IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 2, pp. 362–384. DOI:10.46586/tches.v2025.i2.362-384

Sieving with Streaming Memory Access

Ziyu Zhao[1], Jintai Ding[2,3](✉) and Bo-Yin Yang[4]

[1] Tsinghua University, Beijing, China, ziyuzhao0008@outlook.com
[2] Xi'an Jiaotong-Liverpool University, Suzhou, China
[3] Basque Center For Applied Mathematics, Bilbao, Spain, jintai.ding@gmail.com
[4] Academia Sinica, Taipei, Taiwan, by@crypto.tw

Abstract.
We implement an optimized BGJ (Becker–Gama–Joux 2015) sieve and analyze its behavior in a study of RAM access overheads (and their minimization) in sieving algorithms for large lattice problems. Both experiment and theory points to BGJ's inherent structure being much more memory-efficient than the BDGL (Becker–Ducas–Gama–Laahoven 2016) sieve, which uses asymptotically the fewest logical operations. In particular, a dimension-n BGJ sieve uses only $2^{0.2075n+o(n)}$ ***streaming*** *(non-random)* main memory accesses. A key insight: Bucket sizes decrease by orders of magnitude after each BGJ filtering layer, so that sub-buckets fit into successively much smaller (hence faster) storage areas. Our refined BGJ is competitive at cryptographic sizes and should outperform BDGL for all practically achievable dimensions.
The above is corroborated by the results from our efficient CPU-based BGJ implementation in an optimized framework, which saves about 40% RAM footprint and is $\geq 2^{4.5}\times$ more efficient gate-count-wise compared to the Ducas–Stevens–van Woerden 2021 4-GPU implementation, which like most prior sieving-based SVP computations is a HK3 (Herold–Kirshanova 2017) sieve. Notably, we solved the 183-dimensional SVP Darmstadt Challenge in 30 days on a 112-core server and 0.87 TB of RAM; similarly we also found a short vector in the 796-dimensional Ideal-SVP Challenge. Our implementation may offer further insights into the behavior of asymptotically "fast" sieving algorithms when applied to large-scale problems. Moreover, our refined cost estimation of SVP based on this implementation suggests that some NIST PQC candidates (e.g. Falcon-512), are not sure to meet NIST's security requirements.

Keywords: Sieving · Lattice Cryptanalysis · SVP

1 Introduction

BKZ [Sch87, SE94] is currently one standard method for estimating the concrete hardness of lattice-based cryptosystems. BKZ uses a shortest vector problem (SVP) subroutine, whose concrete hardness estimation therefore plays a central role in the security analysis of lattice-based cryptosystems. Enumeration and sieving are the two main strategies for solving SVP. Enumeration [Poh81, Kan83] solve SVP using superexponential time and polynomial space (in particular the optimized algorithms [GNR10, CN11]). Sieving [AKS01], whose space use is exponential in the dimension of the lattice, is asymptotically faster in gate complexity[1]. But it was not until circa 2018 for sieving to catch up to enumeration, after many asymptotic improvements on sieving itself [NV08, MV10, BGJ13, Laa15, BDGL16], and its frameworks [Duc18a, Duc18b, ADH+19]. Today, a sieving-based SVP subroutine is a standard and required component in lattice-based cryptanalysis.

[1] We will use this term to denote an idealized concept of *time complexity* in which each logical operation costs the same regardless of how far the operands need to travel.

Licensed under Creative Commons License CC-BY 4.0.
Received: 2024-10-15 Accepted: 2024-12-15 Published: 2025-03-04

Sieving's exponential space requirement is a significant challenge when scaling to larger problems, because each random access to a massive 2-dimensional storage array containing N bits of data may be reasonably assumed to incur an $\mathcal{O}(N^{\frac{1}{2}})$ cost, both in terms of time and energy consumption[2], but how — indeed even ***whether*** — to assess the impact of this communication cost on the overall complexity is a subject of ongoing debate [Ber20, Ber23].

Indeed, many sieving algorithms, including the [BDGL16] sieve — usually regarded as the state of the art — use a randomized divide-and-conquer approach to accelerate the process of searching for reducing pairs. E.g., [BDGL16] itself divides the vectors in the main memory first into many small buckets (the bucket size is subexponential in the dimension of the lattice, but each vector is assigned to exponentially many buckets) then search for reducing pairs within each bucket[3]. This strategy is very efficient for ***gate*** complexity, but it requires a larger number of *random* memory accesses in the reduction step. This cost is already significant in real-world sieving implementations: E.g., [DSv21] shows the asymptotically "slower" but more memory-friendly HK3 sieve [HK17] to be equal or superior to the BDGL sieve in all achievable dimensions, due to the limited CPU-GPU bandwidth. Hence the majority of computations for the sieving-based SVP Darmstadt Challenges [SG10] are currently conducted using sieving algorithms that are far from being asymptotically optimal in gate-complexity, with the practical behavior of asymptotically faster sieving algorithms in large-scale problems remaining unclear.

Contributions.

This work presents a detailed implementation and analysis of an optimized version of the BGJ sieve [BGJ15], demonstrating its significant theoretical and practical interest. Intuitively, the BGJ sieve applies successive random filters to create a series of progressively smaller buckets from the main database and searches for reducing pairs only in the smallest buckets. We found that such a structure can be implemented in a highly memory-efficient manner for large-scale sieving attacks. The key idea is that the bucket size decreases by several orders of magnitude after each filtering, allowing sub-buckets to be stored in a much smaller and therefore faster storage device. No communication between sub-buckets is necessary. Under reasonable assumptions, we show the most costly filtering and reducing steps to take only $2^{0.2075n+o(n)}$ streaming main memory accesses, where n is the lattice's dimension. This is corroborated by our empirical results. We also discuss how to insert the shorter reduced vectors back into the main database with streaming memory accesses.

One should keep in mind that streaming memory accesses are significantly cheaper than random access in the real world. From a memory access point of view, this result is therefore much better than the BDGL sieve, which requires at least $2^{0.292n+o(n)}$ random memory accesses. And it makes the implementation of a large-scale BGJ sieve reasonable. However, to argue that the memory cost should not be an essential problem in the concrete security estimation of lattice-based cryptosystems, one also needs to show that the speed of the refined BGJ sieve is competitive when compared to the state-of-the-art.

Following the complexity analysis in [BGJ15] and the idea of the `bgj1` sieve in [ADH+19], our refined BGJ sieve replaces the original simhash-like filters with spherical cap-shaped filters. That is, a vector $\mathbf{v}$ can pass the filter $\mathcal{F}_{\mathbf{c},\alpha}$ if $|\langle \mathbf{v}, \mathbf{c}\rangle| \geqslant \alpha\|\mathbf{v}\|\|\mathbf{c}\|$, where $\mathcal{F}_{\mathbf{c},\alpha}$ is the filter with center $\mathbf{c}$ and radius α. It turns out that such a choice is highly efficient in practice. Our implementation actually shows that the BDGL sieve is still much slower than the refined BGJ sieve for SVP with dimensions around 200, and it seems unlikely that the BDGL sieve will outperform in those dimensions (around 400) of cryptographic interest. One may speculate, based on both theoretical and practical evidence, that this refined BGJ sieve has an asymptotic complexity something close to $2^{0.292n+o(n)}$. However, due to the complex geometric shapes of the filter regions (in our case, the intersection of several

[2] One can assume a probably less practical 3-dimension storage, see e.g., [Jaq24].

[3] We describe a more practical version and not the original [BDGL16] that requires much more memory.

spherical caps) we have not provided such a formula and leave it to future work. In fact, such a formula offers few insights for concrete security estimation because the $o(n)$ term typically plays a significant role for cryptographically relevant sieving dimensions. For example, as estimated in [Duc22], the overhead caused by non-uniformity is approximately 2^6 for sieving dimensions around 380.

Optimized Sieving Framework. Furthermore, we propose some improvements that can be applied to general sieving algorithms. First, our implementation shows that high-precision floating-point numbers are dispensable in the most costly steps of sieving. By appropriately scaling and rounding, the coordinates of the lattice vectors are stored as 8-bit signed integers. We use a "dual-LLL-reduced" basis to efficiently recover the coefficients of these rounded vectors when necessary. Using a lower precision representation can save not only time and memory but, more importantly, memory bandwidth.

We also provide a unified concept for sieving and enumeration. During the current sieving process on a local projected lattice, one will find numerous short vectors that are not short enough to be inserted back into the main database. However, if one finds such vectors during enumeration, they will definitely be lifted to see if they yield a shorter vector in the original lattice. So we choose to further push the idea of *on-the-fly-lifting* [ADH$^+$19] by directly inserting back into the main database the vectors that are still short after lifting. This strategy saves both time and memory by offering a much larger dimension for free.

Implementation and Performance. The implementation constituted the most labor-intensive aspect of our work. We have developed a low-level optimized, multi-threaded, and memory-efficient CPU implementation of the `bgj1`, `bgj2`, and `bgj3` sieves, which correspond to the BGJ sieve with 1, 2, and 3 levels of filtering, respectively. Our implementation also includes a dual hash [DSv21] optimized with a locality sensitive filter. All these algorithms can be invoked via a command-line interface, with parameters such as the number of threads, maximum sieving dimension, sieving context, and so forth, passed as arguments. We aim to provide a tool that is user-friendly and can offer the community deeper insights into these sieving algorithms. The code is available at `https://github.com/zhaoziyu0008/BGJ-Sieve-AMX`.

In terms of performance, the `bgj3` sieve solved a 169-dimensional SVP Darmstadt Challenge in 3.4 days using a 112-core server. This is already several orders of magnitude faster than the previous highest records based on CPU, which required 8 months with 224 cores to solve a 166-dimensional challenge. We further implemented a three-level BGJ sieve on the latest Intel architectures, which we refer to as `bgj3-amx`. The `bgj3-amx` is approximately 7 times faster than the `bgj3`, and it managed to solve the 179-dimensional SVP Darmstadt Challenge in just 11.2 days. This is about 4 times faster than the previous 4-GPU implementation [DSv21], and the RAM cost is also reduced by 40%. Considering the significantly higher computational power of the GPUs, we actually achieve an efficiency gain of about $2^{4.5}$, as shown in Table 5. We also applied our results, in conjunction with recent BKZ techniques [ZD23], to the Ideal Lattice Challenge[4] [PS13]. This resulted in an improvement of approximately 2^{10} in total over the previous highest record, see Table 6.

Refined Security Analysis. A refined concrete hardness estimation for SVP, based on our implementation, is given in Section 7. It shows that some of the NIST PQC candidates, such as Falcon-512, are not sure to meet NIST's security requirements. We suggest, for instance in the case of Falcon-512, to modify the parameters to balance the hardness of forgery and key recovery attacks if a security level of 143 bits is truly necessary.

