View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by tuprints

Computing Shortest Lattice Vectors on Special Hardware

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des Grades
Doktor rerum naturalium (Dr. rer. nat.)
von

Dipl.-Inform. Dipl.-Math. Michael Schneider

geboren in Frankfurt am Main.

Referenten: Prof. Dr. Johannes Buchmann
Prof. Dr. Chen-Mou Cheng

Tag der Einreichung: 7. September 2011
Tag der mündlichen Prüfung: 11. November 2011

Hochschulkennziffer: D 17

Darmstadt 2011

List of Publications

[KSD+11] Po-Chun Kuo, Michael Schneider, Özgür Dagdelen, Jan Reichelt, Johannes Buchmann, Chen-Mou Cheng, and Bo-Yin Yang. Extreme enumeration on GPU and in clouds. In *CHES 2011*, volume 6917 of *LNCS*, pages 176–191. Springer, 2011.

[SG11] Michael Schneider and Norman Göttert. Random sampling for short lattice vectors on graphics cards. In *CHES 2011*, volume 6917 of *LNCS*, pages 160–175. Springer, 2011.

[Sch11b] Michael Schneider. Sieving for shortest vectors in ideal lattices. IACR Cryptology ePrint Archive, Report 2011/458, 2011.

[MS11] Benjamin Milde and Michael Schneider. A parallel implementation of GaussSieve for the shortest vector problem in lattices. In *PaCT 2011*, volume 6873 of *LNCS*, pages 452–458. Springer, 2011.

[CHS11] Pierre-Louis Cayrel, Gerhard Hoffmann, and Michael Schneider. GPU implementation of the Keccak hash function family. In *ISA*, volume 200 of *Communications in Computer and Information Science*, pages 33–42. Springer, 2011.

[Sch11a] Michael Schneider. Analysis of Gauss-sieve for solving the shortest vector problem in lattices. In *WALCOM 2011*, volume 6552 of *LNCS*, pages 89–97. Springer, 2011.

[RSS10] Markus Rückert, Michael Schneider, and Dominique Schröder. Generic constructions for verifiably encrypted signatures without random oracles or NIZKs. In *ACNS 2010*, volume 6123 of *LNCS*, pages 69–86. Springer, 2010.

[DS10] Özgür Dagdelen and Michael Schneider. Parallel enumeration of shortest lattice vectors. In *Euro-Par 2010*, volume 6272 of *LNCS*, pages 211–222. Springer, 2010.

[HSB+10] Jens Hermans, Michael Schneider, Johannes Buchmann, Frederik Vercauteren, and Bart Preneel. Parallel shortest lattice vector enumeration on graphics cards. In *AFRICACRYPT 2010*, volume 6055 of *LNCS*, pages 52–68. Springer, 2010.

[RS10] Markus Rückert and Michael Schneider. Estimating the security of lattice-based cryptosystems. IACR Cryptology ePrint Archive, Report 2010/137, 2010.

[SB10] Michael Schneider and Johannes Buchmann. Extended lattice reduction experiments using the BKZ algorithm. In *Sicherheit 2010*, volume 170 of *LNI*, pages 241–252. GI, 2010.

[SBL09] Michael Schneider, Johannes Buchmann, and Richard Lindner. Probabilistic analysis of LLL reduced bases. In *WEWoRC 2009*, volume 6429 of *LNCS*. Springer, 2009.

[BLRS09] Johannes Buchmann, Richard Lindner, Markus Rückert, and Michael Schneider. Post-quantum cryptography: lattice signatures. *Computing*, 85(1-2):105–129, 2009.

[BDS08] Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle tree traversal revisited. In *PQCrypto 2008*, volume 5299 of *LNCS*, pages 63–78. Springer, 2008.

Acknowledgments

First and foremost, I am indebted to Johannes Buchmann for giving me the opportunity to write this thesis under his guidance. His encouraging and challenging advice made my research the success it finally got. I am thankful for the pleasant end encouraging research atmosphere in our CDC group. On the scientific side, my thanks go to Richard and Markus for introducing me into the topic of lattice-based cryptography as well as Özgür for our joint work. In addition, I would like to thank Paul, Pierre-Louis, Marc, Vangelis, and Dominique for the best office atmosphere that I could imagine, and Marita and Roswitha for their organizational support.

During my work on this thesis I had the great opportunity to visit foreign departments and work with brilliant people. I really enjoyed the work with Chen-Mou Cheng, Jens Hermans, Manfred Madritsch, Po-Chun Kuo, and Bo-Yin Yang. Thanks go to all my further co-authors Erik Dahmen, Norman Göttert, Gerhard Hoffmann, Bemjamin Milde, Jan Reichelt, Bart Preneel and Fre Vercauteren. I also enjoyed the inspiring discussions on lattice problems with Nicolas Gama and our work on the SVP Challenge. For financial support I would like to thank CASED, DFG, and EcryptII. For proofreading preliminary version of my thesis I would like to thank Luisa, Markus, Nadja, Paul, Pierre-Louis, Richard, Steph, and Vangelis.

There are few people who showed me that work and research is not the most important part of life. I am indebted to Alex, Andi, Andy and his family, Ann-Kristin, Carsten, Dirk and his family, Katrin, Kirstin, Nico, Sarah, Sascha, Svenja, and Luisa for making the other parts of my life such a pleasure. Thanks a lot! I would like to thank my parents and my grandmother for their interest and support over the last decades. And last but not least, I am very grateful for the help and support of my brothers Martin and Jochen as well as their partners Nadja and Katharina. I always appreciate being around you.

Darmstadt, November 2011 *Michael Schneider*

Abstract

The shortest vector problem (SVP) in lattices is related to problems in combinatorial optimization, algorithmic number theory, communication theory, and cryptography. In 1996, Ajtai published his breakthrough idea how to create lattice-based one-way functions based on the worst-case hardness of an approximate version of SVP. Worst-case hardness is one of the outstanding properties of all modern lattice-based cryptographic schemes. Furthermore, there are no sub-exponential time algorithms known solving SVP, even on potential, strong quantum computers. These facts distinguish the shortest vector problem as a good basis for modern cryptography.

In order to theoretically assess the security of lattice-based cryptosystems, knowledge of the asymptotic runtime of SVP solvers is an important issue. For selection of practical parameters however, the average-case behaviour of these algorithms is at least as important. SVP solvers are applied as subroutine in so-called lattice basis reduction algorithms. These build the cornerstone of the fastest attacks on lattice-based cryptosystems. Therefore, improving SVP algorithms directly affects the fastest practical attacks on lattice-based cryptosystems.

Building on existing serial SVP algorithms, this thesis presents multiple approaches towards estimating the practical hardness of the shortest vector problem. We employ various special hardware, ranging from multicore CPUs and graphics cards to “supercomputers” and compute clouds. We develop parallel algorithms and assess their practical running times and scalability. Among others, we present our parallel version of the Extreme Pruning Enumeration algorithm, the currently fastest SVP solver available worldwide. Our implementation set the current records in the SVP challenge, the mostly deployed public SVP solver competition.

The influence of our work on the security of lattice-based cryptosystems is twofold. First, we help assessing the strength of worst-case problems that build the theoretical basement of lattice-based cryptography. Second, we show how to improve the fastest practical attacks on these systems in the average case.

As further result, we present a variant of the sieving algorithm to solve the shortest vector problem in ideal lattices. Ideal lattices are the most important type of lattices in cryptography. Our algorithm is the first to exploit their special structure, allowing us to find shortest vectors faster than in regular lattices.

Zusammenfassung

Schwere Berechnungsprobleme bilden die Grundlage für kryptographische Systeme. In der modernen Kryptographie wird versucht, das Spektrum dieser Probleme zu erweitern, und neben den bekannten wie dem Faktorisieren ganzer Zahlen werden neuartige Probleme betrachtet. Darunter befindet sich auch das Problem, kürzeste Vektoren in einem Gitter zu finden ("shortest vector problem" - SVP). Im Jahr 1996 veröffentlichte Ajtai seine bahnbrechende Idee zur Erstellung gitterbasierter Einweg-Funktionen auf der Grundlage einer approximativen Variante des SVP. Das Außergewöhnliche daran ist, dass das Lösen einer zufälligen Instanz des SVP beweisbar mindestens so schwer ist wie das Lösen der schwierigsten Instanzen eines verwandten Problems. Diese "worst-case hardness" ist eine der herausragenden Eigenschaften aller moderner, gitterbasierter Kryptographie-Verfahren. Darüber hinaus sind keine subexponentiellen Algorithmen zur Lösung des SVP bekannt, auch nicht für potenzielle Quantencomputer. Diese Tatsachen zeichnen das "shortest vector problem" als eine gute Grundlage für die moderne Kryptographie aus.

Um die Sicherheit gitterbasierter Kryptosysteme theoretisch zu beurteilen, ist die Kenntnis der *asymptotischen* Laufzeit von SVP-Lösern wichtig. Nur so lässt sich feststellen, ob die Annahmen über die Schwierigkeit von SVP gerechtfertigt sind. Für die Auswahl praktischer Parameter ist jedoch das *durchschnittliche* Laufzeitverhalten dieser Algorithmen ebenso wichtig. SVP-Löser werden als Unterprogramme in sogenannten Gitter-Reduktions-Algorithmen verwendet. Diese bilden die Basis der schnellsten praktischen Angriffe auf Kryptosysteme. Daher wirkt sich die Verbesserung von SVP Algorithmen hier direkt aus.

Aufbauend auf den vorhandenen seriellen SVP-Algorithmen stellt diese Arbeit mehrere Ansätze zur Abschätzung der praktischen Schwierigkeit des SVP vor. Dabei verwenden wir unterschiedliche Spezial-Hardware, wie Multicore-CPUs, Grafikkarten oder "Supercomputer". Wir entwickeln parallele Algorithmen und bewerten ihre praktischen Laufzeiten. Unter anderem präsentieren wir unsere parallele Version des "Extreme Pruning Enumeration"-Algorithmus, derzeit der schnellste verfügbare SVP-Algorithmus weltweit. Unsere Implementierung hält die aktuellen Rekorde in der SVP Challenge, einem öffentlichen Wettbewerb zum Vergleich von SVP-Lösern.

Unsere Arbeit beeinflusst die Sicherheit der Gitterkryptographie in zweierlei Hinsicht. Zum einen liefern wir einen Beitrag zur Beurteilung der Schwere der Worst-Case-Probleme, die die theoretische Sicherheitsgrundlage darstellt. Zum anderen zeigen wir, wie man die schnellsten praktische Angriffe auf diese Systeme im durchschnittlichen Fall verbessern kann.

Als weiteres Ergebnis präsentieren wir eine Variante des “Sieving”-Algorithmus, der ebenfalls kürzeste Vektoren findet, für Idealgitter. Idealgitter sind die wichtigste Art von Gittern in der Kryptographie. Unser Algorithmus ist der Erste, der die spezielle Struktur dieser Gitter ausnutzt, so dass wir kürzeste Vektoren schneller als in regulären Gittern finden können.

Contents

1 Introduction **1**

2 Notation and Definitions **7**
2.1 Lattices . . . 7
2.2 Lattice Reduction . . . 12
2.3 The Shortest Vector Problem . . . 14
2.4 Parallelization and Special Hardware . . . 22

3 Parallel Enumeration on Multicore CPUs **27**
3.1 Parallel Algorithm Design and Implementation . . . 27
3.2 Experimental Results . . . 32

4 Parallel Enumeration on GPU **37**
4.1 Parallel Algorithm Design and Implementation . . . 37
4.2 Experimental Results . . . 41

5 Extreme Pruning Enumeration on GPU and in Clouds **47**
5.1 Cloud Computing and Amazon EC2 . . . 48
5.2 Parallel Algorithm Design and Implementation . . . 50
5.3 Experimental Results . . . 56

6 Parallel Random Sampling on GPU **61**
6.1 Random and Simple Sampling Reduction . . . 62
6.2 Parallel Algorithm Design and Implementation . . . 63
6.3 Experimental Results . . . 67

7 Sieving in Ideal Lattices **73**
7.1 IdealListSieve Algorithm . . . 74
7.2 Predicted Advantage of IdealListSieve . . . 77
7.3 Experimental Results . . . 78

8 Conclusion and Open Problems **87**

1 Introduction

In the past ten years, lattice-based cryptography gained a lot of interest in the scientific community. The security of commonly applied cryptographic systems is mostly based on the hardness of classical number-theoretic problems, like integer factorization or computing discrete logarithms. Lattice-based cryptosystems can instead be based on the hardness of the shortest vector problem (SVP) or, more exactly, its approximate version (α-SVP). Informally, SVP is the search for shortest non-zero elements in lattices. Its approximate version searches for elements with the size of a shortest vector multiplied by a small factor α (the *approximation factor*). Lattices are discrete, additive groups in the n-dimensional real vector space, represented by a set of basis vectors. There are different types of lattices used in cryptography. Random lattices [GM03] are mostly used to test lattice algorithms, since they are believed to behave the same as structured lattices in the average case. Lattice-based cryptosystems apply so-called q-ary and ideal lattices. The latter ones allow for smaller storage space and faster computations and are therefore the most important type of lattices for cryptographic practice. It is still unclear if their special structure is a drawback for cryptanalysis, but so far there is no algorithm taking advantage of this structure and ideal lattices are believed to be as secure as their regular counterparts.

The shortest vector problem was already mentioned more than a century ago, in works of Hermite [Her50, Her05], Minkowski [Min96], Korkine/Zolotarev [KZ73], and Voronoi [Vor08]. The security of lattice-based cryptographic systems can be solely based on *worst-case* α-SVP, which means that breaking a cryptographic system is proven to be at least as hard as solving *any* instance of α-SVP in a slightly smaller dimension (including worst-case instances). This is one outstanding property

of lattice-based cryptography, since this type of reduction from worst-case problems are not known in any other field of cryptographic research. It implies that there are no weak instances of lattice-based systems.

But how hard is it to solve the shortest vector problem? The problem is proven to be NP-hard under randomized reductions. Even the approximate version is NP-hard for any constant approximation factor α [Ajt98, CN99, Kho05, Kho10, Mic00]. Therefore, we do not expect to find polynomial time algorithms to solve it. So far, there is no algorithm known to solve SVP in sub-exponential time (in the input size or the lattice dimension), even not on potential quantum computers. The fastest deterministic algorithm known is based on Voronoi cell computations, and runs in single-exponential time $2^{\mathcal{O}(n)}$ in the lattice dimension n [MV10a]. So-called *sieving algorithms* are also single-exponential in n. They are probabilistic algorithms, i.e., they fail finding a shortest vector with small probability. In practice, the fastest algorithms are exhaustive search algorithms, that perform *enumeration* of all lattice vectors in a specified search region. Their asymptotic runtime is more than single exponential $2^{\mathcal{O}(n)} \cdot n^{\frac{n}{2e}}$, but in practice they outperform Voronoi cell-based and sieving algorithms. Furthermore, enumeration algorithms only require polynomial storage in n, compared to exponential space requirements for the other two algorithm types. In the SVP challenge, a public competition for comparison of SVP solvers, enumeration algorithms lead the hall of fame, especially the extreme pruning enumeration algorithm [GNR10] excels.

SVP algorithms output a single non-zero lattice vector of smallest possible norm. For cryptographic applications, polynomial approximation factors are more important. The security of lattice-based cryptosystems is based on the hardness of worst-case α-SVP with $\alpha = poly(n)$. Therefore, assessing the runtime of SVP and α-SVP solvers (for polynomial α) is directly related to the security of these systems. Unfortunately, no algorithm is known to solve α-SVP with polynomial approximation factors (besides exact SVP solvers).

For α exponential in the dimension n, there are algorithms that output a complete, *reduced* lattice basis, so-called lattice basis reduction algorithms. These are usually applied as a pre-computation routine before running SVP algorithms. In 1982, the famous LLL algorithm was presented by Lenstra, Lenstra, and Lovász [LLL82]. It computes a reduced basis, where the approximation factor of the smallest basis vector is exponential in the dimension of the lattice. In 1991, the BKZ algorithm, a generalization of LLL, was presented by Schnorr and Euchner [SE94]. Today, BKZ is still the mostly used algorithm for lattice basis reduction. No runtime bound for BKZ is proven, in practice the algorithm appears to run in time polynomial in n. The approximation factor reached by BKZ is again exponential in the dimension. The

Random Sampling Reduction [Sch03] and Simple Sampling Reduction [Lud05, BL06] algorithms apply BKZ in combination with a random method that inserts short vectors into the basis, reaching about the same approximation factor. BKZ is the algorithm most commonly used for practical attacks on lattice-based cryptosystems. It makes use of SVP solvers as a subroutine. by calling a SVP oracles in small dimension. Therefore, knowing the runtime of SVP algorithms is also an important issue for practical attacks. Besides attacks against lattice-based cryptosystems, there are applications of lattice basis reduction in classical (non-lattice) cryptanalysis. Among others, lattice algorithms were used to break knapsack cryptosystems [LO85, Odl89], RSA in special settings [May10], DSA signatures in special settings [HGS01], or solve integer factorization [May10].

Special compute hardware inserts more and more into common hardware like desktop or mobile computers. Graphics cards, originally developed for intense computations in electronic games, can be used to support the CPU for fast parallel computations. Multicore CPUs became standard, and even accessing huge compute clusters and compute clouds is rendered possible even for unexperienced users today. As a result, it appears necessary to take these types of device into account when assessing the strength of cryptosystems. Furthermore, public as well as private networks and infrastructures are threatened by high-end attackers, be it industrial or governmental invaders, spending big amounts of money for special hardware.

Algorithms for SVP and α-SVP have been studied for 30 years, but development of parallel algorithms only started recently. In his master thesis [Puj08], Pujol writes about a parallel version of enumeration using heuristic scheduling (in french language). The implementation [Puj06] offers the opportunity to run enumeration in parallel on multicore CPU systems. The ideas of Pujol were later used in [DHPS10] for an FPGA version of enumeration. In the field of communication theory, enumeration algorithms have been implemented on ASICs, optimized for small dimensions only [GN05, SBB08]. In [MS11] the authors present a parallel version of the GaussSieve algorithm. In small lattice dimensions the speedups scale linear with the dimension. In bigger dimensions however, the scaling factor decreases. Parallel versions of the LLL algorithm are known, e.g. [Vil92, BW09].

Building on existing serial algorithms, in this thesis we go a step further and use special compute hardware in order to speed up the computation of shortest lattice vectors. We develop parallel algorithms and show that it is indeed possible to fully utilize the massive compute power of modern hardware, like graphics cards or supercomputers. With this, we set new records in the SVP challenge and present the fastest public SVP solvers worldwide. We are able to assess the security of lattice-based cryptosystems using our experimental results.

We will use the term of *speedup factor* for practical comparison of parallel and serial algorithms. The speedup factor of a parallel algorithm is computed as the runtime of the serial algorithm divided by the runtime of its parallel competitor. By *scalability* we measure the quality of parallelization of an algorithm. On multicore CPU systems, we say an algorithm scales well if using 10 CPU cores it allows for a speedup factor of 10, in other words, doubling the hardware power leads to a runtime divided in half. Since graphics cards have a fixed number of microprocessors, doubling the hardware amount is only possible by doubling the number of cards.

Summary of Results

Chapter 2 offers required background knowledge for the remainder of the thesis. It introduces the applied notation for lattices, lattice basis reduction, the shortest vector problem, algorithms, and special hardware. Chapters 3 - 7 present the results. They are organized as follows: In a short introduction, the main contribution and achievement of the chapter are described. If necessary, we introduce further notation and basic tools in a first section. Then we develop the parallel version of the respective algorithm followed by a section presenting the experimental study using an implementation of the parallel algorithm. Chapter 8 concludes the results and presents open problems in the research area. Here, we present a short summary of Chapters 3 - 7.

Parallel Enumeration on Multicore CPUs (Chapter 3). This chapter is based on [DS10] presented in Euro-Par 2010. We describe the algorithm design of parallel enumeration on multicore CPUs and explain its implementation. Our experimental study shows a speedup factor that scales nearly linear with the number of used CPU cores (16 cores - 14.4 times as fast) in practice. In some cases we even reach a speedup factor more than linear, due to our algorithm improvement.

Parallel Enumeration on GPU (Chapter 4). This chapter is based on [HSB$^+$10] presented in AFRICACRYPT 2010. The work was motivated by the previous chapter. We describe the algorithm design and the implementation of enumeration on GPUs. Our experimental study shows a factor 5 speedup compared to the fastest public single-core CPU implementation in 2010, using a single 2 years old GPU. In theory, the algorithm scales linearly with the number of graphics cards. Ours is the first SVP or lattice reduction algorithm that uses the massive compute power of special hardware like GPUs.

Extreme Pruning Enumeration on GPU and in Clouds (Chapter 5). This chapter is based on [KSD+11] presented in CHES 2011. It describes the algorithm design of *Extreme Pruning Enumeration* on GPUs, which set the current records in the SVP challenge ($1st - 3rd$ place, dimensions $120, 116, 114$). Our implementation of the algorithm was tested on multiple GPUs as well as on the Amazon EC2 compute cloud, in order to test scalability. Today this is the worlds fastest public implementation for solving SVP. The chapter includes a runtime extrapolation to higher lattice dimensions and a cost prediction of solving SVP challenges in high dimensions in US dollars, using the Amazon EC2 compute cloud. With this we propose a new notion of compute cost, replacing Lenstra's dollarday notation. This work is based on the previous two chapters.

Parallel Random Sampling on GPU (Chapter 6). This chapter is based on the paper [SG11] presented in CHES 2011. We describe the algorithm development of Random Sampling Reduction (RSR) on GPUs. We include the description of a GPU implementation. The search space of RSR can be distributed without communication, which renders the theoretical speedup factors nearly linear in the number of GPUs. Compared to BKZ (the strongest lattice reduction algorithm in practice), the speedup factors are marginal (≈ 2). Compared to a CPU version, the experimental study shows a huge speedup (≈ 20 in high dimension, > 100 in small dimensions $n \lesssim 100$) in practice. We increase the number of samples per second from 5200 to more than $120,000$.

Sieving in Ideal Lattices (Chapter 7). This chapter is based on [Sch11c]. We describe the algorithm development of an extension of the ListSieve and GaussSieve algorithms for ideal lattices. Ideal lattices are the most practical and important type of lattices for cryptography. The inherent special structure of these lattices can be used to fasten the sieving process. The IdealSieve algorithm allows for a speedup factor linear in the degree of the field polynomial, in runtime as well as for storage space. The chapter also describes our CPU implementation, which can be used to find shortest vectors in ideal lattices. We are the first to present an approach how to use the special structure of ideal lattices to speed up SVP algorithms. The experimental study shows practical speed-up factors that are even more than linear in the degree of the field polynomial for the tested dimensions.

2

Notation and Definitions

Vectors and matrices are written in bold face, like $\mathbf{v}$ and $\mathbf{M}$, respectively. The $t \times t$ identity matrix is denoted $\mathbf{I}_t$. A t-dimensional row vector consisting of zero and one entries is denoted $\mathbf{0}_t$ and $\mathbf{1}_t$, respectively. The scalar product of two vector elements $\mathbf{x}$ and $\mathbf{y}$ is written $\langle \mathbf{x} \mid \mathbf{y} \rangle$. The Euclidean norm of a vector $\mathbf{v} \in \mathbb{R}^n$ is denoted $\|\mathbf{v}\|$ or $\|\mathbf{v}\|_2$. Different ℓ_p norms are always subscripted, like $\|\mathbf{v}\|_\infty$. The logarithm of an element x to base 2 is denoted $\log(x)$ or $\log_2(x)$. Other logarithms are subscripted, like $\log_e(x)$ or $\log_{10}(x)$. We define the index set $[t] = \{0, 1, \ldots, t-1\}$. Rounding a value $x \in \mathbb{R}$ to the nearest integer is denoted by $\lfloor x \rceil = \lceil x - 0.5 \rceil$. We use a sans serif font for implementation packages and libraries, like library.

2.1 Lattices

Lattices are discrete additive subgroups of $\mathbb{R}^d$. We define a lattice as follows.

Definition 2.1 (Lattice)**.** Let $n \leq d$ and $\mathbf{B} \in \mathbb{R}^{d \times n}$ be a matrix of linearly independent column vectors $\mathbf{b}_i \in \mathbb{R}^d$. The set

$$\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_n) = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i, x_i \in \mathbb{Z} \right\}$$

is called a lattice.

The matrix $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$ is called a basis of the lattice $\mathcal{L}(\mathbf{B})$ spanned by the column vectors $\mathbf{b}_i$. The number of linear independent basis vectors is called the dimension of the lattice, denoted $\dim(\mathcal{L}(\mathbf{B}))$. One-dimensional lattices have exactly two bases. For $n > 1$ every lattice has infinitely many bases. Switching between

bases can be done by multiplication of a basis matrix with a unimodular matrix. A matrix $\mathbf{M} \in \mathbb{Z}^{n \times n}$ is called *unimodular*, if the determinant $\det(\mathbf{M})$ is ± 1. The unimodular matrices form the multiplicative group $GL_n(\mathbb{Z})$, the general linear group over $\mathbb{Z}$.

Definition 2.2 (Basis Transformation)**.** Let $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{d \times n}$ be a lattice basis. Then for any unimodular matrix $\mathbf{M} \in \mathbb{Z}^{n \times n}$, the matrix $\mathbf{B}' = \mathbf{B}\mathbf{M}$ is a basis of the same lattice, i.e., $\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{B}')$.

We will omit the basis $\mathbf{B}$ and write $\mathcal{L}$ instead of $\mathcal{L}(\mathbf{B})$ if it is clear which basis is concerned. There are characteristics of lattices that are invariant under basis transformation, i.e., the lattice determinant and the successive minima.

Definition 2.3 (Lattice Determinant)**.** The determinant of a lattice $\mathcal{L}(\mathbf{B})$ is defined as

$$\det(\mathcal{L}(\mathbf{B})) = \sqrt{\det(\mathbf{B}^T \mathbf{B})} \,.$$

For full-dimensional lattices, where $n = d$, we have $\det(\mathcal{L}(\mathbf{B})) = |\det(\mathbf{B})|$.

It is easy to see that the determinant of a lattice does not change when the basis is transformed. For $\mathbf{B}' = \mathbf{B}\mathbf{M}$ with $|\det(\mathbf{M})| = 1$ it is $\det(\mathbf{M}^T) = \det(\mathbf{M})$ and with that

$$\sqrt{\det(\mathbf{B}'^T \mathbf{B}')} = \sqrt{\det(\mathbf{M}^T \mathbf{B}^T \mathbf{B} \mathbf{M})} = \sqrt{\det(\mathbf{M}^T) \det(\mathbf{B}^T \mathbf{B}) \det(\mathbf{M})} = \sqrt{\det(\mathbf{B}^T \mathbf{B})} \,.$$

There exists a geometric view of the lattice determinant: it is the volume of the parallelepiped spanned by the basis vectors, i.e., the convex hull of the basis vectors:

$$\det(\mathcal{L}(\mathbf{B})) = \operatorname{vol}\left(\left\{\sum_{i=1}^{n} x_i \mathbf{b}_i : x_i \in \mathbb{R}, 0 \leq x_i \leq 1 \,\forall\, i \in [n]\right\}\right) \,.$$

With other words, the volume of this parallelepiped remains unchanged, even when the basis is transformed. The same holds true for the length of short vectors, i.e., the successive minima.

Definition 2.4 (Successive Minima)**.** The i-th successive minimum $\lambda_i(\mathcal{L}(\mathbf{B}))$ is the minimum radius of a sphere centered at the origin that contains i linear independent vectors in the lattice $\mathcal{L}(\mathbf{B})$. The first minimum $\lambda_1(\mathcal{L}(\mathbf{B}))$ is the length of a shortest, non-zero vector of the lattice.

A shortest lattice vector is never unique, there is always more than one vector of length λ_1 (at least $\mathbf{v}$ and $-\mathbf{v}$, if $\mathbf{v}$ is a shortest one). In practice, λ_1 is not always known. In these cases, it is possible to approximate the length of a shortest vector heuristically, using the Gaussian heuristic. The Gaussian heuristic predicts the number of lattice points inside a given set S to be approximately the volume of the set divided by the volume of the lattice parallelepiped (the lattice determinant).

Heuristic 2.5 (Gauss Heuristic)**.** Given a lattice $\mathcal{L}$ and a set S, the number of points in $S \cap \mathcal{L}$ is approximately $\mathrm{vol}(S)/\det(\mathcal{L})$.

For random lattices, this heuristic can be used to guess the length of a shortest lattice vector. Let S be a n-dimensional sphere of radius r, that is supposed to contain only one lattice vector (i.e., $|S \cap \mathcal{L}| = 1$). The volume of S is $\mathrm{vol}(S) = r^n \cdot \frac{\sqrt{\pi}^n}{\Gamma(n/2+1)}$, where $\Gamma(x)$ is the gamma-function. The Gaussian heuristic predicts $|S \cap \mathcal{L}| = \mathrm{vol}\,(S)/\det\,(\mathcal{L}) = 1$, which leads to the following heuristic approximation of the first minimum of $\mathcal{L}$, which we denote $FM(\mathcal{L})$.

Heuristic 2.6 (First Minimum)**.** The norm of a shortest vector of the n-dimensional lattice $\mathcal{L}$ can be estimated to be

$$\lambda_1(\mathcal{L}) \approx FM(\mathcal{L}) = \frac{\Gamma(n/2+1)^{1/n}}{\sqrt{\pi}} \cdot \det(\mathcal{L})^{1/n} .$$

The Gaussian heuristic has shown to be very accurate in practice for random lattices. It is used, among others, to predict the length of shortest vectors in the SVP challenge [GS10], or to estimate the runtime of enumeration algorithms [HS07, GNR10]. In our experiments as well as in the SVP challenge the heuristic shows to be a good estimate of $\lambda_1(\mathcal{L})$. However, there exist also counterexamples to this heuristic, for example in $\mathbb{Z}^n$ [MO90].

For lattice reduction and SVP algorithms we will need the definition of an orthogonalized basis. The orthogonal projection to a basis $\mathbf{B}$ is defined as $\pi_i : \mathbb{R}^n \to \mathrm{span}_{\mathbb{R}}(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1})^\perp$, such that for all $\mathbf{b} \in \mathbb{R}^d$ it is $\pi_i(\mathbf{b}) \in \mathrm{span}_{\mathbb{R}}(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1})^\perp$ and $\mathbf{b} - \pi_i(\mathbf{b}) \in \mathrm{span}_{\mathbb{R}}(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1})$.

