Random Sampling for Short Lattice Vectors on Graphics Cards

Michael Schneider and Norman Göttert

Technische Universität Darmstadt, Germany
mischnei@cdc.informatik.tu-darmstadt.de

Abstract. We present a GPU implementation of the Simple Sampling Reduction (SSR) algorithm that searches for short vectors in lattices. SSR makes use of the famous BKZ algorithm. It complements an exhaustive search in a suitable search region to insert random, short vectors to the lattice basis. The sampling of short vectors can be executed in parallel.

Our GPU implementation increases the number of sampled vectors per second from 5200 to more than 120,000. With this we are the first to present a parallel implementation of SSR and we make use of the computing capability of modern graphics cards to enhance the search for short vectors even more.

Keywords: Lattice reduction, random sampling, SSR, BKZ.

1 Introduction

Lattices are discrete additive groups in the Euclidean vector space. They are known for hundreds of years in mathematics, but their use in cryptography and other fields of computer science started in the last decades of the twentieth century. Roughly speaking, lattice reduction is the search for short vectors with special geometric structure, i.e., vectors that are nearly orthogonal to each other. In 1982, the famous LLL algorithm was presented by Lenstra, Lenstra, and Lovász [16]. It set a starting point for developments and improvements of lattice reduction algorithms until today. In 1991, the BKZ algorithm (for Block-Korkine-Zolotarev reduction) which is a generalization of LLL was presented [27]. Today, BKZ is still the strongest and mostly used algorithm for lattice basis reduction. In 2003, the Random Sampling Reduction (RSR) algorithm was presented [26]. It is an adaption of BKZ, and applies BKZ together with the insertion of some randomly sampled vectors. In 2006, Simple Sampling Reduction (SSR) improved RSR by removing its heuristic assumptions [7].

In cryptology, lattice reduction has applications in cryptography as well as in cryptanalysis. The security of lattice based cryptosystems can be sustained by hard problems in lattices. The fact that makes lattice based cryptography special is the ability to base the security of cryptosystems on *worst case problems* in lattices, whereas usually security is only based on average case problems. This

B. Preneel and T. Takagi (Eds.): CHES 2011, LNCS 6917, pp. 160–175, 2011.

so-called *worst case to average case reduction* is unique for lattices and is not known in other fields.

For estimating the practical security of lattice based cryptosystems, it is necessary to know the strength of lattice reduction algorithms such as LLL, BKZ, and their revisions. Since there is a well-known gap between practical and theoretical strength of these algorithms, it is important to assess their practical borders. Since today, even desktop computers and laptops are equipped with multicore CPUs or graphics cards that support the CPU, this kind of special hardware must be taken into account when talking about security of cryptosystems. Due to the fact that supercomputers and new paradigms such as cloud computing gain more and more importance, the computing capabilities of attackers of cryptosystems rises as well. Therefore it is necessary to examine the strength of lattice reduction algorithms concerning parallelization potential.

The BKZ algorithm is the lattice reduction algorithm most commonly used in practice. It consists of two building blocks. One part is the LLL algorithm, the other part is an enumeration subroutine that performs exhaustive search for shortest vectors. No parallel version of BKZ is known to date. There are approaches of parallelizing LLL in the SIMD model, e.g. [30,2] and also for enumeration [9,15,10]. The combination of both however has not yet been tried.

It is apparent that SSR allows for distributed computing, since sampling short vectors can be performed independently in parallel. The authors of [7] state that most time of SSR is spent on sampling, which would allow for good parallelization.

1.1 Previous Results

Schnorr presented the first sampling algorithm called Random Sampling Reduction (RSR) in [26]. Ludwig and Buchmann refine the algorithm and promise to make sampling practical with their Simple Sampling Reduction (SSR) in [7]. They get rid of two RSR assumptions, namely the *Randomness Assumption* (RA) and the *Geometric Series Assumption* (GSA), which they claim both do not hold in practice. They replace the independent random sampling of vectors in the search space by a deterministic exhaustive search. This makes it impossible to sample the same vector multiple times, which was the case for RSR. Ludwig gives a more detailed view on SSR in [17]. The implementation of Ludwig is available upon request. Comparisons of his SSR implementation with BKZ on cryptographic lattices can be found, e.g., in [6,5].

1.2 Our Contribution

In this paper we present CUDA-SSR, a parallel variant of simple sampling reduction running on graphics cards using NVIDIAs CUDA framework. Our experiments are twofold. First we compare CUDA-SSR to BKZ, and second we compare it to our CPU-SSR implementation to show the strength of the GPU.

Although it is already mentioned in [7] that SSR is a good candidate for parallelizing, we are the first to present a distributed version of SSR. The authors of [7] mention a sampling rate of up to 5200 samples per second (on a 2.4GHz

Intel Pentium 4). On an NVIDIA GTX295 GPU (which was released in 2009) we get rates of more than 120,000 samples per second.

1.3 Organization of the Paper

The remainder of this paper is organized as follows. In Section 2, we present the required background knowledge concerning lattices, random sampling, and GPU computations. In Section 3, we develop a parallel version of SSR and explain how we implemented it on graphics cards. This is the main contribution of our work. Section 4 presents experimental results that show the strength of the GPU version of random sampling.

2 Preliminaries

Let $\|\mathbf{v}\|$ denote the Euclidean norm of the vector $\mathbf{v}$. Other norms are subscripted like $\|\mathbf{v}\|_\infty$.

