Yours Truly

THis is a Book

Contents

List of Figures

List of Tables

I

This is a Part

CHAPTER 1

Optimized Voronoi-based Algorithms for Parallel Shortest Vector Computation

Artur Mariano

INESC TEC & Universidade do Minho

Filipe Cabeleira

University of Coimbra & Instituto de Telecomunicações

Luís Paulo Santos

INESC TEC & Universidade do Minho

Gabriel Falcão

University of Coimbra & Instituto de Telecomunicações

CONTENTS

THIS CHAPTER addresses Voronoi cell-based algorithms, specifically the "Relevant Vectors" algorithm, used to solve the Shortest Vector Problem, a fundamental challenge in lattice-based cryptanalysis. Several optimizations are proposed to reduce the execution time of the original algorithm. It is also shown that the algorithm is highly suited for parallel execution on both CPUs and GPUs. The proposed optimizations are based on pruning, i.e., avoiding computations that will not, with high probability, improve the solution. The pruning criteria is related to the target vectors norm relative to the current best solution vector norm. When pruning is performed without pre-processing, speedups up to 69× are observed compared to the original algorithm. If a pre-process sorting step is performed, which requires storing the norm ordered target vectors and therefore significantly more memory, this speedup increases to 77×. On the parallel processing side, the multi-core version of the optimized algorithm exhibits linear scalability on a CPU with up to 28 threads and keeps scaling, albeit at a lower rate, with Simultaneous Multi-Threading with up to 56 threads. The lack of support for efficient global synchronization among threads in GPUs does not allow for a scalable implementation of the pruning optimization using these devices. Nevertheless, a parallel GPU version of the non-optimized algorithm is demonstrated to be competitive with the parallel non optimized CPU version, although the latter outperforms the former when using 56 threads. It is argued that the GPU version would outperform the CPU for higher lattice dimensions, although this statement cannot be experimentally verified due to the limited memory available on current GPU boards.

1.1 INTRODUCTION

Since the mid-nineties, the cryptography community has been studying alternatives to classical cryptosystems such as RSA and El-Gamal, as these were shown to be vulnerable in the presence of quantum computers. These cryptosystems were based on premises that the factorization of large numbers exhibits an exponential time complexity. Shor's algorithm [26, 25, 3] has shown that this class of problems can be solved in polynomial time on a quantum machine. Therefore, eavesdroppers with access to a sufficiently large quantum machine can hack the systems and access communications.

Many cryptosystems have been proposed since the rise of this so-called post-quantum era. Most of these cryptosystems are designed under the premise (or belief, in most cases) that even if adversaries had access to large-scale quantum computers, they cannot be broken. Lattice-based cryptosystems are a very prominent type of post-quantum cryptosystems. They support advanced cryptographic primitives such as Fully Homomorphic Encryption[1], they are relatively efficient in practice, easy to implement and, of course, believed to be safe against quantum adversaries [18, 3].

Cryptosystems base their security on hard math problems, which are typically easy to solve for the users of the system but hard to solve for external entities. The underlying idea is that the fundamental problems underpinning the security of lattice-based cryptosystems, such as the Shortest Vector Problem (SVP), the Closest Vector Problem (CVP) and derivatives of these cannot be solved (exponentially) faster with quantum computers, when compared to conventional computers. Due to the connection between the problems and the security of the corresponding cryptosystems, the algorithms that solve these problems are commonly referred to as attacks.

Lattices are discrete subgroups of the n-dimensional Euclidean space $\mathbb{R}^n$, with a strong periodicity property[2]. A lattice $\mathcal{L}$ generated by a basis $\mathbf{B}$, a set of linearly independent vectors $\mathbf{b}_1$, ..., $\mathbf{b}_m$ in $\mathbb{R}^n$, is denoted by:

$$\mathcal{L}(\mathbf{B}) = \left\{ \mathbf{x} \in \mathbb{R}^n : \mathbf{x} = \sum_{i=1}^{m} \mathbf{u}_i \mathbf{b}_i,\ \mathbf{u} \in \mathbb{Z}^m \right\}, \tag{1.1}$$

where $m \leq n$ is the *rank* of the lattice. When $m = n$, the lattice is said to be of *full rank*. When n is at least 2, each lattice has infinitely many different bases.

Note that, although there are non-integer lattices, lattice-based cryptography commonly uses integer lattices in practice: solving lattice problems on

[1] A cryptosystem that supports Fully Homomorphic Encryption can implement any operation on encrypted data, without decrypting data, which is particularly useful when e.g. outsourcing sensitive computations on private data to a cloud server. The reader is referred to the survey [17] for a practical perspective of Fully Homomorphic Encryption.

[2] We refer the reader to papers [23, 21] to learn more about lattices, especially in the context of lattice-based cryptography.

integer lattices is still hard, and integer lattices are easier to handle computationally (e.g. there are no precision/numerical problems). As an example, Figure 1.1 shows a lattice in $\mathbb{R}^2$, where the basis is $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2\}$. The vector $\mathbf{b}_3$ shown in the picture is a linear combination of the basis vectors. This linear combination also shows that $\mathbf{b}_1$ can be made shorter (in terms of Euclidean norm, which is the default meaning of shortness in the context of this book chapter) at the cost of $\mathbf{b}_2$, given that $\mathbf{b}_3$ is smaller than $\mathbf{b}_1$. This process, of making lattice vectors (bases) shorter by adding/subtracting other lattice vectors, is often referred to as vector (basis) reduction and is widely used in various lattice algorithms.

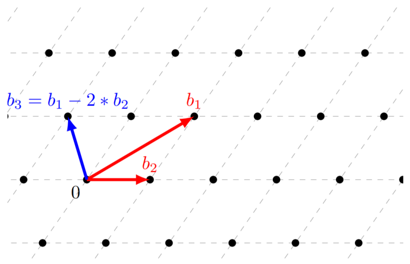


FIGURE 1.1 Example of a lattice in R^2 and its basis ($\mathbf{b}_1$,$\mathbf{b}_2$) in red.

Given that the security of lattice-based cryptosystems is based on problems like the SVP, CVP and approximated versions of these, they have been widely studied over the last decade. In particular, many parallel, highly efficient versions of algorithms that solve these problems have been devised and put to the test, to assess their real hardness. Also, cryptosystems have certain parameters, such as the key size, that have to be determined based on the algorithms' practical potential/performance. Setting these parameters too high would lead to inefficient/slow cryptosystems, but setting them too low leads to insecure systems. As such, a "sweet spot" has to be found in this trade-off, so that systems are simultaneously efficient and secure. This can only happen when the best attacks are implemented on the highest-end computer architectures. This is also part of the work that we conduct and analyze in this very same manuscript.

The SVP has been extensively studied during the last decades and two main families of SVP-solvers have emerged. As this research progressed, they evolved to become the standard algorithms in this context. The first family is the set of sieving algorithms, which repeatedly sieve a list of vectors until the shortest vector is very likely to be arrived at. The second most relevant family of algorithms is the family of enumeration algorithms. These enumerate all the possible vectors within a given search radius around the origin, and therefore the shortest vector of the lattice is the shortest in that set of enumerated vectors. Other than these two SVP algorithm families, many are often mentioned and studied, but to a much smaller extent. Some of those families include random-sampling and Voronoi cell-based SVP solvers. The span of research around SVP algorithms is quite extensive and impractical to cover in this manuscript. To better grasp the history and evolution of this field, we refer the reader to [15, 16, 17, 18, 27].

In this text, we select one type of attack – SVP algorithms based on the Voronoi cell of a lattice – that has been often mentioned in the literature [1, 19, 4], but rarely studied or published about. In fact, it is often said that this algorithm becomes impractical (mainly due to memory issues) somewhere in dimension 14-20, but this support was never evidently backed up by tests. Plus, other classes of SVP-solvers, such as enumeration and sieving, have been subject of intense and ongoing investigation and optimization through the past decade (e.g. [7, 13, 2, 12]). But Voronoi cell-based SVP-solvers, to the best of our knowledge, have not been optimized since their first publication [1], back in 2002. Voronoi cell-based algorithms are, however, of interest to study. First, they are asymptotically very appealing, which means that if the lattice dimension is high enough, they should be competitive with other classes of SVP-solvers. Secondly, there is a big room for practical improvement in Voronoi cell-based algorithms, which could perhaps lead to tractable implementations for high-dimensional lattices. In [5] we presented parallel versions, including an heterogeneous CPU+GPU implementation, of the original algorithm. In this work, we take a first step towards optimizing the original algorithm, therefore reducing the associated computational workload, and further propose parallel implementations of the optimized algorithm.

