Gauss Sieve Algorithm on GPUs

Shang-Yi Yang[1], Po-Chun Kuo[1(✉)], Bo-Yin Yang[2], and Chen-Mou Cheng[1]

[1] Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
{ilway25,kbj,doug}@crypto.tw

[2] Institute of Information Science, Acamedia Sinica, Taipei, Taiwan
by@crypto.tw

Abstract. Lattice-based cryptanalysis is an important field in cryptography since lattice problems are among the most robust assumptions, and have been used to construct most cryptographic primitives. In this research, we focus on the Gauss Sieve algorithm, a heuristic lattice sieving algorithm proposed by Micciancio and Voulgaris. We propose the technique of *lifting* computations in prime-cyclotomic ideals into that in cyclic ideals. Lifting makes rotations easier to compute and reduces the complexity of inner products from $O(n^3)$ to $O(n^2)$. We implemented our Gauss Sieve on GPUs by adapting the framework of Ishiguro et al. in a single GPU, and the one of Bos et al. among multiple GPUs. We found a short vector at dimension 130 in the Darmstadt Ideal SVP Challenge (currently in first place in the Hall of Fame) using 8 GPUs in 824 h using our implementation.

Keywords: Lattice-based cryptography · Sieving algorithm · Gauss Sieve · GPU · Parallelization · Shortest vector problem · SVP · Ideal lattices

1 Introduction

Over the past two decades, lattice-based cryptosystems have attracted widespread interest. Not only are they among the group of PKCs that will potentially defend against the quantum threats, but they also provide the first constructions of many new cryptographic functionalities, e.g. fully-homomorphic encryption and multilinear maps [Gen09,GGH13]. Furthermore, in 1997 Ajtai and Dwork proved that some lattice problems possess worst-case to average-case reductions [AD97], which gives a strong guarantee on the security of lattice-based cryptosystems, and inspired the construction of many cryptographic primitives. Although many such constructions are in practice infeasible, ideal lattices have made both the keys shorter and algorithms faster, which brings many more new ideas closer to practicality.

Many lattice- and ideal-lattice-based schemes claim to base their security on the shortest vector problem (SVP): if the shortest vector in a lattice could be found, the lattice-based cryptosystems would be broken. However, it is unclear how to choose secure yet practical parameters for these schemes, and an accurate

H. Handschuh (Ed.): CT-RSA 2017, LNCS 10159, pp. 39–57, 2017.
DOI: 10.1007/978-3-319-52153-4_3

assessment of their security levels would be indispensable if we are to select suitable parameters for them.

Several exact or approximate algorithms have been proposed for the SVP problem. Exact algorithms include enumeration, sieving, and ones based on Vonoroi cells [MV10]. The Vonoroi cell method, though single exponential both in time and space, proved to be impractical for dimensions higher than 10. On the other hand, lattice enumeration is an exhaustive search algorithm. The time complexity of lattice enumeration is $2^{O(n^2)}$ or $2^{O(n \log n)}$ and space complexity is polynomial [GNR10,KSD+11]. Lastly, sieving algorithms have $2^{O(0.52n)}$ time complexity and $2^{O(n)}$ space complexity [MV10]. In general, lattice enumeration has remained the fastest approach to solve SVP so far since almost all the data could be store in the CPU cache. However, the general approach is ill-suited for parallelizing on GPUs or similar wide vector architectures. For example, the speed-up in [KSD+11] is less than a factor of ten. It is also unclear how to make use of the special structure of ideal lattices when using enumeration.

In contrast, approximate algorithms run in polynomial time but output an approximate solution. Even though the output short vectors have length exponential in dimension, such vectors are in fact good enough for some applications or cryptanalysis. For example, the famous LLL algorithm can find, with high probability, the shortest vector for Goldstein-Mayer random lattices in the SVP challenge [Lat] for dimensions less than 30. For higher dimensions, however, the quality of vectors it outputs is insufficient. On the other hand, in the BKZ algorithm, which uses enumeration in sub-lattices as a subroutine, we can trade off execution time against the approximate factor. Whether sieving algorithms could be used as a sub-routine in the BKZ algorithm is still an open problem. Another open problem is whether there exists a poly-time algorithm which outputs a short vector with a polynomial approximation factor.

Although enumeration seems fastest so far in practice, sieving has a better complexity upper bound and may yet outperform enumeration in higher dimensions. Moreover, as far as we know sieving is currently the only way to use the ideal lattice structures. There are several papers on how to parallelize sieving and how to use the cyclic lattice structure, but how to do this on GPUs and use other ideal lattice structures is not clear yet. Previous works by Ishiguro et al. [IKMT14,MDB14] also seem to be limited to cyclic, anti-cyclic and trinomial ideal lattices. In this paper, we broaden the scope to include prime cyclotomic ideal lattices, and make the following contributions:

– We propose and implement the first lattice sieving algorithm for a single machine with multiple GPUs. Our variant includes two carefully designed layers of parallelism, both inter-GPU and intra-GPU (Sect. 5).
– We show that by *lifting* lattice vectors generated by the polynomial $x^n + \cdots + 1$ into ones generated by $x^{n+1} - 1$, not only do inner products (the critical path of Gauss Sieve) speed up, some register rotation problems on GPUs are mitigated (Sect. 4). Moreover, by heuristically applying *lazy rotation*, the complexity of reduction between two vectors with all their rotations goes down from $O(n^3)$ to $O(n^2)$ (only a constant times slower than the anticyclic lattice, cf. Sect. 4.3).

- We carefully crafted the reduction kernel to exploit both thread- and instruction-level parallelism (Sect. 6). Special care is taken with the layout of vectors in the register file, and some kernel-level heuristics are introduced that use the ideal lattice property.
- Incorporating these improvement into our implementation on GPUs, we were able to solve challenges of dimension 130 within 6583 GPU-hours. Our GPU implementation is 21.5 (resp. 55.8) times faster than a single-core CPU for general (resp. ideal) lattices (Sect. 7.2).
- We provide a lower bound complexity estimation for the SVP compared to the previous work (Sect. 7.4).

2 Preliminary

2.1 Definition and Notation

A *lattice* is a discrete additive group of all integer combinations of a basis $v_1, v_2, ..., v_m \in \mathbb{R}^n$, where $m \leq n$. In cryptography, integer lattices are often used, namely, the basis vectors are defined over $\mathbb{Z}^n$. The bases corresponding to a lattice are not unique, since multiplying a uni-modular matrix to a basis would not change the lattice spanned by the basis. We use $\mathcal{L}(B)$ to denote the lattice spanned by the basis B.