Let $n, d \in \mathbb{N}$, $n \leq d$, and let $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^d$ be linearly independent. Then the set $\mathcal{L}(\mathbf{B}) = \{\sum_{i=1}^{n} x_i \mathbf{b}_i : x_i \in \mathbb{Z}\}$ is the lattice spanned by the basis column matrix $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{d \times n}$. The lattice $\mathcal{L}(\mathbf{B})$ is called n-dimensional. Its basis $\mathbf{B}$ is not unique, unimodular transformations lead to a different basis of the same lattice. The first successive minimum $\lambda_1(\mathcal{L}(\mathbf{B}))$ is the length of a shortest vector of $\mathcal{L}(\mathbf{B})$. The lattice determinant $\det(\mathcal{L}(\mathbf{B}))$ is defined as $\sqrt{\det(\mathbf{B}^t\mathbf{B})}$. It is invariant under basis changes. For full-dimensional lattices, where $n = d$, there is $\det(\mathcal{L}(\mathbf{B})) = |\det(\mathbf{B})|$ for every basis $\mathbf{B}$. In the remainder of this paper we will only be concerned with full-dimensional lattices.

Denote the Gram-Schmidt-orthogonalization (GSO) with $\mathbf{b}_i^* = \pi_i(\mathbf{b}_i)$ where $\pi_i(\mathbf{b}) \to \langle \mathbf{b}_1 \ldots \mathbf{b}_{i-1} \rangle^\perp$ is the orthogonal projection. The GSO is calculated via $\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^*$ for all $1 \leq i \leq n$, where $\mu_{i,j} = \mathbf{b}_i^T \mathbf{b}_j^* / \left\|\mathbf{b}_j^*\right\|^2$ for all $1 \leq j \leq i \leq n$. The values $\mu_{i,j}$ are called *Gram-Schmidt (GS) coefficients*.

Roughly speaking, lattice reduction is the process of transforming a basis of a lattice into a second one consisting of short vectors which are nearly orthogonal. The LLL [16] and BKZ [27] algorithms are the most common algorithms for lattice reduction. BKZ is controlled by a blocksize parameter β, which allows for a trade-off between runtime and reduction quality. Higher values of β lead to better reduced bases at the expense of an exponentially (in β) increasing runtime. LLL is the special case of BKZ with $\beta = 2$. Both LLL and BKZ sort the basis vectors in increasing order, so that $\mathbf{b}_1$ is the shortest among the basis vectors after reduction. Applied to a basis $\mathbf{B}$, LLL provably finds a vector $\mathbf{b}_1$ with $\|\mathbf{b}_1\| \leq 2^{(n-1)/2} \lambda_1(\mathcal{L}(\mathbf{B}))$. When LLL or BKZ is applied to a generator system of a lattice $\mathcal{L}$ it will output a basis of $\mathcal{L}$, so it will remove linear dependent vectors. The first basis vector found by BKZ with $\beta > 2$ is shorter than with LLL, i.e., it holds that $\|\mathbf{b}_1\| \leq (\gamma_\beta)^{(n-1)/(\beta-1)} \lambda_1(\mathcal{L}(\mathbf{B}))$ [25], where γ_β is the β-dimensional Hermite constant. A practical comparison of LLL and BKZ can be found in [12]. Both LLL and BKZ are equipped with a parameter δ, which only slightly controls the reduction quality and is usually set to 0.99. For further information concerning lattices and lattice reduction we refer to [19,20,22].

2.1 Random Sampling

The idea of random sampling was presented by Schnorr in 2003 [26]. It was adopted and improved in [17,7]. The idea of random sampling is the following. Iteratively, it switches between reduction of the basis (using BKZ) and sampling a random short vector of norm $< 0.99 \|\mathbf{b}_1\|^2$, which is then prepended to the reduced basis (cf. Algorithm 1).

Every basis vector $\mathbf{v} = [v_1, \ldots, v_n]$ can be written in its orthogonalized form $\mathbf{v} = \sum_{i=1}^{n} \nu_i \mathbf{b}_i^*$. We can write its squared norm as

$$\|\mathbf{v}\|^2 = \sum_{i=1}^{n} \nu_i^2 \|\mathbf{b}_i^*\|^2 \ . \tag{1}$$

Therefore, shortening a vector $\mathbf{v}$ is done either by decreasing ν_i or by decreasing the $\|\mathbf{b}_i^*\|$.

For a reduced basis $\mathbf{B}$ (either LLL or BKZ reduced), it is known that the norm of the orthogonalized vectors $\|\mathbf{b}_i\|$ decreases for increasing index i. This implies that for higher indices, the influence of the coefficient ν_i in Equation (1) is less noticeable than for smaller indices. This fact helps interpreting the following definition of a search space. For a basis $\mathbf{B} \in \mathbb{Z}^{n \times n}$ and an integer u with $1 \leq u \leq n$ we define the set $\mathcal{S}_{u,\mathbf{B}}$ as the set of all lattice vectors $\mathbf{v} = \sum_{i=1}^{n} \nu_i \mathbf{b}_i^*$ with

$$|\nu_i| \leq \begin{cases} 0.5 & \text{for } 1 \leq i < n-u \\ 1 & \text{for } n-u \leq i < n \end{cases} \quad , \quad \nu_n = 1 \tag{2}$$

and call it the search space. It is $\mathcal{S}_{u,\mathbf{B}} \subseteq \mathcal{L}(\mathbf{B})$, and this search space is supposed to contain short lattice vectors. The algorithm SAMPLE (Algorithm 2, original in [17]) uses as input a lattice basis $\mathbf{B}$ and an integer value x, and as output it computes a vector $\mathbf{v} \in \mathcal{S}_{u,\mathbf{B}}$ in the search space. The bit representation of the integer x controls the sampling deterministically. If the search space $\mathcal{S}_{u,\mathbf{B}}$ consists of 2^u many points, running SAMPLE with all values $x \in \{1, \ldots, 2^u\}$ guarantees that the complete search space is sampled.

