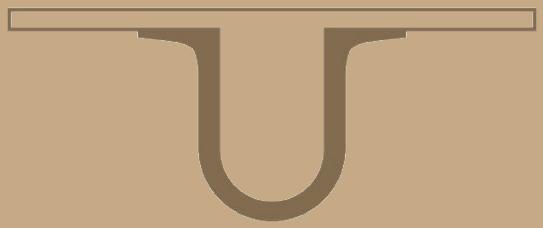
UNIVERSIDADE D
COIMBRA



Filipe António Emílio Cabeleira

ON THE ACCELERATION OF LATTICE BASED ALGORITHMS FOR POST-QUANTUM CRYPTOSYSTEMS

Dissertação no âmbito do Mestrado Integrado em Engenharia Electrotécnica e de Computadores, ramo de Computadores, orientada pelo Professor Doutor Gabriel Falcão Paiva Fernandes e pelo Doutor Artur Miguel Matos Mariano e apresentada ao Departamento de Engenharia Electrotécnica e de Computadores.

Fevereiro de 2019

Acknowledgments

I would like to thank my supervisors Professor Gabriel Falcão and Artur Mariano not only for the knowledge they shared, but also for their availability to help, guidance and insight, all of which were instrumental to the work herein presented.

This work has been supported by Instituto de Telecomunicações and Fundação para a Ciência e a Tecnologia (FCT), who sponsored the research where the findings presented in this dissertation were initially developed, under grant UID/EEA/50008/2013.

To Professor Erik Agrell (Chalmers University of Technology), for his availability and help, who kindly replied to my emails about his algorithm, allowing me to track down and fix a bug with my initial implementation.

To Manuel Rodrigues, that shared his time and his expertise with GPU architectures, allowing further improvements of the CUDA implementation.

To my friends, both old and new, for many shared meals, for those long talks and, above all, for their companionship – you certainly made this journey all the more remarkable.

And last, but certainly not least, I would like to thank my parents, from the bottom of my heart, for their unwavering support throughout my life – your sacrifices made all this possible and, for that and so much more, I will be eternally grateful.

To each and every one of you – Thank you.

Abstract

Classical cryptographic systems, such as RSA and ElGamal, have been found to be vulnerable in the presence of a sufficiently powerful quantum computer. In light of this discovery, the scientific community shifted its attention to the design and study of quantum-immune alternatives. Lattice-based systems are one type of cryptosystem that was proved to be resilient against attacks from quantum computers. These cryptosystems base their security on hard mathematical problems, like the Shortest Vector Problem, Closest Vector Problem, and others. As such, the study of algorithms designed to solve these problems – also known as attacks – is paramount in order to better understand them, and to better select the parameters for the construction of these cryptosystems.

This dissertation focuses on a specific type of attack: Voronoi cell-based, which are often mentioned in the literature as being impractical, but very little research has actually been conducted, especially when compared to other types. In this dissertation, several optimizations of a Voronoi cell-based algorithm are presented, with the goal of reducing both the execution time and the memory requirements of the implementations. In order to harness the full computational power available in modern computers, parallel CPU, parallel GPU and heterogeneous CPU + GPU versions are proposed, reporting linear speedups for the first (and some benefit from Hyper-Threading), and yielding maximum speedups of $13.37\times$ and $15.03\times$, for the GPU and heterogeneous versions, respectively, compared to the baseline, sequential CPU version. Additionally, several heuristic variants with parallel CPU implementations are proposed to accelerate the computation of the solution of the Shortest Vector Problem, with the best pruning strategy devised achieving speedups of up to $256.51\times$, using 8 threads (compared to the sequential, non-pruned implementation), while computing the correct solution for 99.84% of the tested lattice bases.

Finally, conducted in the scope of this dissertation, the integration of all work into the Lattice Unified Set of Algorithms (LUSA) library – a project that aims to provide public access to simple and fast implementations of lattice-related algorithms.

The results obtained in this dissertation show that, while far from state of the art dimensions, it is still possible to greatly accelerate Voronoi cell-based algorithms, maintaining very high success rates.

Keywords

Cryptography; Lattice; Voronoi cell; High Performance Computing; Parallelism; Multi-threading; GPU; Heterogeneous computing

Resumo

Sistemas criptográficos clássicos, como o RSA e o ElGamal, mostraram ser vulneráveis na presença de um computador quântico suficientemente poderoso. Dada esta descoberta, a comunidade científica focou a sua atenção no desenvolvimento e estudo de alternativas imunes a arquiteturas quânticas. Sistemas baseados em treliças são um dos criptossistemas que se provou serem resistentes a ataques por parte de computadores quânticos. Estes criptossistemas baseiam a sua segurança em problemas matemáticos duros, como o Problema do Vetor mais Curto (Shortest Vector Problem, SVP), Problema do Vetor mais Próximo (Closest Vector Problem, CVP), e outros. Assim sendo, o estudo de algoritmos desenhados para resolver estes problemas – também conhecidos como ataques – é de extrema importância, com o objetivo de os entender melhor, e por forma a melhor selecionar os parâmetros de construção destes criptossistemas.

O foco desta dissertação é um tipo específico de ataque: algoritmos baseados em célula de Voronoi, que são frequentemente mencionados na literatura como sendo impráticos, mas que têm sido alvo de muito pouca pesquisa, especialmente quando comparados com outros tipos. Nesta dissertação, várias otimizações para um algoritmo baseado em célula de Voronoi são propostas, com o objetivo de reduzir tanto o tempo de execução, como os requisitos de memória das implementações. De modo a aproveitar todo o poder computacional disponível nos computadores modernos, versões paralelas em CPU, GPU e heterogénea CPU + GPU são propostas, apresentando ganhos lineares no primeiro caso (e algum benefício com Hyper-Threading), e reportando ganhos de $13.37\times$ e $15.03\times$, para as versões GPU e heterogénea, respetivamente, quando comparadas com a versão sequencial base em CPU. Adicionalmente, diversas variantes heurísticas com implementações paralelas em CPU são propostas, com o intuito de acelerar a computação da solução do SVP, com a melhor estratégia de "poda" desenvolvida a alcançar ganhos até $256.51\times$, usando 8 threads (quando comparada com a versão sequencial base em CPU), calculando a solução correta para 99.84% das bases testadas.

Finalmente, conduzida no âmbito desta dissertação, a integração de todo o trabalho na biblioteca Lattice Unified Set of Algorithms (LUSA) – um projeto que ambiciona disponibilizar acesso público a uma coleção de implementações rápidas e fáceis de utilizar de algoritmos relacionados com treliças.

Os resultados obtidos nesta dissertação mostram, ainda que longe das dimensões do estado da arte, que é possível acelerar imensamente um algoritmo baseado em célula de Voronoi, mantendo

taxas de sucesso extremamente elevadas.

Palavras Chave

Criptografia; Treliça; Célula de Voronoi; Computação de Alto Desempenho; Paralelismo; Multi-threading; GPU; Computação heterogénea

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
CVP	Closest Vector Problem
FAQ	Frequently Asked Questions
FHE	Fully Homomorphic Encryption
GCC	GNU Compiler Collection
GMP	GNU Multiple Precision
GPU	Grahpics Processing Unit
HPC	High Performance Computing
HT	Hyper-Threading
LUSA	Lattice Unified Set of Algorithms
SM	Streaming Multiprocessor
SMT	Simultaneous Multi-Threading
SVP	Shortest Vector Problem

1

Introduction

Contents

Cryptography: the art of writing and solving codes.
– Oxford English Dictionary

1.1 Pre- and Post-Quantum Cryptography

Classic cryptographic systems, or cryptosystems (such as RSA and ElGamal), rely their security on hard math problems, such as the factorization of large numbers and the computation of discrete logarithms. These problems are easy to solve for the participants of the system, but hard for an outside attacker.

Recently, it was shown that these problems can be made feasible in the presence of quantum computers, with the use of algorithms specifically designed for this architecture [1–3]. In light of this discovery, the cryptography community shifted its attention to the design and study of "quantum-immune" solutions, thus marking the beginning of the so-called post-quantum era. One of the most prominent type is the lattice-based cryptosystem, which is believed to be safe even in the present of quantum computers [3, 4]. Cryptosystems based on lattices also have other advantages, such as being (relatively) easy to implement and use in practice, and the fact that they support Fully Homomorphic Encryption (FHE), which allows for certain operations to the applied directly to the encrypted data, without the need for decryption [5].

Lattice-based cryptosystems also base their security on hard math problems, such as the Shortest Vector Problem (SVP), Closest Vector Problem (CVP), their approximate variants, and others. Like their name suggests, computing the solution to the SVP consists in finding the shortest, non-zero vector that belongs to the lattice. Similarly, computing the solution to the CVP consists in finding the closest lattice vector to a given vector. This given vector is usually called the *target* vector.

Unlike the base problems of classical systems, these problems are believed to remain hard in the presence of quantum computers, i.e. they cannot be solved exponentially faster on these architectures. The approximate variants of these problems consist in solving them up to some constant α, i.e. compute a solution whose norm is, at most, α times bigger than that of the actual, correct solution for the problem at hand. For example, the α-SVP consists in finding a lattice vector whose norm is up to α times bigger than that of the shortest lattice vector. In 1981, van Emde Boas proved that solving the CVP is NP-hard (**N**on-deterministic **P**olynomial-time hard), and suggested that the case might be the same with the SVP [6]. In the following years, it was further proved that not only the SVP is NP-hard (for randomized reductions [7]), but so too their approximate variants, up to a certain constant. Specifically, Micciancio showed that the α-SVP is NP-hard, up to $\alpha = \sqrt{2}$ [8], and Khot further studied and improved this result [9].

Given the importance of the SVP, CVP, and others, several types of algorithms have been proposed to compute their solution, most notably enumeration, sieving, and Voronoi cell-based algorithms. Computing the solution to these problems is fundamental, as it provides an estimate of how secure the cryptosystems that use them as building blocks are in practice, allowing to safely and correctly select the parameters for the construction of secure systems.

1.2 Objectives

In this dissertation, several algorithmic simplifications and optimizations to a Voronoi cell-based algorithm are proposed, aimed at accelerating the computation of the solution to the SVP, followed by their implementation and analysis. Furthermore, parallel multi-core implementations of both the baseline and optimized algorithms are proposed for CPU, GPU, and heterogeneous (CPU + GPU) architectures.

Additionally, the Voronoi cell-based algorithm implemented for this dissertation and all its optimizations are integrated into the Lattice Unified Set of Algorithms (LUSA) library.

1.3 Contributions

The main contributions of this dissertation are:

- [EUSIPCO19] Filipe Cabeleira, Artur Mariano and Gabriel Falcão: *Optimizing the Memory Usage of Voronoi Cell-based Parallel Kernels for the Shortest Vector Problem on Lattices* (submitted in February 2019), Annex A
- [IEEE-ESL] Filipe Cabeleira, Artur Mariano and Gabriel Falcão: *Energy-efficient lattice-based cryptanalysis attacks on low-power embedded GPUs* (submitted in February 2019), Annex B

1.4 Preliminaries and definitions

Matrices are represented by bold, upper-case letters (**A**, **B**, **C**). Vectors are represented by bold, lower-case letters (**a**, **b**, **c**). The $||\cdot||$ operator represents the Euclidean norm.

A lattice $\mathcal{L}$ in $\mathbb{R}^n$ is the set of all linear combinations of a basis **B**. The basis **B** is a set of linearly independent vectors $\mathbf{b}_1$, ...,$\mathbf{b}_m$. The lattice $\mathcal{L}$ is expressed by Equation (1.1):

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

where m represents the rank of the lattice and if $m = n$, the lattice is of full-rank. Although it is possible to use non-integers to represent lattice bases, integers are normally used as they don't affect the hardness

of the problem, but are significantly easier to handle computationally, given that they avoid the precision issues associated with floating-point representations. Figure 1.1 shows an example lattice in $\mathbb{R}^2$ and its basis vectors.

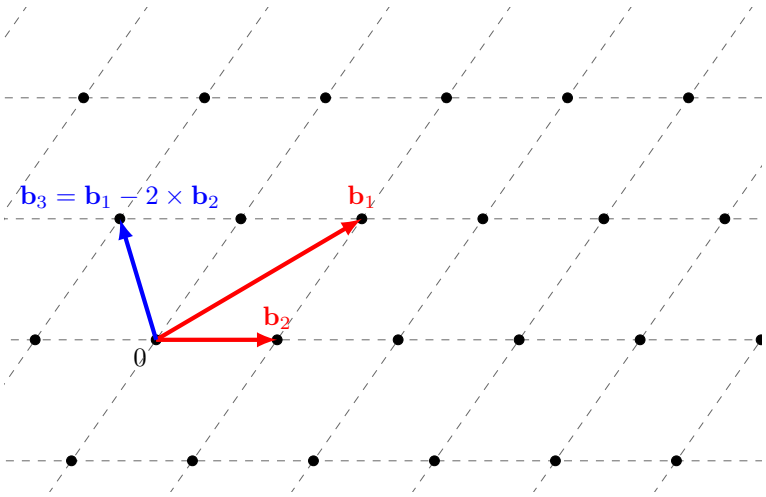


Figure 1.1: Example lattice in $\mathbb{R}^2$ and its basis vectors $\mathbf{b}_1$,$\mathbf{b}_2$ (in red).

The vector $\mathbf{b}_3$ is a linear combination of vectors $(\mathbf{b}_1, \mathbf{b}_2)$, shorter than $\mathbf{b}_1$, and exemplifies a reduced basis vector. The process of making basis vectors shorter is called lattice basis reduction and will be further discussed in Chapter 2.1.

1.5 Structure

This thesis is organized as follows: Chapter 2 provides and overview of algorithms that solve the SVP, with emphasis on Voronoi cell-based algorithms; Chapter 3 shows an optimized version of a Voronoi cell-based algorithm implementation, implemented for CPU, GPU and heterogeneous (CPU + GPU) architectures; Chapter 4 presents some algorithmic optimizations to the original algorithm; Chapter 5 gives an analysis of the LUSA library; and finally, Chapter 6 concludes this thesis with a summary of the conducted work and all conclusions that stemmed from it.

2

State of the Art

Contents

Algorithms to solve lattice-related problems (such as the SVP and CVP) can be split into three main categories: sieving, enumeration and Voronoi cell-based. While sieving and enumeration have been the target of academic research in recent years, yielding several variants of each type, Voronoi cell-based algorithms are not as investigated. This section presents an overview of the state of the art of these types and of lattice basis reduction algorithms.

2.1 Lattice basis reduction

An important type of algorithms used in lattice-based cryptanalysis are lattice basis reduction algorithms. As the name suggests, their objective is to reduce (i.e. shorten) the lattice basis vectors. However, they are not guaranteed to compute the shortest lattice vectors, but instead to compute *short* lattice vectors, all the while improving their orthogonality – a desirable trait in this context. Note that lattice basis reduction does not alter the lattice itself, merely uses different vectors to represent the same lattice, i.e. a different basis of the same lattice.

