A Survey of Algorithms for Addressing the Shortest Vector Problem (SVP)

Errui He, Tianyu Xu, Mengsi Wu, Jiageng Chen, Shixiong Yao, and Pei Li(✉)

Central China Normal University, Wuhan 430079, Hubei, China
peili@ccnu.edu.cn

Abstract. Lattice-based encryption schemes derive their security from the presumed complexity of solving the Shortest Vector Problem (SVP) within the lattice structure. Numerous algorithms have been proposed to address the SVP, targeting both exact and approximate solutions. However, it is noteworthy that the time complexity associated with these algorithms predominantly exceeds polynomial bounds. This paper succinctly delineates these algorithms while providing a comprehensive summary of existing parallel implementations of sieving and enumeration methods. Furthermore, it introduces distinguished instances from the Hall of Fame of the SVP challenge.

Keywords: Lattice reduction · LLL · BKZ · Enumeration · Sieving · Parallel computing

1 Introduction

In recent years, with the rapid development of quantum computer technology, the security of traditional cryptography has been threatened. Therefore, governments and research institutions have begun to study secure cryptographic algorithms under quantum computing models. Lattice-based cryptography has obvious advantages in security, public-private key size, and resistance to quantum attacks. Therefore, it is expected to replace existing cryptographic algorithms and become a new cryptographic standard in the future.

In 1996, Ajtai gave a specification proof that the unique shortest vector problem (SVP) in lattices is under worst-case to that of the small integer solutions (SIS) problem under average-case [1]. This proof can reduce the difficult problem in the lattice-based cryptography under worst-case to the a class of random lattice problems, so the lattice-based cryptosystem can provide a worst-case security proof.

A major factor in the security of lattice-based cryptographic protocols is the difficulty of SVP in the lattice.This problem is crucial in lattice-based cryptography. There are many solutions to SVP problems, such as BKZ, enumeration, and

J. Chen and Z. Xia (Eds.), BlockTEA 2023, LNICST 577, pp. 57–76, 2024.
https://doi.org/10.1007/978-3-031-60037-1_4

sieving. In addition to being used separately, enumeration and sieving can also be embedded in the BKZ algorithm as sub-algorithms. The time complexity of the BKZ algorithm mainly depends on the time complexity of the sub-algorithm. The time complexity of the enumeration algorithm is $2^{\Omega(n)}$, and the time complexity of sieving algorithm is $2^{\Theta(n)}$, where n is the dimension of the lattice. These algorithms are super-polynomial time complexity, and how to accelerate the algorithm has become an important problem.

The enumeration is an exhaustive search algorithm used to find the shortest vector in a lattice. It achieves this by traversing all lattice points within an n-dimensional hypersphere centered at the origin with a radius of R. Practical enumeration can be viewed as traversing an enumeration tree, where each leaf node represents a lattice point. Pruning is a prevalent technique in enumeration, achieved by applying specific parameters to trim subtrees. While this approach introduces a notion of failure rate, it substantially reduces the search space, accelerating the algorithm's pace. Enumeration is often utilized as subroutines within BKZ, as they perform well in low dimensions but exhibit poor performance in high dimensions due to their super-exponential time complexity.

In 2001, [2] proposed the AKS Sieve, which was the first sieving algorithm. Apart from AKS Sieve, there is another sieving algorithm called the MV Sieve, which has a different structure. The ListSieve, introduced by [31], was the first sieving algorithm similar to MV. Both the AKS Sieve and the ListSieve are theoretical algorithms. Researchers subsequently developed heuristic versions based on these theoretical algorithms: NV Sieve [35] and GaussSieve [31].

Over the past decade, researchers have made various improvements to these sieving algorithms. The Level Sieve proposed by [41,45] takes the NV Sieve as the starting point. The Tuple Sieve proposed by [7] is based on the ListSieve. [34] proposed the Linear Sieve. The classic algorithm with the lowest complexity among heuristic algorithms is the LDSieve proposed by [8]. In addition, locality-sensitive hashing technology [23], locality-sensitive filtering technology [8] and quantum Grover search algorithm [25] are all used to speed up the traversal search process in the sieving algorithm. From the actual implementation effect level, the progressive sieving algorithm [24], sub-sieve algorithm [13] and G6K [3] optimize the actual execution effect of the sieving algorithm by using the rank-reduction technique. [27] also proposed a framework for the k-sieve algorithm with less memory usage. The sieving can also be used as a subroutine to improve the efficiency of BKZ [42].

This paper introduces the BKZ algorithm and summarizes the existing parallel implementations of enumeration and sieving. Table 1

2 Preliminaries

Notation. We let $\mathbb{Z}$ denotes the set of integers, $\mathbb{Q}$ denotes the set of rational numbers, and $\mathbb{R}$ denotes the set of real numbers. The nearest integer to a real number a, denoted by $\lceil a \rfloor$. For any real number a in the set $\mathbb{R}$, $|a|$ signifies the absolute value of a. It's noteworthy that all vectors are treated as column vectors.

Table 1. The parallel algorithm mentioned in this paper.

Algorithm	Author	Conference/Journal	Year	Type
HSB+10 [19]	Hermans et al.	AFRICACRYPT	2010	ENUM
DS10 [12]	Dagdelen and Schneider	Euro-Par	2010	ENUM
KSD+11 [22]	Kuo et al.	CHES	2011	ENUM
CMP16 [11]	Correia et al.	PDP	2016	ENUM
p3Enum [10]	Burger et al.	ICCS	2019	ENUM
p3EnumOpt [9]	Burger et al.	HPCS	2019	ENUM
MAP-SVP [39]	Tateiwa et al.	SC	2020	ENUM
EBD+21 [15]	Esseissah et al.	Scientific Programming	2021	ENUM
MS11 [33]	Milde et al.	PaCT	2011	Sieving
IKM+14 [20]	Ishiguro et al.	PKC	2014	Sieving
MTB14 [30]	Mariano et al.	Euro-Par	2014	Sieving
MBL15 [28]	Mariano et al.	ICPP	2015	Sieving
MLB17 [29]	Mariano et al.	PDP	2017	Sieving
AG20 [4]	Andrzejczak et al.	ICA3PP	2020	Sieving
DSW+21 [14]	Ducas et al.	EUROCRYPT	2021	Sieving

The Euclidean norm of a vector $\mathbf{a}$ belonging to the space $\mathbb{R}^m$ is represented as $\|\mathbf{a}\|$. Additionally, the inner product of vectors $\mathbf{a}$ and $\mathbf{b}$, both belonging to the space $\mathbb{R}^m$, is represented by $\langle \mathbf{a}, \mathbf{b} \rangle$.

Lattice. A lattice $\mathcal{L} = \{\mathbf{Bx} : \mathbf{x} \in \mathbb{Z}^n\}$ is denoted as $\mathcal{L}(\mathbf{B})$, where $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\} \in \mathbb{R}^{m*n}$ is an ordered group of linear independent vectors, and $\mathbf{B}$ is called the basis of the lattice.

Gram-Schmidt Orthogonalisation. For a linearly independent ordered vector group $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\} \in \mathbb{R}^{m*n}$, its Gram-Schmidt Orthogonalisation (GSO) set $\mathbf{B}^* = \{\mathbf{b}_1^*, \mathbf{b}_2^*, \ldots, \mathbf{b}_n^*\}$ is given by the following definition.

