The faster lattice sieving for SVP

Huibing Song¹, Chunxiang Gu and Yonghui Zheng

Henan Key Laboratory of Network Cryptography Technology, Zhengzhou 450002, China

¹Email: bingbling_song@163.com

Abstract. The security of lattice-based cryptography is based on the hardness of the difficult problems on lattice, especially the famous shortest vector problem. There are many famous heuristic lattice sieving algorithms to solve SVP, such as the Gauss Sieve, NV Sieve, which deal the full-rank \( n \)-dimensional lattice from start. Inspired by the idea of rank reduction, in this paper we present new technique on lattice sieving to make the algorithm solve the SVP faster. We split the basis of “bigger” lattice into several blocks according to some rule until the sub-lattice is small enough. Then we recursively sieved on these sub-lattices to get short vector lists. With the short vector lists, we can find the shortest vector when the algorithm recovered the original lattice. This lead to obviously speedup with the recursive procedure without extra space overhead. Compared with the Gauss Sieve, the new method converge faster, and achieved at least 70% acceleration.

1. Introduction

In recent years, the researches on quantum computers are deeper and deeper, which compels researchers to pay more attention on the development of quantum resistant (or post-quantum) cryptosystems. The lattice-based cryptography is one of the effectively quantum-resistant schemes proposed till now, which is concerned to be secure against quantum computers. The security of lattice-based cryptography is based on the hardness of lattice problems, the most famous of which is the shortest vector problem (SVP). The goal of SVP is to find the shortest non-zero vector in the lattice, where shortest is defined in terms of the Euclidean norm. As a result, the research on solving SVP is significantly important for the security of lattice-based cryptography.

SVP algorithms. Currently, the algorithms for solving the shortest vector problem are mainly divided into two categories: exact algorithm and approximation algorithm. The exact algorithm is a kind of reliable shortest vector searching algorithms theoretically, and the goal of the approximation algorithm is to search for a non-zero short vector that satisfies the established condition. These two kinds of algorithms complement each other when solving different dimensional SVP: the low-dimensional exact algorithm is often used as a subroutine called by the high-dimensional approximation algorithm, such as the enumeration [1] and sieving [2]; the approximation algorithm, such as the BKZ-reduced [3] and other lattice basis reduce algorithms [4], is needed for preprocessing to get the better lattice basis before the exact algorithm. Recently, the main approaches to solve the SVP are enumeration [1], sieving [2][5][6], the BKZ-reduced [3], and other SVP methods. The space complexity of enumeration [1] (the classical method of finding the shortest vector) is low, but the time complexity is super exponential in the lattice of dimension \( n \). In contrast, the heuristic sieving [2] sacrifices a certain space for acceleration, and has received more and more attention in recent years.
Lattice Sieving. During the last years, lattice sieving seems to become more efficient in practice and has drawn more attention. The main idea of sieve algorithms is to perform some kind of randomized enumeration of a smaller set. The AKS sieving [5] is the first provable lattice sieving proposed by Ajtai, Kumar, and Sivakumar in 2001. It keeps iterating the decrease of the lattice points space to find the shortest vector, which is the first time to reduce the time complexity to single exponential. But the space requirement of the AKS sieving algorithm is exponential in the lattice dimension which make this algorithm not practical. In 2008, Nguyen and Vidick [6] improved the AKS sieving and proposed the first heuristic lattice sieving algorithm based on AKS sieving, with time complexity $2^{0.415n+O(n)}$ and space complexity $2^{0.208n+O(n)}$. In 2011 and 2013, the improved two-level NV sieve [7] and the three-level sieve [8] based on NV sieve were proposed. In practice, the heuristic lattice sieving algorithms are more efficient. In 2010, Micciancio and Voulgaris [2] present the List Sieve and a practical heuristic variant called the Gauss Sieve algorithm, which performs well in practice compared with other variants. And the time and space complexity of Gauss sieving are $2^{0.415n+O(n)}$ and $2^{0.208n+O(n)}$, which still require exponential storage, but with much smaller constants. The Gauss Sieve algorithm is one of the most promising candidates for lattice sieving algorithms in practice, and the parallel algorithms [9] of Gauss Sieve have been widely studied. Many new techniques have been introduced to the lattice sieving, which lead to great improvement. Recently, near neighbour searching techniques have been used to accelerate heuristic sieving algorithms during the searching procedure. In 2015, Laarhoven first applied this technique on sieving in [10] named the Hash sieving, using the locality-sensitive hashing (LSH) to optimize the range of candidate vectors used for reductions, which obviously speeding up the search procedure of near vectors. Other different variants [11] are proposed gradually, using appropriate locality-sensitive hash (or filters) and more precisely parameters. Tuple lattice sieving, first proposed by Bai-Laarhoven-Stehle [12] in 2016, aimed to solve the main drawback of lattice sieving that the memory requirement is huge, by offering a trade-off between sieving and enumeration. In 2018, Thijs Laahoven [13] and Ducas [14] proposed different kinds of rank-reduction (or dimension-reduction) methods respectively, which solved the shortest vector problem of the original lattice by finding the short vector on the low-rank (of low-dimension) sub-lattice. In practice, the method of rank-reduction accelerated the sieving significantly. Inspired by this idea, we proposed a quick Block lattice sieving algorithm [15] in 2018, which divided the original lattice basis into several low-rank blocks and executed the sieving on each blocks independently to get many short vectors. As described in [16], Michael Walter explained how this kind method reduced the average complexity of exhaustive search. And in practice, this block method achieved about 7.1% speed-up to find the shortest vector on the original lattice.

