Memory adds no cost to lattice sieving for computers in 3 or more spatial dimensions

Samuel Jaques

University of Waterloo, Waterloo, Canada

Abstract. The security of lattice-based crytography (LWE, NTRU, and FHE) depends on the hardness of the shortest-vector problem (SVP). Sieving algorithms give the lowest asymptotic runtime to solve SVP, but depend on exponential memory. Memory access costs much more in reality than in the RAM model, so we consider a computational model where processors, memory, and meters of wire are in constant proportions to each other. While this adds substantial costs to route data during lattice sieving, we modify existing algorithms to amortize these costs and find that, asymptotically, a classical computer can achieve the previous RAM model cost of $2^{0.2925d+o(d)}$ to sieve a d-dimensional lattice for a computer existing in 3 or more spatial dimensions, and can reach $2^{0.3113d+o(d)}$ in 2 spatial dimensions, where "spatial dimensions" are the dimensions of the physical geometry in which the computer exists.
Under some assumptions about the constant terms of memory access, we estimate increases in bit security between 7 to 23 bits for different Kyber parameter sets and 8 to 22 bits for Dilithium.

1 Introduction

Major families of modern cryptography – learning-with-errors, NTRU, and current fully homomorphic encryption – rely on the hardness of lattice problems. While these lattice problems are asymptotically hard, we need explicit parameters to resist current and future attacks by powerful adversaries. Two of the new standards for post-quantum cryptography from the National Institute for Standards and Technology (NIST), Kyber and Dilithium (aka ML-KEM [Nat23b] and ML-DSA [Nat23c]) attempt a precise security analysis based on the hardness of an attack using the shortest vector problem (SVP) [ABD+21, BDK+21].

In particular, they estimate the cost based on *sieving* algorithms, which require exponentially large memory, and measure costs in the RAM model. The true cost of large scales of memory is a contentious topic with a long history of debate. In this paper we take the position that the RAM model is inappropriate for large-scale algorithms. Large memories create extra costs in terms of signal latency to travel across the memory device, the energy to send such signals, the construction cost for the wires to carry those signals, the construction cost for the bits of memory, and the opportunity cost for all of the above.

To account for these costs, we make the following basic assumptions:

1. Computing machines should have constant ratios of processors to memory to wires;
2. There is a constant $\Delta \in [0, \frac{1}{2}]$ such that any machine with a radius r contains at most $r^{\frac{1}{\Delta}+o(1)}$ processors.

The first assumption is something of folklore in the community, expressed succinctly in [BL12]: "[A chip with many transistors idle] is obviously highly suboptimal: for essentially the same investment in chip area one can build a much more active chip that stores the same data and that

E-mail: sejaques@uwaterloo.ca (Samuel Jaques)

at the same time performs many other useful computations." It is a natural consequence of the area-time model of computation: see [BL12, BK81, Tho80].

The second point is trivially true because we live in three-dimensional space, implying $\Delta \geq \frac{1}{3}$. In many practical instances this can be ignored because the constant factors (hidden in the $o(1)$) make it irrelevant, but we claim that at cryptographic lattice sieving scales, we cannot ignore this term. We particularly focus on $\Delta = \frac{1}{2}$ as the most realistic case, where computers are fundamentally two-dimensional objects because of the need for heat dissipation.

Naively applying the last two constraints suggests that classical sieving should include an extra factor of $2^{0.2075\Delta d+o(d)}$ in the cost, leading to a total cost of $2^{0.396d+o(d)}$ in two dimensions. However, existing lattice sieves were designed and parameterized for the RAM model. If we account for these memory costs in the algorithm design, can we avoid this memory cost?

In [BGJ15] they construct a recursive sieve, and using one level of recursion it costs $2^{0.349d+o(d)}$ in the RAM model, which [Duc18] conjectured to be achievable with a local architecture. Indeed, for the hardware, implementation, and parameter range of the GPU-based implementations in [DSv21], they found that one level of the [BGJ15] algorithm was more effective than than the algorithm which is best in the RAM model [BDGL16]. Both sieves involve two key steps: sort vectors into "filter buckets", then exhaustively search pairs in the filter buckets. It is somewhat clear that this can be spatially local if the size of each filter bucket is the square root of the size of the overall list: if the list has size L, the time to sort it on a two-dimensional architecture is $L^{1/2+o(1)}$, but this is also the time for a fully parallelized exhaustive search of all pairs of vectors in a bucket of size $L^{1/2}$. Thus, the sort adds no time asymptotically.

Recent comments [Nat23a, Sch23] make this observation, and [Nat23a] note that adjusting the parameters of [BDGL16] and [BGJ15] can give similar strategies for d-dimensional architectures.

1.1 Contributions

Asymptotic Results. We extend the ideas of these recent observations by combining the random product codes from [BDGL16] with the recursive strategy of [BGJ15]. Without the recursion, this captures the result of [Nat23a], and we show that this is the optimal parameterization of [BDGL16] under these memory constraints.

With the recursive strategy, we can go further and reach a cost of

$$\max\left\{\sqrt{\frac{3}{2}}, \sqrt{\frac{4}{3}}^{1+\Delta}\right\}^{d+o(d)} = 2^{\max\{0.2925, 0.2075(1+\Delta)\}(d+o(d))}. \tag{1}$$

In particular, this is $2^{0.3113d+o(d)}$ in two dimensions and $2^{0.2925d+o(d)}$ in three dimensions. This "cost" is in area-time (or higher-dimensional analogues of area), and with $2^{0.2075d+o(d)}$ processors this implies a runtime of $2^{\max\{0.085, 0.2075\Delta\}d+o(d)}$.

We argue that this is essentially optimal: the RAM model cost of $2^{0.2925d+o(d)}$, and the cost of $2^{0.2075(1+\Delta)d+o(d)}$ to sort the list of vectors, should act as lower bounds on the cost of a sieve in area-time. Without a more fundamental breakthrough in sieving algorithms, we should not expect to beat the RAM model cost, and the layout of vectors in memory ought to be random enough that a sieve requires at least one sor't's worth of data movement.

Security of Kyber and Dilithium. We then modify the scripts from [AGPS20] to estimate the costs of the new recursive algorithm in 2 dimensions. Here we directly optimize the parameters (namely, filter bucket strength) instead of relying on the asymptotic analysis. The resulting costs are greater than the estimates used in [ABD+21, BDK+21] by 7, 15, and 23 bits for Kyber 512, 768, and 1024, respectively. This suggests security was slightly underestimated, though the estimates from [ABD+21, BDK+21] do not include the advances in [MAT22], which we include thanks

to an update to [AGPS20][1]. Compared to the RAM model estimates from the updated estimator, memory adds a cost of about 14-31 bits, depending on the lattice size.

Specifically, we estimate 2^{146} instead of 2^{137} for the cost of the sieving subroutine of the primal attack against Kyber-512, suggesting the full primal attack will cost 2^{159} operations (if no other aspect of the attack changes). This is almost exactly equal to NIST's estimate of 2^{160} [Nat23a].

These estimates are based on a cost of $2^{-12.8}N^{3/2}$ to route or sort N bits of data. The constant $2^{-12.8}$ is fairly arbitrary, so we also ran cost estimates with a constant of 1. The resulting costs were about 13-29 bits higher, depending on the dimension of the lattice.

Disclaimer. We emphasize that these conclusions apply based on an analysis of memory in *current* lattice sieves. Future work may find better algorithms for the shortest-vector problem, and the security of module LWE in general is much more complex; lattice sieving is just one step. Rather, this work is better seen as an upper bound on the overhead of memory in lattice sieving: asymptotically only $2^{0.019d+o(d)}$ and only 14-31 bits for cryptographically relevant sizes.

1.2 Open problems

This paper only analyzes 2-sieves, but these can be generalized to k-sieves [HK17, HKL18], which have a higher cost in the RAM model but use less memory. One point on the time-memory tradeoff for 3-sieving in the RAM model is $0.305d + o(d)$ time with $0.1907d + o(d)$ memory [CL23]. If we can reach the same conclusion as for 2-sieving – that the cost exponent is the maximum of the RAM model time exponent, and $(1 + \Delta)$ times the memory exponent – then this would be cheaper than 2-sieving. A similar analysis could be done for quantum k-sieves [KMPM19].

An orthogonal advance in LWE attacks is in [Ber23], based on improved meet-in-the-middle attacks. We suspect that our improved memory-aware sieving can be combined straightforwardly with these techniques.

Giving concrete estimates for these attacks is a computational challenge itself, and we had to cut some corners to make the estimation computations feasible. Better optimization of parameters might reduce the concrete bit security slightly, while incorporating the non-randomness overhead from [Duc22] might increase bit security slightly.

1.3 Outline

We give background on lattice sieving and memory assumptions in Section 2. In Section 3 we analyze the re-parameterization of [BDGL16] and a recursive variant, giving the lowest asymptotic costs. In Section 4 we compute concrete costs for Kyber and Dilithium.

2 Background

2.1 Memory Costs

Our main goal is a more realistic cost accounting than the RAM model. In the RAM model, our computer has an instruction set that includes read and write access to fixed-length words from a memory of unbounded size. We instead assign a cost N^{Δ} to access memory if the memory has size N, for a parameter $\Delta \in [0, \frac{1}{2}]$. There are many reasons to justify this cost, especially at the scales we will consider where memory may be 2^{90} bits or more. Such reasons include:

- Latency: Each bit in memory has some physical size, implying the N bits of memory occupy a physical space with radius at least $N^{1/D+o(1)}$ (if the memory is D-dimensional). The average-case time for a signal to propagate from a random bit of memory to a fixed location is proportional to this radius.

[1] Available at https://github.com/jschanck/eprint-2019-1161/commit/a4d3a53fe1f428fe3b4402bd63ee164ba6cc571c

- Energy: Again relying on the size bounds, a signal in a wire will attenuate as it propagates. The total energy lost will be proportional to the length of the wire.
- Area-time: The computer's builder can decide to purchase processors instead of memory or wires, at a fixed (albeit large) constant ratio. Thus, there is a constant opportunity cost to use any component (e.g., fixed lengths of wires, fixed amounts of memory, single processors) for one time step. More concretely, rather than building a wire of length $N^{1/2+o(1)}$, we could have built $N^{1/2+o(1)}$ processors and used them instead.