- $\mathbf{b}_1^* = \mathbf{b}_1$
- for $i > 1$, $\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^*$

where, for $1 \leq i \leq j \leq n$, the GSO coefficient is calculated by the following formula:

$$\mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}$$

Obviously, for all $1 \leq i \leq n$, $\mu_{i,i} = 1$.

Orthogonal Projections. For a vector $\mathbf{v} \in \mathcal{L}(B)$, its projections $\pi_i(\mathbf{v})$ are defined for $1 \leq i \leq n$ as follows:

- $\pi_1(\mathbf{v}) = \mathbf{v}$
- for $2 \leq i \leq n$, $\pi_i(\mathbf{v})$ is the orthogonal projection of $\mathbf{v}$ to $\text{span}(\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_{i-1})^{\perp}$.

The projection $\pi_1(\mathbf{v})$ is written as follows:

$$\pi_i(\mathbf{v}) = \mathrm{v} - \sum_{j=1}^{i-1} \frac{\langle \mathbf{v}, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}$$

In particular, when the vector $\mathbf{v}$ is the basis vector $\mathbf{b}_k$ of the lattice, it can be written as follows:

$$\pi_i(\mathbf{b}_k) = \mathrm{b}_k - \sum_{j=1}^{i-1} \mu_{k,j} \mathbf{b}_j^* = \mathrm{b}_k^* + \sum_{j=i}^{k-1} \mu_{k,j} \mathbf{b}_j^*$$

In the simplest scenario, $\pi_i(\mathrm{b}_i) = \mathrm{b}_i^*$.

Shortest Vector Problem (SVP). Let $\lambda_1, \ldots, \lambda_n$ denote the successive minima of lattice $\mathcal{L}$, $\lambda_i = \lambda_i(\mathcal{L})$ is defined as the smallest radius r of a ball that is centred at the origin and which contains r linearly independent lattice vectors.

Definition 1. Shortest Vector Problem (SVP). Given a lattice basis $\mathbf{B}$, the task is to find the shortest non-zero lattice vector, denoted as $\mathbf{v} \in \mathcal{L}(\mathbf{B})$, such that its norm satisfies $\|v\| = \lambda_1(\mathcal{L}(\mathbf{B}))$.

Ordering of Basis Vectors. Consider S_n as the group of permutations of elements in $\{1, 2, \ldots, n\}$. For $\sigma \in S_n$, define $\sigma(\mathbf{B}) = \{\mathbf{b}_{\sigma(1)}, \mathbf{b}_{\sigma(2)}, \ldots, \mathbf{b}_{\sigma(n)}\}$ as a vector ordering of $\mathbf{B}$. In particular, for $1 \leq i < k \leq n$, we define the following operation

$$\sigma_{i,k}(j) = \begin{cases} j, & j < i \text{ or } j > k \\ k, & j = i \\ j-1, & i+1 \leq j \leq k \end{cases}$$

For lattice basis $\mathbf{B}$, the operation is as follows:

$$\sigma_{i,k}(\mathbf{B}) = \{\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{b}_k, \mathbf{b}_i, \ldots, \mathbf{b}_{k-1}, \mathbf{b}_{k+1}, \ldots, \mathbf{b}_n\}$$

where $\mathbf{b}_k$ is inserted between $\mathbf{b}_{i-1}$ and $\mathbf{b}_i$, $\mathbf{b}_i, \ldots, \mathbf{b}_{k-1}$ moved one bit back.

3 Blockwise Korkine Zolotarev (BKZ) Reduction

Given the lattice basis $\mathbf{B}$, we can calculate its GSO matrix $\mathbf{B}^*$ and its GSO coefficients $\mu_{i,j}$ in polynomial time.

3.1 Size Reduction

Definition 2. (**Size-reduced basis**) A basis $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\} \in \mathbb{R}^{m*n}$ is considered size reduced if for all $1 \leq j < i \leq n$, $|\mu_{i,j}| \leq \frac{1}{2}$.

Algorithm 1: Size-reduction algorithm for k-column vector of basis $\mathbf{B}$

Input: A basis $\mathbf{B}$ along with its Gram-Schmidt orthogonalization (GSO) coefficients $\mu_{i,j}$, and an index k.

Output: A basis $\mathbf{B}^{'}$ where the kth column vector $\mathbf{b}_k^{'}$ satisfies size-ruduced, and the updated coefficients $\mu_{k,j}$

```
for j = k − 1, ..., 1 do
    if |μ_{k,j}| > 1/2 then
        b_k ← b_k − ⌈μ_{k,j}⌋ b_i
        for i = 1, ..., j do
            μ_{k,i} ← μ_{k,i} − ⌈μ_{k,j}⌋ μ_{j,i}
return Basis B' where the kth column vector b'_k satisfies size-ruduced, and the
updated coefficients μ_{k,j}.
```

Algorithm 1 gives the size-reduced operation. When we set $\mu_{k,j} \leftarrow \mu_{k,j} - \lceil \mu_{k,j} \rfloor \mu_{j,j}$, we get $|\mu_{k,j}| \leq \frac{1}{2}$ ($\mu_{j,j} = 1$). For consistency, we need to update the value of $\mathbf{b}_k$.

$$\mu_{k,j}^{'} = \mu_{k,j} - \lceil \mu_{k,j} \rfloor \mu_{j,j} = \frac{\langle \mathbf{b}_k, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle} - \lceil \mu_{k,j} \rfloor \frac{\langle \mathbf{b}_j, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle} = \frac{\langle \mathbf{b}_k - \lceil \mu_{k,j} \rfloor \mathbf{b}_j, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}$$

Then we set $\mathbf{b}_k^{'} \leftarrow \mathbf{b}_k - \lceil \mu_{k,j} \rfloor \mathbf{b}_j$. Because we changed the value of $\mathbf{b}_k$, all the values of $\mu_{k,i} (1 \leq i < j)$ need to change with it.

$$\mu_{k,i}^{'} = \frac{\langle \mathbf{b}_k - \lceil \mu_{k,j} \rfloor \mathbf{b}_j, \mathbf{b}_i^* \rangle}{\langle \mathbf{b}_i^*, \mathbf{b}_i^* \rangle} = \frac{\langle \mathbf{b}_k, \mathbf{b}_i^* \rangle}{\langle \mathbf{b}_i^*, \mathbf{b}_i^* \rangle} - \lceil \mu_{k,j} \rfloor \frac{\langle \mathbf{b}_j, \mathbf{b}_i^* \rangle}{\langle \mathbf{b}_i^*, \mathbf{b}_i^* \rangle} = \mu_{k,i} - \lceil \mu_{k,j} \rfloor \mu_{j,i}$$

For $1 \leq i < j < k$, we set $\mu_{k,i} \leftarrow \mu_{k,i} - \lceil \mu_{k,j} \rfloor \mu_{j,i}$. Obviously, when upon a size reduction of $\mathbf{b}_k$ with $\mathbf{b}_j$, we also change the value of $\mu_{k,i}$ $(1 \leq i < j)$. Therefore, the value of j must be traversed from $k-1$ to 1.