Contributions. In this manuscript, we propose various improvements for Voronoi cell-based algorithms, in the context of the SVP, and we show that the improved algorithm is still suitable for parallel execution.

We have been able to show that this algorithm can be optimized by using several norm-based optimizations. In particular, we show that computations that are, with high probability, irrelevant in the context of the SVP can be pruned. By considering previous states of the algorithm, namely the norm of the shortest known solution vector, the algorithm's workload can be dramatically reduced. Further workload reduction can be achieved if the target vectors are sorted according to their Euclidean norm prior to the evaluation of the solution vector.

We present a parallel version of the optimized Voronoi cell-based algorithm for the SVP. It is optimized to achieve the shortest vector faster and performs very well on architectures with multiple cores. This version was able to attain linear speedups on CPUs, in our experiments, compared to the baseline, original Voronoi-cell-based SVP-solver. Due to the lack of support for efficient global synchronization among threads on GPUs we can not present a scalable implementation of the optimized algorithm in these devices. Similarly to [5] we show that the non optimized algorithm is highly suited for these architectures and competitive with the non optimized multi core CPU version.

The meaning of our work is two-fold: first, we show that Voronoi cell-based algorithms can be made more practical than previously reported. Such practicality is achieved by introducing the above mentioned norm-based optimizations, which are possible given that the goal is to solve the SVP. This should help to shed further light on this class of algorithms. Secondly, we show that the optimized algorithm is suited for parallelization, which makes it appealing for parameter selection in lattice-based cryptosystems.

Roadmap. The remainder of this chapter is organized as follows. Section 1.2 presents Voronoi-cell-based algorithms including the algorithm exploited in the context of this text. Section 1.3 introduces the experimental setup. Section 1.4 presents an analysis performed on the algorithm, which serves as the motivation for our optimizations, presented in Section 1.5. Section 1.6 describes our parallel implementations, both on CPUs and GPUs, as well as their performance results. Section 1.8 concludes the chapter and points out future lines of work.

1.2 SVP-SOLVERS BASED ON VORONOI CELLS

In this section, we briefly explain a Voronoi-cell-based algorithm by [19], which can be used to solve the SVP, and the algorithm we used in this work, called "Relevant Vectors", presented by [1].

1.2.1 Voronoi-cell-based algorithm by Micciancio et al.

This algorithm, presented in [19], describes a deterministic approach to solve the SVP (and other related problems), using the Voronoi cell $\mathcal{V}$ of a lattice as a way to arrive at the shortest vector of the lattice.

The algorithm works by rank reduction, i.e., the solution in a given dimension k requires the computation of some procedures in dimension $k-1$. Micciancio et al. show that the computation of the n-dimensional Voronoi cell of the lattice can be done by a series of CVP calls for the n-dimension lattice, $\mathcal{V}(\mathcal{L}_n) = k \cdot CVP(\mathcal{L}_n)$, for a given number k of calls (for more details we refer the reader to [19]). Furthermore, they also show that the CVP solution of an n-dimensional lattice can be obtained by a series of CVP computations on the associated $(n-1)$-dimensional lattice, i.e. $CVP(\mathcal{L}_n) = k \cdot CVP(\mathcal{L}_{n-1})$.

Therefore, the Voronoi cell of a lattice in dimension n can be computed iteratively, starting on dimension 1 and working up towards dimension n. The solution of the SVP is the shortest nonzero vector $\mathbf{s} \in \mathcal{L}$, which, within the Voronoi cell context, is given by its shortest vector. More precisely, the solution for the SVP in this case is given by the double of the shortest vector of the Voronoi cell, as the frontier of the latter is, by definition, the midpoint between 0 and the vectors that are closest to 0.

As for the implementation of this algorithm, we start with reducing the basis and initializing the list of Voronoi relevant vectors with the first vector of the reduced basis. With this lower dimension list, we iterate upwards to dimension n, by generating a list of the so-called target vectors. For each of these, a CVP function is computed, so that we end up with the Voronoi cell vectors, which is refined so that it only contains the relevant vectors. The relevant vectors, which form the minimum set of vectors that describe the Voronoi cell of a lattice, are shown in Figure 1.2, for a given 2-dimensional lattice.

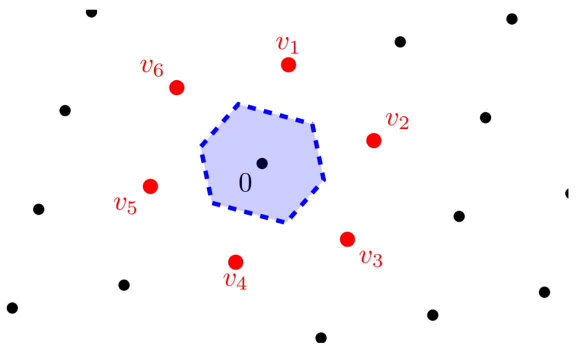


FIGURE 1.2 Example of a Voronoi cell in R^2 (blue), and its relevant vectors (red).

The algorithm's time asymptotic complexity is $\mathcal{O}(2^{2n})$ while its space complexity is $\mathcal{O}(2^n)$, n being the lattice dimension. To fully comprehend this algorithm and its nuances, we refer the reader to [19], as we do not describe it with full detail given that this algorithm is not used in this work.

1.2.2 Relevant vectors by Agrell et al.

The algorithm used (a Voronoi cell-based algorithm called "Relevant vectors"), was presented in [1]. That paper also described several algorithms to determine the solution to the SVP, CVP, and other related problems, and is shown in Algorithm 1.

```
Function AllClosestPoints
  Input: Matrix $\mathbf{M}$, matrix $\mathbf{H}$, matrix $\mathbf{Q}$, vector $\mathbf{s}$
  Output: List of vectors $\mathbf{X}$
  Compute $\mathbf{x} = \mathbf{s}\mathbf{Q}^T$;
  $\mathbf{U}$ = Decode($\mathbf{H}$, $\mathbf{x}$);
  Compute $\gamma$ as the lowest value $||\mathbf{uM} - \mathbf{s}||$ for all $\mathbf{u} \in \mathbf{U}$;
  Compute $\mathbf{X}$ as all $\{\mathbf{uM} : \mathbf{u} \in \mathbf{U}, ||\mathbf{uM} - \mathbf{s}|| = \gamma\}$
  return $\mathbf{X}$
```

Algorithmically, "Relevant vectors" can be described by four, distinct steps. First, it starts by generating the needed target vectors, that are later on used by a CVP-solver, in order to compute the Voronoi relevant vectors of the

Algorithm 1: RelevantVectors

```
Input: Basis matrix $\mathbf{B}$
Output: Relevant Vectors $\mathbf{N}$
$\mathbf{M} = \text{Reduce}(\mathbf{B})$;  /* for example, using the LLL algorithm */
$[\mathbf{Q}, \mathbf{R}] = $ QR decomposition of $\mathbf{M}$;
$\mathbf{G} = \mathbf{R}^T$;
$\mathbf{H} = \mathbf{G}^{-1}$;
$\mathbf{N} = \varnothing$;
forall vectors $\mathbf{s} \in \mathcal{M}$ do
    $\mathbf{X} = \text{AllClosestPoints}(\mathbf{M}, \mathbf{H}, \mathbf{Q}, \mathbf{s})$;
    if $|\mathbf{X}| = 2$ then
        $\mathbf{N} = \mathbf{N} \bigcup \{2\mathbf{x} - 2\mathbf{s} : \mathbf{x} \in \mathbf{X}\}$;
return N
```

lattice. Second, the coordinate system of the data that feeds the CVP-solver is modified (i.e. the lattice basis and the target vectors). Details on the rationale behind these steps can be found in [1]. The third step is then to run an enumeration CVP-solver on each of the generated target vectors, a process we refer to as "decoding". This solver computes a set of vectors, which are then converted to the original coordinate system, thus resulting in the final list of candidate Voronoi relevant vectors. From these, only the valid vectors (in fact Voronoi relevant vectors) are kept.

In terms of implementation, the CVP-solver that "decodes" target vectors is based on the Schnorr-Euchner method [24], which is an enumeration method to compute the SVP and the CVP. This is called "enumeration" because the algorithm enumerates all the possible solutions within a given radius. For more detail on this algorithm, we refer the reader to papers on enumeration algorithms [24, 7, 1].