We will dwell on the area-time cost. Thinking in these terms, a sensible architecture ought to have the number of processors, amount of memory, and amount of wires all roughly proportional to each other. However, this raises connectivity issues: can all processors connect to each other? With P processors, all-to-all connectivity requires $P^{2+o(1)}$ wires. Limited connectivity uses fewer wires, but if each processor has on average one non-local connection (say, length $P^{\frac{1}{2}+o(1)}$ to reach a processor on the other side of the device), the total length of all wires still exceeds P asymptotically.

Thus, we only meet our goal of a constant processor-to-wire ratio with local connections between processors. This justifies a mesh architecture: we have P processors, each with $O(1)$ bits of memory that it can access locally, and $O(1)$ connections to other processors that are physically nearby. The mesh could be three-dimensional, but more likely it would be two dimensional because of heat dissipation: heat must dissipate out of the surface area of of the machine, but this can only grow proportionally to the *square* of the machine's radius.

In this paper we will mainly think in terms of the mesh architecture. The memory access cost is much more direct now: if we let P^{Δ} represent the radius of the mesh (the average length of the shortest path between two nodes), then to move memory from one part of the machine to another the signal must pass through P^{Δ} processors, each of which performs some computation to pass the signal along. Setting $\Delta = 0$ negates these latency concerns, while $\Delta = \frac{1}{3}$ corresponds to a three-dimensional architecture and $\Delta = \frac{1}{2}$ corresponds to a two-dimensional architecture.

Memory access by sorts. A common technique in these mesh architectures is a memory access by a sort. Suppose each processor has one item of a list in its local memory. Using techniques like [SS86] or [Kun87], the machine can sort the list with $P^{1+\Delta+o(1)}$ operations in time $P^{\Delta+o(1)}$. This also works if each processor has $O(1)$ items. We stretch the definition somewhat and allow a logarithmic number of connections for $\Delta = 0$, which allows sorting in poly-log time (e.g., with a bitonic sort).

We can use a sort so that all processors can simultaneously have random access to such a list. The technique is straightforward: each processor j takes its request address i and creates a tuple $(i, 0, \mathsf{request})$. It also takes its item ℓ_j from the list and creates a tuple $(j, \ell_j, \mathsf{mem})$. Then the processors sort all of these tuples by the first component, breaking ties with the last component. This ensures that a request tuple with an address i will be sorted to be physically near the memory tuple $(i, \ell_i, \mathsf{mem})$. Then the processor with both tuples in its local memory can copy the item ℓ_i from one tuple to the other. Once all processors do this (simultaneously), they reverse the sort so that all request tuples are returned to the respective processor.

There are some complications to ensure multiple accesses to the same memory item are handled gracefully, but this is asymptotically solved (e.g., [BBG+13]).

Sort Lower Bound. We prove a brief folklore fact about Δ, namely that if the amount of wire is proportional to the number of processors P, then the radius of the mesh (the number of hops between nodes) is proportional to the physical radius of the device. While lower bounds on sorts in a mesh are well-known (and obvious, since it would take $P^{\Delta+o(1)}$ hops simply to pass a message from one side to the other), we will show that adding extra wires will not improve the time.

This is almost identical to Theorem 1 from [Wie04], with the main difference that we focus on the cost of a random access pattern instead of a worst-case access pattern. This is important for

lattice sieving as we expect the routing necessary to move lattices to their filter bucket will behave like a random permutation.

Define a "parallel architecture" as a sequence of machines, each with P processors, from $P = 1$ to ∞. Each machine has a specific physical layout and connectivity. We define Δ such that in any machine, the maximum number of processors in a radius r around any point is at most $Cr^{1/\Delta}$ for a constant C that may depend on Δ.

We let t_0 be a finite time step and suppose that each wire (i.e., a connection between two processors) can carry one message between those processors in each time step. The routing task we want to solve is that each processor is given a message and the address of another processor, and we want to send all of these messages, with the promise that each address is unique. This is at least as hard as a sort, as the method given previously reduces this problem to sorting.

Proposition 1. *Suppose we have a parallel architecture with wires whose total length is W. Then for all but a negligible fraction of inputs, the routing task defined above requires at least $P^{1+\Delta+o(1)}/W$ time steps.*

Proof. At each time step, each wire in the machine carries at most one message. We can define the distance each message travels as the sum of all lengths of wires that carry that message at some time step. It's clear that if a message travels from processor i to j, this value must be at least as large as the spatial distance between i and j. We can then reason about the sum of all of these distances, denoted D.

Our first claim is that for all but a negligible fraction of address patterns for the messages, $D \geq P^{1+\Delta+o(1)}$. Consider the set S_ϵ of all inputs such that $D \leq P^{1+\Delta-\epsilon}$ for some $\epsilon > 0$. Divide the messages into two types: those that travel a distance greater than L, and those travelling less than L. We will set $L = P^{\frac{\epsilon}{2}}$, and further define $S_{\epsilon,n}$ as the set of all inputs where at most n messages travel further than L. This means $|S_{\epsilon,n}| \leq |S_{\epsilon,n'}|$ for any $n \leq n'$.

Then we (over)count how many possible patterns of addresses produce this. First, we choose which of the P messages will have lengths at least L – there are $\binom{P}{n}$ choices. Then for each of the $P - n$ messages travelling a distance less than L, there are at most $CL^{1/\Delta}$ processors it could reach. Thus, there are at most $\left(CL^{1/\Delta}\right)^{P-n}$ choices for destinations for these messages. For the remaining messages of distance at least L, they might reach any processor, so there are at most P^n choices. Together this gives

$$|S_{\epsilon,n}| \leq \binom{P}{n} P^n (CL^{1/\Delta})^{P-n}. \tag{2}$$

Since $|S_{\epsilon,n}|$ increases in n, we can bound

$$|S_\epsilon| \leq \sum_{n=0}^{n_{max}} |S_{\epsilon,n}| \leq n_{max} \binom{P}{n_{max}} P^{n_{max}} (CL^{1/\Delta})^{P-n_{max}}. \tag{3}$$

To find n_{max}, we see that $D \geq Ln$, since there are at least n messages travelling a distance L. We assumed $D \leq P^{1+\Delta-\epsilon}$ and we chose $L = P^{\frac{\epsilon}{2}}$, giving $n_{max} \leq P^{\Delta-\frac{\epsilon}{2}}$.

Finally, we know that there are $P!$ possible patterns of addresses, as each one induces a unique permutation. Thus, the fraction of messages with $D \leq P^{1+\Delta-\epsilon}$ is at most

$$\frac{n_{max}\binom{P}{n_{max}} P^{n_{max}} (CL^{1/\Delta})^{P-n_{max}}}{P!} = \exp\left(-\frac{\epsilon}{2\Delta} P \log P + o(P)\right) \tag{4}$$

using Stirling's inequality. This decays exponentially in P.

Now we argue that $WT \geq D$. We can let L_i be the sum of lengths of all wires carrying a message at time step i. Because a wire cannot carry two messages in the same time step, we see that the sum of L_i over all time steps must equal D, the total distance travelled by all messages.

Since each L_i is upper-bounded by W, if we use T time steps than we get a bound of $WT \geq D$. However, since the reasoning above shows that $D \geq P^{1+\Delta+o(1)}$ for all but a negligible fraction of inputs, this gives the result.

□

Proposition 1 shows that even if we have some long-range connections, and even if we ignore latency, the time to pass messages arbitrarily across the network will grow proportional to $P^{\Delta+o(1)}$ unless the machine is, asymptotically, almost entirely made of wires. In our cost model, such a machine is suboptimal, and so the time to complete a routing is lower-bounded by the physical radius of the machine.

2.2 Lattice Cryptography

A *lattice* in $\mathbb{R}^n$ is a discrete subgroup of $\mathbb{R}^n$, or equivalently the set of all integer linear combinations of a set of linearly independent vectors $B \in \mathbb{R}^n$. We call such a set a *basis*, and this is the typical representation of a lattice for a computer.

Given a lattice L as a basis B, the *shortest vector problem* (SVP) asks to find the shortest vector in L. Solving this exactly is NP-hard.

Lattice cryptography includes learning-with-errors (LWE) and NTRU, and NIST selected two LWE schemes (Kyber [ABD+21] and Dilithium [BDK+21]) and one NTRU scheme (Falcon [FHK+20]), for standardization. LWE cryptography relies on the hardness of the LWE problem, which has a close connection to SVP: solving LWE reduces to approximately solving SVP, and approximately solving SVP will solve the LWE problem (albeit with a gap between the respective approximation factors). In fact, despite a suite of different algorithms to attack LWE [APS15], Kyber and Dilithium both base their security around the hardness of an attack based on solving SVP.

In brief, the lattice attacks on LWE work by first constructing a lattice from the LWE instance where a moderately short vector solves LWE. A technique known as BKZ (e.g., [CN11]) finds moderately short vectors by solving exact SVP in blocks of much smaller dimension. In this work we only focus on the problem of solving SVP exactly in these blocks, so-called "core-SVP".

There are two main classes of algorithms to solve SVP: *enumeration* and *sieving*. In the RAM model, for a lattice of dimension d, enumeration runs in time $2^{\Theta(d \log d)}$ but with $\mathrm{poly}(d)$ memory, while sieving runs in time $2^{O(d)}$ but uses $2^{O(d)}$ memory. We will only consider sieving.

2.3 Lattice Sieving

Modern lattive sieving is a complex process with many optimizations; see e.g. [ADH+19], [MAT22, DSv21]. In this work we are mostly interested in asymptotics, and so we simplify the description of sieving algorithms for ease of analysis.

As a simplified explanation of the basic idea from [NV08], all two-sieving iterates through a series of lists of lattice vectors $\mathcal{L}_0, \mathcal{L}_1, \ldots$, as follows:

1. Produce an initial list $\mathcal{L}_0$ of random lattice vectors.
2. Repeat for $i = 0, 1, 2, \ldots$ until the vectors in $\mathcal{L}_i$ are small enough:
   (a) Find all *reducing* pairs of vectors $\mathbf{v}, \mathbf{w}$ in $\mathcal{L}_i$: those vectors such that $\|\mathbf{v} - \mathbf{w}\| \leq \gamma \|\mathbf{v}\|$.
   (b) For each reducing pair $\mathbf{v}, \mathbf{w}$, insert the difference $\mathbf{v} - \mathbf{w}$ into $\mathcal{L}_{i+1}$.