Definition 2.7 (GSO)**.** The Gram-Schmidt orthogonalization (GSO) of a matrix $\mathbf{B} \in \mathbb{R}^{d \times n}$ is $\mathbf{B}^* = [\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*] \in \mathbb{R}^{d \times n}$, computed via

$$\mathbf{b}_i^* = \pi_i(\mathbf{b}_i) = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^* \text{ for } i = 1, \ldots, n\,, \text{ with } \mu_{i,j} = \frac{\mathbf{b}_i^T \mathbf{b}_j^*}{\left\|\mathbf{b}_j^*\right\|^2} \,\forall\, 1 \le j \le i \le n\,.$$

We have $\mathbf{B} = \mathbf{B}^* \mu^T$, where $\mathbf{B}^*$ is orthogonal and μ^T is an upper triangular matrix. Note that $\mathbf{B}^*$ is not necessarily a lattice basis of $\mathcal{L}(\mathbf{B})$. The values μ are called the Gram-Schmidt coefficients. For the diagonal entries, it is $\mu_{i,i} = 1$ for $1 \leq i \leq n$. Further, for the orthogonalized vectors $\mathbf{b}^*$ it is $\det(\mathcal{L}(\mathbf{B})) = \prod_{i=1}^{n} \|\mathbf{b}_i^*\|$.

Definition 2.8 (Projected Lattice)**.** Given a lattice basis $\mathbf{B}$ of the lattice $\mathcal{L}$, the projected lattices $\mathcal{L}_i$ are defined as

$$\mathcal{L}_i = \pi_i(\mathcal{L}) = \mathcal{L}\left(\pi_i(\mathbf{b}_i), \pi_i(\mathbf{b}_{i+1}), \ldots, \pi_i(\mathbf{b}_n)\right) .$$

Most of the algorithms considered in this thesis are deterministic algorithms, i.e., the same input will lead to the same output in multiple runs. When measuring average-case runtime of an algorithm, we mean running it on *random lattices*. The notion of random lattices follows from Haar measures of classical groups [GM03]. Measures allow for a probability distribution from which random lattices can be picked. These lattices are used, among many others, in [NS06, GN08b] to test and compare lattice algorithms. Throughout the remainder of this thesis, when mentioning random lattices or estimating average-case runtime, we are concerned with these Goldstein-Mayer lattices [GM03].

Definition 2.9 (Random Lattice)**.** Let $p \in \mathbb{Z}$ be a fixed constant, and let $\mathbf{q} \in \mathbb{Z}^{n-1}$ be a row vector with entries of size $0 \leq q_i < p$ for $1 \leq i \leq n-1$. Then the column matrix

$$\mathbf{B} = \begin{pmatrix} p & \mathbf{q} \\ \mathbf{0}_{n-1}^T & \mathbf{I}_{n-1} \end{pmatrix} \tag{2.1}$$

forms a basis of a *random lattice*.

There is no standard notion of a random basis of a lattice. There are samplers like Klein's that output soundly sampled vectors [Kle00]. Nonetheless no sound standard of random basis is settled so far. Since SVP algorithms mostly require pre-reduced lattices, it seems sufficient to run LLL or BKZ on the standard basis of random lattices (Equation (2.1)) in order to generate lattice bases that behave randomly.

2.1.1 Ideal Lattices

Ideal lattices are lattices with special structure. Let $\mathbf{f} = x^p + \mathbf{f}_p x^{p-1} + \ldots + \mathbf{f}_1 \in \mathbb{Z}[x]$ be a monic polynomial of degree p, and consider the ring $\mathbf{R} = \mathbb{Z}[x]/\langle \mathbf{f}(x) \rangle$. Elements in $\mathbf{R}$ are polynomials of maximum degree $p-1$. If there exists an ideal $\mathbf{I} \subseteq \mathbf{R}$,

each element $\mathbf{v} = \sum_{i=1}^{p} \mathbf{v}_i x^{i-1} \in \mathbf{I}$ naturally corresponds to its coefficient vector $(\mathbf{v}_1, \ldots, \mathbf{v}_p) \in \mathbb{Z}^p$. Since ideals are additive subgroups, the set of all coefficient vectors corresponding to the ideal $\mathbf{I}$ forms a so-called *ideal lattice*. For the sake of simplicity we can switch between the vector and the ideal notations and use the one that is more suitable in each case.

For each $\mathbf{v} \in \mathbf{R}$, the elements $x^i \cdot \mathbf{v}$ for $i \in [p]$ form a basis of an ideal lattice. We call this multiplication with x a *rotation*, since for special polynomials the vector $x \cdot \mathbf{v}$ consists of the rotated coefficients of $\mathbf{v}$. In vector notation, the multiplication of an element with x can be performed by multiplication with the matrix

$$\mathbf{rot} = \left(\begin{array}{c|c} \mathbf{0}_{p-1} & \\ \mathbf{I}_{p-1} & -\overline{\mathbf{f}} \end{array} \right),$$

where $\overline{\mathbf{f}}$ consists of the coefficients of the field polynomial $\mathbf{f}$. If $\mathbf{f} \in \mathbf{R}$ is a monic, irreducible polynomial of degree p, then for any non-zero element $\mathbf{v} \in \mathbf{R} \setminus \{\mathbf{0}\}$, the elements $\mathbf{v}, \mathbf{v}x, \ldots, \mathbf{v}x^{p-1}$ are linearly independent (see for example the proof of Lemma 2.12 in [Lyu08]). For $\mathbf{f}(x) = x^p - 1$, which is not irreducible over $\mathbb{Z}$, it is easy to see that the vectors $\mathbf{v}x^i$ are also linearly independent, unless the vector has very special form.

The column matrices of the bases used in practice are of the form

$$\left(\begin{array}{cc} q\mathbf{I}_p & (\mathbf{rot}^i(\mathbf{v}))_{i \in [p]} \\ \mathbf{0} & \mathbf{I}_p \end{array} \right). \tag{2.2}$$

Here the right part consists of the p rotations of $\mathbf{v}$, which corresponds to the multiplications of the ring element $\mathbf{v}$ with x^i for $i \in [p]$. The left part is necessary to ensure that every element in the lattice can be reduced modulo q. Bases for higher dimensional lattices can be generated using multiple points $\mathbf{v}_i$ and their rotations. The dimension n of the lattice is then a multiple of the field polynomial degree p.

The usage of ideal lattices reduces the storage amount for a basis matrix from np elements to n elements, because every block of the basis matrix is determined by its first column. In addition, for an ideal basis $\mathbf{B}$, the computation $\mathbf{B} \cdot \mathbf{y}$ can be sped up using Fast Fourier transformation from $\mathcal{O}(np)$ to $\tilde{\mathcal{O}}(n)$.

In this thesis we are concerned with three types of ideal lattices, defined by the choice of $\mathbf{f}$:

- *Cyclic lattices*: Let $\mathbf{f}_1(x) = x^p - 1$, i.e., $\overline{\mathbf{f}} = (-1, 0, \ldots, 0)$. We call the ideal lattices of the ring $\mathbf{R}_1 = \mathbb{Z}[x]/\langle \mathbf{f}_1(x) \rangle$ cyclic lattices. $\mathbf{f}_1(x)$ is never irreducible over $\mathbb{Z}$ ($x - 1$ is always a divisor), therefore cyclic lattices do not guarantee worst-case collision resistance. The rotation of $\mathbf{v}$ is $\mathbf{rot}(\mathbf{v}) = (\mathbf{v}_{p-1}, \mathbf{v}_0, \ldots, \mathbf{v}_{p-2})$.

- *Anti-cyclic lattices*: Let $\mathbf{f}_2(x) = x^p + 1$, i.e., $\overline{\mathbf{f}} = (1, 0, \ldots, 0)$. We call the ideal lattices of the ring $\mathbf{R}_2 = \mathbb{Z}[x]/\langle \mathbf{f}_2(x) \rangle$ anti-cyclic lattices. $\mathbf{f}_2(x)$ is irreducible over $\mathbb{Z}$ if p is a power of 2. The rotation of $\mathbf{v}$ is $\mathbf{rot}(\mathbf{v}) = (-\mathbf{v}_{p-1}, \mathbf{v}_0, \ldots, \mathbf{v}_{p-2})$. Anti-cyclic lattices are the ones used most in cryptography.

- *Prime cyclotomic lattices*: Let $\mathbf{f}_3(x) = x^p + x^{p-1} + \ldots + 1$, i.e., $\overline{\mathbf{f}} = (1, \ldots, 1)$. We call the ideal lattices of the ring $\mathbf{R}_3 = \mathbb{Z}[x]/\langle \mathbf{f}_3(x) \rangle$ prime cyclotomic lattices. $\mathbf{f}_3(x)$ is irreducible over $\mathbb{Z}$ if $p + 1$ is prime. The rotation of $\mathbf{v}$ is $\mathbf{rot}(\mathbf{v}) = (-\mathbf{v}_{p-1}, \mathbf{v}_0 - \mathbf{v}_{p-1}, \ldots, \mathbf{v}_{p-2} - \mathbf{v}_{p-1})$. We only consider cyclotomic polynomials of degree p where $p + 1$ is prime. Other cyclotomic polynomials, where $p + 1$ is not prime, have different structure, the rotations are hard to implement, and they are seldom used in practice.

A nice and more detailed overview about ideal lattices is shown in [Lyu08].

2.2 Lattice Reduction

In lattice dimensions above 120, the exponential runtime of SVP algorithms renders them intractable in practice. Lattice basis reduction (or in short lattice reduction) algorithms of polynomial runtime in n allow to search for short vectors in higher dimensions of up to 1000, at the expense of worse approximation factors. Roughly speaking, lattice basis reduction is the process of transforming a basis of a lattice into a second one consisting of short vectors which are nearly orthogonal. There is no fixed definition of the term lattice reduction itself, there are different notations of a *reduced basis*. Most of them output a short vector of approximation factor exponential in the lattice dimension. Here we present some of the most common notations that will occur in the remainder of the thesis.

Definition 2.10 (Size-reduced Basis)**.** A lattice basis $\mathbf{B}$ is called *size-reduced*, if for all its Gram-Schmidt coefficients it is

$$|\mu_{i,j}| \leq 0.5 \quad \text{for} \quad 1 \leq j < i \leq n\,.$$

Definition 2.11 (δ-LLL-reduced Basis)**.** A lattice basis $\mathbf{B}$ is called *LLL-reduced* with parameter $\delta \in (0.25, 1]$, if it is size-reduced and satisfies

$$\delta \left\|\mathbf{b}_{i-1}^*\right\|^2 \leq \left\|\mathbf{b}_i^*\right\|^2 + \mu_{i,i-1}^2 \left\|\mathbf{b}_{i-1}^*\right\|^2 \quad \text{for} \quad 2 \leq i \leq n\,. \tag{2.3}$$

Condition (2.3) is the so-called Lovász-condition. It implies that for a δ-LLL-reduced basis, the lengths of the Gram-Schmidt orthogonalized vectors does not

decrease too fast. Lenstra, Lenstra, and Lovász presented their algorithm that computes a δ-LLL-reduced basis in polynomial time (for $\delta < 1$) in 1982 [LLL82]. Important sequels are the L^2-algorithm [NS05, NS09] and the recent L^1 algorithm [NSV11].

The following strong definition of lattice reduction follows Hermite and Korkine-Zolotarev [LJS90].

Definition 2.12 (HKZ-reduced Basis). A lattice basis $\mathbf{B}$ is called *HKZ-reduced* if it is size-reduced and satisfies

$$\|\mathbf{b}_i^*\| = \lambda_1(\mathcal{L}_i(\mathbf{B})) \qquad \text{for } i = 1 \dots n\,.$$

Especially it is $\|\mathbf{b}_1\| = \lambda_1(\mathcal{L})$.

Schnorr combined the definition of HKZ and LLL reduced bases and presented the definition of a *Block Korkine-Zolotarev* reduced basis (β-BKZ reduced) [Sch87, SE94].

Definition 2.13 (β-BKZ-reduced Basis). A lattice basis $\mathbf{B}$ is called *BKZ-reduced* with parameter $\beta \in [2, n]$, if it is size-reduced and

$$(\pi_i(\mathbf{b}_i), \pi_i(\mathbf{b}_{i+1}) \dots \pi_i(\mathbf{b}_{i+\beta-1}))$$

is an HKZ-reduced basis for $i = 1 \dots n - \beta + 1$.

If a basis $\mathbf{B}$ is $(\beta + 1)$-BKZ reduced it is also β-BKZ reduced. An LLL-reduced basis is the special case of BKZ reduction for blocksize parameter $\beta = 2$. The BKZ algorithm introduced in [SE94] outputs a BKZ reduced basis. It is the algorithm mainly used in practice for lattice reduction. The BKZ algorithm outputs a basis whose first vector has length $\|\mathbf{b}_1\| \leq (\gamma_\beta)^{(n-1)/(\beta-1)}\lambda_1(\mathcal{L}(\mathbf{B}))$ [Sch94]. Here, γ_β is the Hermite constant of dimension β.

Definition 2.14 (Hermite Constant). The Hermite constant in dimension β is defined as

$$\gamma_\beta = \sup\{\lambda_1(\mathcal{L})^2/(\det(\mathcal{L}))^{2/\beta} : \dim(\mathcal{L}) = \beta\}\,.$$

Values for the Hermite constant are only known for dimensions $1 \leq n \leq 8$ and $n = 24$. The constant is closely related to sphere packings [CE03]. Numerical upper bounds on the constant for other dimensions are given in [CE03].

The LLL [LLL82] and BKZ [SE94] algorithms are the most common algorithms for lattice reduction. BKZ's blocksize parameter β allows for a trade-off between

runtime and reduction quality. Higher values of β lead to better reduced bases at the expense of an exponentially (in β) increasing runtime. Both LLL and BKZ sort the basis vectors in increasing order, so that $\mathbf{b}_1$ is the shortest among the basis vectors after reduction. Applied to a basis $\mathbf{B}$, LLL provably finds a vector $\mathbf{b}_1$ with $\|\mathbf{b}_1\| \leq 2^{(n-1)/2}\lambda_1(\mathcal{L}(\mathbf{B}))$. When LLL or BKZ is applied to a generator system of a lattice $\mathcal{L}$ it outputs a basis of $\mathcal{L}$, so it removes linear dependent vectors. A practical comparison of LLL and BKZ can be found in [GN08b]. Both LLL and BKZ are equipped with a parameter δ, which only slightly controls the reduction quality and is usually set to 0.99.

The *Random Sampling Reduction* (RSR) algorithm [Sch03] uses BKZ as a subroutine and complements an exhaustive search in a specified search space, which differs from the enumeration search region. The latest version of random sampling algorithms is the *Simple Sampling Reduction* (SSR) by Buchmann and Ludwig [BL06]. A more detailed description of RSR and SSR is presented in Chapter 6, where we develop the parallel sampling variant.

There are more notions of reduction and algorithms, that are not used in this thesis. Among others, there is slide reduction [GN08a], segment LLL reduction [KS01], and many more. There are reduction algorithms for different norms, e.g. the enumeration algorithm for arbitrary norms of [Rit97] or the infinity norm enumeration algorithm of [Kai94]. For further information concerning lattices and lattice reduction we refer to [MG02, MR08, NV10]. A practical comparison of LLL and BKZ can be found in [NS06, GN08b]. See [NS01, NV10] for an overview of lattice algorithms in cryptanalysis.

2.3 The Shortest Vector Problem

As mentioned before, lattice reduction algorithms affect the whole basis and are usually used as pre-computation routine before running SVP algorithms. Here we define the shortest vector problem and its approximate versions α-SVP and Hermite-SVP. Furthermore, we introduce the SVP algorithms that we will use in the remainder of this thesis.

Definition 2.15 (Shortest Vector Problem (SVP))**.** Given a lattice basis $\mathbf{B}$, the shortest vector problem asks to find a shortest non-zero vector in $\mathcal{L}(\mathbf{B})$, i.e., a vector $\mathbf{v} \in \mathcal{L}(\mathbf{B}) \setminus \{0\}$ subject to $\|\mathbf{v}\| = \lambda_1(\mathcal{L}(\mathbf{B}))$.

The shortest vector problem can be stated in any norm, among which the Euclidean norm is the most usual. Throughout this thesis, we will only consider the

Euclidean norm. The decisional variant of SVP in the infinity norm [vEB81] is NP-hard. In ℓ_p norms it is only NP-hard under randomized reductions [Ajt98]. Ajtai showed a probabilistic reduction from the NP-hard SubsetSum problem to SVP. For a survey on hardness results on SVP and related problems we defer the reader to [MG02, Kho10, Reg10].

Definition 2.16 (Approximate Shortest Vector Problem (α-SVP))**.** Given a lattice basis $\mathbf{B}$ and a constant $\alpha \geq 1$, find a vector $\mathbf{v} \in \mathcal{L}(\mathbf{B}) \setminus \{0\}$ subject to $\|\mathbf{v}\| \leq \alpha\lambda_1(\mathcal{L}(\mathbf{B}))$.

The approximate SVP is solvable in polynomial time in the lattice dimension for approximation factors α that are of size exponential in the lattice dimension, e.g., by the LLL and BKZ algorithm. For constant factors α, the problem is NP-hard.

In practice, the length of a shortest vector $\lambda_1(\mathcal{L})$ is not always known. Therefore, one can compare short vectors to the lattice determinant. For this purpose, we introduce the Hermite Shortest Vector Problem.

Definition 2.17 (Hermite Shortest Vector Problem (HSVP))**.** Given a lattice basis $\mathbf{B}$ and a constant $c > 0$, find a vector $\mathbf{v} \in \mathcal{L}(\mathbf{B}) \setminus \{0\}$ subject to $\|\mathbf{v}\| = c^n \det(\mathcal{L}(\mathbf{B}))^{1/n}$.

Since the determinant of the lattice is always known, HSVP is useful for comparison of lattices where λ_1 is unknown. The constant c is called the *Hermite factor constant*. Following [GN08b], the LLL algorithm practically outputs a first basis vector $\mathbf{b}_1$ with Hermite factor constant $c = 1.0219$, BKZ-20 reaches a Hermite factor constant $c = 1.0128$, and BKZ-28 reaches a factor $c = 1.0109$.

2.3.1 The SVP Challenge

In 2010, Nicolas Gama and Michael Schneider published a set of random lattices in order to offer a unified testing environment for SVP algorithms: the SVP challenge [GS10]. Since May 2010, more than 85 shortest vectors were entered by scientists from all over the world, using a huge variety of algorithms. Figure 2.1 shows the hall of fame of the SVP challenge in November 2011. The challenge is cited and its lattices are used in [Sch11a, LP11, HPS11a, KSD$^+$11], for example. This shows the impact of the SVP challenge to the scientific community.

2.3.2 Algorithms for SVP

There are mainly three different approaches how to solve the shortest vector problem. First, there are probabilistic sieving algorithms [AKS01, NV08, BN09, AJ08,

HALL OF FAME

Position	Dimension	Euclidean norm	Seed	Contestant
1	120	2851	0	Po-Chun Kuo, Michael Schneider
2	116	2825	0	Po-Chun Kuo, Michael Schneider
3	114	2778	0	Po-Chun Kuo, Michael Schneider
4	112	2715	0	Yuanmi Chen and Phong Nguyen
5	112	2748	0	Po-Chun Kuo

Figure 2.1: Hall of fame of the SVP challenge in November 2011.

MV10b]. They output a solution to SVP with high probability only, but allow for single exponential runtime $2^{\mathcal{O}(n)}$. The most promising sieving candidate for SVP in the Euclidean norm in practice at this time is the heuristic GaussSieve algorithm [MV10b]. The same paper introduces ListSieve, which does not use heuristics and provably runs in time $2^{3.199n}$. Further, there exists an algorithm based on Voronoi cell computation [MV10a]. This is the first *deterministic* SVP algorithm running in single exponential time, but experimental results lack so far. In [DPV11] the authors apply M-Ellipsoid Coverings and make use of the Voronoi cell algorithm in order to enumerate lattice points in any convex body. As one of their applications they use their algorithm to solve exact SVP in any norm, requiring deterministic single exponential time and space $2^{\mathcal{O}(n)}$. Third, there is the group of enumeration algorithms that perform an exhaustive search over all lattice points in a suitable search region. Based on the algorithms by Kannan [Kan83, Kan87] and Fincke/Pohst [FP83, FP85], Schnorr and Euchner presented the ENUM algorithm [SE94]. It was analyzed in more details in [PS08, HS07]. The runtime of Kannan's algorithm [Kan87] is $2^{\mathcal{O}(n)} \cdot n^{\frac{n}{2e}}$. The ENUM of [SE94] requires $2^{\mathcal{O}(n^2)}$ polynomial time operations. The latest improvement to enumeration algorithms called Extreme Pruning Enumeration, providing for huge exponential speedups, was shown by Gama, Nguyen, and Regev [GNR10].

In this section we will introduce the enumeration algorithm of [SE94] and the Extreme Pruning Enumeration of [GNR10]. Furthermore, we will give an overview of sieving algorithms.

Enumeration. Here we give an overview of the ENUM algorithm first presented in [SE94]. The ENUM algorithm enumerates over all linear combinations $[u_1, \dots, u_n] \in \mathbb{Z}^n$ that generate a vector $\mathbf{v} = \sum_{i=1}^n u_i \mathbf{b}_i$ in the search space (i.e., all vectors $\mathbf{v}$ with $\|\mathbf{v}\|$ smaller than a specified bound). Those linear combinations are organized in a tree structure. Leafs of the tree contain full linear combinations, whereas inner nodes

contain partly filled vectors. The search for the tree leaf that determines the shortest lattice vector is performed in a depth first search order. The most important part of the enumeration is cutting off parts of the tree, i.e. the strategy which subtrees are explored and which ones cannot lead to a shorter vector. An algorithm listing is shown as Algorithm 2.1. Let t be the current level in the tree, $t = 1$ being at the bottom and $t = n$ at the top of the tree. Each step in the enumeration algorithm consists of computing an *intermediate* squared norm l_t (Line 4), moving one level up or down the tree (to level $t' \in \{t-1, t+1\}$, Lines 7 and 13) and determining a new value for the coordinate $u_{t'}$.

Algorithm 2.1: Basic Enumeration Algorithm

Input: Gram-Schmidt coefficients $(\mu_{i,j})_{1 \le j \le i \le n}, \|\mathbf{b}_1^*\|^2 \dots \|\mathbf{b}_n^*\|^2$
Output: $\mathbf{u}_{min}$ such that $\|\sum_{i=1}^n u_i \mathbf{b}_i\| = \lambda_1(\mathcal{L}(\mathbf{B}))$

```
1  A ← ‖b*_1‖², u_min ← (1,0,...,0), u ← (1,0,...,0), l ← (0,...,0), c ← (0,...,0)
2  t = 1
3  while t ≤ n do
4      l_t ← l_{t+1} + (u_t + c_t)² ‖b*_t‖²
5      if l_t < A then
6          if t > 1 then
7              t ← t − 1                                  ▷ move one layer down in the tree
8              c_t ← Σ_{i=t+1}^n u_i μ_{i,t},   u_t ← ⌊c_t⌉
9          else
10             A ← l_t,   u_min ← u                       ▷ set new minimum
11         end
12     else
13         t ← t + 1                                      ▷ move one layer up in the tree
14         choose next value for u_t using the zig-zag pattern
15     end
16 end
```

To find a shortest non-zero vector of a lattice $\mathcal{L}(\mathbf{B})$ with $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n]$, ENUM takes as input the Gram-Schmidt coefficients $(\mu_{i,j})_{1 \le j \le i \le n}$, the quadratic norm of the Gram-Schmidt orthogonalization $\|\mathbf{b}_1^*\|^2, \dots, \|\mathbf{b}_n^*\|^2$ of $\mathbf{B}$, and an initial bound A. The search space is the set of all coefficient vectors $\mathbf{u} \in \mathbb{Z}^n$ that satisfy $\|\sum_{t=1}^n u_t \mathbf{b}_t\|^2 \le \mathsf{A}$. Starting with a pre-reduced basis, it is common to set $\mathsf{A} = \|\mathbf{b}_1^*\|^2$ in the beginning. If the norm of a shortest vector is known beforehand, it is possible to start with a lower A, which limits the search space and reduces the runtime of the algorithm. If a vector $\mathbf{v}$ of length smaller than A is found, A can be reduced to

the norm of $\mathbf{v}$, that means A always denotes the size of the current shortest vector.

The goal of ENUM is to find a coefficient vector $\mathbf{u} \in \mathbb{Z}^n$ satisfying the equation

$$\left\| \sum_{t=1}^{n} u_t \mathbf{b}_t \right\|^2 = \min_{x \in \mathbb{Z}^n} \left\| \sum_{t=1}^{n} x_t \mathbf{b}_t \right\|^2 . \tag{2.4}$$

Therefore, all coefficient combinations $\mathbf{u}$ that determine a vector of norm less than A are enumerated. In Equation 2.4 we replace all $\mathbf{b}_t$ by their orthogonalization, i.e., $\mathbf{b}_t = \mathbf{b}_t^* + \sum_{j=1}^{t-1} \mu_{t,j} \mathbf{b}_j^*$ and get Equation (2):

$$\left\| \sum_{t=1}^{n} u_t \mathbf{b}_t \right\|^2 = \left\| \sum_{t=1}^{n} \left(u_t \cdot (\mathbf{b}_t^* + \sum_{j=1}^{t-1} \mu_{t,j} \mathbf{b}_j^*) \right) \right\|^2 = \sum_{t=1}^{n} (u_t + \sum_{i=t+1}^{n} \mu_{i,t} u_i)^2 \cdot \|\mathbf{b}_t^*\|^2 .$$

For index k, enumeration is supposed to check all coefficient vectors $\mathbf{u}$ with

$$\sum_{t=n+1-k}^{n} (u_t + \sum_{i=t+1}^{n} \mu_{i,t} u_i)^2 \cdot \|\mathbf{b}_t^*\|^2 < \mathsf{A} \quad , \quad 1 \leq k \leq n . \tag{2.5}$$

Let $\mathbf{c} \in \mathbb{R}^d$ with $c_t = \sum_{i=t+1}^{n} \mu_{i,t} u_i$ (line 8), which is predefined by all coefficients u_i with $n \geq i > t$. The intermediate norm l_t (line 4) is defined as $l_t = l_{t+1} + (u_t + c_t)^2 \ \|\mathbf{b}_t^*\|^2$. This is the norm part of Equation 2 that is predefined by the values u_i with $n \geq i \geq t$.

The algorithm enumerates the coefficients in reverse order, from u_n to u_1. This can be considered as finding a minimum in a weighted search tree. The height of the tree is uniquely determined by the dimension n. The root of the tree denotes the coefficient u_n. The coefficient values u_t for $1 \leq t \leq n$ determine the values of the vertices of depth $(n-t+1)$, leafs of the tree contain coefficients u_1. The inner nodes represent intermediate nodes, not complete coefficient vectors, i.e., a node on level t determines a subtree $(\perp, \ldots, \perp, u_t, u_{t+1}, \ldots, u_n)$, where the first $t-1$ coefficients are not yet set. l_t is the norm part predefined by this inner node on level t. We only enumerate parts of the tree with $l_t < \mathsf{A}$. Therefore, the possible values for u_t on the next lower level are in an interval around c_t with $(u_t + c_t)^2 < (\mathsf{A} - l_{t+1}) / \|\mathbf{b}_t^*\|$, following the definition of l_t.

ENUM iterates over all possible values for u_t, as long as $l_t \leq \mathsf{A}$, the current minimal value. If l_t exceeds A, enumeration of the corresponding subtree can be cut off, the intermediate norm l_t will only increase when stepping down in the tree, as $l_t \leq l_{t-1}$ always holds. The iteration over all possible coefficient values is (due to Schnorr and Euchner) performed in a zig-zag pattern. The values for u_t will be sequenced like either $c_t, c_t+1, c_t-1, c_t+2, c_t-2, \ldots$ or $c_t, c_t-1, c_t+1, c_t-2, c_t+2, \ldots$.

ENUM starts at the leaf $(1, 0, \ldots, 0)$ and gives the first possible solution for a shortest vector in the given lattice. The algorithm performs its search by moving up (when a subtree can be cut off due to $l_t \geq \mathsf{A}$) and down in the tree (lines 13 and 7). The norm of leaf nodes is compared to A. If $l_1 \leq \mathsf{A}$, it stores $\mathsf{A} \leftarrow l_1$ and $\mathbf{u}_{min} \leftarrow \mathbf{u}$ (line 10), which define the current shortest vector and its size. When ENUM moves up to the root of the search tree it terminates and outputs the computed global minimum A and the corresponding shortest vector $\mathbf{u}_{min}$.

In each step of enumeration, the algorithms performs a polynomial number of operations. Following [HS07, GNR10], the total runtime is N times this polynomial number of operations, where N is the total number of tree nodes. Using the Gaussian heuristic, the number of nodes on each level t can be estimated to be about $H_t = 0.5 \cdot \frac{V_t(\mathsf{A})}{\prod_{i=n+1-t}^{n} \|\mathbf{b}_i^*\|}$, where $V_t(\mathsf{A})$ is the volume of a t-dimensional sphere of radius A. So heuristically, the total runtime of enumeration is $\sum_{t=1}^{n} H_t$ times a polynomial in n. When LLL and the BKZ are used for pre-reduction of the lattice, it aims at increasing the norms of the $\mathbf{b}_i^*$, i.e., increasing the product in the denominator of the fraction H_t. With this, pre-reduction diminishes the size of the enumeration tree and by this speeds up the enumeration process. Bigger blocksize β for BKZ leads to a more reduced basis and speeds up enumeration more, but the BKZ runtime grows exponentially in β, so there is a trade-off between BKZ runtime and enumeration runtime. It is an important issue to find suitable blocksize parameters for pre-reduction.

If ENUM is not supposed to find a shortest vector of the lattice but only a vector below bound A, the algorithm stops as soon as a first vector below the bound was found.

Extreme Pruning Enumeration. Schnorr and Hörner already presented an idea to prune some of the subtrees that are unlikely to contain a shorter vector [SH95]. Their pruned enumeration runs deterministically with a certain probability to miss a shortest vector. The [SH95] pruning idea was analyzed and improved by Gama et al. in 2010 [GNR10]. The authors of [GNR10] also showed some flaws in the analysis of [SH95]. Instead of using the same norm bound A on every layer of the enumeration tree (Equation (2.5)), Gama et al. introduce a bounding vector $(R_1, \ldots, R_n) \in [0, 1]^n$, with $R_1 \leq \ldots \leq R_n$. A on the right side of the testing condition (2.5) is replaced by $R_k \cdot \mathsf{A}$. It can be shown that, assuming various heuristics [GNR10], the lattice vectors cut off by this approach only contain a shortest vector with low probability.

With this pruning technique, an exponential speedup compared to deterministic enumeration can be gained. Gama et al. show that using a randomization technique

it is possible to speed up enumeration even more. The idea of *Extreme* Pruning is to randomly generate many enumeration trees. Instead of spending a lot of time searching *one* tree, one randomly generates *many* trees and only spends a small amount of time on each of them by aggressively pruning the subtrees unlikely to yield short vectors using a bounding function. That is, one focuses on the parts of the trees that are more "fruitful" in terms of the likelihood of producing short vectors per unit time spent. In other words, one should try to maximize the success probability of finding a short vector *per unit of computing time spent* by choosing an appropriate bounding function in pruning. In the original paper, various bounding function vectors were presented in theory. For the experiments, the authors use a numerically optimized function.