Contributions. In this paper, we exhibit a variant of the Block lattice sieving and proposed a recursive block lattice sieving to solve the SVP, which can recursive call low-rank sub-lattice sieving on every round and performed well in practice. The recursive block lattice sieving uses the recursive idea of block-to-whole, where we divide the original $n$-dimensional lattice into several low-rank sub-lattices by dividing the initial lattice basis $B = (b_1, b_2, \ldots, b_n)$ into several blocks. To find the short vectors on these sub-lattices, we continue to divide the current round sub-lattices into smaller ones. Finally, by solving the problems on these low-rank sub-lattices, we use the information we got to find short vectors on the upper round lattice gradually. On the lattices of each level, we execute the sieving algorithm to get short vectors we need. By this recursion algorithm, we can find the shortest vector efficiently, which leads to about 70% speedup on average.

Outline. The paper is organized as follows. In Section 2, we give some notations and the related algorithms on lattice sieving. The new method of recursive block lattice sieving is described in Section 3. In Section 4, we present the result of our experiment and give a brief analysis. Finally, in Section 5 we concludes several aspects of recursive block lattice sieving.

2. Preliminaries
In this section, we begin with an overview of the basic notations of the lattice. Here are a few basic definitions.
2.1. Notations and basic definitions

**Definition 1. (Lattice)** Given $m$ linearly independent $n$-dimension vectors $\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_m \in \mathbb{R}^n$, the $n$-dimensional lattice generated by basis $B = (\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_m)$ is denoted as $\mathcal{L}(B)$

$$\mathcal{L} = \left\{ \sum_{i=1}^{m} x_i \mathbf{b}_i : x_i \in \mathbb{Z}, \text{ for } 1 \leq i \leq m \right\}$$

Lattices are discrete additive subgroups of $\mathbb{R}^n$, with a strong periodicity property. The basis of lattice is not unique. The vector set $B$ and $B'$ are both the basis of the same lattice $\mathcal{L}$, when $B$ is equal to $B'$ applied with a unimodular matrix. The integer $n$ is called the dimension of the lattice $\mathcal{L}$, and the rank of the lattice is $m$. Usually, we execute the sieving algorithm on the full-rank lattice (i.e. $m = n$).

**Definition 2. (Shortest Vector Problems)** Given a lattice basis $B$, find the shortest nonzero vector $\mathbf{v}$ in $\mathcal{L}(B)$.

$$\|\mathbf{v}\| = \min \{ \|\mathbf{u}\| : \mathbf{u} \in \mathcal{L}(B), \|\mathbf{v}\| \neq 0 \}$$

The first successive minimum $\lambda_1(\mathcal{L})$ is the length of a shortest vector of $\mathcal{L}(B)$. The volume of a lattice $\mathcal{L}(B)$ is denoted by $\text{vol}(\mathcal{L}) = \sqrt{\text{det}(B^TB)}$.

To analyze lattice algorithms, the heuristic assumptions are commonly used.