To increase performance, it is desirable to reduce the input lattice basis. This can be achieved, e.g., using the LLL algorithm (cf. [14]). Additionally, this enumeration-based CVP-solver function requires the input lattice basis to be in a lower triangular form. When this is not the case, we must transform the basis to this form, while also transforming the input (target) vector(s) as well. This can be done with e.g. a QR decomposition, in the form $\boldsymbol{M} = \boldsymbol{QR}$, where $\boldsymbol{R}$ is a $n \times n$ upper triangular matrix and $\boldsymbol{Q}$ is a $m \times n$ orthonormal matrix. As we deal with full-rank lattices, effectively we end up with $\boldsymbol{R}_{n\times n}$ and $\boldsymbol{Q}_{n\times n}$, as $m = n$ and n is the lattice dimension. We call the QR method on the lattice basis ($\mathbf{M}$ in the decomposition), yielding $\mathbf{R}$, which we must transpose i.e. $\mathbf{R}^T$, to obtain the desired lower-triangular matrix[3].

The other resulting matrix ($\mathbf{Q}$), is used to transform the target vector into the coordinate system of the lattice basis when in the lower-triangular form.

[3]The diagonal elements of this matrix must be positive.

Note that when the QR decomposition is used, it is also needed to transform the output of the decode function back into the original form (i.e. the original coordinate system).

Once the basis is in the desired format, we generate the list $\mathcal{M}$ that contains each of the $\mathbf{s}_i$ target vectors, $i = 1, ..., (2^n - 1)$ (in practice steps 1 and 2 of the mathematical description above can be done together), as shown in Equation 1.2, iteratively.

$$\mathcal{M}(\mathbf{M}) \overset{\text{def}}{=} \Big\{\mathbf{s} = \mathbf{zM} : \mathbf{z} \in \{0, 1/2\}^n - \{\mathbf{0}\}\Big\} \tag{1.2}$$

Afterward, the CVP-solver is executed on these inputs, yielding a list of vectors $\mathbf{U}$, that are processed according to Equation 1.3, resulting in the list of vectors $\mathbf{X}$.

$$\begin{aligned} \gamma &= \min\Big\{||\mathbf{uM} - \mathbf{s}|| \text{ for all } \mathbf{u} \in \mathbf{U}\Big\} \\ \mathbf{X} &= \Big\{\mathbf{uM} : \mathbf{u} \in \mathbf{U}, ||\mathbf{uM} - \mathbf{s}|| = \gamma\Big\} \end{aligned} \tag{1.3}$$

The computation of list $\mathbf{X}$ does not always result in a valid output. This only happens when the list contains 2 vectors and 2 vectors only (they are symmetric to each other, thus having the same norm), which are added to the list of Voronoi relevant vectors $\mathbf{N}$. Similarly to the algorithm in 1.2.1, the solution of the SVP is given by the shortest of the Voronoi relevant vectors.

1.3 EXPERIMENTAL SETUP

Table 1.1 presents the details of the CPU based computing system used to assess the proposed parallel algorithm. The clock frequency in parenthesis shown in the table pertains to the maximum frequency of the CPU, which is achieved using the Turbo Boost Technology. L1 cache values are split between instruction cache (i) and data cache (d). System A runs CentOS x86_64 with kernel version 2.6. The code has been compiled with g++ 7.2.0 with the `-O3` optimization flag, as it delivered the best throughput performance.

The tests conducted in a GPU used system B, specified in Table 1.2, which runs Ubuntu 16.04 x86_64 with kernel version 4.13. CUDA code was compiled with NVIDIA CUDA Compiler 9.1 using the `-O3` optimization flag and the `-arch=sm_61 -lcudadevrt -rdc=true` flags. The GPUs have compute capability 6.1 and allow for dynamic parallelism (a kernel launch within another kernel). The CPU code on this machine was compiled with g++ 5.4.0.

The lattice bases used in all tests were generated with the SVP-Challenge[4] generator software, compiled using NTL version 9.3. The lattice bases generated using this tool are random (Goldstein-Mayer) lattices, which have no specific characteristic to be exploited [9]. Additionally, these lattice bases are reduced using the LLL algorithm before the main loop of the algorithm (c.f. Algorithm 1). Unless specified otherwise, the tests presented in this work are

[4]https://www.latticechallenge.org/svp-challenge/

TABLE 1.1 CPU based computing system. SMT stands for simultaneous multi-threading and HT stands for hyper threading.

System	**A**
Sockets	2
CPU	Intel Xeon E5-2660v4
Clock frequency	2.0 GHz (3.2 GHz)
Cores per socket	14
SMT	Yes (w/ HT, 28 threads)
L1 Cache	448 kB i + 448 kB d
L2 Cache	3.5 MB
L3 Cache	35 MB
RAM	128 GB

TABLE 1.2 Machine for GPU tests used in this work. SMT stands for simultaneous multi-threading and HT stands for hyperthreading.

System	**B**
CPU	Intel Core i3 6100
Clock frequency	3.70 GHz
Cores	2
SMT	Yes (w/ HT, 4 threads)
L1 Cache	32 kB i + 32 kB d
L2 Cache	256 kB
L3 Cache	3 MB
RAM	8 GB
GPU	NVIDIA GeForce 1060 GTX
GPU Clock rate	1759 MHz
GPU RAM	6 GB

conducted with seeds 0 through 999. They represent a total of 1000 bases, up until dimension 10, 100 bases for dimensions 11-15 and 10 bases for dimension 16 or higher. We present the arithmetic average of all tested bases (with different seeds). We have turned off Turbo Boost, in order to have a better sense of the scalability of the algorithm, and the results obtained were fairly consistent. Executing the algorithm for more seeds would not impact the average execution time. In this context, we refer to “a test” as the execution of the program across all the seeds. Also, our tests were only conducted in these (small) dimensions, as the memory requirements of the algorithm grow exponentially. While individually they run relatively fast, performing 1000 runs per dimension would be impractical.

1.4 ALGORITHM ANALYSIS

From Equation 1.2 (c.f. Section 1.2.2), we see that the computation of the Voronoi cell of a lattice involves the execution of the *AllClosestPoints* function, for each of the $(2^n - 1)$ vectors that make up the set $\mathcal{M}$. However, in practice, most of these calculations are unnecessary if our purpose is to find the solution of the SVP. As such, we describe a series of tests that we conducted, which lay the foundation of the proposed optimizations. For these tests, we used Machine A.

1.4.1 Correlation between the norm of target vectors and solution vectors

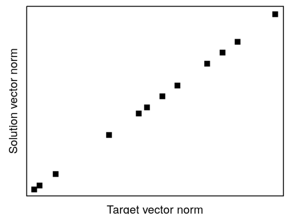
We posed the hypothesis of a possible correlation between the norm of the target vectors and the norm of their respective solution vector, a test we started out with. The motivation to test out this possible correlation stems from the fact that, intuitively, the computation of a target vector with smaller Euclidean norm (i.e. shorter) would also result in a shorter solution vector. If this correlation held, then we could potentially exclude several target vectors, by only decoding a few, small subset of these vectors, given that our purpose is to arrive at the shortest vector.

We investigated this (possibly strong) correlation by testing out several lattice bases, for dimensions 4-8, using different seeds. To this end, we sampled some lattice bases in certain lattice dimensions, and studied the correlation, generalizing it to higher dimensions (note that the correlation cannot be known as target vectors are generated). Due to the impossibility of presenting all the data, we showed three different correlations, for dimension 4 (seed 960), dimension 5 (seed 0) and dimension 8 (seed 456). These are representative of the full spectrum of obtained results.

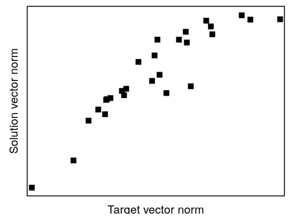
The scatter plots in Figure 1.3 show that our thesis holds true, as we can observe a moderately strong correlation in the terms we pointed out. The actual correlation depends upon the used lattice basis (i.e. dimension and seed). For instance, some bases showed an almost perfect/linear correlation (such as in dimension 4, seed 960), while others continued to show a correlation, although not as evident as the remaining lattices. These results show the best (Figure 1.3(a)), average (Figure 1.3(b)) and worst (Figure 1.3(c)) scenarios for all the lattices we tested out, thus giving us the confidence to affirm that a correlation holds.