3.2 LLL Reduction

Definition 3. (δ **− LLL reduced basis**) Given $\frac{1}{4} < \delta \leq 1$, a basis $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\} \in \mathbb{R}^{m*n}$ is considered $\delta - LLL$ reduced if the following conditions are met.

- $\mathbf{B}$ is size reduced as defined in Definition 2.
- For all $2 \leq k \leq n$, $\delta \|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 \leq \|\pi_{k-1}(\mathbf{b}_k)\|^2$.

Algorithm 2: δ-LLL-reduction algorithm, described in [26]

Input: A basis $\mathbf{B}$ and $\frac{1}{4} < \delta \leq 1$.

Output: A basis $\mathbf{B}'$ which is δ-LLL reduced

Find the GSO basis $\mathbf{B}^*$ and GSO coefficients $\mu_{i,j}$
$k \leftarrow 2$
while $k \leq n$ **do**
    Size-reduce b_k ▷ *Use Algorithm 1*
    if $\|(\mathbf{b}_k^*)\|^2 < (\delta - \mu_{k,k-1}^2)\|(\mathbf{b}_{k-1})\|^2$ **then**
        $\mathbf{B} \leftarrow \sigma_{k-1,k}(\mathbf{B})$
        Update $\mathbf{b}_{k-1}^*, \mathbf{b}_k^*$ and $\mu_{i,j}$s$(i \geq k-1)$.
        $k \leftarrow \max(k-1, 2)$
    else
        $k \leftarrow k+1$
return A basis $\mathbf{B}'$ which is δ-LLL reduced.

As can be seen from the content of Sect. 2, $\|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 = \|(\mathbf{b}_{k-1}^*)\|^2$, and $\|\pi_{k-1}(\mathbf{b}_k)\|^2$ can be written as $\|\mathbf{b}_k^*\|^2 - \mu_{k,k-1}^2\|\mathbf{b}_{k-1}^*\|^2$. Therefore, the condition of δ-LLL reduction can be written equivalently as $(\delta - \mu_{k,k-1}^2)\|(\mathbf{b}_{k-1}) \leq \|(\mathbf{b}_k^*)\|^2$. Based on this condition, we get Algorithm 2 of LLL-reduced.

Algorithm 3 that describes BKZ consists of a LLL algorithm and a sub-algorithm that accurately solves the SVP algorithm. Its specific definitions and algorithms are as follows.

Definition 4. (**δ-Korkine Zolotarev reduced basis**) Given $\frac{1}{4} < \delta \leq 1$, a basis $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\} \in \mathbb{R}^{m*n}$ is said to be δ-Korkine Zolotarev reduced if the following conditions are satisfied.

- The basis $\mathbf{B}$ is size-reduced, as defined in Definition 1.
- For all $1 \leq i \leq n$, $\delta\|\mathbf{b}_i\| = \lambda_1(\pi_i(\mathcal{L}))$.

In Algorithm 3, we use *subalgorithm* to represent the exact solution algorithm for SVP.

Algorithm 3: δ-BKZ reduction algorithm

Input: A basis $\mathbf{B}$, An integer β and $\frac{1}{4} < \delta \leq 1$.

Output: A basis $\mathbf{B}'$ which is δ-BKZ reduced

Find the GSO basis $\mathbf{B}^*$ and GSO coefficients $\mu_{i,j}$
$i \leftarrow 1$
LLL reduce $\mathbf{B}$
while $i \leq n$ **do**
    $k \leftarrow \min(i + \beta - 1, n)$
    subalgorithm $(\{\mathbf{b}_i, \ldots, \mathbf{b}_k\})$
    $T \leftarrow \|\mathbf{b}_i\|$
    LLL reduce $\mathbf{b}_i, \ldots, \mathbf{b}_n$
    if $T = \|\mathbf{b}_i\|$ **then**
        $i \leftarrow i+1$
return A basis $\mathbf{B}'$ which is δ-BKZ reduced.

The sub-algorithms of the sixth line of the algorithm are mostly enumerations and sieving, and the time complexity of these algorithms is exponential. So we divide the entire lattice base into blocks of size b and run sub-algorithms on each block. In the seventh line, we record the length of the first vector in the block and LLL reduce the subsequent vectors. If the length of the vector does not change after LLL reduce, the index value is incremented by 1.

4 Enumeration

This section introduces various variants of enumeration algorithms along with their corresponding developments in parallel implementation.

4.1 The Basic Schnorr-Euchner Enumeration

While the earliest enumeration algorithm can be traced back to the 1980 s [16,21, 36], the first practical enumeration algorithm was introduced in 1994 by Schnorr and Euchner [37] as a subroutine of BKZ algorithm. This algorithm is referred to as the basic Schnorr-Euchner enumeration in [44], which treats the enumeration process as a traversal of the enumeration tree using depth-first search. During the enumeration process, all leaf nodes, representing lattice points, are traversed. Its practicality lies in its earliest introduction of pruning method. Pruning involves setting boundary functions to ensure that different layers of the enumeration tree have corresponding boundary values. If the target value generated by a node in a certain layer does not meet the required boundary value, the subtree corresponding to that node is discarded. The comprehensive description of this algorithm can be found in [37,44].

The enumeration algorithm in [37], as the first practical enumeration algorithm, serves as the foundation for subsequent algorithm improvements. Inspired by this enumeration algorithm, the following three notably significant parallel implementations were introduced in the subsequent years [12,15,19]:

HSB+10 [19]: The earliest application of GPU for accelerating the solution of the SVP using the enumeration algorithm was conducted by Hermans et al. in 2009. This parallel implementation based on [37]. The main idea was to split the enumeration tree at a higher level and then pass the split subtrees to the GPU for enumeration. This splitting approach eliminated communication between subtrees. However, due to the uncertain initial subtree sizes, termination times among threads varied. To tackle this, the authors introduced the Early Termination for improvement. The authors used the NVIDIA GTX 280 GPU, which provided nearly a 5-fold speedup compared to the serial enumeration algorithm in the fplll. However, the article only utilized a single CPU core and did not fully exploit the computational power of the CPU during the subtree enumeration phase. Additionally, more advanced pruning methods were not used.

DS10 [12]: In 2010, Dagdelen et al. introduced a parallelized version of the basic Schnorr-Euchner enumeration [37] based on multi-core CPUs. The main

idea was to divide the entire enumeration tree into different subtrees, which were placed in a shared subtree queue. Whenever a thread became idle, it could retrieve an unenumerated subtree from the subtree queue for enumeration. Each thread that obtained a new subtree was only allowed to partition it once. However, this approach introduced the problem of imbalanced subtree sizes because the subdivision of subtrees was not based on the size of the tree. The authors addressed this issue by considering the depth of the nodes. The authors conducted comparative experiments using different numbers of CPU cores. This parallel implementation achieved superlinear speedup, which was attributed to the shared minimum value A among different threads. This introduced some communication overhead but also resulted in better pruning effects. Although the single-threaded program in the article was slower than the enumeration algorithm provided by fplll, significant acceleration was observed when utilizing multiple cores in parallel. One limitation of this study was that only a single CPU was used, and future research should investigate how to leverage multi-core and multi-node CPUs to accelerate enumeration algorithms.