Sieving. Sieving algorithms were first presented in 2001 by Ajtai, Kumar, and Sivakumar [AKS01]. The runtime and space requirements were proven to be in $2^{\mathcal{O}(n)}$. Nguyen and Vidick [NV08] carefully analyzed this algorithm and presented the first competitive timings and results. They show that the runtime of AKS sieve is $2^{5.90n+o(n)}$ and the space required is $2^{2.95n+o(n)}$. The authors also presented a heuristic variant of AKS sieve without perturbations. Their runtime is $2^{0.41n+o(n)}$ and they require space $2^{0.21n+o(n)}$. In 2010, Micciancio and Voulgaris [MV10b] presented a provable sieving variant called ListSieve and a more practical, heuristic variant called GaussSieve. ListSieve (as well as the algorithms of [AKS01, NV08]) samples perturbed points with a small error instead of lattice points. This allows to prove the generation of non-zero vectors, which is necessary for the runtime proof. ListSieve runs in time $2^{3.199n+o(n)}$ and requires space $2^{1.325n+o(n)}$. For GaussSieve, the maximum list size can be bounded by the kissing number τ_n, whereas, due to collisions, a runtime bound can not be proven. The practical runtime is $2^{0.52n}$, the space requirements is expected to be less than $2^{0.21n}$ and turns out to be even smaller in practice. ListSieveBirthday by Pujol and Stehlé [PS09] uses multiple lists and improves the theoretical bounds of ListSieve [MV10b] using the birthday paradox to runtime $2^{2.465n+o(n)}$ and space $2^{1.233n+o(n)}$. The proof is completed in [HPS11a]. Wang et at. [WLTB10] present a heuristic variant of the Nguyen-Vidick sieve running in $2^{0.3836n+o(n)}$ with space complexity of $2^{0.2557n}$. The work of [BN09] deals with all ℓ_p norms, generalizing the AKS sieve. There is only one public implementation of a sieving algorithm, namely `gsieve` [Vou10], which implements the GaussSieve algorithm of [MV10b].

A more detailed explanation of ListSieve including some pseudo-code is presented in Chapter 7.

2.3.3 Public Implementations

There are public implementations of lattice reduction and SVP algorithms. Some of them, like the LLL and BKZ implementations, will be used throughout the remainder of this thesis, as pre-reduction routines. Some SVP implementations will be used in a comparison with our improved implementations.

- The NTL library of Victor Shoup [Sho] offers implementations of LLL and BKZ using different floating point precision. We will use NTLs LLL and BKZ for pre-reduction of lattices.
- The fpLLL library [CPS] offers an implementation of L^2, as explained in [NS05, NS09]. As an aside, using the switch $-a\ svp$ it allows to run enumeration to solve the shortest vector problem. We will use the enumeration of fpLLL for comparison with our SVP algorithms.
- The gsieve implementation of Panagiotis Voulgaris [Vou10] runs GaussSieve, as explained in [MV10b]. We extended the code of gsieve for sieving in ideal lattices, cf. Chapter 7.
- SSR of Ludwig [Lud05], segment-LLL and primal-dual reduction [Fil02] are available on request, cf. [BLR08].
- SHVEC 1.0 [Val06] from 1999 computes shortest and closest vectors in lattices, using the algorithm by Fincke-Pohst.
- latenum [Puj06] is a library to solve the shortest and the closest vector problem on integer lattices, using floating point arithmetic. It includes a parallel version of enumeration. Besides the parallel enumeration, latenum was integrated into fpLLL.

Most of our implementations are also available online, in order to offer the possibility to reproduce the experiments shown in this thesis. Our public implementations include

- gpuenum [HKS11] implemented the parallel GPU version of enumeration, as explained in Chapter 4. It was later extended by extreme pruning enumeration, as shown in Chapter 5.
- gpussr [GS11] of Chapter 6 contains a CPU version of SSR as well as our GPU implementation.

- idealsieve [Sch11b] extends gsieve and offers faster SVP solving in ideal lattices, as explained in Chapter 7.
- our generator for cryptographic lattices sage.crypto.lattice.gen_lattice has been included into sage 4.5.2 and above [S+]. It produces modular, random, ideal, and cyclotomic lattice bases and their scaled duals in sage and NTL readable format.

For the LLL and BKZ algorithm, the floating point precision used in the implementations plays a major role [SE94, NS09]. The NTL library offers a couple of different versions of LLL and BKZ, concerning floating point precision, while fpLLL offers to specify the precision as parameter. For enumeration, the authors of [PS08] prove that enumeration using double precision values should be possible up to lattice dimension at least 90. Our experience shows that even in dimension 120, precision errors do not occur.

This concludes the introductory part concerning lattices and related topics. For more information on lattices, hard lattice problems, lattice reduction, and SVP we refer to the surveys of [MG02, MR08, NV10]. A recent survey of SVP algorithms can be found in [HPS11a].

2.4 Parallelization and Special Hardware

2.4.1 Graphics Cards

A Graphical Processing Units (GPUs) is a piece of hardware that is specifically designed to perform a massive number of specific graphical operations in parallel. It is used as a coprocessor of the host processor unit. The introduction of platforms like CUDA by NVIDIA [NVI07a, KH10] or CTM by ATI [AMD06], that make it easier to run custom programs instead of limited graphical operations on a GPU, has been the major breakthrough for the GPU as a general computing platform. The introduction of integer and bit arithmetic also broadened the scope to cryptographic applications. GPUs follow the SPMD (single program, multiple data) programming model, where grids of GPU threads run the same program (the kernel), dedicated to perform massively data-parallel computations.

Applications. Many general mathematical packages are available for GPU, like the BLAS library [NVI07b] that supports basic linear algebra operations.

An obvious application in the area of cryptography is brute force searching using multiple parallel threads on the GPU. There are also implementations of AES

[CIKL05, Man07, HW07] and RSA [MPS07, SG08, Fle07] available as well as implementations of the SHA3 hash competition finalists [BS10]. GPU implementations can also be used (partially) in cryptanalysis. In 2008, Bernstein et al. use parallelization techniques on graphics cards to solve integer factorization using elliptic curves [BCC+09]. Using NVIDIA's CUDA parallelization framework, they gained a speed-up of up to 6 compared to computation on a four core CPU. However, to date, no applications based on lattices are available for GPU.

Programming Model. For the work in this paper the CUDA platform will be used. The GPUs from the Tesla range, which support CUDA, are composed of several multiprocessors, each containing a small number of scalar processors. For the programmer this underlying hardware model is hidden by the concept of SIMT-programming: Single Instruction, Multiple Thread. The basic idea is that the code for a single thread is written, which is then uploaded to the device and executed in parallel by multiple threads.

The threads are organized in multidimensional arrays, called blocks. All blocks are again put in a multidimensional array, called the *grid*. When executing a program (a grid), threads are scheduled in groups of 32 threads, called *warps*. Within a warp threads should not diverge, as otherwise the execution of the warp is serialized.

Memory Model. The Tesla GPUs provide multiple levels of memory: registers, shared memory, global memory, texture and constant memory. Registers and shared memory are on chip and close to the multiprocessor and can be accessed with low latency. The number of registers and shared memory is limited, since the number available for one multiprocessor must be shared among all threads in a single block.

Global memory is off-chip and is not cached. As such, access to global memory can slow down the computations drastically, so several strategies for speeding up memory access should be considered (besides the general strategy of avoiding global memory access). By coalescing memory access, e.g. loading the same memory address or a consecutive block of memory from multiple threads, the delay is reduced, since a coalesced memory access has the same cost as a single random memory access. By launching a large number of blocks the latency introduced by memory loading can also be hidden, since other blocks can be scheduled in the meantime. The constant and texture memory are cached and can be used for specific types of data or special access patterns.

Instruction Set. Modern GPUs provide the full range of (32 and) 64 bit floating point, integer and bit operations. Addition and multiplication are fast, other op-

erations can, depending on the type, be much slower. There is no point in using other than 32 or 64 bit numbers, since smaller types are always cast to larger types. Most GPUs have a specialized FMAD instruction, which performs a floating point multiplication followed by an addition at the cost of only a single operation. This instruction can be used during the BKZ enumeration.

One problem that occurs on GPUs is the fact that today GPUs are not able to deal with higher precision than 64 bit floating point numbers. For lattice reduction, sometimes higher bit sizes are required to guarantee the correct termination of the algorithms. For an n-dimensional lattice, using the floating point LLL algorithm of [LLL82], one requires a precision of $\mathcal{O}(n \log B)$ bits, where B is an upper bound for the length of the d-dimensional vectors [NS05]. For the L^2 algorithm of [NS05], the required bit size is $\mathcal{O}(n \log_2(3))$, which is independent of the norm of the input basis vectors. For more details on the floating point LLL analysis see [NS05] and [NS06].

Degree of Parallelization. The goal of parallelization on graphics cards is to occupy all microprocessors of the GPU as much as possible. Since GPUs work in SIMD mode, branching is one of the main drawbacks of algorithms for GPU implementations. So-called diverging branches leads to a loss in total speedup, since some warps are idle while others compute their branch. When speaking about *linear speedup* on GPUs we consider the use of multiple cards, i.e., if considering twice the number of cards leads to half the runtime.

2.4.2 Multicore CPUs

There exist many parallel environments to perform operations concurrently. Basically, on today's machines, one distinguishes between shared memory and distributed memory passing. A *multi-core microprocessor* follows the shared memory paradigm in which each processor core accesses the same memory space. Nowadays, such computer systems are commonly available. They possess several cores, while each core acts as an independent processor unit. The operating system is responsible to deliver operations to the cores. There exist multiple parallelization libraries for most programming languages, like Boost or MPI for C++.

Parallel algorithms such as graph search algorithms may benefit from communication, in such a way that fewer operations need to be computed. As soon as the number of saved operations exceeds the communication overhead, an efficiency of more than 1.0 might be achieved. For instance, branch-and-bound algorithms for Integer Linear Programming might have superlinear speedup, due to the interdependency between the search order and the condition which enables the algorithm

to disregard a subtree. The enumeration algorithm falls into this category as well.

When dealing with multicore CPUs, our goal is to occupy all CPU cores as much as possible. We measure this by speedup factor, i.e., the desired outcome is always a linear speedup factor in the number of cores. A second question is the scalability of the parallelization. We ask if the speedup factors gained with small number of CPU cores still hold when we increase their number, and to which extent.

3 Parallel Enumeration on Multicore CPUs

In this chapter we describe our first parallel algorithm for enumeration of a shortest lattice vector and its multicore CPU implementation. The algorithm is a parallel version of the enumeration algorithm presented in Chapter 2.3.2. It performs full enumeration, i.e., processes the enumeration tree completely without pruning or early termination, and outputs a vector of length $\lambda_1(\mathcal{L})$. We are aiming at linear speedup in the number of CPU cores. First, we explain our main idea of parallel enumeration and the motivation for our strategy. Second, we present a detailed description. Algorithms 3.1 and 3.2 depict our parallel enumeration algorithm. Third, we show some improvements that speed up the parallelization in practice. Finally, we present an experimental study that shows the strength of our parallel enumeration algorithm in concerns of scalability (and also pure runtime). We will use the notation presented in the previous chapter.

A preliminary version of this chapter was published in Euro-Par 2010 [DS10]. The dissertation author was one of the principal investigators and authors of this paper.

3.1 Parallel Algorithm Design and Implementation

3.1.1 Parallel Lattice Enumeration - High Level Description

Our main idea for parallelization of the enumeration algorithm is the following. Different subtrees of the complete search tree are enumerated in parallel independently from each other representing them as threads (Sub-ENUM threads). Using

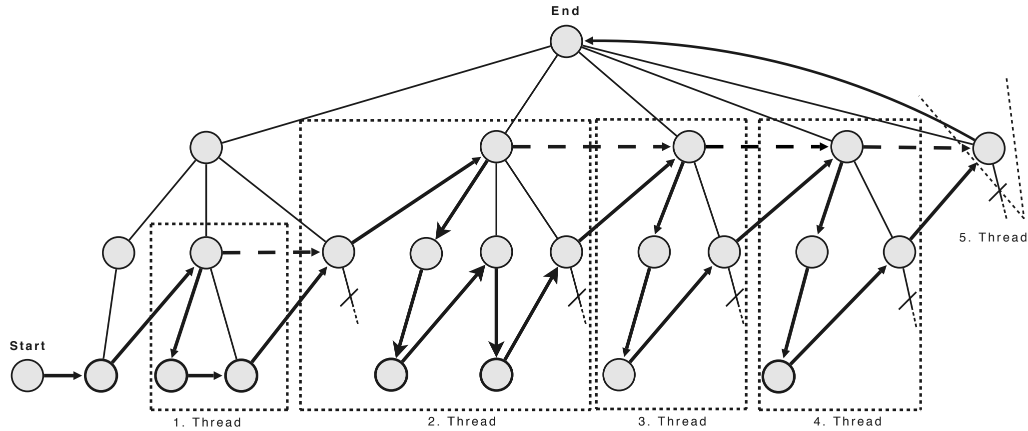


Figure 3.1: Comparison of serial (solid line) and parallel (dashed line) processing of the search tree.

numcores processors, numcores subtrees can be enumerated at the same time. All threads ready for enumeration are stored in a thread list L, and each CPU core that has finished enumerating a subtree picks the next subtree from the list. Each of the subtrees is an instance of SVP in smaller dimension; the initial state of the sub-enumeration can be represented by a tuple $(\mathbf{u}, l, c, t)$ (cf. Chapter 2.3.2).

The main challenge is selecting suitable subtrees of the enumeration tree. A naive way of selection would be to select numerous subtrees by enumerating the top of the tree using breadth first search strategy. This (serial) approach of selection would lead to a huge number of threads, which can be processed in parallel. The main disadvantage of this approach is the serial enumeration of the top tree, which avoids perfect parallelization.

Our selection strategy is different. We stick to the original depth first search strategy and enumerate the tree in usual manner. As soon as the algorithm steps one level up in the tree we can generate new subtrees for enumeration. We generate new enumeration instances "on the fly", which minimizes the serial part of the algorithm. More exactly, when the ENUM algorithm increases the level in the search tree, the center (c_t) and the range $(\mathsf{A} - l_{t+1}) / \|\mathbf{b}_t^*\|$ of possible values for the current coefficient value are calculated. Therefore, it is possible to open one thread for every value in this range. Depending on the size of the interval of possible values, this number of threads is sufficient to fully occupy all CPU cores at hand.

Figure 3.1 shows a 3-dimensional example and compares the flow of the serial ENUM with our parallel version. Beginning at the starting node the procession order of the serial ENUM algorithm follows the directed solid edges to the root.

In the parallel version dashed edges represent the preparation of new Sub-ENUM threads which can be executed by a free processor unit. Crossed-out edges point out irrelevant subtrees. Threads terminate as soon as they reach either a node of another thread or the root node. Chapter 3.1.2 presents our detailed algorithm for parallel enumeration.

Extra Communication - Updating the Shortest Vector. Like in the previous chapter, we denote the current, global minimum, as A. In our parallel version, it is the global minimum of all threads. As soon as a thread has found a new minimum, the Euclidean norm of this vector is written back to the shared memory, i.e. A is updated. At a certain point every thread checks the global minimum whether another thread has updated A and, if so, uses the updated one. The smaller A is, the faster a thread terminates, because subtrees that exceed the current minimum can be cut off in the enumeration. The memory access for this update operation is minimal, only one integer value has to be written back or read from shared memory. This is the only type of communication among threads, all other computations can be performed independently without communication overhead.

3.1.2 The Algorithm for Parallel Enumeration

Algorithm 3.1 shows the main thread for the parallel enumeration. It is responsible to initialize the first Sub-ENUM thread and manage the thread list L. A Sub-ENUM thread (SET) is represented by the tuple $(\mathbf{u}, l, c, t)$, where $\mathbf{u}$ is the coefficient vector, l the intermediate norm of the root to this subtree, c the search region center and t the lattice dimension minus the starting depth of the parent node in the search tree.

Whenever the list contains a SET and free processor units exist, the first SET of the list is executed. The execution of SETs is performed by Algorithm 3.2. We process the search tree in the same manner as the serial algorithm (Algorithm 2.1), except the introduction of the loop bound *bound* and the handling of new SETs (lines $9 - 11$). First, the loop bound controls the termination of the subtree and prohibits that nodes are visited twice. Second, only the SET whose bound is set to the lattice dimension is allowed to create new SETs. Otherwise, if we allow each SET to create new SETs by itself, this would lead to an explosion of the number of threads and each thread would have too few computations to perform. We denote the SET with bound set to n by *unbounded SET* (USET). At any time, there exists only one USET that might be stored in the thread list L.

As soon as a USET has the chance to find a new minimum within the current subtree (lines $5-6$), its bound is set to the current t value. Thereby, it is transformed

Algorithm 3.1: Main thread for parallel enumeration

Input: Gram-Schmidt coefficients $(\mu_{i,j})_{1\leq j\leq i\leq n}$, $\|\mathbf{b}_1^*\|^2 \ldots \|\mathbf{b}_n^*\|^2$
Output: $\mathbf{u}_{min}$ such that $\|\sum_{i=1}^{n} u_i \mathbf{b}_i\| = \lambda_1(\mathcal{L}(\mathbf{B}))$

```
1  A ← ||b*_1||², u_min ← (1, 0, ..., 0)                ▷ Global variables
2  u ← (1, 0, ..., 0), l ← 0, c ← 0, t ← 1             ▷ Local variables
3  L ← {(u, l, c, t)}                                   ▷ Initialize list
4  while L ≠ ∅ or threads are running do
5      if L ≠ ∅ and cores available then
6          pick Δ = (u, l, c, t) from L
7          start Sub-ENUM thread Δ = (u, l, c, t) on new core
8      end
9  end
```

Algorithm 3.2: Sub-ENUM thread (SET)

Input: Gram-Schmidt coefficients $(\mu_{i,j})_{1\leq j\leq i\leq n}$, $\|\mathbf{b}_1^*\|^2 \ldots \|\mathbf{b}_n^*\|^2$, $(\bar{\mathbf{u}}, \bar{l}, \bar{c}, \bar{t})$

```
1  u ← ū, l ← (0, ..., 0), c ← (0, ..., 0)
2  t ← t̄, l_{t+1} ← l̄, c_t ← c̄, bound ← n
3  while t ≤ bound do
4      l_t ← l_{t+1} + (u_t + c_t)² ||b*_t||²
5      if l_t < A then
6          if t > 1 then
7              t ← t − 1                                ▷ move one layer down in the tree
8              c_t ← Σ_{i=t+1}^{n} u_i μ_{i,t},   u_t ← ⌊c_t⌉
9              if bound = n then
10                 L ← L ∪ (u, l_{t+2}, c_{t+1}, t + 1)  ▷ insert new SET in list L
11                 bound ← t
12             end
13         else
14             A ← l_t,   u_min ← u                     ▷ set new global minimum
15         end
16     else
17         t ← t + 1                                    ▷ move one layer up in the tree
18         choose next value for u_t using the zig-zag pattern
19     end
20 end
```

to a SET and the recent created SET becomes the USET.

3.1.3 Improvements

We presented a first solution for the parallelization of the ENUM algorithm providing a runtime speedup by a divide and conquer technique. We distribute subtrees to several processor units to search for the minimum. Our improvements deal with the creation of SETs and result in significantly shorter running time. By now we call a node, where a new SET can be created, a candidate. Note that a candidate can only be found in a USET.

The following paragraphs present possible worst case situations for the presented parallel ENUM algorithm and present possible solutions to overcome the existing drawbacks. The parallelization approach shown so far is also suitable for different architectures. The following improvements however showed good speedups for our multicore CPU implementation and might be disadvantageous on different hardware. Therefore, a new examination should be carried out for different platforms.

Threads within threads. So far only the unbounded USET is allowed to create new sub threads. If a USET creates a new SET at a node of depth 1, then this new SET enumerates a subtree of height $n-1$ sequentially on one processor core. In this case, where the depth of a node is sufficiently far away from the depth t of the starting node, the creation of a new SET is advantageous considering the number of simultaneously occupied processors. Therefore, we introduce a bound s_{deep} which expresses what we consider to be sufficient far away, i.e. if a SET visits a node with depth k fulfilling the equation $k - t \geq s_{deep}$ where t stands for the depth of the starting node and it is not a USET, then this SET is permitted to create a new SET once.

Thread Bound. We achieve additional performance improvements by the following idea. Instead of generating SETs in each possible candidate, we consider the depth of the node. This enables us to avoid big subtrees for new SETs by introducing an upper bound s_{up} representing the minimum distance of a node to the root to become a candidate. If ENUM visits a node with depth t fulfilling $n - t > s_{up}$ we do not generate a new SET. Instead we advance like the serial ENUM algorithm. Good choices for the above bounds s_{deep} and s_{up} are evaluated in the next section.

3.2 Experimental Results

We performed numerous experiments to test our parallel enumeration algorithm. We created 5 different random lattices of each dimension $n \in \{42, \ldots, 56\}$ in the sense of Goldstein and Mayer [GM03]. The bit size of the entries of the basis matrices were in the order of magnitude of $10n$. We started with bases in Hermite normal form and LLL-reduced the bases (using LLL parameter $\delta = 0.99$). The experiments were performed on a compute server equipped with four AMD Opteron (2.3GHz) quad core processors. We compare our results to the highly optimized, serial version of fpLLL in version 3.0.12, the fastest ENUM implementation known, on the same platform. The programs were compiled using gcc version 4.3.2. For handling parallel processes, we used the Boost-Thread-sublibrary in version 1.40. Our C++ implementation uses double precision to store the Gram-Schmidt coefficients $\mu_{i,j}$ and the $\|\mathbf{b}_1^*\|^2, \ldots, \|\mathbf{b}_n^*\|^2$. Due to [PS08], this is suitable up to dimension 90, which seems to be out of the range of pure enumeration algorithms.

We tested the parallel ENUM algorithm for several s_{deep} values and concluded that $s_{deep} = \frac{25}{36}(n - t)$ is a good choice, where t is the depth of the starting node in a SET instance. The exact selection of this parameter only slightly influences the total runtime of enumeration. The same holds for s_{up}, which was set to $s_{up} = \frac{5}{6}n$ in our experiments. Clearly s_{up} should be set close to the root on level n, but small enough to guarantee sufficiently small size of the selected subtrees.

Table 3.1 and Figure 3.2 present the experimental results that compare our parallel version to our serial algorithm and to the fpLLL library. We only present the timings, as the output of the algorithms is in all cases the same, namely a shortest non-zero vector of the input lattice.

The corresponding speedup factors are shown in Figure 3.4. To show the strength of parallelization of the lattice enumeration, we first compare our multicore versions to our single-core version. For 4 and 8 cores the speedup is exactly linear (4 and 8 times as fast). The best speedup factors are 4.5 ($n = 50$) for 4 cores, 8.6 ($n = 50$) for

n	42	44	46	48	50	52	54	56
1 core	3.81	27.7	37.6	241	484	3974	10900	223679 (62h)
4 cores	0.99	7.2	8.8	55	107	976	2727	56947 (16h)
8 cores	0.62	4.0	4.8	28	56	504	1390	28813 (8h)
16 cores	0.52	2.6	3.5	18	36	280	794	16583 (5h)
fpLLL 1 core	3.32	23.7	29.7	184	367	3274	9116	184730 (51h)

Table 3.1: Average time in seconds for enumeration of lattices in dimension n.

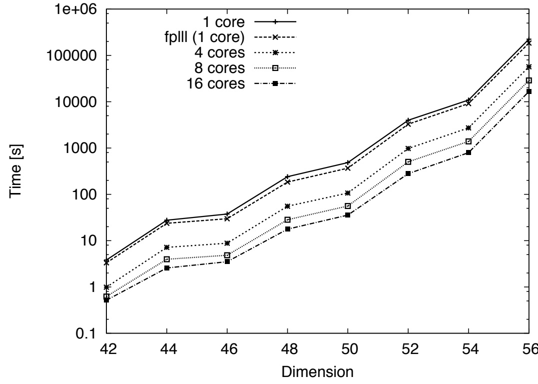


Figure 3.2: Average runtimes of enumeration of 5 random lattices in each dimension, comparing our multicore implementation to fpLLL's and our own single-core version.

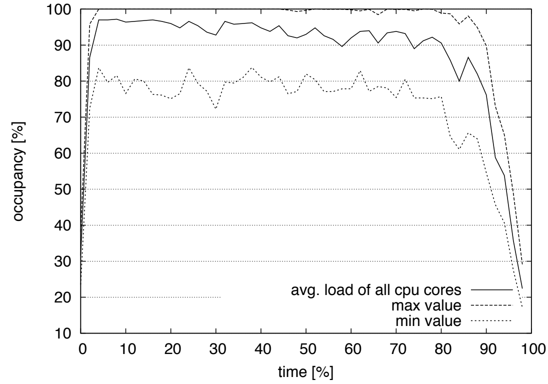


Figure 3.3: Occupancy of the cores. The x-axis marks the percentage of the complete runtime, the y-axis shows the average occupancy of all CPU cores over 5 lattices.

8 cores, and 14.2 ($n = 52$) for 16 cores. This shows that, using numcores processor cores, we sometimes gain speedup factors of more than numcores, which corresponds to an efficiency of more than 1. The efficiency computes as the speedup factor divided by the number of used processors. An efficiency of 1.0 means that numcores processors lead to a speed-up factor of numcores which can be seen as a "perfect" parallelization. This is a very untypical behavior for (standard) parallel algorithms, but understandable for graph search algorithms like our lattice enumeration. It is caused by the extra communication for the write-back of the current minimum A.

The highly optimized enumeration of fpLLL is around 10% faster than our serial version. Compared to the fpLLL algorithm, we gain a speedup factor of up to 6.6 ($n = 48$) using 8 CPU cores and up to 11.7 ($n = 52$) using 16 cores. This corresponds to an efficiency of 0.825 (8 cores) and 0.73 (16 cores), respectively.

Figure 3.3 shows the average, the maximum, and the minimum occupancy of all CPU cores during the runtime of 5 lattices in dimension $n = 52$. The average occupancy of more than 90% points out that all cores are nearly optimally loaded; even the minimum load values are around 80%. These facts show a good balanced behaviour of our parallel algorithm.

On a computer equipped with 24 CPU cores, we ran a second series of our experiments, in order to show the scalability on more than 16 cores. The computer contains four AMD Opteron 8435 processors, each containing 6 cores running at 2.6 GHz. Table 3.2 shows the runtime and Figure 3.5 shows the corresponding speedups. The results on this machine are comparable to the results seen before. The maximum speedups using 24 cores are 23.4 in dimension 50 and 23.2 in dimension 56.

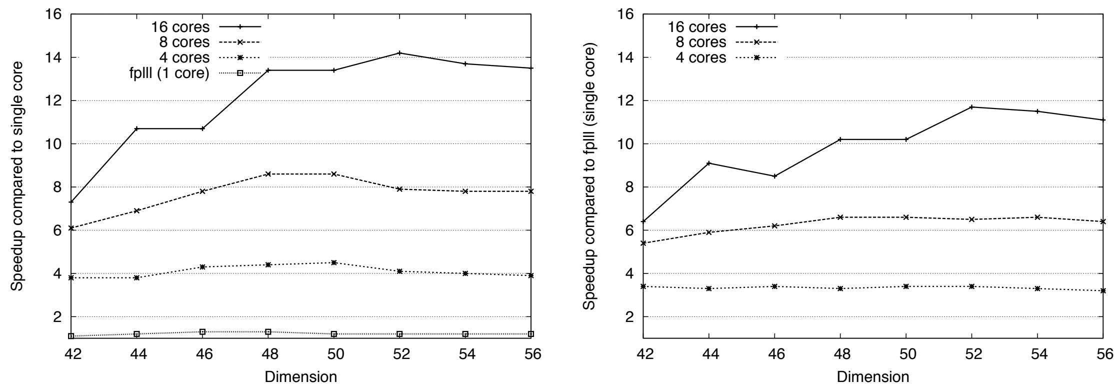


Figure 3.4: Average speedup factor of parallel ENUM compared to our single-core version (left) and compared to `fpLLL`'s single-core version (right).

n	42	44	46	48	50	52	54	56
1 core	3.26	24.2	29.7	194	250	3361	9428	200491 (55.7h)
4 cores	0.83	6.1	7.5	48	53	842	2365	49222 (13.7h)
8 cores	0.43	3.1	3.8	24	27	417	1179	24734 (6.9h)
16 cores	0.33	1.7	2.2	13	14	206	591	12470 (3.5h)
24 cores	0.31	1.4	1.9	10	11	149	423	8635 (2.4h)
`fpLLL` 1 core	2.95	20.9	26.0	161	323	2888	8046	163580 (45.4h)

Table 3.2: Average time in seconds for enumeration of lattices in dimension n, on a 24-core AMD Opteron machine.

Further Comments. The experiments in this chapter use lattices in dimension less than 60. Using pruning strategies, it would be possible to extend the experiments to higher dimensions of more than 100. Our parallelization technique is also applicable for pruned enumeration, and we expect the same linear speedups in higher dimensions with pruning. Pruning leads to thinner trees, but since the number of possible parameter selections on each tree level increases with the lattice dimension, we expect that it is still possible to generate sufficient many subtrees on each level of the tree.

Concerning scalability, we expect the linear speedup to hold even for huge numbers of CPU cores. In fixed dimension, the number of subtrees that are processed in parallel is upper bounded. For the tested dimensions however, this bound was large (> 10.000 subtrees were queued in our experiments). When the lattice dimension increases, the number of existing subtrees increases as well. Therefore we expect the linear speedup to scale even for large numbers of CPU cores.

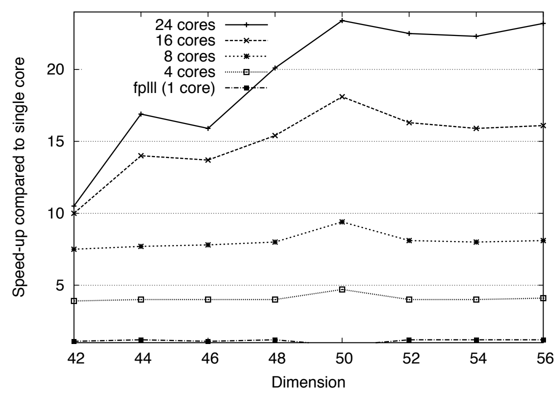


Figure 3.5: Average speedup factor of parallel ENUM compared to our single-core version, on a 24-core machine.

4

Parallel Enumeration on GPU

In this chapter we present our parallel algorithm for shortest vector enumeration in lattices using graphics cards. Our goal is to develop an algorithm that occupies a single GPU as much as possible. Using multiple graphics cards, the speedup factor is desired to be linear in the number of cards.

We present the basic idea for multi-thread enumeration in Section 4.1.1 and we explain our parallel algorithm in detail in Section 4.1.2. Again, we aim at full enumeration, i.e., we solve the exact shortest vector problem. On graphics cards, parallel threads have to be started at the same time. Applying the parallel algorithm of Chapter 3 would lead to a massive number of idle threads on a GPU. Therefore, we use a different approach in this chapter, by enumerating the top of the tree in serial manner before applying the parallel hardware acceleration in subtrees. Section 4.2 presents an experimental study using our GPU implementation. Again we will use the notation introduced in Chapter 2.