This process is parameterized by the sizes of the lists $\mathcal{L}_i$, and the factor γ. Since the lengths of vectors decreases exponentially in each subsequent list, the number of lists is only polynomial in the dimension, and the dominant term in the cost is finding the close vectors. Thus, we parameterize so that γ is as close to 1 as possible.

The analysis relies on Heuristic 1, which breaks down as the the vectors in the list become smaller, but seems to hold for the initial iterations.

Heuristic 1. *The vectors in $\mathcal{L}_i$ are uniformly distributed on a $d-1$-dimensional hypersphere (i.e., the surface of a d-dimensional ball).*

This implies that two vectors are reducing if and only if the angle between them is less than $\pi/3$. This means the probability of two vectors being close is $\sin(\pi/3)^{d+o(d)} = 2^{-0.2075d+o(d)}$.

This fact provides some justification for the heuristic: the probability that two vectors will be at an angle closer than $\pi/3$ decreases exponentially in the dimension. Thus, if a pair of vectors $\mathbf{v}$ and $\mathbf{w}$ do reduce each other, we expect $\|\mathbf{v}-\mathbf{w}\| \approx \gamma\|\mathbf{v}\|$ with high probability. Hence, it seems reasonable to assume that vectors in $\mathcal{L}_i$ all have the same length.

Since the number of *pairs* of vectors in $\mathcal{L}_i$ is $\binom{|\mathcal{L}_i|}{2} = |L_i|^{2+o(1)}$, we expect the number of reducing pairs in $\mathcal{L}_i$, and hence the size of $\mathcal{L}_{i+1}$, to be $|\mathcal{L}_i|^{2+o(1)}2^{-0.2075d+o(d)}$. If this is less than $|\mathcal{L}_i|$, the lists shrink exponentially with each round, and if it's greater than $|\mathcal{L}_i|$, they grow exponentially. Neither is effective, so we choose $|\mathcal{L}_i| = 2^{0.2075d+o(d)}$ so the lists stay at approximately the same size. We will denote this size with L. We will use $\mathcal{L}$ to refer to the list itself.

Already, this approach has a lower bound of $2^{0.2075d+o(d)}$ in memory and time, simply to construct each list of vectors. All remaining design choices go into the methods to find reducing pairs of vectors. For comparison, a brute-force search (as in the original work [NV08]) would take time $L^{2+o(1)} = 2^{0.415d+o(d)}$, and we hope to improve on this.

Locality Sensitive Hashing. Many cryptographic problems look like collision-finding algorithms, and a productive strategy is divide-and-conquer: since the cost is quadratic to compare all pairs of elements in a list, we are better off dividing the list into many sub-lists and looking for pairs in each sublist. This needs to be done in such a way that collisions end up in the same sub-list. The celebrated collision-finding of [vW94] does this with distinguished points, but lattices are more difficult because we seek vectors which are geometrically close, not exactly equal. We need a *locality sensitive* hash.

Introduced to lattice sieving in [Laa15], with a locality sensitive hash we collect vectors with the same hash, which will probably be close to each other, into smaller lists (or "buckets"). We can generalize this to a collection of "filters" (locality sensitive boolean functions), one for each bucket, so that a vector may end up in multiple buckets. A natural type of filter is to choose another vector $\mathbf{c}$ and put all vectors from $\mathcal{L}$ which are sufficiently close to $\mathbf{c}$ into a bucket labelled by $\mathbf{c}$.

Random product codes [BDGL16] provide an efficient method to perform this step, and state-of-the-art classical and quantum lattice sieves use them. A random product code defines a set C of random vectors such that the time to find all vectors $\mathbf{c} \in C$ which are close to a fixed vector $\mathbf{v}$ is proportional to the number of vectors in C which are close to $\mathbf{v}$, not the size of C itself (up to an additive subexponential factor; see [MAT22, Duc22]).

The sieve uses a filter for each codeword $\mathbf{c}$, with parameter α. Given a codeword $\mathbf{c}$, a list vector $\mathbf{v}$ will pass the filter for $\mathbf{c}$ if $|\mathbf{c}\cdot\mathbf{v}| \geq \alpha$.

Suppose that $|C| = L^c$ for some c. We assume the codewords are, like vectors in $\mathcal{L}$, uniformly distributed on the surface of a d-dimensional sphere. Thus, we can define a such that $L^{-a} = (1-\alpha^2)^{-d+o(d)}$ is the probability that two random vectors are α-close [BDGL16]. After iterating through $\mathcal{L}$ to fill the filter buckets, we expect each bucket to contain $L \cdot L^{-a} = L^{1-a}$ vectors on average, and each vector in $\mathcal{L}$ to be placed in L^{c-a} buckets. This means the cost to fill all the buckets is

$$L^{1+o(1)} \cdot \max\{L^{c-a}, L^{o(1)}\} \tag{5}$$

while the memory cost to store all the buckets and their contents will be

$$L^{c+1-a+o(1)}. \tag{6}$$

We will also want to sort the buckets in some way, so that either the vectors in each bucket are physically close, or that we know which memory addresses correspond to which bucket. Such a

sort will thus cost [Kun87]

$$\max\{L^{1+\Delta+o(1)}, L^{(1+\Delta)(1+c-a)+o(1)}\}. \tag{7}$$

The reason for the first term is that if we have $c - a < 0$ then some vectors will not end up in any buckets, but we will still need to sort the output from all input vectors, even if they are not in any bucket.

Finding all solutions. Broadly, we have two main approaches to find all reducing pairs from the bucket. In a "vector-first" approach, we can iterate over each vector $\mathbf{v}$ in $\mathcal{L}$, decode to each codeword $\mathbf{c}$ close to $\mathbf{v}$, then compare $\mathbf{v}$ to all $\mathbf{w}$ in the filter buckets for each $\mathbf{c}$. Alternatively, in a "bucket-first" approach, we iterate over all filter buckets and compare all pairs of vectors in each filter bucket. In the RAM model these lead to equivalent costs, and the vector-first approach clearly requires more long-range memory access, so we will only consider the bucket-first approach.

We now consider how many solutions we find. Specifically, for $\kappa \in [0, \frac{1}{3}]$ and $\alpha \in [0, \sqrt{\kappa}]$, we want the probability that two vectors $\mathbf{v}$ and $\mathbf{w}$ satisfy $\langle \mathbf{v}, \mathbf{w} \rangle \geq \kappa$, given that they are both α-close to a uniformly random codeword $\mathbf{c}$. We define a variable t (implicitly a function of α and κ) such that L^{-t} equals this probability. From [BDGL16], it can be expressed as:[1]:

$$L^{-t} = \left(\frac{(1-\alpha^2)^2}{(1-\kappa)(1+\kappa-2\alpha^2)} \right)^{-d/2+o(d)}. \tag{8}$$

Whatever the method, for each codeword, we will end up checking all pairs of vectors which are close to the codeword. That means we check

$$L^{c+2(1-a)+o(1)} \tag{9}$$

pairs of vectors, so the expected number of reducing pairs is

$$L^{c+2(1-a)-t+o(1)}. \tag{10}$$

Recall that we need L reducing pairs to replenish the list. It could be that we choose c such that $L^{2+c-a-b-t+o(1)} < L$, in which case we would select a different random product code and try again. The average number of repetitions will be

$$\frac{L}{L^{2+c-2a-t+o(1)}} = L^{2a+t-1-c+o(1)}. \tag{11}$$

Putting this all together, the cost to sieve is

$$\max\{1, L^{2a+t-1-c+o(1)}\} \left(L^{1+\Delta+o(1)} + L^{(1+\Delta)(1+c-a)+o(1)} + L^{c}\mathsf{Bucket} \right). \tag{12}$$

where Bucket is the cost to search each bucket. The remainder of the paper will focus on different approaches to search the buckets.

To help analyze these costs, we will use the following Lemma:

Lemma 1. *With a and t defined as in Section 2, for $\lambda \in [0, 1)$, $L^{\lambda a+t}$ is either non-decreasing in α or has a local minimum in α between 0 and 1. Specifically:*

- *if $\lambda = 0$, L^t is non-decreasing in α.*
- *if $\kappa = \frac{1}{3}$, then $L^{a+t} \geq L$ for all α.*

[1] Often quoted as the probability that a codeword will be close to the two vectors of a reducing pair; a simple application of Bayes' rule brings it to this form.

Proof. We can express

$$L^{\lambda a+t} = \left(\frac{(1-\alpha)^{2-\lambda}}{(1-\kappa)(1+\kappa-2\alpha^2)}\right)^{d/2+o(d)} \tag{13}$$

and simply take the derivative in terms of α. The derivative has the form

$$-C(2\alpha^2(\lambda-1)+2\kappa-\lambda\kappa-\lambda) \tag{14}$$

where C is a non-negative function of α. We see that for $\lambda \leq 1$, the derivative is increasing for $\alpha \geq 0$ with potentially one root at $\alpha = \sqrt{\frac{\kappa(2-\lambda)-\lambda}{2(1-\lambda)}}$. For $\lambda \leq 1$ (specifically including the case of $\lambda = 0$), this root does not exist (and thus $L^{\lambda a+t}$ is non-decreasing) if $\frac{2\kappa}{1+\kappa} \leq \lambda$. If the root (a minimum of $L^{\lambda a+t}$) does exist, it is at most 1 when $\lambda \leq 2$.

If $\lambda = 1$, then the cost is non-decreasing so the minimum is at $\alpha = 0$, but this is precisely equal to $(\frac{4}{3})^{d/2+o(d)} = L$. □

3 Improved Memory-aware Sieves

Recall that our cost is given by Equation (12). We analyze two methods to search the buckets.

3.1 Exhaustive Bucket Search

We first consider an exhaustive search, which is essentially just a reparameterization of [BDGL16]. Each bucket has size $L^{1-a+o(1)}$, so it will require $L^{2(1-a)+o(1)}$ operations to search it. We can ignore memory constraints for searching within the buckets by imagining any arrangement of processors which admits of a Hamiltonian cycle of local connections (such as a d-dimensional mesh). Each processor holds one vector $\mathbf{v}_i$ throughout the search, and sends a copy of that vector to the next processor in the cycle. Once a processor compares an incoming vector $\mathbf{v}_j$ to its local vector $\mathbf{v}_i$, it sends $\mathbf{v}_j$ to the next processor in the cycle. After $L^{1-a+o(1)}$ iterations, all pairs have been compared.