**Definition 3. (Gaussian Heuristic)** For a full rank lattice $\mathcal{L}(B) \subset \mathbb{R}^n$, the first minimum $\lambda_1(\mathcal{L})$ is expected as

$$\lambda_1(\mathcal{L}) = \frac{n}{2\pi e} \cdot \text{vol}(\mathcal{L})^{1/n}$$

For an $n$-dimension lattice $\mathcal{L}$ of $\text{vol}(\mathcal{L}) = 1$, the $\lambda_1(\mathcal{L}) = \sqrt{n/2\pi e}$.

The basis-reduced algorithm is the main method for solving the approximate lattice problems, and it is also an important tool for preprocessing of the exact SVP algorithms. The reduced basis refers to the lattice basis owning some good properties, such as the short length of basis vectors or high degree of orthogonality. Now we will introduce the classical LLL-reduced basis [7]:

**Definition 4. (LLL reduced basis)**[17] Given lattice basis $B = (\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n)$, the Gram-Schmidt matrix $\mu = (\mu_{ij})$, $1 \leq j < i \leq n$, and Gram-Schmidt orthogonalization basis $B^* = (\mathbf{b}_1^*, \mathbf{b}_2^*, ..., \mathbf{b}_n^*)$, we define $B$ as the $\delta$-LLL ($1/4 < \delta < 1$) reduced basis, if $B$ is satisfied:

1. $|\mu_{ij}| \leq 1/2$.  
2. $\|\mathbf{b}_i^* + \mu_{i-1,j} \mathbf{b}_{i-1}^*\|^2 \geq \delta \|\mathbf{b}_{i-1}^*\|^2$.

The LLL-reduced basis vectors are short and close to orthogonal in directions. As input basis, it can effectively reduce the complexity of the SVP algorithms.

2.2. Lattice sieving algorithms

In this part, the different classical lattice sieving algorithms are as shown below.

**The Nguyen-Vidick Sieving.** The heuristic version sieve presented by Nguyen and Vidick in [6] is the first practical instantiation after the idea of lattice sieving introduced in [5]. The algorithm collects the short vectors by directly sieving the sampled lattice vectors. Firstly, sample vectors over the whole lattice. And then apply the sieve on the vector list iteratively to produce a new list with fewer and shorter lattice vectors until we find the shortest vector of the lattice. The procedure of sieve is pairwise combining the vectors of the input list, and judging if any of the sums or differences are shorter than the original vectors. And the new produced short vectors are thrown to the list which is prepared for the next iteration. At the same time, the long list vectors are discarded. This process of throwing away lots of vectors in the lattice is wasteful, therefore the NV sieve is not practical.

**The Gauss Sieving.** The Gauss Sieve algorithm presented by Micciancio and Voulgaris in 2010 [2] is commonly used in practice. The algorithm iteratively builds a longer and longer list $L$ of lattice vectors, and the lengths of the list vectors will be reduced in the process. Ultimately, we will find the shortest vector of the lattice in the list $L$ at some point. The Gauss Sieve is described in Algorithm 1. The reductions in Line 5 and 6 follow the rule:

Reduce $\mathbf{u}_1$ with $\mathbf{u}_2$ : if $\|\mathbf{u}_1 \pm \mathbf{u}_2\| < \|\mathbf{u}_1\|$ then $\mathbf{u}_1 \leftarrow \mathbf{u}_1 \pm \mathbf{u}_2$. 

Besides reducing new lattice points \( v \) against the points already in the list \( L \), the algorithm also reduce the points in \( L \) against \( v \), and against each other. This implies that two list vectors \( \mathbf{w}_1, \mathbf{w}_2 \in L \) always have an angle of at least \( \pi/3 \). As the result, the size of the list is bounded by the kissing constant \( \tau_n \) in dimension \( n \), which is defined as the highest number of points that can be placed on a sphere, while keeping the minimal angle between any two points at least \( \pi/3 \).