EBD+21 [15]: In 2021, Esseissah et al. made improvements to [19] using three strategies. For the preprocessing part, they employed a combination of randomization and Gaussian heuristics to obtain better bases. Concerning communication, [19] required CPU-to-GPU data transfer for each enumeration, leading to significant overhead. The authors proposed generating a portion of lattice points on the GPU to reduce communication load and enhance efficiency. The experiments were conducted using NVIDIA GeForce GTX 1060 GPU. Compared to [19], the proposed method achieved over 2.5 times speedup in a 110-dimensional lattice. Using two GPUs resulted in nearly linear or even superlinear speedup compared to using a single CPU. This indicates the scalability of the algorithm. The performance using two GPUs is superior to Correia's implementation [11] using a 16-core CPU in a 60-dimensional lattice. However, the article also has some limitations, such as not utilizing the latest GPU architecture and only conducting experiments with two GPUs. In the future, further research could explore large-scale experiments with multiple GPUs.

4.2 Extreme Pruning

Since 1994, when pruning methods were introduced into enumeration by Schnorr and Euchner [37], pruning has undergone several developments, solidifying its key role in enumeration. In 1995, Schnorr and Hörner [38] introduced an alternative pruning method based on the Gaussian volume heuristic. Herman et al. [19] also mentioned their intention to integrate this pruning approach into future implementations. However, both of these pruning methods lacked detailed analysis. It wasn't until 2010 that Gama et al. [17] presented a thorough theoretical analysis of diverse pruning approaches and introduced the concept of extreme pruning. Extreme pruning outperformed previous pruning-based enumeration algorithms in practice. In 2018, Aono et al. [5] demonstrated the lower bound of the cost of extreme pruning technology, further solidifying its effectiveness.

Starting from 2010, the parallel implementation of extreme pruning method has also garnered widespread attention from researchers [9,10,22]:

KSD+11 [22]: In 2011, Kuo et al. proposed a parallel enumeration algorithm using extreme pruning. Due to the introduction of the failure rate in the extreme pruning, in this approach, it was necessary to enumerate different enumeration trees corresponding to different bases of the same lattice. The SVP solver combines MapReduce technology. The GPU and MapReduce implementations of this algorithm were primarily based on [19], which employed a polynomial pruning function. In the experiment, the authors resolved the 114-dimensional SVP Challenge in approximately 40 h on a single workstation with eight NVIDIA GeForce GTX 480 GPUs. Additionally, through integrating recent techniques and leasing an Amazon server for $2,300, they successfully addressed the 120-dimensional SVP Challenge. However, the authors also acknowledged the difficulty in finding a good pruning function.

p3Enum [10]: In 2019, Burger et al. proposed p3Enum, an open-source framework tailored for parallelizing enumeration with extreme pruning, specifically devised for addressing the SVP. This parallel implementation was inspired by [6] and [22]. This paper introduced a new parameter, v, to the pruning function. This parameter increased the probability of successful enumeration and the workload, allowing for higher success rates when running for the same amount of time as the single-threaded program. The authors conducted experimental comparisons with well-known algorithms at the time and found that p3Enum was the fastest SVP solver in the 66–88 dimensional range. And p3Enum surpasses the GPU enumeration from [19] within dimensions 76–90. Moreover, on a 60-core system, they achieved a parallel efficiency surpassing 0.9. The authors intended to enhance the performance of p3Enum in their future work by incorporating MPI technology.

p3EnumOpt [9]: In 2019, Burger et al. made two improvements to the SVP solver p3Enum, called p3EnumOpt. First, they proposed a novel parallel implementation for extreme pruning. Second, they introduced the p3Enum extreme pruning function generator (p3Enum-epfg), which utilizes a parallelized simulated annealing approach to generate optimized extreme pruning functions for p3Enum's pruned enumeration. The authors conducted experiments to compare the efficiency of this new algorithm with p3Enum [10], fplll, and SubSieve. They found that the new algorithm had the fastest execution speed for solving SVP in the 66 to 92-dimensional range. However, the serial version of this algorithm was slower than fplll, and there was a lack of exploration for SVP in higher-dimensional lattices. This presents a potential direction for future research.

MAP-SVP [39]: In 2020, Tateiwa et al. introduced the world's inaugural distributed and asynchronous parallel SVP solver, referred to as the MAssively Parallel solver for SVP (MAP-SVP). The system is made up of a lot of Solvers and a management procedure named LoadCoordinator. In addition to running the DeepBKZ and ENUM algorithms, each Solver also exchanges short vectors through a vector pool that is under the control of a load coordinator. Extreme pruning procedures are used by every Solver. More aggressive pruning is used to

shorten the computation time for each Solver as the number of Solvers grows. Using 100,032 cores, the authors carried out the SVP research's largest-scale studies. In addressing instances of the SVP Challenge, they achieved new marks for dimensions 104, 111, 121, and 127 using this concurrent program. On the Fugaku supercomputer at RIKEN, the authors also want to carry out much larger-scale SVP-solving experiments involving millions of processes.

4.3 The Closest Point Search Algorithm: SE++

SE++ is a variant of an enumeration algorithm proposed by Ghasemmehdi and Agrell [18]. Experiments indicate that on a 60-dimensional lattice, it can reduce 75% of floating-point computations. The parallel implementation of this method was studied by Correia et al. [11]:

CMP16 [11]: In 2016, Correia et al. introduced a parallelized version of the practical enumeration algorithm SE++ [18] for solving the CVP, which can also be used to solve the SVP. This parallel implementation involved processing different branches of the tree in parallel and performing separate enumerations for each branch. Additionally, it avoided redundant calculations for symmetric branches in the tree, thus improving efficiency. The authors conducted experiments and comparisons, and found that SE++ outperformed the algorithm in [12] by 35% to 60% in terms of speed.

4.4 Comparison

In a broader perspective, the advancements in GPU and multi-core CPU technologies have propelled the evolution of parallel implementation within enumeration algorithms, significantly expediting the resolution speed of the Shortest Vector Problem (SVP). According to Table 2, the most efficient algorithm for lower dimensions (66–92) is currently p3EnumOpt [9]. Moreover, the MAP-SVP [39], utilizing extensive parallelism, has effectively addressed the 127-dimensional SVP Challenge. Both of these algorithms rely on extreme pruning, underscoring the pivotal role of this technique in parallel implementation. Nevertheless, up to the present point, the record holder for the SVP Challenge remains sieve, achieving a solution for the 186-dimensional SVP Challenge. This reality signifies the substantial scope for improvement within enumeration algorithms.

5 Sieving

The core concept of the sieving algorithm involves creating a list, denoted as L, comprising N lattice vectors. Subsequently, the algorithm engages in pairwise reduction, systematically replacing the longest vector within the list. The process concludes when the list saturates a ball with a radius R. In other words, the algorithm terminates when L encompasses a substantial portion of short vectors, each having a length no greater than R. Algorithm 4 provides a concise summary of this straightforward sieving algorithm.

Table 2. Performance comparison of parallel enumeration algorithms.