Algorithm 1. SSR

Input: Lattice basis $\mathbf{B} \in \mathbb{Z}^{n \times n}$, GS-coefficients $\mathbf{R} \in \mathbb{Q}^{n \times n}$, bound $u_{max} \in \mathbb{N}$, blocksize β, norm bound A
Output: reduced basis $\mathbf{B}$ s.t. $\|\mathbf{b}_1\| < A$

```
1 B ← BKZ([b_1, ..., b_n], β)
2 while ||b_1|| > A do
3 |   for x = 1 to 2^{u_max} do
4 |   |   v ← SAMPLE(B, R, x)
5 |   |   if ||v||^2 ≤ 0.99 ||b_1||^2 then break
6 |   end
7 |   if x = 2^{u_max} then terminate("No short vector found")
8 |   B ← BKZ([v, b_1, ..., b_n], β)
9 end
```

Algorithm 2. SAMPLE

Input: Lattice basis $\mathbf{B} \in \mathbb{Z}^{n \times n}$, GS-coefficients $\mathbf{R} \in \mathbb{Q}^{n \times n}$, $x \in \mathbb{Z}$
Output: vector $\mathbf{v} \in \mathcal{S}_{u,\mathbf{B}}$

```
1  v ← b_n, ν ← r_n
2  for j = n − 1 to 1 do
3  |  y ← ⌈ν_j − 0.5⌋
4  |  if x = 1 mod 2 then
5  |  |  if ν_j − y ≤ 0 then y ← y − 1
6  |  |          else  y ← y + 1
7  |  end
8  |  x ← ⌊x/2⌋, v ← v − y b_j, ν ← ν − y r_j
9  end
10 return v
```

Algorithm 1 shows a pseudo-code listing of SSR, Algorithm 2 shows a listing of SAMPLE. For more details on random sampling we refer to the works of [26,17,7].

2.2 GPU Computation

Graphical Processing Units (GPUs) were developed to perform huge numbers of graphical operations in parallel. The introduction of computing platforms such as CUDA by NVIDIA [23] and CTM by ATI [1] opened graphics cards equipped with GPUs for running custom user programs. The development of these computing frameworks where the starting point of the breakthrough these processing units had over the last years. The existence of standard libraries like BLAS [24] for linear algebra made GPUs interesting for cryptographic applications as well.

In the field of cryptography, there are (among others) implementations of AES [8,18,14] and RSA [21,29,11] available as well as implementations of the SHA3 hash competition finalists [4]. In cryptanalysis, Bernstein et al. use parallelization techniques on graphics cards to solve integer factorization using elliptic curves [3]. Concerning lattices and lattice reduction, there is an implementation of the ENUM algorithm on graphics cards [15]. We are not aware of other work in the field of lattice reduction.

Programming Model. We will be using the CUDA framework from NVIDIA on an NVIDIA GTX 295 card. The description might be slightly different for newer cards of the Fermi architecture. A CUDA-capable GPU is equipped with several multiprocessors, which contain small numbers of scalar processors each. The programmer can stick to the *single instruction - multiple thread* (SIMT) programming model. The programmer writes code for single threads, which is uploaded to the device and executed in parallel by multiple threads.

The threads altogether are organized in *blocks*, which again are organized in *grids*. A kernel is a program running on a graphics device. When a kernel (a grid) is executed, 32 threads are scheduled in a so-called *warp*. These 32 threads should perform the same computation, since otherwise the threads are handled in serial, not in parallel.

Memory Model. One big issue on NVIDIAs GPUs is the different types of memory available. There are 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 slow, since it is off-chip and there is no cache for it. Constant and texture memory are parts of the global memory, but they are cached and can be used for specific types of data or special access patterns.

3 GPU Algorithm CUDA-SSR

The CUDA-SSR approach in Algorithm 3 is a slightly changed variant of the original SSR algorithm. In each outer while loop, up to $2^{u_{max}}$ vectors are sampled in parallel, and the m shortest samples are added to the basis. The main difference to the original SSR is the sampling of new vectors $\mathbf{v}$, which is done on the GPU and returns not only a single vector but multiple ones within a bound of m. The calculated vectors $[\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m]$ are added to the front of the lattice $\mathbf{B}$ in a sorted order, before the extended lattice is reduced by the BKZ algorithm. With the adding of multiple vectors we get a benefit of a more stabilized reduction, as we will see in the experiments section.

The algorithm terminates if a given norm of the first vector of $\mathbf{B}$ is undercut by a new vector $\mathbf{v}$ or if no smaller vector is found in the given search space.