[4] Viewed as an approximate-SVP instance.

Roadmap.

In Section 2, we introduce necessary notations and basic definitions. Subsequently, the refined BGJ sieve is presented in Section 3, followed by an analysis of its performance in Section 4. Section 5 is dedicated to discussing optimizations for the general sieving framework. The specifics of our implementation, along with its performance, are detailed in Section 6. In Section 7, we provide a refined security analysis of SVP based on our implementation. Finally, we conclude our work and discuss future directions in Section 8.

2 Preliminaries

2.1 Lattices and the Shortest Vector Problem

We start counting at zero. All vectors are denoted by bold lowercase letters and are to be read as column vectors. Matrices are denoted by bold capital letters. $\mathbb{S}^{n-1}$ is the unit sphere in $\mathbb{R}^n$. For a full rank matrix $\mathbf{B} = (\mathbf{b}_0, \mathbf{b}_1, \cdots, \mathbf{b}_{n-1})$, we denote the lattice generated by the basis $\mathbf{B}$ as $\mathcal{L}(\mathbf{B}) = \{\mathbf{Bx} | \mathbf{x} \in \mathbb{Z}^n\}$. The dual lattice of $\mathcal{L}(\mathbf{B})$ is defined to be $\mathcal{L}(\mathbf{B}^\vee)$ where $\mathbf{B}^\vee = (\mathbf{b}_0^\vee, \mathbf{b}_1^\vee, \cdots, \mathbf{b}_{n-1}^\vee)$ such that the dot product

$$\langle \mathbf{b}_i^\vee, \mathbf{b}_j \rangle = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise} \end{cases}$$

and $\text{span}\langle \mathbf{b}_0^\vee, \mathbf{b}_1^\vee, \cdots, \mathbf{b}_{n-1}^\vee \rangle = \text{span}\langle \mathbf{b}_0, \mathbf{b}_1, \cdots, \mathbf{b}_{n-1} \rangle$.

The Euclidean norm of a vector $\mathbf{v}$ is denoted by $\|\mathbf{v}\|$, and the volume of a lattice $\mathcal{L}(\mathbf{B})$ is $\text{Vol}(\mathcal{L}(\mathbf{B})) = \sqrt{\det(\mathbf{B}^T\mathbf{B})}$. For a lattice $\mathcal{L}$, $\lambda_1(\mathcal{L})$ denotes the length of the shortest nonzero vector in $\mathcal{L}$.

Definition 1 (Shortest Vector Problem (SVP))**.** Given a lattice basis $\mathbf{B}$, the shortest problem asks to find a nonzero vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ such that $\|\mathbf{v}\| = \lambda_1(\mathcal{L}(\mathbf{B}))$.

The hardness of the shortest vector problem is the cornerstone of the security of lattice-based cryptosystems. No efficient (quantum) algorithm is known for solving SVP. However, the length of the shortest vector in *random* lattices can be efficiently estimated as follows

Theorem 1 (Gaussian Heuristic)**.** *Suppose K is a measurable body in $\mathbb{R}^n$, for "random" full-rank lattice $\mathcal{L} \subset \mathbb{R}^n$, the number of lattice points in K is approximately $Vol(K)/Vol(\mathcal{L})$. In particular, $\lambda_1(\mathcal{L}) \approx \sqrt{n/(2\pi e)}\, Vol(\mathcal{L})^{\frac{1}{n}} =: \mathsf{gh}(\mathcal{L})$.*

2.2 Local Projected Lattices, Sieving and Dimension For Free

The Gram-Schmidt orthogonalization of a lattice basis $\mathbf{B}$ is denoted by $\mathbf{B}^* = (\mathbf{b}_0^*, \cdots, \mathbf{b}_{n-1}^*)$, which satisfies

$$\mu_{ij} = \frac{\langle \mathbf{b}_j^*, \mathbf{b}_i \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle} \qquad \text{and} \qquad \mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=0}^{i-1} \mu_{ij} \mathbf{b}_j^*.$$

We denote the projection orthogonally to $\text{span}\langle \mathbf{b}_0^*, \mathbf{b}_1^*, \cdots, \mathbf{b}_{i-1}^* \rangle$ by π_i, for $i = 0, 1, \cdots, n$. For $0 \leqslant l \leqslant r \leqslant n$, if $\mathcal{L}$ is the lattice generated by $\mathbf{B}$, the local projected lattice $\mathcal{L}_{[l,r]}$ with respect to $\mathbf{B}$ (only one fixed basis will be used in the paper, thus we ignore $\mathbf{B}$ in the notation) is defined as the lattice generated by $\mathbf{B}_{[l,r]} = (\pi_l(\mathbf{b}_l), \pi_l(\mathbf{b}_{l+1}), \cdots, \pi_l(\mathbf{b}_{r-1}))$. Also, for $\mathbf{v} \in \mathcal{L}_{[l',r]}$ where $l' \leqslant l$, we denote $\pi_l(\mathbf{v})$ by $\mathbf{v}_{[l,r]}$.

Given a vector $\mathbf{v} = \sum_{i=l}^{r-1} \lambda_i \pi_l(\mathbf{b}_i) \in \mathcal{L}_{[l,r]}$, $l' \leqslant l$, one can efficiently obtain a *lifted* vector which we denote by $\mathsf{Lift}_{l'}(\mathbf{v})$ through a "size-reduction". That is, first compute

$\tilde{\mathbf{v}} = \sum_{i=l}^{r-1} \lambda_i \pi_{l'}(\mathbf{b}_i)$ in $\mathcal{L}_{[l',r]}$, then repeat the process $\tilde{\mathbf{v}} = \tilde{\mathbf{v}} - \lceil \frac{\langle \tilde{\mathbf{v}}, \mathbf{b}_j^* \rangle}{\|\mathbf{b}_j^*\|^2} \rfloor \pi_{l'}(\mathbf{b}_j)$ for $j = l-1, \cdots, l'$.

Now we briefly recall the concept of sieving and dimension for free. Sieving algorithms, first proposed by Ajtai et al. [AKS01], are the asymptotically best known algorithms to solving SVP in terms of their gate complexity, which (as well as their space complexity) are both exponentially large in the dimension of the lattice. A generic form of sieving is summarized in Algorithm 1. By "saturate the ball of radius R", we mean that a constant ratio of the lattice points within the ball are found[5]. The saturation radius is typically chosen to be $\sqrt{4/3} \cdot \mathsf{gh}(\mathcal{L})$.

Algorithm 1 Sieving Algorithm [NV08, ADH+19]

```
Require: The basis $\mathbf{B}$ of an $n$-dimensional lattice $\mathcal{L}$, a saturation radius $R$.
Ensure: A list $L$ of lattice vectors.
1: $L \leftarrow$ a set of $2^{0.2075n+o(n)}$ random vectors in $\mathcal{L}$.
2: while $L$ does not saturate the ball of radius $R$ do
3:     for each pair $\mathbf{u}, \mathbf{v} \in L$ do
4:         if $\|\mathbf{u} - \mathbf{v}\| < \max_{\mathbf{w} \in L} \|\mathbf{w}\|$ then
5:             replace the longest vector in $L$ with $\mathbf{u} - \mathbf{v}$.
6: return $L$
```

The sieving algorithm starts with an exponentially large database of lattice vectors. Then it tries to find *reducing pairs*, i.e., pairs of vectors whose difference is short, and replaces those longer vectors in the list with the difference, until the list saturates the ball of radius R[6]. After the sieving procedure, for example, 50% of the lattice points in the ball of radius R will be found, which contains the shortest vector with high probability.

To reduce time and memory costs, it is suggested in [Duc18a] to first sieve on a locally projected lattice $\mathcal{L}_{[l,n]}$, and then lift all the vectors in the list to $\mathcal{L}_{[0,n]}$. Note that if $\mathbf{v}$ is a short vector in $\mathcal{L}_{[0,n]}$, then $\mathbf{v}_{[l,n]}$ is likely also short, and thus contained in the list. Therefore, we may successfully find the shortest vector in $\mathcal{L}_{[0,n]}$ by sieving on $\mathcal{L}_{[l,n]}$, thereby gaining l dimensions "for free". This is the so-called "dimension for free" technique. According to heuristic analysis, the free dimension l is asymptotically $\frac{n \ln(4/3)}{\ln(n/2\pi e)}$. In practice, as demonstrated in [Duc18a, DSv21], it can reach up to $n/\ln(n)$.

3 The Sieving Algorithms

3.1 Sieving with locality sensitive filters

The most time-consuming step in Algorithm 1 is the search for reducing pairs. A naive approach that checks all pairs would lead to a gate complexity quadratic in the size of the list. Since practical sieving algorithms like [MV10, NV08] were proposed, a long series of work [BGJ13, BGJ15, Laa15, BDGL16] has been dedicated to accelerating the search for reducing pairs. The key idea is to use a randomized divide-and-conquer approach called locality sensitive filters.

Generally speaking, after the initial database of lattice vectors is generated, the locality sensitive filter based sieving algorithms repeat three steps, *filtering*, *reducing*, and *inserting*, until the list saturates the ball of radius R. In the filtering step, the vectors in the list are filtered into many small buckets, such that reducing pairs are more likely to enter the

[5] The exact number of vectors in the ball is typically hard to know. It is commonly estimated using the Gaussian heuristic.

[6] Of course, one should also stop the algorithm when no pair of list vectors generates shorter vectors. This is unlikely to happen with sane choices of R.

same bucket. For example, the state-of-the-art BDGL sieve uses filters that correspond to spherical cap-shaped *filter regions* in the unit ball. That is, a vector $\mathbf{v}$ can pass the filter $\mathcal{F}_{\mathbf{c},\alpha}$ with center $\mathbf{c}$ and radius α if and only if $|\langle \mathbf{v}, \mathbf{c}\rangle| \geqslant \alpha\|\mathbf{v}\|\|\mathbf{c}\|$. Then, in the reducing step, the vectors in the same buckets are pairwise checked for reducing pairs. Finally, during the inserting step, the longest vectors in the list are replaced with the shorter vectors found in the reducing step.

3.2 The Refined BGJ Sieve

The original BGJ sieve, introduced by Becker, Gama, and Joux in 2015 [BGJ15], has a gate complexity of $2^{0.311n+o(n)}$. This algorithm efficiently generates buckets by applying a series of random filters to the main database, creating progressively smaller buckets. The search for reducing pairs is confined to the smallest buckets.