There are several reduction algorithms published in the literature, but the more widely used in the context of lattice-based cryptanalysis, in practice, are the Lenstra-Lenstra-Lovász (LLL) algorithm [10] and the Block Korkine-Zolotareff (BKZ) algorithm [11]. Even though it is possible to use integer arithmetic in the LLL algorithm, the most efficient implementations, in practice, use a floating-point representation. Notable implementations of the LLL algorithm include Schnorr and Euchner's LLL-SE [12] and Nguyen and Stehlé's L^2 [13].

The BKZ algorithm is a generalization of LLL, and allows to control the trade-off between execution time and basis quality, by means of a parameter β. This parameter defines the size of the block of the lattice to which several Korkine-Zolotareff reductions are applied, until the algorithm completes. The most efficient BKZ implementations to date are variants of BKZ, including Chen and Nguyen's BKZ 2.0 [14], Aono's et al. progressive BKZ [15] and Micciancio and Walter's primal-dual BKZ [16].

2.2 Enumeration

Enumeration algorithms have been the most widely researched family of lattice-related solvers. The name stems from the fact that they work by examining, i.e. enumerating, all the lattice points that can be found inside an hypersphere (Pohst in [17]) or, later, a parallelepiped (Kannan in [18]). The algorithms that were proposed afterwards are based on Pohst's and Kannan's techniques.

After the initial works of Pohst and Kannan, variants presented in the following years include Fincke and Pohst's [19] in 1985, and Kannan's [20] improvements in 1987.

Schnorr and Euchner published an improved version of Pohst's method in 1994 [12], that was later

shown (by Agrell et al. in [21]) to be faster than its predecessors. Schnorr and Euchner's algorithm is known in the literature as ENUM.

In 2002, Agrell et al. proposed an improved version of the ENUM algorithm, which became known as SE [21]. Later, Ghasemmehdi and Agrell presented a way to avoid unnecessary (or redundant) calculations during the enumeration procedure, showing significant gains with their technique. This algorithm is an improvement of SE and became known as SE++ [22].

In 2010, Gama et al. published the enumeration with extreme pruning algorithm in [23]. Pruning consists in discarding certain computations at the expense of the certainty of the results. However, the running time of the algorithm decreases much faster than the probability that the correct solution is discarded, i.e. the algorithm can be made much faster, while still maintaining a high degree of probability that the correct result is found.

Micciancio and Walter presented in [24] a low overhead variant of enumeration that matches the best asymptotic complexities of the class, and delivers practical performance as well.

There have also been proposed parallel versions of enumeration algorithms, such as the one presented by Dagdelen and Schneider in [25], which is based on the Schnorr and Euchner variant [12].

Parallel versions for GPU architectures have also been proposed, notably the work of Hermans et al., achieving speedups of $5\times$ in comparison to a sequential CPU implementation [26]. Kuo et al. presented a CPU and GPU heterogeneous implementation, based on Hermans' et al., and were able to find the solution to the SVP of basis in dimension 114 in less than two days [27].

More recently, in 2016, Correia et al. published in [28] a parallel version of the algorithm proposed by Ghasemmehdi and Agrell in [22], where they show improvements in scalability and overall throughput over Dagdelen and Schneider's version, achieving (almost) linear speedups for 16 threads.

2.3 Sieving

Sieving algorithms work by randomly selecting vectors from the lattice basis, to which a slight perturbation is added. Afterwards, these perturbed vectors are "sieved", i.e. their length is iteratively reduced, until all are bound by a certain radius. This means that short lattice vectors and their closest neighbors are discovered (with a high probability of success), and the solution to the SVP is then found by the pairwise differences of the previously computed vectors.

The first sieving algorithm was introduced in 2001 by Ajtai, Kumar and Sivakumar, and is known in the literature as AKS, after its authors [29]. The algorithms that followed were based on the same idea, and differ mainly on how the reduction is accomplished. Additionally, some of these algorithms are not proved to solve the SVP, as they rely on heuristics in order to speed up their execution, like the heuristic version of AKS presented by Nguyen and Vidick in [30].

Following AKS and Nguyen and Vidick's work, Micciancio and Voulgaris presented two new sieving algorithms: ListSieve, and its heuristic counterpart, GaussSieve, reporting improved time and space complexities over AKS [31]. The GaussSieve algorithm briefly dethroned enumeration as the fastest SVP-solver, until enumeration with extreme pruning (discussed earlier in Section 2.2) was introduced later in the same year. Given the performance of GaussSieve, several authors presented parallel implementations of the algorithm, such as Milde and Schneider, with nearly linear speedups up to 5 threads [32]. Ishiguro et al. improved the efficiency for higher thread counts [33]. Mariano et al. also presented a parallel GaussSieve implementation, achieving linear and almost linear speedups, using up to 64 threads, and generally improving upon previous work [34]. Bos et al. combine several optimizations, further reducing execution time and memory usage, when compared against previous work [35].

Another breakthrough in sieving algorithms came when Laarhoven presented its HashSieve algorithm, which replaced a brute-force portion of previous sieving algorithms by Charikar's angular locality-sensitive hashing method [36]. Later the same year, Mariano et al. presented a parallel version of HashSieve [37].

After HashSieve, Becker et al. presented the (eventually known as) LDSieve algorithm that, in comparison to the former, improved on the design of a data structure in order to speed up the reduction process [38]. In the following year, Mariano et al. proposed a parallel variant of the LDSieve algorithm in [39].

More recently, Ducas et al. presented a "generalized sieve kernel", that performed significantly better than the previous first place of the SVP-Challenge – a publicly available hall of fame and contest for solving the SVP, available at `https://www.latticechallenge.org/svp-challenge/` – by computing the shortest vector of a 155-dimensional lattice in under 15 days [40].

2.4 Voronoi cell-based

The working principle of this type of algorithm, as the name suggests, relies on the computation of the Voronoi cell of the lattice. This cell contains, by definition, the set of all points closest to zero than any other lattice point. In practice, this is represented by the set of the smallest lattice vectors. The vectors that comprise the minimum set required for a complete description of the Voronoi cell are called *relevant vectors*. Minkowski proved that the Voronoi cell of an n-dimensional lattice has, at most, $2 \times (2^n - 1)$ facets [41]. However, they are not all necessarily *relevant*. Figure 2.1 exemplifies both a Voronoi cell, and its relevant vectors.

This family of SVP-solvers has not been subject to the same amount of research as the previously mentioned types. This lack of research is mainly due to the fact that they are not as practical as enumeration and sieving types, despite having better time and space complexities in theory. One of the aims

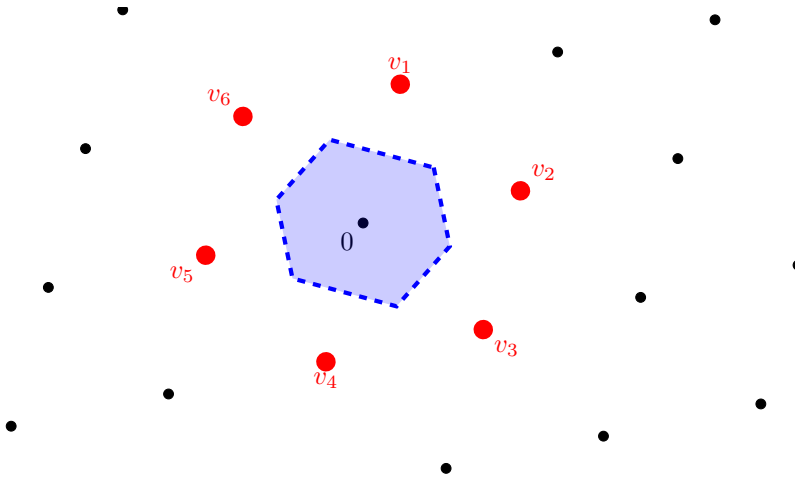


Figure 2.1: Example of a Voronoi cell in $\mathbb{R}^2$ (blue), and its relevant vectors (red).

and contributions of this dissertation is to propose and analyze optimizations to one such algorithm.

One of the first published works of this kind is Viterbo and Biglieri's, who present an algorithm to compute the Voronoi cell of a lattice, named the diamond-cutting algorithm [42]. The first step of the algorithm is to construct the polytope – a bounded region – defined by the basis vectors. This polytope is guaranteed to contain the Voronoi cell of the lattice, however "short" the basis vectors are (see Figure 2.1; even if the relevant vectors were chosen to construct the polytope, the Voronoi cell of the lattice is contained inside). Afterwards, the polytope is iteratively cut until a stopping condition is met, i.e. only the Voronoi cell of the lattice remains.

Following this was the work of Agrell et al., who presented an improvement of the Schnorr-Euchner enumeration algorithm [12], with the main goal of solving the CVP [21]. Additionally, several variants to solve other lattice-related problems are proposed, e.g. the computation of the solution to the SVP, the determination of the kissing number of a lattice (the number of shortest, non-zero lattice vectors), and the computation of the Voronoi relevant vectors of a lattice. This dissertation is based on the latter, and makes use of the enumeration-based CVP-solver in order to compute said relevant vectors.

Later, Micciancio and Voulgaris presented an algorithm that uses the Voronoi cell of the lattice as a hint for a CVP solver, naturally computing said cell as well [43]. They show that the Voronoi cell of a lattice in a given dimension k can be computed by a series of CVP computations in dimension k, and the solution to a CVP computation in dimension k can be done by a series of CVP calls in dimension $k-1$. Therefore, the Voronoi cell of a lattice can be iteratively computed, starting on dimension 1 and working upwards to dimension n.

3

An optimized version of the Voronoi cell algorithm for the SVP

Contents

The work conducted in this dissertation is based on the algorithm presented by Agrell et al. in [21], specifically the *Relevant Vectors* algorithm, which allows to compute the Voronoi relevant vectors of an arbitrary lattice.

This section presents the chosen algorithm and details its implementation, with emphasis on computational optimizations performed to benefit its execution. Afterwards, both CPU and GPU parallel implementations of the algorithm are presented and discussed.

The hardware used for all the work conducted in this thesis is depicted in Table 3.1. Hyper-Threading (HT) is the name given to Intel's Simultaneous Multi-Threading (SMT) implementation. The clock frequency in parenthesis is attained with Turbo Boost technology.

Table 3.1: Specifications of the Machines used throughout the dissertation.

Machine	**A**	**B**
CPU	Intel Core i7 740QM	Intel Core i3 6100
Clock frequency	1.73 GHz (2.93 GHz)	3.70 GHz
Cores	4	2
SMT	Yes (w/HT, 8 threads)	Yes (w/HT, 4 threads)
L1 Cache	32 kB i + 32 kB d	32 kB i + 32 kB d
L2 Cache	256 kB	256 kB
L3 Cache	6 MB	3 MB
RAM	8 GB	8 GB
GPU	—	NVIDIA GeForce 1060 GTX
GPU Clock rate	—	1759 MHz
GPU RAM	—	6 GB
CPU/OpenMP Compiler	GCC (g++) 7.2.0	—
OpenACC Compiler	—	PGI Compiler Suite 18.4 (pgcc)
CUDA Compiler	—	CUDA Toolkit 9.1 Compiler (nvcc)

Machine A is running Ubuntu 17.10 x86_64, with kernel version 4.13. Machine B is running Ubuntu 16.04 x86_64, with kernel version 4.13. All programs were developed in C/C++ and compiled using the `-O3` optimization flag. When compiled with GCC, the `-march=native` optimization flag was employed as well. The OpenACC implementation is compiled with the `-acc -Mcuda=9.1 -ta=tesla:cc60` flags, which instructs the compiler to enable OpenACC directives, use the CUDA toolkit version 9.1 and to generate code for a GPU with Compute Capability 6, respectively (the Compute Capability of a GPU, as per the CUDA FAQ, "determines its general specifications and available features"). Similarly, the CUDA implementation is compiled with the `-arch=sm_61`, to inform the compiler that a GPU with Compute Capability 6.1 is to be used.

The lattice bases used were generated with SVP-Challenge's basis generator (available at https://www.latticechallenge.org/svp-challenge/), compiled with NTL version 9.3 (available at https:

//www.shoup.net/ntl/). Unless otherwise stated, 1000 different bases are used for dimensions 4 to 10, 100 for dimensions 11 to 15, and 10 for dimensions 16 and upwards. A "run" consists in executing the algorithm for all test bases in the dimensions shown. The run times depicted consist of the arithmetic average of all bases executed, for each dimension, and represent the full execution time of the program, with the exception of file input/output. Machine A is used for CPU runs, and Machine B is used for GPU runs.

3.1 The Voronoi cell-based algorithm

The algorithm used as basis for the work developed on this dissertation is presented by Agrell et al. in [21], which is an improvement of Schnorr and Euchner's work published in [12]. This paper proposes an enumeration-based CVP solver that can be adapted to solve the SVP, compute the Voronoi cell of a lattice, as well as other problems. Formally, the Voronoi cell $\mathcal{V}$ of a lattice $\mathcal{L}$ – the set of all lattice points closest to zero than any other lattice point – is given by Equation 3.1,

$$\mathcal{V}(\mathcal{L}) = \{\mathbf{x} \in \mathbb{R}^n : ||\mathbf{x}|| \leq ||\mathbf{x} - \mathbf{v}|| \quad \forall\, \mathbf{v} \in \mathcal{L}\}, \tag{3.1}$$

where n is the dimension of the lattice.

As previously mentioned, enumeration algorithms work by examining all possible lattice points inside a certain radius – an hypersphere or a parallelepiped. The algorithm at hand is based on the former strategy.

The *Relevant Vectors* algorithm makes use of this enumeration-based CVP-solver in order to compute the Voronoi cell of an arbitrary lattice. This algorithm can be split into four distinct steps, starting with the generation of the target vectors, that will be later fed into the enumeration-based CVP-solver.

Afterwards, the lattice basis is converted to a lower triangular form (that exists for all lattice bases), required by the CVP-solver, so that lattices of any type can be used with this algorithm. This modification can be thought of as a change in coordinate system. The target vectors are also transformed in an appropriate fashion into the new coordinate system.

Following these transformations, the modified lattice basis and target vectors are fed into the CVP-solver, and the same nomenclature that is used by the algorithm's authors regarding this step will be used in this dissertation. As such, this will be called the "decode" step, and decoding a target vector consists of executing the decode function for the given target vector.

Finally, the output of the decode function is converted back to the original coordinate system and processed. If this result is valid, i.e. it is a relevant vector, then it is added to the list of relevant vectors.

The pseudo-code of the implementation just described is shown in Algorithm 3.1.