The first successive minimum $\lambda_1(\mathcal{L})$ is the length of the shortest nonzero vector of the lattice $\mathcal{L}$. The shortest vector problem (SVP) asks for the shortest nonzero vector in a given lattice. The SVP is NP-hard under randomized reduction [Ajt97]. The approximation shortest vector problem (SVP$_\alpha$) asks for a short vector of length shorter than $\alpha\lambda_1(L)$.

Extending the idea into rings, we have *ideal lattices*, a special class of lattices. Consider an ideal of a ring $\mathcal{I} = \langle g \rangle \subseteq \mathbb{Z}[x]/f(x)$, where f is a monic irreducible polynomial of degree n, an ideal lattice is $\mathcal{L}(B) \in \mathbb{Z}^n$ such that $B = \{g \mod f : g \in \mathcal{I}\}$. The polynomial of a ring affects its structure and computation cost. Thus, cryptographers are concerned with four type of ideal lattices defined by the polynomial $f(x)$:

- *Cyclic* ideal lattice, with $f_{cyclic}(x) = x^n - 1$, are the simplest ones and easy to compute. However, since the polynomials are always divided by $x - 1$, this kind of ideal lattice does not guarantee the worst-case collision resistance.
- *Anti-cyclic* ideal lattice, with $f_{anti\text{-}cyclic}(x) = x^n + 1$, are also eligible for easy multiplication and convolution. Such polynomials are irreducible over $\mathbb{Z}$ if n is a power of 2. This kind of ideal lattice is commonly used in cryptography.
- *Prime-cyclotomic* ideal lattices, with $f_{prime\text{-}cyclotomic}(x) = x^n + x^{n-1} + \cdots + 1$, are the main type we focus on. If $n + 1$ is prime, $f_{prime\text{-}cyclotomic}(x)$ is irreducible.
- *Trinomial* ideal lattices, with $f_{trinomial}(x) = x^n + x^{n/2} + 1$ where $n/2$ is a power of three, are the ones considered in [IKMT14].

By the definition of an ideal lattice, the vector $u = (u_0, u_1, \cdots, u_{n-1}) \in \mathbb{Z}^n$ also indicates a polynomial $u(x) = u_0 + u_1 x + \cdots + u_{n-1}x^{n-1} \in \mathbb{Z}[x]/f(x)$, the polynomial $x \cdot u(x)$ is still in the ideal. Thus, the vector corresponding to such polynomial is called the (*first*) *rotation* of u, denoted as $rot(u)$. For example, consider $f(x) = x^n - 1$, the rotation of $u = (u_0, u_1, \cdots, u_{n-1})$ is $\mathbf{rot}(u) = (u_{n-1}, u_0, u_1, \cdots, u_{n-2})$.

The central notation of the Gauss Sieve is Gauss reduction. Two vectors $u, v \in \mathcal{L}(B)$ satisfying $\|u \pm v\| \geq \max(\|u\|, \|v\|)$ are called *Gauss-reduced*. Given two arbitrary vectors u and v, we can reduce u with respect to v by $u \leftarrow u - \lfloor \frac{\langle u,v\rangle}{\langle v,v\rangle} \rceil v$. Thus, given two arbitrary vectors u and v, we can convert them into Gauss-reduced ones by repetitively applying the reduction procedure alternatingly, in a Euclidean algorithm-like manner, until the vectors no longer change. If any two vectors in a set are Gauss-reduced, it is *pairwise-reduced*.

Algorithm 1 shows the pseudo-code for reducing the list U with the list V. Algorithm 2 is the ideal lattice counterpart. In Algorithm 2, *times* represents the number of possible rotations in the input lattice. In other words, $x^{times} = \pm 1$. As concrete examples, for anti-cyclic lattices, $times = n$; for prime cyclotomic lattices, $times = n + 1$.

Algorithm 1. Gauss reduction between two lists for general lattices

```
Input  : Lists $U$ and $V$
Output: Reduced list $U$
1  foreach $u \in U$ do
2      foreach $v \in V$ do
3          if $2 \cdot |\langle u,v\rangle| > \langle v,v\rangle$ then
4              $u \leftarrow u - \lfloor \frac{\langle u,v\rangle}{\langle v,v\rangle} \rceil v$
5              Mark $u$ as reduced.
```

Algorithm 2. Gauss reduction between two lists for ideal lattices

```
Input  : Lists $U$ and $V$
         Number of rotations: $times$
Output: Reduced list $U$, with all possible rotations
1  foreach $u \in U$ do
2      foreach $v \in V$ do
3          for $i \leftarrow 0$ to $times - 1$ do
4              $w \leftarrow x^i v$
5              for $j \leftarrow 0$ to $times - 1$ do
6                  $(s, t) \leftarrow (x^j u, x^j w)$
7                  $m \leftarrow \lfloor \langle s,t\rangle / \langle t,t\rangle \rceil$
8                  if $m \neq 0$ then
9                      $u \leftarrow s - mt$
10                     Mark $u$ as reduced.
```

2.2 CUDA Programming

Here we provide a minimalist CUDA programming introduction, including only relevant information that our implementation takes into consideration. For more details, please refer to the CUDA C Programming Guide [CUD15].

Graphics processing units (GPUs) are high throughput, many-core architectures. Currently, the most widely used GPU development toolchain is CUDA by NVIDIA. CUDA supports writing fine-tuned programs for NVIDIA graphic cards. In this paper, we will especially focus on GPUs of the Maxwell architecture.

The CUDA programming model requires programmers to think in the *single instruction, multiple thread* (SIMT) programming model. The model exposes three key abstractions to programmers: a hierarchy of thread groups, shared memories, and barrier synchronization. Threads are first organized in blocks, and blocks are then organized in grids. A grid of GPU threads must run the same program (the kernel).

At the system level, blocks are independently dispatched to different processors. Since each block has a dedicated on-chip cache called the shared memory, threads within a block can only exchange data through the shared memory. However, this requires an explicit synchronization barrier that halts all the threads in a block, and thus can be a huge performance overhead for critical applications.

Fortunately, starting from the Kepler architecture, data exchange within a *warp* can be done using the *warp shuffle* instructions without any explicit synchronization barrier. A *warp*, consisting of 32 consecutive threads, is the smallest batch that can be scheduled and issued at once by a processor. For example, using the warp shuffle instructions, summing different values from threads with in a warp can be done relatively fast through the *parallel reduction* paradigm. If the threads in a warp are executing different instructions – most likely because of different branch conditions – severe *warp divergence* can occur, drastically lowering the warp utilization.