In the following sections, we will use "AllPairSearch" to denote the procedure that identifies a significant portion of reducing pairs, such as 50% or 99% of all possible pairs in the vector list, with high probability. The primary distinction among most sieving algorithms lies in how they implement AllPairSearch. Therefore, our focus will be on this procedure.

Algorithm 2 AllPairSearch - BGJ15

```
Require: A list $L$ of $N$ lattice vectors, a minimum number $N_{min}$, a number of repetitions
    $B$, a goal norm $\ell$, and a set of filters $\mathcal{F}$.
Ensure: A list of reducing pairs in $L$ with a sum/difference shorter than $\ell$.
if $N \leqslant N_{min}$ then
    return $(\mathbf{v}, \mathbf{u}) \in L^2$ s.t. $\|\mathbf{v} \pm \mathbf{u}\| < \ell$.
$\mathcal{N} \leftarrow \emptyset$.
for $i = 0, 1, \cdots, B-1$ do
    Pick a random filter $f$ from $\mathcal{F}$
    $L'$ is defined as the set of vectors $\mathbf{v}$ in $L$ that can pass the filter $f$.
    $\mathcal{N} \leftarrow \mathcal{N} \cup$ AllPairSearch$(L', N_{min}, B, \ell, \mathcal{F})$.
return $\mathcal{N}$.
```

Algorithm 2 shows the AllPairSearch used in BGJ15, without specifying the details of the filters. We replace the original filters with spherical cap-shaped filters in our refined BGJ sieve, as these filters have been shown to be optimal in terms of gate complexity in [KL21], and the bgj1 sieve in [ADH+19] has proven efficient in practice.

We will refer to the refined BGJ sieve with 1, 2, and 3 levels of filtering as bgj1, bgj2, and bgj3, respectively. A general version with k levels of filtering will be denoted by bgjk. Algorithm 3 illustrates the AllPairSearch used in bgj3. From Algorithm 3, it should be clear what bgjk with a number of repetitions $(B_0, \cdots, B_{k-1})$ and filter radius $(\alpha_0, \cdots, \alpha_{k-1})$ looks like.

Here a vector $\mathbf{v}$ can pass the filter $\mathcal{F}_{\mathbf{c},\alpha}$ if and only if $|\langle \mathbf{v}, \mathbf{c}\rangle| \geqslant \alpha\|\mathbf{v}\|\|\mathbf{c}\|$, as mentioned in Section 3.1. Now, we discuss how we choose the parameters α_i's and B_i's in our implementation. In a real implementation, we do not need to find almost all reducing pairs at once, so the choice of B_i's is quite flexible. Usually, we do insertions and resort the database according to length after $0.025N_0$ reducing pairs are found. The key points are that the B_i's should be large enough to ensure that sufficient computations occur each time we read the vectors from the database, thereby minimizing the memory access overhead. They should also be small enough to keep the RAM usage by these temporary buckets acceptable. Typical values of B_i/B_{i-1} range from 64 to 512, and this largely depends on the architecture.

The choice of the α_i's is more delicate. The goal is to balance the cost of the filtering and the quality of the buckets. As shown for the case of bgj1 in [ADH+19], the asymptotically

Algorithm 3 AllPairSearch - bgj3

```
Require: A list $L$ of $N_0$ ($n$-dimensional) lattice vectors, number of repetitions $(B_0, B_1, B_2)$, filter radius $(\alpha_0, \alpha_1, \alpha_2)$ and a goal norm $\ell$.
Ensure: A list of reducing pairs in $L$ with a sum/difference shorter than $\ell$.
$\mathcal{N} \leftarrow \emptyset$.
for $i = 0, 1, \cdots, B_0 - 1$ do
  Pick a random filter center $\mathbf{c}_0$ from $\mathbb{S}^{n-1}$.
  Compute $L_i := \{\mathbf{v} \in L | \mathbf{v}$ can pass $\mathcal{F}_{\mathbf{c}_0, \alpha_0}\}$
  for $j = 0, 1, \cdots, B_1/B_0 - 1$ do
    Pick a random filter center $\mathbf{c}_1$ from $\mathbb{S}^{n-1}$.
    Compute $L_{ij} := \{\mathbf{v} \in L_i | \mathbf{v}$ can pass $\mathcal{F}_{\mathbf{c}_1, \alpha_1}\}$
    for $k = 0, 1, \cdots, B_2/B_1 - 1$ do
      Pick a random filter center $\mathbf{c}_2$ from $\mathbb{S}^{n-1}$.
      Compute $L_{ijk} := \{\mathbf{v} \in L_{ij} | \mathbf{v}$ can pass $\mathcal{F}_{\mathbf{c}_2, \alpha_2}\}$
      $\mathcal{N} \leftarrow \mathcal{N} \cup \{(\mathbf{u}, \mathbf{v}) \in L_{ijk}^2 | \|\mathbf{u} \pm \mathbf{v}\| < \ell\}$.
return $\mathcal{N}$.
```

optimal choice ($\alpha_0 = 0.366$) can be far from the practical optimum ($\alpha_0 = 0.315 \sim 0.325$). We directly provide the optimal values we selected for our bgj1, bgj2, bgj3, and bgj3-amx in Table 1. These values were obtained through a brute force search, meaning we ran the codes with all reasonable choices of α_i's and chose the fastest one[7]. It's worth noting that even a small change of approximately 0.01 in α_i's can result in a noticeable slowdown.

Table 1: Chosen Filter Radius in bgj1, bgj3, bgj3, and bgj3-amx

Algorithm	α_0	α_1	α_2
bgj1	0.325	-	-
bgj2	0.257	0.280	-
bgj3	0.200	0.210	0.280
bgj3-amx	0.210	0.215	0.285

4 Performance Analysis

4.1 Gate Complexity

The gate complexity of the algorithm in Algorithm 2 is intrinsically tied to the class of filters, $\mathcal{F}$. Following [BGJ15], we model the lattice points in the sieving database as random points in the high-dimensional sphere. We use P_f to represent the probability that a vector will pass a random filter from $\mathcal{F}$, and P_p is used to denote the probability that a pair of vectors, which form an angle of $\pi/3$, are both accepted by the same random filter. The effectiveness of the filters is typically assessed by the exponent ρ such that $P_f^\rho = P_p$, to which the time complexity is tightly related.

Theorem 2 (Complexity of AllPairSearch-BGJ15, Theorem 1 in [BGJ15])**.** *Suppose L is a list of N uniformly random vectors in the sphere of dimension n, ρ is the exponent such that $P_f^\rho = P_p$, then the gate complexity of Algorithm 2 is $\tilde{\mathcal{O}}(N^\rho)$.*

[7] We first did a (coarse) grid search within the boundaries [0.15, 0.35], then performed stochastic gradient descent around those promising points several times. The global minimum point appears to be unique

In our case, $\mathcal{F}$ is a set of spherical cap-shaped filters $\mathcal{F}_{\mathbf{c},\alpha}$ for random centers $\mathbf{c}$ and a certain radius α. To compute P_f and P_p in this case, we need to know the volume of spherical caps $\mathcal{C}_{\mathbf{c},\alpha} = \{\mathbf{x} \in \mathbb{R}^n | \|\mathbf{x}\|^2 = 1, \langle \mathbf{x}, \mathbf{c} \rangle \geqslant \alpha \|\mathbf{c}\|\}$ and wedges (i.e. intersections of spherical caps) $\mathcal{W}_{\mathbf{c}_1,\alpha_1,\mathbf{c}_2,\alpha_2} = \mathcal{C}_{\mathbf{c}_1,\alpha_1} \cap \mathcal{C}_{\mathbf{c}_2,\alpha_2}$.

Lemma 1 (Volume of spherical caps and wedges, Lemma 2.1, 2.2 in [BDGL16]). *Let μ be the canonical Lebesgue measure over $\mathbb{R}^n$, $\mathcal{S}^{n-1}$ be the unit sphere in $\mathbb{R}^n$, then for any $\alpha \in (0, 1)$ we have*

$$\frac{\mu(\mathcal{C}_{\mathbf{c},\alpha})}{\mu(\mathcal{S}^{n-1})} = \mathsf{poly}(n) \cdot \left(\sqrt{1-\alpha^2}\right)^n .$$

Furthermore, if the angle between $\mathbf{c}_1$ and $\mathbf{c}_2$ is θ, then

$$\frac{\mu(\mathcal{W}_{\mathbf{c}_1,\alpha,\mathbf{c}_2,\alpha})}{\mu(\mathcal{S}^{n-1})} = \mathsf{poly}(n) \cdot \left(\sqrt{1 - \frac{2\alpha^2}{1+\cos\theta}}\right)^n .$$

According to Lemma 1, we have $P_f = \mathsf{poly}(n)\cdot(1-\alpha^2)^{n/2}$ and $P_p = \mathsf{poly}(n)\cdot(1-\frac{4}{3}\alpha^2)^{n/2}$. This implies that asymptotically

$$\rho \approx \ln(1 - \frac{4}{3}\alpha^2)/\ln(1-\alpha^2)$$

This equation suggests that by choosing a very small α and using multiple levels of filters, our AllPairSearch can achieve the asymptotically optimal [KL21] time complexity of $\tilde{\mathcal{O}}(N^{4/3})$ in the *sparse regime*[8] ($N = 2^{o(n)}$). This result is better than the original BGJ sieve where $\rho = 1.5$. Thus, we did the implementation with the anticipation that it would be highly efficient in practice. It turns out that the refined BGJ sieve is not only memory-friendly but also competitive in terms of gate complexity compared to the state-of-the-art.