Theorem 1. *Including the costs for memory movement, the sieve of [BDGL16] achieves a cost of*

$$\left(\frac{4^{1+2\Delta}\cdot 3}{3^{1+2\Delta}\cdot 4 - 4^2 3^{2\Delta} + 12^{1+\Delta}}\right)^{d/2+o(d)} \tag{15}$$

or $2^{r(\Delta)d+o(d)}$ *where* $r(0) = 0.2925$*,* $r(\frac{1}{3}) = 0.3294$*, and* $r(\frac{1}{2}) = 0.3495$*. These costs are obtained at* $\alpha = \sqrt{1-(\frac{3}{4})^{1-\Delta}}$ *and* $c = \min\{\left(\frac{3}{2}\right)^{d/2+o(d)}, a\}$*.*

Proof. Substituting a cost of $L^{2-2a+o(1)}$ for Bucket in Equation (12) gives

$$\max\{L^{1+\Delta+o(1)}, L^{(1+\Delta)(1+c-a)+o(1)}, L^{c+2-2a+o(1)}\}. \tag{16}$$

The middle term grows fastest in c and is increasing in c, so we should take $c \leq \max\{a, \frac{1-\Delta}{\Delta}(1-a)\}$ to ensure that the middle term is not the largest cost. On the opposite end, if $c \leq 2a+t-1$ and we use multiple codes, the cost is non-increasing in c so we should take the maximum allowable c in this regime. If $c \geq 2a+t-1$ and we use one code, the cost is non-*decreasing* in c so we should take the minimum allowable c. Thus, we should set $c = 2a+t-1$ unless $2a+t-1 > \max\{a, \frac{1-\Delta}{\Delta}(1-a)\}$, noting that Lemma 1 implies that $2a+t-1 > a$ for any parameters.

If we assume $2a+t-1 \leq \frac{1-\Delta}{\Delta}(1-a)$, setting $c = 2a+t-1$ gives a total cost of $\max\{L^{1+\Delta+o(1)}, L^{1+t+o(1)}\}$.

Otherwise, we set $c = \frac{1-\Delta}{\Delta}(1-a)$, but then the cost is also $\max\{L^{1+\Delta+o(1)}, L^{1+t+o(1)}\}$.

If $c = a$, the cost is $L^{1+t+\Delta+o(1)}$. This cost is greatest, so we want to avoid this regime, but doing so requires $a \leq \frac{1-\Delta}{\Delta}(1-a)$, or $a \leq 1-\Delta$.

In all cases, the cost depends on t but not a, so by Lemma 1, we want to take the maximum possible α to minimize it. Under the constraint that $a \leq 1-\Delta$, this gives us

$$\alpha = \sqrt{1 - \left(\frac{3}{4}\right)^{1-\Delta}}. \tag{17}$$

Substituting in these values for α, c, and $\kappa = \frac{1}{3}$ gives the main result. □

As expected, when $\Delta = 0$ this matches the behaviour of the vector-first sieve and the RAM model cost. However, the costs diverge for larger Δ. For $\Delta = \frac{1}{2}$, the optimal α gives buckets whose size is $L^{\frac{1}{2}+o(1)}$, which matches the size from [BGJ15]. For $\Delta = \frac{1}{k}$ for $k \geq 2$, the costs and the algorithm itself match precisely what [DSv21] propose. That is, the optimal parameters for sieving with GPUs were the same as the asymptotically optimal parameters for sieving with memory constraints. This result is also in [Nat23a], more or less.

3.2 Recursive Algorithm

Because the optimal buckets are exponentially large, a recursive strategy will be more effective. We can define the following problem:

Problem 1 (SphereFind(n, d, κ))**.** *Let $\mathcal{N}$ be a list of $N = n^{d/2}$ vectors randomly distributed on the surface of a sphere of d dimensions. Find all pairs of vectors $\boldsymbol{v}, \boldsymbol{w} \in \mathcal{N}$ such that $\langle \boldsymbol{v}, \boldsymbol{w} \rangle \geq \kappa$.*

We say that a SphereFind problem is *sparse* if $n \leq \frac{1}{1-\kappa^2}$. This criteria ensures the total number of solutions does not exceed N, so that we do not need extra memory to store all the solutions.

The core subroutine of lattice sieving is solving SphereFind$(\frac{4}{3}, d, \frac{1}{2})$. We defined n such that $N = n^{d/2}$ to avoid square roots in the notation.

We will use random products as in [BDGL16], but use a recursive strategy to search each bucket as in [BGJ15]. This is expressed in Algorithm 1, which is essentially the same as Algorithm 3 in [BGJ15] except we specifically analyze random product codes and re-order the steps to minimize memory movement.

The parameter ℓ controls how deep the recursion goes, and we will determine this later.

Strictly speaking, Algorithm 1 is not correct, because the solutions returned at each step are pairs of projections of vectors. To fix this, we imagine each vector is stored with a data structure that also contains some "original vector", from the first recursive call. In the final exhaustive search, these original vectors are compared and returned. This makes Algorithm 1 trivially correct, i.e., all solutions returned will be reducing pairs.

Completeness is easy to show in an asymptotic sense by noting that there is some non-zero probability that two reducing pairs will be captured by all the layers of filters. To bound the runtime more carefully, we will first make a heuristic assumption related to the probability that two vectors in an α-filter will be close to the surface of this cap.

More precisely, for two vectors $\mathbf{v}$ and $\mathbf{w}$ in a single filter bucket around a codeword $\mathbf{c}$, we can let $\mathbf{v} = \alpha_v \mathbf{c} + \sqrt{1-\alpha_v^2}\mathbf{v}'$ and $\mathbf{w} = \alpha_w \mathbf{c} + \sqrt{1-\alpha_w^2}\mathbf{w}'$ for unit vectors $\mathbf{v}'$ and $\mathbf{w}'$. The assumption underlying Algorithm 1 is that $\langle \mathbf{v}, \mathbf{w} \rangle \geq \kappa$ if and only $\langle \mathbf{v}', \mathbf{w}' \rangle \geq \frac{\kappa-\alpha^2}{1-\alpha^2}$. Neither direction is exactly true; however, the problems only arise when $\langle \mathbf{v}, \mathbf{w} \rangle < \kappa$ or α_v or α_w are significant smaller than α. A common heuristic in lattice sieving is that if two random vectors are within some fixed angle of each other, they are almost exactly at the boundary, since the surface of a high-dimensional ball occupies most of its volume. We can make a more precise claim for this exact case:

Heuristic 2. *Given three random unit vectors $\boldsymbol{v}$, $\boldsymbol{w}$, and $\boldsymbol{c}$ with $\langle \boldsymbol{v}, \boldsymbol{w} \rangle \geq \kappa$, $\langle \boldsymbol{v}, \boldsymbol{c} \rangle \geq \alpha$, and $\langle \boldsymbol{w}, \boldsymbol{c} \rangle \geq \alpha$, then with $\boldsymbol{v}'$ and $\boldsymbol{w}'$ defined as above, $\langle \boldsymbol{v}', \boldsymbol{w}' \rangle \geq \frac{\kappa-\alpha^2}{1-\alpha^2}$ with probability $\Omega(1)$ in d.*

Algorithm 1 SphereFilter

1: **Parameters**: A maximum depth $\ell \geq 0$, a filter angle $\alpha \in (0,1)$
2: **Input**: A list $\mathcal{N}$ of d-dimensional vectors, dimension d, parameter $\kappa \in (0, \frac{1}{2})$, an integer depth (default value 1)
3: **Output**: A list of all pairs $(\mathbf{v}, \mathbf{w}) \in \mathcal{N}^2$ such that $\langle \mathbf{v}, \mathbf{w} \rangle \geq \kappa$.
4: **if** depth $\geq \ell$ **then**
5: Exhaustively search $\mathcal{N}$ and return
6: **end if**
7: Solutions $\leftarrow \emptyset$
8: **while** $|\text{Solutions}| \leq \left(\frac{n}{1-\kappa^2}\right)^{d/2}$ **do**
9: Select a random product code $\mathcal{C}$ of size $(1-\alpha^2)^{-d/2+o(d)}$
10: Decode each $\mathbf{v} \in \mathcal{N}$ to all codewords $\mathbf{c}$ that it is α-close to
11: Sort all pairs $(\mathbf{v}, \mathbf{c})$ by $\mathbf{c}$; let $B_{\mathbf{c}}$ be the set of all vectors paired with $\mathbf{c}$
12: **for all** $B_{\mathbf{c}}$ with $\mathbf{c} \in \mathcal{C}$ **do**
13: Construct $B'_{\mathbf{c}} = \{\mathbf{v} - \text{proj}_{\mathbf{c}}(\mathbf{v}) : \mathbf{v} \in B_{\mathbf{c}}\}$ and normalize all vectors in $B'_{\mathbf{c}}$
14: Add SphereFilter($B'_{\mathbf{c}}, d-1, \frac{\kappa-\alpha^2}{1-\alpha^2}$, depth $+ 1$) to Solutions
15: **end for**
16: Remove all pairs $(\mathbf{v}, \mathbf{w})$ from Solutions with $\langle \mathbf{v}, \mathbf{w} \rangle < \kappa$.
17: **end while**
18: Return Solutions

We show in the appendix how to numerically compute this probability, and justify this heuristic numerically by showing that in dimension 375, the probability is at least 0.5. More intuitively, we can recall from Section 2.3 that the probability that two vectors in a single filter bucket are reducing is

$$\left(\frac{(1-\alpha^2)^2}{(1-\kappa)(1+\kappa-2\alpha^2)}\right)^{-d/2+o(d)} \tag{18}$$

and the probability that two random unit vectors are $\frac{\kappa-\alpha^2}{1-\alpha^2}$-close is

$$\left(1-\left(\frac{\kappa-\alpha^2}{1-\alpha^2}\right)^2\right)^{d/2+o(d)} \tag{19}$$

and these are the same up to subexponential factors. Thus, the expected number of false positives and false negatives should be subexponential in d.

Starting with a fixed (n_0, κ_0), if we choose a particular $\alpha^2 \in (0,1)$ then the subproblem is SphereFind$(n, d-1, \kappa)$ where $n = (1-\alpha^2)n_0$ and $\kappa = \frac{\kappa_0-\alpha^2}{1-\alpha^2}$. In fact, if we recurse, these are the only parameters we will encounter:

Lemma 2. *Given an instance of SphereFind(n_0, κ_0), all recursive sub-problems are instances of SphereFind(n, d', κ) for $d' < d$ and (n, κ) such that $n = (1-x)n_0$ and $\kappa = \frac{\kappa_0-x}{1-x}$ for $x \in [0, \min\{1-\frac{1}{n_0}, \kappa_0\}]$.*

Proof. We prove inductively, with the trivial base case being (n_0, κ_0).