Year	Algorithm	Based	Device	Performance
2010	HSB+10 [19]	SE94 [37]	Tesla GPU	5x fplll-3.0.11 (dim=50–56, pre-reduction=LLL) (dim=58, pre-reduction=BKZ-20)
2010	DS10 [19]	SE94 [37]	16 Cores CPU	14x single-core (dim=52) 11–12x fplll-3.0.12 (dim=52)
2011	KSD+11 [22]	GNR10 [17]	Fermi GPU	New Records in 114, 116, 120
2016	CMP16 [11]	SE++ [18]	16 Cores CPU	1.35–1.6x DS10 [19] (dim=40, 50, 60)
2019	p3Enum [10]	GNR10 [17]	multi-core CPU	Best in 66–88
2019	p3EnumOpt [9]	GNR10 [17]	multi-core CPU	Best in 66–92
2020	MAP-SVP [39]	GNR10 [17]	100,032 cores	New Records in 104, 111, 121, 127
2021	EBD+21 [15]	SE94 [37]	Pascal GPU	1 GPU: 2.5x HSB+10 [19] 2 GPU: 5x HSB+10 [19] (dim=100, 110)

With the development of multi-core CPUs, GPUs, and FPGAs, many software developers and researchers are adopting parallel programming techniques to take advantage of these hardware resources. The sieving is no exception. In 2011, [33] proposed a parallel GaussSieve algorithm, but once the number of threads exceeds 10, the acceleration factor will not exceed about 5. So, [20] proposed a more practical parallel GaussSieve algorithm based on this algorithm, and [43] also used this framework to implement it on GPUs. Then, [30] proposed a parallel lock-free GaussSieve using a shared linked list. And then [28] also proposed a parallel lock-free HashSieve. With the proposal of LDSieve, [29] also used a similar lock-free mechanism to implement parallel LDSieve. Subsequently, [4] proposed a multiplatform parallel approach using FPGAs. Finally, [14] proposed two parallel bucketing methods, BGJ-like bucketing and BDGL-like bucketing, and a parallel reducing method using Tensor cores.

These algorithms can be roughly divided into three types of parallel optimization algorithms, parallel GaussSieve, parallel HashSieve and parallel LDSieve algorithms. The performance of these algorithms is contrasted in Table 3.

Algorithm 4: Sieving algorithm, described in [14]

Input: A basis $\mathbf{B} = (\mathbf{b}_1, ..., \mathbf{b}_n)$ of a lattice $\mathcal{L}$, a saturation radius R, and a list L
Output: A list L comprising short vectors that saturate the ball of radius R

1: Sample a lot of vectors from the lattice $\mathcal{L}$ in L.
2: **while** L does not cover the entire ball with a radius of R **do**
3: Find reduction and replace the longest vector in L.
return L

Table 3. The Performance, TC (Time Complexity) and SC (Space Complexity) comparison of each algorithm.

Year	Algorithm	Service	Performance	TC	SC
2011	MS11 [33]	CPU	thread$\leq$ 5, linear growth	-	-
2014	IKM+14 [20]	CPU	using 29,994 CPU hours in 128	$2^{0.52n}$	$<2^{0.2n}$
2014	MTB14 [30]	CPU	1.12x MS11 [33], 1.50x IKM+14 [20]	-	-
2020	AG20 [4]	FPGA	8x and only 6% CPU overhead	-	-
2015	MBL15 [28]	CPU	2.5x GuassSieve	$2^{(0.32n-15)}$ - $2^{(0.33n-16)}$	-
2017	MLB17 [29]	CPU	50x over the original LDSieve	-	-
2021	DSW+21 [14]	GPU	New Record in 180	$2^{0.367n}$ and $2^{0.338n}$	$2^{0.2075n}$

5.1 Parallel Gauss Sieve Algorithms

The main idea of the GaussSieve algorithm is to sample lots of vectors through a sampling algorithm. First, the algorithm uses the vectors in the list to reduce the sampling vectors, and then uses the reduced sampling vectors to reduce the vectors in the list. If reduction occurs, the sampling vectors will replace the corresponding vectors in the list.

The initial parallel implementation of the GaussSieve was introduced in 2011 by [33]. This algorithm aims to create a parallelized version version of the original GaussSieve algorithm. In this algorithm, let t represent the number of threads. Every thread possesses its unique instance and independently performs the sieving algorithm. These instances are ring-connected to one another. If the vector $\mathbf{v}$ is unchanged after reduction steps, it will be transmitted to the buffer Q_{i+1} in the subsequent instance. On the contrary, if $\mathbf{v}$ has any changes, it will be transferred to the distributed stack S_i within its respective instance. If a vector $\mathbf{v}$ successfully traverses all instances, it will be appended to the distributed list L_i. For a small number of up to 5 parallel threads, the scalability of the parallel version is nearly linear. Nevertheless, the algorithm fails to guarantee the sustained pairwise reduction of the entire list L. This is attributed to the fact that, upon the addition of a vector to the distributed list, another instance might introduce additional vectors. Consequently, the overall performance of this algorithm does not experience acceleration, particularly when dealing with a substantial number of threads.

In 2014, an alternative parallel version of GaussSieve was introduced by [20], designed for both shared and distributed memory systems, exhibiting enhanced scalability, especially up to 8 threads. The implementation capitalizes on the property of pairwise-reduced sets that have been unified, which is as follows: If two sets, A and B, have undergone pairwise reduction, and every vector pair ($\mathbf{a}$, $\mathbf{b}$) is Gauss-reduced for $\forall \mathbf{a} \in A$, $\mathbf{b} \in B$, then the union $A \cup B$ remains pairwise-

reduced. The algorithm takes a list V comprising r samples as input and follows a three-step reduction process.

During the initial phase, the sample vectors in V undergo simultaneous reduction with respect to the vectors in L, following the approach of the initial GaussSieve algorithm. Throughout this process, each modified vector is incorporated into the stack S, while unaltered vectors are transferred to the list V'. In the subsequent step, the initial vectors in V' undergo parallel reduction with each other. Modified vectors are relocated to the stack S, whereas unaltered vectors are shifted to the list V''. In the last step, each thread executes reduction operations on a portion of L against the vectors in V''. Similarly, if any vector changes, it is appended to the stack S; Otherwise, it is included in L'. Much like V'', L' also maintains pairwise reduction.

Upon completion of each iteration, the lists L' and V'' are combined to form the new list L, and the vectors in S populate the list V. This entire process continues until the count of collisions, denoted as K, reaches a predetermined threshold value, denoted as c. So this algorithm can solve the problem that there are non-Gauss-reduced pairs in the reduced list. However, this algorithm has two disadvantages: utilizing r samples elevates the computational load, and determining the optimal value for the parameter r in advance is not feasible. Additionally, insertions in S occur sequentially, posing limitations on scalability.

Therefore, [30] proposed a GuassSieve shared memory parallel algorithm based on a lock-free linked list. They have better scalability than [20] and better performance than the results reported in [33]. [30] used the Harris linked list to store vectors according to the length of the inner product. To avoid two threads modifying the same vector at the same time, a thread intending to modify the vector should first retrieve it from the list, and subsequently, insert the modified version. To achieve this, they made slight adjustments to the *Reduce* function in the gsieve library. The *Reduce* function was divided into two separate functions, namely $TestReduce$ and $eReduce$. $TestReduce$ evaluates whether $\mathbf{v}$ should undergo reduction against $\mathbf{w}$. If the outcome is true, the vector $\mathbf{w}$ is taken out from L and then duplicated into another variable. The copy of $\mathbf{w}$ is subsequently modified using $eReduce$. If the outcome is false, the $eReduce$ operation is not invoked. Thus the algorithm ensures that the vector is never modified in L. The entire algorithm process is as follows : first, all threads perform the reduction of $\mathbf{v}$ by $\mathbf{w}$, and $\mathbf{w} \in L$. Subsequently, vector $\mathbf{v}$ is scheduled for insertion into set L in a manner that ensures L maintains its order.