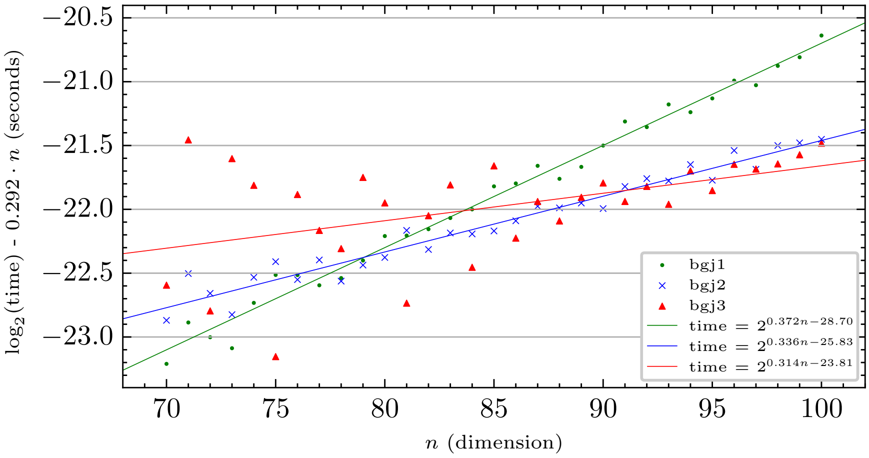

Figure 1: Comparison of bgj1, bgj2 and bgj3

We ran a left progressive sieve [ADH+19] on a 100-dimensional lattice from Darmstadt SVP Challenge[SG10] using the bgj1, bgj2, and bgj3 sieves. The tests were conducted on a

[8] It is not guaranteed that filters optimal in the sparse regime will remain optimal in the *dense regime* ($N = 2^{\mathcal{O}(n)}$), i.e. in the case of lattice sieving. For instance, cross-polytope hashing is known to be optimal in the sparse regime [TT07, AIL+15], but it leads to a suboptimal gate complexity of $2^{0.2972n+o(n)}$ [LdW15] when applied to lattice sieving. Our main interest lies in the practical performance of sieving for dimensions related to cryptography (approximately 380), so we leave the asymptotic complexity analysis to future work.

machine equipped with an Intel Xeon Gold 6338 CPU, running at around 2.8GHz, using a single thread. The results are shown in Figure 1. The timings represent the amount of time spent in each sieving dimension before reaching a saturation of 37.5% with a database size of $3.2 \cdot 2^{0.2075n}$. We can see that the crossover point between bgj1 and bgj2 is only around 77, and the crossover point between bgj2 and bgj3 is around 92. That is, the refined BGJ sieve quickly benefits from the improved bucket quality provided by the second and third levels of filtering.

For sieving dimension 140, which is close to the largest practical sieving dimension, we compare our bgj3-amx with 2-bdgl_gpu from [DSv21]. To run a left progressive sieve up to a sieving dimension of 140 with a saturation ratio of 37.5%, our bgj3-amx takes approximately $2^{15.2}$ seconds using 112 threads on a dual Intel Xeon Platinum 8479 CPU server. In contrast, 2-bdgl_gpu requires about twice the wall time (estimated from Fig.7 of [DSv21]) and around eight times more floating-point operations. Even with a sieving dimension 240 larger, the BDGL sieve would only gain at most $2^{240\cdot(0.3112-0.2925)} = 2^{4.5}$ times more speed up than the original BGJ sieve from the leading term in the exponent. Therefore, it's unlikely that the BDGL sieve can beat the refined BGJ sieve in a sieving dimension of 380, unless more improvements are made in the $o(n)$ term of $2^{0.292n+o(n)}$.

4.2 Solving the Memory Access Issue

In this section, we provide a detailed analysis of the memory access overhead in bgjk sieves. It turns out that $2^{0.2075n+o(n)}$ streaming main memory accesses are sufficient for the entire sieving process, and the memory access overhead can be negligible.

Firstly, we note that streaming memory access is inexpensive, and unlike random access, its speed should not decrease by a factor square root in the size of the storage device. If an attacker can afford the GPUs for conducting the computations in sieving for dimension 380 within a reasonable time, for example within 10 years, then such an attacker should certainly be able to afford the disks to store the database of the lattice vectors. The data in different disks can be streamed out in parallel. Therefore, it is reasonable to assume that the streaming memory access only slows down by a constant factor, regardless of the size of the sieving database. Here a subtle point is the necessity of processing the read data locally. Transferring the data elsewhere adds a cost linear to the distance, which is $\mathcal{O}(N^{3/2})$ (with a very small constant term for streaming transfer) if we want to move the data out of a 2D storage device containing N bytes of data. This movement cost is negligible for cryptographic size sieving[9], so we ignore it in the following analysis.

Before proceeding through each steps, we illustrate the key idea by giving a comparison with the BDGL sieve. If n is the sieving dimension, BDGL uses a single filter layer to generate $2^{\mathcal{O}(n)}$ small buckets. Putting the vectors in the main database into some of these buckets requires *randomly accessing* the exponentially large space for these buckets. However, if BGJ uses $\mathcal{O}(\log(n))$ successive filter layers to generate progressively smaller buckets, the number of subbuckets for each bucket can be $2^{\mathcal{O}(n/\log(n))}$, which is subexponential (in practice in $[64; 512]$, cf. page367). Thus, BGJ only needs to randomly access a subexponential space for the subbuckets, which is clearly more efficient.

Now we address the most costly part of sieving, the filtering and reducing steps. To compute all the subbuckets from a large database (a larger bucket or the main database), it is sufficient to stream the vectors in the larger database *once* to a machine that contains all the centers of the filters. This machine checks whether the streamed vectors can pass those filters and *moves* them into corresponding buckets. The pairwise dot product in the last-level buckets in the reducing step is just a matrix multiplication, which can be

[9] For a sieving dimension of 380, if we use 16TB disks to store the vectors, each occupying 0.01m^2, the sieving database will cover an area of 300km × 300km. Transporting 2^{50} vectors (by car) over 300km may cost approximately \$100, but the corresponding computation will be at least $(2^{50})^{0.292/0.208} \times 380 \approx 2^{79}$ FMA operations, which would cost more than \$100,000.

implemented trivially with streaming memory accesses. Therefore, all memory access in these steps can be streamed, and only $2^{0.2075n+o(n)}$ such accesses to the main database are necessary. To make the memory access overhead negligible, we need to further ensure that the memory access cost of generating each subbucket is less than the cost of further computations within this subbucket. Note that the computations required are superlinear (with an exponent range from $0.292/0.2075 \approx 1.4$ to 2) in the size of the subbucket. Thus, as long as the subbucket size is larger than some constant, the memory access overhead caused by the streaming memory access, which is only linear in the bucket size, is negligible. A sketch of why we can do everything in $2^{0.2075n+o(n)}$ (streaming) accesses is presented in Appendix A.

For the last several levels of very small buckets, the memory access may slow down the computations if the hardware architectures are not well designed. A similar phenomenon can already be observed in our bgj3-amx implementation. Intel's AMX instructions were initially designed for AI applications, and thus are not entirely suitable for sieving, where we need to compute many dot products of vectors with no more than 160 entries. It only provides 8 tmm registers, so each register can only be used in $2 \sim 3$ tdpbssd instructions after loading the data from the cache. As a result, our bgj3-amx suffers significantly from the latency of tileloadd and tilestored instructions. Nevertheless, this slowdown can only be at most a constant factor, and such a problem, which is only related to the speed and size of small caches, is completely different from the issue of accessing random vectors from an exponentially large database. We also note that this problem is minor for most of the current CPU (with AVX2 or even AVX512) and GPU architectures.

The insertion step, although it only requires $2^{0.2075n+o(n)}$ computations, is somewhat tricky to stream in practice. Current sieving implementations usually maintain a hash table to check whether a vector is already in the list. Such a check, although it has a constant gate complexity, requires random memory access. If we do not check for duplicates before a vector is inserted into the list, the list will soon be "polluted" by duplicated vectors. Our solution is to first mergesort the list vectors and the newly found short vectors together, and then remove the duplicates and the longest ones. This procedure can be performed using a streaming memory accesses at the cost of slightly increasing the time complexity of the insertion step to $\mathcal{O}(2^{0.2075n+o(n)}\log(2^{0.2075n+o(n)})) = 2^{0.2075n+o(n)}$, which should be acceptable.

The concept in this subsection is also demonstrated in our bgj3-amx implementation. Actually, the authors' idea of how to address the memory access issue in sieving was initially inspired by the implementation. The detailed computation speed, time, and bucket size for different steps are listed in Table 2. Data were collected while sieving in dimension 140 with 112 threads on the dual Intel Xeon Platinum 8479 CPU server. From this, we can see that although the speed of the first two filters suffers heavily from the poor RAM bandwidth, this cost only slightly affects the overall performance, because most of the computations happen when the buckets get small enough to fit into faster caches, where the memory bandwidth is no longer a bottleneck.

Table 2: Profiling Data of bgj3-amx

Step	Filter-0	Filter-1	Filter-2	Reducing
Speed (TOPS)	11.81	11.10	39.19	116.4
Bucket size	278.8GB	3.386GB	80.75MB	556.7KB
Data in	RAM	RAM	L3-Cache	L2-Cache
Total Time	544.7s	451.4s	762.4s	3397s

5 Optimizations

5.1 Sieving with Low Precision

In all of the sieving algorithms we implemented (bgj1, bgj2, bgj3, bgj3-amx), the entries of the lattice vectors in the main sieving database are stored as 8-bit signed integers. Most of the computationally intensive parts of the sieving, including the filtering and reducing steps, are also performed with 8-bit precision. Only during the insertion step do we recover the newly found vectors with 32-bit precision. These vectors are then properly scaled, rounded, and carefully checked before being inserted into the main database. Vectors that do not pass the check are discarded (e.g., because the norm is too large or not all the entries lie in the range $[-127, 127]$). It turns out that if one carefully maintains the main database, the relative error of the dot product results is typically less than 1% and the outliers are rejected during the check before being inserted into the main database. Thus the precision loss does not significantly affect the sieving procedure.

We provide details on the most intricate parts of the 8-bit implementation. For the choice of the scaling factor, if $\mathbf{B}$ is the basis of the lattice, using the notation in Section 2, the scaling factor in our implementation is chosen to be $254.0 \cdot (\max_{0 \leqslant i \leqslant n-1} \|\mathbf{b}_i^*\|)^{-1}$. This ensures that no vector entry exceeds 127 in absolute value after size reduction. To recover the accurate vectors from the 8-bit representations, we first use short dual vectors of the basis to compute the integer coefficients with respect to the basis $\mathbf{B}$. We then recover the vector using these coefficients. Such short dual vectors can be obtained by first computing the dual basis of $\mathbf{B}$, and running the LLL algorithm [LLL82] on the dual basis. The LLL algorithm here is necessary.

Compared to previous sieving implementations based on 32-bit floating-point numbers, the use of 8-bit precision leads to a $2 \sim 4$ times improvement, both in terms of speed and memory usage. These improvements contribute significantly to the refined security estimation in Section 7.

5.2 Seeking More Dimension for Free

From Dual Hash to LSF-based Dual Hash. In practice, to reduce both time and memory costs, a common strategy is to aim for more *dimension for free*, i.e., to find short enough vectors with a smaller sieving dimension. One way to achieve this is by using the *dual hash* technique proposed in [DSv21][10]. Once sieving on the local projected lattice $\mathcal{L}_{[l,n]}$ is complete, i.e., exponentially many short vectors in $\mathcal{L}_{[l,n]}$ have been found, the dual hash technique suggests lifting all pairwise sums/differences of these vectors to $\mathcal{L}_{[0,l]}$. The lifting is done cleverly: a hash value is computed for each vector, and only the "lift-worthy" vector pairs with hash values close to each other are lifted. Checking for hash value pairs is much faster than lifting, thus the dual hash technique can significantly reduce the lifting cost and works well in the GPU implementation in [DSv21].