Implementation-wise, it is desirable to begin by using a lattice basis reduction algorithm on the input

Function AllClosestPoints

Input: Matrix $\mathbf{M}$, matrix **H**, matrix **Q**, vector $\mathbf{s}$
Output: List of vectors $\mathbf{X}$

Compute $\mathbf{x} = \mathbf{s}\mathbf{Q}^T$; `/* conversion of the target vector to the modified coordinate system */`
$\mathbf{U}$ = Decode($\mathbf{H}$, $\mathbf{x}$);
Compute γ as the lowest value $||\mathbf{u}\mathbf{M} - \mathbf{s}||$ for all $\mathbf{u} \in \mathbf{U}$;
Compute $\mathbf{X}$ as all $\{\mathbf{u}\mathbf{M} : \mathbf{u} \in \mathbf{U}, ||\mathbf{u}\mathbf{M} - \mathbf{s}|| = \gamma\}$
return $\mathbf{X}$

Algorithm 3.1: Relevant Vectors

Input: Basis matrix $\mathbf{B}$
Output: Relevant Vectors $\mathbf{N}$

$\mathbf{M}$ = Reduce($\mathbf{B}$); `/* for example, using the LLL algorithm */`
[$\mathbf{Q}$, $\mathbf{R}$] = QR decomposition of $\mathbf{M}$;
$\mathbf{G} = \mathbf{R}^T$;
$\mathbf{H} = \mathbf{G}^{-1}$; `/* lattice basis on modified coordinate system */`
$\mathbf{N} = \varnothing$; `/* list of relevant vectors */`

forall vectors $\mathbf{s} \in \mathcal{TV}$ **do**
  $\mathbf{X}$ = AllClosestPoints($\mathbf{M}$, **H**, **Q**, $\mathbf{s}$);
  if $|\mathbf{X}| = 2$ **then**
    $\mathbf{N} = \mathbf{N} \bigcup \{2\mathbf{x} - 2\mathbf{s} : \mathbf{x} \in \mathbf{X}\}$;
return $\mathbf{N}$

basis **B**, resulting in the reduced basis **M**. This allows to increase the performance and numerical stability of the algorithm, given that, among other advantages, a lattice basis reduction algorithm shortens the basis vectors to some extent. This can be accomplished by means of an LLL or BKZ reduction.

In practice, following basis reduction, the necessary computations that allow for the coordinate system transformation are performed, given that they are constant for a given lattice, and can be computed only once. This can be achieved, for example, with a QR decomposition (a matrix **A** can be decomposed into the product of an orthonormal matrix **Q** and an upper triangular matrix **R**, such that **A** $=$ **QR**).

Afterwards, the $\mathbf{s}_i$, $i = 1, ..., 2^n - 1$ target vectors (where n is the dimension of the lattice) are generated according to Equation 3.2. Following this, the main loop of the algorithm is started, where each iteration decodes a target vector.

$$\mathcal{TV}(\mathbf{M}) = \left\{\mathbf{s} = \mathbf{z}\mathbf{M} : \mathbf{z} \in \{0, 1/2\}^n - \{\mathbf{0}\}\right\} \tag{3.2}$$

The result **U** of the execution of the decode procedure is then processed according to Equation 3.3, resulting in matrix **X**.

$$\begin{aligned} \gamma &= \min\left\{||\mathbf{u}\mathbf{M} - \mathbf{s}|| \text{ for all } \mathbf{u} \in \mathbf{U}\right\} \\ \mathbf{X} &= \left\{\mathbf{u}\mathbf{M} : \mathbf{u} \in \mathbf{U}, ||\mathbf{u}\mathbf{M} - \mathbf{s}|| = \gamma\right\} \end{aligned} \tag{3.3}$$

If matrix **X** has two, and only two vectors, then the computed solution is valid, a relevant vector has

been found, and it is added to the list of relevant vectors **N**. Furthermore, if this is the case, these vectors are symmetric to one another and, therefore, have the same norm. Conversely, if only one or more than two vectors are returned in matrix **X**, the result is *not* valid, and is discarded.

Finally, the solution to the SVP is given by the shortest vector of the computed Voronoi relevant vectors (matrix **N**).

3.2 Optimizations

Several optimizations and simplifications were used in the implementation of the algorithm, both general-purpose and algorithm-specific ones. These optimizations serve to reduce the memory footprint of the implementation and/or reduce its execution time.

One such optimization consists in using larger, single-allocation arrays for both vector and matrix structures. While matrices made up of arrays of arrays provide a more natural indexing notation, they require several calls to memory allocation functions and, later, deallocation. This not only increases the time required to allocate the necessary memory for a matrix, but is also detrimental during execution, as each "sub-array" (row or column of the matrix, depending on how it is implemented) is not guaranteed to be continuous in RAM, and reading/writing to such a structure requires two memory accesses – the first for the address of the "sub-array", and the second for the value itself. The use of a larger, single-allocation array allows for faster memory accesses and guarantees that the entire structure is continuous in RAM, at the expense of slightly inconvenient indexing. However, this indexing can be facilitated by the use of a compiler macro. The indexing used for the implementations for this dissertation is exemplified in Listing 3.1.

Listing 3.1: Example of the indexing used for vectors and matrices.

```
#define ind(a, b) a*NUM_COLUMNS+b    // NUM_COLUMNS represents the number of
    columns of the matrices

float *vec, *mat;    // vec (pointer to vector) and mat (pointer to matrix)
    allocation omitted

// Indexing a vector (standard usage)
vec[i] = 1.0f;

// Indexing a matrix (using the macro)
mat[ind(x, y)] = 1.0f;
```

Note that this indexing method does require the computation of the index for matrix accesses, but the benefits outweigh the costs nonetheless.

In regards to algorithm-specific optimizations, and because we are only interested in the solution to the SVP, storing the full description of the Voronoi cell is unnecessary. Storing the entire Voronoi cell would require matrix **N** to have $2 \times (2^n - 1)$ rows – the maximum number of relevant vectors for dimension n – thus increasing in size exponentially with the dimension of the lattice (if a given vector is relevant, so it is its symmetric, resulting in the factor 2 on the equation). As such, matrix **N** was transformed into a single vector, and only the shortest relevant vector found up to that point in the execution is stored. This strategy requires the use of an auxiliary variable (named `ShortestNormFound`, for example), where the norm of the aforementioned relevant vector is stored. At the end of each iteration of the main loop of the algorithm, the norm of the computed vector (if valid) is compared against `ShortestNormFound` and, if smaller, both **N** and `ShortestNormFound` are updated with the new-found vector. This modification does not adversely impact the execution time of the implementation given that the shortest vector would need to be found from among all relevant vectors that would be stored in the full matrix **N**.

Another simplification was performed for matrix **X**. Recall that the output of the decode function is only valid if it returns (through matrix **X**) two, and only two, vectors. As such, during the computation of said matrix, if a third vector is found eligible to be part of it, the function can return immediately with the information that **X** has, at least, three vectors and, therefore, it is not a valid solution. This simplification not only potentially saves time due to the (potentially early) interruption of the computation of the **X** matrix, but also saves memory by allowing the size of the matrix to be fixed at 2 rows, instead of an unknown, dynamically-growing number of rows.

The pseudo-code of the implementation with the described algorithm-specific optimizations is shown in Algorithm 3.2.

Function OptimizedAllClosestPoints

Input: Matrix $\mathbf{M}$, matrix **H**, matrix **Q**, vector $\mathbf{s}$
Output: List of vectors $\mathbf{X}$

Compute $\mathbf{x} = \mathbf{s}\mathbf{Q}^T$;
$\mathbf{U}$ = Decode($\mathbf{H}$, $\mathbf{x}$);
Compute γ as the lowest value $||\mathbf{u}\mathbf{M} - \mathbf{s}||$ for all $\mathbf{u} \in \mathbf{U}$;
NumRowsX = 0; `/* number of rows of the matrix X */`
forall vectors $\mathbf{u}\mathbf{M} : \mathbf{u} \in \mathbf{U}$ **do**
    if $||\mathbf{u}\mathbf{M} - \mathbf{s}|| = \gamma$ **then**
        NumRowsX++;
        if NumRowsX ≥ 2 **then**
            break;
        $\mathbf{X} = \mathbf{X} \bigcup \{\mathbf{u}\mathbf{M}\}$;
return $\mathbf{X}$

Algorithm 3.2: Optimized Relevant Vectors

Input: Basis matrix $\mathbf{B}$
Output: Relevant Vectors $\mathbf{N}$

$\mathbf{M}$ = Reduce($\mathbf{B}$); /* for example, using the LLL algorithm */
[$\mathbf{Q}$, $\mathbf{R}$] = QR decomposition of $\mathbf{M}$;
$\mathbf{G} = \mathbf{R}^T$;
$\mathbf{H} = \mathbf{G}^{-1}$;
$\mathbf{N} = \varnothing$;
ShortestNormFound = ∞;
forall vectors $\mathbf{s} \in \mathcal{TV}$ **do**
  $\mathbf{X}$ = OptimizedAllClosestPoints($\mathbf{M}$, **H**, **Q**, $\mathbf{s}$);
  if $|\mathbf{X}| = 2$ **and** $||2\mathbf{x} - 2\mathbf{s}|| <$ ShortestNormFound **then**
    ShortestNormFound = $||2\mathbf{x} - 2\mathbf{s}||$;
    $\mathbf{N} = \{2\mathbf{x} - 2\mathbf{s} : \mathbf{x} \in \mathbf{X}\}$;
return $\mathbf{N}$

3.3 Parallel CPU implementation

The algorithm can be classified as “embarrassingly parallel”, given that the iterations that comprise the main loop of the algorithm are independent from each other. Therefore, linear speedups should be attainable from a parallel implementation. One such implementation was devised, to test this hypothesis and further accelerate the algorithm, making use of OpenMP – a programming interface based on compiler directives that allows developers to express parallelism. These directives were applied to the main loop of the algorithm, where the heaviest computational work-load is present.

In regards to the implementation, the main difference when compared to the sequential one lies in memory allocation, where some structures must be made private to each thread. Because the original implementation uses dynamic memory allocations, this must be done manually. As such, the aforementioned private structures are now allocated and deallocated at the begin and at the end, respectively, of each iteration of the loop. The need for private structures causes the memory usage of algorithm to increase with the number of threads used, but such usage remains low.

Additionally, given the fact that each iteration takes a different amount of time to complete, OpenMP’s dynamic scheduler is used, in order to prevent work-load imbalances across the threads (dynamic scheduling works by assigning each thread a new iteration as they finish their current one, contrarily to the static scheduler, that splits the iterations between the available threads when the loop is started). Even though this scheduling policy incurs in additional overhead, tests conducted showed no adverse effects from its use.

Despite the fact that the iterations of the original algorithm are completely independent, our optimization of keeping only the shortest relevant vector found adds the necessity of some communication between threads. This is achieved by the use of OpenMP’s critical region, that guarantees that only one thread at a time can enter the region of code where the comparison and, if necessary, the updating of

the shortest vector is performed. Despite this new-found need for thread communication, the execution times of the implementation were not affected, possibly due to the fact that it represents a very small part of each iteration and, given that each decode may take a different amount of time, there is the possibility that the working threads seldom compete for entry to the critical region.

Algorithm 3.3 shows the pseudo-code of the parallel OpenMP implementation, with the smaller **N** matrix simplification of Chapter 3.2.

Algorithm 3.3: OpenMP Parallel Relevant Vectors

Input: Basis matrix $\mathbf{B}$
Output: Relevant Vectors $\mathbf{N}$

$\mathbf{M}$ = Reduce($\mathbf{B}$); `/* for example, using the LLL algorithm */`
[$\mathbf{Q}$, $\mathbf{R}$] = QR decomposition of $\mathbf{M}$;
$\mathbf{G} = \mathbf{R}^T$;
$\mathbf{H} = \mathbf{G}^{-1}$;
$\mathbf{N} = \varnothing$;
ShortestNormFound = ∞;

```
#pragma omp parallel for
```
forall vectors $\mathbf{s} \in \mathcal{M}$ **do**
    $\mathbf{X}$ = OptimizedAllClosestPoints($\mathbf{M}$, **H**, **Q**, $\mathbf{s}$);
    `#pragma omp critical`
    if $|\mathbf{X}| = 2$ **and** $||2\mathbf{x} - 2\mathbf{s}|| <$ ShortestNormFound **then**
        ShortestNormFound = $||2\mathbf{x} - 2\mathbf{s}||$;
        $\mathbf{N} = \{2\mathbf{x} - 2\mathbf{s} : \mathbf{x} \in \mathbf{X}\}$;
return $\mathbf{N}$

As previously mentioned, the algorithm is expected to scale linearly with the number of cores used. In order to test this hypothesis, a run of the implementation was performed with Turbo Boost disabled, due to the fact that this technology affects the clock frequency of the CPU differently with the number of cores used, i.e. the less cores are being used, the higher the boost. Figure 3.1 shows the execution times of the implementation of Algorithm 3.3, for lattice dimensions 10 through 20. Table 3.2 shows the scalability of the parallel implementation.

Table 3.2: Speedup of the parallel CPU implementation, for 2, 4 and 8 threads, on Machine A, with Turbo Boost disabled, lattice dimensions 16 through 20. Values in bold represent the highest speedup for a given thread count.

Dimension	16	17	18	19	20
2 Threads	1.9794	1.9784	1.9783	**1.9876**	1.9874
4 Threads	3.9331	3.9613	3.9662	3.9812	**3.9863**
8 Threads (HT)	5.3053	5.4100	5.4305	5.4284	**5.4382**

These results show that the parallel CPU implementations does indeed scale linearly with the number of cores used, and can benefit from the use of Hyper-Threading, albeit by a smaller degree, thus achieving a maximum speedup of $5.44\times$, when compared to the sequential implementation.

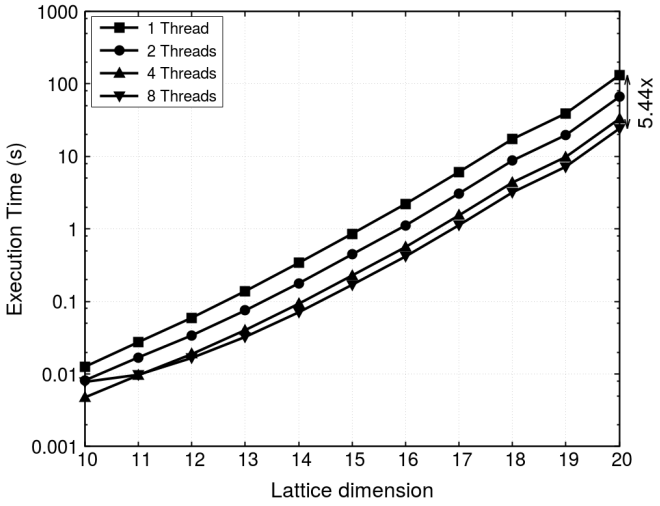


Figure 3.1: Execution times of the parallel CPU implementation on Machine A (Turbo Boost disabled), lattice dimensions 10 through 20.