3 Background

3.1 Sieving Algorithms

The first sieving algorithm was proposed by Ajtai et al. in 2001 [AKS01]. They proved that the time/space complexity is $2^{O(n)}$, which is the first single-exponential time algorithm solving SVP. Following works either provided tighter theoretical bounds on the complexity [NV08,MV10,Sch11,Sch13], or improved the algorithm [MS11,MDB14,MBL15].

3.2 Gauss Sieve

The Gauss Sieve algorithm was proposed by Micciancio and Voulgrais in [MV10] and is the most practical version of sieving algorithms. The main idea of the algorithm is to mutually reduce samples with a list of vectors by Gauss reduction.

After Gauss reduction, the angle between any pair of two vectors is larger than $60°$. By the Kabatiansky-Levenshtein theorem, one can bound the number of such vectors, and thus obtain the time complexity of the algorithm.
The Gauss Sieve algorithm has been implemented on CPU in [IKMT14] and some improvements have been proposed by Bos et al. in [BNvdP14]. The work of Ishiquro et al. improved and implemented the parallel version of the Gauss Sieve algorithm proposed by Schneider [Sch13], and they also adapt to a specific ideal lattice called negacyclic ideal lattices. However, we promote this into more general polynomial ring and improve the performance. Later, Bos et al. proposed a different variant of the parallel Gauss Sieve algorithm which is more suited for high dimension lattice [BNvdP14]. We will expound on their ideas in Sect. 5. Moreover, Laarhoven incorporated locality-sensitive hashing into the algorithm [Laa15, BDGL16, BL16]. Instead of searching all the vector in the list, they group together near vectors using hash functions. Therefore, vectors are only reduced with more geometrically possible ones.

Prime Cyclotomic Rotation. First we state a nice property of anti-cyclic lattices.

Lemma 1 [BNvdP14]. *Let $a, b \in R = \mathbb{Z}[x]/(x^n + 1)$ with coefficient vector a, b. If $2\|\langle a, x^l \cdot b\rangle\| \leq \min\{\langle a, a\rangle, \langle b, b\rangle\}$ for all $0 \leq l < n$, then $x^i \cdot a$ and $x^j \cdot b$ are Gauss-reduced for all $i, j \in \mathbb{Z}$.*

In contrast to the anti-cyclic case described in Lemma 1, prime cyclotomic lattices do not possess this property. Therefore, all the rotations of two vectors can contribute to the global status. Thus, the list size might be even smaller.

However, prime cyclotomic lattices may have some disadvantages. To illustrate the computational overhead to find the norms of (all the) rotations of a vector, consider the vector $v = (5, 4, 3, 2, 1)$ in an ideal lattice generated by the polynomial $f(x) = x^5 + x^4 + x^3 + x^2 + x + 1$. The first rotation of v is

$$\mathbf{rot}(v) = (-1, 5-1, 4-1, 3-1, 2-1) = (-1, 4, 3, 2, 1).$$

Squaring and summing, we have the squared norm for $x \cdot v$:

$$\|\mathbf{rot}(v)\|^2 = (-1)^2 + 4^2 + 3^2 + 2^2 + 1^2 = 31.$$

Calculating norms like this can be slow, because only when the vector $\mathbf{rot}(v)$ is ready can we calculate the sum of squares. However, the value that is required in Gauss reduction is just $\|\mathbf{rot}(v)\|^2$, but not $\mathbf{rot}(v)$ per se. For processors with ADD, MUL and FMAD (fused multiply-add) instructions, it takes $2n$ operations to calculate the norm of an n-dimensional vector.

In this paper, we will see how to circumvent this by *lifting* a vector. By doing this, not only is the computation easier, but it also enables optimizations that are not possible without lifting.

4 Lifting Ideal Lattices

We now develop the properties for prime cyclic lattices and see how they can facilitate computation.

4.1 Lifting Prime Cyclotomic Polynomials

The idea behind lifting lattices is to supplement vectors with a bit of redundant information to ease computation. Specifically, we will express an n-dimensional vector with an $(n+1)$-dimensional one. Let $\mathcal{L}$ be a lattice generated by $x^n + x^{n-1} + \cdots + 1$, and $\overline{\mathcal{L}}$ by $x^{n+1} - 1$. We wish to seek a way to connect the two lattices according to the following criteria:

- The conversion of vectors between the two lattices is simple.
- The rotation of vectors must be preserving, so that the complicated rotation in $\mathcal{L}$ can be done instead cyclically in $\overline{\mathcal{L}}$.

Technically speaking, we are looking for simple ring homomorphisms between $\mathbb{F}[x]/(x^n + x^{n-1} + \cdots + 1)$ and $\mathbb{F}[x]/(x^{n+1} - 1)$.

An intuitive clue to accomplish this comes from the observation that the polynomial $x^{n+1} - 1$ factorizes as

$$x^{n+1} - 1 = (x-1)(x^n + x^{n-1} + \cdots + 1).$$

This suggests we connect u and its *lift* $\bar{u}$ by thinking of $\bar{u}$ as reduced modulo $x^n + x^{n-1} + \cdots + 1$:

$$u \equiv \bar{u} \pmod{x^n + x^{n-1} + \cdots + 1}.$$

Note that this choice also preserves rotation.

As an example, lifting directly $u = (1,2,3,4,5)$ in a lattice generated by $x^4 + x^3 + x^2 + x + 1$ gives $\bar{u} = (1,2,3,4,5,0)$ in a lattice generated by $x^5 - 1$. This is not the only way to lift u. Another possibility is $\bar{u}' = (2,3,4,5,6,1)$, since $(2-1, 3-1, 4-1, 5-1, 6-1) = (1,2,3,4,5)$. In general, to lift any u, we can choose p arbitrarily and lift u as $\bar{u} = (u_0 + p, u_1 + p, \cdots, u_n + p, p)$.

From now on, we will write a bar on top of a symbol to indicate that it is lifted from its underlying form. For example, $\bar{u}$ is a lift of the vector u and $\overline{L}$ is a lift of the lattice L. Whenever we see a lifted vector $\bar{u}$, we should keep in mind that it is merely a surface form representing its underlying original vector.

4.2 Norms and Inner Products

During the Gauss reduction, we are especially interested in the norms and inner products of rotations of vectors. Let us see how to derive these quantities for the underlying lattice directly, without converting from the lifted lattice.