However, the cost of the dual hash technique is nevertheless quadratic in the size of the list. In our preliminary CPU implementation, we tried the dual hash technique and found it to be unacceptably slow in sieving dimensions only around 120. A natural idea to improve this is to use locality sensitive filters. We choose to use the filters $\mathcal{G}_{\mathbf{c},\ell}$ such that $\mathbf{v} \in \mathcal{L}_{[l,n]}$ can pass $\mathcal{G}_{\mathbf{c},\ell}$ iff $\|\mathsf{Lift}_0(\mathbf{v} \pm \mathbf{c})\|^2 - \|\mathbf{v} \pm \mathbf{c}\|^2 < \ell^2$. Pairwise checks for the dual hash values are only done for vectors that can pass the same filter. This leads to a $2 \sim 10$ times improvement in efficiency for sieving dimensions ranging from 100 to 140. However, even the LSF-based dual hash still takes extra time at least comparable to the time for sieving itself, thus not "free", which is still unsatisfactory.

Observations. Our current solution is based on two simple observations. Firstly, we propose

[10] We refer to Section 4.5 of van Woerden's thesis [vW23] for a more detailed and expository discussion.

a unified concept for the left progressive sieve and enumeration. During the sieving process on a local projected lattice, numerous found vectors are not short enough to be inserted back into the main database. However, many of these vectors are only slightly longer (for example, 5%∼10% longer) than the threshold. If such short vectors in the local projected lattice $\mathcal{L}_{[l,n]}$ are found during enumeration, they will certainly be lifted to see if they yield a short vector in $\mathcal{L}_{[l',n]}$ for some $l' \leqslant l$. In our preliminary implementation, these vectors are simply discarded, wasting potential short vectors that could be helpful in enumeration. This suggests that our implementation may not be optimal and could benefit from these short vectors.

The second observation is that if a vector $\mathbf{v} \in \mathcal{L}_{[l,n]}$ is short after lifting to $\mathcal{L}_{[0,n]}$, then it cannot be too long. According to Gaussian heuristics, if we are sieving on $\mathcal{L}_{[l,n]}$ and lifting all sums/differences of the vectors in the list to $\mathcal{L}_{[0,n]}$, we suggest estimating the minimal length of those lifted vectors as follows:

$$\min_{0\leqslant\alpha\leqslant 0.5}\left(\left(N^2\cdot(1-\alpha^2)^{n/2}\right)^{-2/l}\cdot \mathsf{gh}(\mathcal{L}_{[0,l]})^2+(2-2\alpha)\cdot\left(1.18\cdot\mathsf{gh}(\mathcal{L}_{[l,n]})\right)^2\right)^{1/2}$$

where N is the size of the list, and the number 1.18 is chosen because after the sieving is done, the median length of the vectors in the list is around $1.18 \cdot \mathsf{gh}(\mathcal{L}_{[l,n]})$. The second term represents the norm in $[l, n]$ of the difference between two vectors with angle $\arccos \alpha$, and the first term is the expected minimal norm in $[0, l]$ among these differences. This estimation is slightly pessimistic, as many vectors in the list are much shorter than the median length. In practice, this estimated length can be achieved on average after exhausting 15%∼25% of the search space. If we are targeting a dimension for free of around 30, it turns out that the α to minimize the estimation formula is typically larger than 0.4, even larger than 0.45 when the sieving dimension is large (e.g., 140). This suggests that even if a pair of vectors pass the dual hash test, they are still unlikely to be "lift-worthy" because the cosine of the angle between them is usually much less than 0.4, i.e., the sum/difference of the two vectors is already too long without lifting. This observation suggests that it may be possible to further reduce the number of lifts while maintaining comparable lifting quality.

Our Solution. In our final implementation, we assign a score to each vector in the list, rather than naively using the length to assess the quality of the vectors. The score of a vector $\mathbf{v} \in \mathcal{L}_{[l,n]}$ is computed as

$$\mathsf{score}(\mathbf{v}) = \min_{0\leqslant l'\leqslant l.}\frac{\|\mathsf{Lift}_{l'}(\mathbf{v})\|}{\mathsf{gh}(\mathcal{L}_{[l',n]})}$$

Vectors with smaller scores are preferred, and we use newly found vectors with small scores to replace the vectors in the list with larger scores during the insertion step. To find vectors with small scores, in the filtering and reducing steps, if s is the score of a 77% quantile vector in the list, we aim to find vectors in $\mathcal{L}_{[l,n]}$ with a length less than $1.07 \cdot s \cdot \mathsf{gh}(\mathcal{L}_{[l,n]})$. During insertion, scores for these vectors are computed and only 0.5% ∼ 2% of these collected vectors turn out to have a small enough score to be inserted into the list. The score computing speed in our implementation is about $2^{21} \sim 2^{22}$ vectors/(second·core), and the overall cost of computing the scores is typically less than 10% of the sieving time, which is negligible.

In summary, our final implementation further pushes the idea of *on-the-fly-lifting* [ADH+19] by directly inserting vectors that are short after lifting into the main database. It turns out that the left progressive sieve procedure even becomes slightly faster after enabling this new technique. Intuitively, this is because some work in the next few sieving dimensions is done by reusing the vectors discarded in the original implementation. Both the overall time and space cost for solving SVP are significantly reduced by the larger dimension for free. For a comparison with the state of the art, one can compare the dimension for free of our Darmstadt SVP Challenge results (those challenges with dimension $\leqslant 162$ were solved with the preliminary code) in Section 6.4 with Table 1 in [DSv21].

6 Implementation Details

6.1 General Design Principles

Our implementation, crafted in C++, extensively utilizes intrinsic functions for low-level optimizations. We compile the code using the clang-17.0.6 compiler, with the -O3 -march=native optimization flag enabled to support the latest Intel CPU architectures. For multi-threading, we have opted for the OpenMP library.

Most of our implemented algorithms, including bgj1, bgj2, bgj3, bgj3-amx, and the locality sensitive filter-based dual hash, are accessible directly from a command-line interface. Parameters such as the number of threads, maximum sieving dimension, sieving context, among others, can be passed as arguments. Notably, all our SVP challenges were solved using this command-line tool, eliminating the need for direct interaction with the C++ interfaces. We anticipate that this tool will be user-friendly and provide the community with deeper insights into these sieving algorithms.

6.2 Vector Representation and Data Structures

In our implementation of the sieving algorithms, we manage the following data: the lattice basis, the main database of lattice vectors, a unique identifier (uid) hash table, and a list of "compressed vectors" for sorting.

Each coordinate of the lattice basis is stored as the quad_float type in NTL [Sho], offering 106-bit precision. We perform the computation of Gram-Schmidt orthogonalization, the local projected lattice, and the LLL reductions using the quad_float type to ensure the original basis remains unaffected by numerical errors. Notably, we have developed inline assembly code based on AVX512 instructions for the basic quad_float vector operations, which makes these basis-related computations extremely fast.

For each vector in the main database, we record its coordinates (8-bit signed integers, aligned to 32 bytes), the square of the norm (32-bit signed integers), the sum of all coordinates (32-bit signed integers), and a 64-bit uid. Before inserting a vector into the primary database, we first recover it to 32-bit precision and then round it, preventing the accumulation of numerical errors during the sieving procedure. Consequently, this most space-consuming part of the data only requires 176 bytes per vector. The rationale for maintaining the sum of the vector coordinates will be explained in Section 6.3.

The uid hash table serves to check whether a vector already exists in the main database. We initially tried the std::unordered_set in the standard C++ library, and found it extremely inefficient in terms of RAM usage. It consumed $40 \sim 50$ bytes per uid, which is more than 25% of the space cost of the main database. Therefore, we strongly recommend replacing the std::unordered_set with a better unordered set implementation, for instance, Sparsepp [Pop] to reduce both time and RAM costs. Also, one should keep in mind that for sieving dimensions $\geqslant 140$, the uid of different vectors may collide with a high probability.

To efficiently sort the vectors in the list, we follow the approach in [ADH$^+$19] to maintain a list of "compressed vectors". Our compressed vectors contain a 16-bit norm/score of the vector and a 32-bit integer to record the address of the corresponding vector in the main database. Sorting is only performed with these "compressed vectors" to minimize the cost of data movement.

6.3 Low-level Optimizations

In the ensuing subsection, we give details of those computationally intensive parts of the sieving implementation. Predominantly, the computations involved in the filtering and reduction phases consist of dot products followed by comparisons to check whether the result exceeds a predetermined threshold.

In our "AVX2" implementations, namely bgj1, bgj2, and bgj3, the dot products of int8_t vectors are first computed by vpdpbusd on ymm registers. We then use the vphaddd instruction to horizontally add the 32-bit results in the ymm registers, simultaneously for 8 dot products. The final comparison is conducted using vpcmpgtd. As a result, the theoretical throughput for a dot product computation is less than 4 clock cycles. Consequently, we have opted not to use the simhash trick [Cha02, FBB$^+$15, Duc18a], which gives no improvement even in our preliminary bgj1 implementation based on 32-bit floating-point numbers. Furthermore, in our bgj2, bgj3, and particularly in the bgj3-amx implementation, we have chosen to discard the 3-reductions [HK17] due to the excessive cost of additional comparisons and data movements.

The vpdpbusd instruction, originally designed for AI applications, can only compute the dot product of a uint8_t vector and a int8_t vector. Therefore, when computing the dot product, we first need to add 0x80 to each entry of one of the vectors to convert it into a uint8_t vector. Subsequently, we subtract 128 times the sum of the entries of the other vector, after the dot product is done.

In our bgj3-amx implementation, the dot products are computed using the tdpbssd instruction in Intel's Advanced Matrix Extensions (AMX). The AMX extension equips each core with eight 1KB tile registers (tmm0 to tmm7) and a set of instructions designed to efficiently handle matrix operations. It supports both int8(tdpbssd) and fp16(tdpbf16ps) data types, achieving an 8x throughput improvement over the previous AVX-512 VNNI extension for int8 FMA. AMX is primarily designed for efficiently computing large matrix multiplications, thus not very suitable for sieving. In fact, while only 3 tdpbssd instructions are sufficient to compute 256 dot products, a significant amount of time is spent on loading the data and storing the results with tileloadd and tilestored. Additionally, we need to transpose one of the 16 by 64 int8_t matrices before it is loaded into the tmm registers for computing dot products. Our current implementation uses vpunpckldq, vpunpckhdq, and vshufi64x2 instructions to accomplish this, which is relatively slow. Furthermore, the comparisons performed after the dot products are also costly. As a result, our bgj3-amx implementation only achieves a speedup of 6 to 7 times compared to bgj3, which is not satisfactory. We plan to further optimize the code to improve performance.