A preliminary version of this chapter was published in AFRICACRYPT 2010 [HSB+10]. The dissertation author was one of the principal investigators and authors of this paper.

4.1 Parallel Algorithm Design and Implementation

4.1.1 Multi-Thread Enumeration

In comparison to the parallel CPU enumeration (cf. Chapter 3), we will first use serial enumeration on top of the tree in our GPU version. The SIMD nature of graphics cards requires to perform the same operation on each microprocessor at

the same time. On CPU, we generate subtrees "on the fly" during enumeration and start threads at any time. This would lead to a huge number of diverging threads and idle processor units on the GPU. Therefore, we enumerate the top of the tree first on CPU, in order to have enough *start vectors* at hand to start parallel enumeration on GPU. Since the size of these subtrees differ a lot we use an iterated enumeration process, which switches between top enumeration on CPU and subtree enumeration on GPU iteratively.

The search tree of combinations that is explored in the enumeration algorithm can be split at a high level of the tree, distributing subtrees among several threads. Each thread then runs an enumeration algorithm, keeping the first coefficients fixed. These fixed coefficients are called *start vectors*. The subtree enumerations can run independently, which limits communication between threads. The top level enumeration is performed on CPU and outputs start vectors for the GPU threads. When the number of postponed subtrees is higher than the number of threads that we can start in parallel, then we copy the start vectors to the GPU and let it enumerate the subtrees. After all threads have finished enumerating their subtrees we proceed in the same manner: caching start vectors on CPU and starting a batch of subtree enumerations on GPU. Figure 4.1 illustrates this approach. The variable α defines the region where the initial enumeration is performed. The subtrees where GPU threads work are also depicted in Figure 4.1.

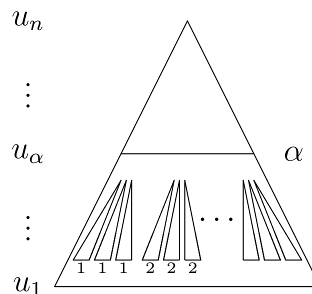


Figure 4.1: Illustration of the algorithm flow. The top part is enumerated on CPU, the lower subtrees are explored in parallel on GPU. The tiny numbers illustrate which subtrees are enumerated in the same iteration.

If a GPU subtree enumeration finds a new optimal vector, it writes back the coordinates $\mathbf{u}$ and the squared norm A of this vector to the main memory. The other GPU threads will directly receive the new value for A, which will allow them to cut away more parts of the subtree.

Early Termination. The computation power of the GPU is used best when as many threads as possible are working at the same time. Recall that the GPU uses warps

as the basic execution units: all threads in a warp are running the same instructions (or some of the threads in the warp are stalled in the case of branching).

In general, more starting vectors than there are GPU threads are uploaded in each run of the GPU kernel. This allows us to do some load balancing on the GPU, to make sure all threads are busy. To avoid the GPU being stalled by a few long running subtree enumerations, the GPU stops when just a few subtrees are left. We call this process, by which the GPU stops some subtrees even though they are not finished, *early termination*.

At the end of Section 4.1.2 details are included on the exact way early termination and our load balancing algorithm works. For now it suffices to know that, because of early termination, some of the subtree enumerations are not finished after a single launch of the GPU kernel. This is the main reason why the entire algorithm is iterated several times. At each iteration the GPU launches a mix of enumerations: new subtrees (start vectors) from the top enumeration and subtrees that were not finished in one of the previous GPU launches. Experimental results without early termination of threads are presented in Table 4.3 the experiments section.

4.1.2 The Iterated Parallel ENUM Algorithm

Algorithm 4.1: High-level Iterated Parallel ENUM Algorithm

Input: $\mathbf{b}_i (i = 1, \ldots, n)$, A, α, n

1. Compute the Gram-Schmidt orthogonalization of $\mathbf{b}_i$
2. **while** *true* **do**
3. $\quad S = \{(\mathbf{u}_k, \mathbf{\Delta u}_k, L_k = \alpha, s_k = 0)\}_k \leftarrow$ Top enum: generate at most NUMSTARTPOINTS $- \#T$ vectors
4. $\quad R = \{(\bar{\mathbf{u}}_k, \mathbf{\Delta u}_k, L_k, s_k)\}_k \leftarrow$ GPU enumeration, starting from $S \cup T$
5. $\quad T \leftarrow \{R_k :$ subtree k was not finished$\}$
6. $\quad$ **if** $\#T <$ CPUTHRESHOLD **then**
7. $\quad\quad$ Enumerate the starting points in T on the CPU.
8. $\quad\quad$ Stop
9. $\quad$ **end**
10. **end**

Output: $(u_1, \ldots, u_n)$ with $\|\sum_{i=1}^{n} u_i \mathbf{b}_i\| = \lambda_1(\mathcal{L})$

Algorithm 4.1 shows the high-level layout of the GPU enumeration algorithm. Details concerning the update of the bound A, as well as the write-back of newly discovered optimal vectors have been omitted. The actual enumeration is also not

shown: it is part of several subroutines which are called from the main algorithm.

The whole process of launching a grid of GPU threads is iterated several times (Line 2), until the whole search tree has been enumerated either on GPU or CPU.

In Line 3, the top of the search tree is enumerated, to generate a set S of starting vectors $\mathbf{u}_k$ for which enumeration should be started at level α. More detailed, the top enumeration in the region between α and n outputs distinct vectors

$$\mathbf{u}_k = [0, \ldots, 0, u_{k,\alpha}, \ldots, u_{k,n}] \qquad \text{for } k = 1 \ldots \text{NUMSTARTPOINTS} - \#T\,.$$

The top enumeration will stop automatically if a sufficient number of vectors from the top of the tree have been enumerated. The rest of the top of the tree is enumerated in the following iterations of the algorithm.

Line 4 performs the actual GPU enumeration. In each iteration, a set of starting vectors and starting levels $\{\mathbf{u}_k, L_k\}$ is uploaded to the GPU. These starting vectors can be either vectors generated by the top enumeration in the region between α and n (in which case $L_k = \alpha$) or the vectors (and levels) written back by the GPU because of early termination, so that the enumeration will continue. In total NUMSTARTPOINTS vectors (a mix of new and old vectors) are uploaded at each iteration. For each starting vector $\mathbf{u}_k$ (with associated starting level L_k) the GPU outputs a vector

$$\bar{\mathbf{u}}_k = [\bar{u}_{k,1}, \ldots, \bar{u}_{k,\alpha-1}, u_{k,\alpha}, \ldots, u_{k,n}] \qquad \text{for } k = 1 \ldots \text{NUMSTARTPOINTS}$$

(which describes the current position in the search tree), the current level L_k, the number of enumeration steps s_k performed and also part of the internal state of the enumeration. This state $\{\bar{\mathbf{u}}_k, \mathbf{\Delta u}_k, L_k\}$ can be used to continue the enumeration later on. The vectors $\mathbf{\Delta u}_k$ are used in the enumeration to generate the zig-zag pattern and are part of the internal state of the enumeration [SE94]. This state is added to the output to be able to efficiently restart the enumeration at the point it was terminated. The values for c and l can be computed out of $\mathbf{u}$ and L_k. Other than on CPU, we aim at saving storage on GPU and do only copy the necessary information to the GPU device.

Line 5 will select the resulting vectors from the GPU enumeration that were terminated early. These will be added to the set T of *leftover* vectors, which will be relaunched in the next iteration of the algorithm. If the set of leftover vectors is too small to get an efficient GPU enumeration, the CPU takes over and finishes off the last part of the enumeration.

GPU Threads and Load Balancing. In Section 4.1.1 the need for a load balancing algorithm was introduced: all threads should remain active and to ensure this, each

thread in the same warp should run the same instruction. One of the problems in achieving this is the length difference of each subtree enumeration. Some very long subtree enumeration can cause all the other threads in the warp to become idle after they finish their subtree enumeration.

Therefore the number of enumeration steps that each thread can perform on a subtree is limited by $\mathbf{M}$. When $\mathbf{M}$ is exceeded, a subtree enumeration is forced to stop. After this, all threads in the same warp will reinitialise: they will either continue the previous subtree enumeration (that was terminated by reaching $\mathbf{M}$) or they will pick a new starting vector of the list $S \cup T$ delivered by the CPU. Then the enumeration starts again, limited to $\mathbf{M}$ enumeration steps.

In our experimental setting, NUMSTARTPOINTS was around 20-30 times higher than NUMTHREADS, which means that on average every GPU thread enumerated 20-30 subtrees in each iteration. $\mathbf{M}$ was chosen to be around 50-200. The influence of all these parameters are depending on the GPU in use. Here we present the values that led to the best performance on our GTX 280 card.

4.2 Experimental Results

In this section we present some results of the CUDA implementation of our algorithm. For comparison we used the highly optimized ENUM algorithm of the fpLLL library in version 3.0.11 from [CPS]. NTL does not allow to run ENUM as a standalone SVP solver, but [Puj08] and the ENUM timings of [GN08b] show that fpLLL's ENUM runs faster than NTL's (the bit size of the lattice bases used in [GN08b] is higher than what we used, therefore a comparison with those timings is to be drawn carefully).

The CUDA program was compiled using nvcc, for the CPU programs we used g++ with compiler flag `-O2`. The tests were run on an Intel Core2 Extreme CPU X9650 (using one single core) running at 3 GHz, and an NVIDIA GTX 280 graphics card. We run up to 100000 threads in parallel on the GPU. The code of our program can be found online [HKS11].

We chose random lattices following the construction principle of [GM03] with bit size of the entries of $10 \cdot n$. This type of lattices was also used in [GN08b] and [NS06]. We start with the basis in Hermite normal form and LLL-reduce them with $\delta = 0.99$. At the end of this section, we present some timings using BKZ-20 reduced bases, to show the capabilities of stronger pre-reduction.

Both algorithms, the enumeration of fpLLL (run with parameter `-a svp`) and our CUDA version, always output the same coefficient vectors and therefore a lattice

vector with shortest possible length. We now compare the throughput of GPU and CPU concerning enumerations steps. Section 2.3.2 gives the explanation what is computed in each enumeration step. On the GPU, up to 200 million enumeration steps per second can be computed, while similar experiments on CPU only yielded 25 million steps per second. We choose the parameter $\alpha = n - 11$ for our experiments, this shapes up to be a good choice in practice. If the value is too close to the root at level n, the serial part on top is minimized, but the number of subtrees is small and their size enormous (and enumeration of each subtree takes too long). If α is too small, the total number of subtrees is huge, which leads to a big number of iterations. Our choice is a compromise between both extremes. Table 4.1 and Figure 4.2 illustrate the experimental results.

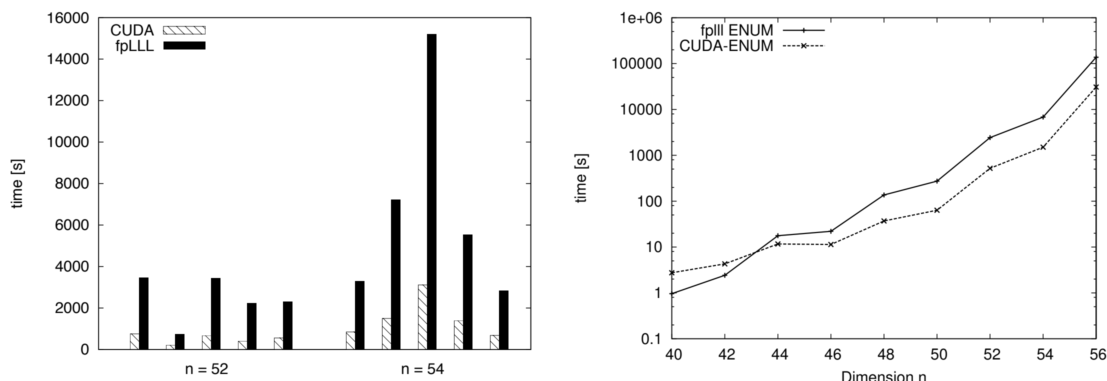


Figure 4.2: Timings for enumeration, in linear (left) and logarithmic scale (right). The left graph shows the time needed for enumerating five different LLL-reduced random lattices in each dimension n, the right graph presents average times. Both compare the ENUM algorithm of the fpLLL-library with our parallel CUDA version.

The figure shows the runtimes of both algorithms when applied to five different lattices of each dimension. The left picture indicates the runtime difference between different instances in one dimension. In the fight picture as one notices that in dimension above 44, our CUDA implementation always outperforms the fpLLL implementation. Both curves show super-exponential runtime, which conforms with theory.

Table 4.1 shows the average value over all five lattices in each dimension. Again one notices that the GPU algorithm demonstrates its strength in dimensions above 44, where the time goes down to 22% in dimensions 54 and 56 and down to 21% in dimension 52. Therefore we state that the GPU algorithm gains big speedups in dimensions higher than 45, which are the interesting ones in practice. In dimension

n	40	42	44	46	48	50	52	54	56	60
fpLLL - ENUM	0.96	2.41	17.7	22.0	136	273	2434	6821	137489 (38h)	-
CUDA - ENUM	2.75	4.29	11.7	11.4	37.0	63.5	520	1504	30752 (9h)	274268 (76h)
	286%	178%	66%	52%	27%	23%	21%	22%	22%	-

Table 4.1: Average time in seconds needed for enumeration of LLL pre-reduced lattices in each dimension n. The table presents the percentage of time that the GPU version takes compared to the fpLLL version.

60, fpLLL did not finish the experiments in time, therefore only the average time of the CUDA version is presented in the table.

Table 4.2 presents the timing of the same bases, pre-reduced using BKZ algorithm with blocksize 20. The time of the BKZ-20 reduction is not included in the timings shown in the table (it is the same for both implementations). For dimension 64 we changed α (the subtree dimension) from the usual $n-11$ to $\alpha = n-14$, as this leads to lower timings in high dimensions. First, one can notice that both algorithms run much faster when using stronger pre-processing, a fact that was already mentioned in [HS07]. Second, we see that the speedup of the GPU version goes down to 13% in the best case (dimension 62).

n	48	50	52	54	56	58	60	62	64
fpLLL - ENUM	2.96	7.30	36.5	79.2	190	601	1293	7395	15069 (4.2h)
CUDA - ENUM	3.88	5.42	16.9	27.3	56.8	119	336	986	4884 (1.4h)
	131%	74%	46%	34%	30%	20%	26%	13%	32%

Table 4.2: Average time needed for enumeration of BKZ-20 pre-reduced lattices in each dimension n. The time for pre-reduction is omitted in both cases.

As pruning would speed up both the serial and the parallel enumeration, we expect the same speedups with pruning.

It is hard to give an estimate of the achieved speedup compared to the number of threads used: since GPUs have hardware-based scheduling, it is not possible to know the number of active threads exactly. Other properties, like memory access and divergent warps, have a much greater influence on the performance and cannot be measured in thread counts or similar figures. When comparing only the number of double fmadds, the GTX 280 should be able to do 13 times more fmadd's than a single Core2 Extreme X9650. A GTX 280 can do 30 double fmadds in a 1.3GHz cycle, a single Core2 core can do 2 double fmadds in every two 3GHz cycle, which gives us a speedup of 13 for the GTX 280. Based on our results we fill only 30 to 40% of the GPUs ALUs. Using the CUDA Profiler, we determine that in our

experiments around 12% of branches was divergent, which implies a loss of parallelism and also some ALUs being left idle. There is also a high number of warp serializations due to conflicting shared and constant memory access. The ratio warp serializations/instructions is around 35%.

To compare CPUs and GPUs, we can have a look at the cost of both platforms in dollardays, similar to the comparison in [BCC+09]. We assume a cost of around $2200 for our CPU (quad core) + 2x GTX 295 setup. For a CPU-only system, the cost is only around $900. Given a speedup of 5 for a GPU compared to a CPU, we get a total speedup of 24 (4 CPU cores + 4 GPUs) in the $2200 machines and only a speedup of 4 in the CPU-only machine, assuming we can use all cores. This gives $225 \cdot t$ dollardays for the CPU-only system and only $91 \cdot t$ dollardays for the CPU+GPU system, where t is the time. This shows that even in this model of expense, the GPU implementation gains an advantage of around 2.4.

To show the necessity of load balancing we include timings of our GPU enumeration in the same LLL pre-reduced lattices using the same hardware without the early termination approach in Table 4.3. The runtimes can be compared to those in Table 4.1. In dimension 54 for example the runtime of enumeration decreases from 2599 to 1504 seconds when early termination is used. The fpLLL times were performed twice and therefore differ slightly in both tables. The comparison of both

n	45	48	50	52	54
fpLLL	18.3	139	277	2483	6960 (116min)
CUDA	20.2	92	133	959	2599 (43min)
	110%	66%	48%	39%	37%

Table 4.3: Average time in seconds needed for enumeration in dimension n, without early termination.

tables shows that the naive approach of top enumeration itself is not sufficient for a good parallelization. It requires some more sophisticated scheduling in order to occupy the GPU and gain more meaningful speedups.

Further Comments. In this chapter we only show experimental results up to dimension 64, since running pure enumeration without pruning is only suitable in small dimensions. The ideas of this chapter can as well be applied in combination with pruning techniques, as shown in Chapter 5. In order to test scalability of the algorithm, it is possible to distribute the start vectors to several graphics cards, and perform subtree enumeration in parallel on multiple cards. Since the subtrees are

independent and the memory overhead is limited, we expect linear speedup with the number of cards, i.e., doubling the number of GPUs will lead to about half runtime.

The parameter α controls the number of iterations (GPU calls), which is already at least bigger than 10 in all tested cases ($\alpha \leq 14$). This offers the possibility to generate more start points by increasing α, i.e., enlarging the top part of the enumeration tree. With increasing lattice dimension, the number of start points increases directly. Therefore, it is possible to use generate sufficiently many subtrees and distribute them to multiple GPUs. For small numbers of cards (say ≤ 15) this allows for linear speedup in the number of cards. Using more cards than that is more practical for the extreme pruning approach shown in the next chapter.

5 Extreme Pruning Enumeration on GPU and in Clouds

In this chapter, we describe our improvements to Extreme Enumeration, in order to evolve the fastest public SVP solver implementation. We aim at combining the fastest existing SVP algorithm with fast compute hardware. First, we integrate the Extreme Pruning idea of Gama et al. [GNR10] into the GPU implementation of Chapter 4. Second, we extend the implementation by using a more flexible bounding function in polynomial representation. We run it on multiple GPUs as well as on Amazon's EC2 compute cloud (via the MapReduce framework) in order to harness the immense computational power of such cloud services. Third, we extrapolate our average-case runtimes in order to estimate the running time of our implementation for solving α-SVP instances of the SVP Challenge in higher dimensions. Consequently, we set new records for the SVP challenge in dimensions 114, 116, and 120. The previous record was for dimension 112. Our implementation allows us to find a short vector at dimension 114 using 8 NVIDIA video cards in less than two days. Spending 2,300 USD, Amazon's cloud service solved the 120-dimensional challenge and set the new SVP challenge record.

As a result, the average "cost" of solving α-SVP with our implementation can henceforth be measured directly in U.S. dollars, as rent paid to cloud-computing service providers, taking Lenstra's *dollarday* metric [Len05] to a next level. That is, the cost will be shown literally as an amount on your invoice, e.g., the effort in our solving a 120-dimensional instance of the SVP Challenge translates to a $2,300$ USD bill from Amazon. Moreover, this new measure of complexity is simpler and more practical in that the parallelizability of the algorithm or the parallelization of the

implementation is *explicitly* taken into account, as opposed to being assumed or unspecified in the dollarday metric. Needless to say, such a cost should be understood as an upper bound obtained based on our implementation, which can certainly be improved, e.g., by using a better bounding function or better programming.

Throughout the rest of this chapter, our goal will be to find a vector below $1.05 \cdot FM(\mathcal{L})$, the same as in the SVP challenge. We do *not* aim at solving *exact* SVP. In this chapter we first present an overview of cloud computing, focusing on Amazon's EC2. Second, we explain our algorithm design and details of the implementations. The crucial part here is the selection of a suitable bounding function for pruning. Third, we present experimental results including a cost function that allows to estimate the cost of breaking SVP challenges in higher dimensions.

A preliminary version of this chapter was published in CHES 2011 [KSD+11]. The dissertation author was one of the principal investigators and authors of this paper. Parts of the computations of this chapter were performed on the compute clusters of the *Center for Scientific Computing Frankfurt* [csc].

5.1 Cloud Computing and Amazon EC2

Cloud computing is an emerging computing paradigm that allows data centers to provide large-scale computational and data-processing power to the users on a "pay-as-you-go" basis. Amazon Web Services (AWS) is one of the earliest and major cloud-computing providers, who provides, as the name suggests, web services platforms in the cloud. The Elastic Compute Cloud (EC2) provides compute capacity in the cloud as a foundation for the other products that AWS provides. With EC2, the users can rent large-scale computational power on demand in the form of "instances" of virtual machines of various sizes, which is charged on an hourly basis. The users can also use popular parallel computing paradigms such as the MapReduce framework [DG04], which is readily available as the AWS product "Elastic MapReduce." Furthermore, such a centralized approach also frees the users from the burden of provisioning, acquiring, deploying, and maintaining their own physical compute facilities.

Naturally, such a paradigm is economically very attractive for most users, who only need large-scale compute capacity occasionally. For large-scale computations, it may be advisable to buy machines instead of renting them because Amazon presumably expects to make a profit on renting out equipment, so our extrapolation might over-estimate the cost for long-term computations. However, we believe that these cloud-computing service providers will become more efficient in the years to

	Elastic Compute Cloud	1 Year Reserved Pricing	Elastic MapReduce
`cc1.4xlarge`	1.60 USD/hour	4290 USD + 0.56 USD/hour	0.33 USD/hour
`cg1.4xlarge`	2.10 USD/hour	5630 USD + 0.74 USD/hour	0.42 USD/hour

Table 5.1: Pricing information from `http://aws.amazon.com/ec2/pricing/`.

come if cloud computing indeed becomes the mainstream paradigm of computing. Moreover, trade rumors has it that Amazon's profit margins are around 0% (break-even) as of mid-2011, and nowhere close to 100%, so we can say confidently that Amazon rent cannot be more than 2× what a large-scale user would have spent if he bought and maintained his own computers and networking. Thus, Amazon prices can still be considered a realistic measure of computing cost and a good yardstick for determining the strength of cryptographic keys.

In estimating complexity such that of solving (α-)SVP or problems of the same or similar nature, Amazon EC2 can be used to provide a common measure of cost as a metric in comparing alternative or competing cryptanalysis algorithms and their implementations. Moreover, when using the Amazon EC2 metric, the parallelizability of the algorithm or the parallelization of the implementation is *explicitly* taken into account, as opposed to being assumed or unspecified. In addition to its simplicity, we argue that the EC2 metric is more practical than the *dollardays* metric of [Len05], and a recent report by Kleinjung, Lenstra, Page, and Smart [KLPS11] also agrees with us in taking a similar approach and measure with Amazon's EC2 cloud.

AWS offers several different compute instances for their customers to choose based on their computational needs. The one that interests us the most is the largest instance called "Cluster Compute Quadruple Extra Large" (`cc1.4xlarge`) which is designed for high-performance computing. Each such instance consists of 23 GB memory provide 33.5 "EC2 Compute Units" where each unit roughly "provides the equivalent CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor," according to Amazon.

Starting from late 2009, AWS also adds to its inventory a set of instances equipped with GPUs, which is called "Cluster GPU Quadruple Extra Large" (`cg1.4xlarge`), which is basically a `cc1.4xlarge` plus two NVIDIA Tesla "Fermi" M2050 GPUs. As of the time of writing, the prices for renting the above compute resources are shown in Table 5.1. The computation time is always rounded up to the next full hour for pricing purposes.

For computations lasting less than 172 days it is cheaper to use on-demand pricing.

For longer runs, there is an option to "reserve" an instance for 1 year (or even 3), which means that the user pays an up-front cost (see table above) to cut the on-demand cost of these instances.

5.2 Parallel Algorithm Design and Implementation

For each randomized basis, we use LLL-XD followed by BKZ-FP of the NTL library [Sho] with $\delta = 0.99$, different blocksizes β, and pruning parameter $p = 15$. NTL allows to prune the enumeration tree in the enumeration subroutine of BKZ, in the sense of [SH95]. The effect of this pruning is not well-understood since its theory was flawed [GNR10], but since it allows for bigger block sizes for pre-reduction we decided to run pruned BKZ. As already mentioned above, the problem we address is finding a vector below a search bound $1.05 \cdot FM(\mathcal{L})$ that heuristically guesses the length of a shortest vector of the input lattice. Adapting our implementations to other goal values is straight forward. It will only change the success probability and the runtime, therefore, we have to fix the bound for this work.

5.2.1 Bounding Function

As mentioned above, selecting a suitable bounding function is an important part of extreme enumeration. It influences the runtime as well as the success probability of each enumeration tree. The bounding function we use is a polynomial $p(x)$ of degree eight that aims to fit the best bounding function of [GNR10] in dimension 110. We use

$$p(x) = \sum_{i=0}^{8} v_i x^i$$

where $\mathbf{v} = (9.1\cdot10^{-4}, 4\cdot10^{-2}, -4\cdot10^{-3}, 2.3\cdot10^{-4}, -6.9\cdot10^{-6}, 1.21\cdot10^{-7}, -1.2\cdot10^{-9}, 6.2\cdot 10^{-12}, -1.29\cdot10^{-14})$ to fit the 110-dimensional bounding function. For dimension n we use $p(x\cdot 110/n)$. Figure 5.1 shows our polynomial bounding function $p(x)$, scaled to dimension 90. This bounding function gave the best results compared to linear, piecewise linear, and step bounding functions introduced in the theoretic part of [GNR10]. Table 5.2 shows example results of Extreme Enumeration, indicating that the polynomial function is superior among the bounding functions. We ran Extreme Pruning using different bounding functions in dimension 80. Higher dimensions are impractical due to the huge runtime of some bounding functions. We use BKZ-15 pre-reduction, since in dimension 80, stronger BKZ with higher blocksize (say 20 or 25) would directly find shortest vectors. The timings show the typical behaviour of

	polynomial	linear	piecewise linear	step
1. runtime [s]	1937	32795	> 12000	7006
2. enum calls	48	12	> 6000	82
3. single enum time [s]	490	32755	≈ 1	1072
(2.) $\cdot$ (3.)	23,520	393,060	-	87,904

Table 5.2: Extreme Pruning using different bounding functions on 12 CPU cores, in an 80-dimensional lattice, with BKZ-15 pre-reduction. The BKZ pre-reduction took about 1s for each instance and is omitted in the table. With piecewise linear pruning, the experiments were stopped after 6000 enum calls without result.

the different bounding functions. The linear bounding function takes quite a long time for single enumeration, and finds a shortest vector in the first run (12 enum calls indicates that in the first iteration on 12 cores a shortest vector was found). Piecewise linear pruning cuts off too many parts of the search tree and therefore runs very fast (1 second for an enum instance) but does not find a shortest vector. Stepwise bounding function cuts off the "wrong" parts of the tree. Compared to the polynomial function, it takes more time for a single enumeration and it has to start more instances. So the polynomial bounding function promises the best total runtime, since it gives the best trade-off between single enum runtime and the number of instances that have to be started (success probability) among the tested candidates.

Using an MPI-implementation for CPU we gained a success probability of finding a vector below $1.05 \cdot FM(\mathcal{L})$ of $p_{succ} > 10\%$. We ran experiments using the SVP challenge lattices, in order to assess the practical success probability (the probability of a single ENUM run to find a short vector) of extreme pruning using the polynomial bounding function $p(x)$. Using a multicore CPU implementation we started extreme pruning on up to $10,000$ lattices in each dimension (we stopped each experiment after 20 hours of computation). Figure 5.3 shows the average success rate of BKZ (with pruning parameter 15) and ENUM in dimensions 80 to 96 for different BKZ blocksizes. The values shown are the number of successfully reduced lattices divided by the number of started lattices in each dimension.

With BKZ blocksize 20, the pre-reduction was not strong enough, so neither BKZ nor ENUM could find a vector below the search bound in dimensions ≥ 96 within 20 hours. In dimension 100, the number of finished enumeration trees was already too small to derive a meaningful success rate.

The success rate of BKZ vanishes in higher dimensions. For each BKZ blocksize,

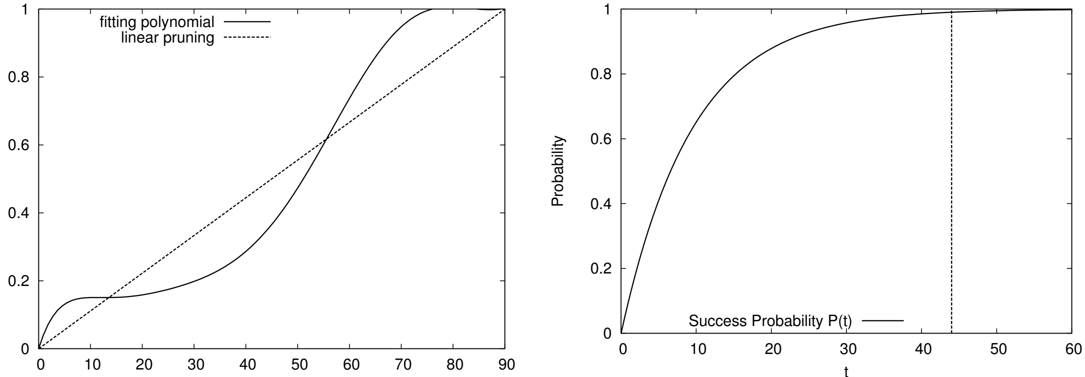


Figure 5.1: The new polynomial bounding function $p(x)$, scaled to lattice dimension 90. The dashed line shows a linear bounding function.

Figure 5.2: The final success probability of Extreme Enum assuming a success probability $p_{succ} = 10\%$ for one single tree. On average, we have to start 44 trees to finish with success probability $> 99\%$.