This algorithm may have the same problem as [33], but they relaxed the condition. While a vector $\mathbf{v}$ may not be reduced relative to another vector $\mathbf{w}$, the vectors in their reduced forms are likely to undergo further reduction in relation to each other. In addition, two optimizations of the algorithm were also suggested by [30]: the samples utilized in the sieving process are of shorter length, and the reduction process predominantly targets the shorter vectors themselves. The final achieved performance and scalability are much better than [33], and outperforms [20], with almost 1.12x and 1.50x performance improvement.

Finally, in 2020, [4] proposed an accelerator to accelerate the GuassSieve algorithm, and gave a multiplatform parallel implementation approach. The accelerator mainly accelerates two parts of the Reduce algorithm: dot product calculation and the modification of the vector's value. In the computation of the dot product, the initial step entails multiplication with the respective coefficients. Subsequently, the result is directed to an additive tree composed of $\lceil log_2(n)/\beta \rceil$ additive layers. Here, β represents the number of additions executed in a single clock cycle, with shorter delay paths compared to multiplications. Then, the clock delay for the dot product calculation is $1+\lceil log_2(n)/\beta \rceil$. After the dot product calculation, the algorithm need compute $q = \lceil dot/||\mathbf{u}||^2 \rfloor$. This operation does not perform division, but rather checks conditions. This operation requires only one clock cycle. For the update of the vector's value $||\mathbf{v}||^2 += q^2 \cdot ||\mathbf{u}||^2 - 2 \cdot q \cdot dot$, the hardware performs the norm update function in three sequential steps, with each step consuming one clock cycle. In the first step, the hardware calculates q is multiplied with the two interim results. Subsequently, the subtraction and addition operations are conducted. Finally, the delay of reducing a pair of vector depends solely on the dimensions of the lattice. In the case of an n-dimensional lattice, the delay $f_{cl}(n)$ is exactly equal to $f_{cl}(n) = \lceil log_2(n)/\beta \rceil + 5$.

A multi-platform approach to caching can eliminate data transfer bottlenecks. This algorithm is divided into three steps. The initial step involves reducing the set S by utilizing the reduced vectors from the list L. Let the size of S be $k = f_{tl}(n, w)$. The data transfer delay for a vector, denoted as $k = f_{tl}(n, w) = \lceil (n + 2) \cdot 16/\omega \rceil$. First, this approach transfers the k vectors of the set S from the CPU to the FPGA, and a vector of the list L, and then begins to reduce. The next vector of the list L is transmitted while reducing. The delay is $f_{el}^{p1}(n, w) = k^2 + k \cdot |L| + f_{cl}(n)$. During the second phase, the vectors in the set S are self-reduced, and as S is already resident in the FPGA, communication overhead is effectively eliminated. Hence, the delay is $f_{el}^{p2}(n, w) = k^2/2 \cdot f_{cl}(n) + k^2/2 + f_{cl}(n)$. In the final phase, the vectors in list L undergo reduction by the vectors in set S, akin to the first step. So the delay is $f_{el}^{p3}(n, w) = k \cdot |L| + f_{cl}(n)$. For the specified 160-dimensional lattice, the suggested resolution is anticipated to yield a performance improvement of 8.32 times compared to CPU. And the proposed architecture requires only 6% of the CPU-based cost.

5.2 Parallel HashSieve Algorithms

The HashSieve algorithm iterates through the following steps: (1) It randomly samples a lattice vector $\mathbf{v}$ (or fetches one from the stack S). (2) The algorithm identifies a neighboring candidate vector $\mathbf{w}$ from the hash table to perform reduction on $\mathbf{v}$. (3) Subsequently, the algorithm employs the vector $\mathbf{v}$ after reduction to perform reduction on another vector $\mathbf{w}$ from the hash table (if $\mathbf{w}$ undergoes reduction, it is moved onto the stack); (4) Ultimately, the algorithm includes $\mathbf{v}$ in either the stack or the hash table. The HashSieve uses LSH to construct T buckets, i.e., $H_1, ..., H_T$ tables. Each table only stores pointers corresponding

vectors and stores them in arrays. At the same time, only one thread is modifying a bucket.

Inspired by [30], and HashSieve is faster than GuassSieve in low dimensions. [28] proposed a scalable parallel implementation and handles concurrency based on a possible lock-free system. An important point is to combine (2), (3) into one step. The algorithm can directly use $\mathbf{v}$ to reduce $\mathbf{w}$ without waiting for $\mathbf{v}$ to be reduced. The latch mechanism proposed in [28] is to set a value for each vector, "0" means that no thread is using this vector, and "1" means that a thread is reading or writing to the vector. If contention occurs, subsequent threads will ignore this vector. This can lead to ignored reductions, but this problem is not serious. Because the probability of contention is low, and these ignored reductions may not be suitable for reduction. In addition, due to the increase in dimensions, the number of buckets will increase exponentially, which may cause only the first iteration to be executed without thread blocking. Therefore, at this time, multiple threads need to be added to operate on the same bucket. When the lock-free mechanism is enabled, the performance of this algorithm is approximately 2.2 times higher compared to GuassSieve, and when the lock-free mechanism is disabled, the performance is approximately 2.5 times higher. The main disadvantage of this algorithm is the large amount of memory used.

5.3 Parallel LDSieve Algorithms

The design of the buckets in LDSieve is straightforward: it randomly selects a direction in space and includes a vector in the bucket if its normalized inner product with vector $\mathbf{c}$ exceeds a certain constant α, expressed as α: $\frac{\langle \mathbf{v},\mathbf{c} \rangle}{||\mathbf{v}|| \cdot ||\mathbf{c}||} > \alpha$. Unlike employing T randomly chosen vectors $\mathbf{c_1}, \ldots, \mathbf{c_T}$, [8] introduces a different approach by selecting a random subcode $S \subset R^{n/m}$. The code words $C = \{\mathbf{c_1}, ..., \mathbf{c_T}\}$ are defined as the product code $C = S \times S \times ... \times S = S^m$, where a code word $\mathbf{c_i}$ is constructed by concatenating m code words $\mathbf{c_{i_1}}, ..., \mathbf{c_{i_m}}$ from S.

LDSieve first randomly generates a subcode S of dimension n/m and size $t^{1/m}$, defining a concatenated product code $C = S^m$. When processing the vector $\mathbf{v}$, LDSieve calculates the partial dot products among its m sub-vectors with the complete code C, storing the results in lists $L_1, ..., L_m$. The decoding algorithm efficiently sorts each list based on the inner product size. Subsequently, within a tree-like structure resembling a depth-first search, it identifies all buckets associated with a vector $\mathbf{c}$ in C that satisfy the condition of the normalized inner product between $\mathbf{c}$ and $\mathbf{v}$ being greater than α.