6.4 Performance and SVP Challenge results

We now proceed to showcase the results of the Darmstadt SVP challenge[SG10], as a means to justify our work and compare it with the current state-of-the-art. The specifics of the machines used for our SVP challenges are detailed in Table 3. Most of the large-scale challenges were solved with a combination of CPU times on X_1 and X_2, differing only in the amount of RAM, hence we do not distinguish between them and simply refer to them as X in Table 4. Also, in our low-level optimized implementations, hyperthreading does not offer any benefits.

The performance details of our implementation for solving Darmstadt SVP Challenges are provided in Table 4. Here, "D4F", "MSD", and "dh" denote "dimension for free", "maximum sieving dimension" and "locality-sensitive filter-based dual hash" (see Section 5.2 for more details), respectively. The RAM usage for most of the smaller challenges was not meticulously recorded, hence it is not displayed in the table. We just report that our implementation only requires 0.87TB of RAM for a sieving dimension of 146, with approximately $3.2 \cdot (4/3)^{146/2} \approx 2^{32}$ vectors in the main database. This is merely 60% of

Table 3: Details of the Machines Used in the Challenges

Machine	CPUs	base freq.	cores	RAM
D	2xIntel Xeon Gold 6338	2.00Ghz	64	256GB
Y	2xIntel Xeon Platinum 8336C	2.30Ghz	64	256GB
X_1	2xIntel Xeon Platinum 8479	2.00Ghz	112	512GB
X_2	2xIntel Xeon Platinum 8479	2.00Ghz	112	1024GB

the RAM usage of the currently most memory-efficient implementation [DSv21], and only 25% of the RAM usage of previous CPU implementations.

As proof, we present our short vector for the 183-dimensional Darmstadt SVP Challenge with seed 0:

(155, -136, 243, 312, -81, 355, -116, 714, -632, 102, -711, 48, 201, -224, -60, -672,
151, -45, 197, -223, -153, 143, 133, 38, -56, 133, -482, -41, -102, 201, 220, 87, -116,
-141, 116, -690, -246, 104, -209, 152, 422, 165, -51, -452, -308, -366, 424, 122, 308,
-109, -277, -244, -273, 30, 33, 221, -484, -19, -112, 116, 206, -151, 69, 63, 37, 111,
-240, 128, -48, 93, -157, -354, -216, 263, -87, -61, -212, 254, -120, 210, 309, -164,
52, -19, -6, 91, -124, -74, 181, 369, -237, 133, -10, -26, -607, -50, -132, -6, 123,
-345, -130, -147, -3, -64, 174, 65, -375, 57, -673, 466, 83, 51, -465, -254, -8, -221,
17, -159, -142, -524, 24, 284, 99, -32, 492, -95, 251, -68, -108, 29, -577, 984, 301,
111, -58, 394, -102, -330, 17, -225, -151, -46, -35, 381, -211, -24, -207, 304, 133,
-189, -37, 59, 245, -53, 44, -97, -94, 104, -475, 326, 271, -115, -575, -69, -330, 199,
-238, 2, 316, -170, -164, -100, 5, -66, -532, 64, 258, -316, 66, 315, 167, -236, 52)

Table 4: Darmstadt SVP Challenge Results

Dim	D4F	MSD	Norm	Norm/GH	CPU time	Wall time	Machine	Algorithm
100†	18	82	2214	0.87028	44.7s	44.7$s^\ddagger$	D	bgj1
120†	21	99	2654	0.95660	73.7m	73.7$m^\ddagger$	D	bgj2
130†	26	104	2812	0.97516	9.92h	11.2m	D	bgj2
140†	20	120	2875	0.96283	74.1h	77.3m	D	bgj3
150	31	119	3084	0.99791	14.5d	5.70h	D	bgj3 & dh
151	24	127	3195	1.03167	58.2d	22.3h	D	bgj3 & dh
153	20	133	3173	1.01477	109d	41.6h	D	bgj3 & dh
157	23	134	3271	1.03367	185d	70.2h	D	bgj3
161	31	130	3344	1.04346	266d	4.20d	Y	bgj3 & dh
162	26	136	3325	1.03752	211d	3.30d	Y	bgj3 & dh
165*	40	125	3370	1.04215	117d	1.05d	X	bgj3
166*	28	138	3376	1.03988	352d	3.14d	X	bgj3
169*	33	136	3415	1.04120	1.05y	3.43d	X	bgj3
179*	32	147	3523	1.04651	3.40y	11.2d	X	bgj3-amx
183*	34	149	3536	1.04034	9.20y	30d	X	bgj3-amx & dh

† The seed is not zero.
‡ Only a single thread is used.
* The technique in Section 5.2 is enabled for these instances.

As seen in Table 4, even without AMX accelerations, the plain bgj3 implementation is already extremely fast. It solved the 169-dimensional challenge in just 3.43 days using 112 cores. This is several hundred times faster compared to the previous highest record based on CPU, which took eight months with 224 cores to solve a 166-dimensional challenge. Moreover, when AMX acceleration is enabled, bgj3-amx solved the 179-dimensional challenge in only 11.2 days. This is approximately four times faster than the previous 4-GPU implementation in [DSv21], which solved the 180-dimensional challenge in 51.6 days.

We believe direct GPU-CPU time comparisons are not apples-to-apples due to the significantly higher computational power of GPUs. A comparison of our results with previous GPU-based records, in terms of gate count, is summarized in Table 5. It shows our efficiency gain relative to [DSv21] is approximately $2^{4.5}$, as `fp16` $> 2 *$ `int8`.

Table 5: Comparison with Previous GPU Records

Dim	Walltime	Platform	FLOP
179	11.2d	112 cores, no GPU*	$2^{66.0} \approx 2^{13.6} \cdot (3/2)^{179/2}$ int8 operations
183	30d	112 cores, no GPU*	$2^{67.4} \approx 2^{13.9} \cdot (3/2)^{183/2}$ int8 operations
180	51.6d	4 × Nvidia RTX 2080ti	$2^{69.9} \approx 2^{17.3} \cdot (3/2)^{180/2}$ fp16 operations†
186	50.3d	4 × Nvidia A100	$2^{71.4} \approx 2^{17.0} \cdot (3/2)^{186/2}$ fp16 operations‡

[*] The average speed is approximately 70TOPS.
[†] See Table 1 in [DSv21] for more details.
[‡] Unclear how many floating-point operations the 186 took. This number is estimated as $2^{69.9} \cdot (50.3/51.6) \cdot (312/107) \approx 2^{71.4}$, where 312/107 represents the ratio of the theoretical performance of the A100 to the 2080ti.

Moreover, we have integrated our sieving implementation into the BKZ framework [ZD23] and successfully found a vector of length 761113 in a 796-dimensional Ideal-SVP Challenge[PS13]. A detailed comparison with the previous record is presented in Table 6. It is once again a bit subtle to compare a 4-A100 system with a pure CPU server, but it is reasonable to estimate[11] the ratio to be between 10 and 100, whether we are comparing price or computational capability. Therefore, we estimate our efficiency gain to be $(3/2)^{(121-109)/2} \cdot (49/9.9) \cdot 10^{1\sim 2} \approx 2^{9\sim 12}$. Note that for this challenge, we mainly used `bgj3`, which only achieves about 10 TOPS.

Table 6: Comparison with Previous Approximate-SVP Records

Dim	Walltime	Platform	Hermite Factor
796	9.9d	64 cores, no GPU	$1.00834^{796} \approx (\Gamma(\frac{121}{2}+1)^{1/121}/\sqrt{\pi})^{796/121}$
750	49d	48 cores, 4 × Nvidia A100	$1.00878^{750} \approx (\Gamma(\frac{109}{2}+1)^{1/109}/\sqrt{\pi})^{750/109}$

7 Refined Security Analysis

Finally, we present a refined security analysis of lattice-based schemes based on the results from the previous sections. We will mainly focus on Falcon [PFH$^+$] which will be a NIST PQC standard, but this work is also applicable to Kyber [BDK$^+$] and Dilithium [DLL$^+$].

In Falcon's document [PFH$^+$], the BKZ block size B required to forge a Falcon-512 signature is estimated to be 411. The cost of BKZ is computed as $\frac{n^3}{4B^2}$ times the cost of solving the shortest vector problem instances in dimension B, according to [ADH$^+$19]. Taking into account the dimensions for free, the actual sieving dimension B' is estimated to be $B - \left\lfloor \frac{B\ln(4/3)}{\ln(B/2\pi e)} \right\rceil = 374$. Therefore, considering only the first asymptotic term in the complexity of a sieve leads to a number of $\frac{n^3}{4B^2} \cdot (\sqrt{1.5})^{B'} \approx 2^{120.0}$ classical gates. Now the key point is, in Falcon's document, the constant term of the sieving complexity was estimated based on the real performance of the sieving implementation in [ADH$^+$19], which, however, has been significantly reduced in our work.

For example, on the dual Intel Gold 6338 server mentioned in Table 3, a 100-dimensional left progressive sieve with `bgj3` takes only 1056.85 seconds on a single core. Profiling data

[11] On Sept. 24, 14:00 GMT we looked up spot prices on AWS instances closest to us; the cheapest instance for the former is 32.77 USD/hr and for the latter 2.176 USD/hr for a ratio ≈ 15.

indicates that the total number of dot products during the left progressive sieve does not exceed 650G. Therefore, if we model the gate cost of an int8_t FMA operation as $2 \cdot 8^2 + 8 = 136$ classical gates, an upper bound of the gate cost for a 100-dimensional left progressive sieve is

$$650 \cdot 2^{30} \cdot 100 \cdot 136 \approx 2^{23.8} \cdot (\sqrt{1.5})^{100},$$

which suggests the constant term to be at most around $2^{23.8}$. Thus, the total number of gates required to forge a Falcon-512 signature is now estimated to be $2^{120.0} \cdot 2^{23.8} \approx 2^{143.8}$, using the *same methodology as in Falcon's document*, without considering

1. The refined BKZ strategy, for example, as described in [ZD23]. Expected influence on the gate-count estimate: $\mathbf{2^{-4} \sim 2^{-3}}$.

2. The asymptotic slowdown of the BGJ sieve compared to the BDGL sieve, which is at most $2^{(0.311-0.2925)\cdot 274} \approx 2^5$ times more slowdown for sieving dimension 374. Expected influence on the gate-count: $\mathbf{2^0 \sim 2^5}$.