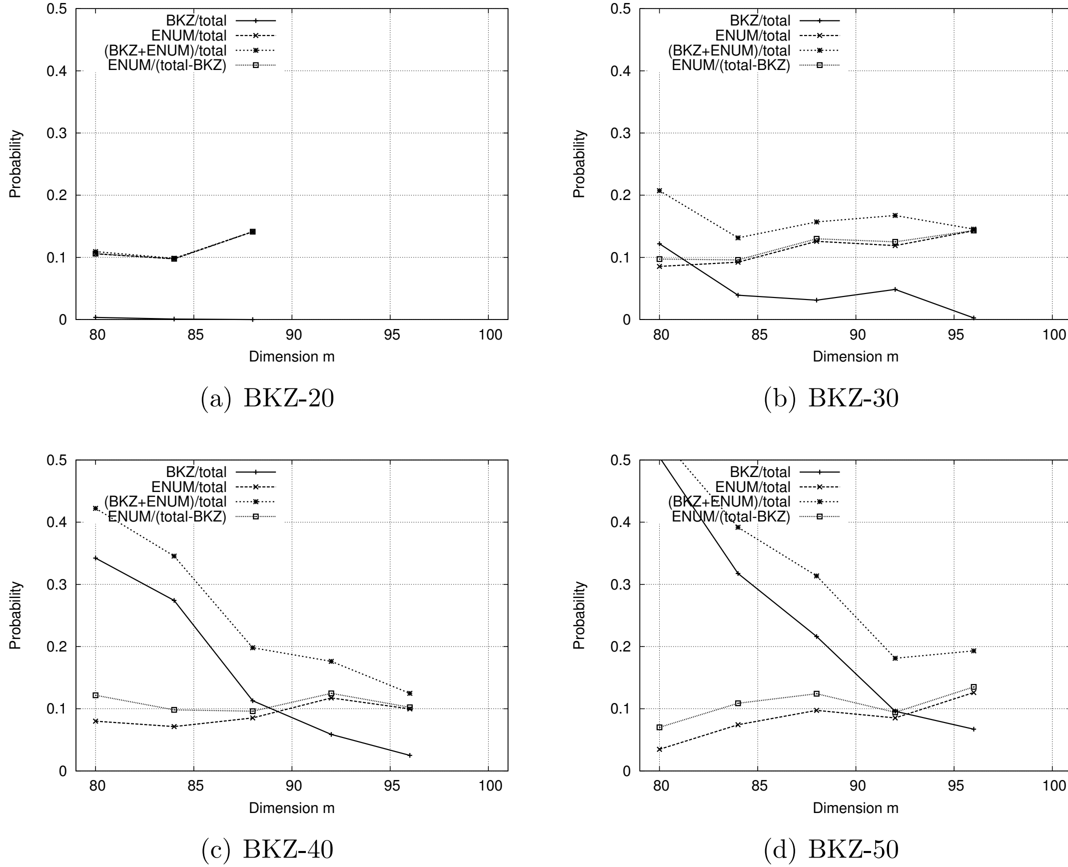
the success rate of ENUM stabilizes at a value $> 10\%$. Since the success rate is higher than this value in almost every case, we assume a value of $p_{succ} = 10\%$ for our polynomial bounding function $p(x)$.

Figure 5.3 shows the expectation values of the success of BKZ and ENUM. More exactly, it shows the expectation value $E(X)$ of $P(X \leq t)$, which gives a success probability of $p = 1/E(X)$. For higher dimensions $m > 90$ the success probability of BKZ tends to zero in every tested case. $P(t) = 1 - (1 - p_{succ})^t$ is the success probability to find a shortest vector below $1.05 \cdot FM(\mathcal{L})$ when starting t enumeration trees in parallel. Figure 5.2 shows the success probability P for $p_{succ} = 10\%$. This implies that on average we have to start 44 trees to find a vector below the given bound with probability $P(t) > 99\%$ (and not $1/p_{succ}$ many trees, as one could imagine).

For a comparable bounding function, the authors of [GNR10] get a much smaller success probability. This is due to the fact that in our case we expect about 1.05^n many vectors below the larger search bound, whereas the analysis of Gama et al. assumes that only a single vector exists below their bound.

5.2.2 Parallelization of Extreme Pruning using GPU and Clouds

Our overall parallelization strategy of Extreme Pruning Enumeration follows the model shown in Figure 5.4. For success, it is sufficient if one randomized instance of ENUM finishes. The number of instances we start depends on the success probability



(a) BKZ-20

(b) BKZ-30

(c) BKZ-40

(d) BKZ-50

Figure 5.3: Average values of success of the polynomial bounding function. total = number of samples; BKZ = number of samples solved by BKZ; ENUM = number of samples solved by pruned enumeration.

of each instance, which itself is depending on the bounding function used. The high-level algorithm run by each multicore-Enum or GPU-Enum instance is illustrated in Figure 5.5.

For the calculation of the cost, it makes no difference if we use 8 cores for a multicore-tree or only one core. In practice, however, we can stop the whole computation if one of the trees has found a vector below the bound. Therefore, using multiple cores for enumeration may have some influence on the running time.

GPU Implementation. We used the implementation of Chapter 4 and included pruning according to [GNR10]. The GPU enumeration uses enumeration on top of the tree, which is performed on CPU, to collect a huge number of *starting points*, as shown in Figure 5.5. These starting points are vectors $(\times, \ldots, \times, x_{n-\alpha+1}, \ldots, x_n)$,

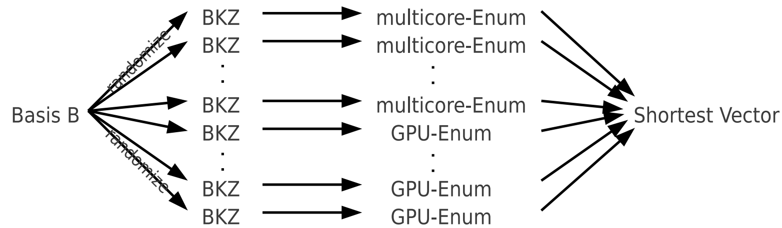


Figure 5.4: The model of our parallel SVP solver. The basis **B** is randomized, and each instance is solved either on CPU or on GPU. In the end, the shortest of all found vectors is chosen as output. Since we use pruned enumeration, not all instances will find a vector below the given bound.

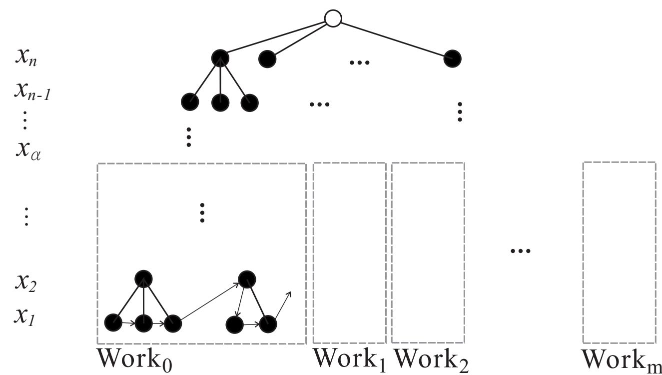


Figure 5.5: Illustration of the parallel enumeration process. The top tree $x_n, x_{n-1}, ..., x_\alpha$ is enumerated on a single core, and the lower trees $x_{\alpha-1}, ..., x_2, x_1$ are explored in parallel on many mappers.

where only the last α coefficients are set. A starting point can be seen as the root of a subtree in the enumeration tree. All starting points are copied to the GPU and enumerated in parallel. Due to load balancing reasons, this approach is done iteratively, until no more start points exist on top of the tree (see Chapter 4 for more details).

Since the code of extreme pruning only changes a few lines compared to usual enumeration, including pruning to the GPU implementation is straightforward. The improvement mentioned in [GNR10] concerning storage of intermediate sums was in parts already contained in the implementation of Chapter 4, so only slight changes were integrated into the GPU ENUM.

The GPU implementation allows the usage of different bounding functions, but for simplicity reasons we stick to the polynomial function specified above. Our implementation is available online [HKS11].

MapReduce Implementation. Our MapReduce implementation is also based on Chapter 4. The MapReduce framework requires to start all ENUM instances at the same time, which is only possible with the GPU approach, using top level enumeration first. The overall search process is illustrated in Figure 5.6. Specifically, we divide the search tree to top and lower trees. A top tree, which consists of levels x_n through x_α, is enumerated by a single thread in a DFS fashion, outputting all possible starting points $(x_\alpha, \ldots, x_n)$ to a WorkList. When a mapper receives a starting point $(x_\alpha, \ldots, x_n)$ from the WorkList, it first populates the unspecified coordinates $x_1, \ldots, x_{\alpha-1}$ and obtains the full starting point

$$(x_1 = \lceil -\sum_{k=2}^{n} \mu_{k,1}, x_k \rfloor, \ldots, x_{\alpha-1} = \lceil -\sum_{k=\alpha}^{n} \mu_{k,\alpha-1}, x_k \rfloor, x_\alpha, ..., x_n).$$

It then starts enumerating the lower tree from level 1 through $\alpha - 1$.

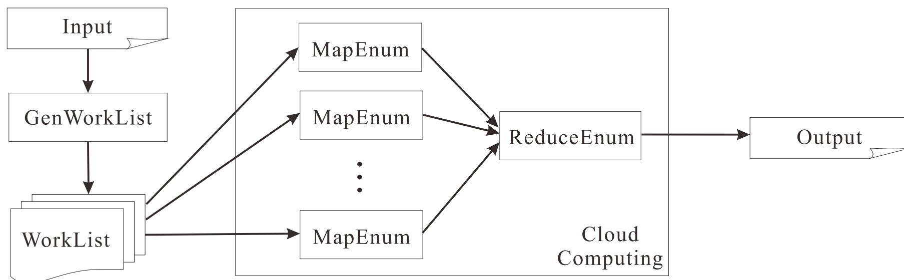


Figure 5.6: Illustration of our MapReduce implementation of the enumeration algorithm.

Because we scan the coefficients in a zigzag path, the lengths of the starting points usually show an increasing trend from the first to the last starting point. This can result uneven work distribution among the mappers. Therefore, we subdivide and randomly shuffle the WorkList so that each mapper gets many random starting points and hence have roughly equal amount of work among themselves. The effect is evident from the fact that the load-balancing factor, i.e., average running time divided by that of the slowest mapper, increases from 24% to 90%.

5.3 Experimental Results

In this section, we present the experimental results for our algorithmic improvements and parallel implementations on GPU and with MapReduce.

5.3.1 GPU Implementation

The GPU enumeration using extreme pruning solved the 114-dimensional SVP-challenge in about 40 hours using one single workstation with eight NVIDIA GeForce GTX 480 cards in parallel. Each GTX 480 has one GPU with 480 cores running at 1.4 GHz. The performance decreases from 200 Msteps/s to $\approx$ 100 Msteps/s using polynomial bounding function compared with an instance without pruning. With linear pruning, the decrease is less noticeable, but still apparent. This decrease is caused by the fact that subtrees are much thinner when pruning the tree. The number of starting points per second increases a lot, which coincides with the fact that subtrees, even though their dimension is much bigger now, are processed faster than without pruning.

We use 10 different lattices of the SVP challenge in each dimension 80–104 on the workstation equipped with eight GTX 480 cards to generate the timings of Figures 5.7 and 5.8 as well as Tables 5.3 and 5.4. The tables omit some dimensions n, whereas the figure does not contain a graph for each blocksize, due to readability reasons.

Workload Distribution between BKZ and ENUM. We note that in general, if we spend more time in BKZ to produce a better basis, we would have a higher probability of finding a short vector in the subsequent ENUM phase. A natural question is, what is the optimal breakdown of workload between BKZ and ENUM? We conjecture that the distribution should be roughly equal, as is supported by empirical evidence that we obtained from our experiments (cf. Figure 5.9). In our experiments, BKZ 40 performs the best in 104-dimensional instances, whereas in

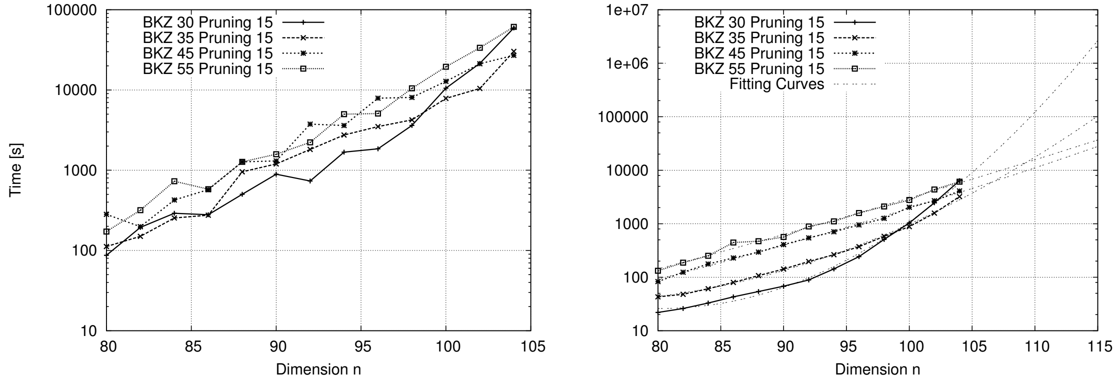


Figure 5.7: Total running time for solving SVP instances from dimension 80 to 104 using different BKZ blocksizes and NTL pruning parameter 15..

Figure 5.8: Running time for one single instance of pruned ENUM, including fitting curves $t_{30}(n)$ to $t_{55}(n)$.

n	80	84	88	92	96	100	104
BKZ-30	87	291	502	736	1847	10523	59317 (16h)
BKZ-35	113	254	957	1819	3502	7854	30256 (8h)
BKZ-40	143	350	1153	3246	4026	13307	21930 (6h)
BKZ-45	283	427	1272	3758	7896	12800	27021 (8h)
BKZ-50	122	456	1222	4521	11442	25691	45488 (13h)
BKZ-55	172	731	1272	2225	5081	19457	61131 (17h)

Table 5.3: Total running time in seconds for solving SVP instances with Extreme Pruning Enumeration on 8 GPUs from dimension 80 to 104 using different BKZ blocksizes and NTL pruning parameter 15.

n	80	84	88	92	96	100	104
BKZ-30	22	33	55	90	243	1032	6378 (106min)
BKZ-35	43	62	108	198	373	882	3219 (54min)
BKZ-40	65	130	183	342	601	1358	3225 (54min)
BKZ-45	83	178	296	545	951	2032	4094 (68min)
BKZ-50	111	217	340	655	1204	2447	4891 (82min)
BKZ-55	132	252	471	890	1588	2780	6113 (102min)

Table 5.4: Running time for one single instance of pruned ENUM.

Figure 5.9, it has a ratio that is the closest to 0.5. Similar trends can be observed for dimensions 86–97, for which the best BKZ block size is 30.

We use the data shown in Figure 5.9 to assess which of the curves from Figure 5.8 is the fastest one, and we use the extrapolation of this curve gained from data in dimension 80–104. This results in the cost function shown in Conjecture 5.1.

Conjecture 5.1 (GPU timing function)**.** Running BKZ and our implementation of pruned enumeration once on an NVIDIA GTX 480 GPU takes time

$$time_{GPU}(n) = \begin{cases} t_{30}(n) = 2^{0.0138n^2-2.2n+93.2} & \text{for } n \leq 97 \\ t_{35}(n) = 2^{0.0064n^2-0.92n+38.4} & \text{for } 98 \leq n \leq 104 \\ t_{45}(n) = 2^{0.001n^2+0.034n-2.8} & \text{for } 105 \leq n \leq 111 \\ t_{55}(n) = 2^{0.00059n^2+0.11n-5.8} & \text{for } 112 \leq n \end{cases} \quad sec\,.$$

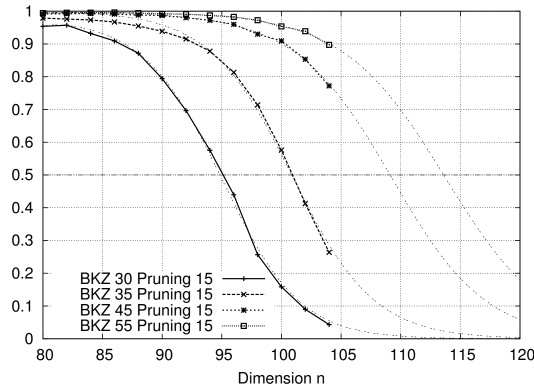


Figure 5.9: Ratio of BKZ runtime to total runtime for a single enumeration tree.

A more theoretic way to extrapolate the runtime would be to compute BKZ reduced bases, note the slope of the orthogonalized basis vectors, and use the runtime function of [GNR10] to compute the runtime. This approach ignores the runtime of BKZ (which is up to 50%) of the total runtime and relies on the Gaussian heuristic, while we are interested in practical runtime.

From the regression results shown in Figures 5.7 and 5.8, we can see that the run times for BKZ and ENUM are indeed polynomial and super-exponential, respectively. However, we notice that a larger BKZ block size does have a positive effect on the per-round running time of subsequent ENUM.

One difference is that Amazon uses M2050 GPU, not GTX 480 (like in our experiments). The M2050 has better double precision performance. Since many operations in enumeration are performed using double precision operations, we expected a

huge speed-up for enumeration. However, tests on M2050 GPUs did not show large speed-ups. One possible explanation is as follows. On the GPU, many additional operations have to be performed in integer-precision in order to split the work and reach a good load balancing. Therefore, double-precision operations are less than a fourth of the total number of operations, which makes the speed-ups on M2050 GPUs minor.

5.3.2 MapReduce Implementation

Our MapReduce implementation is compiled by `g++` version 4.4.4 x86_64 with the options `-O9 -ffast-math -funroll-loops -ftree-vectorize`. Using the MapReduce implementation, we are able to solve the 112-dimensional SVP-challenge in a few days. More exactly, we were using 10 nodes, 84 physical cores (totaling 140 virtual cores as some of the cores are hyperthreaded), which gives a total number of 334 GHz.

We note that the bounding function used in this computation is different from the polynomial bounding function described earlier. We were lucky in that only after 101 hours, or 1/9 of the estimated time, a shorter vector was found. We also noticed that the runtime scales linearly with the number of CPU cores used in total, meaning if we increase the number of CPU cores by a factor of 10, the runtime will decrease by factor 10.

Overall, from the test data of solving SVPs at dimension 100, 102, and 104 using the same set of seeds, we found that a GTX 480 is roughly two to three times faster than a four-core, 2.4 GHz Intel Core i7 processor for running our SVP solvers. We conjecture that the running time for our MapReduce implementation is also similar to that of our GPU implementation, as shown in Conjecture 5.1.

5.3.3 Final Pricing

We use Conjecture 5.1 to derive the final cost function for solving SVP challenges in higher dimensions $n \geq 112$. Recall that Amazon instances have to be paid for complete hours, therefore we round the runtime in hours to the next highest integer value. Using 44 enumeration trees leads to a success probability of at least 99%.

Conjecture 5.2 (Final Pricing)**.** Solving an SVP challenge with our implementation in dimension $n \geq 112$ with a success probability of $\geq 99\%$ on Amazon EC2 (using on demand pricing) costs

$$cost_{GPU}(n) = \lceil time_{GPU}(n)/3600 \rceil \cdot 44 \cdot 2.52 \text{ USD}\,.$$

Following Conjecture 5.2 solving the 120-dimensional instance of the challenge costs $1,885$ USD, which is a bit less than the amount we paid for practically solving it (due to conservative reservation of compute resources on EC2). We actually fired up 50 `cg1.4xlarge` instances for a total of 946 instance-hours, and incurred a bill of $2,300$ USD. For instance, solving the 140-dimensional challenge would cost roughly $72,405$ USD.

5.3.4 Scalability

Our implementation allows for arbitrary hardware effort. To solve α-SVP with probability of more than 99% we require to run 44 random enumeration instances. Each instance can be solved either on single or multiple CPU, GPU, or a clustered combination of both. Therefore, the scalability is as in Chapters 3 and 4. An upper bound is only given by the available amount of money and the cloud resources.

6 Parallel Random Sampling on GPU

In this chapter we present CUDA-SSR, a GPU implementation of the Simple Sampling Reduction (SSR) algorithm that searches for short vectors in lattices. Here, the SVP approximation factor is bigger than in the last chapters. SSR changes the whole lattice basis, therefore falls into the category of lattice basis reduction algorithms. SSR makes use of the BKZ algorithm and complements an exhaustive search in a suitable search region to insert random, short vectors to the lattice basis. The sampling of short vectors can be executed in parallel. Although it is already mentioned in [BL06] that SSR is a good candidate for parallelization, we are the first to present a distributed version of SSR.

Our main goal is to develop a parallel algorithm that allows for good occupation of the ALUs of a single GPU. Considering multiple GPUs, linear speedup in the number of cards is desired. The main challenge here is the distribution of sampled vectors to the GPU. Adequate scheduling of vectors is crucial in order to get high performance on the GPU. We compute the maximum number of points that fit to the storage of the graphics card, and upload the exact amount of vectors to the card. This results in reasonably good occupation of the card and guarantees minimal number of calls to the GPU. This is the obvious way to perform scheduling, and it is already sufficient to occupy the GPU, as our results indicate.

The authors of [BL06] mention a sampling rate of up to $5,200$ samples per second (on a 2.4GHz Intel Pentium 4). On an NVIDIA GTX295 GPU (which was released in 2009) we get rates of more than $120,000$ samples per second. With this we are the first to present a parallel implementation of SSR and we make use of the computing capability of modern graphics cards to enhance the search for short vectors even more.

In this chapter we first present the required background knowledge about SSR in Section 6.1. Then we introduce the parallel algorithm design in Section 6.2. Finally, Section 6.3 shows the experimental study using the GPU implementation of SSR. Our experiments are twofold. First we compare CUDA-SSR to BKZ, and second we compare it to our CPU-SSR implementation to show the strength of the GPU.

A preliminary version of this chapter was published in CHES 2011 [SG11]. The dissertation author was the principal investigator and author of this paper. In order to gain more meaningful results, the experiments of Section 6.3.2 were extended for this thesis using more test lattices than [SG11].

6.1 Random and Simple Sampling Reduction

Schnorr presented the first sampling algorithm called Random Sampling Reduction (RSR) in [Sch03]. It is an adaption of BKZ, and applies BKZ together with the insertion of some randomly sampled vectors. Ludwig and Buchmann refine the algorithm and promise to make sampling practical with their Simple Sampling Reduction (SSR) in [BL06]. They get rid of two RSR assumptions, namely the *Randomness Assumption* (RA) and the *Geometric Series Assumption* (GSA), which they claim both do not hold in practice. They replace the independent random sampling of vectors in the search space by a deterministic exhaustive search. This makes it impossible to sample the same vector multiple times, which was the case for RSR. Ludwig gives a more detailed view on SSR in his dissertation thesis [Lud05]. The implementation of Ludwig is available upon request. Comparisons of his SSR implementation with BKZ on cryptographic lattices can be found, e.g., in [BLR08, BL09].

The idea of random sampling is the following. Iteratively, it switches between reduction of the basis (using BKZ) and sampling a random short vector of norm $< 0.99 \left\|\mathbf{b}_1\right\|^2$, which is then prepended to the reduced basis (cf. Algorithm 6.1).

Every basis vector $\mathbf{v} = [v_1, \ldots, v_n]$ can be written in its orthogonalized form $\mathbf{v} = \sum_{i=1}^{n} \nu_i \mathbf{b}_i^*$. We can write its squared norm as

$$\left\|\mathbf{v}\right\|^2 = \sum_{i=1}^{n} \nu_i^2 \left\|\mathbf{b}_i^*\right\|^2 . \tag{6.1}$$

Therefore, shortening a vector $\mathbf{v}$ is done either by decreasing ν_i or by decreasing the $\|\mathbf{b}_i^*\|$.

For a reduced basis $\mathbf{B}$ (either LLL or BKZ reduced), it is known that the norm of the orthogonalized vectors $\|\mathbf{b}_i\|$ decreases for increasing index i. This implies that for higher indices, the influence of the coefficient ν_i in Equation (6.1) is less noticeable

than for smaller indices. This fact helps interpreting the following definition of a search space. For a basis $\mathbf{B} \in \mathbb{Z}^{n\times n}$ and an integer u with $1 \le u \le n$ we define the set $\mathcal{S}_{u,\mathbf{B}}$ as the set of all lattice vectors $\mathbf{v} = \sum_{i=1}^{n} \nu_i \mathbf{b}_i^*$ with

$$|\nu_i| \le \begin{cases} 0.5 & \text{for } 1 \le i < n-u \\ 1 & \text{for } n-u \le i < n \end{cases} \quad , \quad \nu_n = 1 \tag{6.2}$$

and call it the search space. It is $\mathcal{S}_{u,\mathbf{B}} \subseteq \mathcal{L}(\mathbf{B})$, and this search space is supposed to contain short lattice vectors. The algorithm SAMPLE (Algorithm 6.2, original in [Lud05]) uses as input a lattice basis $\mathbf{B}$ and an integer value x, and as output it computes a vector $\mathbf{v} \in \mathcal{S}_{u,\mathbf{B}}$ in the search space. The bit representation of the integer x controls the sampling deterministically. If the search space $\mathcal{S}_{u,\mathbf{B}}$ consists of 2^u many points, running SAMPLE with all values $x \in \{1, \dots, 2^u\}$ guarantees that the complete search space is sampled.

Algorithm 6.1: SSR

Input: Lattice basis $\mathbf{B} \in \mathbb{Z}^{n\times n}$, GS-coefficients $\mathbf{R} \in \mathbb{Q}^{n\times n}$, bound $u_{max} \in \mathbb{N}$, blocksize β, norm bound A

Output: reduced basis $\mathbf{B}$ s.t. $\|\mathbf{b}_1\| < A$

```
1 B ← BKZ([b_1, ..., b_n], β)
2 while ||b_1|| > A do
3     for x = 1 to 2^{u_max} do
4         v ← SAMPLE(B, R, x)
5         if ||v||^2 ≤ 0.99 ||b_1||^2 then break
6     end
7     if x = 2^{u_max} then terminate("No short vector found")
8     B ← BKZ([v, b_1, ..., b_n], β)
9 end
```

Algorithm 6.1 shows a pseudo-code listing of SSR, Algorithm 6.2 shows a listing of SAMPLE. For more details on random sampling we refer to the works of [Sch03, Lud05, BL06].

6.2 Parallel Algorithm Design and Implementation

The CUDA-SSR approach in Algorithm 6.3 is a slightly changed variant of the original SSR algorithm. In each outer while loop, up to $2^{u_{max}}$ vectors are sampled in parallel, and the m shortest samples are added to the basis. The main difference

Algorithm 6.2: SAMPLE

Input: Lattice basis $\mathbf{B} \in \mathbb{Z}^{n \times n}$, GS-coefficients $\mathbf{R} \in \mathbb{Q}^{n \times n}$, $x \in \mathbb{Z}$
Output: vector $\mathbf{v} \in \mathcal{S}_{u,\mathbf{B}}$

```
1  v ← b_n, ν ← r_n
2  for j = n − 1 to 1 do
3      y ← ⌈ν_j − 0.5⌉
4      if x = 1 mod 2 then
5          if ν_j − y ≤ 0 then  y ← y − 1
6                        else  y ← y + 1
7      end
8      x ← ⌊x/2⌋, v ← v − y b_j, ν ← ν − y r_j
9  end
10 return v
```

to the original SSR is the sampling of new vectors $\mathbf{v}$, which is done on the GPU and returns not only a single vector but multiple ones within a bound of m. The calculated vectors $[\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m]$ are added to the front of the lattice $\mathbf{B}$ in a sorted order, before the extended lattice is reduced by the BKZ algorithm. With the adding of multiple vectors we get a benefit of a more stabilized reduction, as we will see in the experiments section.

The algorithm terminates if a given norm of the first vector of $\mathbf{B}$ is undercut by a new vector $\mathbf{v}$ or if no smaller vector is found in the given search space.

The subroutine PAR-SAMPLE (which is now executed on GPU) is a slightly changed variant of SAMPLE (Algorithm 6.2). The original sample algorithm was parallelized, so that it computes a huge number of vectors per call. The possibility of parallelization is based on the independence of the samples. The only difference among two samples is the input value x, which can be interpreted as an unique identifier or seed.

One sample is stored in the shared memory of a CUDA block. The amount of shared memory, which is used for producing one sample, consists of memory for the vector $\mathbf{v}$ ($4Byte \cdot dimension$), for the vector ν ($4Byte \cdot dimension$), for y ($4Byte$), and for a valid-Byte ($1Byte$). For one CUDA block a number of

$$samplesPerBlock = \left\lfloor \frac{available\ shared\ memory}{(4 + 4Byte) \cdot dimension + 4Byte + 1Byte} \right\rfloor$$

vectors are produced. If we use all available CUDA blocks, the overall number of samples is $65535 \cdot samplesPerBlock$ per call. For example, at a dimension of 80 one call calculates $65535 \cdot \lfloor \frac{16344}{8 \cdot 80+5} \rfloor = 1,638,375$ samples. The shared memory of

Algorithm 6.3: CUDA-SSR

Input: Lattice basis $\mathbf{B} \in \mathbb{Z}^{n \times n}$, GS-coefficients $\mathbf{R} \in \mathbb{Q}^{n \times n}$, bound $u_{max} \in \mathbb{N}$, blocksize β, norm bound A, add vector bound m

Output: BKZ-β reduced basis $\mathbf{B}$ s.t. $\|\mathbf{b}_1\| \leq A$

1 $\mathbf{B} \leftarrow BKZ([\mathbf{b}_1, \ldots, \mathbf{b}_n], \beta)$
2 foundSmaller $= true$
3 xOffset $= 0$
4 **while** $\|\mathbf{b}_1\| > A$ *and foundSmaller* $= true$ **do**
5   **while** $xOffset < 2^{u_{max}}$ **do**
6   **parallel** $[i = xOffset \ldots xOffset + maxSamplesPerCall]$ **do**
7             $[\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_m}]$, foundSmaller $\leftarrow$ PAR-SAMPLE$(\mathbf{B}, \mathbf{R}, x_i, m)$
8   **end**
9   **if** *foundSmaller* $= true$ **then** break inner while loop
10   xOffset += maxSamplesPerCall
11   **if** $xOffset \geq 2^{u_{max}}$ **then** terminate
12   **end**
13     $\mathbf{B} \leftarrow BKZ([\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m, \mathbf{b}_1, \ldots, \mathbf{b}_n], \beta)$
14 **end**

$16384 Byte$ is decreased by the parameters of the kernel call, which are also stored in shared memory ($16 Byte$ for dimBlock and dimGrid, $24 Byte$ for 3 pointers). These values might change for other CUDA compute capabilities.

6.2.1 Parallel Implementation of Subroutine Sample

Here we describe how we implemented the sampling of *samplesPerBlock* many vectors in $\mathcal{S}_{u,\mathbf{B}}$ on GPU. This is the main contribution of the paper.

The first step for determining *samplesPerBlock* samples in one CUDA block is to copy the entries of the last vector of the matrices $\mathbf{B}$ and $\mathbf{R}$ to $\mathbf{v}$ and ν in parallel. The matrices $\mathbf{B}$ and $\mathbf{R}$ resist in the texture memory, because they are read multiple times and this memory is cached.

The second step is to compute the factor y for every sample and build new vectors inside a for-loop. A single y is processed by one CUDA thread, therefore all $y's$ of one CUDA block can be calculated in parallel. Afterwards the temporary new vectors $\mathbf{v}$ and ν are built, whereby all entries of a vector are assigned in one parallel step. If an integer overflow is noticed in this step, the sample will be indicated as invalid.

When the loop is finished, the square norms of the new samples are calculated with the common *vector reduction* approach, after squaring all entries of $\mathbf{v}$ in parallel.