Algorithm 1. The standard Gauss Sieve algorithm.

```
Algorithm 1: The standard Gauss Sieve algorithm

Require: \( n \)-dimensional lattice basis \( B \), the collision number \( c \).
Ensure: the shortest vector \( v \).
1: Initialize an empty list \( L \leftarrow \emptyset \) and an empty stack \( S \leftarrow \emptyset \)
2: Initialize COLLISIONS+ \( \leftarrow 0 \)
3: while true do
4:     Get a vector \( v \) from the stack \( S \) or sample a new one from \( L(b_1, b_2, \ldots, b_n) \)
5:     Reduce \( v \) with all \( \mathbf{w} \in L \)
6:     Reduce all \( \mathbf{w} \in L \) with \( v \)
7:     Move reduced vectors \( \mathbf{w} \in L \) from the list \( L \) to the stack \( S \)
8:     if \( \mathbf{v} \) has not changed then
9:         Add \( \mathbf{v} \) to the list \( L \)
10:    else
11:        if \( \mathbf{v} \neq \emptyset \) then
12:            Add \( \mathbf{v} \) to the stack \( S \)
13:        else
14:            COLLISIONS++
15:            if COLLISIONS== c then
16:                return argmin_{\mathbf{x} \in L} ||\mathbf{x}||
17:            end if
18:        end if
19:    end if
20: end while
```

Figure 1. Algorithm design of Block Lattice Sieve.

The Block Lattice Sieving. In 2018, we proposed the Block lattice sieving [15] (see in Algorithm 2) to solve SVP, and the result is satisfactory in practice. As described in Figure 1, to find the shortest vector of the full-rank lattice, we get many short vectors on low-rank sub-lattices. Firstly, we divide the original full-rank lattice \( L(B) \) into \( k \) sub-lattices \( L(B_i) \) where the rank of them is \( m \). Note that, basis vectors of sub-lattices are all from the original lattice basis vectors. Next, we execute the sieving algorithm on these sub-lattices to get relevant lists containing many short vectors, which will eventually contribute to finding the exact solution faster. Finally, we combine all bases \( B_i \) together to recover the original \( n \)-dimensional lattice and execute the last sieving on it to find the shortest vector of the \( n \)-dimension lattice.

Other variant of lattice sieving. During the last few years, lots of extensions and variants of the heuristic sieves have been researched. The near neighbor searching technique [10][18][19] has been
applied in Line 5-6 of Algorithm 1, which significantly reduce the comparison space for pairwise reduction, so can obtain an obvious speedup in the searches. More recently, the tuple sieve is proposed \[11\][12] to reduce the memory of lattice sieving, considering a larger number of vectors \( \mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_k \in L \) from the list to form short combinations \( \mathbf{v} \) with \( \mathbf{v} \pm \mathbf{w}_1 \pm \mathbf{w}_2 \pm \cdots \pm \mathbf{w}_k \).

**Algorithm 2.** The Block Sieve algorithm.

3. The recursive block lattice sieve

In this section, we will describe our recursion algorithm for solving the shortest vector problem.

Given a full-rank lattice basis, classic lattice sieving approaches always deal with vectors from the full-dimensional lattice space from the start. We need around \( 2^{0.21n+O(n)} \) vectors to saturate the space and make significant progress in obtaining shorter vectors because of the pairwise-reductions between lattice vectors. It means that only when the list size approaches \( 2^{0.21n+O(n)} \), can the approximate shorter vectors be find. Intuitionally, this means the time complexity to find any shorter vectors at all is at least \( 2^{0.42n+O(n)} \). But just like BKZ \[3\], dealing with the low-rank lattice may help finding shorter vectors easier and faster.

3.1. The idea of recursive block lattice sieving

Just like the Block lattice sieving, to find the shortest vector of the full-rank lattice, we need to get many short vectors on low-rank sub-lattices. The block-divided method of Block lattice sieving is coarse-grained, which cannot generate high quality short vectors in low-rank lattices. To improve the quality of list vectors on sub-lattices, we use the recursive approach on every blocks.

The idea of Recursive Block lattice sieving is straightforward: when the block size of sub-lattices is "big", it is supposed to split the current block into several smaller ones until the block size satisfied the requirement. Note that, for different blocks produced by the same higher-rank block lattice, there are overlapping part of the lattice basis vectors. After that, we will sieve on each small block lattices to get the short vectors, and return the merged list to the upper round blocks until recover the original lattice, where we can find the shortest vector. In other words, we recursively call the split algorithm on each round, until the block size is "small" enough.
3.2. Combining with the gauss sieve

We need to make the following modifications on Gauss Sieve before applying the recursive block lattice sieving to it. First, we start with the sub-lattices of the original lattice, which are spanned by the relevant block-basis vectors. Then we run the sieving on each round sub-lattices to get the relevant short vectors in list $L_{ij}$, until we reach the stopping criterion (a certain number of "collisions"). Next, we combine all lists $L_i = L_{i,1} \cup L_{i,2} \cup \cdots \cup L_{i,k}$ on each round and execute the same procedure on the upper round with the set $L_i$ until reach the original lattice to find an exact solution. The result of applying the recursive block lattice sieving to the Modified GaussSieve is described in Algorithm 3, where the main modifications are (1) vectors are sampled from sub-lattices, and (2) the rank counter is increased and the collision counter is reset when we reach 100 collisions.