3. The simhash trick [Cha02, FBB+15, Duc18a], which, although not beneficial in our implementation, can significantly reduce the cost if we focus solely on the gate count. It's possible to perform a gate-saving xor-popcnt check on the simhash values before each int8_t dot product, and only compute the dot product if the xor-popcnt check passes. Being conservative, we discount this part.

As a result, we estimate that the minimal number of classical gates required to forge a Falcon-512 signature falls within the range $[2^{139.8}, 2^{145.8}]$. This is not sure to meet NIST's security requirement of 143 bits for level 1 security. More importantly, now the cost of memory access should no longer be used to argue for even one more bit of security in this range. We suggest modifying the parameters for Falcon-512 to balance the difficulty of forgery and key recovery attacks if a security level of 143 bits is truly necessary.

In the end, in Table 7, we present our refined bit security estimates for Falcon, Kyber, and Dilithium, taking into account the three factors mentioned in the previous paragraph. Specifically, an upper bound of $\log_2(\text{gates})$ is estimated as $\log_2(n^3/4B^2) + \log_2(1.5) \cdot B'/2 + 23.8 - 3 + 5$. The BKZ block size B and the sieving dimension B' in the table are directly taken from their respective documents without any modifications.

Table 7: Refined Estimation of Bit Security

	Falcon-512	Kyber-512	Dilithium-2
B	411	413	433
B'	374	375	394
$\log_2(\text{gates})$	$[139.8, 145.8]$	$[140.1, 146.1]$	$[148.5, 154.5]$

8 Conclusion and Future Directions

In this work, we revisit the BGJ sieve, which proves to be of both theoretical and practical interest. We show that the BGJ sieve is inherently memory-friendly and exhibits performance comparable to the state-of-the-art BDGL sieve, at least for problem scales related to cryptography. This could provide a solution to the long-debated issue of estimating RAM access overhead in large-scale, sieving-based lattice attacks. Supported by our implementation, it suggests that the NIST PQC standardization candidate Falcon-512, especially, may not meet NIST's security requirements. Parameter adjustments are recommended if 143 bits security is really necessary. A deeper theoretical analysis of the refined BGJ sieve could be a direction for future work.

From a practical perspective, we have provided a highly optimized implementation of the BGJ sieves based solely on CPU, which even surpasses the current state-of-the-art GPU implementations. The improvement stems from the smaller $o(n)$ term of the BGJ sieve, smaller bucket size, better locality, an improved sieving framework, and a highly optimized implementation. Such an implementation should be helpful for a deeper understanding of these asymptotically faster sieving algorithms.

Finally, we would like to note that our choice to implement the BGJ sieve solely based on CPU does not mean that a GPU implementation is infeasible. Indeed, if one uses a 20 to 30 TB disk to store the main database, places the buckets after the first filter in a 1 to 2 TB system RAM, and puts the buckets after further filters in each RTX 4090 GPU's 24GB with ECC memory, one may be able to do a disk-based bgj3- or bgj4-gpu that gets close to the GPUs' theoretical throughput. This is because the buckets transferred to the GPU RAM are still very large and should not suffer much from the poor bandwidth from system RAM to GPU RAM, which led to a significant slowdown in the case of 2-bdgl_gpu in [DSv21]. We estimate that the 1.05-Hermite-SVP challenge with dimensions ranging from 200 to 210 should be solvable in a reasonable time, for example, on an 8x Nvidia RTX 4090 machine, which has a theoretical peak performance of $8 \cdot 660 = 5280$TOPS for 8-bit precision, with a sieving dimension of 160 to 170. However, for dimensions greater than 210, the challenge should be considered hard due to the bottleneck of computational resources, even though increasing the disk space for a larger sieving dimension is not too difficult.

Nevertheless, such an implementation is by no means easy. Therefore, we have chosen to develop a CPU-based implementation, which is much easier to develop, tune, debug, and use. Moreover, it is sufficient to illustrate most of the concepts we aim to demonstrate, and it is already fast enough to validate the improvements.

Parallel Work. We recently became aware of a parallel work by Samuel Jaques [Jaq24], which also explores the memory access cost of sieving algorithms. [Jaq24] focuses more on the theoretical side, while our research is rooted in and supported by concrete implementation. Another very recent study by Martin R. Albrecht and Joe Rowell [AR24] explores the possibility of distributed sieving implementation. Their work concentrates on the BDGL sieve and the results should be considered orthogonal to ours. Although our approaches and findings are independent and differ significantly, we appreciate the contributions of their research, and believe that both our works collectively advance the understanding of the concrete cost of lattice attacks.

Acknowledgements.

This work is supported by National Key R&D Program of China (No. 2021YFB3100100) and Beijing Natural Science Foundation (No. M22001). Also by Academia Sinica Grand Challenge Program AS-GCP-114-M01.

References

[ADH+19] Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 717–746. Springer, Cham, May 2019.

[AIL+15] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. In C. Cortes,

N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[AKS01] Miklós Ajtai, Ravi Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *33rd ACM STOC*, pages 601–610. ACM Press, July 2001.

[AR24] Martin R. Albrecht and Joe Rowell. Scaling lattice sieves across multiple machines. Cryptology ePrint Archive, Report 2024/747, 2024.

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In Robert Krauthgamer, editor, *27th SODA*, pages 10–24. ACM-SIAM, January 2016.

[BDK$^+$] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms-2022.

[Ber20] D. J. Bernstein. round 2 official comment: crystals-kyber, 2020.

[Ber23] D. J. Bernstein. Structure of memory access in sieving, 2023.

[BGJ13] Anja Becker, Nicolas Gama, and Antoine Joux. Solving shortest and closest vector problems: The decomposition approach. Cryptology ePrint Archive, Report 2013/685, 2013.

[BGJ15] Anja Becker, Nicolas Gama, and Antoine Joux. Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search. Cryptology ePrint Archive, Report 2015/522, 2015.

[Cha02] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *34th ACM STOC*, pages 380–388. ACM Press, May 2002.

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 1–20. Springer, Berlin, Heidelberg, December 2011.

[DLL$^+$] Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms-2022.

[DSv21] Léo Ducas, Marc Stevens, and Wessel P. J. van Woerden. Advanced lattice sieving on GPUs, with tensor cores. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 249–279. Springer, Cham, October 2021.

[Duc18a] Léo Ducas. Shortest vector from lattice sieving: A few dimensions for free. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 125–145. Springer, Cham, April / May 2018.

[Duc18b] Léo Ducas. Shortest vector from lattice sieving: a few dimensions for free (talk), April 2018.

[Duc22] Léo Ducas. Estimating the hidden overheads in the BDGL lattice sieving algorithm. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022*, pages 480–497. Springer, Cham, September 2022.

[FBB$^+$15] Robert Fitzpatrick, Christian H. Bischof, Johannes Buchmann, Özgür Dagdelen, Florian Göpfert, Artur Mariano, and Bo-Yin Yang. Tuning GaussSieve for speed. In Diego F. Aranha and Alfred Menezes, editors, *LATINCRYPT 2014*, volume 8895 of *LNCS*, pages 288–305. Springer, Cham, September 2015.

[GNR10] Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 257–278. Springer, Berlin, Heidelberg, May / June 2010.

[HK17] Gottfried Herold and Elena Kirshanova. Improved algorithms for the approximate k-list problem in euclidean norm. In Serge Fehr, editor, *PKC 2017, Part I*, volume 10174 of *LNCS*, pages 16–40. Springer, Berlin, Heidelberg, March 2017.

[Jaq24] Samuel Jaques. Memory adds no cost to lattice sieving for computers in 3 or more spatial dimensions. Cryptology ePrint Archive, Report 2024/080, 2024.

[Kan83] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *15th ACM STOC*, pages 193–206. ACM Press, April 1983.

[KL21] Elena Kirshanova and Thijs Laarhoven. Lower bounds on lattice sieving and information set decoding. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 791–820, Virtual Event, August 2021. Springer, Cham.

[Laa15] Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 3–22. Springer, Berlin, Heidelberg, August 2015.

[LdW15] Thijs Laarhoven and Benne de Weger. Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *LATINCRYPT 2015*, volume 9230 of *LNCS*, pages 101–118. Springer, Cham, August 2015.

[LLL82] Arjen K. Lenstra, Hendrik W. Lenstra, and László Miklós Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[MV10] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In Moses Charika, editor, *21st SODA*, pages 1468–1480. ACM-SIAM, January 2010.

[NV08] Phong Q. Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2):181–207, 2008.

[PFH$^+$] T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms-2022.

[Poh81] Michael Pohst. On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications. *SIGSAM Bull.*, 15(1):37–44, feb 1981.

[Pop] Gregory Popovitch. Sparsepp: A fast, memory efficient hash map for c++. https://github.com/greg7mdp/sparsepp.

[PS13] Thomas Plantard and Michael Schneider. Creating a challenge for ideal lattices. Cryptology ePrint Archive, Report 2013/039, 2013.

[Sch87] C.P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theor. Comput. Sci.*, 53(2):201–224, jun 1987.

[SE94] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994.

[SG10] Michael Schneider and Nicolas Gama. Darmstadt svp challenges, 2010.

[Sho] Victor Shoup. Ntl: A library for doing number theory. https://libntl.org/.

[TT07] Kengo Terasawa and Yuzuru Tanaka. Spherical lsh for approximate nearest neighbor search on unit hypersphere. In Frank Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *Algorithms and Data Structures*, pages 27–38, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[vW23] Wessel van Woerden. *Lattice cryptography: from cryptanalysis to New Foundations.* PhD thesis, Leiden University, 2023.

[ZD23] Ziyu Zhao and Jintai Ding. Practical improvements on bkz algorithm. In Shlomi Dolev, Ehud Gudes, and Pascal Paillier, editors, *Cyber Security, Cryptology, and Machine Learning*, pages 273–284, Cham, 2023. Springer Nature Switzerland.

A Details of the Bucketing Procedure

We consider a slight variation of Algorithm 3 here: we choose the i-th filter center $\mathbf{c}_i$ to be orthogonal to the previous centers $\mathbf{c}_0, \dots, \mathbf{c}_{i-1}$ for all $i = 1, \dots, k-1$, instead of choosing them uniformly from the unit sphere in $\mathbb{R}^n$. From now on, for any vector $\mathbf{v} \in \mathbb{R}^n$, we denote its entries by $v_0, \dots, v_{n-1}$. For a set A, we mean sampling x uniformly at random from A by writing $x \leftarrow A$. By abuse of language, we denote the canonical measure on high-dimensional spheres by μ.