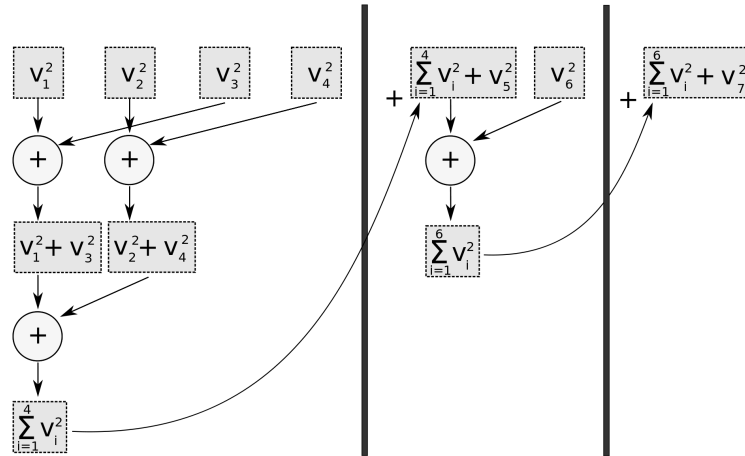


Figure 6.1: Computation of the norm of a single vector $\mathbf{v}$ in parallel.

Figure 6.1 illustrates this procedure. Once a square norm of 2^x (with $x = \max\{y \in \mathbb{Z} : 2^y \leq \mathit{dimension}\}$) is determined, the result will be added to the first entry of the next interval. This procedure continues, until there is no more than one entry left.

Because the square norms of all vectors are calculated step by step, we can register the smallest square norm of a CUDA block. Therefore a CUDA block writes only the smallest vector back to the global memory, assumed that the square norm is less than 99% of $\|\mathbf{b}_1\|^2$ and the sample is valid. With this we save a lot of global memory. Instead of writing $65535 \cdot \mathit{samplesPerBlock}$ many vectors to global memory we use shared memory for *samplesPerBlock* many vectors of each block and only write 65535 many vectors to the device.

For achieving higher performance we introduce a counter, which increases if a vector with a square norm less than 99% of $\|\mathbf{b}_1\|^2$ is found. When m vectors below this bound have been found, we break the parallel sampling. The counter is increased with so called *atomic operations*, which provides an exclusive *read-modify-write* operation for one CUDA thread. The parallel processing of the CUDA framework is only "semi-parallel", because only a part of all CUDA blocks are processed parallel for real (we have 65535 blocks but only 30 multiprocessors available). Therefore we can abort further calculations, if the counter m reached a defined value. A flow chart of our GPU algorithm of SAMPLE is shown in Appendix A of [SG11]. In order to remove serialization we also tested replacing the condition in Line 4 of SAMPLE by arithmetic computations, but recognized no speedups. Since there is no *else*-block, the fact that (on average) half of the threads are idle does not influence the total runtime.

For establishing the gain of parallel sampling we also implement a CPU version of the SSR algorithm (called CPU-SSR), which produces new vectors step by step. Our CPU as well as the GPU implementation are available online [GS11].

6.3 Experimental Results

We are using an NVIDIA GTX 295 GPU for our experiments. The CPU that we use is an Intel Core2 Duo E8400 CPU running at 3GHz. The lattices we use are the SVP challenge lattices [GS10]. For LLL and BKZ reduction we use the NTL library [Sho] in version 5.5.2. The parameter δ is always set to the standard value 0.99. We run LLL with precision RR followed by BKZ with precision QP.

First we compare our results of CUDA-SSR to BKZ, and second we present experiments comparing CUDA-SSR to CPU-SSR.

6.3.1 Comparison of CUDA-SSR and BKZ

Let $\mathbf{B}$ be the basis of $\mathcal{L}(\mathbf{B})$ in dimension n and c be a constant. Using BKZ with blocksize β, Gama and Nguyen [GN08b] predict the average norm of the first basis vector after BKZ reduction to be

$$gn = c^n \det(\mathcal{L}(\mathbf{B}))^{1/n}, \tag{6.3}$$

where the *Hermite factor constant* c relies on the blocksize used. For BKZ-20, e.g., they experimentally gain a value of $c = 1.0128$.

Our experiments are performed as follows. First, we reduce a lattice basis with BKZ with increasing blocksize, until we reach a vector of desired goal norm gn, cf., Equation (6.3). We use a value of $c = 1.0129$ to calculate our goal norm (due to a typo, which has minor influence). In order to reduce the total runtime of the experiment we only use one lattice per dimension. The resulting runtimes and the reached norms are shown in Figures 6.2 and 6.3. Second, we use CUDA-SSR with half the blocksize (rounded up) that BKZ needed to reach the goal norm and run CUDA-SSR on the same lattice; i.e., random sampling has to close the gap between BKZ with half blocksize and BKZ with full blocksize. We stop the GPU sampling when $m = 0.25 \cdot n$ vectors below $0.99 \cdot \|\mathbf{b}_1\|^2$ were found by PAR-SAMPLE.

Figure 6.4 shows the blocksize that BKZ needed to find the resulting vector. The picture shows that the blocksize is around 20 in most of the cases, as predicted by [GN08b]. Figure 6.5 shows the speedup factor of CUDA-SSR compared to BKZ.

We notice that with both approaches, BKZ as well as CUDA-SSR, we find vectors of comparable length (Figure 6.2). CUDA-SSR is always faster (up to 40%). For

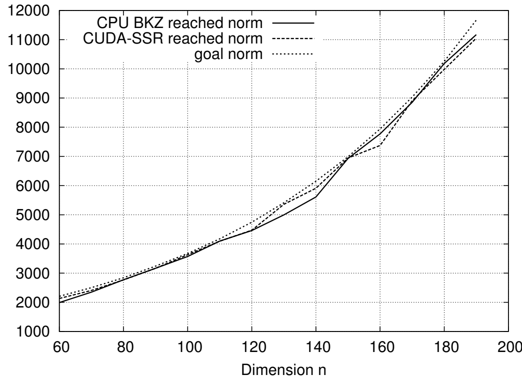


Figure 6.2: Reached norm of BKZ and CUDA-SSR.

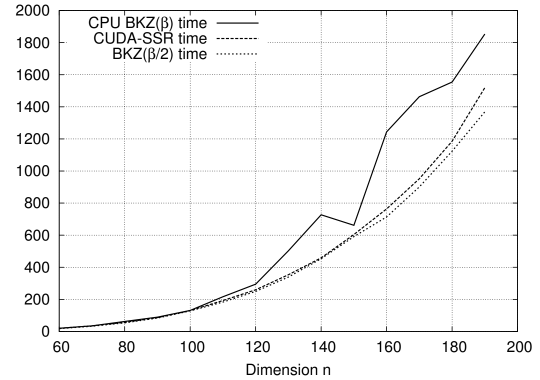


Figure 6.3: Required time in seconds for BKZ with blocksize β and for CUDA-SSR to find a vector below the same goal norm.

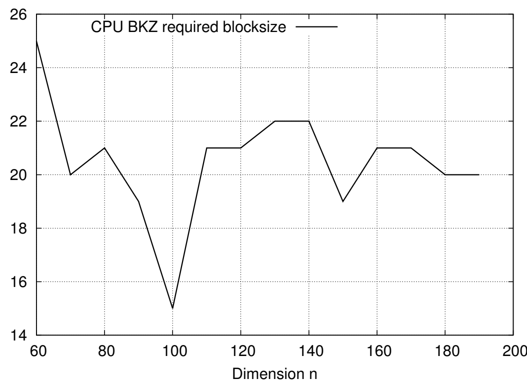


Figure 6.4: Required blocksize of BKZ-20 to reach the desired goal norm.

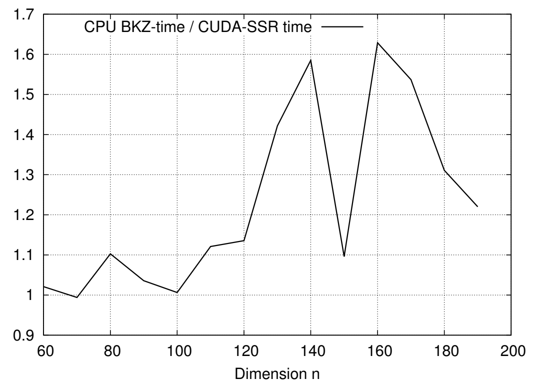


Figure 6.5: Speedup factor of CUDA-SSR in comparison to BKZ.

comparison reason, Figure 6.3 includes the runtime of BKZ with blocksize $\lceil\beta/2\rceil$, the pre-processing step of SSR (Line 1 of Algorithm 6.3). The picture shows that it takes a huge part of the random sampling time (dashed line). This implies that the later part of SSR (sampling - BKZ - sampling - ...) takes a lot less time (the time difference between the dotted and dashed curve) than the initial BKZ. Therefore, the total SSR runtime cannot profit too much from the parallel sampling part in this setting.

The runtime speedup factor (Figure 6.5) seems to increase with the dimension, from 1.1 in dimension 80 to a maximum value of 1.6 in dimension 160. The peek in dimension 150 is also apparent in Figure 6.4 and seems to result from special structure in the lattice (SSR is working less in this lattice).

6.3.2 Comparison of GPU and CPU Variant of SSR

Our second block of experiments is supposed to show the strength of parallelization on GPU of the SSR algorithm. For this, we run our CPU implementation and our GPU implementation of SSR for the same lattices until they undercut the goal norm. The values noted are average values for 10 different lattices in each dimension. For pre-reduction, we use LLL only. We note the reached norm (cf. Figure 6.6) and the runtime (cf. Figure 6.7). Figure 6.8 shows the speedup factor gained by the GPU version. We prepend $m = 0.1 \cdot n$ vectors to the basis in each GPU iteration. Figure 6.9 compares a typical behaviour of SSR on GPU and CPU over time, concerning the norm of the sampled vectors.

On CPU, the sampling rate was about 160 samples per second for a 180- dimensional lattice. The GTX 295 GPU reached about $120,000$ samples per second for a 180 dimensional lattice. In smaller dimension, sampling rates of more than $250,000$ are possible, e.g. in dimension 60.

We noticed that the runtime of SSR on GPU is very stable compared to the CPU version. We conclude that sampling multiple vectors in each iteration helps SSR to run much more stable. Figure The speedup factor shown in Figure 6.8 shows the potential of the CUDA version compared to the CPU version. In small dimension we gain speedup factors of up to 180. On GPU, in the first iteration a vector below the bound is already found, whereas on CPU multiple iterations have to be performed. In bigger dimensions, the speedup factor decreases, depending on the behaviour pattern.

Figure 6.9 shows a typical behaviour of SSR on CPU and GPU. CUDA-SSR starts with lower norm, which implies that the first iterations of SSR decreases the norm much more than on CPU. We noticed that in the first iterations, there exists a huge

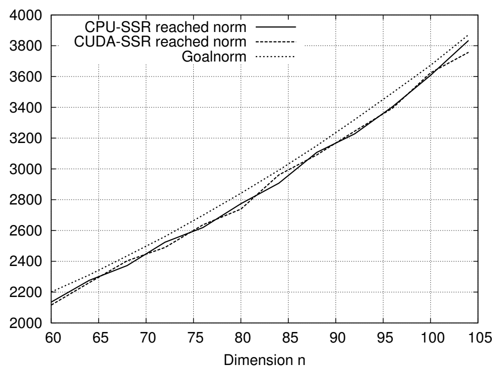


Figure 6.6: Reached norm of CPU-SSR and CUDA-SSR.

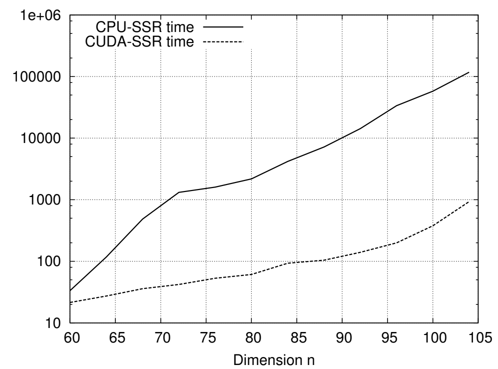


Figure 6.7: Required time in seconds of CPU-SSR and CUDA-SSR to find a vector below the same goal norm.

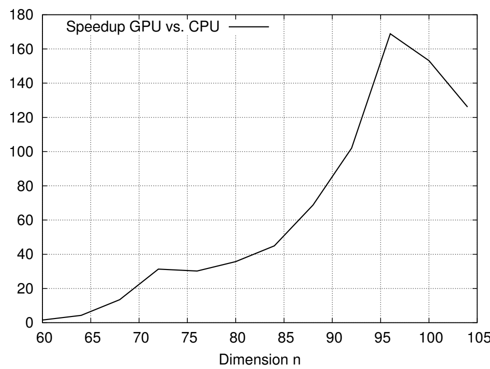


Figure 6.8: Speedup factor of GPU compared to CPU variant of SSR.

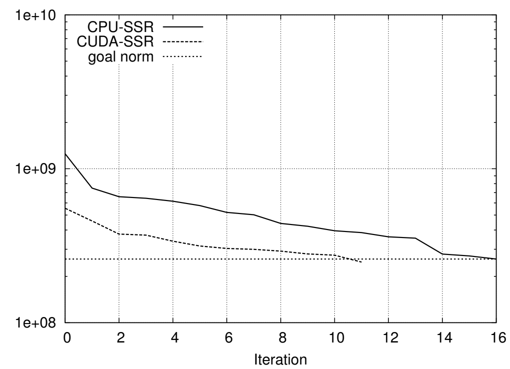


Figure 6.9: Development of the squared norm of SSR over time, in a 190 dimensional lattice. The ordinate shows the squared norm of the vectors found by sampling.

number of vectors below the $0.99 \|\mathbf{b}_1\|^2$ bound. Therefore, on GPU we have good chance to find a much shorter vector. On CPU only the first vector below the bound is picked, whereas on GPU multiple vectors are prepended to the basis, and all these vectors are potentially smaller than the CPU one.

To show the strength of our GPU version, Figure 6.10 shows the time needed by CUDA-SSR and CPU-SSR to sample the same amount of vectors, namely 2^{21} many. It is evident that on GPU, the sampling is much faster, with a maximum factor 14.5 in dimension 190. This speedup is even a bit higher than the maximum speedup factor in the number of FMADD operations that the hardware allows for (factor 13, cf. Chapter 4). Therefore, sampling itself reaches a good occupation of a single GPU.

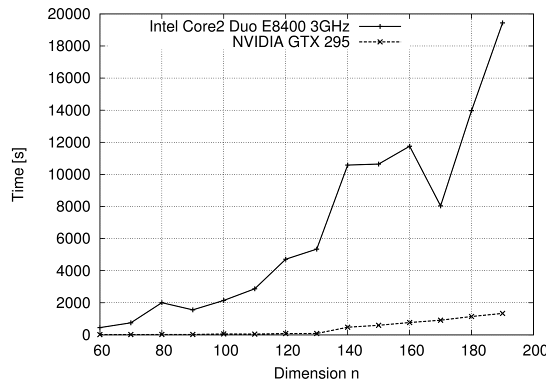


Figure 6.10: Time to sample 2^{21} many vectors using CUDA-SSR and CPU-SSR.

Further Comments. The percentage of BKZ of the total runtime was up to 97%. This is not optimal, since BKZ does not apply the hardware acceleration of the graphics card. LLL took 67% of the total runtime in dimension 100. The potential of parallelization is visible, but SSR does not take full advantage of it.

In this chapter we considered higher approximation factors exponential in the lattice dimension. We also tried to solve SVP challenge instances with lower approximation factors using SSR.This was only possible in reasonable runtime in dimension up to 85 (this took already more than a week). In higher dimensions, the goal norm of the challenge (i.e., the SVP approximation factor) is too small for SSR to be successful. Apparently SSR is stronger in higher lattice dimensions with higher approximation factors.

Distributing the samples to multiple graphics cards is possible by allocating the seeds $1, \ldots 2^u$ to different cards. The parameter u controls the number of samples

in the search space. With increasing dimension u can be chosen larger, therefore occupying multiple cards is possible with increasing dimension.

7 Sieving in Ideal Lattices

So far there is no SVP algorithm making use of the special structure of ideal lattices. It is widely believed that solving SVP (and all other lattice problems) in ideal lattices is as hard as in regular lattices [MR08, ADL$^+$08, Lyu09]. Our intention is to show how sieving algorithms can be strengthened in ideal lattices using their circular structure. The idea was already mentioned in [MV10b]. There, the authors assume that the amount of storage required by their algorithm decreases with a factor of p, where p is the degree of the field polynomial. We show that in practice not only the storage but as well the practical runtime of sieving algorithms decreases by a factor of p.

Micciancio and Voulgaris propose to use the cyclic rotations of each sampled vector to reduce the size of the vectors. For ideal lattices, the "rotation" of each lattice vector is still an element of the lattice. Therefore, it can be used in the sieving process, for ListSieve as well as GaussSieve. They expect a reduction of ListSieve's list size linear in the degree of the field polynomial, and a substantial impact on the practical behaviour of the GaussSieve algorithm. In this chapter, we present experimental results using this approach. We implement ListSieve and IdealListSieve without perturbations as well as IdealGaussSieve (based on an existing GaussSieve implementation). So far, there is not much insight to the behaviour of ListSieve and GaussSieve. Therefore, the main challenge is to understand and explain the performance of these algorithms, in order to allow for comprehension of the effect of rotations. There is a huge difference between worst case runtime of ListSieve using perturbations and the heuristic variants that we use in this chapter.

Our experiments show that indeed the storage requirements decrease as expected by [MV10b]. But even more, sieving in ideal lattices can find a shortest lattice

vector much faster, with a practical speedup factor linear in the degree of the field polynomial. To explain the results, we use the assumption that the number of vector reductions used in the sieving process stays the same in both the original and the ideal case. We will show that this assumption conforms with our experiments. To give an example, the measured and fitted runtime of IdealListSieve in cyclic lattices is $2^{0.51n-21.2}$ seconds, compared to $2^{0.67n-26.8}$ seconds for ListSieve. In dimension $n = 60$, the runtime difference is about 4 hours, which corresponds to a time advantage of 94% for IdealListSieve. The worst-case runtime of IdealListSieve remains the same as for ListSieve, since considering all rotations cancels out the factor p in theory.

To our knowledge, this is the first SVP algorithm that uses the special structure of ideal lattices. (For cyclic NTRU lattices, there is a LLL-variant using the cyclic rotations [MS01].) Since the runtime of sieving algorithms is exponential in p, this linear speedup does not effect the asymptotic runtime of sieving algorithms. It only helps to speed up sieving in ideal lattices in practice noticeably. For the fully homomorphic encryption challenges for example p is bigger than 2^{10}, which would result in a speedup of more than 1000 for sieving. The signature scheme of [Lyu09] uses $p \geq 512$. These numbers show that even if we only allow for a linear speedup this still might give huge speedups in practice.

In this chapter we aim at solving exact SVP, i.e., given a basis $\mathbf{B}$ of a lattice find a non-zero vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ with norm equal to $\lambda_1(\mathcal{L}(\mathbf{B}))$. In comparison to enumeration, sieving algorithms output a shortest lattice vector only with high probability. There is an exponentially small probability (in the lattice dimension) of missing a shortest vector. In this case, sieving algorithms output an approximate solution of SVP only. In Section 7.1 we describe the original ListSieve and develop the IdealListSieve algorithm. In Section 7.2 we present the theoretical analysis of the algorithm, and Section 7.3 shows experimental results of our implementation. We will use the notation introduced in Chapter 2.

A preliminary version of this chapter appeared in [Sch11c]. The dissertation author was the principal investigator and author of this paper. Parts of this chapter were presented in WALCOM 2011 [Sch11a].

7.1 IdealListSieve Algorithm

In this section we will present the IdealSieve algorithm of [MV10b] and introduce the ideal lattice variant IdealListSieve. More details about the implementation will follow in Section 7.3.

ListSieve. The idea of ListSieve is the following. A list L of lattice vectors is stored. In each iteration of the algorithm, a new random vector $\mathbf{p}$ is sampled uniformly at random from a set of bounded vectors. This vector $\mathbf{p}$ is then reduced using the list L in the following manner. If a list vector $\mathbf{l} \in L$ can be subtracted from $\mathbf{p}$ lowering its norm more than a factor $\delta < 1$, $\mathbf{p}$ is replaced by $\mathbf{p} - \mathbf{l}$. With this, $\mathbf{p}$ gets smaller every time. When the vector has passed the list it is appended to L. In the end, L will contain a vector of minimal length with high probability. If the sampled vector $\mathbf{p}$ is a linear combination of smaller list vectors it will be reduced to $\mathbf{0}$ and not be appended. This rare case is called a *collision*. Collisions are important for runtime proofs (they avert a runtime proof for GaussSieve, for example). For practical issues, they are negligible, since they occur very seldom. Algorithm 8 shows a pseudo-code of ListSieve without perturbations. Function `ListReduce` is shown on Page 76.

Algorithm 7.1: ListSieve($\mathbf{B}$, targetNorm)

```
1 List L ← LLL(B)                                  ▷ Pre-reduction with the LLL algorithm
2 while currentBestNorm > targetNorm do
3   | p ← sampleRandomLatticeVector(B)                               ▷ Sampling step
4   | ListReduce(p, L, δ = 1 − 1/n)                                  ▷ Reduction step
5   | if p ≠ 0 then
6   |   | L.append(p)                                                 ▷ Append step
7   | end
8 end
9 return l_best
```

Originally, ListSieve does not work with lattice points $\mathbf{p}$, but with a perturbed point $\mathbf{p} + \mathbf{e}$ with a small error $\mathbf{e}$. The use of perturbations is necessary in order to upper bound the probability of collisions, which is essential for proving runtime bounds for the algorithm. Since in practice collisions play a minor role we will skip perturbations in our implementation. For the sampling of random vectors in Line 3 the authors of [MV10b] use Kleins randomized rounding algorithm [Kle00], which we will also apply for our implementations.

IdealListSieve. One of the properties of ideal lattices is that for each lattice vector $\mathbf{v}$, rotations of this vector are also contained in the lattice. This is due to the property of the ideal $\mathbf{I}$ corresponding to the ideal lattice. Ideals in $\mathbf{R}$ are closed under multiplication with elements from $\mathbf{R}$, and since vectors in ideal lattices are the same as elements of the ideal, multiplications of these vectors are also elements of the lattice.

To compute the rotation of a vector $\mathbf{v}$ one has to rotate each block of length p of $\mathbf{v}$. If $n = 2p$, the first half of $\mathbf{v}$, which belongs to the $q\mathbf{I}_p$ part in the first rows of the basis matrix (2.2), is rotated and so is the second half. So when ListSieve tries to reduce the sample $\mathbf{p}$ with a vector $\mathbf{l} = (\mathbf{l}_1, \ldots, \mathbf{l}_p, \mathbf{l}_{p+1}, \ldots, \mathbf{l}_n)$, we can also use the vectors

$$\mathbf{l}^{(j)} = (\mathbf{rot}^j((\mathbf{l}_1, \ldots, \mathbf{l}_p)), \mathbf{rot}^j((\mathbf{l}_{p+1}, \ldots, \mathbf{l}_n))), \text{ for } j = 1 \ldots p-1\,,$$

where the first and the second half of the vector is rotated. Therefore, the sample $\mathbf{p}$ can be more reduced in each iteration. Instead of reducing with one single vector $\mathbf{l}$ per entry in the list L, p vectors $\mathbf{l}^{(j)}$ can be used.

Function `IdealListReduce` shows a pseudo-code of the function that is responsible for the reduction part. Compared to the ListSieve algorithm of [MV10b], this function replaces the `ListReduce` function. Unfortunately, only the case where n is a multiple of p allows the usage of rotations of lattice vectors. For the case where $p \nmid n$, it is not possible to apply the rotation to the last block of a lattice vector $\mathbf{v}$.

Func. ListReduce($\mathbf{p}, L, \delta$)

1 **while** $(\exists \mathbf{l}' \in L : \|\mathbf{p} - \mathbf{l}'\| \leq \delta \|\mathbf{p}\|)$ **do**
2 $\quad \mathbf{p} \leftarrow \mathbf{p} - \text{round}(\frac{\langle \mathbf{p} \mid \mathbf{l}' \rangle}{\langle \mathbf{l}' \mid \mathbf{l}' \rangle}) \cdot \mathbf{l}'$
3 **end**
4 **return p**

Func. IdealListReduce($\mathbf{p}, L, \delta$)

1 **while** $(\exists j \in [p], \mathbf{l} \in L, \mathbf{l}' = \mathbf{rot}^j(\mathbf{l}) : \|\mathbf{p} - \mathbf{l}'\| \leq \delta \|\mathbf{p}\|)$ **do**
2 $\quad \mathbf{p} \leftarrow \mathbf{p} - \text{round}(\frac{\langle \mathbf{p} \mid \mathbf{l}' \rangle}{\langle \mathbf{l}' \mid \mathbf{l}' \rangle}) \cdot \mathbf{l}'$
3 **end**
4 **return p**

The while loop condition in Line 1 introduces the rotation step. The reduction step in Line 2 differs from the original ListSieve description in [MV10b]. It uses the reduction step known from the Gauss (respectively Lagrange) algorithm (an orthogonal projection), that is also used in the LLL algorithm [LLL82]. This step is not explained in [MV10b], whereas their implementation [Vou10] already uses this improvement. The slackness parameter $\delta = 1 - 1/n$ is used to ensure that the norm decrease is sufficient for each reduction in order to guarantee polynomial runtime in the list size.

IdealGaussSieve. For ListSieve, when a vector joined the list once it remains unchanged forever. GaussSieve introduces the possibility to remove vectors from the list if they can be more reduced by a newly sampled vector. The reduction process is twofold in GaussSieve. First, the new vector $\mathbf{p}$ is reduced as in ListSieve. Second, all list vectors are reduced using $\mathbf{p}$. If a vector from the list is shortened, it will leave the

list, be stored on a stack, and pass the list again in one of the next iterations (called *stackpoints*). Therefore the list will consist of less and shorter vectors than in the ListSieve case. GaussSieve is the heuristic variant of ListSieve with better practical runtime, with less theoretical background knowledge about its runtime behaviour.

It is straightforward to include the rotations into GaussSieve in the same manner as for ListSieve. We can replace the function `GaussReduce` of [MV10b] by `IdealGaussReduce`, which uses the rotations twice. First it is used for the reduction of $\mathbf{p}$, second for the reduction of list vectors. The rest of GaussSieve remains unchanged. IdealGaussSieve is also included in our implementation. Since the behaviour of the GaussSieve variant is even harder to predict, it is more convenient to study the influence of rotations on ListSieve first.

7.2 Predicted Advantage of IdealListSieve

In this section we theoretically analyze the IdealSieve algorithm and try to predict the results concerning number of iterations I, the total number of reductions R, and the maximum size L of the list L. For comparison of an algorithm and its ideal lattice variant we will always use the quotient of a characteristic of the non-ideal variant divided by the ideal variant. We will always denote it with *speedup*. For example, the speedup in terms of reductions is $\mathsf{R}_{orig}/\mathsf{R}_{ideal}$.

Recall that the only change we made in the algorithm is that in the reduction step, all rotations $\mathbf{rot}^j(\mathbf{l})$ (for $j \in [p]$) of $\mathbf{l} \in L$ are considered instead of only considering $\mathbf{l}$. The runtime proof for ListSieve in [MV10b] uses the fact that the number of vectors of bounded norm can be bounded exponentially in the lattice dimension. Therefore, the list size L cannot grow unregulated. All list vectors have norm less than or equal $n\,\|\mathbf{B}\|$. For cyclic and anti-cyclic lattices, the norm of a vector remains unchanged when rotated. Therefore each list vector corresponds to p vectors of the same size, which results in a proven list size of factor p smaller. For prime cyclotomic lattices, the norm might increase when rotated (the expansion factor is > 1 in that case), therefore it is a bit harder to prove bounds on the size of the list.

We assume that for finding a shortest vector in the lattice, the total number of reductions is the same. Our experiments show that this assumption is reasonable (cf. Section 7.3). In this case we predict the number of iterations of IdealListSieve compared to ListSieve. When ListSieve performs t iterations (sampling - reducing - appending), we assume that IdealListSieve would take t/p iterations, since in t/p steps it can use the same number of list vectors for reduction, namely $p \cdot t/p$. Therefore, we expect the number of iterations for IdealListSieve to be a p-th fraction

of ListSieve.

Since in every iteration one single vector is sampled and appended to the list, the maximum list size will be in the order of magnitude of the iteration count. The runtime of the whole algorithm is depending on the number of iterations itself. Since we are performing p possible reductions instead of one in each iteration, the time needed for one iteration is supposed to increase a bit. So we expect the total runtime to increase a bit less than the number of iterations. Figure 7.1 shows the expected speedup factors of IdealListSieve compared to ListSieve.

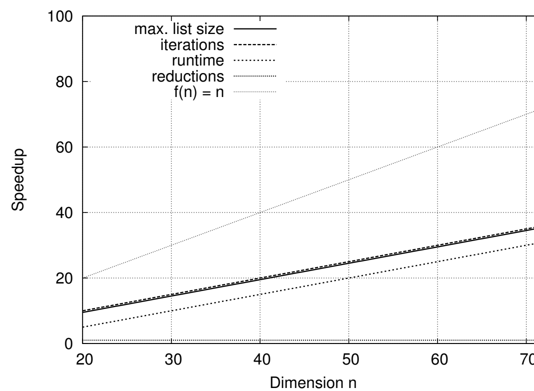


Figure 7.1: Predicted speedups for IdealListSieve compared to ListSieve. The values for the number of iterations I and the list size L are computed with $f(n) = n/2$. The number of reductions R is expected to be close to 1, the total runtime a bit less than I.

Recall that the speedups predicted in this section are asymptotic. They do not necessarily hold in practice, since we can only run experiments in dimensions $n \leq 80$. In the next section, we present experimental results comparing the two algorithms to show if our predictions hold in practice.

7.3 Experimental Results

The public implementation of [Vou10] (called `gsieve`) implements the GaussSieve algorithm. Based on this, we implemented ListSieve, IdealListSieve, and Ideal-GaussSieve. ListSieve is essentially the `gsieve` implementation without stack functionality. IdealListSieve uses the function `IdealListReduce` of Section 7.1 in addition. Both algorithms do not use perturbations. IdealGaussSieve implements GaussSieve with the additional function `IdealGaussReduce`. All implemented algorithms are published online [Sch11b].

Since we are using the NTL-library [Sho], it would be possible to implement a generic function `IdealReduce` for arbitrary field polynomials f. However, specializing on a particular class of polynomials allows some code improvements and leads to a huge speed-up in practice. Therefore, we have implemented three different functions, namely `AntiCyclicReduce`, `CyclicReduce`, and `CyclotomicReduce`. These functions can be used for sieving in anti-cyclic, cyclic, or prime cyclotomic lattices, respectively. Here we present experimental results for cyclic and prime cyclotomic lattices.

Pre-Reduction. We only apply LLL as pre-reduction, not BKZ. This is due to the fact that BKZ-reduction is too strong in small dimensions, and the sieving algorithms are not doing any work if BKZ already finds the shortest vector. Interestingly, we encountered in our experiments that the effect of pre-reduction for sieving is much less noticeable as in the enumeration case. To give more evidence to this, we generated 20 random bases, pre-reduced them with BKZ using different block sizes from 2 to 35 and measured the runtime of gsieve applied to the reduced matrices [Sch11a]. These experiments were performed on an Intel Core2 Duo 3GHz CPU. The results (for clearness reason some block sizes are omitted) are shown in Figures 7.2 and 7.3.