Figure 3.2 shows the execution times of the parallel implementation, in the same conditions as before, but with Turbo Boost enabled on the CPU.

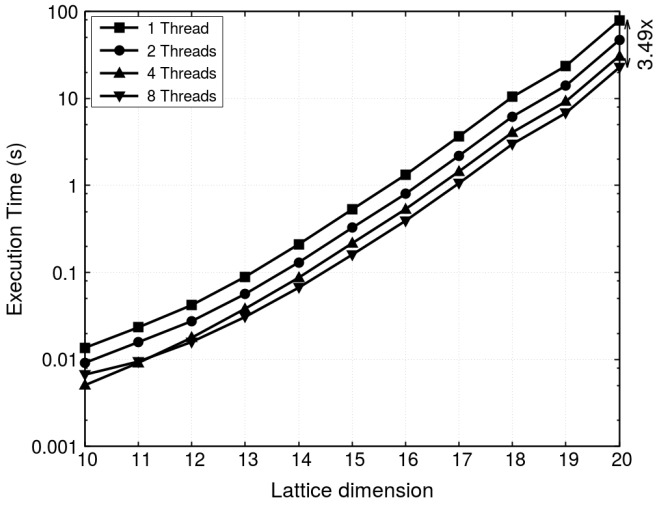


Figure 3.2: Execution times of the parallel CPU implementation on Machine A (Turbo Boost enabled), lattice dimensions 10 through 20.

The speedup in this case is smaller – due to Turbo Boost, as explained previously – but overall execution time is lower than with Turbo Boost disabled, achieving a maximum speedup of $3.49\times$, when compared to the sequential implementation.

3.4 Parallel GPU implementation

A parallel implementation of the algorithm for GPU architectures was also devised, using both OpenACC and CUDA. OpenACC is a pragma-based (compiler directives) programming tool, similar to OpenMP, and it is capable of generating executable code for both multi-core CPUs and GPUs. CUDA is a programming model, with a C/C++ interface, that allows computational kernels to be run on a GPU.

While OpenACC masks some of the complexity of low-level GPU-oriented languages, e.g. CUDA, it allows programmers to quickly develop a GPU version of an algorithm with very little effort, all the while achieving very good results. Because of this, and even if more control is required, an OpenACC implementation can be regarded as the benchmark for a potential CUDA implementation. Even though most of the choices can be left to the compiler, there is also the possibility for developers to change and tweak some settings, most notably the launch configuration of the generated kernels. As such, OpenACC allows users to modify the number of gangs, number of workers and vector length. These parameters directly relate to CUDA's number of blocks, warp size, and threads per block, respectively.

As with the parallel CPU implementation, the compiler directives for the OpenACC implementation are applied to the main loop of the algorithm, in a similar fashion as the OpenMP version. The main difference in the OpenACC implementation is the need to explicitly inform the compiler of the size of the arrays used. This is due to the fact the arrays are dynamically allocated, and because some data must be copied to the device (GPU), and later back to the host (CPU), their size must be known. This is not (generally) needed in an OpenMP implementation, as all memory resides on the main RAM, and no memory transfers are necessary. Algorithm 3.4 shows the pseudo-code of the OpenACC implementation.

Algorithm 3.4: OpenACC Parallel Relevant Vectors

Input: Basis matrix $\mathbf{B}$
Output: Relevant Vectors $\mathbf{N}$

$\mathbf{M}$ = Reduce($\mathbf{B}$); `/* for example, using the LLL algorithm */`
[$\mathbf{Q}$, $\mathbf{R}$] = QR decomposition of $\mathbf{M}$;
$\mathbf{G} = \mathbf{R}^T$;
$\mathbf{H} = \mathbf{G}^{-1}$;
$\mathbf{N} = \varnothing$;

```
#pragma acc parallel loop independent
```
forall vectors $\mathbf{s} \in \mathcal{M}$ **do**
  $\mathbf{X}$ = OptimizedAllClosestPoints($\mathbf{M}$, **H**, **Q**, $\mathbf{s}$);
  if $|\mathbf{X}| = 2$ **then**
    $\mathbf{N} = \mathbf{N} \bigcup \{2\mathbf{x} - 2\mathbf{s} : \mathbf{x} \in \mathbf{X}\}$;
return $\mathbf{N}$

The OpenACC compiler automatically parallelizes the code in such a way that each GPU thread decodes a single target vector, and chose a default vector length L of 128. This choice is empirically

supported by follow-up tests, that confirmed it to be the best value for L. The gang size T (number of blocks) depends on the number of target vectors, and is given by Equation (3.4), so that each thread decodes one target vector.

$$T = \left\lceil \frac{\text{Number of Target Vectors}}{L} \right\rceil \tag{3.4}$$

Figure 3.3 depicts the execution flow of the GPU implementations.

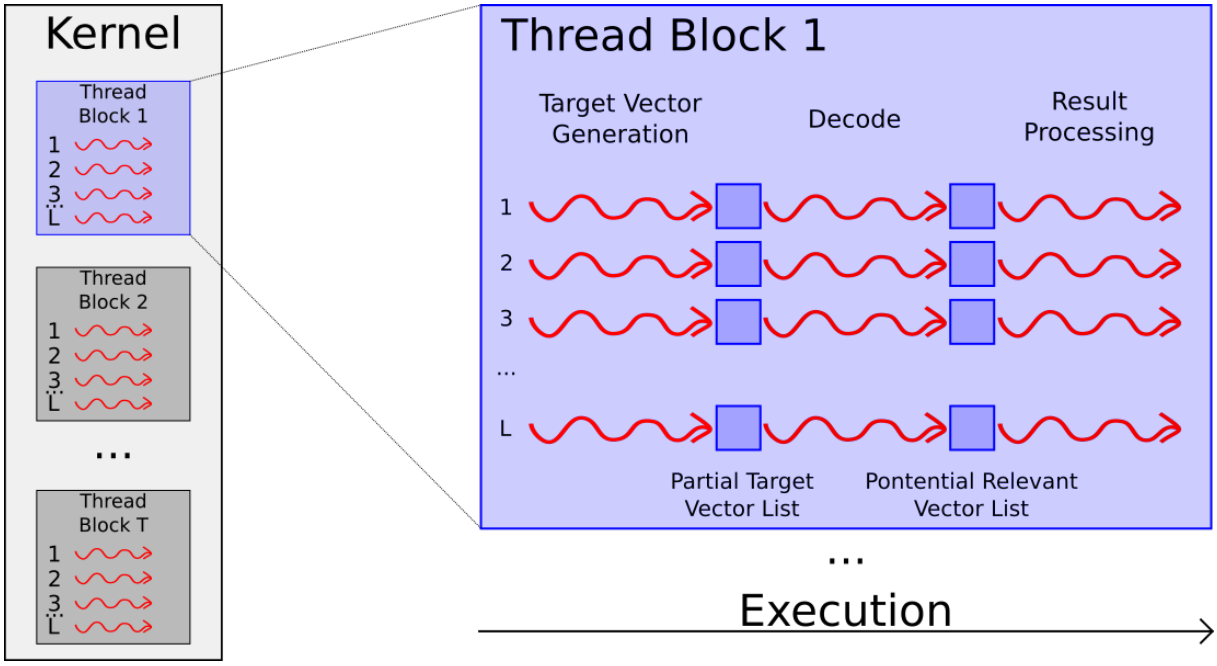


Figure 3.3: Execution flow of the GPU implementations – the red arrows represent each individual thread.

Since OpenACC does not provide a critical region construct, the smaller target vector list **N** of the CPU implementation cannot be used here. However, the original algorithm's description would lead to the storage of both a relevant vector and its symmetric, thus requiring $2 \times (2^n - 1)$ rows for matrix **N**. As such, only one of the two vectors is stored, thus decreasing the memory requirements of matrix **N** by half, when compared to the original, requiring only $2^n - 1$ rows.

Due to the fact that some arrays must be private to each executing thread, the memory requirements of this implementation are largely superior than its CPU counterpart, as the number of threads is also much higher. Because of this, the memory usage of the OpenACC implementation, for dimension 19, reaches almost 6 GB – the amount of memory available on the GPU used. In order to be able to run the implementation in higher dimensions, a trade-off is necessary: executing more than one iteration per thread, thus decreasing – or maintaining – the memory usage. Therefore, runs in dimension 20 are launched with the same gang size T as dimension 19, resulting in 2 target vectors being decoded by each thread. Figure 3.4 shows the execution times of the OpenACC implementation, achieving a

maximum speedup of $12.15\times$, when compared against the sequential CPU implementation.

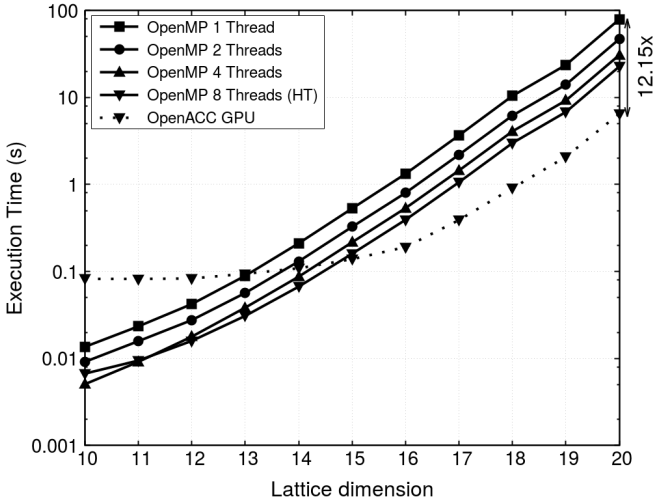


Figure 3.4: Execution times of the sequential CPU implementation (Machine A) and parallel GPU OpenACC (Machine B), lattice dimensions 10 through 20.

In regards to the CUDA implementation, the same strategy was used as in the OpenACC one: $L = 128$ threads per block and the number of blocks T depending on the dimension of the problem, as given by (3.4), with each thread decoding a single target vector, except for higher dimensions, when memory usage hits the limit of the GPU. Additionally, the same strategy regarding matrix **N** is used as in the OpenACC implementation – the 1-row simplification of Chapter 3.2 is not used, but only half of the original algorithm's required space is needed for this matrix. Figure 3.5 shows the execution times of the baseline CUDA implementation against the OpenACC version.

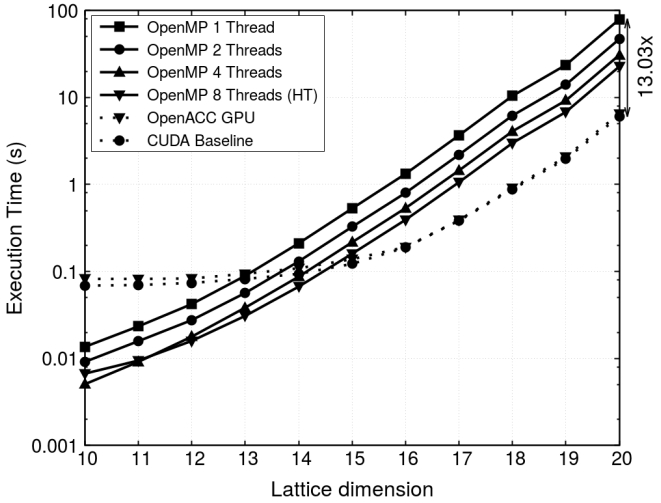


Figure 3.5: Execution times of the sequential CPU, baseline CUDA and OpenACC implementations, lattice dimensions 10 through 20.

The baseline CUDA implementation reports a similar performance to the OpenACC one, with a

gain of 7.23% for dimension 20, therefore achieving a speedup of $13.03\times$, when compared against the sequential CPU implementation.

A different configuration for the GPU implementation was also tested, in which thread blocks of n threads would decode a target vector, i.e. $L = n$, and $T =$ Number of Target Vectors $= 2^n - 1$. However, this was quickly discarded as it proved to be severely slower than the first described configuration. This is due to the fact that, even though the iterations of the main loop of the algorithm are independent, there is no advantage to using several threads to process each target vector, as the decode procedure is inherently sequential and is executed by a single thread only.

Given the fact that CUDA allows for a finer control over the low-level details of the implementation, several optimizations were performed, starting with the use of constant memory for matrices **M**, **Q** and **H**, as they remain unchanged after their computation. Because it is cached, constant memory accesses are much faster than global memory accesses. Shared memory was tested for this purpose as well, given that it resides on-chip (contrarily to global or constant memory) but, even though it showed improvement compared to global memory, constant memory was faster still.

Additionally, the contents of the aforementioned matrices can be transferred to the GPU as soon as they are computed. This is achieved by using asynchronous memory transfers, that immediately return control to the CPU while the transfer takes place, allowing for work to be done while data is copied over.

Regarding the private structures necessary for each thread, and in order to avoid several calls to memory allocation functions – an expensive operation on the GPU – a single, very large array is requested with all the necessary memory for all structures. The various arrays needed are then obtained by pointing to different zones of this larger array, thus "emulating" several smaller arrays.

These optimizations allow for a 9.43% maximum gain compared to the baseline CUDA implementation, for the tested dimensions. The execution times of the CUDA implementation with all optimizations are shown in Figure 3.6, compared against the OpenACC and CPU implementations.

The figure shows that the optimized CUDA implementation is the fastest of all GPU implementations, achieving a speedup of $13.37\times$, for dimension 20, when compared against the sequential CPU reference.

Common to all the GPU implementations presented, due to the fact each thread decodes a single target vector, and this process may take different amounts of time for different target vectors, is the fact that warp divergence is prominent – different threads following different paths based on conditional execution, i.e. `if` statements.

Another important metric for GPU implementations is occupancy, which indicates how many warps can run concurrently on a Streaming Multiprocessor (SM). The occupancy of the proposed CUDA implementation is limited by register usage, which was empirically set at 84 registers per thread. Even though this many registers limits the maximum occupancy of the implementation to 31.2%, it was the value at which the best performance was attained, showing that a higher occupancy does not necessarily

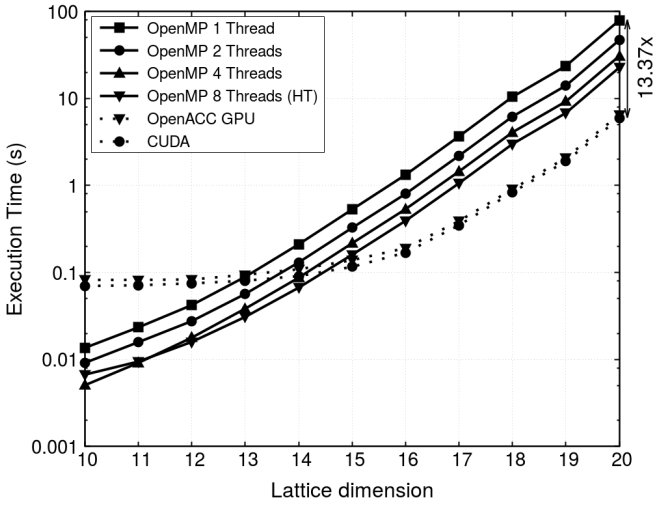


Figure 3.6: Execution times of the optimized CUDA and OpenACC implementations, lattice dimensions 10 through 20.

translate into higher performance.