Suppose $\bar{u} = (\bar{u}_0, \bar{u}_1, \cdots, \bar{u}_{n-1}, \bar{u}_n)$ in $\overline{\mathcal{L}}$. We first reduce $\bar{u}$ modulo the polynomial $x^n + x^{n-1} + \cdots + 1$ to get its underlying form:

$$\begin{aligned} u &= (\bar{u}_0 - \bar{u}_n, \bar{u}_1 - \bar{u}_n, \cdots, \bar{u}_{n-1} - \bar{u}_n) \\ &= (\bar{u}_0 - p, \bar{u}_1 - p, \cdots, \bar{u}_{n-1} - p). \end{aligned}$$

Here, we rewrite $\bar{u}_n$ as p interchangeably, since $\bar{u}_n$ acts as a *pivot* for the vector.

We can now calculate the norm of u:

$$\begin{aligned} \langle u, u\rangle^2 &= \sum_{i=0}^{n-1} (\bar{u}_i - \bar{u}_n)^2 \\ &= \sum_{i=0}^{n} (\bar{u}_i - \bar{u}_n)^2 && \text{since } \bar{u}_n - \bar{u}_n = 0. \\ &= \sum_{i=0}^{n} \bar{u}_i^2 - 2\bar{u}_n \sum_{i=0}^{n} \bar{u}_i + (n+1)\bar{u}_n^2 \\ &= \boxed{\langle \bar{u}, \bar{u}\rangle^2} - 2p \boxed{\sum_{i=0}^{n} \bar{u}_i} + (n+1)p^2, \end{aligned}$$

where the boxed terms remain constant throughout all cyclic rotations of $\bar{u}$, and thus can be saved beforehand. Note that we do not need to know what u is at all.

Similarly, the inner product of two vectors u and v is

$$\langle u, v\rangle^2 = \langle \bar{u}, \bar{v}\rangle^2 - p \sum_{i=0}^{n} \bar{v}_i - q \sum_{i=0}^{n} \bar{u}_i + (n+1)pq,$$

where q is the pivot of $\bar{v}$.

Simplifying Formulae. Although these formulae may look intimidating, we can always simplify them by choosing the "right" pivot. If we set $\sum_{i=0}^{n} \bar{u}_i = 0$, and solve for p:

$$\begin{aligned} 0 &= \bar{u}_0 + \bar{u}_1 + \cdots + \bar{u}_{n-1} + p && \text{rewrite } \bar{u}_n \text{ as } p. \\ &= (u_0 + p) + (u_1 + p) + \cdots + (u_{n-1} + p) + p \\ &= (u_0 + u_1 + \cdots + u_{n-1}) + (n+1)p, \end{aligned}$$

we can choose the pivot as

$$p = -\frac{\sum_{i=0}^{n-1} u_i}{n+1}.$$

This is our standard way to lift a vector.

Carrying out the same procedure for v, we can now write the inner product succinctly:

$$\langle u, v\rangle^2 = \langle \bar{u}, \bar{v}\rangle^2 + (n+1)pq.$$

Lifting in this manner, we amend Algorithm 2 into Algorithm 3. Note that the underlying vectors s and t are no longer needed on line 12. Since we do not have to track and update $\sum u_i$ anymore, simplifying in this manner eases some computational burden and memory overhead in the innermost loop for GPUs. However, integer vectors are now represented by floating points, which may lead to error accumulation after several rounds. We choose to rectify these vectors by unlifting and rounding the numbers when they are taken out from the stack for later rounds.

We could also eliminate the $n+1$ by "normalizing" and dividing vectors by $\sqrt{n+1}$, but this is less intuitive.

Algorithm 3. Gauss reduction between two lists for prime cyclotomic lattices (lifted)

```
   Input  : Lifted lists $\overline{U}$ and $\overline{V}$
   Output : Reduced, lifted list $\overline{U}$
1  foreach $\bar{u} \in \overline{U}$ do
2      foreach $\bar{v} \in \overline{V}$ do
3          for $i \leftarrow 0$ to $n$ do
4              $\bar{w} \leftarrow x^i \bar{v}$
5              $\langle \bar{w}, \bar{w}\rangle \leftarrow \langle \bar{v}, \bar{v}\rangle + (n+1)\bar{v}_{n-i}^2$
6              for $j \leftarrow 0$ to $n$ do
7                  Calculate $\langle \bar{u}, \bar{w}\rangle$.
8                  $\langle s, t\rangle \leftarrow \langle \bar{u}, \bar{w}\rangle + (n+1)\bar{u}_{n-j}\bar{w}_{n-j}$
9                  $\langle t, t\rangle \leftarrow \langle \bar{w}, \bar{w}\rangle + (n+1)\bar{w}_{n-j}^2$
10                 $m \leftarrow \lfloor \langle s, t\rangle / \langle t, t\rangle \rceil$
11                 if $m \neq 0$ then
12                     $(\bar{s}, \bar{t}) \leftarrow (x^j \bar{u}, x^j \bar{w})$
13                     $\bar{u} \leftarrow \bar{s} - m\bar{t}$
14                     Mark $\bar{u}$ as reduced.
```

4.3 Lazy Rotation

We now address two GPU performance bottlenecks in Algorithm 3, and provide two kernel-level heuristics to solve these problems.

First, on lines 12–14, whenever $\bar{u}$ is reduced, it is assigned as the difference of two rotated vectors $\bar{s}$ and $m\bar{t}$. However, such register indexing, unlike on CPUs, can cause spills on GPUs. Since $\bar{s} - m\bar{t} = x^j(\bar{u} - m\bar{w})$, we can instead write $\bar{u} \leftarrow \bar{u} - m\bar{w}$ and choose to rotate $\bar{u}$ back *lazily* after the kernel finishes. Now