Also, inspired by [28–30] proposed a scalable parallel iteration of LDSieve featuring a probabilistic lock-free mechanism. The proposed variant relaxed certain properties of the algorithm to accommodate parallelism, demonstrating that LDSieve scales effectively on shared memory systems and utilizes significantly less memory than HashSieve, all while achieving comparable or even reduced execution time on random lattices. And LDSieve ($2^{0.09n}$) stores fewer pointers than GuassSieve ($2^{0.129n}$). Although contention increases as the dimension

grows, the number of buckets in LDSieve is sufficiently large that the actual lock is rarely used. Subsequently, [29] made some optimizations to the computation and storage of vectors. They also employed software-based prefetching and manual insertion of prefetch instructions to hide memory request latency. In the end, this algorithm was tested to be 50 times faster than the original implementation of LDSieve.

With the development of GPUs, Tensor cores make their appearance. [14] proposed two parallel bucketing methods, BGJ-like bucketing and BDGL-like bucketing, and a parallel reducing method using Tensor cores.

For BGJ-like bucketing(triple_gpu), the approach initially selects m bucket centers, preferably uniformly distributed on the sphere within the database. Subsequently, each vector $\mathbf{v}$ belonging to the set L in the database is linked with its respective bucket B_{k_v}, where $k_v = \underset{1 \le k' \le m}{argmax} |\langle \mathbf{b}'_{\mathbf{k}} / \|\mathbf{b}'_{\mathbf{k}}\|, \mathbf{v} \rangle|$. [14] slightly relaxed this condition, allowing vectors to be associated with up to M buckets. In every iteration, fresh bucket centers are selected, normalized, and stored on each individual GPU. Subsequently, the method streams through the GPU for processing the database $\mathbf{v_1}, ..., \mathbf{v_N}$. The distribution of bucket centers is allocated among 16 threads, and each thread is assigned to store only the optimal bucket encountered for each vector. Finally, the best M buckets are returned.

The bucketing approach for BDGL, resembling the asymptotically optimal method outlined in [8], shares similarities with bgj1 and relies on spherical caps. Similar to BGJ-like bucketing, it consistently returns the M best bucket centers for each vector. In the GPU implementation, each warp concurrently handles multiple vectors, with $\mathbf{v_i}$ stored in thread i $(mod 32) \in \{1, .., 32\}$. Depending on the dimensionality, the approach employs the Hadamard transform on the initial 32 or 64 coefficients, extracting 32 or 64 inner products simultaneously. The method can randomly permute the coordinates of the vector before the Hadamard transform to acquire additional bucket centers. To minimize branches and reduce overhead, each thread stores only the best-encountered bucket for every vector it handles. Consequently, for each vector, thread i, where i ranges from 1 to 32, stores the optimal buckets among i, $i + 32$, $i + 64$, and so forth. Out of these 32 results, the approach chooses the best M buckets.

In the reducing phase, calculating the inner product between all pairs is essentially equivalent to calculating one-half of the matrix product $Y_t Y$. Thus, utilizing Tensor cores significantly accelerates matrix computation, enhancing reduction speed. The obtained results facilitate filtering close pairs using two inequalities from [14]. Reported in [14], new records were achieved in the SVP Darmstadt Challenge, reaching dimension 180, surpassing the previous record of 155.

6 SVP Challenge

The SVP Challenge is a competition aimed at addressing the Shortest Vector Problem (SVP), as the hardness of SVP is closely related to the security of

lattice-based cryptography. As a result, it has captured the attention of numerous researchers.

Based on dimensions and random seeds set by participants, the SVP Challenge generates lattice basis matrices of Goldstein-Mayer type. Participants are tasked with finding an approximate exact solution to the shortest vector problem for the provided lattice basis. To enter the SVP Challenge Hall of Fame, participants need to provide an approximate exact solution shorter than the current record vector in the same dimension or a solution in a higher-dimensional lattice.

The current top 10 participants in the SVP Challenge Hall of Fame are presented in Table 4. The highest SVP Challenge record dimension is 186, accomplished on Jul. 25, 2023. This record was achieved using a sieving algorithm with GPU acceleration, taking 50.3 d. Analyzing the top 10 records shows a consistent trend: all algorithms utilized sieving. Furthermore, nearly all implementations use GPUs for acceleration.

The top 10 records have primarily been broken by several distinct teams. The first three records were achieved by L. Wang and B. Wang, who employed the pnj-BKZ during the preprocessing. Meanwhile, Sun and Chang, along with Ducas et al., used the algorithm introduced by Ducas et al. [14] in their work. The ninth-ranked record was established by L. Wang and Y. Wang [40].

In summary, sieving, G6K, pnj-BKZ, and GPU-based parallel acceleration are the key methods for solving high-dimensional SVP in the current landscape.

Table 4. Top 10 of the SVP Challenge Hall of Fame, data from [32].

Pos	Dim	Norm	Contestant	Alg	Date	GPU	Time
1	186	3484	L. Wang, B. Wang	Sieving	20230725	✓	50.3d
2	184	3494	L. Wang, B. Wang	Sieving	20230703	✓	43.2d
3	182	3444	L. Wang, B. Wang	Sieving	20230528	✓	35.6d
4	181	3544	Y. Sun, S. Chang	Sieving	20230717	✓	31d
5	180	3499	L. Wang, B. Wang	Sieving	20230526	✓	28.1d
6	180	3509	L. Ducas, M. Stevens, W. van Woerden	Sieving	20210208	✓	51.6d
7	178	3447	L. Ducas, M. Stevens, W. van Woerden	Sieving	20210208	✓	22.8d
8	176	3487	L. Ducas, M. Stevens, W. van Woerden	Sieving	20201013	✓	12.5d
9	170	3378	L. Wang, Y. Wang	Sieving	20221003	-	-
10	170	3438	L. Ducas, M. Stevens, W. van Woerden	Sieving	20200512	✓	8d

Acknowlegments. This work is funded by Major Project of Scientific and Technological R&D of Hubei Agricultural Scientific and Technological Innovation Center [grant number 2020–620–000–002–03], Natural Science Foundation of Hubei Province [grant number 2023AFB394], Knowledge Innovation Program of Wuhan-Shuguang Project [grant number 2022010801020283].