Note that when the correlation is not as strong (for instance in Figure 1.3(c)), it does continue to hold for the shorter target vectors. In other words, although there are large target vectors that result in large solution vectors, it generally holds true that many small target vectors result in small solution vectors, thus supporting the proposed thesis.

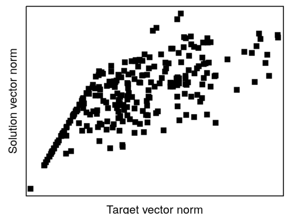
Given this data, we can conclude that, in general, as a correlation applies, meaning that a shorter target vector yields a shorter solution vector, then the



(a) Correlation for the basis in dimension 4 (seed 960).



(b) Correlation for the basis in dimension 5 (seed 0).



(c) Correlation for the basis in dimension 8 (seed 456).

FIGURE 1.3 Correlation between the norm of the target vectors and the norm of their respective solution vector, for three dimensions and seeds. We omitted both axes values as they are irrelevant for correlation purposes and added considerable complexity to the figures, thus making it difficult to read them.

shortest of the target vectors should, in general, result in the solution to the SVP. The correlation may become looser as we increase the lattice dimension, but it seems to continue to hold for the smaller target vectors (cf. the left-most vectors in Figure 1.3(c)), and we take advantage of that fact, as we show in the next section. This correlation is actually the basis of some of the algorithmic improvements we show in Section 1.5.

1.4.2 Percentage of target vectors that generate the shortest vector

From the previous results, we posed the hypothesis of whether we should only decode a small percentage of target vectors; these would necessarily contain the shortest solution vector (and, as a result, the solution for the SVP). However, note that this is only true when the target vectors are sorted by increasing norm.

Also, in general, the percentage of these target vectors should be larger as the correlation gets weaker (i.e. we would need to pick more target vectors as the correlation gets weaker for that specific basis). However, note that even if the correlation for a specific basis is off in general, but holds true for the first, shortest target vectors, then we would also need to decode a very small percentage of target vectors. In fact, as shown in Figure 1.3, even in the worst case of our tests, there is a correlation for the shortest target vectors, which supports our rationale.

To test this second hypothesis, we generated all target vectors, chose the smallest, and decoded it (i.e. computed its solution vector). We observed that the solution vector of the first target vector was always also the solution of the SVP, except for a handful of bases, which are shown in Table 1.3 (check the position column, which shows the position of the shortest vector when the first vector is not the shortest). This means that the shortest target vector does not yield the shortest solution vector (and the shortest vector of the lattice) in less than 0.27% of the bases we tested.

TABLE 1.3 Position of the target vectors that originate the shortest vector for the bases that failed.

Dimension	**#Incorrect solutions**	**Position(s)**	**%**
4	0	—	—
5	0	—	—
6	1	21	33.33
7	0	—	—
8	1	77	30.2
9	3	146, 260, 92	50.88
10	3	204, 200, 85	19.94
11	0	—	—
12	0	—	—
13	1	252	3.08
14	2	1246, 12865	78.53
15	5	4228, 911, 14181, 3495, 13599	43.28
16	1	2205	3.36
17	0	—	—
18	0	—	—
19	1	3010	0.57
20	2	11328, 856105	81.64

The percentage shown in the table regards the worst verified case of target vectors that need to be decoded so that we arrive at the optimal solution. However, note that these percentages may seem very high as we are testing very low dimensions. Moreover, the maximum percentage of target vectors we need to decode is highly dependent on the lattice basis we test. For instance, decoding 0.57% of the target vectors in dimension 19 would suffice to arrive at the shortest vector, while in dimension 20 one specific basis required as much as 81.64% of the target vectors. This said, there should be no clear trend in this regard.

As a result, we can affirm that, in general, sorting the target vectors by increasing norm will, very likely, lead us to find the shortest vector faster than randomly decoding target vectors as we generate them. This motivates a series of optimizations, which we explore in Section 1.5.

1.5 ALGORITHMIC OPTIMIZATIONS

In the following, we show a series of optimizations that are based on the previous analysis of the algorithm. In order to test lattices with such an execution time that allowed us to see the effects of the optimizations we implemented, we decided to use Machine A, as specified in Section 1.3 and g++ with the `O0` optimization flag. If we were to use Machine B and the `O3` optimization flag, some tests would run too fast, thus making it impossible to infer proper conclusions (increasing the lattice dimension would quickly lead us to hit the memory wall and impede proper testing).

1.5.1 Pruned decoding

Many of our optimizations stem from the fact that there is a relatively strong correlation between the norm of the target vectors and the solution yielded by the decoding process, as shown in Section 1.4. Therefore, we employ a key idea: we can filter out (or "prune") some of the target vectors, along with the decoding process, if their norm "is big". In particular, we should - with some confidence degree - be able to prune out target vectors that have a norm larger than the shortest norm (for any target vector) found at any given instant. In theory, we could also use the norm of the solution vectors (and in particular the norm of the shortest solution vectors found up until a certain execution point of the algorithm) to prune out some of the target vectors. Note, however, that this may introduce some uncertainty as a bigger target vector than the shortest (*solution*) vector found at any point of the algorithm may actually generate an even shorter solution vector. This is because target vectors may yield, throughout the decoding process, shorter solution vectors. Note that we have not studied this angle in our correlation analysis, presented in Section 1.4; this is merely an intuitive hypothetical relation that should work well in practice.

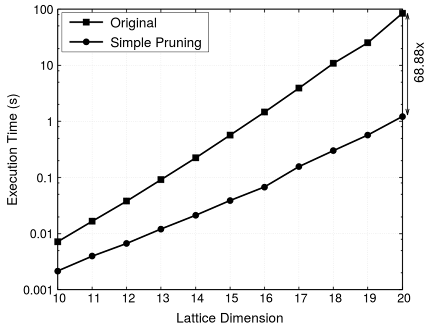
In fact, in our experiments, this has proven to be a very effective optimization, almost without compromising the solution. In other words, even with this optimization - which we generally call pruning as we prune the set of target vectors to test - we achieved the shortest vector of the lattice in almost 99.999% of the experiments we carried out in this section.

If we regard the target vectors - which are to be decoded - as a set, we can employ our optimization in the form of pruning. This may have several variants, but during our experiments, we found out that two forms are particularly effective.

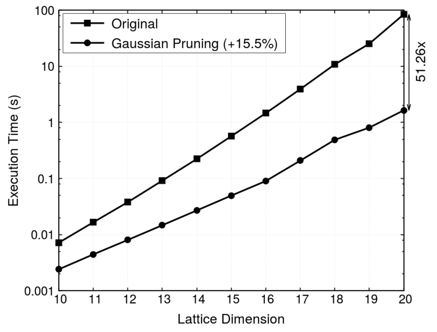
1.5.1.1 Simple pruning.

The first - and simplest - form of pruning we have employed is based on discarding target vectors whose norm is larger than the norm of the shortest (*solution*) vector found so far. We call this optimization "simple pruning". We do this by keeping a record of the shortest (*solution*) vector found throughout the execution. This optimization has resulted in significant speedups, as shown in Figure 1.4(a). The speedup of simple pruning also increased with the lattice dimension.

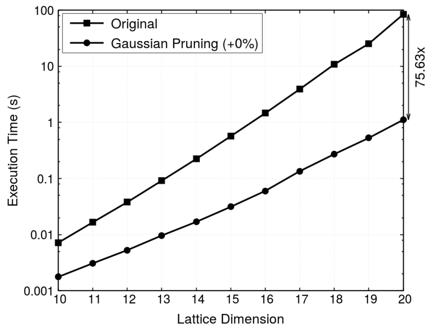
We note that although in theory this optimization may result in a compromised solution (because we may filter out the target vector that results in the shortest vector, as mentioned before), in practice, it barely happens (in our experiments it failed for 11 bases out of 7550). We tested this optimization for one thousand seeds of each dimension. The result was always coherent with that of a deterministic SVP-solver, therefore showing that simple pruning did not compromise the result in practice. We also expect this to be the case for the vast majority of lattices in higher dimensions.



(a) Original algorithm and the simple pruning version.



(b) Original algorithm and the Gaussian pruning version (added margin of 15.5%).



(c) Original algorithm and the Gaussian pruning version (added margin of 0.0%).