Suppose it holds up to m recursive calls, and the current problem has parameters $(n, \kappa) = (n_0(1-x), \frac{\kappa_0-x}{1-x})$. For the $m+1$ call, we make filter buckets with parameter $\alpha \in [0,1]$, so the new n is $n(1-\alpha^2) = n_0(1-x)(1-\alpha^2) = n_0(1-(x+\alpha^2-x\alpha^2))$. We thus take $x' = x+\alpha^2-x\alpha^2$.

The new κ is

$$\kappa' = \frac{\kappa-\alpha^2}{1-\alpha^2} = \frac{\frac{\kappa_0-x}{1-x}-\alpha^2}{1-\alpha^2} = \frac{\kappa_0-(x+\alpha^2-x\alpha^2)}{1-(x+\alpha^2-x\alpha^2)} = \frac{\kappa_0-x'}{1-x'} \tag{20}$$

giving the result. □

Corollary 1. *Any recursive subproblem of a sparse SphereFind instance is also sparse.*

Proof. Sparse means $n \leq \frac{1}{1-\kappa^2}$. Thus, when we have $n' = (1-x)n$ and $\kappa = \kappa'(1-x) + x$ we can show

$$n' = (1-x)n \leq \frac{1-x}{1-\kappa^2} = \frac{1}{1-\kappa'^2 + x(\kappa'-1)^2} \leq \frac{1}{1-\kappa'^2}. \tag{21}$$

□

Theorem 2. *For any $\epsilon > 0$, SphereFilter (Algorithm 1) can solve SphereFind(n, d, κ) with cost*

$$\left(\max\{n^{1+\Delta+\epsilon}, \Gamma(n,\kappa)n^{\epsilon}\}\right)^{d/2+o(d)}, \tag{22}$$

where

$$\Gamma(n,\kappa) = \frac{\kappa+1}{\kappa + 2n^{-1} - 1}. \tag{23}$$

Proof. The choice of code size ensures that the total memory in all filter buckets is the same for each recursive instance. By Corollary 1, the total number of solutions in any filter bucket will not exceed the size of the filter bucket. This means the total memory use is $|\mathcal{N}|^{1+o(1)}\ell$.

First notice that by Heuristic 2, the solutions in each subroutine have $\Omega(1)$ false negatives. This means each level of recursion must repeat a constant number of times. Altogether this implies a $2^{O(\ell)}$ time overhead, but we will find that this is constant in d.

Similarly, we may obtain some false positives from each recursive call, but by the same reasoning as that following Heuristic 2, the overhead from this will be at most subexponential in d.

Thus, up to $o(d)$ factors in the exponent, we can assume that the recursive calls have neither false positives nor false negatives.

Given this assumption, let $\mathsf{SF}(n,\kappa)$ be the cost of SphereFilter with an input list of size $n^{d/2}$ and κ. Letting N_c be the necessary number of repetitions (i.e., the number of codes), letting C be the size of each code, and noting that each filter bucket will have size $(n(1-\alpha^2))^{d/2+o(d)}$, if SphereFilter recurses it will have cost

$$\mathsf{SF}(n,\kappa) = N_c\left(n^{(1+\Delta)(d/2+o(d)} + C\cdot \mathsf{SF}\left(n(1-\alpha^2), \frac{\kappa-\alpha^2}{1-\alpha^2}\right)\right). \tag{24}$$

The first term is the sort cost (since the list has size $n^{d/2+o(d)}$) and the second term is the cost of solving all the recursive subproblems.

If SphereFilter does not recurse it has cost $\mathsf{SF}(n,\kappa) = (n^2)^{d/2+o(d)}$ via a quadratic exhaustive search.

To analyze N_c, recall:

- We set $C = (1-\alpha^2)^{-d/2+o(d)}$.
- There are $(n^2(1-\kappa^2))^{d/2+o(d)}$ expected solutions.
- Each pair of vectors in each filter bucket has a probability $\left(\frac{(1-\alpha^2)^2}{(1-\kappa)(1+\kappa-2\alpha^2)}\right)^{-d/2+o(d)}$ of being reducing.

This means the expected number of solutions from each filter bucket is

$$(n^2(1-\alpha^2))^{d/2+o(d)}\left(\frac{(1-\alpha^2)^2}{(1-\kappa)(1+\kappa-2\alpha^2)}\right)^{-d/2+o(d)} \tag{25}$$

$$= \left(\frac{1-\alpha^2}{n^2(1-\kappa)(1+\kappa-2\alpha^2)}\right)^{-d/2+o(d)} \tag{26}$$

This gives a total number of codes we must try as

$$N_c = \frac{(n^2(1-\kappa^2))^{d/2+o(d)}}{C\left(\frac{1-\alpha^2}{n^2(1-\kappa)(1+\kappa-2\alpha^2)}\right)^{-d/2+o(d)}} = \frac{(1+\kappa)(1-\alpha^2)}{1+\kappa-2\alpha^2} \tag{27}$$

Substituting into Equation 24 and ignoring subexponential factors gives a cost of

$$\mathsf{SF}(n,\kappa)^{2/d} = \frac{(1+\kappa)(1-\alpha^2)}{1+\kappa-2\alpha^2}\max\left\{n^{1+\Delta}, (1-\alpha^2)^{-1}\mathsf{SF}\left(n(1-\alpha^2), \frac{\kappa-\alpha^2}{1-\alpha^2}\right)^{2/d}\right\} \tag{28}$$

The rest of the proof is simply solving this recursion.

Because we repeat this process using the same α to define the filter buckets at each step, then if n_i and κ_i are the parameters in the ith step, we can show inductively that

$$\kappa_i = \frac{\kappa + (1-\alpha^2)^i - 1}{(1-\alpha^2)^i},\ n_i = n(1-\alpha^2)^i \tag{29}$$

by noting that $\kappa_{i+1} = \frac{\kappa_i - \alpha^2}{1-\alpha^2}$ and $n_{i+1} = n_i(1-\alpha^2)$.

After ℓ recursions, we stop and the cost is $\mathsf{SF}(n_\ell, \kappa_\ell) = (n_\ell^2)^{d/2+o(d)}$. Substituting this gives the following total cost, letting $\kappa_0 = \kappa$ and $n_0 = n$:

$$\max\left\{\max_{0\leq i\leq \ell-1}\left\{n^{1+\Delta}(1-\alpha^2)^{1+i(1+\Delta)}\prod_{j=0}^{i}\frac{1+\kappa_j}{1+\kappa_j-2\alpha^2}\right\},\right.$$
$$\left. n^2(1-\alpha^2)^{2\ell}\prod_{j=0}^{\ell-1}\frac{1+\kappa_j}{1+\kappa_j-2\alpha^2}\right\} \tag{30}$$

We can then use our formula for κ_i to show that

$$\frac{1+\kappa_j}{1+\kappa_j-2\alpha^2} = \frac{\kappa+2(1-\alpha^2)^i-1}{\kappa+2(1-\alpha^2)^{i+1}-1} \tag{31}$$

and this means all intermediate terms in this product cancel out:

$$\prod_{j=0}^{i}\frac{1+\kappa_j}{1+\kappa_j-2\alpha^2} = \frac{\kappa+1}{\kappa+2(1-\alpha^2)^{i+1}-1} \tag{32}$$

giving us a cost of

$$\max\left\{\max_{0\leq i\leq\ell-1}\left\{n^{1+\Delta}\frac{(1-\alpha^2)^{1+i(1+\Delta)}(\kappa+1)}{\kappa+2(1-\alpha^2)^{i+1}-1}\right\}, n^2(1-\alpha^2)^{2\ell}\frac{\kappa+1}{\kappa+2(1-\alpha^2)^\ell-1}\right\} \tag{33}$$

In the middle term, one can show that the cost either increases in i or has a local minimum; in either case, the maximum will be found at either $i=0$ or $i=\ell$, giving a cost of

$$\max\left\{n^{1+\Delta}\frac{(\kappa+1)(1-\alpha^2)}{1+\kappa-2\alpha^2}, n^{1+\Delta}\frac{(\kappa+1)(1-\alpha^2)^{1+(\ell-1)(1+\Delta)}}{\kappa+2(1-\alpha^2)^\ell-1},\right.$$
$$\left. n^2(1-\alpha^2)^{2\ell}\frac{\kappa+1}{\kappa+2(1-\alpha^2)^\ell-1}\right\} \tag{34}$$

Let $\epsilon > 0$. Let $\alpha_0^2 > 0$ be small enough such that

$$\frac{(\kappa+1)(1-\alpha_0^2)}{1+\kappa-2\alpha_0^2} \leq n^\epsilon. \tag{35}$$

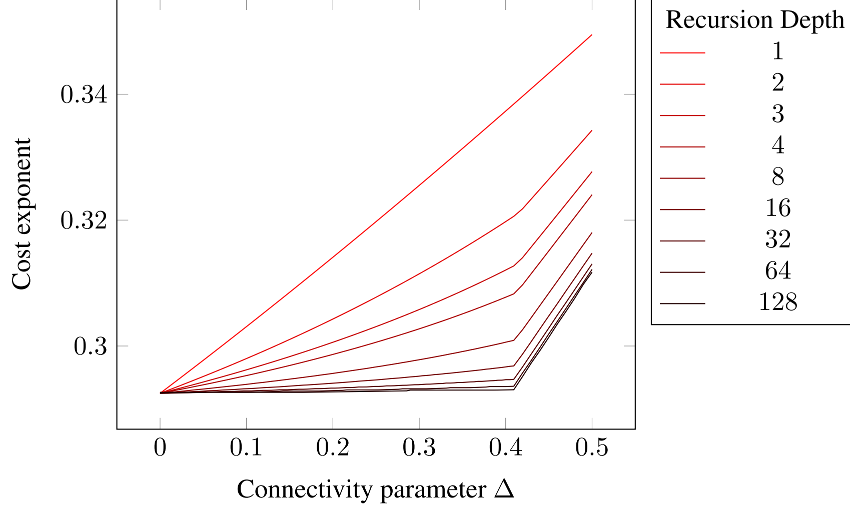

Figure 1: The leading exponent c in the cost $2^{cd+o(d)}$ for the recursive sieve as a function of the connectivity parameter Δ. Each curve is a different depth of recursion ("ℓ" in Algorithm 1); 1 is the approach from Section 3.1.

which is always possible as the left term converges to 1. Then choose ℓ_0 as the minimum value such that $n^{\frac{\Delta}{\ell_0}} \leq n^{\epsilon}$. Finally, choose $\alpha \leq \alpha_0$ and $\ell \geq \ell_0$ such that $(1-\alpha^2)^{\ell} \leq n^{-1}$. Substituting these values gives the desired result:

$$\max\left\{n^{1+\Delta+\epsilon}, \frac{\kappa+1}{\kappa+2n^{-1}-1}n^{\epsilon}\right\} \tag{36}$$

with the final term in Equation 34 being less than the middle term.