Algorithm 4. Gauss reduction between two lists for prime cyclotomic lattices (lifted, with lazy rotation)

```
Input  : Lifted lists $\overline{U}$ and $\overline{V}$
Output : Reduced, lifted list $\overline{U}$
1  foreach $\bar{u} \in \overline{U}$ do
2      $norm \leftarrow \langle \bar{u}, \bar{u} \rangle + (n+1)\bar{u}_n^2$
3      foreach $\bar{v} \in \overline{V}$ do
4          for $i \leftarrow 0$ to $n$ do
5              $\overline{w} \leftarrow x^i \bar{v}$
6              $\langle \overline{w}, \overline{w} \rangle \leftarrow \langle \bar{v}, \bar{v} \rangle + (n+1)\bar{v}_{n-i}^2$
7              Calculate $\langle \bar{u}, \overline{w} \rangle$.
8              for $j \leftarrow 0$ to $n$ do
9                  $\langle s, s \rangle \leftarrow \langle \bar{u}, \bar{u} \rangle + (n+1)\bar{u}_{n-j}^2$
10                 $\langle s, t \rangle \leftarrow \langle \bar{u}, \overline{w} \rangle + (n+1)\bar{u}_{n-j}\overline{w}_{n-j}$
11                 $\langle t, t \rangle \leftarrow \langle \overline{w}, \overline{w} \rangle + (n+1)\overline{w}_{n-j}^2$
12                 $m \leftarrow \lfloor \langle s, t \rangle / \langle t, t \rangle \rceil$
13                 $norm_{new} \leftarrow \langle s, s \rangle - 2m\langle s, t \rangle + m^2 \langle t, t \rangle$
14                 if $norm_{new} < norm$ then
15                     $m_{best} \leftarrow m$
16                     $norm \leftarrow norm_{new}$
17             if $m_{best} \neq 0$ then
18                 $\bar{u} \leftarrow \bar{u} - m_{best}\overline{w}$
19                 $\langle \bar{u}, \bar{u} \rangle \leftarrow \langle \bar{u}, \bar{u} \rangle - 2m_{best}\langle \bar{u}, \overline{w} \rangle + m_{best}^2 \langle \overline{w}, \overline{w} \rangle$
20                 Mark $\bar{u}$ as reduced.
```

the lazy version of $\bar{u}$, however, may be representing a vector much longer than it should. To prevent reducing with a lazy $\bar{u}$ in later rounds, we need to keep track of the current correct norm of u. This is done on lines 14–16 in Algorithm 4.

Second, because $\bar{u}$ may have changed in the previous round, $\langle \bar{u}, \overline{w} \rangle$ must be recalculated on line 7. To avoid recalculating $\langle \bar{u}, \overline{w} \rangle$ repeatedly, observe that the probability of reducing $\bar{u}$ more than once is not high in the innermost loop. We can keep track of the best m so far, moving the entire if statement on lines 11–14 out and after the for loop.

Applying these two heuristics, we now reach Algorithm 4. This amended algorithm is more efficient because (1) the body of the most inner loop runs in constant time, thus reducing the complexity to calculate all inner products of two vectors from $O(n^3)$ to $O(n^2)$, and (2) the need to rotate $\bar{u}$ is completely eliminated.

4.4 Generalizing Lifting

The regularity of terms in the quotient polynomial $f(x)$ plays a role in the computation of rotations. Consider a cyclotomic polynomial $p(x)$. There might exist another low degree polynomial $r(x)$ such that $p(x)r(x) = x^n \pm 1$. This suggests we promote a vector with dimension $\deg(p(x))$ into dimension $\deg(p(x))+\deg(r(x))$, thus lowering the computation cost. The same technique of choosing the right pivots can be applied as well.

For example, the next unsolved ideal lattice challenge is dimension 132. One of the ideal lattices in the challenge is generated by the polynomial $f(x) = x^{132} - x^{130} + x^{128} - \cdots + x^4 - x^2 + 1$. Since $(x^2 + 1)f(x) = x^{134} + 1$, we can convert this lattice into a 134-dimensional anti-cyclic lattice with two pivots, one for the $+1$ terms and the other for -1 terms.

5 Parallelization

Let us now look at our parallel variant of Gauss Sieve for a single machine with multiple GPUs. Two layers of parallelization naturally arise in this setting: the workload should first be split (1) across different GPUs, then (2) to different processors within a GPU. These two layers of architectures differ in communication cost. Broadcasting data from the host memory across all the GPUs through PCIe is much more expensive than broadcasting data from the on-chip memory to different processors within a single GPU.

We carefully design these two layers in hope to mitigate communication overhead. Specifically, we view each GPU as an independent sieve (inner layer), and all the GPUs cooperate as a complete parallel sieve (outer layer). In the following subsections, we will see (1) how the problem is divided into independent sub-sieves on different GPUs, so that each sub-sieve acts as a blackbox, ordinary Gauss Sieve, and (2) how the sub-sieve is designed to maximize GPU power.

5.1 Outer Layer

To distribute the work among the GPUs on a single machine, we first recall the work by Bos et al. [BNvdP14], which was originally designed for computer clusters. In their work, each node acts as an independent Gauss Sieve, maintaining its own local list while reducing the same batch of samples broadcast over all the nodes. These nodes communicate only at the end of each iteration, putting any sample that is ever reduced in any of the nodes to the stack. The advantage of this approach is that the long, local lists are never completely moved out of the nodes; only a limited amount of reduced vectors and samples are involved in communication. Communication cost is thus small.

Here, we adapt their method to a machine with multiple GPUs using the following analogy: A cluster is to the machine what a node is to the GPU. As a result, a GPU now works as if it were a node, having its own local list, and communication is done on the host. At each iteration, all the GPUs are given the same batch of samples, either newly generated or from the stack. Each GPU then first reduces its samples mutually with its local list, using the method described in Subsect. 5.2. Next, for each sample, if ever reduced in one GPU, the host compares and chooses the shortest "representative", putting it to the stack.

Reduced vectors from local lists are also put on the stack. Last, the “surviving” samples are appended to the shortest local list. The vectors on the stack become the input for later rounds.

5.2 Inner Layer

The inner layer is a modified version of Ishiguro et al.’s idea. As mentioned in the previous subsection, each GPU can be thought of as an independent sieve, reducing its local list with a batch of samples. First, the local list is reduced with the samples. Then, such samples are mutually reduced with each other. Finally, the samples are reduced with the local list. If any vector is ever reduced during any step, it is marked and later collected on the stack. As showed by Ishiguro et al., any pair of surviving vectors in the local list remains reduced during the process.

Since these three steps share the same pattern — they all reduce one list with another, the same GPU kernel can be used. See Algorithm 1 for general lattices and Algorithm 2 for ideal lattices. To reduce list A with list B, the kernel takes as inputs list A and list B, and in-place outputs the reduced list A. In the kernel, list A is sliced into adequate chunks and distributed to different processors, while list B is broadcast to all processors. The kernel is crafted with care to ensure high throughput, as will be described in the next section.