The subroutine PAR-SAMPLE (which is now executed on GPU) is a slightly changed variant of SAMPLE (Algorithm 2). The original sample algorithm was

Algorithm 3. CUDA-SSR

Input: Lattice basis $\mathbf{B} \in \mathbb{Z}^{n \times n}$, GS-coefficients $\mathbf{R} \in \mathbb{Q}^{n \times n}$, bound $u_{max} \in \mathbb{N}$, blocksize β, norm bound A, add vector bound m
Output: BKZ-β reduced basis $\mathbf{B}$ s.t. $\|\mathbf{b}_1\| \leq A$

```
1  B ← BKZ([b_1, ..., b_n], β)
2  foundSmaller = true
3  xOffset = 0
4  while ||b_1|| > A and foundSmaller = true do
5      while xOffset < 2^{u_max} do
6          parallel [i = xOffset ... xOffset + maxSamplesPerCall] do
7              [v_1, v_2, ..., v_m], foundSmaller ← PAR-SAMPLE(B, R, x_i, m)
8          end
9          if foundSmaller = true then break inner while loop
10         xOffset += maxSamplesPerCall
11         if xOffset ≥ 2^{u_max} then terminate
12     end
13     B ← BKZ([v_1, v_2, ..., v_m, b_1, ..., b_n], β)
14 end
```

parallelized, so that it computes a huge number of vectors per call. The possibility of parallelization is based on the independence of the samples. The only difference among two samples is the input value x, which can be interpreted as an unique identifier or seed.

One sample is stored in the shared memory of a CUDA block. The amount of shared memory, which is used for producing one sample, consists of memory for the vector $\mathbf{v}$ ($4Byte \cdot dimension$), for the vector ν ($4Byte \cdot dimension$), for y ($4Byte$), and for a valid-Byte ($1Byte$). For one CUDA block a number of

$$samplesPerBlock = \left\lfloor \frac{available\ shared\ memory}{(4 + 4Byte) \cdot dimension + 4Byte + 1Byte} \right\rfloor$$

vectors are produced. If we use all available CUDA blocks, the overall number of samples is $65535 \cdot samplesPerBlock$ per call. For example, at a dimension of 80 one call calculates $65535 \cdot \lfloor \frac{16344}{8 \cdot 80 + 5} \rfloor = 1,638,375$ samples.[1]

3.1 Parallel Implementation of Subroutine Sample

Here we describe how we implemented the sampling of $samplesPerBlock$ many vectors in $\mathcal{S}_{u,\mathbf{B}}$ on GPU. This is the main contribution of the paper.

The first step for determining $samplesPerBlock$ samples in one CUDA block is to copy the entries of the last vector of the matrices $\mathbf{B}$ and $\mathbf{R}$ to $\mathbf{v}$ and ν in parallel. The matrices $\mathbf{B}$ and $\mathbf{R}$ resist in the texture memory, because they are read multiple times and this memory is cached.

The second step is to compute the factor y for every sample and build new vectors inside a for-loop. A single y is processed by one CUDA thread, therefore all $y's$ of one CUDA block can be calculated in parallel. Afterwards the temporary new vectors $\mathbf{v}$ and ν are built, whereby all entries of a vector are assigned in one parallel step. If an integer overflow is noticed in this step, the sample will be indicated as invalid.

When the loop is finished, the square norms of the new samples are calculated with the common *vector reduction* approach, after squaring all entries of $\mathbf{v}$ in parallel. Figure 1 illustrates this procedure. Once a square norm of 2^x (with $x = \max\{y \in \mathbb{Z} : 2^y \leq dimension\}$) is determined, the result will be added to the first entry of the next interval. This procedure continues, until there is no more than one entry left.

Because the square norms of all vectors are calculated step by step, we can register the smallest square norm of a CUDA block. Therefore a CUDA block writes only the smallest vector back to the global memory, assumed that the square norm is less than 99% of $\|\mathbf{b}_1\|^2$ and the sample is valid. With this we save a lot of global memory. Instead of writing $65535 \cdot samplesPerBlock$ many vectors to global memory we use shared memory for $samplesPerBlock$ many vectors of each block and only write 65535 many vectors to the device.

[1] The shared memory of $16384Byte$ is decreased by the parameters of the kernel call, which are also stored in shared memory ($16Byte$ for dimBlock and dimGrid, $24Byte$ for 3 pointers). These values might change for other CUDA compute capabilities.

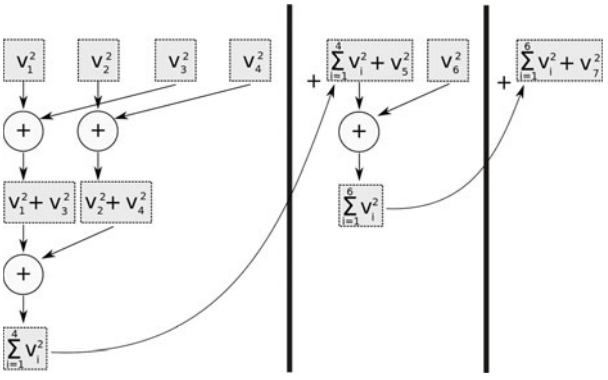


Fig. 1. Computation of the norm of a single vector $\mathbf{v}$ in parallel

For achieving higher performance we introduce a counter, which increases if a vector with a square norm less than 99% of $\|\mathbf{b}_1\|^2$ is found. When m vectors below this bound have been found, we break the parallel sampling. The counter is increased with so called *atomic operations*, which provides an exclusive *read-modify-write* operation for one CUDA thread. The parallel processing of the CUDA framework is only "semi-parallel", because only a part of all CUDA blocks are processed parallel for real (we have 65535 blocks but only 30 multiprocessors available). Therefore we can abort further calculations, if the counter m reached a defined value. A flow chart of our GPU algorithm of SAMPLE is shown in Appendix A. In order to remove serialization we also tested replacing the condition in Line 4 of SAMPLE by arithmetic computations, but recognized no speedups. Since there is no *else*-block, the fact that (on average) half of the threads are idle does not influence the total runtime.