![Algorithm 3](image-url)

Algorithm 3. The Recursive Block Lattice Sieving.

As shown in Algorithm 3, when given a lattice, the recursive function RESIEVE first judge whether the rank of current lattice is bigger than $d$.

Step 3: if \( \text{rank} > d \), split the lattice basis $B_{r-1}$ into $k$ different blocks $B_{r,1}, B_{r,2}, \ldots, B_{r,k}$ and go into the next Round $r$. Notice the basis vectors will appear repeatedly in different blocks (on the same round).

Split procedure (Step 5-7): Continue to judge the size of block lattice rank, until the rank of every block is smaller than $d$ (\( \text{rank} < d \)), and then the split procedure will be end.

Sieve procedure (Step 8-16): On the smallest blocks, the algorithm recall the sieving with the sampled vectors, and get many short vector lists from each small blocks. And then, merge all the lists on this round $L_r = L_{r,1} \cup L_{r,2} \cup \cdots \cup L_{r,k}$. Input the list $L_r$ into the sieving on round $r - 1$, continue sieving until recover the original lattice and find the shortest vector.

Step 17: if \( \text{rank} < d \), directly execute the Modified GaussSieve, and get the shortest vector and several vector lists.

According to our experiment results, we find that sieving on lattice, where the rank is below 30 or 40, generally finished immediately. So setting the any initial rank below 40 will lead to the similar results. As a result, for simplicity we set $d = 40$ as the minimum block size, and the size of the block bigger than 40 will be thought "big".

Specifically, to solve the SVP on $m$-dimension lattice, we first split the lattice basis into $k$ blocks, where the size of each block is $m - n$, where $n$ is the step size of each round. If $m - n > 40$, the blocks of this round will be split in the same way and the round counter will be added, until the block
size is less than 40 on some round. After sieving on the last round, we need to merge all the lists getting from each block on this round. Return the merged list to the upper round and do the sieving, until we recover the original $n$-dimension lattice, and find the shortest vector. To make it clear, Figure 2 shown an example of recursive sieving on 70-dimension lattice, with the step $n = 10$, block numbers $\text{num}=3$, minimum block size $d = 40$.

First step, as shown by the dotted line, the split procedure divided the original lattice into several blocks on each round, until the rank $\leq 40$. Second step, the sieving procedure was described vividly by green lines. On Round 3, the sieving sampled the random vectors on the smallest blocks, and by sieving got the lists $L_3 = L_{3,1} \cup L_{3,2} \cup L_{3,3}$ as shown by the red circle. And on Round 2, the algorithm do the sieving on the first block with the list $L_3$. By the same way, the shortest vector will be found on the Round 0.

3.3. Other sieving algorithms

Naturally, the recursive idea can be applied to most other sieving algorithms as well. Combining with the NV Sieve, Hash Sieve and the Tuple Sieve are both feasible. Just like the near neighbor searching, which only affect the searching procedure for finding reducing pairs of vectors, can combine the recursive sieving well. Also, we can ran tuple sieving on sub-lattices to find the short vectors, after that we skip to the upper round.

4. Experiments

In this section, we describe and conduct the experiments of our recursive block sieving relying on the fplll library [20]. In Section 4.2 we give a brief analysis about the results on the space and time requirements.
4.1. Experiment setting
Based on the fplll library [20], we implement our algorithm with Intel(R) Xeon(R) CPU E5-2620 v4 @2.10GHz. Furthermore, we compared our results with standard sieving approaches and the Progressive sieving [13] proposed in 2018, where we use the Gauss Sieve [2] without near neighbor searching or tuple method as the baseline. The termination condition of all experiments is a bound on the number of "collisions" (two vectors are reduced to the all-zero vector). Similar to the Gauss Sieve [2], we still use the Klein’s sampler. All of the experiments were performed on lattices generated from the SVP challenge [21], where the input lattice basis vectors were reduced by the LLL algorithm in fplll [20] first.

