale-duale Segment-Reduktion (2004), `http://www.mi.informatik.uni-frankfurt.de/research/papers.html`

[Len83] Lenstra, H.W.: Integer programming with a fixed number of variables. Math. Oper. Res. 8, 538–548 (1983)

[LLL82] Lenstra, A., Lenstra, H., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen 261(4), 515–534 (1982)

[LM08] Lyubashevsky, V., Micciancio, D.: Asymptotically efficient lattice-based digital signatures. In: Canetti, R. (ed.) TCC 2008. LNCS, vol. 4948, pp. 37–54. Springer, Heidelberg (2008)

[LMPR08] Lyubashevsky, V., Micciancio, D., Peikert, C., Rosen, A.: Swifft: A modest proposal for fft hashing. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 54–72. Springer, Heidelberg (2008)

[LO85] Lagarias, J.C., Odlyzko, A.M.: Solving low-density subset sum problems. Journal of the ACM 32(1), 229–246 (1985)

[Lyu09] Lyubashevsky, V.: Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 598–616. Springer, Heidelberg (2009)

[Man07] Manavski, S.A.: Cuda Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In: IEEE International Conference on Signal Processing and Communications — ICSPC, pp. 65–68. IEEE Computer Society Press, Los Alamitos (2007)

[May10] May, A.: Using LLL-reduction for solving RSA and factorization problems. In: Nguyen, P.Q., Vallée, B. (eds.) The LLL algorithm, pp. 315–348. Springer, Heidelberg (2010)

[MG02] Micciancio, D., Goldwasser, S.: Complexity of Lattice Problems: a cryptographic perspective. The Kluwer International Series in Engineering and Computer Science, vol. 671. Kluwer Academic Publishers, Boston (2002)

[MPS07] Moss, A., Page, D., Smart, N.P.: Toward Acceleration of RSA Using 3D Graphics Hardware. In: Galbraith, S.D. (ed.) Cryptography and Coding 2007. LNCS, vol. 4887, pp. 364–383. Springer, Heidelberg (2007)

[MV10] Micciancio, D., Voulgaris, P.: Faster exponential time algorithms for the shortest vector problem. In: Proceedings of the Annual Symposium on Discrete Algorithms — SODA 2010 (2010)

[NS05] Nguyen, P.Q., Stehlé, D.: Floating-point LLL revisited. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 215–233. Springer, Heidelberg (2005)

[NS06] Nguyen, P.Q., Stehlé, D.: LLL on the average. In: Hess, F., Pauli, S., Pohst, M. (eds.) ANTS 2006. LNCS, vol. 4076, pp. 238–256. Springer, Heidelberg (2006)

[NV08] Nguyen, P.Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. J. of Mathematical Cryptology 2(2) (2008)

[Nvi07a] Nvidia. Compute Unified Device Architecture Programming Guide. Technical report (2007)

[NVI07b] NVIDIA. CUBLAS Library (2007)

[otCC09] 1363 Working Group of the C/MM Committee. IEEE P1363.1 Standard Specification for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices (2009), `http://grouper.ieee.org/groups/1363/`

[Pei09a] Peikert, C.: Bonsai trees (or, arboriculture in lattice-based cryptography). Cryptology ePrint Archive, Report 2009/359 (2009), `http://eprint.iacr.org/`

[Pei09b] Peikert, C.: Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In: Proceedings of the Annual Symposium on the Theory of Computing — STOC 2009, pp. 333–342 (2009)

[PS08] Pujol, X., Stehlé, D.: Rigorous and efficient short lattice vectors enumeration. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 390–405. Springer, Heidelberg (2008)

[Puj08] Pujol, X.: Recherche efficace de vecteur court dans un réseau euclidien. Masters thesis, ENS Lyon (2008)

[RR06] Regev, O., Rosen, R.: Lattice problems and norm embeddings. In: Proceedings of the Annual Symposium on the Theory of Computing — STOC 2006, pp. 447–456. ACM Press, New York (2006)

[RV92] Roch, J.-L., Villard, G.: Parallel gcd and lattice basis reduction. In: Bougé, L., Robert, Y., Trystram, D., Cosnard, M. (eds.) CONPAR 1992 and VAPP 1992. LNCS, vol. 634, pp. 557–564. Springer, Heidelberg (1992)

[Sch91] Schnorr, C.-P.: Factoring integers and computing discrete logarithms via diophantine approximations. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 281–293. Springer, Heidelberg (1991)

[Sch03] Schnorr, C.-P.: Lattice reduction by random sampling and birthday methods. In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 145–156. Springer, Heidelberg (2003)

[SE91] Schnorr, C.-P., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In: Budach, L. (ed.) FCT 1991. LNCS, vol. 529, pp. 68–85. Springer, Heidelberg (1991)

[SG08] Szerwinski, R., Guneysu, T.: Exploiting the Power of GPUs for Asymmetric Cryptography. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 79–99. Springer, Heidelberg (2008)

[SH95] Schnorr, C.-P., Hörner, H.H.: Attacking the Chor-Rivest cryptosystem by improved lattice reduction. In: Guillou, L.C., Quisquater, J.-J. (eds.) EUROCRYPT 1995. LNCS, vol. 921, pp. 1–12. Springer, Heidelberg (1995)

[Sho] Shoup, V.: Number theory library (NTL) for C++, `http://www.shoup.net/ntl/`

[SSTX09] Stehlé, D., Steinfeld, R., Tanaka, K., Xagawa, K.: Efficient public key encryption based on ideal lattices. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 617–635. Springer, Heidelberg (2009)

[Vil92] Villard, G.: Parallel lattice basis reduction. In: International Symposium on Symbolic and Algebraic Computation — ISSAC, pp. 269–277. ACM Press, New York (1992)

[Wet98] Wetzel, S.: An efficient parallel block-reduction algorithm. In: Buhler, J.P. (ed.) ANTS 1998. LNCS, vol. 1423, pp. 323–337. Springer, Heidelberg (1998)