For establishing the gain of parallel sampling we also implement a CPU version of the SSR algorithm (called CPU-SSR), which produces new vectors step by step. Our CPU as well as the GPU implementation are available online.[2]

4 Experimental Results

We are using an NVIDIA GTX 295 GPU for our experiments. The CPU that we use is an Intel Core2 Duo E8400 CPU running at 3GHz. The lattices we use are the SVP challenge lattices [13] with seed 0, so we use only one lattice per dimension. For LLL and BKZ reduction we use the NTL library [28] in version 5.5.2. The parameter δ is always set to the standard value 0.99. We run LLL with precision RR followed by BKZ with precision QP.

First we compare our results of CUDA-SSR to BKZ, and second we present experiments comparing CUDA-SSR to CPU-SSR.

[2] `http://www.cdc.informatik.tu-darmstadt.de/mitarbeiter/mischnei.html`

4.1 Comparison of CUDA-SSR and BKZ

Let $\mathbf{B}$ be the basis of $\mathcal{L}(\mathbf{B})$ in dimension n and c be a constant. Using BKZ with blocksize β, Gama and Nguyen [12] predict the average norm of the first basis vector after BKZ reduction to be

$$gn = c^n \det(\mathcal{L}(\mathbf{B}))^{1/n}, \tag{3}$$

where the *Hermite factor constant* c relies on the blocksize used. For BKZ-20, e.g., they experimentally gain a value of $c = 1.0128$.

Our experiments are performed as follows. First, we reduce a lattice basis with BKZ with increasing blocksize, until we reach a vector of desired goal norm gn, cf., Equation (3). We use a value of $c = 1.0129$ to calculate our goal norm. The resulting run times and the reached norms are shown in Figures 2 and 3. Second, we use CUDA-SSR with half the blocksize (rounded up) that BKZ needed to reach the goal norm and run CUDA-SSR on the same lattice; i.e., random sampling has to close the gap between BKZ with half blocksize and BKZ with full blocksize. We stop the GPU sampling when $m = 0.25 \cdot n$ vectors below $0.99 \cdot \|\mathbf{b}_1\|^2$ were found by PAR-SAMPLE.

Figure 4 shows the blocksize that BKZ needed to find the resulting vector. The picture shows that the blocksize is around 20 in most of the cases, as predicted by [12]. Figure 5 shows the speedup factor of CUDA-SSR compared to BKZ.

We notice that with both approaches, BKZ as well as CUDA-SSR, we find vectors of comparable length (Figure 2). CUDA-SSR is always faster (up to 40%). For comparison reason, Figure 3 includes the runtime of BKZ with blocksize $\lceil \beta/2 \rceil$, the pre-processing step of SSR (Line 1 of Algorithm 3). The picture shows that it takes a huge part of the random sampling time (dashed line). This implies that the later part of SSR (sampling - BKZ - sampling - ...) takes a lot less time (the time difference between the dotted and dashed curve) than the initial BKZ. Therefore, the total SSR runtime cannot profit too much from the parallel sampling part in this setting.

The runtime speedup factor (Figure 5) seems to increase with the dimension, from 1.1 in dimension 80 to a maximum value of 1.6 in dimension 160. The peek in dimension 150 is also apparent in Figure 4 and seems to result from special structure in the lattice (SSR is working less in this lattice).

4.2 Comparison of GPU and CPU Variant of SSR

Our second block of experiments is supposed to show the strength of parallelization on GPU of the SSR algorithm. For this, we run our CPU implementation and our GPU implementation of SSR for the same lattices until they undercut the goal norm. For pre-reduction, we use LLL only. We note the reached norm (cf. Figure 6) and the runtime (cf. Figure 7). Figure 8 shows the speedup factor gained by the GPU version. We prepend $m = 0.1 \cdot n$ vectors to the basis in each GPU iteration. Figure 9 compares a typical behaviour of SSR on GPU and CPU over time, concerning the norm of the sampled vectors.

On CPU, the sampling rate was about 160 samples per second for a 180-dimensional lattice. The GTX 295 GPU reached about 120,000 samples per

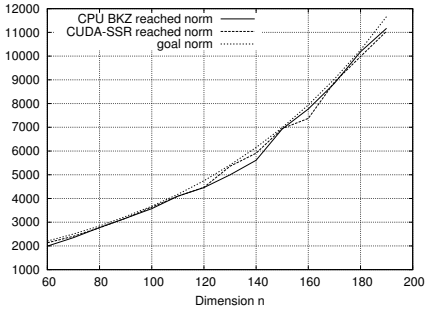


Fig. 2. Reached norm of BKZ and CUDA-SSR

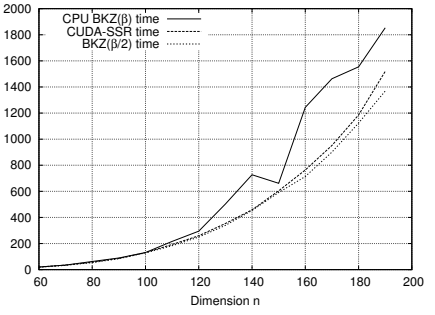


Fig. 3. Required time in seconds for BKZ with blocksize β and for CUDA-SSR to reach the same goal norm