FIGURE 1.4 Original algorithm and pruned versions, from lattice dimension 10 to lattice dimension 20, on Machine A.

1.5.1.2 Gaussian pruning.

The Gaussian heuristic, presented in Equation 1.4, is a popular heuristic in the context of SVP-solvers. This heuristic estimates the length of the shortest vector of the lattice. It serves as the reference in the SVP-challenge[5], which

[5]https://www.latticechallenge.org/svp-challenge/

accepts entries of vectors whose norm is at most, 5% larger than the Gaussian heuristic. In this work, we refer to this delta, i.e. the amount added to the Gaussian heuristic, as the "added margin", in Equation 1.4 as α.

$$\alpha \cdot \frac{\Gamma(n/2+1)^{1/n}}{\sqrt{\pi}} \cdot (\det\ \mathcal{L})^{1/n} \tag{1.4}$$

$$\Gamma(x) = (x-1)!\ ,\ x \in \mathbb{Z}^{+} \tag{1.5}$$

As we observed a relatively strong correlation between the norm of the target vectors and the resulting solution vectors, together with the good results of simple pruning, we decided to test a pruned version based on the Gaussian heuristic (which we call Gaussian pruning). This reasoning is based on the fact that, in theory, if simple pruning works well, a pruning based on the Gaussian heuristic should also work well. This is due to two main ideas. First, there is a connection (although obviously not linear, otherwise the shortest target vector would always result in the shortest solution vector) between the norms of the target vector and the solution vector, as we can infer from the results in Section 1.4.2. Second, given this connection, applying the Gaussian pruning to the target vectors would indirectly allow us to reduce the set of target vectors that are likely to generate the shorter solution vectors. Given that these connections are not linear, although improbable, the algorithm may fail to find the shortest vector if Gaussian pruning is applied. In fact, following the same rationale, we can say that this is true for both simple and Gaussian pruning.

We tested Gaussian pruning with several error margins, for lattices in dimensions 10-20, testing 500 seeds from dimensions 11 to 15 and 50 seeds for dimensions 16-20 (due to time constraints). As Figure 1.4(b) shows, Gaussian pruning also works very well in practice, achieving speedup factors of as much as 51.26x. Again, we also expect the trend to continue as we increase the lattice dimension.

In our experiments, Gaussian pruning only yields an invalid solution, with an added margin of 15.5%, in 11 bases out of 7550. That is, in 7539 lattice instances, the algorithm always found the shortest vector.

We tested the added margin of the Gaussian pruning extensively. We started by using an added margin of 0% and the algorithm only failed to find the shortest vectors in 24 lattice bases (out of 7550). Therefore, in the vast majority of the lattice bases, the Gaussian pruning without an added margin works very well. However, to be comparable with the baseline - the reference algorithm - we needed to include an added margin that ensures the shortest vector is found.

We selected an added margin of 15.5% for the experiments which outputs the same number of wrong results (11 out of 7550) as the simple pruning, thus allowing us to compare both versions in terms of execution time. We note that although this added margin always resulted in an optimal solution, that may not be the case for all lattices in all dimensions, in which case we need to update the added margin accordingly.

Yet, we tested the performance of the Gaussian heuristic for various added margins. Not surprisingly, no added margin (i.e. Gaussian pruning with no added margin) showed to attain the best performance, which we depict in Figure 1.4(c). Note that running Gaussian pruning without any added margin only failed in 24 out of 7550 lattice bases.

This indicates that there may be potential on a Gaussian pruned version which works without added margin but another mechanism that detects over-pruning, i.e. discarding the target vectors that would lead to better solution vectors. Due to time limitations, we pushed this problem to future work.

1.5.1.3 Combined pruning.

Given the results of the two previous forms of pruning, we decided to combine them, i.e. executing them one after the other. Figure 1.5 shows the performance of a combination of simple and Gaussian pruning, in both orders, against the performance of the individual pruning optimizations and the baseline.

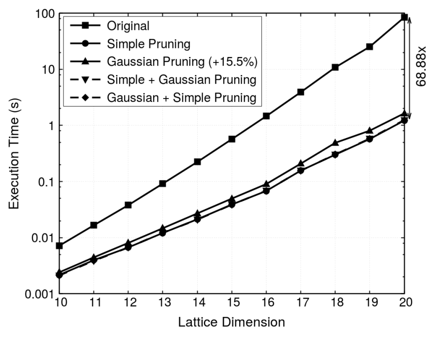


FIGURE 1.5 Original algorithm and the Gaussian (added margin of 15.5%), by both orders, combined and isolated, from lattice dimension 10 to lattice dimension 20, on Machine A.

As the figure shows, the combination of simple pruning with Gaussian pruning (with an added margin of 15.5%) does not deliver a speedup. Nevertheless, we were able to obtain a performance improvement in a large number of cases we conducted when refining some parameters (as these setups were overall less efficient than those in the figure and thus not very relevant, we refrained from showing them). We obtain a speedup of as much as 68.88x when compared to the baseline. This version also fails in only 12 bases (one more than simple or Gaussian pruning) out of all the 7550 instances tested.

1.5.2 Increasing norm sort

Given the effectiveness of the pruning optimizations, we decided to design a way so that shorter target vectors are executed first. To this end, we sort all target vectors by increasing norm before the actual execution of the algorithm, a process we refer to as "pre-sorting". This is also motivated by the results we arrived at in Section 1.4.2; From those results, we can conclude that executing the shorter target vectors first will lead us to shorter solutions first, thus increasing the pruning extent. Furthermore, as we will show throughout this text, memory usage is a problem in Voronoi-cell algorithms, and this optimization can theoretically improve this, as there is a much smaller set of target vectors to be decoded.

In theory, this enhances pruning as the number of pruned target vectors will be larger - with simple pruning or combined pruning (but not with Gaussian pruning) - if they are sorted (it does nothing if no pruning is applied, as all target vectors are executed either way). In particular, we know for a fact that the additional pruning is "safe" in the sense that it does not decrease the likelihood of solving the SVP.

We call this "safety" as, by pre-sorting the target vectors, we are prioritizing shorter target vectors that, as we saw in Figure 1.3, lead to shorter solution vectors in general. As such, we are effectively pruning out larger target vectors that would not lead to the solution of the SVP either way (with a high probability). In fact, they could have been decoded if pre-sorting was not used and they were some of the first target vectors in line of execution. Therefore, the solution provided by these larger target vectors would eventually be superseded by the solution vectors of smaller target vectors.

To implement this optimization, we have to re-arrange the computation of the algorithm, namely by generating the target vectors upfront (in contrast to calculating them on the fly - iteration by iteration - as it happens in the original algorithm), so that we can sort them (by increasing norm). We have implemented both the merge sort [11] and the quicksort [10] algorithms, and compared them against std::sort, the C++ standard sorting library. The latter, std::sort, performed better than the former and therefore we have performed the rest of the tests with this implementation (the GNU Compiler Suite also provides a parallel, OpenMP version of the std::sort algorithm, which is useful for higher dimensions).

Given that we want to sort the target vectors (which are stored in a matrix, in a row-wise manner), we have to compute an auxiliary vector with the norm of each target vector. This is obviously not required, but it does avoid computing the norm of a target vector each time it is needed. With the std::sort implementation, given the nature of the library function, we store the norm of each vector in a "struct", where each element holds one target vector and its norm; in this case, the sorting procedure is done by simply swapping the memory pointers to the elements, instead of actually having to move the data around.

The performance results for combined pruning with pre-sorting are shown in Figure 1.6. As the figure shows, we obtain a speedup of as much as 76.59x by pre-sorting the target vectors in function of increasing norm (which compares to 68.88x without sorting). As it happens without pre-sorting, the order by which the pruning techniques are applied with pre-sorting has very little significance in the execution time of the algorithm.

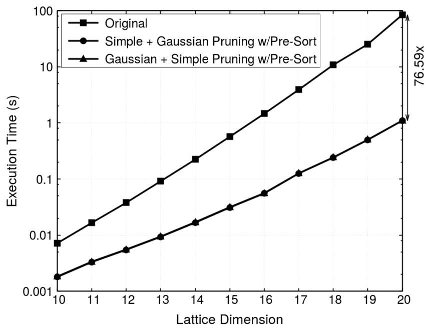


FIGURE 1.6 Original algorithm and the Gaussian (added margin of 15.5%) and Combined pruning, by both orders, with pre-sorting, from lattice dimension 10 to lattice dimension 20, on Machine A.