4.2. Results and analysis
The lattice sieving algorithms were required to search over all pairs of vectors in the list heuristically (when not using the near neighbour searching techniques), which lead to a quadratic time complexity of the list size. For the recursive block sieving, this argument still hold, and the asymptotic improvement of our algorithm will not be visible in the leading time or memory exponents. However, there are rather large hidden order terms experimentally, and these may become smaller in the recursive block lattice sieving.

We evaluate our algorithm and compare it with standard Gauss Sieve and Progressive sieving by experiments to get insight into the improvement. First, we produced 20 sets random lattice bases of each dimension with different seeds, using the basis generated algorithm from fplll [20]. We conducted 20 sets experiments on Gauss Sieve, Recursive Block lattice sieving and Progressive sieving on lattices of dimension 40 to 70, in order to get reliable complexity estimates for arbitrary dimensions. The experiments are certainly not exhaustive, but we believe they are sufficient to show the practical performance of our algorithm, at least compared with previous lattice sieving techniques.

The time complexity. The results of the time complexity comparison are displayed in Figure 3. As shown in Figure 3(a), we can get the idea how recursive block sieving differs from classical sieving in practice. These results clearly shown, compared with the GaussSieve, a large decrease in the time complexities for sieving in arbitrary dimensions, and the speed achieved at least 70% acceleration on average.

And as can be found in Figure 3(b), the results clearly described that the time of recursive block lattice was better than progressive sieving. The progressive sieve gradually introduced new basis vectors only when the sieve has stabilized on the previous basis vectors. While in our algorithm, the step between near round sub-lattices is adjustable, which lead to less rounds to generate new sub-lattices and can reduce the total comparison of vector pairs.

![Comparison of time](image1.png)

(a) The time compared with Gauss Sieve

![Comparison of time](image2.png)

(b) The time compared with progressive sieving

Figure 3. The comparison of time complexity.

The recursive block sieving benefits greatly when given a relaxed termination condition on sub-lattices. On lower-rank sub-lattices, especially with the more reduced input basis, the algorithm will already find shorter lattice vectors and add them to the list L. There are more short vectors in the list,
which means the new sampled vectors are supposed to be reduced faster and more efficiently. Since it takes the classical sieving much longer time to find (approximate) short lattice vectors. Besides, we apply the recursive methods on every block lattices, which can take full use of the short vectors getting from each low-rank sub-lattices, which makes it perform better than the progressive sieving. We believe this mostly explains the bigger time improvements to the recursive block lattice sieving.

As we all know, during the sieving procedure, huge space are needed to store the list vectors, which is the main aspect of space requirement in sieving algorithms. To evaluate the size complexity of the sieving, we measure the maximum list size. Because the size of list is changing during the reductions, as the list points collide with each other, and the collisions will shrink the size of the list. On the other hand, we show the space complexity comparison in Figure 4. The main bottleneck in sieving is the great requirements of memory, which means the new algorithms are supposed not bring extra memory overhead. As described in Figure 3, the space complexity of recursive block sieving is much less compared with both the classical Gauss Sieve and progressive method, which meets our expectations. To release the space requirement, the recursive sieve will throw away the list vectors got from sub-lattices when finished the reduction on this round. As a result, these block lattices did not lead to extra space coast.

Summarizing, the experiments results proved our new recursive method can significantly accelerate the algorithm without extra space overhead, especially on the higher dimensional lattices.

On the other aspect, the sampler procedure is important before the sieving part. The quality and quantity of the sampler vectors influent the final results a lot. But we all know that choosing the best parameters for the sampler to get the best performance in classical sieving is a complicated procedure. Using the recursive block method, we can decrease the dependence on sampler algorithms for sieving, although the parameters choosing still influent the performance of our algorithm. Due to the list having many short vectors early on, the long sampled vectors can be reduced in length faster and more efficiently, which means sampling longer vectors is less of an issue.

![Comparison of list size](image)

(a) The size compared with Gauss Sieve        (b) The size compared with progressive sieving

**Figure 4.** The comparison of space complexity.

5. Conclusions

Combining the idea of rank reduction with the classical lattice sieving, in this paper, we proposed the recursive block lattice sieving to solve the SVP, and implement the algorithm with the analysis. The main bottleneck in sieving is the memory requirements, and working on low-rank lattices requires fewer vectors to make progress without extra space overhead. Especially with the recursive method, we can take full use of the vectors got from the low-rank lattices to achieve deeper reduction, allowing great acceleration achieved without any sampled vectors waste.