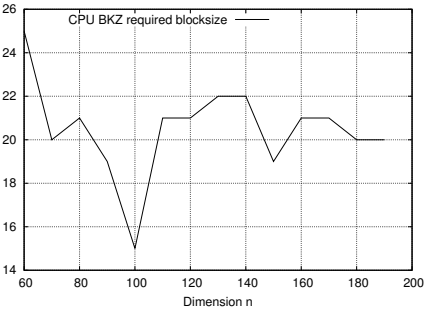


Fig. 4. Required blocksize of BKZ to reach the desired goal norm

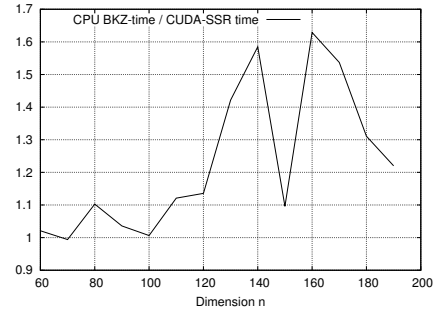


Fig. 5. Speedup factor of CUDA-SSR in comparison to BKZ

second for a 180 dimensional lattice. In smaller dimension, sampling rates of more than 250,000 are possible, e.g. in dimension 60.

From Figure 7 we conclude that the runtime of SSR on GPU is very stable, whereas on CPU (solid curve), we see two different behavior patterns. In some dimensions, e.g. 90 or 110, SSR finds shorter vectors very early, and the runtime is comparable to the CUDA-SSR runtime. In other cases we see huge peeks in the runtime curve, e.g. in dimension 100 or 120, which suggest that on CPU it takes a long time until shorter vectors are found. We conclude that sampling multiple vectors in each iteration helps SSR to run much more stable.

The speedup factor shown in Figure 8 shows the potential of the CUDA version compared to the CPU version. In small dimension we gain speedup factors of up to 180. On GPU, in the first iteration a vector below the bound is already found, whereas on CPU multiple iterations have to be performed. In bigger dimensions, the speedup factor decreases, depending on the behaviour pattern.

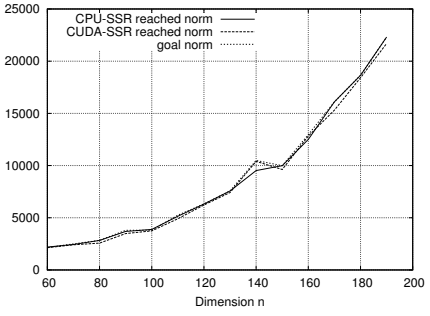


Fig. 6. Reached norm of CPU-SSR and CUDA-SSR

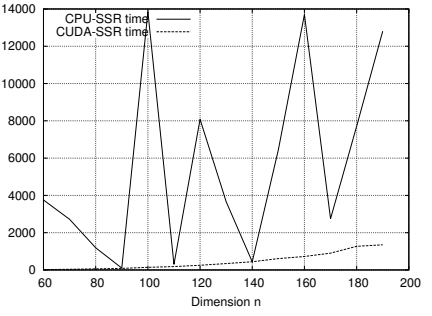


Fig. 7. Required time in seconds of CPU-SSR and CUDA-SSR to reach the same goal norm

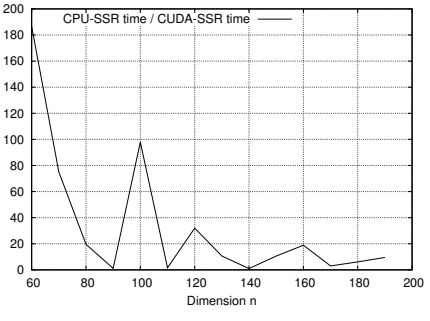


Fig. 8. Speedup factor of GPU compared to CPU variant of SSR

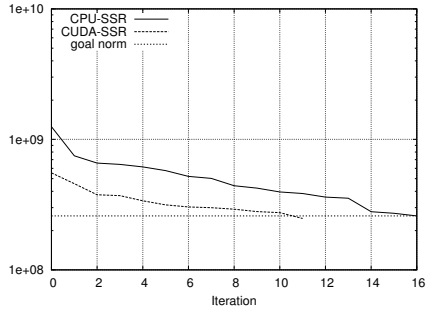


Fig. 9. Development of the squared norm of SSR over time, in a 190 dimensional lattice. The ordinate shows the squared norm of the vectors found by sampling.

Figure 9 shows a typical behaviour of SSR on CPU and GPU. CUDA-SSR starts with lower norm, which implies that the first iterations of SSR decreases the norm much more than on CPU. We noticed that in the first iterations, there exists a huge number of vectors below the $0.99 \|\mathbf{b}_1\|^2$ bound. Therefore, on GPU we have good chance to find a much shorter vector. On CPU only the first vector below the bound is picked, whereas on GPU multiple vectors are prepended to the basis, and all these vectors are potentially smaller than the CPU one.

To show the strength of our GPU version, Figure 10 shows the time needed by CUDA-SSR and CPU-SSR to sample the same amount of vectors, namely 2^{21} many. It is evident that on GPU, the sampling is much faster, with a maximum factor 14.5 in dimension 190.