The results obtained show that an optimized CUDA implementation can still outperform OpenACC, but requires significantly more developer time. However, if fine-grained control over the minute details of the implementation is not required, or if a benchmark of the performance of a given algorithm is warranted, OpenACC has proven to be a very effective tool to quickly accelerate code that exhibits some level of parallelism.

3.5 An heterogeneous implementation of the Voronoi algorithm

In order to harness the full computational power available in modern computer systems, an heterogeneous CPU + GPU implementation was devised.

To this end, the StarPU framework was used – a programming library for heterogeneous, multi-core systems. This framework is based around the concept of "codelets" and "tasks". As per its documentation, the former "describes a computational kernel that can possibly be implemented on multiple architectures such as a CPU, a CUDA device or an OpenCL device", and the execution of the latter "consists in applying a codelet on a data set, on one of the architectures on which the codelet is implemented". Similarly to CUDA kernels, tasks in StarPU are launched asynchronously. Workers are processing units (or parts of one) that are capable of executing a given task, e.g. a CPU core or a GPU. By default, StarPU uses only physical CPU cores and, when a GPU implementation is also present, one of the CPU cores is reserved to control the GPU.

3.5.1 CPU + GPU concerns

The main concerns regarding an heterogeneous implementation are workload division and scheduling, and memory management.

The fact that the algorithm itself is comprised of a series of independent decodes, StarPU's tasks provide a natural way to deal with the former, by assigning any number of decodes to a task. However, because CUDA kernels require a certain amount of threads to take advantage of a GPU's architecture, the question of how many decodes to perform per task arises. Too few decodes per task, and not only the GPU is severely underutilized, but the kernel launch overhead becomes a significant part of all kernel executions; too many decodes per task, and the risk of workload imbalances between the processing units rises, i.e. having one processing unit still executing a task, and the other simply waiting for a significant amount of time if no more work is available. Naturally, a compromise is necessary in order to reduce the workload imbalance, while keeping kernel overhead to a minimum.

Regarding memory management, given that different processing units may have distinct memory spaces (e.g. the CPU and the GPU), data availability and consistency must be guaranteed. StarPU provides mechanisms that automatically manage these necessities, freeing the programmer from the low-level details of memory management.

3.5.2 Implementation and results

As previously mentioned, the implementation of the heterogeneous Voronoi cell was achieved using the StarPU Application Programming Interface (API).

The task-based structure of StarPU, and the fact that the Voronoi cell algorithm consists of decoding several, independent target vectors, led to a natural mapping of decoding target vectors in each task. This allows any number of decodes to be performed per task, from a single one to all of them. However, given the concerns expressed earlier in regards to the coarseness of this workload distribution, several decoding procedures must be performed per task. Tests conducted showed that decoding $P_H = 2^{14} = 16384$ target vectors per task resulted in a good compromise between workload balance and overall overhead. A power of two was selected given that the number of target vectors, for dimension n, is $2^n - 1$, therefore allowing for a uniform partitioning of the target vectors by the various tasks (with only the last task decoding one less target vector than all the others). Additionally, the chosen value for P_H leads to a single task for dimensions lower than 15; however, given the relatively low workload for these dimensions, a single task is actually more efficient. As such, the number of tasks NT_H per dimension is given by (3.5).

$$NT_H(n) = \left\lceil \frac{2^n - 1}{P_H} \right\rceil \tag{3.5}$$

The CPU implementation was straightforwardly adapted to this scheme, needing only the information of which task is executing, so as to generate and decode the correct target vectors. A similar situation was encountered in regards to the CUDA implementation, with the main difference being the launch configuration of the modified CUDA kernel. The optimal configuration for this modified kernel was found to be $L_H = 32$ threads per block and T_H blocks, given by (3.6).

$$T_H = \left\lceil \frac{\text{Number of Target Vectors Per Task}}{L_H} \right\rceil = \left\lceil \frac{P_H}{L_H} \right\rceil \tag{3.6}$$

Note, however, that the same caveat as with the stand-alone CUDA implementation applies here (and is not shown in the equation): this launch configuration results in a single target vector being decoded by each thread but, for dimension 20, due to memory constraints, each thread decodes two target vectors.

StarPU also provides a layer of abstraction regarding memory management, by relieving the programmer of the necessity to manually enforce data consistency and availability. To this effect, there is only the need to inform StarPU of the relevant data structures, which then manages said structures between the different memory spaces (even using asynchronous transfers, if possible), and guarantees data consistency and availability.

Since the decoding process is partitioned across different computational units, the results also originate from different sources. To address this, StarPU provides a partitioning function, that allows to split memory structures – in this case, the **N** matrix where the relevant vectors are stored – into as many sub-structures as there are tasks. This is useful so that each task only gets the part of the **N** matrix it requires, and is also a way to inform StarPU that there are no dependencies involving this matrix. After all tasks complete their execution, the matrix is un-partitioned, and all of its pieces are gathered. Note that, as with the stand-alone GPU implementations (OpenACC and CUDA), the smaller **N** matrix simplification of Chapter 3.2 was not used here.

In regards to task scheduling, StarPU provides several models to the user, like a basic policy, where workers draw tasks from a central task queue, to priority-aware policies, to more advanced schedulers based on performance models. These models can be based on historical data from previous executions (for problems where data sizes does not vary significantly), or can be made to use regression functions (for problems where data sizes vary considerably). Given that the Voronoi cell algorithm can be associated to the latter, and because the execution times rise exponentially with the dimension, a non-linear regression-based performance model was used. Naturally, these models require calibration in order to successfully schedule tasks between workers, so for the purpose of registering the execution times of the heterogeneous implementation, an initial calibration run is performed on all test bases, followed by another run on the same bases, using the calibrated model.

Figure 3.7 shows the execution times of the heterogeneous implementation of the Voronoi cell-based algorithm, on Machine B (GPU + 1 CPU core), using the aforementioned non-linear regression perfor-

mance model.

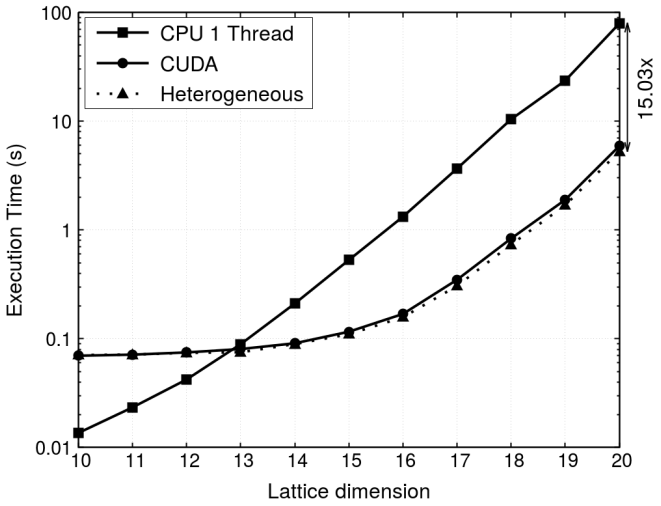


Figure 3.7: Execution times of the heterogeneous implementation of the algorithm, on Machine B (GPU + 1 CPU core), compared against the sequential CPU (Machine A) and stand-alone GPU (Machine B) implementations, lattice dimensions 10 to 20.

The obtained results show that even with the additional overhead from several CUDA kernel launches, and the overhead from StarPU itself, the heterogeneous implementations of the algorithm, using the GPU and only 1 CPU core, is capable of achieving a speedup of $15.03\times$ when compared to the sequential CPU version, an improvement of 9.91% over the stand-alone CUDA implementation.

While the reported speedup is relatively modest, compared to the stand-alone CUDA implementation, additional overhead is present from the added kernel launches and from StarPU itself, and only one CPU core is available for actual work – recall that Machine B's CPU only has 2 cores, and StarPU reserves one for GPU control. As such, if a CPU with a higher physical core count were to be used, the performance of the implementation would surely improve.

The obtained results show that the implementation is capable of taking advantage of heterogeneous architectures, further improving its performance.

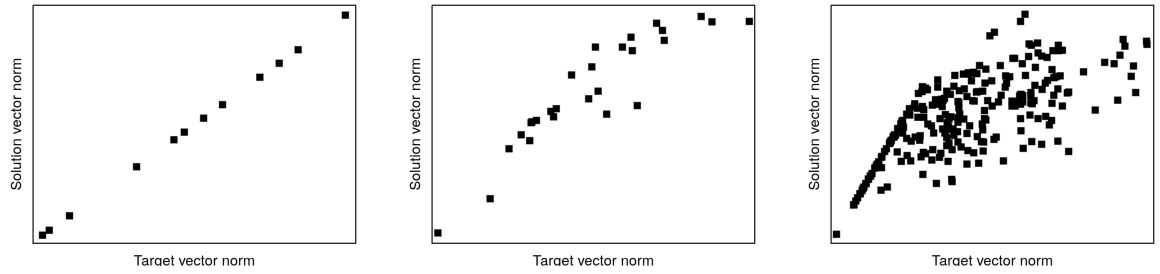
4

Voronoi cell 2.0

Contents

The Voronoi cell 2.0 algorithm consists of a series of optimizations applied to the Voronoi cell-based implementation described in Chapter 3. This is motivated by the fact that, for the purpose of computing the SVP, a full description of the Voronoi cell of a lattice is not needed.

Given the importance of the norm as a metric in lattice-based cryptosystems, the first test performed was to analyze any possible correlation between the norm of a target vector, and its respective relevant vector (note that the result from the decode procedure may **not** be a relevant vector; only target vectors that yielded valid solutions, were taken into consideration in this test). Figure 4.1 shows the correlation of target vectors and their respective relevant vectors, for three different lattice bases.



(a) Scatter plot for dimension 4, seed 960.

(b) Scatter plot for dimension 5, seed 0.

(c) Scatter plot for dimension 8, seed 456.

Figure 4.1: Correlation between the norm of target vectors and relevant vectors, for three different lattice bases.

The scatter plots of Figure 4.1 exemplify the three distinct cases observed for this test: very strong correlation (Figure 4.1a), moderate correlation (Figure 4.1b), and somewhat weak correlation (Figure 4.1c). Even though the correlation may not hold strongly for all bases, it still remains very strong for the shorter norms, where the solution to the SVP (the shortest relevant vector) lies; the shorter target vectors yield the shorter relevant vectors. Note, however, that this correlation is not guaranteed to hold for all bases, in all dimensions; this is an empirical observation and, as such, the optimizations based on this correlation are heuristic. Recall that even approximate variants of the SVP (α-SVP), with an approximation factor α as high as $\sqrt{2}$ were proven to be hard to solve.

The observed correlation motivated a follow-up test: to check whether or not the absolute shortest target vector of a lattice is decoded into the solution of the SVP – the shortest relevant vector. In order to achieve this, the necessity arose to generate all target vectors beforehand, and pick the shortest. While this greatly decreases the execution time of the algorithm, it failed for 20 of the tested bases, and thus confirming that the shortest target vector isn't always decoded into the shortest relevant vector. As such, less extreme optimizations were devised and applied to the algorithm.

4.1 The optimizations

As previously mentioned, the rules described in this chapter are norm-based, as this is one of the most important metrics in lattice algorithms – the base of the mathematical problems that support these cryptographic systems. Specifically, the majority of these rules are used to decide whether or not to decode a certain target vector and, as such, are used to "prune" the potential target vectors, while others are used to decide whether or not to continue the execution of the program.

4.1.1 Gaussian heuristic stopping criterion

The first optimization applied to the algorithm provides a mechanism to abort the execution as soon as a short relevant vector is found. This is accomplished by means of the Gaussian heuristic, an estimate of the length of the shortest vector of a lattice, given by Equation (4.1):

$$\alpha \cdot \frac{\Gamma(n/2+1)^{1/n}}{\sqrt{\pi}} \cdot (\det \mathcal{L})^{1/n} \tag{4.1}$$

where α is a safety margin added to the heuristic, $\Gamma(\cdot)$ is the gamma function (an extension of the factorial function) and n is the dimension of the lattice.

The motivation for this optimization, as previously mentioned, is to allow the program to terminate early, as soon as the (estimated) shortest vector is found – in fact, some implementations of enumeration- and sieving-based solvers do exactly this in order to decrease their running time. For the Gaussian heuristic stopping criterion, a safety added margin of 5% was used ($\alpha = 1.05$), given that that is the value used by the SVP-Challenge in order to accept an entry into their database.

Regarding the implementation of Gaussian pruning, due to the fact that both $\Gamma(n/2+1)$ and $(\det \mathcal{L})$ can be very large, native data types may not suffice to store these numbers. As such, the computation of the Gaussian heuristic is performed using GCC's quad-math library. It provides the `__float128` – a 128-bit, quadruple precision floating-point data type – and the usual mathematical functions, extended to this representation, including the gamma function. The determinant of the lattice is given by the product of the diagonal entries of the triangular matrix **R** from the QR decomposition. Afterwards, however, the result is small enough to be stored in a native data type and is then compared against the norm of the relevant vectors computed during the main loop of the algorithm. If a relevant vector with norm equal to or smaller than the heuristic is found, the loop is terminated, and the program finished.

Simply using this optimization without any order to the target vectors, however, led to finding the shortest vector only for 87.6% of all tested bases, which motivated the strategy that follows.

4.1.2 Gaussian heuristic stopping criterion with (target vector) pre-sorting

Motivated by the relatively low success rate of the Gaussian heuristic stopping criterion, target vector pre-sorting was applied to this strategy, i.e. generating all target vectors beforehand – as opposed to all other implementations, where the target vectors are computed on-the-fly during the main loop of the algorithm – and then sorting them by increasing norm. A simple `struct` was implemented to allow the storage of both a target vector and its norm, to avoid the necessity of computing the latter several times during the sorting process. As such, the full target vector list is an array of this `struct`, and is sorted using C++'s standard library function `sort`.

This way, due to the observed correlation between the norm of the target vectors and their respective relevant vectors, the shorter target vectors are decoded first, therefore (potentially) avoiding unnecessary computations. This is done to maximize the probability of finding short, relevant vectors early on. Note that because all target vectors must be generated before being sorted, the memory usage of this variant increases considerably, given that there are $2^n - 1$ target vectors for dimension n. The term pre-sorting will be used to refer to the process of generating and sorting the target vectors before the main loop of the algorithm. The full motivation for target vector pre-sorting is discussed in Chapter 4.1.6.