Finally, we note that ℓ and α depend only on κ, n, and ϵ, not on d. This means we are free to replace the actual runtimes with their asymptotic expressions: since the number and form of these expressions does not depend on d, we can simply choose the maximum dimension d such that all the asymptotic expressions hold. We also see that all the overhead terms that depend on ℓ or ϵ (e.g., the memory overhead) can be included in the $o(d)$ term in the exponent. □

Corollary 2. *For any $\Delta \in [0, \frac{1}{2}]$, there is an algorithm to solve SVP at cost*

$$2^{\max\{0.2925, 0.20752(1+\Delta)\}d+o(d)} \leq 2^{0.3113d+o(d)} \tag{37}$$

Proof. Using the previous theorem, one can show that $\Gamma(\frac{4}{3}, \frac{1}{2}) = \frac{3}{2}$, and since $\log_2(\sqrt{\frac{3}{2}}) < 0.2925$, the result follows. The truncated decimal expansion hides the factor of $\epsilon > 0$. □

The unusual conclusion is that for $\Delta \leq 0.4094$ (or dimension at least 2.45), *all* the latency costs can be amortized away with such a recursive algorithm.

The second requirement on ℓ is that $n^{\frac{\Delta}{\ell}} \leq n^{\epsilon}$. This means that the number of recursions increases with Δ, and in fact it immediately shows that when $\Delta = 0$ we need only one recursion, precisely capturing the existing result of [BDGL16] in the RAM model.

In Figure 1, we show how the exponent decreases as a function of the number of levels of recursion by numerically optimizing α.

3.3 Discussion

To explain somewhat more intuitively, the choice of filter strength gives a trade-off: stronger filters are easier to search because the buckets are smaller, but are less likely to catch any given reducing

pair. In our memory-constrained regime the size of any one code is limited (since we do not want the memory for all filter buckets to exceed the original list), so stronger filters require more codes.

The problem with more codes is that we need to re-sort the list for each code. With high memory costs, this sorting is the expensive step. Hence, we parameterize so that we have weaker filters and use fewer codes.

As shown in Lemma 2, recursive weak filtering produces buckets which are identical to the buckets obtained after one strict filter. What advantage do we gain from using the layered filter structure? The key difference is the arrangement of the buckets themselves in memory. That is, in the [BDGL16] sieve, the memory layout of filter buckets is effectively randomized. Sorting buckets in a reasonable way would be hard because it echoes the fundamental problem of lattice sieving; namely, that there is no total order on d-dimensional vectors.

However, the layered filtering means in each level of recursion, the code vectors are all close to each other (as vectors in $\mathbb{R}^d$) because they are in the same filter from the previous level. They are *also* physically close to each other in memory because they are in the same filter. Thus, code vectors which are close as vectors also end up close in memory. That way, the buckets can be merged together and re-filtered with a new code with minimal data movement.

In fact, an implementation of this strategy could decode a vector to all filters simultaneously, then with one sort, all vectors would be in the correct filter buckets at the lowest level of recursion. If we label codewords by a hash of length $O(\log |C|)$, then this is a small amount of extra memory per vector.

4 Concrete Costs

4.1 Optimizing parameters

The previous sections gave asymptotically optimal parameters; however, for fixed problem sizes, the precise parameters (filter angles, code sizes, product code structure, etc.) are more difficult to optimize, especially for the recursive approach.

We adapt the code from [AGPS20] to account for memory costs and permit a recursive strategy, and use the following techniques to efficiently find optimal parameters[1]. To account for Heuristic 2, we explicitly compute the false negative rate with an approach detailed in Appendix A.

There are three main costs to the sieve: the "query cost" to decode all vectors into their respective filter buckets; the "routing cost" to route the vectors in the same filter buckets to contiguous regions of memory, and the "search cost" to find all reducing pairs in one filter bucket. All of these costs are multiplied by the expected number of codes.

If the cost to search a bucket of size N is N^γ for $1+\Delta \leq \gamma \leq 2$, then the costs are, respectively:

- Query: $\max\{1, L^{2a+t-1-c+o(1)}\} \cdot L^1$
- Routing: $\max\{1, L^{2a+t-1-c+o(1)}\} \cdot (L^{1+\Delta+o(1)} + L^{(1+\Delta)(1+c-a)+o(1)})$
- Search: $\max\{1, L^{2a+t-1-c+o(1)}\} \cdot L^{c+(1-a)\gamma+o(1)}$

We start by optimizing c. For one code, all terms increase in c except the query cost and the routing cost for just the original list of vectors. Thus, if either of those terms are the greatest cost, then the cost would decrease with a larger code (since we would need to repeat fewer times). If any of the other costs are greatest, the cost would decrease with a *smaller* code. This gives us criteria to check for a binary search to find the optimal c.

As in the asymptotic case, the optimal c is roughly of order a. This means the search cost is lower than the query cost.

The query cost hides a hard-to-compute extra factor: the random product codes are not perfectly random. Recall that a random product code is defined by m lists of B random unit vectors of

[1] Code available at https://github.com/sam-jaques/sieve-memory-estimates.

dimension approximately d/m, so that the code consists of all products of all vectors in these lists. The cost to decode is $O(mB)$, and since we need $B^m = L^c$, the cost is actually $mL^{c/m+o(1)}$. We see that as m decreases, the cost becomes exponential. However, if m increases, the code becomes less random. This may add several bits of difficulty in practice [Duc22].

For a more accurate estimate, we should estimate the non-randomness overhead as in [Duc22], but this is computationally intensive and we leave this for future work. For $m = 2$ it may add little overhead, though it will be more for $m = 8$. It is also true that if there is a multiplicative overhead of $f(m_i)$ for the ith level of recursion, the total overhead will be $f(m_1)f(m_2)f(m_3)$ (and so on). Thus, we attempt to set m as small as possible.

[BDGL16] set m to be poly-logarithmic in d to obtain subexponential decoding. However, [DSv21] notes that on GPUs, it was more efficient to use completely random codes as in [BGJ15], equivalent to $m = 1$. What we notice is that $m = 1$ is not the optimal choice when memory operations are cheaper by constant factors. Instead, we choose the smallest m such that the query cost is at most $1/4$ the routing cost (using $1/4$ as an arbitrary constant). This results in relatively small m: we will find $m = (2, 3, 8)$ for a 3-level recursive sieve. In [DSv21] they note that larger m would be efficient, even with memory bottlenecks, for larger lattice problems, which is roughly what we see here.

With the code size and m optimized, our script then optimizes the filter angle. Applying Lemma 1 with $\lambda = 2 - \gamma$, the search cost either has a local minimum in α or is increasing. We thus assume the entire cost has a local minimum between $\alpha = 0$ and $\alpha = \sqrt{\kappa}$, and we use a divide-and-conquer search to find it.

This approach worked reasonably well for dimension up to 576, but gave unusually large results for higher dimensions. Thus, we also tried the asymptotic approach of Section 3.2, with the same filter angle at each level of recursion, and exhaustively checked filter angles between $\frac{\pi}{3}$ and $\frac{\pi}{2}$. We took the smaller estimate from the two approaches.

Using the same filter angle also allowed us to check higher depths of recursion, since the binary search approach has a runtime exponential in the recursion depth.

4.2 Concrete memory costs

The arguments in Section 2.1 apply asymptotically, but in practice small memory operations are substantially cheaper than other kinds of bit operations.

To represent this, we assign a memory cost of $C \cdot N^{3/2}$ to route N bits of data, where N is the number of bits in memory and C is a fixed constant. That is, we are only considering a two-dimensional memory. This could represent the cost to sort or route on a two-dimensional architecture, or the lower bounds based on wire costs.

It is also reasonable to assume that small blocks of memory can be sorted or routed without consideration of memory costs, and only at larger sizes is it necessary to resort to low-connectivity sorts like a mesh sort. However, this can still be modelled in the same way. That is, suppose memory up to size M_0 can be sorted at cost $O(M_0 \log M_0)$ (for example), after which blocks of memory of size M_0 are sorted on a mesh. Then the cost of this mesh sort, with N total bits of memory, will grow as $C \cdot (\frac{N}{M_0})^{3/2} + O(M_0 \log M_0)$, where C is the memory constant. From here, we can set $C' = C/M_0^{3/2}$ as a new memory constant, and the cost of memory access has the same form for N large enough that the $O(M_0 \log M_0)$ term is irrelevant.

It thus remains to decide on a reasonable value for C. We will proceed here by attempting to balance wire costs to memory and processor costs.

The nVidia GeForce RTX 4090 has 576 tensor cores (hence 16384 cuda cores), 24 GB of memory, runs at 2.235 GHz, and costs USD1600 MSRP [NVI23]. We will take a tensor core as the unit of processor-like object. Each core has 41.7 MB of memory.

Currently, one meter of 100 Gb/s fiber optic cable costs USD550 [Lc23]. Since the time for a signal to propagate 1 meter is negligible compared to 1 second, this means the cable can handle 100 gigabit-meters/second of physical data movement. Matching the cost of cabling and processors

means $2^{28.9}$ bit-meters/processor-second.

Assume this large-scale lattice computer grows with a density comparable to the Frontier supercomputer. Fronter has 8335360 “compute units” in an area of 680 m^2 [Cho22]; this suggests $2^{-13.6}$ m^2 per compute unit. The average distance between two random points on a disk is $\frac{128r}{45\pi}$; asymptotically, the extra distance from the height is negligible. Thus, with P cores they take a radius of $2^{-7.6}\sqrt{P}$ meters, and thus the average distance between them is $2^{-7.8}\sqrt{P}$ meters.