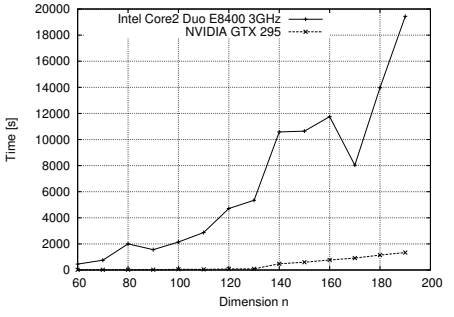


Fig. 10. Time to sample 2^{21} many vectors using CUDA-SSR and CPU-SSR

5 Conclusion and Further Work

We have presented a parallel version of random sampling and an implementation on GPU. Our results show the strength of parallelism for this type of algorithm. Our proposal CUDA-SSR allows for more than 120,000 sampled vectors per second, which is the maximum stated in literature. Unfortunately, the speedups compared to BKZ are not too impressive, due to the big fraction of the runtime that BKZ takes. The percentage of BKZ of the total runtime was up to 97%. This is not optimal, since BKZ does not apply the hardware acceleration of the graphics card. LLL took 67% of the total runtime in dimension 100. [7] mention that sampling takes most of the time, but we were not able to reproduce that.

The speedup in sampling rates is much higher than the speedups in runtime. So the potential of parallelization is visible, but SSR does not take full advantage of it.

The SVP challenge comes with a generator for lattices, to allow participants not only to download one lattice in each dimension but to generate multiple instances. To present smoother graphs it is necessary to run our experiments on multiple instances in each dimension.

In order to allow for good parallelization, we did not include new search spaces as proposed in [7]. Ludwig and Buchmann present *check search space size* (CSSS) functions in order to sample from smaller sets of vectors. It would be interesting to compare how this influences the rate of parallelism and if usage of CSSS could speed up CUDA-SSR even more.

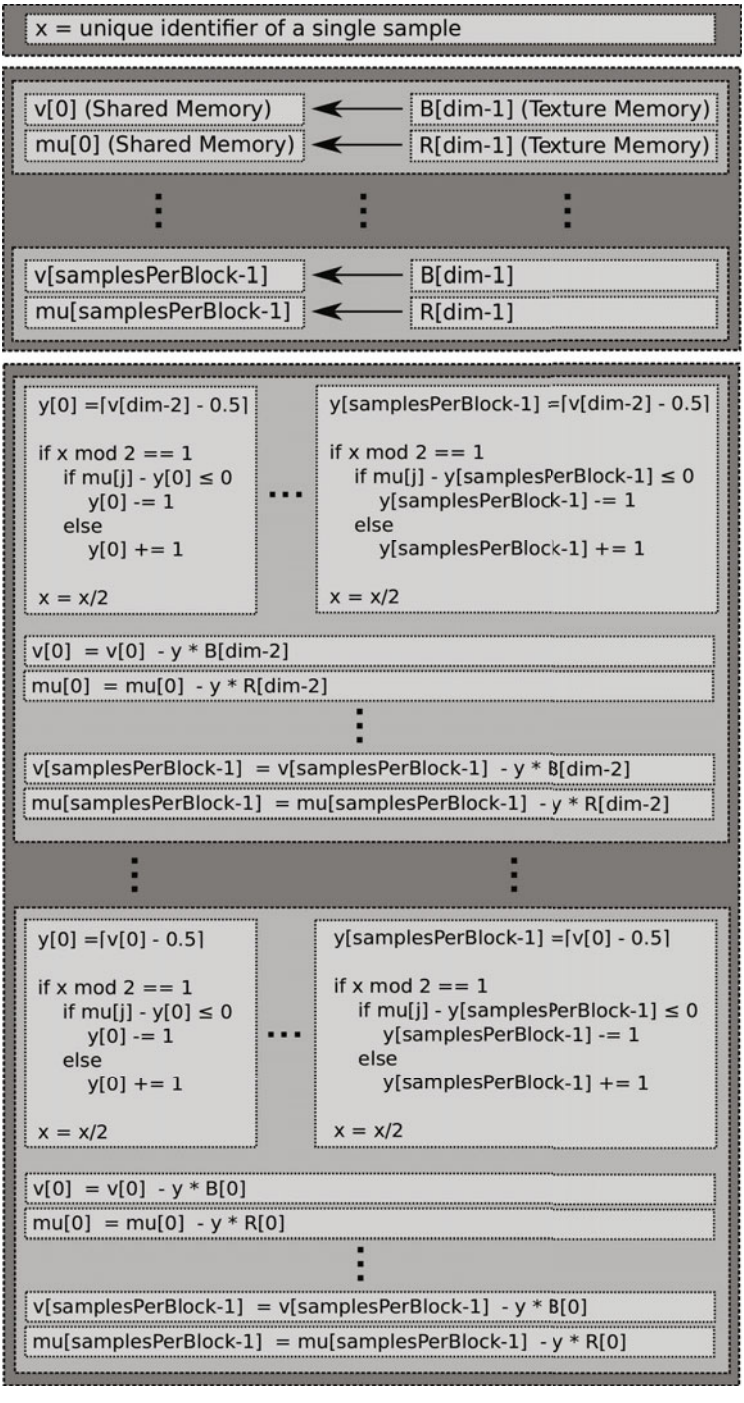
Acknowledgements. Michael Schneider is supported by project BU 630/23-1 of the German Research Foundation (DFG). This work was supported by CASED (`www.cased.de`). We thank the anonymous reviewers for their comments.