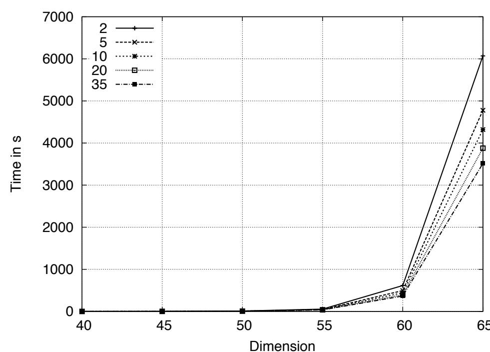


Figure 7.2: Runtime of gsieve on BKZ-reduced lattices with block size from 2 to 35.

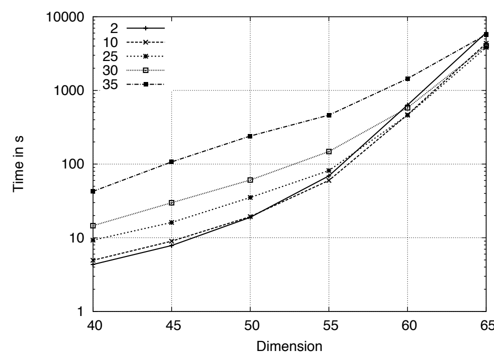


Figure 7.3: Sum of the runtimes of BKZ and gsieve.

It is noticeable that the runtime of gsieve decreases when the pre-reduction quality of the bases increases. In dimension 55 the runtime reduces from 55.8 seconds (blocksize 2) to 37.0 seconds (blocksize 35). Compared to enumeration, this effect

is much smaller for sieving algorithms. Measured with the fpLLL library [CPS], the runtime for enumeration decreases from 29451 seconds (blocksize 2) to 83.0 seconds (blocksize 35) for 55-dimensional lattices. Figure 7.3 presents the sum of the runtimes of gsieve and the BKZ pre-reduction. We can state that the bigger the dimension grows, the bigger the blocksize for pre-reduction is required to get the smallest possible runtime in total. In dimension $n = 65$, a blocksize of $\beta = 25$ gives the best results in our setting. In small dimension < 60 however LLL is sufficient to guarantee good runtimes for sieving.

IdealSieve Experiments. The results shown in the remainder of this section are average values of 10 random lattices in each dimension. All experiments were performed on an AMD Opteron (2.3GHz) quad core processor, using one single core. Since we do not know the length of a shortest vector in these lattices, we ran an SVP algorithm first to assess the norm. So we can stop our sieving algorithms as soon as we have found a vector of that norm. For cyclic and prime cyclotomic lattice we chose $p \in \{10, 12, 16, 18, 20, 22, 28, 30, 32\}$ and $n = 2p$. These are the values where $p + 1$ is prime, which is important for prime cyclotomic lattices. We chose these values for cyclic lattices as well in order to have results for both lattice types in the same dimensions. Our generator of the ideal lattices is included in Sage [S+] since version 4.5.2. The modulus q was fixed as 257. Naturally, the determinant of the lattices is q^p, i.e., 257^p. This value is comparable to the determinant of the SVP challenge lattices. For a second series of experiments, we generate cyclic and prime cyclotomic lattices with $n = 4p$. We choose $p \in \{6 \dots 15\}$ (cyclic) and $p \in \{6, 10, 12, 16\}$ (prime cyclotomic), q is again 257.

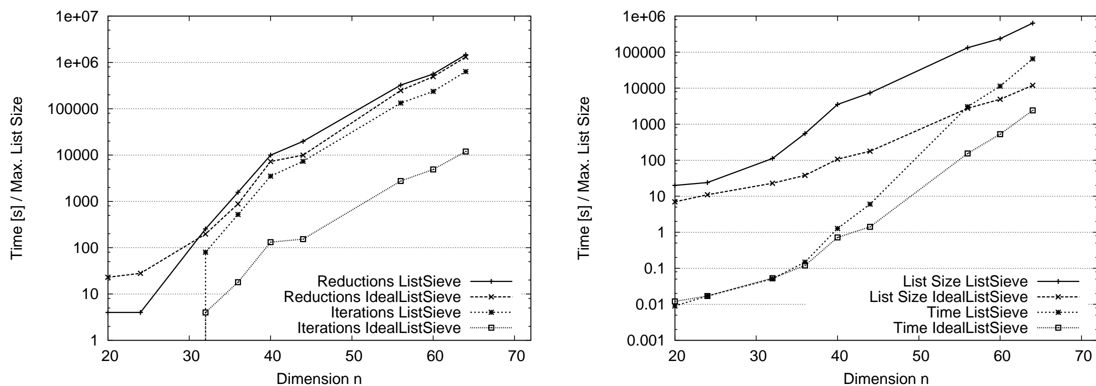


Figure 7.4: Results for cyclic lattices. Left: The number of reductions is comparable for ListSieve and IdealListSieve, whereas the number of iterations differs. Right: The maximum list size as well as the runtime goes down for IdealListSieve.

		Time	ListSize L	Iterations I	Reductions R
cyclic	ideal	$2^{0.51n-21.2}$	$2^{0.29n-5.3}$	$2^{0.31n-6.4}$	$2^{0.36n-2.3}$
	orig	$2^{0.67n-26.8}$	$2^{0.34n-2.5}$	$2^{0.35n-2.7}$	$2^{0.33n-0.6}$
cyclotomic	ideal	$2^{0.52n-19.7}$	$2^{0.29n-1.7}$	$2^{0.27n-1.1}$	$2^{0.32n+2.9}$
	orig	$2^{0.64n-24.0}$	$2^{0.30n+0.4}$	$2^{0.30n+0.5}$	$2^{0.29n+2.4}$

Table 7.1: Fitted values for cyclic and cyclotomic lattices with $n = 2p$.

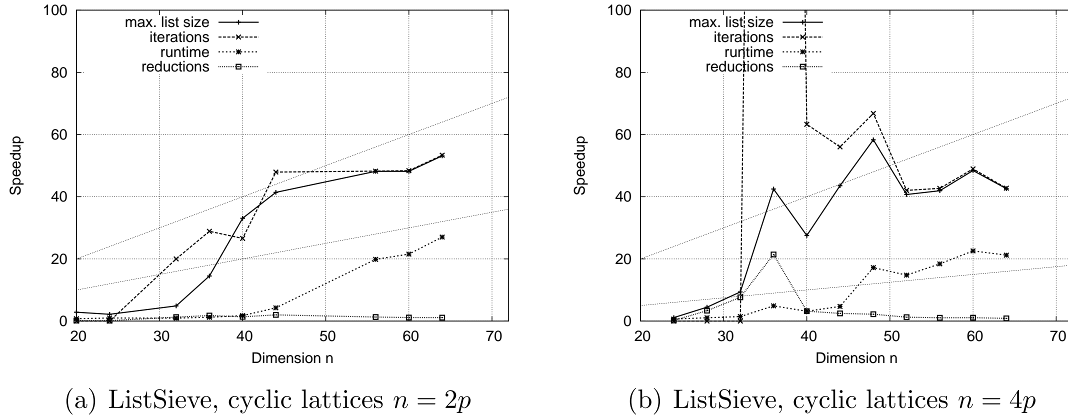
		Time	ListSize L	Iterations I	Reductions R
cyclic	ideal	$2^{0.57n-24.7}$	$2^{0.32n-6.4}$	$2^{0.43n-12.6}$	$2^{0.45n-7.3}$
	orig	$2^{0.67n-26.5}$	$2^{0.33n-1.5}$	$2^{0.33n-1.5}$	$2^{0.32n+0.5}$
cyclotomic	ideal	$2^{0.55n-21.9}$	$2^{0.30n-2.6}$	$2^{0.30n-2.9}$	$2^{0.34n+1.4}$
	orig	$2^{0.62n-23.3}$	$2^{0.29n+1.0}$	$2^{0.29n+1.0}$	$2^{0.28n+2.8}$

Table 7.2: Fitted values for cyclic and cyclotomic lattices with $n = 4p$.

Table 7.1 and Figure 7.4 show the results concerning R, I, L, and the runtime for cyclic lattices for $n = 2p$. The speedups for cyclic lattices are shown in Figure 7.5 and for prime cyclotomic lattices in Figure 7.6. Figure 7.5(a) shows the speedups of IdealListSieve compared to ListSieve. More exactly it shows the values for the number of iterations I, the maximum list size L, the runtime, and the total number of reductions R of ListSieve divided by the same values for IdealListSieve in the same lattices. Table 7.2 and Figure 7.5(b) show the same data for $n = 4p$. Figures 7.6(a) and (b) show the same data using cyclotomic lattices. All graphs contain a line for the identity function $f(n) = n$, and a line for $f(n) = n/2$ or $f(n) = n/4$, in order to ease comparison with the prediction of Figure 7.1.

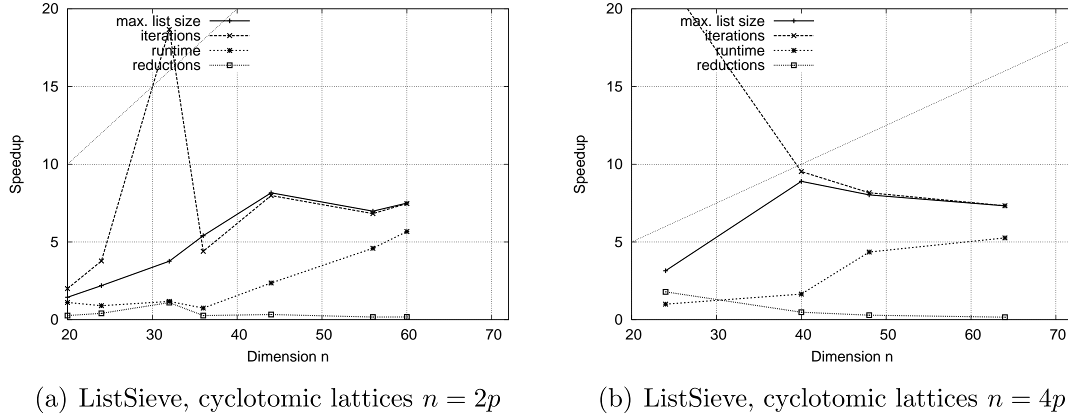
Interpretation. In small dimensions, the results are kind of abnormal. In some cases, the ideal lattice variant of an algorithm finds a shortest vector very quickly, which results in speedups of more than 100, e.g. in dimension $n = 36$ in Figure 7.5(b). Therefore, small dimensions of (say) less than 40 should be taken into account only carefully. Testing higher dimensions > 64 failed due to time reasons. Neither better pre-reduction nor searching for longer vectors helped decreasing the runtime noticeably.

A first thing that is apparent is that the number of reductions R stays nearly the same in all cases. With increasing dimension the speedup tends to 1 in all cases. Our assumption was reasonable, namely that the number of reductions required to



(a) ListSieve, cyclic lattices $n = 2p$ (b) ListSieve, cyclic lattices $n = 4p$

Figure 7.5: Speedup (original value divided by ideal variant value) of IdealListSieve compared to ListSieve, for cyclic lattices.



(a) ListSieve, cyclotomic lattices $n = 2p$ (b) ListSieve, cyclotomic lattices $n = 4p$

Figure 7.6: Speedup (original value divided by ideal variant value) of IdealListSieve compared to ListSieve, for cyclotomic lattices.

find a shortest vector is the same for the ideal and the non-ideal variant of ListSieve.

The higher the dimension gets, the closer the list size L and the iteration counter I get. Again this is how we expected the algorithms to behave. The runtime grows slower than the number of iterations. In dimension $n = 64$ for example, IdealListSieve finds a shortest vector 53 times faster than ListSieve.

Considering the number of iterations I, we see that our prediction was too pessimistic. For cyclic lattices, the speedups of IdealListSieve are higher than the predicted factor p; the factor is between p and n (for both $n = 2p$ and $n = 4p$). This implies that compared to the non-ideal variant, the same number of reductions is reached in less iterations. In other words, rotating a list vector $\mathbf{l}$ is better than sampling new vectors, for cyclic lattices. Unfortunately, it is not possible from our experiments to reasonably predict the asymptotic behaviour. Testing higher dimension is not possible due to time restrictions.

In case of prime cyclotomic lattices, the situation is different. The speedup of iterations is much smaller than for cyclic lattices (≤ 10 in all dimensions). The only difference between both experiments is the type of lattices. The rotations of prime cyclotomic lattices are less useful than those of cyclic lattices. A possible explanation for this is that rotating a vector of a cyclic lattice does not change the norm of the vector, whereas the rotations of prime cyclotomic lattice vectors have increased norms. The expansion factor of a ring $\mathbf{R}$ denotes the maximum "blow up" factor of a ring element when multiplied with a second one. More exactly, the expansion factor $\theta_2(\mathbf{f})$ of a polynomial $\mathbf{f} \in \mathbf{R}$ in the Euclidean norm is defined

$$\theta_2(\mathbf{f}) = \min\left\{c : \left\|\mathbf{a}x^i\right\|_2 \leq c\left\|\mathbf{a}\right\|_2 \forall \mathbf{a} \in \mathbb{Z}[x]/\langle\mathbf{f}\rangle \text{ for } 0 \leq i \leq p-1\right\} .$$

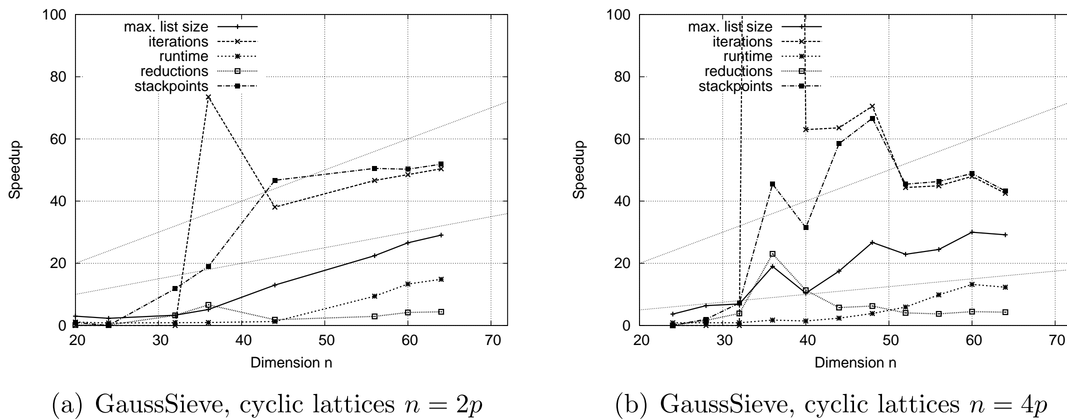
The expansion factor in the Euclidean norm is considered in [SSTX09]. For cyclic (and anti-cyclic) lattices it is easy to see that this factor equals 1. For prime cyclotomic lattices, it is

$$\theta_2(\mathbf{f}) = \sqrt{\frac{p+1}{2} + \sqrt{\left(\frac{p+1}{2}\right)^2 - 1}} \approx \sqrt{p}\,.$$

For a proof see Appendix A of [Sch11c]. So when the norm of the rotated list vectors $\mathbf{l}$ increases, this lowers the probability of a vector to be useful for reduction of the new sample. Therefore, compared to cyclic lattices, the speedup for iterations decreases. But still, sieving in prime cyclotomic lattices using the IdealListSieve is up to 10 times faster than in the original case.

IdealGaussSieve. We also performed experiments using the GaussSieve implementation of Voulgaris and our IdealGaussSieve version. Here we present our results

comparing GaussSieve to IdealGaussSieve. Figure 7.7 shows the experimental data. The speedup factors are comparable to those of IdealListSieve. The number of iterations I decreases by a factor of more than p, as well as the stackpoints (the points that are removed from the list and pushed on the stack, cf. [Sch11a]). A difference to ListSieve can be noted in the number of reductions. GaussSieve performs more reduction than IdealGaussSieve. With this, our original assumption that the same number of reductions holds for both sieving variants is no more true for GaussSieve and IdealGaussSieve.



(a) GaussSieve, cyclic lattices $n = 2p$ (b) GaussSieve, cyclic lattices $n = 4p$

Figure 7.7: Speedup (original value divided by ideal variant value) of IdealGaussSieve compared to GaussSieve, for cyclic lattices.

Anti-Cyclic Lattices. Lattices corresponding to ideals in the ring factored with $\mathbf{f}(x) = x^p + 1$ behave exactly as cyclic lattices. The algebra of both rings differs, but the algorithmic behaviour is exactly the same. In order to have the polynomial f irreducible, we choose $p \in \{2, 4, 8, 16, 32\}$ and $n = 2p$.

Ideal Enumeration. The enumeration algorithm for exhaustive search for shortest lattice vectors can also exploit the special structure of cyclic lattices. In the enumeration tree, linear combinations $\sum_{i=1}^{p} x_i \mathbf{b}_i$ in a specified search region are considered. For cyclic (and also anti-cyclic) lattices, a coefficient vector $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_p)$ and its rotations $\mathbf{rot}^i(\mathbf{x})$ for $i \in [p]$ specify the same vector. Therefore it is sufficient to enumerate the subtree predefined by one of the rotations. It is for example possible to choose only the coefficient vectors where the top coordinate $\mathbf{x}_p$ is the biggest entry, i.e., $\mathbf{x}_p = \max_i(\mathbf{x}_i)$. This would decrease the number of subtrees in the enumeration tree with a factor of up to p.

Unfortunately, this approach is only applicable if the input matrix has circular structure. When LLL-reducing the basis, usually the special structure of the matrix is destroyed. Therefore, when applying enumeration for ideal lattices one loses the possibility of pre-reducing the lattice. Even the symplectic LLL [GHGN06] does not maintain the circulant structure of the basis.

A second flaw of the ideal enumeration is that it is not applicable to prime cyclotomic lattices. For cyclic lattices it is easy to specify which rotations predefine the same vector, which does not work in the non-cyclic case.

8 Conclusion and Open Problems

We have presented parallel versions of the enumeration algorithm on multicore CPU (Chapter 3) and on NVIDIA graphics cards (Chapter 4). We have extended the latter algorithm by Extreme Pruning and have run it on GPU clusters as well as the Amazon EC2 cloud (Chapter 5). Further, we have presented a GPU version of Simple Sampling Reduction (Chapter 6) and an adaption of ListSieve as well as GaussSieve for ideal lattices (Chapter 7). We are the first to make use of the circular structure of ideal lattices to speed up SVP or similar algorithms. The CPU enumeration algorithm scales linearly in the number of used processor cores, sometimes even more, due to extra communication. The GPU algorithm requires some more sophisticated scheduling, and allows for good speedups of up to 6 compared to a single-core CPU. Using multiple GPUs, the speedup of our GPU enumeration scales linearly in the number of graphics cards. The experience of both parallelization techniques led to the extreme pruning enumeration of Chapter 5, which is the fastest published SVP solver of today.

Impact to Lattice-Based Cryptography

We have extended the practical limits of SVP algorithms and shown what is reachable with today's algorithms on powerful hardware. Our work points out that assumptions made in cryptographic practice are reasonable, i.e., that today SVP with small approximation factors is indeed intractable in the proposed dimensions. The hardness of breaking lattice-based cryptographic schemes is either based on the SIS problem or on the LWE problem. Both the SIS and the LWE problem can be proven to be as hard as approximate versions of SVP in lattices of a certain smaller

dimension (so-called *worst-case to average-case reduction*). A successful attacker of a cryptographic system is able to solve α-SVP for $\alpha = poly(n)$ in all lattices of a smaller dimension. As an example, the treeless signature scheme of [Lyu09] (using the smallest parameter set proposed) works in lattice dimension $n = 2048$. If the system is not strongly unforgeable, there exists a polynomial time algorithm that solves the shortest vector problem with approximation factor $\alpha \in \tilde{\mathcal{O}}(n^2)$ in *all* lattices of dimension $n = 512$ corresponding to ideals in the ring $\mathbb{Z}[x]/\langle x^n + 1\rangle$ [Lyu09, RS10].

Figure 8.1 summarizes the most important data for lattice-based cryptanalysis. The graph shows the approximation factors and the corresponding algorithm runtime. The approximation factor reached by BKZ-20 is not sufficient to threaten cryptosystems. On the other side, extreme enumeration takes far too much time, and reaches an approximation factor that is smaller than necessary. The figure includes expected runtimes for polynomial approximation factors (for worst-case instances) as well es the factor required to break the system of [LP11] practically (details given below).

The worst-case reduction gives a basement for security of lattice-based schemes. Nevertheless, it is important to consider direct attacks against the instantiated schemes as well. Practical attacks against cryptographic systems directly with enumeration algorithms is intractable in practice. But since there is no algorithm that reaches approximation factors polynomial in the lattice dimension, exact algorithms are the only algorithms finding short enough vectors. As an example, we consider the LWE-based cryptosystem of [LP11], and its "medium" parameter set $n = 256$, $q = 4093$, $s = 8.35$, $\epsilon \approx 0$. According to Figure 4 of [LP11], breaking the system using the stronger decoding attack requires a vector that reaches a Hermite factor of $\delta = 1.0052$ in a 640-dimensional q-ary lattice of determinant $q^n = 4093^{256} \approx 2^{3072}$. Our extreme enumeration implementation of Chapter 5 would run for 2^{195} years (cf. Conjecture 5.1). It would reach a vector of size less than or equal to $1.05 \cdot FM(\mathcal{L})$, which corresponds to a Hermite factor of $(1.05 \cdot \Gamma(n/2+1)^{1/640}/\sqrt{\pi})^{1/640} = 1.0029$ (in random lattices of determinant 2^{6400}). This factor is smaller than required. Adapting extreme enumeration to higher (more practical) approximation factors requires the effort of developing new bounding functions, which we leave for future work.

Real attacks on cryptosystems mostly apply approximation algorithms, like BKZ with exponential approximation factors. Our experience with BKZ shows that in higher blocksizes of about 50, enumeration takes more than 99% of the complete runtime. Therefore, our speedup of enumeration will directly speed up BKZ reduction, which in turn affects direct attacks on lattice-based cryptosystems. Implementations of BKZ including our improvements are left for future work.

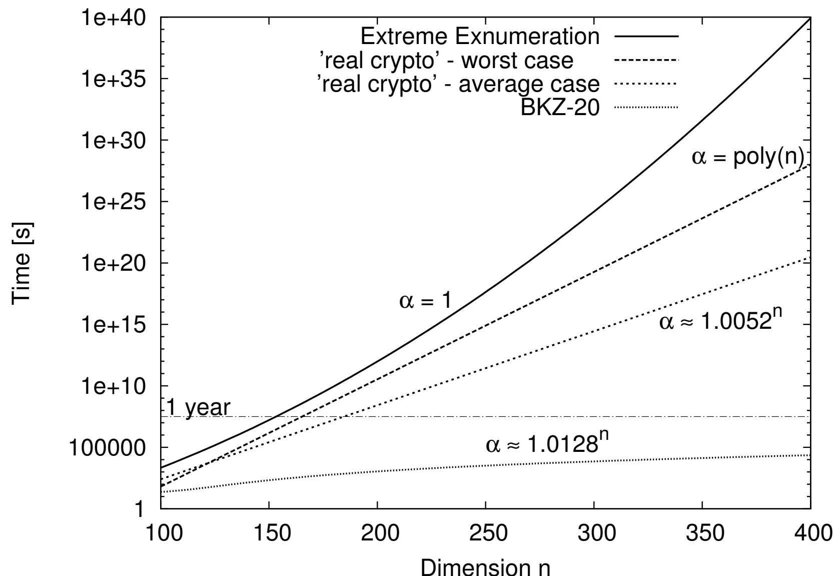


Figure 8.1: Expected runtime for BKZ-20 and extreme enumeration (Conjecture 5.1), including the reached approximation factors α. The runtime of BKZ-20 was fitted from Figure 12 of [GN08b] as $t_{\text{BKZ-20}}(n) = 0.00075n^3 - 0.2n^2 + 17.6n - 506$ seconds. Breaking lattice-based cryptosystems is at least as hard as solving the worst case SVP with $\alpha = poly(n)$. Practical attacks require approximation factor about $\alpha \approx 1.0052^n$ in dimension $n > 500$, like the LWE scheme of [LP11]. The runtime of worst and average case (two middle lines) in the picture is drafted.

To predict the runtime of BKZ with extreme pruning one needs to know the number of SVP subroutine calls that BKZ performs during reduction. The recent, theoretical work of [HPS11b] analyzes the Hermite factor that BKZ reaches when it is terminated after a fixed number of enumeration calls. More exactly, it states that after

$$C\frac{n^3}{\beta^2} \cdot \left(\log n + \log\log(\max_i \frac{\|\mathbf{b}_i\|}{(\det(\mathcal{L}))^{1/n}})\right)$$

calls of enumeration, the first vector output by the BKZ algorithm reaches a Hermite factor of

$$2(1+\frac{\beta}{4})^{\frac{n-1}{2(\beta-1)}+\frac{3}{2}} .$$

Here C is a constant factor, and we used the upper bound $\max_{i\leq\beta}(\gamma_i) \leq (1+\frac{\beta}{4})$ for the Hermite constants. For the LWE system parameter set used above, terminating BKZ after 2^{19} enumeration calls using blocksize $\beta = 100$ reaches a Hermite factor

of $\delta = 1.0255$, which is too big to threaten the system. Even blocksize $\beta = 500$, which is far from being practical, would only reach $\delta = 1.0174$. This analysis shows that this theoretical work is not applicable for practical attacks. The practical number of enumeration subroutine calls, that are required to reach a certain Hermite factor (and with this to break cryptographic systems), is unknown. To get evidence about this number requires a huge set of experiments using big blocksizes of extreme pruning enumeration inside BKZ. This implementation is as well future work.

We nevertheless try to estimate the runtime of BKZ with extreme pruning against the LWE system of [LP11]. It is known that the upper bound on the number of enumeration calls in BKZ, which is $(n\beta)^n$ [GN08b, HPS11b], is too huge. Therefore, we ran experiments with NTL's BKZ implementation, in order to estimate the number of enumerations calls in BKZ. Figure 8.2 shows the results. From Figure 8.2 we expect the number of iterations of BKZ to grow single exponential in the blocksize β and in the order of a degree three polynomial in the dimension n, extending the analysis of [GN08b, Figure 14]. Therefore, we assume a number of

$$iter(n, \beta) = 2^{a\beta+b} \cdot n^3$$

calls of the enumeration subroutine. Least-squares fitting outputs a parameter set $(a, b) = (0.214, -9.73)$. Therefore we assume that running BKZ with blocksize β in a n-dimensional lattice calls $iter(n, \beta) = 2^{0.214\cdot\beta-9.73} \cdot n^3$ enumeration oracles.

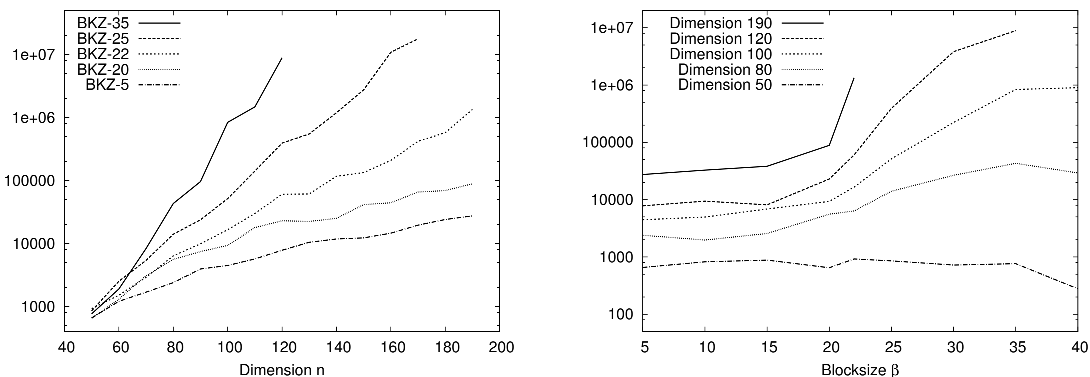


Figure 8.2: Number of calls to enumeration subroutine in BKZ, gained with the NTL implementation, for different blocksizes β and dimensions n. In order to estimate conservatively, we assume a polynomial growth of degree 3 ($\approx n^3$) in n (left) and single exponential growth ($\approx 2^{a\beta+b}$) in β (right).

In order to reach a Hermite factor of $\delta < 1.0052$, we experienced that blocksize $\beta = 60$ is a suitable value [SB10]. Using Conjecture 5.1, the runtime of a single

extremely pruned enumeration call is 1885 seconds, which gives a total runtime of BKZ-60 in a 640-dimensional lattice of

$$iter(640, 60) \cdot 1885 = 2^{31.1} \cdot 1885 \approx 2^{42} \text{ seconds},$$

which is about 130,000 years. Assuming a hardware cost of 1000 USD for a common CPU and one GTX-480 GPU, this corresponds to an attack cost of $2^{35.6}$ dollardays. This is the minimum effort an attacker has to spend using BKZ with extreme pruning enumeration in order to run a successful decoding attack against the LWE scheme of [LP11]. We will now apply the framework of [RS10, Rüc10] to compare this attack effort to known values.

The works of [BDR+96, ECR11] define different attacker classes, depending on their hardware and time effort, measured in dollardays (time $\times$ money). The classes range from "hacker" (willing to spend one day and 400 dollars $\hat{=}$ 400 dollardays) to a powerful "intelligence agency" (spending 300 Million USD and one year $\hat{=}$ 108 Billion dollardays). The authors of [RS10] complement an attacker class called "Lenstra" (40 Million dollardays), in order to allow a comparison between lattice-based attacks and the work of [Len05] concerning symmetric ciphers and classical asymmetric schemes. Following Lenstras "double Moore law", in order to estimate future hardware as well as algorithmic developments, an attackers compute capability grows by a factor of $2^{x \cdot 12/9}$ in x years. By the year y, the attacker Lenstra can spend $40M \cdot 2^{(y-2011) \cdot 12/9}$ dollardays, and in the year 2019 his effort of $2^{35.6}$ dollardays is sufficient to run the decoding attack against the LWE scheme. Following [Len05, Blu11] this corresponds to a symmetric security of 81 bit or a key length of 1523 bit for asymmetric keys (e.g. RSA). The framework of [RS10] uses BKZ without extreme pruning and estimates that a value of $\delta = 1.0052$ is not reachable for the Lenstra attacker until year 2054 ($\hat{=}$ 104 bit symmetric security). This shows that the exponential speedup of Extreme Enumeration compared to regular enumeration has a huge impact on the security of LWE, and since the analysis is applicable to all lattice-based systems the influence is the same in the whole field of research.

The hacker will not be able to break the LWE system before 2032. The powerful intelligence agency might break the scheme today already.

The asymptotic runtime of GaussSieve and ListSieve in ideal lattices remains unchanged, our improvements of Chapter 7 influence the practical runtime by a linear factor in the degree of the field polynomial. In BKZ the search for shortest vectors is performed in projected lattices. Since the projection of an ideal lattice is no more ideal, the ideal sieving variant is not applicable in BKZ. Solving α-SVP directly with sieving is intractable due to runtime reasons again. Therefore, our ideal lattice variants do not directly threaten cryptosystems based on ideal lattices.