For sieving dimension n, take the number of filter layers $k = \mathcal{O}(\log n)$, $\alpha = 1/\sqrt{4k}$, then for each $i = 1, \dots, k$ an i-th layer bucket corresponds to regions isomorphic to

$$W_{i,\alpha} := \{\mathbf{x} = (x_0, \cdots, x_{n-1}) \in \mathbb{S}^{n-1} : x_0 > \alpha, x_1 > \alpha, ..., x_{i-1} > \alpha\}$$

Theorem 3 (Collision Probabilities for $W_{i,\alpha}$)**.** *The probability that a random vector lies in $W_{i,\alpha}$ is*

$$\mathbb{P}[\mathbf{v} \in W_{i,\alpha} | \mathbf{v} \leftarrow \mathbb{S}^{n-1}] = n^{\mathcal{O}(i)} \left(1 - i\alpha^2\right)^{n/2+o(n)} = \left(1 - \frac{i}{4k}\right)^{\frac{n}{2}+o(n)}.$$

Furthermore, the probability of a pair uniformly sampled from $W_{k,\alpha} \times W_{k,\alpha}$ being a reducing pair is

$$\mathbb{P}[\langle \mathbf{u}, \mathbf{v}\rangle > 0.5 | \mathbf{u}, \mathbf{v} \leftarrow W_{k,\alpha}] \geqslant 2^{\mathrm{poly}(\log n)} \left(1 - \frac{\left(0.5 - k\alpha^2\right)^2}{\left(1 - k\alpha^2\right)^2}\right)^{n/2+o(n)} = \left(\frac{8}{9}\right)^{\frac{n}{2}+o(n)}.$$

That is, we only lose a $2^{\mathrm{poly}(\log n)}$ factor from the non-spherical filter regions, and thus taking the batch size $B = (\frac{3}{2})^{n/2k+o(n/k)}$, a possible strategy may be:

	number of buckets	bucket size
main database	1	$\left(\frac{4}{3}\right)^{\frac{n}{2}+o(n)}$
i-th layer	B^i	$\left(\frac{4}{3}\right)^{n/2+o(n)}\left(1-\frac{i}{4k}\right)^{n/2+o(n)}$
k-th layer	$B^k = \left(\frac{3}{2}\right)^{n/2+o(n)}$	$2^{o(n)}$

where the gate complexity is $\left(\frac{3}{2}\right)^{n/2+o(n)}$ and we could collect $\left(\frac{3}{2}\right)^{n/2+o(n)}\left(\frac{8}{9}\right)^{n/2+o(n)} = \left(\frac{4}{3}\right)^{n/2+o(n)}$ reducing-pairs from the k-th layer buckets, if the distribution for the k-th layer bucket centers is idealized to be uniform. We are currently not sure whether the non-uniformity of the bucket centers also causes only a subexponential loss as in [BDGL16]. However, if this is not the case, we can increase the batch size B without increasing the main database access.

proof of Theorem 3. For the first part, we have

$$
\begin{aligned}
&\mathbb{P}[\mathbf{v} \in W_{i,\alpha} | \mathbf{v} \leftarrow \mathbb{S}^{n-1}] = \frac{\mu(W_{i,\alpha})}{\mu(\mathbb{S}^{n-1})} \\
&= \frac{1}{\mu(\mathbb{S}^{n-1})} \int_{x_0>\alpha,\cdots,x_{i-1}>\alpha,\sum_{\ell=0}^{n-2} x_\ell^2<1} \frac{2}{\sqrt{1-\sum_{\ell=0}^{n-2} x_\ell^2}} \mathrm{d}x_0 \cdots \mathrm{d}x_{n-2} \\
&= \frac{\mu(\mathbb{S}^{n-i-1})}{\mu(\mathbb{S}^{n-1})} \int_{x_0>\alpha,\cdots,x_{i-1}>\alpha,\sum_{\ell=0}^{i-1} x_\ell^2<1} \left(1-\sum_{\ell=0}^{i-1} x_\ell^2\right)^{\frac{n-i-1}{2}} \mathrm{d}x_0 \cdots \mathrm{d}x_{i-1} \\
&= \frac{\mu(\mathbb{S}^{n-i-1})}{\mu(\mathbb{S}^{n-1})} \int_{\sqrt{i/4k}}^{1} g(r)(1-r^2)^{\frac{n-i-1}{2}} \mathrm{d}r
\end{aligned}
$$

where $g(r)$ is the surface area of $\{x_0 > \alpha, ..., x_{i-1} > \alpha, \sum_{\ell=0}^{i-1} x_\ell^2 = r\}$. We know $\mu(\mathbb{S}^{n-i-1})/\mu(\mathbb{S}^{n-1})$ is $n^{\mathcal{O}(i)}$ from the sphere area formula. It remains to estimate the integral $I := \int_{\sqrt{i/4k}}^{1} g(r)(1-r^2)^{\frac{n-i-1}{2}} \mathrm{d}r$.

On the one hand,

$$
I \leqslant \int_{\sqrt{i/4k}}^{1} g(r)\left(1-\frac{i}{4k}\right)^{\frac{n-i-1}{2}} \mathrm{d}r \leqslant \left(1-\frac{i}{4k}\right)^{\frac{n-i-1}{2}} = \left(1-\frac{i}{4k}\right)^{\frac{n}{2}+o(n)}
$$

On the other hand, taking $\varepsilon = \varepsilon(n) = 1/n$, we have

$$
\begin{aligned}
I &\geqslant \int_{\sqrt{i/4k}}^{\sqrt{i/4k+\varepsilon}} g(r)\mathrm{d}r \cdot \left(1-\frac{i}{4k}-\varepsilon\right)^{\frac{n-i-1}{2}} \\
&= \mathrm{Vol}\left(\left\{\sum_{\ell=0}^{i-1} x_\ell^2 < \frac{i}{4k}+\varepsilon, x_0 > \alpha, ..., x_{i-1} > \alpha\right\}\right) \cdot \left(1-\frac{i}{4k}-\varepsilon\right)^{\frac{n-i-1}{2}} \\
&\geqslant \left(\sqrt{\frac{1}{4k}+\frac{\varepsilon}{i}}-\sqrt{\frac{1}{4k}}\right)^{i} \cdot \left(1-\frac{i}{4k}-\varepsilon\right)^{\frac{n-i-1}{2}} = \left(1-\frac{i}{4k}\right)^{\frac{n}{2}+o(n)}.
\end{aligned}
$$

as desired.

In the proof of the second part, we write $W := W_{k,\alpha}$ for short, and $\mu_{\mathbf{u}}, \mu_{\mathbf{v}}$ for the spherical measures on the first and second spheres, respectively. For $\varepsilon = \varepsilon(n) = 1/n$, denote by $p_\varepsilon = \left(1 - \left(\frac{1}{3-4\varepsilon}\right)^2\right)^{n/2+o(n)}$ the probability that two random vectors in $\mathbb{S}^{n-k-1}$ have a dot product of at least $1/(4(1-(0.25+\varepsilon)))$. We have

$$\begin{aligned}
&\int_{(\mathbf{u},\mathbf{v})\in W\times W} \mathbf{1}_{\langle\mathbf{u},\mathbf{v}\rangle>0.5}\mathrm{d}\mu_{\mathbf{u}}\mathrm{d}\mu_{\mathbf{v}} \\
&= \int_{0.5}^{1}\mathrm{d}h_{\mathbf{u}}\int_{0.5}^{1}\mathrm{d}h_{\mathbf{v}}\int_{(\mathbf{u},\mathbf{v})\in W\times W,\sum_{\ell=0}^{k-1}u_\ell^2=h_{\mathbf{u}}^2,\sum_{\ell=0}^{k-1}v_\ell^2=h_{\mathbf{v}}^2} \mathbf{1}_{\langle\mathbf{u},\mathbf{v}\rangle>0.5}\frac{\mathrm{d}\mu_{\mathbf{u}}}{\mathrm{d}_{h_{\mathbf{u}}}}\frac{\mathrm{d}\mu_{\mathbf{v}}}{\mathrm{d}_{h_{\mathbf{v}}}} \\
&\geqslant \int_{0.5}^{\sqrt{0.25+\varepsilon}}\mathrm{d}h_{\mathbf{u}}\int_{0.5}^{\sqrt{0.25+\varepsilon}}\mathrm{d}h_{\mathbf{v}}\int_{(\mathbf{u},\mathbf{v})\in W\times W,\sum_{\ell=0}^{k-1}u_\ell^2=h_{\mathbf{u}}^2,\sum_{\ell=0}^{k-1}v_\ell^2=h_{\mathbf{v}}^2} \mathbf{1}_{\langle\mathbf{u},\mathbf{v}\rangle>0.5}\frac{\mathrm{d}\mu_{\mathbf{u}}}{\mathrm{d}h_{\mathbf{u}}}\frac{\mathrm{d}\mu_{\mathbf{v}}}{\mathrm{d}h_{\mathbf{v}}} \\
&\geqslant \int_{0.5}^{\sqrt{0.25+\varepsilon}}\mathrm{d}h_{\mathbf{u}}\int_{0.5}^{\sqrt{0.25+\varepsilon}}\mathrm{d}h_{\mathbf{v}}\int_{(\mathbf{u},\mathbf{v})\in W\times W,\sum_{\ell=0}^{k-1}u_\ell^2=h_{\mathbf{u}}^2,\sum_{\ell=0}^{k-1}v_\ell^2=h_{\mathbf{v}}^2} p_\varepsilon\frac{\mathrm{d}\mu_{\mathbf{u}}}{\mathrm{d}h_{\mathbf{u}}}\frac{\mathrm{d}\mu_{\mathbf{v}}}{\mathrm{d}h_{\mathbf{v}}} \\
&= p_\varepsilon\int_{(\mathbf{u},\mathbf{v})\in W\times W,\sum_{\ell=0}^{k-1}u_\ell^2<0.25+\varepsilon,\sum_{\ell=0}^{k-1}v_\ell^2<0.25+\varepsilon}\mathrm{d}\mu_{\mathbf{u}}\mathrm{d}\mu_{\mathbf{v}} \\
&\geqslant p_\varepsilon\int_{(\mathbf{u},\mathbf{v})\in W\times W,u_\ell,v_\ell<\alpha+\varepsilon/3\alpha k,\ell=0,\ldots,k-1}\mathrm{d}\mu_{\mathbf{u}}\mathrm{d}\mu_{\mathbf{v}} \\
&\geqslant p_\varepsilon\mathcal{O}\left(\left(\frac{\varepsilon}{3\alpha k}\right)^{2k}\right)\mu(W)^2
\end{aligned}$$

which concludes the proof.

□