Evidently, it would be a very strong optimization if we were able to find a method to stop the algorithm once the shortest vector is reached, which is a common problem for many SVP-solvers. Nevertheless, we were still unable to come up with rules to stop the algorithm briefly after the shortest vector is found, as it happens with other SVP-solvers, such as sieving [22, 20]. However, we push this problem to future work.

1.6 PARALLEL IMPLEMENTATIONS FOR CPUS AND GPUS

In this section, we present both CPU- and GPU-parallel versions of the RelevantVectors algorithm, the algorithm that served as the basis of this work. In theory, this algorithm is embarrassingly parallel, as there are no dependencies between iterations, i.e., we can execute several *Decode*s concurrently. We used OpenMP for the parallel CPU version and CUDA for the GPU version.

1.6.1 CPU

The OpenMP compiler directives were applied to the main loop of the algorithm Line 6 in Algorithm 1, where the generation of the target vectors takes place, followed by the decoding of the mentioned vectors. As this process is independent between iterations, and there are no subsequent data races, threads can run concurrently.

On top of parallelizing the algorithm, we have employed other optimizations to the algorithm, regarding general memory usage/consumption and memory access.

First, the result of decoding a target vector, if valid, yields two solution vectors. As such, if we were to store every result of every decode, a matrix of dimension $2(2^n - 1) \times n$ would be required, for an n-dimensional lattice. This is impractical, as the memory requirement for this matrix grows exponentially with the lattice dimension. To solve this issue, instead of storing every solution vector, we only store the shortest vector found at the end of each decode procedure. This decreases the size of the matrix used to store the solution vector to $1 \times n$ (or $2 \times n$ if we were to store both results of each decode). It requires the use of a critical region so that threads cannot simultaneously access these variables, which would lead to data races and a potentially incorrect result.

Second, originally the matrices were implemented as an array of arrays; while this provides a very natural indexing notation, it is not very efficient from a memory standpoint. Not only it requires several memory allocations (and deallocations) for each matrix, there is no guarantee that the required memory is allocated continuously in RAM. Therefore, the implementation of matrices was changed (from an array of arrays) to a single, large vector. This increases indexing computation slightly but improves memory locality considerably.

Algorithm 2 shows the pseudo-code of the OpenMP-based parallel version of the RelevantVectors algorithm.

Algorithm 2: Parallel RelevantVectors

```
Input: Basis matrix $\mathbf{B}$
Output: Relevant Vectors $\mathbf{N}$
$\mathbf{M}$ = Reduce($\mathbf{B}$);  /* for example, using the LLL algorithm */
[$\mathbf{Q}$, $\mathbf{R}$] = QR decomposition of $\mathbf{M}$;
$\mathbf{G} = \mathbf{R}^T$;
$\mathbf{H} = \mathbf{G}^{-1}$;
$\mathbf{N} = \varnothing$;
min_norm = $\infty$;
#pragma omp parallel for
forall vectors $\mathbf{s} \in \mathcal{M}$ do
    $\mathbf{X}$ = AllClosestPoints($\mathbf{M}$, $\mathbf{H}$, $\mathbf{Q}$, $\mathbf{s}$);
    #pragma omp critical
    if $||2\mathbf{x} - 2\mathbf{s}|| <$ min_norm then
        min_norm = $||2\mathbf{x} - 2\mathbf{s}||$;
        $\mathbf{N} = \{2\mathbf{x} - 2\mathbf{s} : \mathbf{x} \in \mathbf{X}\}$;
return N
```

1.6.1.1 Original version (no pruning and no pre-sorting).

We first parallelized the original version of the algorithm (i.e. without pruning and without pre-sorting). Given that the execution time of each iteration is different (as decoding different target vectors may be faster or slower), there may be work imbalance among threads. In preliminary tests, we tested the OpenMP static scheduler, but the results were not, unsurprisingly, optimal. For that reason, the experiments we report were conducted with the OpenMP dynamic scheduler, which assigns work to threads as they complete the previous tasks, thus balancing out the workload. Although this strategy does not guarantee perfect load balancing, it usually minimizes the imbalance substantially (usually at the cost of a given overhead, which may be smaller or bigger depending on circumstances). As such, we still expect some threads to finish ahead of others.

Figure 1.7 shows the execution time of the algorithm, on Machine A, for lattices in dimension 16-20, and 1-56 threads. For readability purposes, we display the speedups in Table 1.4.

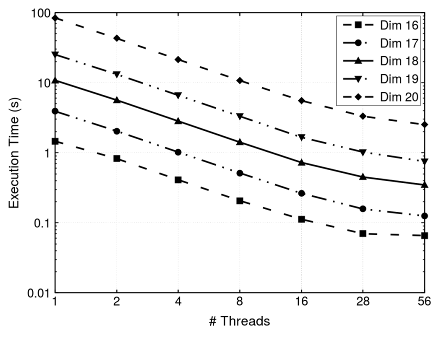


FIGURE 1.7 Execution time for the parallel algorithm, on lattices in dimensions 16-20, using 1-56 threads on Machine A.

TABLE 1.4 Speedups on Machine A, parallel non-pruned implementation running with 1-56 threads, in comparison to the parallel non-pruned version running with a single thread.

Dimension	**16**	**17**	**18**	**19**	**20**
2 Threads	1.772	1.917	1.915	1.914	1.961
4 Threads	3.556	3.833	3.824	3.815	3.933
8 Threads	7.072	7.615	7.642	7.604	7.849
16 Threads	12.990	14.830	14.890	15.170	15.190
28 Threads	20.910	24.820	24.020	24.660	25.370
56 Threads	22.210	31.190	31.140	33.600	33.440

We achieved higher speedups for higher lattice dimensions, due to lower thread creation latency and improved the overall workload distribution. This is particularly important because we aim at using our implementation in large dimensions - as large as possible.

We also developed a parallel version of the optimized version, with pre-sorting both turned on and off, with OpenMP. As we have seen, the order of the optimizations was not relevant for performance, so we only used one order. We also parallelized the generation of target vectors, as they can be executed independently (both with and without pre-sorting).

1.6.1.2 Pruned version without sorting.

As shown in Figure 1.8, the parallel combined pruning version also scales well (cf. Table 1.5 for readability purposes). Nevertheless, scalability is overall a little lower than the non-optimized implementation due to the critical section necessary for the optimizations of this particular version (note the contention for more than 8 threads in Figure 1.8).

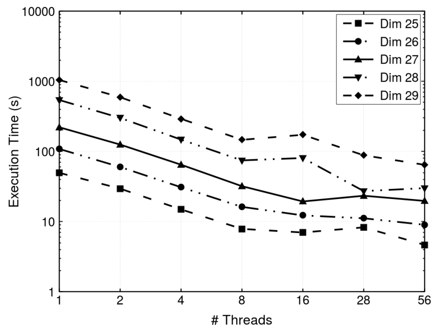


FIGURE 1.8 Execution time for the combined Gaussian + Simple pruned version of the algorithm, on dimensions 25-29, using 1-56 threads on Machine A.

TABLE 1.5 Speedup of the parallel implementation for Gaussian and Simple pruning, on Machine A.

Dimension	25	26	27	28	29
2 Threads	1.681	1.810	1.766	1.800	1.763
4 Threads	3.322	3.514	3.419	3.667	3.581
8 Threads	6.297	6.694	6.916	7.254	7.128
16 Threads	7.085	8.856	11.412	6.706	6.040
28 Threads	5.986	9.732	9.464	19.870	11.850
56 Threads	10.630	12.050	11.220	18.020	16.190

As mentioned in Section 1.5.1.2, there is the possibility that the chosen added margin may fail for some lattice bases that were not tested before, as it is impossible to know upfront which added margin guarantees the optimal solution. The number of bases that the algorithm did not return the optimal solution was still residual. We also note that arriving at a short vector, as opposed to the shortest vector, is still important, especially in the context of a relaxed version of the SVP, usually referred to as α-SVP. In this work, we will not expand on this topic, even though we note that short vectors are still an important result in this context.

1.6.1.3 Pruned version with sorting.

Regarding the optimized version with the pre-sorting phase, in this setup, we need to generate all target vectors upfront, and sort them, before the algorithm actually computes the SVP. This uses up more memory than the original algorithm, as we will show next.