Figure 4.2 shows the execution times of the Gaussian heuristic stopping criterion strategy with pre-sorting, compared against the sequential CPU implementation.

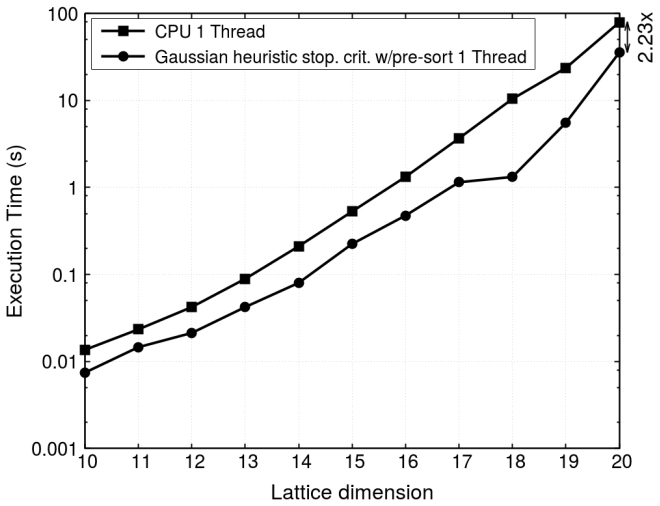


Figure 4.2: Execution times of Gaussian heuristic stopping criterion (added margin of 5%), with pre-sorting, on Machine A, lattice dimensions 10 through 20.

Not only does pre-sorting accelerate this strategy – almost twice as fast, with a maximum improvement of $1.94\times$ over the version without pre-sorting – it also computes the correct solution to the SVP for *all* bases tested. This success rate further validates that the shorter target vectors lead to short relevant vectors.

The maximum speedup attained with the Gaussian heuristic stopping criterion with pre-sorting, com-

pared to the sequential CPU implementation is of $7.88\times$, for dimension 18 (the speedup value shown in the figure is for dimension 20).

4.1.3 Simple pruning

The first form of pruning, henceforth named “simple pruning”, consists in skipping the decode procedure for target vectors whose norm is greater than that of the shortest relevant vector found up to that point. In theory, this form of pruning may discard a target vector that results in an even shorter relevant vector, given that no analysis was performed regarding the individual relation between the absolute values of the norms of a target vector and its respective relevant vector.

Implementation wise, this is achieved by storing the value of the shortest relevant vector found (in a variable named `ShortestNormFound`, for example, which is already present from the smaller **N** matrix modification of Chapter 3.2), and simply comparing the norm of the target vector to be decoded against it. The `ShortestNormFound` variable is initialized to infinity and updated every time a shorter relevant vector is computed. The execution times of simple pruning, compared to the parallel CPU implementation, are shown in Figure 4.3.

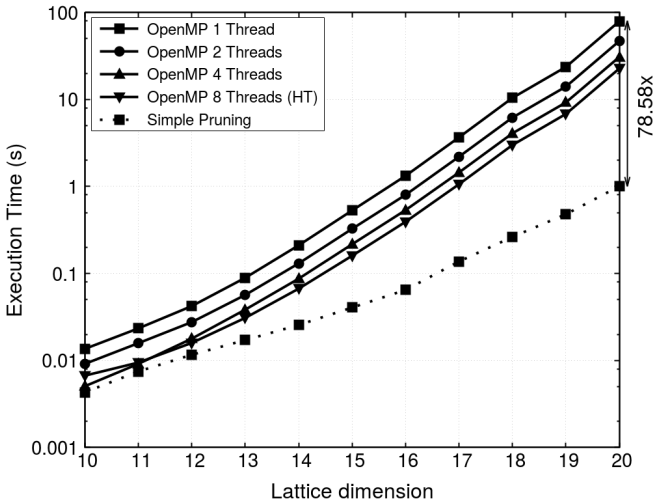


Figure 4.3: Execution times of simple pruning compared to the parallel CPU implementation, on Machine A, lattice dimensions 10 through 20.

In practice, this has proven to be an effective form of pruning, yielding the correct result in 99.85% of the bases tested, and attaining a maximum speedup of $78.58\times$, when compared to the sequential CPU implementation.

4.1.4 Gaussian pruning

The second form of pruning – Gaussian pruning – is named after and based on the Gaussian heuristic, which is an estimate of the norm of the shortest vector of a lattice, as previously mentioned, given by (4.1). Similarly to simple pruning, target vectors are discarded if their norm is greater than the Gaussian heuristic. This form of pruning retains the concerns of simple pruning in regards to the certainty of the solution.

Contrarily to the Gaussian heuristic stopping criterion, where the Gaussian heuristic was used to compare against the norm of the computed relevant vectors, in this strategy it is used to compare against the norm of the target vectors to be decoded. Note that even though both optimizations use the Gaussian heuristic, their values are stored in different variables so as to not interfere with one another, and to allow different safety margins to be used.

After testing, an added margin of 15.5% was chosen, as it provided a good balance between correctness and performance. Figure 4.4 shows the execution times of the Gaussian heuristic, compared against the parallel CPU implementation.

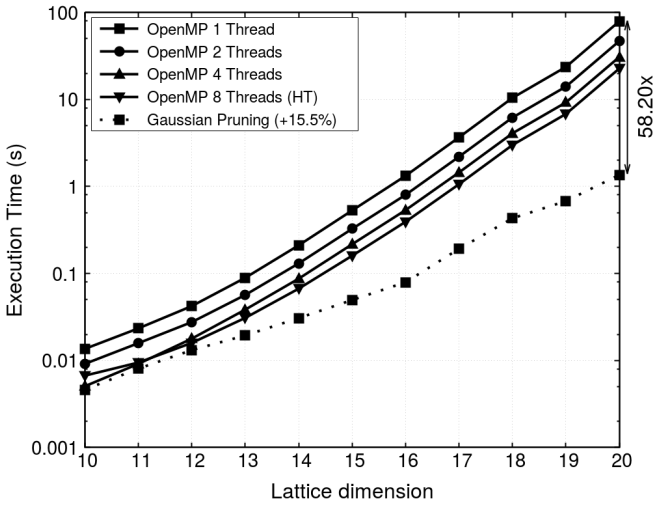


Figure 4.4: Execution times of Gaussian pruning (with an added margin of 15.5%) compared to the parallel CPU implementation, on Machine A, lattice dimensions 10 through 20.

This form of pruning computed the correct result for 99.85% of all bases tested, and achieved a maximum speedup of $58.20\times$, when compared against the sequential CPU implementation.

4.1.5 Combined pruning

Given the effectiveness of both simple and Gaussian pruning, a version that combines both forms was devised, thus potentially increasing the amount of pruning.

In regards to the implementation, because the pruning conditions are checked independently, there

are two possible orders: either perform simple pruning followed by Gaussian pruning, or the other way around. As with Gaussian pruning, a safety margin of 15.5% was used, so that the performance of the various forms of pruning may be directly comparable. Figure 4.5 shows the execution times of the combined pruning variant, by both orders, compared to the parallel CPU implementation.

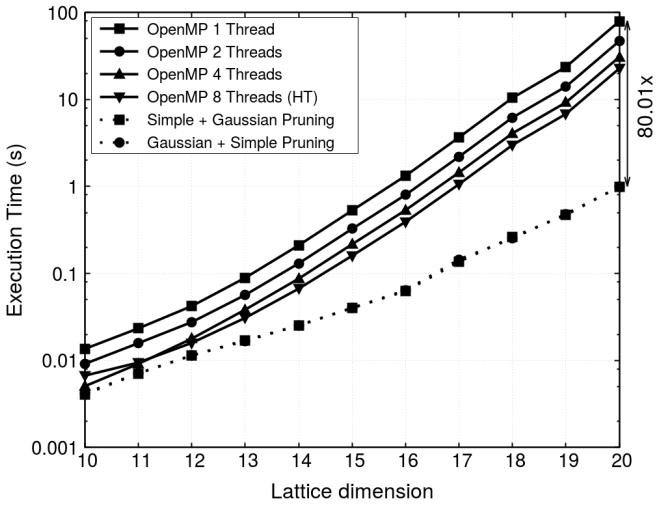


Figure 4.5: Execution times of combined pruning (with an added margin of 15.5%, both orders) compared to the parallel CPU implementation, on Machine A, lattice dimensions 10 through 20.

The results depicted show that there is no meaningful difference to the order by which the individual pruning strategies are applied, attaining similar performance. The combined pruning strategies computed the correct solution for 99.84% of all bases tested, and achieved a maximum speedup of $80.01\times$, compared against the sequential CPU implementation.

4.1.6 Combined pruning with (target vector) pre-sorting

Simple pruning relies on the norm of the computed relevant vectors as a comparison criterion. In the worst case scenario, this form of pruning may end up taking the same amount of time as no pruning at all, if the norm of the relevant vectors being computed keeps decreasing and is always larger than the norm of the target vectors. As shown earlier, there is a strong correlation between the norm of the target and relevant vectors for the shorter norms, i.e. decoding target vectors with shorter norms will generally lead to shorter relevant vectors as well. If the algorithm starts by decoding the target vectors with shorter norms then, due to the observed correlation, the norm of the computed relevant vectors will be “short” too. Thus, target vector pre-sorting potentially increases the pruning extent given that target vectors with larger norms would, almost inevitably, be superseded by those of smaller norms, therefore avoiding unnecessary decodes.

This way, as mentioned previously, the shorter target vectors are decoded first, therefore (potentially) avoiding unnecessary computations. Note that, as with the Gaussian heuristic stopping criterion with

(target vector) pre-sorting, the memory usage of this strategy increases considerably, when compared to the variant without pre-sorting.

Given the fact that the Gaussian heuristic is a constant value for a given lattice basis, pre-sorting does not change the target vectors that are decoded, merely the order by which they are so. Only simple and combined pruning *may* benefit from this.

Figure 4.6 shows the execution times of the combined pruning with pre-sorting, by both orders and with an added margin of 15.5%, compared against the combined pruning without pre-sorting.

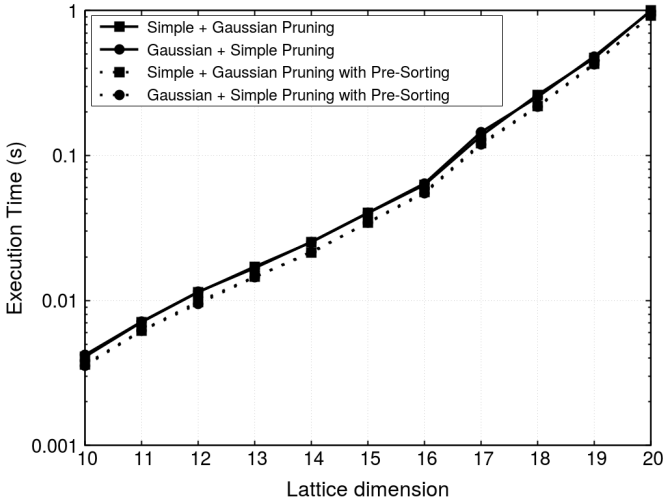


Figure 4.6: Execution times of combined pruning (with an added margin of 15.5%, both orders), with and without pre-sorting, on Machine A, lattice dimensions 10 through 20.

Pre-sorting the target vectors by increasing norm is indeed beneficial, improving the performance of the combined pruning strategy by an additional 22.32% in the best case, and maintaining a success rate of 99.84%. However, and as mentioned previously, the trade-off of this improvement is the increased memory usage of the implementation.

Common to all forms of pruning presented, is the maximum difference between the norm of the computed solution and the norm of the correct one, for the bases that failed to yield the correct result, at 4.86%. This value is within the margin of the 1.05-SVP (5% α-SVP), therefore confirming the potential of these forms of pruning.

Table 4.1 summarizes the results obtained with the various optimizations devised and implemented.

As the table shows, the best result was obtained with the combined pruning with pre-sorting optimization, achieving a maximum speedup of $86.10\times$, compared to the sequential CPU implementation, while failing to compute the shortest lattice vector for merely 0.16% of the tested bases. Furthermore, and as previously mentioned, for the bases that did not yield the shortest vector, the maximum difference between the norm of the computed solution and the actual shortest vector was of 4.86%. Note that the maximum speedup was achieved in dimension 20 for all optimizations, except for the Gaussian stopping

Table 4.1: Maximum speedup factors attained for the various strategies devised.

Optimization	Maximum speedup factor	Success rate
Gaussian stopping criterion with pre-sorting	$7.88\times$	100%
Simple pruning	$78.58\times$	99.85%
Gaussian pruning	$58.20\times$	99.85%
Combined pruning	$80.01\times$	99.84%
Combined pruning with pre-sorting	$86.10\times$	99.84%

criterion with pre-sorting, that yielded the maximum speedup in dimension 18.

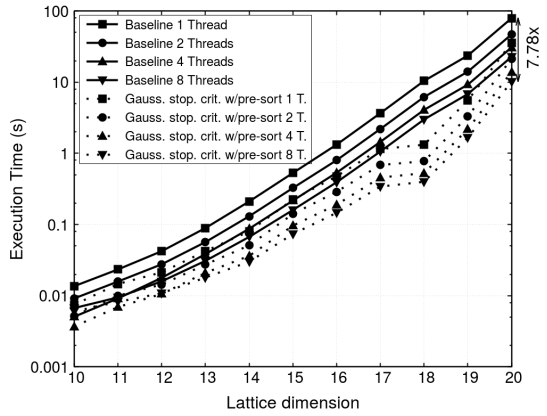
4.2 Parallel implementations

Parallel version of these optimizations were implemented as well, although only for the CPU, due to the fact that OpenACC does not provide a critical region like OpenMP, and CUDA discourages such constructs as well.

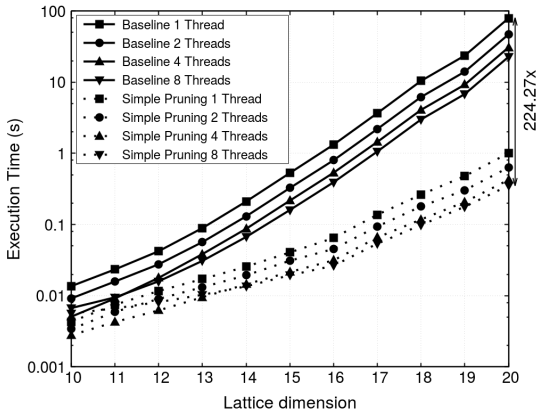
In practice, this parallel version was virtually ready when the optimizations were implemented, given the fact that its parallelization consists of applying the OpenMP directives to the main loop of the algorithm, like in the parallel CPU implementation of the original algorithm, described in Chapter 3.3. Furthermore, the only operation that must be done inside a critical region is the computation of the shortest norm found, which is already performed as part of the smaller **N** matrix simplification, described in Chapter 3.2. Additionally, because GCC implements a parallel `sort`, so can the target vector pre-sorting process be performed in parallel.