References

1. Ajtai, M.: Generating hard instances of lattice problems. In: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, pp. 99–108 (1996)
2. Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: Proceedings of the thirty-third annual ACM symposium on Theory of computing, pp. 601–610 (2001)
3. Albrecht, M.R., Ducas, L., Herold, G., Kirshanova, E., Postlethwaite, E.W., Stevens, M.: The General Sieve Kernel and New Records in Lattice Reduction. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11477, pp. 717–746. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17656-3_25
4. Andrzejczak, M., Gaj, K.: A Multiplatform Parallel Approach for Lattice Sieving Algorithms. In: Qiu, M. (ed.) ICA3PP 2020. LNCS, vol. 12452, pp. 661–680. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60245-1_45
5. Aono, Y., Nguyen, P.Q., Seito, T., Shikata, J.: Lower Bounds on Lattice Enumeration with Extreme Pruning. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10992, pp. 608–637. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96881-0_21
6. Aono, Y., Wang, Y., Hayashi, T., Takagi, T.: Improved Progressive BKZ Algorithms and Their Precise Cost Estimation by Sharp Simulator. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9665, pp. 789–819. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49890-3_30
7. Bai, S., Laarhoven, T., Stehlé, D.: Tuple lattice sieving. LMS J. Comput. Math. **19**(A), 146–162 (2016)
8. Becker, A., Ducas, L., Gama, N., Laarhoven, T.: New directions in nearest neighbor searching with applications to lattice sieving. In: Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms,pp. 10–24. SIAM (2016)
9. Burger, M., Bischof, C., Krämer, J.: A new parallelization for p3enum and parallelized generation of optimized pruning functions. In: 2019 International Conference on High Performance Computing and Simulation (HPCS), pp. 931–939. IEEE (2019)
10. Burger, M., Bischof, C., Krämer, J.: p3Enum: A new parameterizable and shared-memory parallelized shortest vector problem solver. In: Rodrigues, J.M.F., Cardoso, P.J.S., Monteiro, J., Lam, R., Krzhizhanovskaya, V.V., Lees, M.H., Dongarra, J.J., Sloot, P.M.A. (eds.) ICCS 2019. LNCS, vol. 11540, pp. 535–542. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22750-0_48
11. Correia, F., Mariano, A., Proenca, A., Bischof, C., Agrell, E.: Parallel improved schnorr-euchner enumeration SE++ for the CVP and SVP. In: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 596–603. IEEE (2016)
12. 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). https://doi.org/10.1007/978-3-642-15291-7_21
13. Ducas, L.: Shortest vector from lattice sieving: a few dimensions for free. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10820, pp. 125–145. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78381-9_5
14. Ducas, L., Stevens, M., van Woerden, W.: Advanced Lattice Sieving on GPUs, with Tensor Cores. In: Canteaut, A., Standaert, F.-X. (eds.) EUROCRYPT 2021. LNCS, vol. 12697, pp. 249–279. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77886-6_9

15. Esseissah, M.S., Bhery, A., Daoud, S.S., Bahig, H.M.: Three strategies for improving shortest vector enumeration using GPUS. Sci. Program. **2021**, 1–13 (2021)
16. Fincke, U., Pohst, M.: Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. Math. Comput. **44**(170), 463–471 (1985)
17. 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). https://doi.org/10.1007/978-3-642-13190-5_13
18. Ghasemmehdi, A., Agrell, E.: Faster recursions in sphere decoding. IEEE Trans. Inf. Theory **57**(6), 3530–3536 (2011)
19. 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). https://doi.org/10.1007/978-3-642-12678-9_4
20. 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). https://doi.org/10.1007/978-3-642-54631-0_24
21. Kannan, R.: Improved algorithms for integer programming and related lattice problems. In: Proceedings of the fifteenth annual ACM symposium on Theory of computing, pp. 193–206 (1983)
22. 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.) Extreme enumeration on GPU and in clouds. LNCS, vol. 6917, pp. 176–191. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23951-9_12
23. 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). https://doi.org/10.1007/978-3-662-47989-6_1
24. Laarhoven, T., Mariano, A.: Progressive lattice sieving. In: Lange, T., Steinwandt, R. (eds.) Progressive lattice sieving. LNCS, vol. 10786, pp. 292–311. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-79063-3_14
25. Laarhoven, T., Mosca, M., van de Pol, J.: Solving the shortest vector problem in lattices faster using quantum search. In: Gaborit, P. (ed.) PQCrypto 2013. LNCS, vol. 7932, pp. 83–101. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38616-9_6
26. Lenstra, A.K., Lenstra, H.W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische annalen **261**(ARTICLE), 515–534 (1982)
27. Loyer, J., Chailloux, A.: Classical and quantum 3 and 4-sieves to solve SVP with low memory. Cryptology ePrint Archive (2023)
28. Mariano, A., Bischof, C., Laarhoven, T.: Parallel (probable) lock-free hash sieve: a practical sieving algorithm for the SVP. In: 2015 44th International Conference on Parallel Processing, pp. 590–599. IEEE (2015)
29. Mariano, A., Laarhoven, T., Bischof, C.: A parallel variant of lDSieve for the SVP on lattices. In: 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pp. 23–30. IEEE (2017)
30. Mariano, A., Timnat, S., Bischof, C.: Lock-free gausssieve for linear speedups in parallel high performance SVP calculation. In: 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing, pp. 278–285. IEEE (2014)

31. Micciancio, D., Voulgaris, P.: Faster exponential time algorithms for the shortest vector problem. In: Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms, pp. 1468–1480. SIAM (2010)
32. Michael, S., Nicolas, G.: SVP Challenge (2010). https://www.latticechallenge.org/SVP-challenge/
33. 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). https://doi.org/10.1007/978-3-642-23178-0_40
34. Mukhopadhyay, P.: Faster provable sieving algorithms for the shortest vector problem and the closest vector problem on lattices in l_p norm. Algorithms **14**(12), 362 (2021)
35. Nguyen, P.Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. J. Math. Cryptology **2**(2), 181–207 (2008)
36. Pohst, M.: On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications. ACM Sigsam Bulletin **15**(1), 37–44 (1981)
37. Schnorr, C.P., Euchner, M.: Lattice basis reduction: improved practical algorithms and solving subset sum problems. Math. Program. **66**, 181–199 (1994)
38. Schnorr, C.P., Hörner, H.H.: Attacking the Chor-Rivest cryptosystem by improved lattice reduction. In: Guillou, L.C., Quisquater, J.-J. (eds.) EUROCRYPT 1995. LNCS, vol. 921, pp. 1–12. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-49264-X_1
39. Tateiwa, N., et al.: Massive parallelization for finding shortest lattice vectors based on ubiquity generator framework. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15. IEEE (2020)
40. Wang, L., Wang, Y., Wang, B.: A trade-off SVP-solving strategy based on a sharper PNJ-BKZ simulator. In: Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, pp. 664–677 (2023)
41. Wang, X., Liu, M., Tian, C., Bi, J.: Improved nguyen-vidick heuristic sieve algorithm for shortest vector problem. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pp. 1–9 (2011)
42. Xia, W., Wang, L., Gu, D., Wang, B., et al.: Improved progressive BKZ with lattice sieving. Cryptology ePrint Archive (2022)
43. Schnorr, C.P., Hörner, H.H.: Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction. In: Guillou, L.C., Quisquater, J.-J. (eds.) EUROCRYPT 1995. LNCS, vol. 921, pp. 1–12. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-49264-X_1
44. Yasuda, M.: A survey of solving SVP algorithms and recent strategies for solving the SVP challenge. In: Takagi, T., Wakayama, M., Tanaka, K., Kunihiro, N., Kimoto, K., Ikematsu, Y. (eds.) A survey of solving SVP algorithms and recent strategies for solving the svp challenge. MI, vol. 33, pp. 189–207. Springer, Singapore (2021). https://doi.org/10.1007/978-981-15-5191-8_15
45. Zhang, F., Pan, Y., Hu, G.: A Three-level sieve algorithm for the shortest vector problem. In: Lange, T., Lauter, K., Lisoněk, P. (eds.) A three-level sieve algorithm for the shortest vector problem. LNCS, vol. 8282, pp. 29–47. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43414-7_2