As mentioned in Section 1.5.2, the GNU Compiler Suite already implements an OpenMP version of the std::sort algorithm, which we used in the sorting phase. This way, the procedure is entirely parallel, including the generation and sorting of the target vectors, except for the synchronization among all the threads (implemented with a critical section) to update the shortest norm found, should they find one.

Figure 1.9 shows the execution time up to dimension 29, which is the highest lattice dimension we can test with 128 GB of RAM. In dimension 29, using pre-sorting results in a speedup of almost 30% (for 28 threads) and 40% (for 56 threads), compared to the non-sorted version. With 56 threads, using pre-sorting is also faster than the non-sorting version for dimension 28, by approximately 26%. Until dimension 28 (for 56 threads) and dimension 29 (for 28 threads), the non-sorting version is faster, given that the time to generate all target vectors, store/read them from memory and sort them is higher than the gain throughout the algorithm.

It is worth note that this version requires more memory than the previous ones. In the pruned version without pre-sorting, target vectors are decoded and generated one by one (and the algorithm should stop long before all target vectors are explored, per our optimizations). In this version, all target vectors are generated (in order to be sorted) upfront, which consumes more memory than in the non-sorted version. In essence, the pre-sorting creates a trade-off between memory and execution time (because the solution is achieved faster). The execution time to sort the vectors upfront was never relevant in our tests (i.e. even with the sorting phase, the final time to solution was always lower).

Regarding memory, we should note that in the parallel versions, the memory required increases exponentially with the lattice dimension (as per the original algorithm) but also linearly with the number of threads being used. This happens because each iteration (and thus each thread) requires its own auxiliary structures for the correct working of the decode function. These ma-

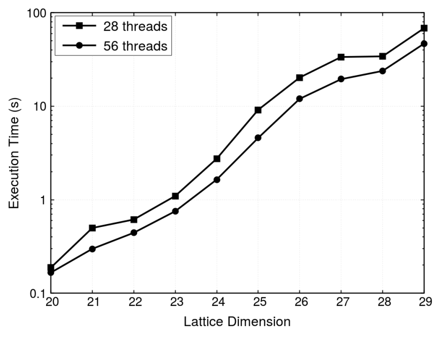


FIGURE 1.9 Execution time for the combined Gaussian + Simple pruned version of the algorithm, with pre-sorting, for dimensions 20-29, using 28 and 56 threads on Machine A.

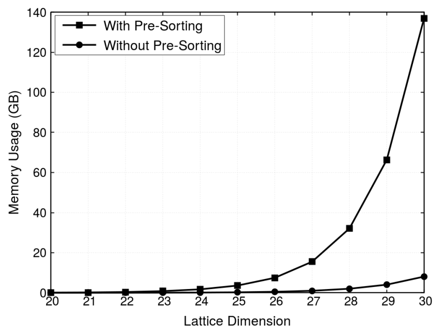


FIGURE 1.10 The calculated memory usage required by our combined pruning implementation (essentially for the target vector-matrix), both with and without pre-sorting. The number of threads does not make a difference.

trices are initially allocated with a certain number of rows (the number of columns is equal to the dimension of the problem) and, when needed, are extended via reallocation.

Figure 1.10 shows the (calculated) memory usage of the implementation in the worst-case scenario for a given dimension (i.e. the largest size measured among all bases in a given dimension), both when sorting is used and when it is not. As machine A has 128 GB of RAM memory, we were able to test the optimized version with pre-sorting up until dimension 29. Running dimension 30 would require around 137 GB of memory available.

Table 1.6 shows the estimated memory usage of the combined pruning implementation with pre-sorting, for dimensions 20 through 80 in increments of 10, using 28 threads.

Dimension	**Estimated memory usage**
20	83.89 MB
30	128.85 GB
40	175.92 TB
50	225.18 PB
60	276.70 EB
70	330.57 ZB
80	386.86 YB

TABLE 1.6 Estimated memory usage of the combined pruning implementation for dimensions 20-80, with pre-sorting.

1.6.2 GPU

As mentioned at the beginning of Section 1.6, we also present a parallel version for GPUs, in CUDA, similar to [5]. Due to the inefficiency of software-based critical sections in CUDA, we were forced to employ a larger matrix to hold all solution vectors, similarly to the original algorithm. Should we be able to implement an efficient critical section, threads would be able to compare the solution vector they arrive at, against the shortest one found so far, without the need to store the larger ones. This same reason prevented implementing a CUDA version of the pruning optimization. Results are, therefore, presented for the non-optimized algorithm. In this version, we calculate the target vectors on the fly, as in the non-pruned and pruned versions without pre-sorting (and in contrast to our CPU version with pre-sorting).

Our CUDA implementation contains a single kernel. We set up the kernel so that each thread decodes a single target vector unless the available memory is not enough (note that each thread allocates memory for auxiliary structures). For instance, running dimension 20 on our GPU implies that each thread will decode more than one vector. The first step of the kernel is to generate the target vectors, decodes them and stores the solution vectors in the final matrix (list of vectors), similar to the CPU version. In the meantime, the result of the decode is checked; if the test holds, the vector is stored in the final matrix, otherwise, the thread dies (until dimension 19) or proceeds to the following iteration (in dimension 20).

Until dimension 19, we set the number of threads equal to the target vectors (in practice we set the number of blocks and threads per block). We set 128 threads per block, so the number of blocks changes based on the number of target vectors. As we said, each thread is responsible for generating and decoding a single target vector.

Figure 1.11 shows the execution time of our CUDA implementation, running with as many threads as target vectors (except for dimension 20, where that number is halved), for several lattice dimensions.

The figure also includes the execution time of our CPU version of the

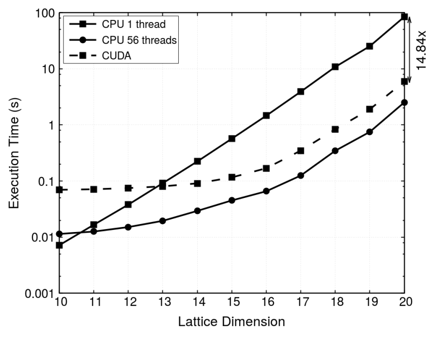


FIGURE 1.11 Execution time for dimensions 10-20, for the non-pruned CPU algorithm (1/56 threads on Machine A) and the parallel GPU non-pruned algorithm (on Machine B). Note: The CPU execution times are those of Section 1.6.1 for the non-pruned algorithm.

non-optimized algorithm, both with one and 56 threads. Up until dimension 13, the GPU implementation is slower than the CPU implementation, as the penalty for transferring memory (matrices M, H, Q and N) over the PCI-express bus to the GPU only becomes diluted/clouded for bigger dimensions and the launch time of the GPU kernel (not excluded in the results) is also diluted for bigger dimensions.

The GPU implementation is almost 15x times faster than the sequential version and about 2,34x slower than the CPU running with 56 threads. As we can see in the figure, the difference between both implementations gets smaller with the lattice dimension, which is expected due to the penalty of memory transfer and GPU initialization. Thus, we expect this GPU version to beat the CPU version with 56 threads, for a sufficiently large lattice dimension (which we cannot test due to memory limitations).

1.7 DISCUSSION

There are two different angles of our research that deserve comments. First, the algorithmic optimizations that we propose, which greatly improve the algorithm. The idea of speeding up SVP-solvers by empirical observations on vectors is not new e.g. [6], but it was never applied to Voronoi cell-based algorithms, to our best knowledge. The study of the correlation between the norm of target vectors and their solution is, to our knowledge, unprecedented. The "simple" and the Gaussian pruning is motivated by some optimizations implemented in other SVP-solvers (for instance, the simple pruning is used in sieving algorithms, while Gaussian pruning is used in enumeration algorithms as a way to define a radius for a search space or prune the enumeration tree [8]).

Objectively, these optimizations greatly improve the algorithm, and sorting target vectors is another great optimization as it decreases time to solution even further. The sorting procedure is also very efficient and can be done in parallel, therefore we see no concerns regarding adding this pre-processing to the overall algorithmic routines.

A second angle for discussion is the performance of our parallel versions, both for CPUs and GPUs. Although not all of our CPU implementations scale linearly, they do scale fairly well. We are confident that, if further developments are made to the algorithm, these implementations could be used for high lattice dimensions. The GPU implementation, on the other hand, has to be revisited in order to integrate the proposed pruning optimizations. GPUs currently lack efficient synchronisation mechanisms. This prevents sharing of information (such as the current pruning norm) among threads while still maintaining scalability. An interesting line of research in this context should be to re-write the algorithm differently so that synchronization could either be avoided or made parallel.