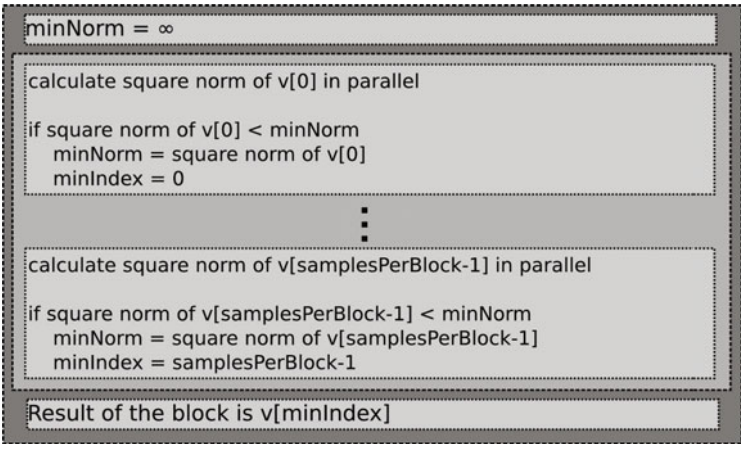
References

1. Advanced Micro Devices. ATI CTM Guide. Technical report, ATI (2006)
2. 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. LNCS, vol. 5704, pp. 960–973. Springer, Heidelberg (2009)
3. 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)
4. Bos, J.W., Stefan, D.: Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 279–293. Springer, Heidelberg (2010)
5. Buchmann, J., Lindner, R.: Secure parameters for SWIFFT. In: Roy, B., Sendrier, N. (eds.) INDOCRYPT 2009. LNCS, vol. 5922, pp. 1–17. Springer, Heidelberg (2009)
6. Buchmann, J., Lindner, R., Rückert, M.: Explicit hard instances of the shortest vector problem. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 79–94. Springer, Heidelberg (2008)
7. 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)
8. 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)
9. Dagdelen, Ö., Schneider, M.: Parallel enumeration of shortest lattice vectors. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 211–222. Springer, Heidelberg (2010)
10. Detrey, J., Hanrot, G., Pujol, X., Stehlé, D.: Accelerating lattice reduction with FPGAs. In: Abdalla, M., Barreto, P.S.L.M. (eds.) LATINCRYPT 2010. LNCS, vol. 6212, pp. 124–143. Springer, Heidelberg (2010)
11. 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)
12. Gama, N., Nguyen, P.Q.: Predicting lattice reduction. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 31–51. Springer, Heidelberg (2008)
13. Gama, N., Schneider, M.: SVP Challenge (2010), `http://www.latticechallenge.org/svp-challenge`
14. 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)
15. Hermans, J., Schneider, M., Buchmann, J., Vercauteren, F., Preneel, B.: Parallel shortest lattice vector enumeration on graphics cards. In: Bernstein, D.J., Lange, T. (eds.) AFRICACRYPT 2010. LNCS, vol. 6055, pp. 52–68. Springer, Heidelberg (2010)
16. Lenstra, A., Lenstra, H., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen 4, 515–534 (1982)
17. Ludwig, C.: Practical Lattice Basis Sampling Reduction. PhD thesis, Technische Universität Darmstadt (2005), `http://elib.tu-darmstadt.de/diss/000640/`
18. Manavski, S.A.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: ICSPC, pp. 65–68. IEEE Computer Society Press, Los Alamitos (2007)

19. Micciancio, D., Goldwasser, S.: Complexity of Lattice Problems: a cryptographic perspective. Kluwer Academic Publishers, Dordrecht (2002)
20. Micciancio, D., Regev, O.: Lattice-based cryptography. In: Bernstein, D.J., Buchmann, J.A., Dahmen, E. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 147–191. Springer, Heidelberg (2008)
21. 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)
22. Nguyen, P.Q., Vallée, B.: The LLL Algorithm - Survey and Applications. Springer, Heidelberg (2010)
23. NVIDIA. Compute Unified Device Architecture Programming Guide. Technical report, NVIDIA (2007)
24. NVIDIA. CUBLAS Library (2007)
25. Schnorr, C.-P.: Block reduced lattice bases and successive minima. Combinatorics, Probability & Computing 3, 507–522 (1994)
26. Schnorr, C.-P.: Lattice reduction by random sampling and birthday methods. In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 146–156. Springer, Heidelberg (2003)
27. Schnorr, C.-P., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. Mathematical Programming 66, 181–199 (1994)
28. Shoup, V.: Number theory library (NTL) for C++, `http://www.shoup.net/ntl/`
29. Szerwinski, R., Güneysu, 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)
30. Villard, G.: Parallel lattice basis reduction. In: ISSAC, pp. 269–277. ACM, New York (1992)

A Flow Chart of Parallel Sampling in CUDA-SSR




minNorm = ∞
calculate square norm of v[0] in parallel
if square norm of v[0] < minNorm
minNorm = square norm of v[0]
minIndex = 0
calculate square norm of v[samplesPerBlock-1] in parallel
if square norm of v[samplesPerBlock-1] < minNorm
minNorm = square norm of v[samplesPerBlock-1]
minIndex = samplesPerBlock-1
Result of the block is v[minIndex]