Figure 4.7 shows the performance of the parallel versions of the Gaussian heuristic stopping criterion with pre-sorting (4.7a), simple pruning (4.7b), Gaussian pruning (4.7c), combined pruning without pre-sorting (4.7d) and combined pruning with pre-sorting (4.7e), compared against the baseline parallel CPU implementation. Note that for the combined pruning strategy, only one order is shown, as they were proven to be virtually identical.

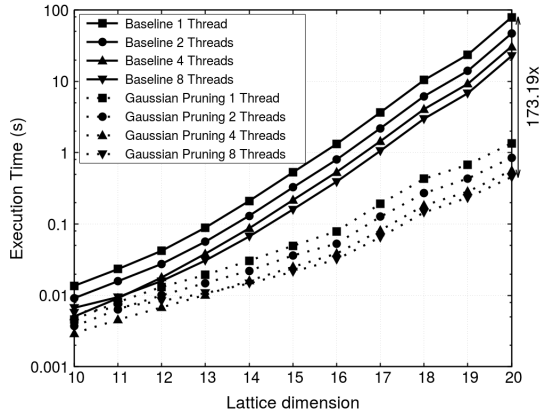
Similarly to the baseline parallel CPU implementation, so do the parallel optimized variants achieve further speedups, up to $256.51\times$, compared to the baseline sequential CPU version.



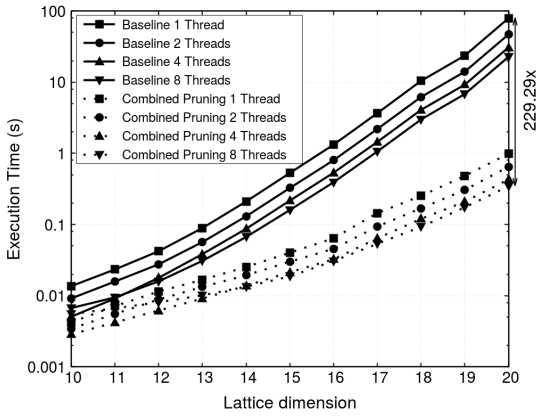
(a) Parallel Gaussian heuristic stopping criterion ($+5\%$ safety margin) with Pre-Sorting



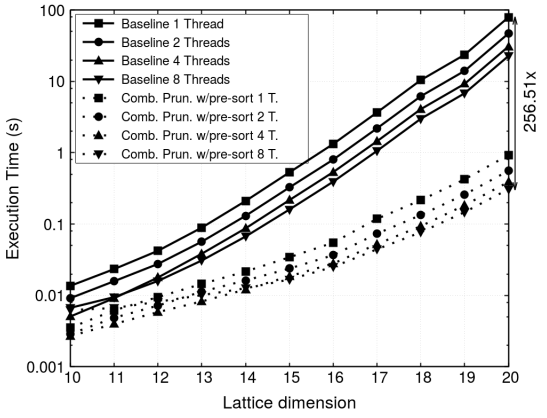
(b) Parallel Simple Pruning



(c) Parallel Gaussian Pruning ($+15.5\%$ safety margin)



(d) Parallel Combined Pruning without Pre-Sorting ($+15.5\%$ safety margin)



(e) Parallel Combined Pruning with Pre-Sorting ($+15.5\%$ safety margin)

Figure 4.7: Parallel Gaussian heuristic stopping criterion with pre-sorting, simple pruning, Gaussian pruning and combined (with and without pre-sorting) pruning, compared against the baseline parallel CPU implementation on Machine A, lattice dimensions 10 to 20.

5

Exploring the limits of the LUSA library

Contents

This chapter is dedicated to the Lattice Unified Set of Algorithms (LUSA), a High Performance Computing (HPC) library for lattice-related problems. Specifically, LUSA's performance is compared against other libraries that are usually used in the context of lattice-based cryptography, such as NTL [44] and fplll [45]. LUSA is available at `http://alfa.di.uminho.pt/~ammm/lusa.html`.

The main contribution to the LUSA library was the integration of the work presented in this dissertation, the integration of an enumeration with extreme pruning routine, the writing of the documentation, the standardization of the function interfaces, and the preparation of the code as a ready-to-use library.

5.1 The LUSA library

The main goals of LUSA are performance and simplicity: not only it aims to offer comparable or better performance than similar libraries and parallel implementations of most routines, but also to be simple to both install and use, depending upon no other external library, and having clear methods for the functions it provides.

Lattice bases are often represented by large numbers, i.e. numbers that can have dozens, if not hundreds of digits. This means that the native data types of C/C++ (the computer language LUSA is implemented in) are not capable of handling such numbers. As such, LUSA provides a custom multi-precision data type for integers, and an extended exponent, double precision data type for real numbers (same precision as the native double data type, but with a larger exponent). These "big numbers" are usually handled with a third-party library, such as the GNU Multiple Precision (GMP) library, but for ease of installation – part of LUSA's second goal – custom data types are built-in.

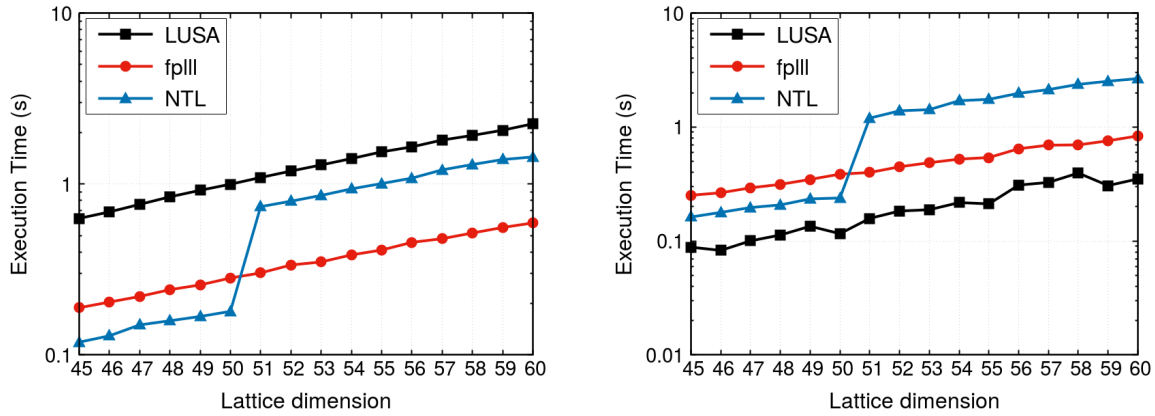
5.2 Testing LUSA

This section shows the performance of LUSA compared against similar libraries – NTL and fplll. The testing procedure in this chapter is identical to the one described in Chapter 3, differing only in the number of bases tested in each dimension that, unless otherwise stated, is set at 10 bases per dimension.

5.2.1 Basis reduction

All of LUSA's routines expect the basis to have been LLL-reduced and converted to C/C++'s native data types, so this is the most computationally-intensive step of the library. LUSA provides methods for both LLL and BKZ reductions, using both exact and approximate arithmetic. The fastest available LLL method is an heuristic, floating-point variant, based on Schnorr and Euchner's work [12]. Figure 5.1a shows the performance of LUSA's LLL reduction against NTL and fplll.

As mentioned previously, all LUSA's routines expect the basis to be LLL-reduced, and that is the case for the BKZ reduction as well. Figure 5.1b shows the performance of LUSA's BKZ reduction against NTL and fplll.



(a) Comparison of the performance of the LLL reduction method in LUSA, NTL and fplll, lattice dimensions 45 to 60.

(b) Comparison of the performance of the BKZ reduction method in LUSA, NTL and fplll, using block size 20, lattice dimensions 45 to 60.

Figure 5.1: Comparison of various reduction routines present in LUSA, NTL and fplll, on Machine A.

The gap in the execution times of NTL's implementations is due to the fact that, up to dimension 50, double precision versions of the reductions are used but, for dimensions 51 and higher, due to the recommendation of the library, extended exponent double precision variants are utilized instead in order to handle the large numbers involved. The figures show that while LUSA's LLL implementation is slightly slower than both fplll's and NTL's, LUSA's BKZ implementation is faster than the others. This can be attributed to the fact that LUSA's BKZ implementation assumes that the basis is already LLL-reduced, therefore starting with an advantage over the other libraries.

5.2.2 Enumeration functions

Currently, LUSA implements a non-pruned variant of enumeration, based on the algorithm proposed in [12], with a parallel version also available. Figure 5.2 shows the execution times of the enumeration function, compared with fplll's enumeration.

The figure shows that LUSA's enumeration is slightly slower than fplll's, when using a single thread, but that can be attributed to the necessary changes of LUSA's implementation to support multi-threading. However, the multi-threaded performance of LUSA scales well with the number of threads used, surpassing fplll's single-threaded implementation. Note that even though it is beneficial, Hyper-Threading provides a more modest speedup than an actual physical core.

LUSA also provides an enumeration with extreme pruning routine, based on [23]. Because fplll does not provide an enumeration with extreme pruning implementation – at least directly, from the pre-

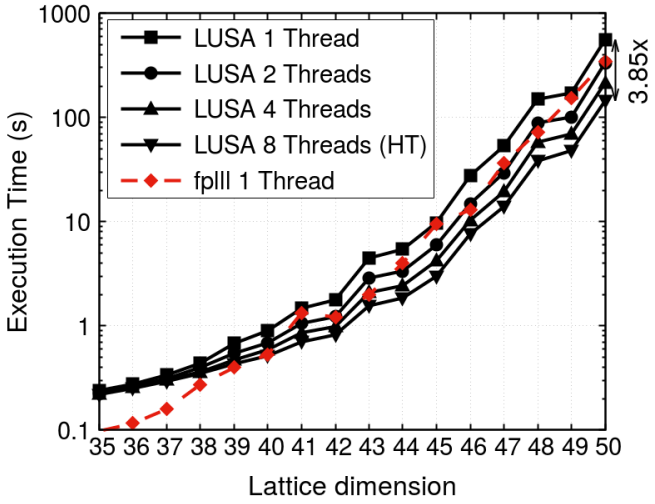


Figure 5.2: Comparison of the performance of the non-pruned enumeration routines present in LUSA using 1, 2, 4 and 8 threads, and fplll using 1 thread, on Machine A, lattice dimensions 35 to 50.

compiled executable – an alternative is used to showcase the performance of LUSA's implementation. The plll library [46] (distinct from the fplll library) offers lattice-related implementations of several algorithms, notably enumeration with extreme pruning based on [23]. The execution times of LUSA's implementation, compared against plll's, are shown in Figure 5.3.

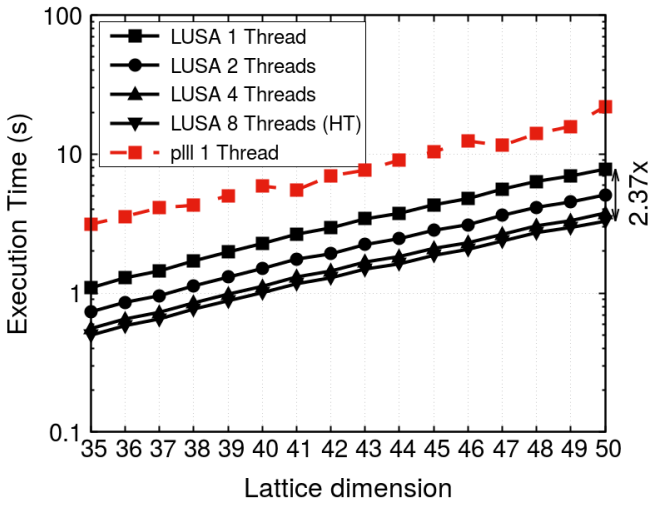


Figure 5.3: LUSA's enumeration with extreme pruning routine using 1, 2, 4 and 8 threads, compared against plll's implementation using 1 thread, on Machine A, lattice dimensions 35 to 50.

The figure shows a better overall performance of LUSA's implementation, for all dimensions tested. Further speedups are attained from the use of multiple cores, achieving a maximum speedup of $2.37\times$, when compared to the sequential implementation.

5.2.3 Sieving functions

Regarding sieving algorithms, LUSA currently provides a multi-threaded HashSieve implementation, based on the work by Mariano et al. [37, 47]. Figure 5.4 shows the execution times of this routine using a variable number of threads, compared to a reference implementation.

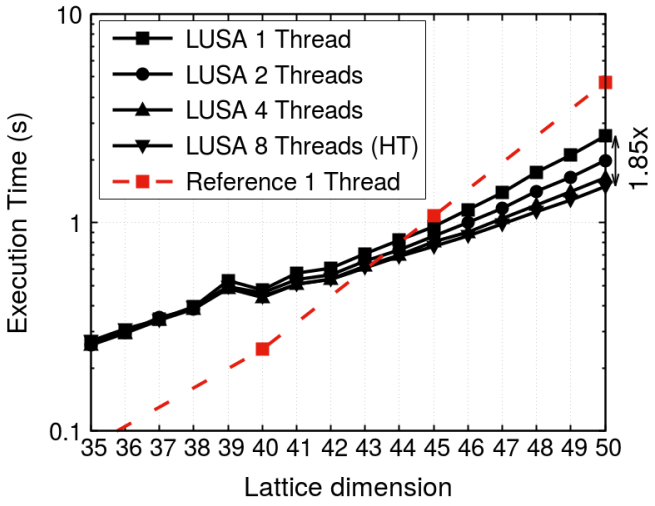


Figure 5.4: LUSA's HashSieve routine using 1, 2, 4 and 8 threads, compared to the reference HashSieve implementation using 1 thread, on Machine A, lattice dimensions 35 to 50.

Due to the fact that fplll does not implement an HashSieve function, LUSA's performance is compared against Laarhoven's publicly available sequential HashSieve implementation [48]. Even though multi-threading does not appear to be beneficial in lower dimensions, the execution times start to diverge in higher dimensions, showing the multi-threaded approach to be advantageous as the dimension of the lattice increases. Similarly to enumeration, and despite still improving upon performance, Hyper-Threading provides a more modest speedup than an actual physical core.

5.3 Integrating Voronoi cell in LUSA

Integrating the Voronoi cell algorithm (and the enumeration with extreme pruning one, as well) in LUSA meant that several functions had to be changed or implemented from scratch. The basis reduction step, previously used the NTL, is now performed by LUSA's own routines. At one point, the Voronoi cell algorithm requires the computation of the inverse of a matrix; this had to be performed from scratch, as LUSA did not have such a function implemented yet.

The Gaussian heuristic (Equation 4.1, necessary for the Gaussian pruning, as described in Chapter 4) depends upon the computation of the determinant of the basis matrix (which may overflow native data types), followed by a power operation with real exponent, neither of which were implemented in LUSA.