1.8 CONCLUSIONS

Attacks to post-quantum lattice-based cryptosystems require solving the computationally hard Shortest Vector Problem (SVP). Different families of SVP-solvers have been suggested over the last two decades, including Voronoi cell-based algorithms. Proposed back in 2002 by Agrell et al., this family of algorithms has not been optimized since, under the claim that its memory complexity (exponential with the number of dimensions) renders it unpractical even for low dimensions. However, Voronoi cell-based algorithms exhibit a number of characteristics that justify a thorough study of their practicality when a few optimizations are employed. In particular, their time complexity is asymptotically very interesting, which could allow them to become competitive with other SVP-solvers if the memory barrier can be overcome. Indeed there is plenty of room for practical optimizations, which can eventually lead to tractable implementations for high-dimensional lattices, unleashing their true potential.

This work addressed the reduction of execution time of the "Relevant Vectors" Voronoi cell-based algorithm, by tackling two different axes: i) algorithmically reducing the number of operations required to reach a solution to the SVP, and ii) parallelizing it for both CPUs and multi-core CPUs.

In order to reduce the workload we hypothesized that there is a correlation between the norm size of the target vectors and the solution vectors. This correlation was demonstrated to hold, which allowed us to propose pruning target vectors based on the length of the shortest solution vector observed this far and/or the Gaussian heuristic. Also, we have shown that pre-sorting the target vectors by increasing norm allows for a more effective pruning (by reordering computations), accelerating our optimized version further, notwithstanding the additional sorting time. Altogether, our optimizations improved the throughput performance approximately 77x compared to the baseline implementation. Adding sorting on top of pruning provided an additional speedup of as much as 40% up to dimension 29, but we estimate that it would be considerably superior, should we be able to test higher dimensions.

The main drawback with sorting is that it currently requires storing all target vectors, which results on a huge memory consumption. Naturally, we could ignore/cut-off a substantial percentage (e.g., 70%) of the largest target vectors right in the pre-processing stage, significantly reducing memory usage. However, we have not done so yet as we would like to look for a cut-off formula that translates to an approximated likelihood of still finding the shortest vector (which would be a very interesting result in the context of the approximate SVP problem). We note the potential of this idea, given that Voronoi is not tractable in practice solely because target vectors become a bottleneck memory-wise.

Additionally, we have shown that the algorithm and our optimizations are well suited for multi-core CPU machines, as we devised and implemented a scalable parallel version. We also optimized the algorithm's memory map and found that dynamic scheduling is mandatory since decoding time varies for different target vectors. Our implementation scales linearly on multi-core CPUs up to 28 threads and can even take advantage of SMT, although the benefit is reduced given that the problem is compute-bound. We found no reason why similar scalability will not hold for higher thread counts.

We also implemented the original algorithm on a GPU, using the CUDA framework. The optimized pruning algorithm could not be tested, since it would require recurrent use of critical sections, currently not efficiently supported by CUDA. Therefore, we could not use pruning to reduce the workload and had to store all computed solution vectors, further increasing memory consumption. Although our GPU version was never faster than our 56 threads CPU version, we observed that the gap between the CPU and the GPU generally decreases with lattice dimension. This is a very promising result, hinting that the GPU could become faster for higher lattice dimensions. Also the CPU/GPU memory transfer penalty will be diluted for such higher dimensions, further contributing to the GPU advantage. In the future, we expect to develop better data structures for the GPU and optimize the CUDA code, such that experiments with higher lattice dimensions become feasible.

This chapter represents a step forward on making Voronoi cell-based SVP-solvers practical. It has shown that there is plenty of room for algorithmic optimizations, namely workload reduction by pruning large target vectors. It has also demonstrated that multi-core CPU parallel solutions scale and are efficient. GPU solutions show a promising trend as the lattice dimensionality increases, but further support is required for synchronisation primitives enabling efficient critical regions controlled access. The exponential space complexity of Voronoi cell-based algorithms remains a challenge which has not been directly addressed in this chapter. Educated discarding of a percentage of the largest target vectors on the sorting stage could represent a first step on the right direction, reducing the constants associated with this complexity.

1.8.1 Open problems

This work leads to many lines of future work. In particular, we think that it would be interesting to:

* Find a stopping criterion so that our optimized algorithm stops shortly after the solution is found;
* Reduce the memory requirements of our GPU implementation by developing new data structures;
* Optimize our GPU implementation further, to take full advantage of the architecture;
* Implement a heterogeneous version of our algorithm;
* Reduce the parallel CPU version memory requirements, by using only a small part of the Voronoi cell.

1.9 ACKNOWLEDGMENTS

This work was financed by national funds from INESC TEC and the Portuguese funding agency, Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020. This work was also financially supported by Instituto de Telecomunicações and Fundação para a Ciência e a Tecnologia under project UIDB/50008/2020. Artur Mariano was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—Projektnummer 382285730.

Bibliography

[1] Erik Agrell, Thomas Eriksson, Alexander Vardy, and Kenneth Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, Aug 2002.

[2] 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*, pages 717–746, Cham, 2019. Springer International Publishing.

[3] Daniel Bernstein, Johannes Buchmann, and Erik Dahmen, editors. *Post-quantum cryptography.* Springer, 2009.

[4] Fabio Correia. Assessing the hardness of SVP algorithms in the presence of CPUs and GPUs. Master's thesis, University of Minho, Braga, Portugal, 2014.

[5] G. Falcao, F. Cabeleira, A. Mariano, and L. Paulo Santos. Heterogeneous implementation of a voronoi cell-based svp solver. *IEEE Access*, 7:127012–127023, 2019.

[6] Robert Fitzpatrick, Christian Bischof, Johannes Buchmann, Özgür Dagdelen, Florian Göpfert, Artur Mariano, and Bo-Yin Yang. Tuning GaussSieve for speed. In *LATINCRYPT*, pages 288–305, 2014.

[7] Nicolas Gama, Phong Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT*, pages 257–278, 2010.

[8] Nicolas Gama, Phong Q. Nguyên, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT*, pages 257–278, 2010.

[9] Daniel Goldstein and Andrew Mayer. On the equidistribution of Hecke points. *Forum Mathematicum*, 15:165–189, 01 2003.

[10] Charles Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–, July 1961.

[11] Donald Knuth. *The art of computer programming*, volume 3, Sorting and searching. Addison-Wesley, 1998.

[12] Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *CRYPTO*, pages 3–22, 2015.

[13] Thijs Laarhoven. Evolutionary techniques in lattice sieving algorithms. *CoRR*, abs/1907.04629, 2019.

[14] A.K. Lenstra, H.W.jun. Lenstra, and Lászlo Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.

[15] A. Mariano, T. Laarhoven, F. Correia, M. Rodrigues, and G. Falcão. A practical view of the state-of-the-art of lattice-based cryptanalysis. *IEEE Access*, 5:24184–24202, 2017.

[16] Artur Mariano. *High performance algorithms for lattice-based cryptanalysis*. PhD thesis, Technische Universität Darmstadt, Darmstadt, Germany, 2016.

[17] Paulo Martins, Artur Mariano, and Leonel Sousa. A survey on fully homomorphic encryption: an engineering perspective. In *ACM Computing Surveys*, page To appear, 2017.

[18] Daniele Micciancio and Oded Regev. *Post-Quantum Cryptography*, chapter Lattice-based Cryptography, pages 147–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[19] Daniele Micciancio and Panagiotis Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on Voronoi cell computations. *STOC*, pages 351–358, 2010.

[20] Daniele Micciancio and Panagiotis Voulgaris. Faster Exponential Time Algorithms for the Shortest Vector Problem. *SODA*, pages 1468–1480, 2010.

[21] Phong Nguyen and Jacques Stern. The Two Faces of Lattices in Cryptology. In *CaLC*, pages 146–180, 2001.

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

[23] Oded Regev. *Lattice-Based Cryptography*, pages 131–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[24] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(2–3):181–199, 1994.

[25] Peter Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, pages 124–134, Washington, DC, USA, 1994. IEEE Computer Society.

[26] Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997.

[27] Joop van de Pol. Lattice-based cryptography. Master's thesis, Technische Universiteit Eindhoven, The Netherlands, 2011.