With P processors we have $2^{28.3}P$ bits of memory. The sort must thus move a total distance of $2^{20.5}P^{3/2}$ bit-meters. The cabling gives us $2^{28.9}P$ bit-meters per second, giving $2^{-14.4}\sqrt{P}$ seconds for a sort.

Each GPU can theoretically do 82.6 terraFLOPS ($2^{46.2}$); divided by the 576 cores, that's $2^{37.1}$ FLOPS/core. Multiplying the total number of FLOPS over the machine ($2^{37.1}P$) by the time for a sort ($2^{-14.4}\sqrt{P}$), gives $2^{22.7}P^{3/2}$ operations for a sort. Since the total memory count is $N = 2^{28.3}P$, we have $2^{-19.8}N^{3/2}$ floating point operations per sort. Finally, we can estimate 128 bit operations per FLOP as a rough equivalence for 64-bit floats, and claim a cost of $2^{-12.8}N^{3/2}$ bit operations per sort.

Compared to a RAM model cost of $1.39N \lg N$ bit ops for a sort, the crossover occurs at a somewhat plausible $N =$15 TB.

Since recursive sieves might have lower memory requirements, we take the maximum of $C \cdot N^{3/2}$ and $1.39N \lg N$ as the cost to route N bits of data.

4.3 Results

We evaluate lattices of dimension 375, 586, and 829, as these are the estimated sieve sizes to attack Kyber [ABD+21], and 394, 587, and 818 for Dilithium [BDK+21]. Table 1 summarizes the results. As expected, costs increasse from previous estimates. Memory adds a cost between 14 and 31 bits, with more in larger dimensions. The increase in cost compared to the NIST submissions [ABD+21, BDK+21] is smaller because their security estimates do not include the decoding advancements from [MAT22]. In other words, [MAT22] lowered the claimed security while memory costs raise the claimed security, and the two effects nearly cancel out for Kyber-512.

Table 1: Cost estimates for the sieve from Section 3.2 with memory costs, all at recursion depth 3 except Kyber-1024 and Dilithium-5 at depth 4. The sieve cost is based on Algorithm 1. The primal attack cost is extrapolated from [ABD+21, BDK+21] by assuming that all other aspects of the attack are unchanged, but the sieving step of the core-SVP subroutine changes to the cost in this table. Costs are in log base 2.

Scheme	Sieve Dimension	Sieve Cost	Primal Attack Cost	Change from [ABD+21] [BDK+21]	Change from [AGPS20] update[1]
Kyber-512	375	145.7	158.7	+ 7.2	+14.3
Kyber-768	586	216.5	229.9	+14.8	+22.6
Kyber-1024	829	273.6	310.2	+22.9	+30.6
Dilithium-2	394	152.1	166.5	+ 7.9	+15.0
Dilithium-3	587	216.9	231.6	+14.9	+22.3
Dilithium-5	818	293.4	308.4	+22.0	+30.7

To illustrate the parameters of the sieve, we include a summary of parameters and data for that attack in dimension 375 in Table 2. For comparison, the optimal filter size α in the RAM model is 0.5 and the asymptotically optimal filter size for a two-dimensional architecture with no recursion is $\alpha = 0.366$. By Lemma 2, vectors in the final filter buckets for the 3-level recursive sieve are equivalent to vectors filtered once with $\alpha = 0.439$. That is, the final filtration is nearly as strong as optimal filters in the RAM model.

Table 2: Parameters and results for sieving in dimension 375. Each row represents a subproblem (finding all reducing pairs in a single filter bucket for the problem in the row above). Total cost is the full cost of the sieve (log base 2); list size is the number of vectors in the list to search (log base 2); filter (α) is the strength of the filter around each codeword; num. codes is the number of different codes that must be tried (log base 2); code size is the number of code words (log base 2); m is the number of products that form the random product code; query, memory, and search costs are the costs *per code* (log base 2).

Recursion Level	Total Cost	List Size	Filter (α)	Num. Codes	Code Size	m	Subroutine Costs		
							Query	Memory	Search
1	141.5	98	0.26	8	24	2	114	136	136
2	112.8	75	0.28	11	25	2	93	101	99
3	74.0	46	0.25	9	21	4	60	63	62

Since the constant for memory access is the most tenuous assumption of this analysis, we also ran the estimates for dimension 375 for a constant of 1, shown in Table 3. For comparison, the total memory needed for this dimension is 2^{98} bits, so $N^{3/2} = 2^{147}$ and the sieve cost is about 10-20 bits higher than just the cost to route the data.

Table 3: Costs for sieving of Kyber dimensions under different memory assumptions. "Memory cost" is the coefficient of $N^{3/2}$ for the cost to route data; "recursion depth" is the number of recursive calls (1 is the same as [BDGL16]).

Memory Cost	Recursion Depth	Total Cost (log base 2) Dimension 375
0	1	132.6
	2	134.9
	3	135.8
$2^{-12.8}$	1	153.4
	2	148.4
	3	145.7
1	1	162.7
	2	158.7
	3	158.5

Finally, we give some general results about the behaviour of the recursion. We computed the cost for sieving in dimensions $100, 200, \ldots, 900$, by choosing a constant filter angle for each recursion and fitting the log of the costs with a linear model. This gave a "concrete" cost exponent, shown in Table 4. This simple model gives a cost of $c_1 2^{c_2 n}$ for dimension n for each recursion, with a correlation of $r^2 > 0.99$ between dimension and log cost.

The constant c_1 increased with recursion depth, as expected. Thus we also computed the minimum dimension n where we would expect improvement from a higher recursion depth. The fact that it predicts 4 levels of recursion to be optimal for dimension 575, while our experiments show that 3 levels is better, is likely just due to approximation error from suboptimal parameterizations.

Table 4: Cost exponents for two-dimensional memory based on asymptotic analysis (i.e., the same as Figure 1) or a linear fit to the concrete estimates. The minimum worthwhile dimension is the minimum dimension at which the linear model predicts that the given recursion depth will give better performance.

	Recursion Depth				
	1	2	3	4	5
Asymptotic Exponent	0.349	0.334	0.328	0.324	0.322
Concrete Exponent	0.357	0.340	0.331	0.325	0.322
Minimum worthwhile dimension	–	56	230	339	810

Acknowledgements.

We would like to thank Léo Ducas, Thijs Laarhoven, Eamonn Postlethwaite, John Schanck, and Dan Shepherd for both answering and asking the right questions about this work.

References

[ABD+21] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, , and Damien Stehlé. CRYSTALS-Kyber (version 3.02) – submission to round 3 of the NIST post-quantum project. 2021. .

[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, *Advances in Cryptology – EUROCRYPT 2019, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 717–746, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-17656-3_25.

[AGPS20] Martin R. Albrecht, Vlad Gheorghiu, Eamonn W. Postlethwaite, and John M. Schanck. Estimating quantum speedups for lattice sieves. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part II*, volume 12492 of *Lecture Notes in Computer Science*, pages 583–613, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-64834-3_20.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015. URL: https://doi.org/10.1515/jmc-2015-0016 [cited 2023-12-20], doi:doi:10.1515/jmc-2015-0016.

[BBG+13] Robert Beals, Stephen Brierley, Oliver Gray, Aram W. Harrow, Samuel Kutin, Noah Linden, Dan Shepherd, and Mark Stather. Efficient distributed quantum computing. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 469(2153):20120686, May 2013. doi:10.1098/rspa.2012.0686.

[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 Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 10–24, Arlington, VA, USA, January 10–12, 2016. ACM-SIAM. doi:10.1137/1.9781611974331.ch2.

[BDK+21] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium – submission to round 3 of the NIST post-quantum project. 2021. .

[Ber23] Daniel J. Bernstein. Asymptotics of hybrid primal lattice attacks. Cryptology ePrint Archive, Report 2023/1892, 2023. .

[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. https://eprint.iacr.org/2015/522.

[BK81] R. P. Brent and H. T. Kung. The area-time complexity of binary multiplication. *J. ACM*, 28(3):521–534, jul 1981. doi:10.1145/322261.322269.

[BL12] Daniel J. Bernstein and Tanja Lange. Non-uniform cracks in the concrete: the power of free precomputation. Cryptology ePrint Archive, Report 2012/318, 2012. https://eprint.iacr.org/2012/318.

[Cho22] Charles Q. Choi. The beating heart of the world's first exascale supercomputer. IEEE Spectrum, 2022. .

[CL23] André Chailloux and Johanna Loyer. Classical and quantum 3 and 4-sieves to solve svp with low memory. In *Post-Quantum Cryptography: 14th International Workshop, PQCrypto 2023, College Park, MD, USA, August 16–18, 2023, Proceedings*, page 225–255, Berlin, Heidelberg, 2023. Springer-Verlag. doi:10.1007/978-3-031-40003-2_9.

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20, Seoul, South Korea, December 4–8, 2011. Springer, Heidelberg, Germany. doi:10.1007/978-3-642-25385-0_1.

[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, *Advances in Cryptology – EUROCRYPT 2021, Part II*, volume 12697 of *Lecture Notes in Computer Science*, pages 249–279, Zagreb, Croatia, October 17–21, 2021. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-77886-6_9.

[Duc18] Léo Ducas. Shortest vector from lattice sieving: a few dimensions for free. Presentation at Eurocrypt, 2018. URL: https://eurocrypt.iacr.org/2018/Slides/Monday/TrackB/01-01.pdf.

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

[FHK+20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over NTRU (specification v1.2) – submission to round 3 of the nist post-quantum project. 2020. .

[HK17] Gottfried Herold and Elena Kirshanova. Improved algorithms for the approximate k-list problem in euclidean norm. In Serge Fehr, editor, *PKC 2017: 20th International Conference on Theory and Practice of Public Key Cryptography,*

Part I, volume 10174 of *Lecture Notes in Computer Science*, pages 16–40, Amsterdam, The Netherlands, March 28–31, 2017. Springer, Heidelberg, Germany. doi:10.1007/978-3-662-54365-8_2.

[HKL18] Gottfried Herold, Elena Kirshanova, and Thijs Laarhoven. Speed-ups and time-memory trade-offs for tuple lattice sieving. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018: 21st International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 10769 of *Lecture Notes in Computer Science*, pages 407–436, Rio de Janeiro, Brazil, March 25–29, 2018. Springer, Heidelberg, Germany. doi:10.1007/978-3-319-76578-5_14.