Open Problems

Concerning extreme pruning enumeration, it is crucial to examine the influence of different bounding functions. The functions used today are gained by experimental results and can be improved. For different approximation factors, different bounding functions perform best. On graphics cards, the hardware performance of enumeration decreases with pruning. The development of bounding functions is henceforth also influenced by the hardware in use.

Our new versions of enumeration can be integrated into BKZ in order to speed up the fastest practical solver of approximate SVP. Extreme enumeration in BKZ using big blocksizes of more than 50 promises the best improvement. For solving α-SVP it is sufficient to terminate BKZ earlier, as shown in [HPS11b]. This early-termination approach together with extreme pruning will lead to the fastest BKZ possible at the moment.

There is no evidence about the influence of pre-reduction of input bases before running SVP algorithms. For Extreme Pruning we assumed that BKZ pre-reduction should take about 50% of the total runtime. Heuristics like this require more theoretical basis as well as more experimental work. While for enumeration algorithms, the influence of pre-reduction to runtime is known in parts, for sieving algorithms we have no information.

The Voronoi cell algorithm of [MV10a] has not been implemented so far. It would be interesting to see if it competes with enumeration algorithms. Developing heuristic improvements is an important open task. The asymptotic runtime of Voronoi cell and sieving algorithms is smaller than for enumeration. Still in small dimensions enumeration (combined with heuristics like Extreme Pruning) outperforms all concurrent algorithms. It is an interesting question to find out the crossover dimension of these different algorithms. As an example, the practical GaussSieve runtime $2^{0.52n}$ of [MV10b] and our ExtremeEnum runtime of Chapter 5 overlap at dimension $n = 708$, i.e., in dimensions higher than 708 GaussSieve runs faster than ExtremeEnum.

Dealing with floating point arithmetic is an important issue for implementations of lattice reduction and SVP algorithms. For graphics cards double precision computations are slow, and therefore should be avoided. Up to dimension 120, we did not experience problems with enumeration or sieving algorithms. For higher dimension, as well as for LLL and BKZ, using a multi-precision framework on special hardware is essential. It is future work to develop this kind of instrument, in order to render SVP solvers suitable for modern compute hardware.

To threaten cryptographic schemes one only needs to find approximate solutions

to SVP and comparable problems. It is to examine if algorithms that solve α-SVP with polynomial approximation factor can be developed, with polynomial or at least sub-exponential runtime. These algorithms are more interesting for cryptography than exact SVP solvers.

Concerning sieving, no implementation uses the improvement of [PS09]. It will improve the efficiency of the algorithm. Sieving algorithms that abandon perturbations (like the GaussSieve) are more efficient in practice, but few is known about this heuristic in theory. A parallel version of GaussSieve allows only for minor speedups [MS11]. Since the slower ListSieve is more suitable for parallelization, it is to examine if parallelization can overcome the disadvantages of ListSieve compared to GaussSieve.

The Voronoi cell algorithm as well as the probabilistic sieving algorithms require exponential storage, whereas the polynomial space algorithms like enumeration have runtime higher than single exponential. The main open problem in theory in the area of SVP algorithms is the development of a deterministic SVP algorithm running in single exponential time and only using polynomial space.

Bibliography

[ADL+08] Yuriy Arbitman, Gil Dogon, Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFTX: A proposal for the SHA-3 standard, 2008. In The First SHA-3 Candidate Conference. Cited on page 73.

[AJ08] Vikraman Arvind and Pushkar S. Joglekar. Some sieving algorithms for lattice problems. In *FSTTCS*, volume 2 of *LIPIcs*, pages 25–36. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008. Cited on page 16.

[Ajt98] Miklós Ajtai. The shortest vector problem in L_2 is NP-hard for randomized reductions. In *STOC 1998*, pages 10–19. ACM, 1998. Cited on pages 2 and 15.

[AKS01] Miklos Ajtai, Ravi Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *STOC 2001*, pages 601–610. ACM, 2001. Cited on pages 16 and 20.

[AMD06] Advanced Micro Devices. ATI CTM Guide. Technical report, ATI, 2006. Cited on page 22.

[BBD08] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors. *Post-Quantum Cryptography.* Springer, 2008. Cited on page 102.

[BCC+09] Daniel J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, and Bo-Yin Yang. ECM on graphics cards. In *EUROCRYPT 2010*, volume 5479 of *LNCS*, pages 483–501. Springer, 2009. Cited on pages 23 and 44.

[BDR+96] Matt Blaze, Whitfield Diffie, Ronald L. Rivest, Bruce Schneier, Tsutomu Shimomura, Eric Thompson, and Michael Wiener. Minimal key lengths for symmetric ciphers to provide adequate commercial security. A report by an ad hoc group of cryptographers and computer scientists, 1996. Cited on page 91.

[BL06] Johannes Buchmann and Christoph Ludwig. Practical lattice basis sampling reduction. In *ANTS 2006*, volume 4076 of *LNCS*, pages 222–237. Springer, 2006. Cited on pages 3, 14, 61, 62, and 63.

[BL09] Johannes Buchmann and Richard Lindner. Secure parameters for SWIFFT. In *INDOCRYPT 2009*, volume 5922 of *LNCS*, pages 1–17. Springer, 2009. Cited on page 62.

[BLR08] Johannes Buchmann, Richard Lindner, and Markus Rückert. Explicit hard instances of the shortest vector problem. In *PQCrypto 2008*, LNCS, pages 79–94. Springer, 2008. Cited on pages 21 and 62.

[Blu11] BlueKrypt. Cryptographic key length recommendation. `http://www.keylength.com`, 2011. Cited on page 91.

[BN09] Johannes Blömer and Stefanie Naewe. Sampling methods for shortest vectors, closest vectors and successive minima. *Theor. Comput. Sci.*, 410(18):1648–1665, 2009. Cited on pages 16 and 20.

[BS10] Joppe W. Bos and Deian Stefan. Performance analysis of the SHA-3 candidates on exotic multicore architectures. In *CHES 2010*, volume 6225 of *LNCS*, pages 279–293. Springer, 2010. Cited on page 23.

[BW09] Werner Backes and Susanne Wetzel. Parallel lattice basis reduction using a multi-threaded Schnorr-Euchner LLL algorithm. In *Euro-Par 2009*, volume 5704 of *LNCS*, pages 960–973. Springer, 2009. Cited on page 3.

[CE03] Henry Cohn and Noam Elkies. New upper bounds on sphere packings I. *Annals of Mathematics*, 157:689–714, 2003. Cited on page 13.

[CIKL05] Debra L. Cook, John Ioannidis, Angelos D. Keromytis, and Jake Luck. Cryptographics: Secret key cryptography using graphics cards. In *CT-RSA 2005*, volume 3376 of *LNCS*, pages 334–350. Springer, 2005. Cited on page 23.

[CN99] Jin-Yi Cai and Ajay Nerurkar. Approximating the SVP to within a factor $(1+1/\dim^{\epsilon})$ is NP-hard under randomized reductions. *J. Comput. Syst. Sci.*, 59(2):221–239, 1999. Preliminary version in CCC 1998. Cited on page 2.

[CPS] David Cadé, Xavier Pujol, and Damien Stehlé. fpLLL - a floating point LLL implementation. Available at Damien Stehlé's homepage

at école normale supérieure de Lyon, http://perso.ens-lyon.fr/damien.stehle/english.html. Cited on pages 21, 41, and 80.

[csc] Center for scientific computing Frankfurt - CSC, Goethe-Universität Frankfurt. http://csc.uni-frankfurt.de/. Cited on page 48.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI 2004*, pages 137–150. 2004. Cited on page 48.

[DHPS10] Jérémie Detrey, Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Accelerating lattice reduction with FPGAs. In *LATINCRYPT 2010*, volume 6212 of *LNCS*, pages 124–143. Springer, 2010. Cited on page 3.

[DPV11] Daniel Dadush, Chris Peikert, and Santosh Vempala. Enumerative algorithms for lattice problems in any norm via m-ellipsoid coverings. In *FOCS 2011*. IEEE Computer Society, 2011. To appear. Cited on page 16.

[DS10] Özgür Dagdelen and Michael Schneider. Parallel enumeration of shortest lattice vectors. In *Euro-Par 2010*, volume 6272 of *LNCS*, pages 211–222. Springer, 2010. Cited on pages 4 and 27.

[ECR11] ECRYPT2. Yearly report on algorithms and keysizes (2010-2011) - Report D.SPA.17, 2011. Available at http://www.ecrypt.eu.org/documents/D.SPA.17.pdf. Cited on page 91.

[Fil02] Bartol Filipović. *Implementierung der Gitterbasenreduktion in Segmenten*. Masters thesis, Goethe-Universität Frankfurt, 2002. Cited on page 21.

[Fle07] Sebastian Fleissner. GPU-accelerated montgomery exponentiation. In *ICCS 2007*, volume 4487 of *LNCS*, pages 213–220. Springer, 2007. Cited on page 23.

[FP83] U. Fincke and Michael Pohst. A procedure for determining algebraic integers of given norm. In *EUROCAL 1983*, volume 162 of *LNCS*, pages 194–202. Springer, 1983. Cited on page 16.

[FP85] U. Fincke and Michael Pohst. Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, 44(170):463–471, 1985. Cited on page 16.

[GHGN06] Nicolas Gama, Nick Howgrave-Graham, and Phong Q. Nguyen. Symplectic lattice reduction and NTRU. In *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 233–253. Springer, 2006. Cited on page 85.

[GM03] Daniel Goldstein and Andrew Mayer. On the equidistribution of Hecke points. *Forum Mathematicum 2003, 15:2*, pages 165–189, 2003. Cited on pages 1, 10, 32, and 41.

[GN05] Zhan Guo and P. Nilsson. VLSI architecture of the soft-output sphere decoder for MIMO systems. In *Midwest Symposium on Circuits and Systems 2005*, volume 2, pages 1195–1198. 2005. Cited on page 3.

[GN08a] Nicolas Gama and Phong Q. Nguyen. Finding short lattice vectors within Mordell's inequality. In *STOC 2008*, pages 207–216. ACM, 2008. Cited on page 14.

[GN08b] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 31–51. Springer, 2008. Cited on pages 10, 14, 15, 41, 67, 89, and 90.

[GNR10] Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 257–278. Springer, 2010. Cited on pages 2, 9, 16, 19, 47, 50, 52, 53, 55, and 58.

[GS10] Nicolas Gama and Michael Schneider. SVP Challenge, 2010. Available at `http://www.latticechallenge.org/svp-challenge`. Cited on pages 9, 15, and 67.

[GS11] Norman Göttert and Michael Schneider. gpussr v0.1 - Sampling for short lattice vectors on graphics cards, 2011. Available at Michael Schneider's homepage at TU Darmstadt `http://www.cdc.informatik.tu-darmstadt.de/mitarbeiter/mischnei.html`. Cited on pages 21 and 67.

[Her50] Charles Hermite. Extraits de lettres de M. Hermite à M. Jacobi sur différents objets de la théorie de nombres, deuxieme lettre. *Journal für die Reine und Angewandte Mathematik*, 40:279–290, 1850. Cited on page 1.

[Her05] Charles Hermite. Œuvres de Charles Hermite publiées sous les auspices de l'académie des sciences, par Emile Picard. Gauthiers-Villars, 1905. Cited on page 1.

[HGS01] Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Des. Codes Cryptography*, 23(3):283–290, 2001. Cited on page 3.

[HKS11] Jens Hermans, Po-Chun Kuo, and Michael Schneider. gpuenum - lattice enumeration on graphics cards (Heidelberg version), 2011. Available at `http://homes.esat.kuleuven.be/~jhermans/gpuenum/`. Cited on pages 21, 41, and 55.

[HPS11a] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Algorithms for the shortest and closest lattice vector problems. In *IWCC 2011*, volume 6639 of *LNCS*, pages 159–190. Springer, 2011. Cited on pages 15, 20, and 22.

[HPS11b] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Analyzing blockwise lattice algorithms using dynamical systems. In *CRYPTO 2011*, volume 6841 of *LNCS*, pages 447–464. Springer, 2011. Cited on pages 89, 90, and 92.

[HS07] Guillaume Hanrot and Damien Stehlé. Improved analysis of Kannan's shortest lattice vector algorithm. In *CRYPTO 2007*, volume 4622 of *LNCS*, pages 170–186. Springer, 2007. Cited on pages 9, 16, 19, and 43.

[HSB+10] Jens Hermans, Michael Schneider, Johannes Buchmann, Frederik Vercauteren, and Bart Preneel. Parallel shortest lattice vector enumeration on graphics cards. In *AFRICACRYPT 2010*, volume 6055 of *LNCS*, pages 52–68. Springer, 2010. Cited on pages 4 and 37.

[HW07] Owen Harrison and John Waldron. AES encryption implementation and analysis on commodity graphics processing units. In *CHES 2007*, volume 4727 of *LNCS*, pages 209–226. Springer, 2007. Cited on page 23.

[Kai94] Michael Kaib. *Gitterbasenreduktion für beliebige Normen*. Ph.D. thesis, Goethe-Universität Frankfurt, 1994. Cited on page 14.

[Kan83] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *STOC 1983*, pages 193–206. ACM, 1983. Cited on page 16.

[Kan87] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Math. Oper. Res.*, 12:415–440, 1987. Cited on page 16.

[KH10] David B. Kirk and Wen-Mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 1 edition, 2010. Cited on page 22.

[Kho05] Subhash Khot. Hardness of approximating the shortest vector problem in lattices. *J. ACM*, 52(5):789–808, 2005. Cited on page 2.

[Kho10] Subhash Khot. Inapproximability results for computational problems on lattices. In Nguyen and Vallée [NV10], pages 453–473. Cited on pages 2 and 15.

[Kle00] Philip N. Klein. Finding the closest lattice vector when it's unusually close. In *SODA 2000*, pages 937–941. ACM, 2000. Cited on pages 10 and 75.

[KLPS11] Thorsten Kleinjung, Arjen K. Lenstra, Dan Page, and Nigel P. Smart. Using the cloud to determine key strengths. Cryptology ePrint Archive, Report 2011/254, 2011. `http://eprint.iacr.org/`. Cited on page 49.

[KS01] Henrik Koy and Claus-Peter Schnorr. Segment LLL-reduction of lattice bases. In *CaLC*, volume 2146 of *LNCS*, pages 67–80. Springer, 2001. Cited on page 14.

[KSD+11] Po-Chun Kuo, Michael Schneider, Özgür Dagdelen, Jan Reichelt, Johannes Buchmann, Chen-Mou Cheng, and Bo-Yin Yang. Extreme enumeration on GPU and in clouds. In *CHES 2011*, volume 6917 of *LNCS*, pages 176–191. Springer, 2011. Cited on pages 5, 15, and 48.

[KZ73] A. Korkine and G. Zolotareff. Sur les formes quadratiques. *Mathematische Annalen*, 6:366–389, 1873. Cited on page 1.

[Len05] Arjen Lenstra. Key lengths. In Hossein Bidgoli, editor, *Handbook of Information Security*. Wiley, 2005. Cited on pages 47, 49, and 91.

[LJS90] J. C. Lagarias, Hendrik W. Lenstra Jr., and Claus-Peter Schnorr. Korkin-zolotarev bases and successive minima of a lattice and its reciprocal lattice. *Combinatorica*, 10(4):333–348, 1990. Cited on page 13.

[LLL82] Arjen Lenstra, Hendrik Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 4:515–534, 1982. Cited on pages 2, 13, 24, and 76.

[LO85] J. C. Lagarias and Andrew M. Odlyzko. Solving low-density subset sum problems. *Journal of the ACM*, 32(1):229–246, 1985. Cited on page 3.

[LP11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In *CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer, 2011. Cited on pages 15, 88, 89, 90, and 91.

[Lud05] Christoph Ludwig. *Practical Lattice Basis Sampling Reduction.* Ph.D. thesis, TU Darmstadt, 2005. Cited on pages 3, 21, 62, and 63.

[Lyu08] Vadim Lyubashevsky. *Towards Practical Lattice-Based Cryptography.* Ph.D. thesis, University of California, San Diego, 2008. Cited on pages 11 and 12.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 598–616. Springer, 2009. Cited on pages 73, 74, and 88.

[Man07] Svetlin A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *ICSPC 2007*, pages 65–68. IEEE, 2007. Cited on page 23.

[May10] Alexander May. Using LLL-reduction for solving RSA and factorization problems. In Nguyen and Vallée [NV10], pages 315–348. Cited on page 3.

[MG02] Daniele Micciancio and Shafi Goldwasser. *Complexity of Lattice Problems: a cryptographic perspective.* Kluwer Academic Publishers, 2002. Cited on pages 14, 15, and 22.

[Mic00] Daniele Micciancio. The shortest vector in a lattice is hard to approximate to within some constant. *SIAM J. Comput.*, 30(6):2008–2035, 2000. Cited on page 2.

[Min96] Hermann Minkowski. *Geometrie der Zahlen.* Teubner-Verlag, 1896. Cited on page 1.

[MO90] J. E. Mazo and Andrew M. Odlyzko. Lattice points in high-dimensional spheres. *Monatshefte für Mathematik*, 110:47–61, 1990. Cited on page 9.

[MPS07] Andrew Moss, Dan Page, and Nigel P. Smart. Toward acceleration of RSA using 3D graphics hardware. In *IMA International Conference 2007*, volume 4887 of *LNCS*, pages 364–383. Springer, 2007. Cited on page 23.

[MR08] Daniele Micciancio and Oded Regev. Lattice-based cryptography. In Bernstein et al. [BBD08], pages 147–191. Cited on pages 14, 22, and 73.

[MS01] Alexander May and Joseph H. Silverman. Dimension reduction methods for convolution modular lattices. In *CaLC*, volume 2146 of *LNCS*, pages 110–125. Springer, 2001. Cited on page 74.

[MS11] Benjamin Milde and Michael Schneider. A parallel implementation of GaussSieve for the shortest vector problem in lattices. In *PaCT 2011*, volume 6873 of *LNCS*, pages 452–458. Springer, 2011. Cited on pages 3 and 93.

[MV10a] Daniele Micciancio and Panagiotis Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations. In *STOC 2010*, pages 351–358. ACM, 2010. Cited on pages 2, 16, and 92.

[MV10b] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *SODA 2010*, pages 1468–1480. ACM/SIAM, 2010. Cited on pages 16, 20, 21, 73, 74, 75, 76, 77, and 92.

[NS01] P. Q. Nguyen and J. Stern. The two faces of lattices in cryptology. In *Cryptography and Lattices Conference 2001*, volume 2146 of *LNCS*, pages 146–180. Springer, 2001. Cited on page 14.

[NS05] Phong Q. Nguyen and Damien Stehlé. Floating-point LLL revisited. In *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 215–233. Springer, 2005. Cited on pages 13, 21, and 24.

[NS06] Phong Q. Nguyen and Damien Stehlé. LLL on the average. In *ANTS 2006*, volume 4076 of *LNCS*, pages 238–256. Springer, 2006. Cited on pages 10, 14, 24, and 41.

[NS09] Phong Q. Nguyen and Damien Stehlé. An LLL algorithm with quadratic complexity. *SIAM J. Comput.*, 39(3):874–903, 2009. Cited on pages 13, 21, and 22.

[NSV11] Andrew Novocin, Damien Stehlé, and Gilles Villard. An LLL-reduction algorithm with quasi-linear time complexity: extended abstract. In *STOC*, pages 403–412. ACM, 2011. Cited on page 13.

[NV08] Phong Q. Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *J. of Mathematical Cryptology*, 2(2), 2008. Cited on pages 16 and 20.

[NV10] Phong Q. Nguyen and Brigitte Vallée, editors. *The LLL Algorithm - Survey and Applications*. Springer, 2010. Cited on pages 14, 22, 100, 101, and 104.

[NVI07a] NVIDIA. Compute Unified Device Architecture Programming Guide. Technical report, NVIDIA, 2007. Cited on page 22.

[NVI07b] NVIDIA. CUBLAS Library, 2007. Cited on page 22.

[Odl89] Andrew M. Odlyzko. The rise and fall of knapsack cryptosystems. In *Cryptology and Computational Number Theory, Proceedings of Symposia in Applied Mathematics*, volume 42, pages 75–88. American Mathematical Society, 1989. Cited on page 3.

[PS08] Xavier Pujol and Damien Stehlé. Rigorous and efficient short lattice vectors enumeration. In *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 390–405. Springer, 2008. Cited on pages 16, 22, and 32.

[PS09] Xavier Pujol and Damien Stehlé. Solving the shortest lattice vector problem in time $2^{2.465n}$. Cryptology ePrint Archive, Report 2009/605, 2009. `http://eprint.iacr.org/`. Cited on pages 20 and 93.

[Puj06] Xavier Pujol. LatEnum - Library to solve the shortest and the closest vector problem on integer lattices, using floating point arithmetic, 2006. `http://perso.ens-lyon.fr/xavier.pujol/index_en.html`. Cited on pages 3 and 21.

[Puj08] Xavier Pujol. *Recherche efficace de vecteur court dans un réseau euclidien*. Masters thesis, ENS Lyon, 2008. Cited on pages 3 and 41.

[Reg10] Oded Regev. On the complexity of lattice problems with polynomial approximation factors. In Nguyen and Vallée [NV10], pages 475–496. Cited on page 15.

[Rit97] Harald Ritter. *Aufzählung von kurzen Gitterverktoren in allgemeiner Norm*. Ph.D. thesis, Goethe-Universität Frankfurt, 1997. Cited on page 14.

[RS10] Markus Rückert and Michael Schneider. Selecting secure parameters for lattice-based cryptography. Cryptology ePrint Archive, Report 2010/137, 2010. `http://eprint.iacr.org/`. Cited on pages 88 and 91.

[Rüc10] Markus Rückert. *Lattice-based Signature Schemes with Additional Features*. Ph.D. thesis, TU Darmstadt, 2010. Cited on page 91.

[S+] W. A. Stein et al. *Sage Mathematics Software*. The Sage Development Team. `http://www.sagemath.org`. Cited on pages 22 and 80.

[SB10] Michael Schneider and Johannes Buchmann. Extended lattice reduction experiments using the BKZ algorithm. In *Sicherheit 2010*, volume 170 of *LNI*, pages 241–252. GI, 2010. Cited on page 90.

[SBB08] Christoph Studer, Andreas Burg, and Helmut Bölcskei. Soft-output sphere decoding: algorithms and VLSI implementation. *IEEE Journal on Selected Areas in Communications*, 26(2):290–300, 2008. Cited on page 3.

[Sch87] Claus-Peter Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theor. Comput. Sci.*, 53:201–224, 1987. Cited on page 13.

[Sch94] Claus-Peter Schnorr. Block reduced lattice bases and successive minima. *Combinatorics, Probability & Computing*, 3:507–522, 1994. Cited on page 13.

[Sch03] Claus-Peter Schnorr. Lattice reduction by random sampling and birthday methods. In *STACS 2003*, volume 2607 of *LNCS*, pages 146–156. Springer, 2003. Cited on pages 3, 14, 62, and 63.

[Sch11a] Michael Schneider. Analysis of Gauss-sieve for solving the shortest vector problem in lattices. In *WALCOM 2011*, volume 6552 of *LNCS*, pages 89–97. Springer, 2011. Cited on pages 15, 74, 79, and 84.

[Sch11b] Michael Schneider. Idealsieve v01gamma - Sieving for shortest vectors in ideal and cyclic lattices, 2011. Available at Michael Schneider's homepage at TU Darmstadt `http://www.cdc.informatik.tu-darmstadt.de/mitarbeiter/mischnei.html`. Cited on pages 22 and 78.

[Sch11c] Michael Schneider. Sieving for shortest vectors in ideal lattices. Cryptology ePrint Archive, Report 2011/458, 2011. `http://eprint.iacr.org/`. Cited on pages 5, 74, and 83.

[SE94] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66:181–199, 1994. Cited on pages 2, 13, 16, 22, and 40.

[SG08] Robert Szerwinski and Tim Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In *CHES 2008*, volume 5154 of *LNCS*, pages 79–99. Springer, 2008. Cited on page 23.

[SG11] Michael Schneider and Norman Göttert. Random sampling for short lattice vectors on graphics cards. In *CHES 2011*, volume 6917 of *LNCS*, pages 160–175. Springer, 2011. Cited on pages 5, 62, and 66.

[SH95] Claus-Peter Schnorr and Horst Helmut Hörner. Attacking the Chor-Rivest cryptosystem by improved lattice reduction. In *EUROCRYPT 1995*, volume 921 of *LNCS*, pages 1–12. Springer, 1995. Cited on pages 19 and 50.

[Sho] Victor Shoup. Number theory library (NTL) for C++. `http://www.shoup.net/ntl/`. Cited on pages 21, 50, 67, and 79.

[SSTX09] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In *ASIACRYPT 2009*, volume 5912 of *LNCS*. Springer, 2009. Cited on page 83.

[Val06] Frank Vallentin. SHVEC: shortest and closest vectors in lattices, 2006. `http://fma2.math.uni-magdeburg.de/~latgeo/`. Cited on page 21.

[vEB81] Peter van Emde Boas. Another NP-complete problem and the complexity of computing short vectors in a lattice. Technical Report 81-04, Mathematisch Instituut, Universiteit van Amsterdam, 1981. Cited on page 15.

[Vil92] Gilles Villard. Parallel lattice basis reduction. In *ISSAC 1992*, pages 269–277. ACM, 1992. Cited on page 3.

[Vor08] Georgy Feodosevich Voronoi. Nouvelles applications des paramètres continus à la théorie de formes quadratiques. *Journal für die reine und angewandte Mathematik*, 134:198–287, 1908. Cited on page 1.

[Vou10] Panagiotis Voulgaris. Gauss Sieve beta 0.1, 2010. Available at Panagiotis Voulgaris' homepage at the University of California, San Diego, `http://cseweb.ucsd.edu/~pvoulgar/impl.html`. Cited on pages 20, 21, 76, and 78.

[WLTB10] Xiaoyun Wang, Mingjie Liu, Chengliang Tian, and Jingguo Bi. Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. Cryptology ePrint Archive, Report 2010/647, 2010. `http://eprint.iacr.org/`. Cited on page 20.

Index

A

Algorithms for SVP 15
Amazon EC2 47
Anti-Cyclic Lattices 12, 78
Approximate SVP 15

B

Basis 7
Basis Transformation 8
BKZ Algorithm 13
  Iterations 90
BKZ-reduced Basis 13
Bounding Function 20, 50

C

Cloud Computing 48, 52
Cryptographic Impact 87
Cyclic Lattices 11, 78

D

Determinant 8

E

Efficiency 33
Encryption 88
Enumeration 16, 27, 37, 47
Extreme Pruning 19, 47

F

First Minimum 9
fpLLL 21

G

Gauss Heuristic 9
GaussSieve 20, 73
GPU Computation 22, 37, 53, 61
  Applications 22
  Instruction Set 23
  Memory Model 23
  Programming Model 23
gpuenum 21
gpussr 21
Gram-Schmidt Orthogonalization 9
gsieve 21

H

Hermite Constant 13
Hermite Shortest Vector Problem 15
HKZ-reduced Basis 13

I

Ideal Lattice 10, 73
Ideal Lattice Enumeration 84
IdealGaussSieve 76
IdealListSieve 74
Implementations 21

L

Lattice 7
- Basis 7
- Determinant 8
- First Minimum . . . . . 8 f.
- Reduction 12
- Successive Minima 8

ListSieve 20, 73
LLL Algorithm 13
LLL-reduced Basis 12
LWE Encryption 88

M

MapReduce 48, 55
Multicore CPUs 24, 32

N

NP-Hardness 15
NTL 21

O

Open Problems 92
Orthogonal Projection 9

P

Parallelepiped 8
Parallelization 22
Pricing Function 59
Prime Cyclotomic Lattices 12, 78
Projected Lattice 10
Pruning 19

R

Random Lattice 10
Random Sampling Reduction . . 14, 61

S

Shortest Vector Problem (SVP) 14
Sieving 20, 73
Simple Sampling Reduction 14, 61
Size-reduced Basis 12
Successive Minima 8
SVP Challenge 15

U

Unimodular Matrix 8

V

Voronoi Cell Algorithm 16

Notation

$d \in \mathbb{Z}$	Dimension of the embedding space
$n \in \mathbb{Z}, n \leq d$	Dimension of lattices
$\mathbf{B}$	Lattice basis
$\mathcal{L}(\mathbf{B})$	Lattice generated by basis $\mathbf{B}$
$\mathbf{b}_i$ for $1 \leq i \leq n$	Basis vector
$\lambda_1(\mathcal{L}(\mathbf{B}))$	Length of a shortest, non-zero vector of the lattice $\mathcal{L}(\mathbf{B})$
$\det(\mathcal{L}(\mathbf{B}))$	Determinant of the lattice $\mathcal{L}(\mathbf{B})$
$\left\Vert\mathbf{b}\right\Vert, \left\Vert\mathbf{b}\right\Vert_2$	Euclidean norm of vector $\mathbf{b}$
$\left\Vert\mathbf{b}\right\Vert_\infty$	Maximum norm of vector $\mathbf{b}$
$\lfloor x \rceil$	Rounding to the nearest integer, $\lceil x - 0.5 \rceil$
$\mathbf{I}_t$	$t \times t$ identity matrix
$\mathbf{0}_t$, $\mathbf{1}_t$	t-dimensional row vector consisting of zero and one entries
$\log(x)$, $\log_2(x)$	logarithm of x to base 2
$\log_e(x)$, $\log_{10}(x)$	logarithm of x to base e and 10
$[t]$	index set $\{0, \ldots, t-1\}$
$\mathbf{B}^*$	Gram-Schmidt orthogonalization of basis $\mathbf{B}$
$FM(\mathcal{L})$	Estimation of the first minimum of $\mathcal{L}$
$\langle \mathbf{x} \mid \mathbf{y} \rangle$	Scalar product of vectors $\mathbf{x}$ and $\mathbf{v}$

Wissenschaftlicher Werdegang

Mai 2008 - November 2011

Wissenschaftlicher Mitarbeiter in der Arbeitsgruppe von Professor Johannes Buchmann, Fachbereich Informatik, Fachgebiet Theoretische Informatik - Kryptographie und Computeralgebra an der Technischen Universität Darmstadt

August 2010 - November 2011

Wissenschaftlicher Mitarbeiter im Forschungsprojekt BU 630/23-1: *Gitterreduktion mit Grafikkarten* der Deutschen Forschungsgemeinschaft (DFG)

Mai 2008 - Juli 2010

Promotions-Stipendium der Technischen Universität Darmstadt

Oktober 2003 - April 2008

Studium der Informatik (Diplom) an der Technischen Universität Darmstadt

April 2003 - April 2008

Studium der Mathematik (Diplom) an der Technischen Universität Darmstadt

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit - abgesehen von den in ihr ausdrücklich genannten Hilfen - selbständig verfasst habe.

Darmstadt, November 2011 ______________________________