6 Implementation Details

In this section, we will first see how common performance tuning techniques can be applied to our algorithm. This includes thread- and instruction-level parallelism. Next, we point out more kernel optimization tricks. Finally, we describe two more heuristics that can significantly improve the execution time.

6.1 Vector Layout

On GPUs, each thread has a physical register number limit; depending on how many resources each thread requires, each processor also has a runtime limit for thread numbers. For example, consider the kernel for $n = 100$. On a Maxwell GPU, each thread can use up to 255 registers. If we put both $\bar{u}$ and $\overline{w}$ in one thread, we need $2 \times (100 + 1) = 202$ registers. Although fewer than 255, this is still so much that the processors can only schedule a few threads, limiting thread-level parallelism. At the other extreme, if we spread a vector across too many threads, the overhead of parallel reduction to calculate inner products collectively will take over.

Empirically, we choose to spread a vector across 4 threads. For our target dimension 130, this means each thread takes $\lceil 130/4 \rceil = 33$ elements, with the extra two elements padded with 0. This choice not only reduces register pressure, but also makes the vector length a multiple of 4, which is essential for cache line alignment. To this end, parallel reductions are needed both on line 7 to collectively sum inner products, and before line 17 (after the for loop) to agree on the best m. To make the code more readable, we use the CUB library [Mer] for block load and store in the kernel.

6.2 Instruction-Level Parallelism

Yet another commonly applied trick to increase GPU utilization is to exploit instruction-level parallelism. The idea is to issue independent instructions at once to increase the pipeline usage. However, since the algorithm is very much inherently dependent from line to line, the direct implementation will run very slowly. We do not overlap two independent copies of kernel at the same time, because the register usage is immediately multiplied by two. Instead, we unroll the loop on line 4 with an empirical factor of 8 to facilitate register reuse on line 7. This technique is possible only if the lattice is lifted, since a lattice point is represented in its cyclic form.

Next, we identify two new heuristics due to loop unrolling. First, at the end of each 8th iteration, we choose the best m_{best} from eight possible m_{best}'s. Second, since the prime n is never a multiple of 8, empirically we just omit the remainder of the unrolled loop.

6.3 More Kernel Optimizations

– The rotation on line 5 is tricky because vectors are padded with zero. Therefore, the last thread that contains a vector would have to deal with these zeros. In fact, the padded vector $\bar{v}$ is first stored in the shared memory, then rotated one by one at each iteration. More specifically, at the end of one iteration, the first padded zero is replaced with the next "right" element, and at the start of the next iteration, the vector $\overline{w}$ is read at the "right" offset.
– The vectors in the lists $\overline{U}$ and $\overline{V}$ are loaded in bulks and put in a shared-memory buffer to increase global memory throughput.
– In practice, we choose the first element of a lifted vector as its pivot and rotate reversely. This transforms the index $n - j$ on lines 9–10 to an easier j.
– To ensure high kernel throughput, we empirically tune all the parameters mentioned in the above sections as well as kernel launch parameters, although it is not feasible to try all possible combinations.

6.4 On Faster Convolution

The question naturally arises: why not use FFT or the Karatsuba algorithm to calculate inner products? The reasons are:

– The Karatsuba algorithm reduces the number of multiplications, while adding a lot more additions. On GPUs, however, FMAD is fastest.
– If on line 5, we use them to produce results for all i's simultaneously, there will not be enough register to hold both the results and all the intermediate values during computation.

– The dimension is not a power of 2, which makes the convolution more difficult to be designed efficiently.

There are several techniques to convert non-power-of-2 DFT's into convolutions or FFTs of the same or larger dimensions. The best approach we are aware of is Devil's convolution [Cra96], but this is not easily applicable on GPUs.

6.5 Heuristics

Besides the heuristics for kernel optimization, we also applied two heuristics to speed up in conjunction with the techniques above.

First, as already mentioned in Voulgaris's implementation [Gau], the lists in step 1 are sorted so that only longer vectors (before rotation) are reduced with shorter ones. We also tried to see if this can be applied to steps 2 and 3. Empirically we found that it is not as effective, probably because vectors are shorter during steps 2 and 3; they are less likely to be reduced. We also use the CUB library to sort data on GPUs.

Second, empirically we choose to iterate the innermost loop over only the first 16 values of j (line 8). This is because the rotations of prime cyclotomic vectors have larger norms. The expansion factor for prime cyclotomic lattices is discussed in [Sch13].

7 Experiments

For our experiments, we use a total of eight NVIDIA GeForce GTX TITAN X graphics cards. Four of these cards are installed on the main machine, while the other four are installed on a PCIe extension box.

We use the bases from the Ideal Lattice Challenge [Ide]. Since for dimension n, the prime cyclotomic polynomial has index $n+1$, as an example, we choose the basis for dimension 126 from the file `ideallatticedim126index127seed0.txt`. The input bases for Gauss Sieve are first reduced by BKZ with block size 30 and $\delta = 0.99$.

7.1 Parallel Efficiency

Here we show the parallel efficiency of the outer layer of our parallel framework in Fig. 1. The *(parallel) efficiency* for N GPUs is defined in [BNvdP14] as

$$E = \frac{\text{runtime for } N \text{ GPUs}}{\text{N} \cdot \text{runtime for 1 GPU}}.$$

For the dimension 108, the efficiency is 74%, 72%, 55% and 45% for 2, 4, 6 and 8 GPUs, respectively. However, the dimension is so low that the efficiency is quite low as the number of GPUs exceeds 6.

On the other hand, for dimension 112, the efficiency scales better with the number of GPUs. However, we do not yet have the running time for one GPU. If we base the efficiency on 2 GPUs, then the efficiency is the 86%, 81% and 74% for 4, 6 and 8 GPUs, respectively. We believe that in high enough dimensions, the efficiency of 8 GPUs will be more than 70%.

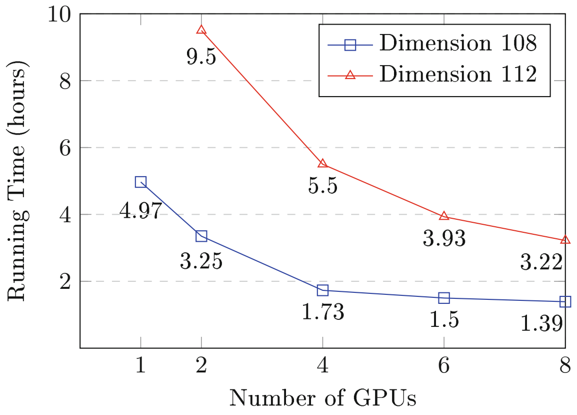

Fig. 1. Parallel efficiency: the numbers of GPUs versus running time and the exact running time is labelled below the node

7.2 Ideal Lattices Versus General Lattices

For general lattices, we use the bases from the SVP Challenge [Lat]. Our single-GPU implementation takes 9.3 h to solve the challenge of dimension 96. In contrast, the implementation from [IKMT14] requires 200 CPU-hour. That is, our single-GPU implementation on general lattices is 21.5 times faster than the CPU version.