As we can see, the recursive block lattice sieving has a better (experimental) performance than classic sieving methods in many aspects. First of all, it makes sieving become faster on convergence with less memory requirements. Second, the recursive block lattice sieving less rely on the parameter choices of the sampler, and the high quality of the input lattice basis can benefit a lot.
On the other hand, there are still many works need us research to improve the recursive block sieve. Obviously, the parallelism can be applied to this new algorithm to decrease the runtime. And the theoretical analysis of complexity still needs to be discussed in depth, which will make our research more complete.

References
[1] Gama N, Nguyen P Q, Regev O 2010 Lattice enumeration using extreme pruning[C]// Annual International Conference on the Theory and Applications of Cryptographic Techniques Springer, Berlin, Heidelberg, 257-278
[2] Micciancio D, Voulgaris P 2010 Faster Exponential Time Algorithms for the Shortest Vector Problem,[C]// Acm-siam Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics
[3] Chen Y, Nguyen P Q 2011 BKZ 2.0: Better Lattice Security Estimates[C]// Advances in Cryptology-asiacrypt-international Conference on the Theory & Application of Cryptology & Information Security DBLP
[4] Schnorr C P, Euchner M 1994 Lattice basis reduction: Improved practical algorithms and solving subset sum problems[J]. Mathematical programming 66(1-3) 181-199
[5] Miklós Ajtai, Kumar R, Sivakumar D 2001 A sieve algorithm for the shortest lattice vector problem[C]// ACM
[6] Nguyen P Q, Vidick T 2008 Sieve algorithms for the shortest vector problem are practical[J]. Journal of Mathematical Cryptology 2(2) 181-207
[7] Wang X, Liu M, Tian C , et al. 2011 Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem[C]// Acm Symposium on Information ACM
[8] Zhang F, Pan Y, Hu G 2013 A Three-Level Sieve Algorithm for the Shortest Vector Problem[M]// Selected Areas in Cryptography-SAC 2013 Springer Berlin Heidelberg
[9] Bos J W, Naehrig M, Van De Pol J 2014 Sieving for Shortest Vectors in Ideal Lattices: a Practical Perspective[J]. IACR Cryptology ePrint Archive 2014 880
[10] Laarhoven T 2015 Sieving for shortest vectors in lattices using angular locality-sensitive hashing.[M]// Advances in Cryptology-CRYPTO 2015 Springer Berlin Heidelberg
[11] Laarhoven T 2017 Faster tuple lattice sieving using spherical locality-sensitive filters[J]. arXiv preprint arXiv:1705.02828
[12] Shi B, Laarhoven T, Stehlé D 2016 Tuple lattice sieving[J]. Lms Journal of Computation & Mathematics 19(A) 146-162
[13] Laarhoven T, Mariano A 2018 Progressive lattice sieving[C]// International Conference on Post-Quantum Cryptography Springer, Cham, 2018 292-311
[14] Ducas L 2018 Shortest Vector from Lattice Sieving: A Few Dimensions for Free[C]// International Conference on the Theory & Applications of Cryptographic Techniques Springer, Cham
[15] Song H B, Gu C X, Zheng Y H, Sun Z D. A Quick Block Lattice Sieving to Solve SVP[J]. Journal of Information Engineering University, in press
[16] Walter M 2015 Lattice point enumeration on block reduced bases[C]// International Conference on Information Theoretic Security Springer, Cham 2015 269-282.
[17] Lenstra A K, Lenstra Jr H W, Lovász L 1982 Factoring polynomials with rational coefficients[J]. Mathematische Annalen 261(4) 515-534
[18] Becker A, Ducas L, Gama N, et al. 2016 New directions in nearest neighbor searching with applications to lattice sieving[C]// Twenty-seventh Acm-siam Symposium on Discrete Algorithms Society for Industrial and Applied Mathematics
[19] Becker A, Laarhoven T 2015 Efficient (Ideal) Lattice Sieving Using Cross-Polytope LSH[J]
[20] The FPLLL development team. Fplll, a lattice reduction library. Available at https://github.com/fplll/fplll, 2016
[21] Lindner R. The lattice challenge homepage. https://www.latticechallenge.org/