[KMPM19] Elena Kirshanova, Erik Mårtensson, Eamonn W. Postlethwaite, and Subhayan Roy Moulik. Quantum algorithms for the approximate k-list problem and their application to lattice sieving. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019, Part I*, volume 11921 of *Lecture Notes in Computer Science*, pages 521–551, Kobe, Japan, December 8–12, 2019. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-34578-5_19.

[Kun87] Manfred Kunde. Optimal sorting on multi-dimensionally mesh-connected computers. In Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87*, pages 408–419, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[Laa15] Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 3–22, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany. doi:10.1007/978-3-662-47989-6_1.

[Lc23] L-com. Active optical cable QSFP28 100Gbps, 1 meter, Cisco compatible. L-com product page, 2023. .

[MAT22] MATZOV. Report on the security of LWE: Improved dual lattice attack, 2022. .

[Nat23a] National Institute of Standards and Technologies. FAQ on Kyber512. 2023. .

[Nat23b] National Institute of Standards and Technology. Module-lattice-based digital signature standard. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 203 (draft), U.S. Department of Commerce, Washington, D.C., 2023. doi:10.6028/NIST.FIPS.203.ipd.

[Nat23c] National Institute of Standards and Technology. Module-lattice-based digital signature standard. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 204 (draft), U.S. Department of Commerce, Washington, D.C., 2023. doi:10.6028/NIST.FIPS.204.ipd.

[NV08] Phong Q. Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2), January 2008. URL: http://dx.doi.org/10.1515/JMC.2008.009, doi:10.1515/jmc.2008.009.

[NVI23] NVIDIA Corporation. GeForce RTX 4090. NVIDIA product page, 2023. .

[Sch23] John M. Schanck. When sorting your data costs more than cracking AES-128, 2023. .

[SS86] Claus-Peter Schnorr and Adi Shamir. An optimal sorting algorithm for mesh connected computers. In *18th Annual ACM Symposium on Theory of Computing*, pages 255–263, Berkeley, CA, USA, May 28–30, 1986. ACM Press. doi:10.1145/12130.12156.

[Tho80] C. D. Thompson. *A Complexity Theory for VLSI*. Phd thesis, Carnegie Mellon University, 6 1980. Available at . doi:10.1184/R1/6714269.v1.

[vW94] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with application to hash functions and discrete logarithms. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, and Ravi S. Sandhu, editors, *ACM CCS 94: 2nd Conference on Computer and Communications Security*, pages 210–218, Fairfax, Virginia, USA, November 2–4, 1994. ACM Press. doi:10.1145/191177.191231.

[Wie04] Michael J. Wiener. The full cost of cryptanalytic attacks. *Journal of Cryptology*, 17(2):105–124, March 2004. doi:10.1007/s00145-003-0213-5.

A False Negatives in Subproblems

Here we explain how we compute the fraction of false negatives from the subproblem in Algorithm 1. In our code, we evaluate the integral expression numerically.

Lemma 3. *If* $\mathbf{v}$, $\mathbf{w}$, *and* $\mathbf{c}$ *are uniformly random vectors on the surface of a* d*-dimensional sphere, and* $\mathbf{v}'$ *and* $\mathbf{w}'$ *are* $d-1$*-dimensional unit vectors proportional to* $\mathbf{v} - proj_{\mathbf{c}}(\mathbf{v})$ *and* $\mathbf{w} - proj_{\mathbf{c}}(\mathbf{w})$ *respectively, then for any* $\kappa \in [0, \frac{1}{2}]$ *and* $\alpha \in [0, \sqrt{\kappa}]$,

$$Pr\left[\langle \mathbf{v}', \mathbf{w}'\rangle \geq \frac{\kappa - \alpha^2}{1-\alpha^2} \middle| \begin{matrix} \langle \mathbf{v}, \mathbf{w}\rangle \geq \kappa \\ \langle \mathbf{v}, \mathbf{c}\rangle \geq \alpha \\ \langle \mathbf{w}, \mathbf{c}\rangle \geq \alpha \end{matrix}\right] \tag{38}$$

is equal to

$$\frac{\int_0^\alpha \int_0^\alpha F_1(d, \alpha_v, \alpha_w, \kappa, \alpha) p(\alpha_v) p(\alpha_w) d\alpha_v d\alpha_w}{\int_0^\alpha \int_0^\alpha F_2(d, \alpha_v, \alpha_w, \kappa) p(\alpha_v) p(\alpha_w) d\alpha_v d\alpha_w} \tag{39}$$

where $p(x)$ *is the probability density function for the probability that a random unit vector has inner product exactly* x *with a fixed vector,*

$$F_1(d, \alpha_v, \alpha_w, \kappa, \alpha) = \begin{cases} 0 & , 1 \leq \kappa_0 \\ C_d\left(\max\left\{\kappa_0, \frac{\kappa-\alpha^2}{1-\alpha^2}\right\}\right) & , \kappa_0 < 1 \end{cases} \tag{40}$$

for $C_d(x)$ *as the probability that a random unit vector has inner product at least* x *with another fixed vector,*

$$\kappa_0 = \frac{\kappa - \alpha_v \alpha_w}{\sqrt{(1-\alpha_v^2)(1-\alpha_w^2)}}, \tag{41}$$

and

$$F_2(d, \alpha_v, \alpha_w, \kappa, \alpha) = \begin{cases} 0 & , \kappa_0 \geq 1 \\ C_d(\kappa_0) & , 0 \leq \kappa_0 < 1 \\ \frac{1}{2} + C_d(-\kappa_0) & , -1 < \kappa_0 < 0 \\ 1 & , \kappa_0 \leq -1 \end{cases} \tag{42}$$

Proof. We can Bayes' theorem to obtain

$$\Pr\left[\langle \mathbf{v}', \mathbf{w}'\rangle \geq \frac{\kappa - \alpha^2}{1-\alpha^2} \middle| \begin{matrix} \langle \mathbf{v}, \mathbf{w}\rangle \geq \kappa \\ \langle \mathbf{v}, \mathbf{c}\rangle \geq \alpha \\ \langle \mathbf{w}, \mathbf{c}\rangle \geq \alpha \end{matrix}\right] = \frac{\Pr\left[\begin{matrix} \langle \mathbf{v}, \mathbf{w}\rangle \geq \kappa, & \langle \mathbf{v}', \mathbf{w}'\rangle \geq \frac{\kappa-\alpha^2}{1-\alpha^2} \\ \langle \mathbf{v}, \mathbf{c}\rangle \geq \alpha, & \langle \mathbf{w}, \mathbf{c}\rangle \geq \alpha \end{matrix}\right]}{\Pr\left[\begin{matrix} \langle \mathbf{v}, \mathbf{w}\rangle \geq \kappa, & \\ \langle \mathbf{v}, \mathbf{c}\rangle \geq \alpha, & \langle \mathbf{w}, \mathbf{c}\rangle \geq \alpha \end{matrix}\right]} \tag{43}$$

We first consider the numerator. We can express this as

$$\int_0^\alpha \int_0^\alpha \int_{\frac{\kappa-\alpha^2}{1-\alpha^2}}^1 \Pr\left[\langle \mathbf{v},\mathbf{w}\rangle \geq \kappa \middle| \begin{matrix}\langle \mathbf{v}',\mathbf{w}'\rangle = \kappa' \\ \langle \mathbf{v},\mathbf{c}\rangle = \alpha_v \\ \langle \mathbf{w},\mathbf{c}\rangle = \alpha_w\end{matrix}\right] p(\kappa')p(\alpha_v)p(\alpha_w)d\kappa' d\alpha_v d\alpha_w \tag{44}$$

The probability density splits into a product like this because the vectors $\mathbf{v}'$ and $\mathbf{w}'$ are independent of $\langle \mathbf{v},\mathbf{c}\rangle$ and $\langle \mathbf{w},\mathbf{c}\rangle$, since $\mathbf{v}'$ and $\mathbf{w}'$ are normalized.

For the conditional probability, we see that since $\mathbf{v} = \alpha_v \mathbf{c} + \sqrt{1-\alpha_v^2}\mathbf{v}'$ (similarly for $\mathbf{w}$), we have that

$$\langle \mathbf{v},\mathbf{w}\rangle = \alpha_v\alpha_w + \sqrt{(1-\alpha_v^2)(1-\alpha_w^2)}\kappa'. \tag{45}$$

That is, given κ', α_v, and α_w, the inner product $\langle \mathbf{v},\mathbf{w}\rangle$ is fixed. We can thus conclude that this is at least κ if and only if

$$\kappa' \geq \frac{\kappa - \alpha_v\alpha_w}{\sqrt{(1-\alpha_v^2)(1-\alpha_w^2)}} =: \kappa_0. \tag{46}$$

That is, the conditional probability is either 0 or 1 depending on κ', α_v, and α_w. The integral over κ' is 0 if $\kappa_0 \geq 1$. If $0 \leq \kappa_0 < \frac{\kappa-\alpha^2}{1-\alpha^2}$, then we are simply integrating a spherical wedge. That is, we will have

$$\int_{\frac{\kappa-\alpha^2}{1-\alpha^2}}^1 \Pr\left[\langle \mathbf{v},\mathbf{w}\rangle \geq \kappa \middle| \begin{matrix}\langle \mathbf{v}',\mathbf{w}'\rangle = \kappa' \\ \langle \mathbf{v},\mathbf{c}\rangle = \alpha_v \\ \langle \mathbf{w},\mathbf{c}\rangle = \alpha_w\end{matrix}\right] p(\kappa')d\kappa' = C_d\left(\max\left\{\kappa_0, \frac{\kappa-\alpha^2}{1-\alpha^2}\right\}\right). \tag{47}$$

We can similarly express the denominator as

$$\int_0^\alpha \int_0^\alpha \int_{-1}^1 \Pr\left[\langle \mathbf{v},\mathbf{w}\rangle \geq \kappa \middle| \begin{matrix}\langle \mathbf{v}',\mathbf{w}'\rangle = \kappa' \\ \langle \mathbf{v},\mathbf{c}\rangle = \alpha_v \\ \langle \mathbf{w},\mathbf{c}\rangle = \alpha_w\end{matrix}\right] p\begin{pmatrix}\langle \mathbf{v}',\mathbf{w}'\rangle = \kappa' \\ \langle \mathbf{v},\mathbf{c}\rangle = \alpha_v \\ \langle \mathbf{w},\mathbf{c}\rangle = \alpha_w\end{pmatrix} d\kappa' d\alpha_v d\alpha_w \tag{48}$$

This is identical except that κ' can extend to -1. A similar reasoning applies, giving us the formula for F_2. □