For ideal lattices, our 4-GPU implementation requires 5 min to solve the challenge of dimension 96 and our single-GPU implementation requires 8.6 min. In contrast, the implementation from [IKMT14] requires 8 CPU-hours. That is, our 4-GPU (resp. single-GPU) implementation on general lattices is 96 (resp. 55.8) times faster than the CPU version. Note that the polynomial we use is prime-cyclotomic ($x^{96} + x^{95} + \cdots + 1$), which is more complicated than the trinomial polynomial ($x^{96} + x^{48} + 1$) used by [IKMT14].

Combining these two cases, the speed-up from using the property of prime-cyclotomic ideal lattices is $\frac{9.3 \text{ hrs}}{8.6 \text{ mins}} = 64.9$ in dimension 96. Applying the complexity estimation from [MV10], we estimate the ratio to be $\frac{9.3 \text{ hrs} \cdot 2^{0.52 \cdot 30}}{2734 \text{ hrs}} = 169$ in dimension 126 and $\frac{9.3 \text{ hrs} \cdot 2^{0.52 \cdot 34}}{6583 \text{ hrs}} = 297$ in dimension 130.

In contrast, [IKMT14] shows that the speed-up ratio of using the property of anti-cyclic ideal lattices is around 600 in dimension 128. This gives an evidence that the SVP over prime-cyclotomic ideal lattices is harder than over anti-cyclic ideal lattices by a factor of around 2.

7.3 Chronological Behavior

The chronological behavior of the sieving algorithms is studied intensively in [MV10, Sch11]. We can observe the same behavior in Fig. 2. (a) shows that the

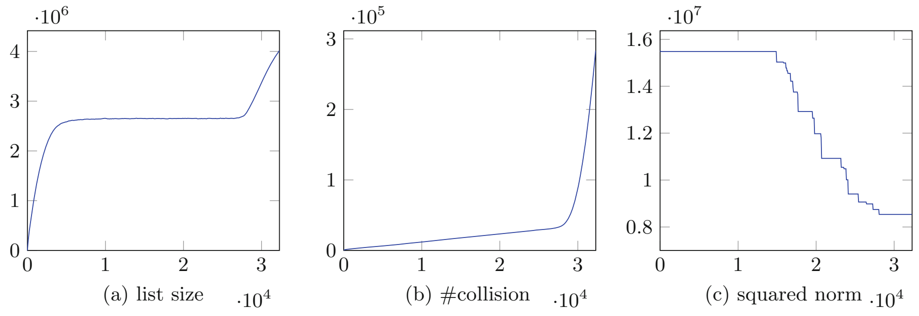

Fig. 2. Behavior of Gauss Sieve for dimension 126 with 8 GPUs: (a) the list size versus iteration; (b) the number of collisions versus iteration; (c) the squared norm of current shortest vector versus iteration

size of the list grows very fast in the beginning, later on it reaches a plateau, and finally when the shortest vector is found, it grows rapidly again. (b) shows the number of collision grows almost linearly but goes up very fast after the shortest vector is found. (c) shows the squared norm of the current shortest vector. The norm starts to drop half-way, and keeps descending until the shortest vector is found. One possible improvement is to reduce the basis by the current founded short vector, as in the work [FK15]. However, since the very first "shorter" vector only shows up half-way, the speed-up ratio by this method is limited by 2.

Our result in Table 1 is the fastest implementation of the Gauss Sieve algorithm so far. A rough space usage estimation is $2 \times 4 \times ListSize \times Dimension$. The factor 4 is due to the data type, 4-byte float, and the 2 is due to an extra buffer for sorting on the device. Therefore, it requires around 0.37, 2.59 and 4.35 GB of memory for dimension 112, 126 and 130, respectively.

Table 1. Results of ideal lattice challenge

Dimension	112	126	130
Number of vectors	444,341	2,759,903	4,490,083
Running time (GPU-hours)	32	2,734	6,583

7.4 Hardness Estimation

Finally, Fig. 3 compares our results with previous works. Obviously, our results are below the estimation of [LP11]. Even more, the slope of ours is flatter than theirs, which means that there is an exponential speed-up. Some of the data from the SVP and Ideal SVP Challenge is computed using an accelerated random sampling algorithm [FK15], but in higher dimensions (say, higher than 136), our

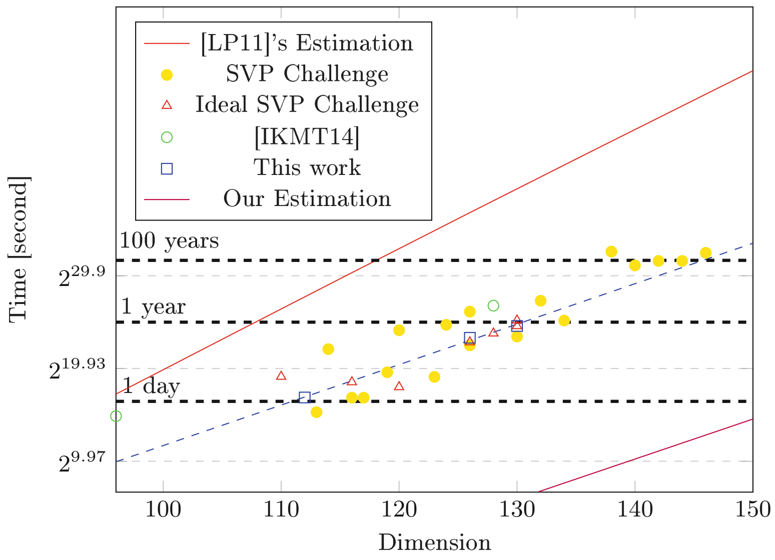

Fig. 3. Comparison of time: the dimension of SVP Challenge versus the running time

fitting curve is also below their results. This might imply that the running time of the Gauss Sieve algorithm grows quite slowly.

Fitting our data using least-square regression, we have $y = 2^{0.435x-31.8}$, as depicted in Fig. 3. To resist attacks using super powerful special-purpose hardware, our conservative model of SVP hardness in ideal lattices, with approximation factor 1.05, is

$$time(SVP^{Ideal}_{\alpha=1.05}) = 2^{0.43n-50}(\text{seconds})$$

However, we emphasize that the space complexity of the sieve algorithm is exponential, but estimation models of [LP11,CN11] are based on BKZ or BKZ 2.0, which requires only polynomial space. More precisely, our implementation requires $2^{0.19n+7.3}$ bytes of memory.

8 Conclusion

In this work, we propose the lifting technique for prime-cyclotomic ideal lattices, which accelerates the Gauss Sieve algorithm. Moreover, by applying a sequence of transformations described in Sect. 4, the cost to reduce two vectors with all of their rotations decreases from $O(n^3)$ to $O(n^2)$. We also designed and implemented a Gauss Sieve that includes these technique both on a single GPU and on several GPUs. Our implementation is more than 21.5 (resp. 55.8) times faster than the best prior known result on a single CPU core for general (resp. ideal) lattice. Finally, we give a reasonable model to estimate the running time of solving SVP in ideal lattices. Although our model requires an exponential space usage due to the natural property of sieving algorithms, it suggests a bound much lower than the previous model [LP11].

We will release the code to the public domain after we finish up with all the details such as providing a less hostile interface, doing a clean up, and so on.

Acknowledgement. Partially sponsored by MoST projects 105-2923-E-001-003-MY3 and 105-2221-E-001-020-MY3. We would also like to thank Dr. Shinsaku Kiyomoto of KDDI Research for the fruitful discussion.

References

[AD97] Ajtai, M., Dwork, C.: A public-key cryptosystem with worst-case/average-case equivalence. In: STOC 1997, pp. 284–293. ACM, New York (1997)

[Ajt97] Ajtai, M.: The shortest vector problem in l_2 is np-hard for randomized reductions. In: Electronic Colloquium on Computational Complexity (ECCC), vol. 4, no. 47 (1997)

[AKS01] Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: STOC 2001, pp. 601–610. ACM, New York (2001)

[BDGL16] Becker, A., Ducas, L., Gama, N., Laarhoven, T.: New directions in nearest neighbor searching with applications to lattice sieving. In: SODA 2016, pp. 10–24 (2016)

[BL16] Becker, A., Laarhoven, T.: Efficient (ideal) lattice sieving using cross-polytope LSH. In: Pointcheval, D., Nitaj, A., Rachidi, T. (eds.) AFRICACRYPT 2016. LNCS, vol. 9646, pp. 3–23. Springer, Heidelberg (2016). doi:10.1007/978-3-319-31517-1_1

[BNvdP14] Bos, J.W., Naehrig, M., van de Pol, J.: Sieving for shortest vectors in ideal lattices: a practical perspective. IACR Cryptology ePrint Archive 2014, 880 (2014)

[CN11] Chen, Y., Nguyen, P.Q.: BKZ 2.0: better lattice security estimates. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 1–20. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25385-0_1

[Cra96] Crandall, R.E.: Topics in Advanced Scientific Computation. Springer-Telos, New York (1996)

[CUD15] CUDA C programming guide 7.5 (2015). http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[FK15] Fukase, M., Kashiwabara, K.: An accelerated algorithm for solving SVP based on statistical analysis. JIP **23**(1), 67–80 (2015)

[Gau] Gauss Sieve implementation by panagiotis voulgaris. https://cseweb.ucsd.edu/~pvoulgar/impl.html

[Gen09] Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Annual ACM Symposium on Theory of Computing – STOC, pp. 169–178 (2009)

[GGH13] Garg, S., Gentry, C., Halevi, S.: Candidate multilinear maps from ideal lattices. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 1–17. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38348-9_1

[GNR10] Gama, N., Nguyen, P.Q., Regev, O.: Lattice enumeration using extreme pruning. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 257–278. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13190-5_13

[Ide] Idea Lattice Challenge. http://www.latticechallenge.org/ideallattice-challenge/

[IKMT14] Ishiguro, T., Kiyomoto, S., Miyake, Y., Takagi, T.: Parallel Gauss Sieve algorithm: solving the SVP challenge over a 128-dimensional ideal lattice. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 411–428. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54631-0_24

[KSD+11] Kuo, P.-C., Schneider, M., Dagdelen, Ö., Reichelt, J., Buchmann, J., Cheng, C.-M., Yang, B.-Y.: Extreme enumeration on GPU and in clouds. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 176–191. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23951-9_12

[Laa15] Laarhoven, T.: Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9215, pp. 3–22. Springer, Heidelberg (2015). doi:10.1007/978-3-662-47989-6_1

[Lat] SVP Challenge. http://www.latticechallenge.org/svp-challenge/

[LP11] Lindner, R., Peikert, C.: Better key sizes (and attacks) for LWE-based encryption. In: Kiayias, A. (ed.) CT-RSA 2011. LNCS, vol. 6558, pp. 319–339. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19074-2_21

[MBL15] Mariano, A., Bischof, C.H., Laarhoven, T.: Parallel (probable) lock-free hash sieve: a practical sieving algorithm for the SVP. ICPP **2015**, 590–599 (2015)

[MDB14] Mariano, A., Dagdelen, Ö., Bischof, C.: A comprehensive empirical comparison of parallel ListSieve and GaussSieve. In: Lopes, L., et al. (eds.) Euro-Par 2014. LNCS, vol. 8805, pp. 48–59. Springer, Heidelberg (2014). doi:10.1007/978-3-319-14325-5_5

[Mer] Duane (Nvidia Coorporation) Merrill. The CUB Library

[MS11] Milde, B., Schneider, M.: A parallel implementation of GaussSieve for the shortest vector problem in lattices. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 452–458. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23178-0_40

[MV10] Micciancio, D., Voulgaris, P.: Faster exponential time algorithms for the shortest vector problem. In: SODA 2010, pp. 1468–1480 (2010)

[NV08] Nguyen, P.Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. J. Math. Cryptol. **2**(2), 181–207 (2008)

[Sch11] Schneider, M.: Analysis of Gauss-Sieve for solving the shortest vector problem in lattices. In: Katoh, N., Kumar, A. (eds.) WALCOM 2011. LNCS, vol. 6552, pp. 89–97. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19094-0_11

[Sch13] Schneider, M.: Sieving for shortest vectors in ideal lattices. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) AFRICACRYPT 2013. LNCS, vol. 7918, pp. 375–391. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38553-7_22