One way of computing the determinant of a matrix is through its QR decomposition, specifically through the product of the diagonal entries of the triangular matrix **R**, as mentioned in Chapter 4.1.4. Since the QR decomposition was already necessary for the algorithm, the computation of the determinant was straightforwardly implemented, with the potential overflow avoided using LUSA's extended-exponent double precision data type. Regarding the power function (with real exponent), it can be computed using the square root function, the latter of which is already present in LUSA. The results of the implementation were validated after the integration, to guarantee that no errors were introduced during the process.

Figure 5.5 shows the execution times of the integrated Voronoi cell-based implementation. The only known public implementation of a Voronoi cell-based algorithm is in the plll library, which implements Micciancio and Voulgaris' algorithm, presented in [43]. Due to the higher execution times of plll's implementation, only results up to dimension 18 are shown and, for dimensions 15 and up, only one lattice basis is tested per dimension.

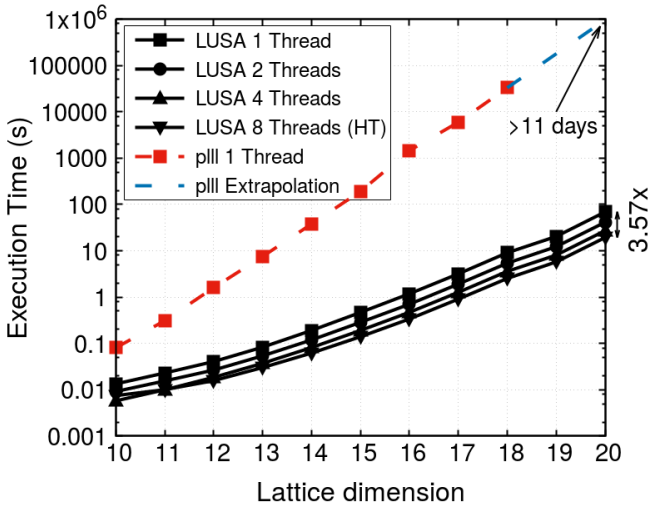


Figure 5.5: LUSA's Voronoi cell-based routine using 1, 2, 4 and 8 threads, compared against plll's Voronoi cell-based implementation using 1 thread, on Machine A, lattice dimensions 10 to 20.

The figure shows a virtually identical scalability as the stand-alone Voronoi cell-based implementation, presented in Chapter 3.3, reporting a maximum speedup of $3.57\times$, for 8 threads.

The results presented in this chapter show that LUSA's implementations are generally faster than the usual alternatives, aim to be simpler to use and, above all, offer parallel versions that are not available on the other major libraries.

6

Conclusions

The work conducted for this dissertation showed that is indeed possible to accelerate a Voronoi cell-based algorithm for the SVP. Furthermore, the various implementations described in this dissertation feature general-purpose programming optimizations, aimed at improving the performance of the applications, such as the use of larger, single-dimension arrays for the storage of matrices, thus accelerating memory accesses. Additionally, algorithm-specific optimizations were employed, such as the **N** matrix simplification, that allowed to severely decrease the memory requirements of the CPU implementations.

Parallel implementations of the algorithm were also devised for the CPU, GPU, and heterogeneous (CPU + GPU) architectures. As expected, due to the independent nature of the iterations of the main loop of the algorithm, the parallel CPU version, implemented using OpenMP, achieves linear speedups when using physical cores, and also benefits from the use of Hyper-Threading, although by a smaller degree. As such, this version yields a maximum speedup of $5.44\times$ using 8 threads (HT, Turbo Boost disabled). Concerning the GPU, both OpenACC and CUDA versions were implemented, with the former achieving a maximum speedup of $12.15\times$, despite the fact that the enumeration-based CVP-solver, upon which the algorithm is based, is essentially divergent in execution. Regarding the CUDA implementation, several architecture-specific optimizations were employed, such as the use of asynchronous memory transfers (when possible), the use of constant memory for applicable data, and a single, larger memory allocation for all required private structures (later partitioned using pointer arithmetic), all contributing to a further improvement over the OpenACC version, with a reported speedup of $13.37\times$ over the sequential CPU version. An heterogeneous CPU + GPU variant was also devised, in order to take advantage of both processing units simultaneously, by making use of performance models in order to better suit the computational needs of the implementation to the available hardware. Despite the added overhead from more finely-grained partitioning of work, compared to the standalone CUDA implementation, the help of a single CPU thread results in a maximum improvement of 9.91% over the standalone CUDA version. As such, the heterogeneous implementation outperformed the standalone CUDA implementation, and attained a maximum speedup of $15.03\times$ using the GPU and a single CPU thread, when compared to the baseline CPU implementation.

A stopping criterion for the algorithm was also implemented, based on the Gaussian heuristic – an estimate of the length of the shortest vector of a lattice – stopping the execution of the program as soon as a relevant vector whose norm is less than or equal to the Gaussian heuristic (with an added safety margin). This stopping criterion was implemented both with and without target vector pre-sorting but, given the relatively low success rate of the latter (87.6%), only the former is of practical interest. As such, the Gaussian heuristic stopping criterion with (target vector) pre-sorting achieved a maximum speedup of $7.88\times$, while computing the correct solution for the SVP for all tested bases. Simple pruning, the first norm-based pruning strategy devised, allowed for the pruning of the target vectors by using information that was readily available on the original algorithm – the norm of the relevant vectors. This

form of pruning achieved a maximum speedup of $78.58\times$, with a success rate of 99.85%, confirming the potential of norm-based pruning. The second form of pruning implemented, Gaussian pruning, relies on the computation of the Gaussian heuristic and uses this heuristic to prune the target vectors, achieving a maximum speedup of $58.20\times$, with the same 99.85% success rate. Motivated by the effectiveness of both optimizations, a combined form was implemented that, intuitively, should further increase the amount of pruning to be made. This was proved to be true, with the combined pruning achieving a maximum speedup of $80.01\times$, better than any of the individual strategies, while the success rate was virtually unchanged at 99.84%. Given the observed correlation between the norm of the target vectors and their respective relevant vectors, and also due to the characteristics of simple pruning, generating and sorting the target vectors (by increasing norm) before the main loop of the algorithm could, theoretically, prove to be advantageous. This was indeed the case, with the last form of pruning – combined pruning with (target vector) pre-sorting – further improving upon the performance of combined pruning, despite having to spend time sorting the target vectors, yielding a maximum speedup of $86.10\times$, while maintaining the success rate of combined pruning, at 99.84%, thus becoming the fastest pruning strategy developed. Furthermore, the norm of the solution of all the bases that did not yield the correct shortest vector were, at most, 4.86% larger than that of the actual solution for the SVP, for the given lattice. As such, even when the correct, shortest vector was not found, a *short* vector was computed, still within the bounds of the approximate 5% α-SVP.

Additionally, parallel versions of the pruned variants were devised for the CPU, implemented using OpenMP. Similarly to the non-pruned implementation, further speedups are achieved through parallelization. The parallel CPU implementation of the combined pruning strategy with (target vector) pre-sorting – the fastest pruning strategy devised – achieves a maximum speedup of $256.51\times$ when using 8 threads (HT), compared against the sequential, non-pruned CPU implementation.

Despite still being far from current state-of-the-art dimensions, this work proves that there are still improvements to be made in Voronoi cell-based algorithms, and that this family of algorithms should not yet be discarded.

Bibliography

[1] P. W. Shor, “Algorithms for quantum computation: Discrete logarithms and factoring,” in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, ser. SFCS ’94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 124–134.

[2] ——, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997.

[3] D. J. Bernstein, J. Buchmann, and E. Dahmen, *Post Quantum Cryptography*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[4] D. Micciancio and O. Regev, “Lattice-based cryptography,” in *Post-Quantum Cryptography*, D. J. Bernstein, J. Buchmann, and E. Dahmen, Eds. Springer Berlin Heidelberg, 2009, pp. 147–191.

[5] P. Martins, L. Sousa, and A. Mariano, “A survey on fully homomorphic encryption: An engineering perspective,” *ACM Comput. Surv.*, vol. 50, no. 6, pp. 83:1–83:33, Dec. 2017.

[6] P. van Emde Boas, *Another NP-complete Partition Problem and the Complexity of Computing Short Vectors in a Lattice*, ser. Mathematical preprints series. Universiteit van Amsterdam. Mathematisch Instituut, 1981.

[7] M. Ajtai, “The shortest vector problem in L2 is NP-hard for randomized reductions (extended abstract),” in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, ser. STOC ’98. New York, NY, USA: ACM, 1998, pp. 10–19.

[8] D. Micciancio, “The shortest vector in a lattice is hard to approximate to within some constant,” *SIAM J. Comput.*, vol. 30, no. 6, pp. 2008–2035, Dec. 2001.

[9] S. Khot, “Hardness of approximating the shortest vector problem in lattices,” in *45th Annual IEEE Symposium on Foundations of Computer Science*, Oct 2004, pp. 126–135.

[10] A. K. Lenstra, H. W. Lenstra, and L. Lovász, “Factoring polynomials with rational coefficients,” *Mathematische Annalen*, vol. 261, no. 4, pp. 515–534, Dec 1982.

[11] C. P. Schnorr, “A hierarchy of polynomial time lattice basis reduction algorithms,” *Theoretical Computer Science*, vol. 53, no. 2, pp. 201 – 224, 1987.

[12] C. P. Schnorr and M. Euchner, “Lattice basis reduction: Improved practical algorithms and solving subset sum problems,” *Math. Program.*, vol. 66, no. 2, pp. 181–199, Sep. 1994.

[13] P. Q. Nguyen and D. Stehlé, “An LLL algorithm with quadratic complexity,” *SIAM J. Comput.*, vol. 39, no. 3, pp. 874–903, Aug. 2009.

[14] Y. Chen and P. Q. Nguyen, “BKZ 2.0: Better lattice security estimates,” in *Proceedings of the 17th International Conference on The Theory and Application of Cryptology and Information Security*, ser. ASIACRYPT’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–20.

[15] Y. Aono, Y. Wang, T. Hayashi, and T. Takagi, “Improved progressive BKZ algorithms and their precise cost estimation by sharp simulator,” in *Advances in Cryptology – EUROCRYPT 2016*, M. Fischlin and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 789–819.

[16] D. Micciancio and M. Walter, “Practical, predictable lattice basis reduction,” in *Proceedings, Part I, of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9665*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 820–849.

[17] M. Pohst, “On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications,” *SIGSAM Bull.*, vol. 15, no. 1, pp. 37–44, Feb. 1981.

[18] R. Kannan, “Improved algorithms for integer programming and related lattice problems,” in *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, ser. STOC ’83. New York, NY, USA: ACM, 1983, pp. 193–206.

[19] U. Fincke and M. Pohst, “Improved methods for calculating vectors of short length in a lattice, including a complexity analysis,” *Mathematics of Computation*, vol. 44, no. 170, pp. 463–471, 1985.

[20] R. Kannan, “Minkowski’s convex body theorem and integer programming,” *Math. Oper. Res.*, vol. 12, no. 3, pp. 415–440, Aug. 1987.

[21] E. Agrell, T. Eriksson, A. Vardy, and K. Zeger, “Closest point search in lattices,” *IEEE Transactions on Information Theory*, vol. 48, no. 8, pp. 2201–2214, Aug 2002.

[22] A. Ghasemmehdi and E. Agrell, “Faster recursions in sphere decoding,” *IEEE Transactions on Information Theory*, vol. 57, no. 6, pp. 3530–3536, June 2011.

[23] N. Gama, P. Q. Nguyen, and O. Regev, “Lattice enumeration using extreme pruning,” in *Proceedings of the 29th Annual International Conference on Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 257–278.

[24] D. Micciancio and M. Walter, “Fast lattice point enumeration with minimal overhead,” in *Proceedings of the Twenty-sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’15. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2015, pp. 276–294.

[25] Ö. Dagdelen and M. Schneider, “Parallel enumeration of shortest lattice vectors,” in *Euro-Par 2010 - Parallel Processing*, P. D’Ambra, M. Guarracino, and D. Talia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 211–222.

[26] J. Hermans, M. Schneider, J. Buchmann, F. Vercauteren, and B. Preneel, “Parallel shortest lattice vector enumeration on graphics cards,” in *Proceedings of the Third International Conference on Cryptology in Africa*, ser. AFRICACRYPT’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 52–68.

[27] P.-C. Kuo, M. Schneider, O. Dagdelen, J. Reichelt, J. Buchmann, C.-M. Cheng, and B.-Y. Yang, “Extreme enumeration on GPU and in clouds: How many dollars you need to break SVP challenges,” in *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 176–191.

[28] F. Correia, A. Mariano, A. Proença, C. Bischof, and E. Agrell, “Parallel improved Schnorr-Euchner enumeration SE++ for the CVP and SVP,” in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Feb 2016, pp. 596–603.

[29] M. Ajtai, R. Kumar, and D. Sivakumar, “A sieve algorithm for the shortest lattice vector problem,” in *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, ser. STOC ’01. New York, NY, USA: ACM, 2001, pp. 601–610.

[30] P. Nguyen and T. Vidick, “Sieve algorithms for the shortest vector problem are practical,” *Journal of Mathematical Cryptology*, vol. 2, no. 2, p. 181–207, Jul 2008.

[31] D. Micciancio and P. Voulgaris, “Faster exponential time algorithms for the shortest vector problem,” in *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’10. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2010, pp. 1468–1480.

[32] B. Milde and M. Schneider, “A parallel implementation of GaussSieve for the shortest vector problem in lattices,” in *Proceedings of the 11th International Conference on Parallel Computing Technologies*, ser. PaCT’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 452–458.

[33] T. Ishiguro, S. Kiyomoto, Y. Miyake, and T. Takagi, “Parallel Gauss Sieve algorithm: Solving the SVP challenge over a 128-dimensional ideal lattice,” in *Proceedings of the 17th International Conference*

on Public-Key Cryptography — PKC 2014 - Volume 8383. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 411–428.

[34] A. Mariano, S. Timnat, and C. Bischof, "Lock-free GaussSieve for linear speedups in parallel high performance SVP calculation," in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, Oct 2014, pp. 278–285.

[35] J. W. Bos, M. Naehrig, and J. V. D. Pol, "Sieving for shortest vectors in ideal lattices: a practical perspective," *International Journal of Applied Cryptography*, vol. 3, no. 4, pp. 313–329, 2017.

[36] T. Laarhoven, "Sieving for shortest vectors in lattices using angular locality-sensitive hashing," in *Advances in Cryptology – CRYPTO 2015*, R. Gennaro and M. Robshaw, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 3–22.

[37] A. Mariano, C. Bischof, and T. Laarhoven, "Parallel (probable) lock-free Hash Sieve: A practical sieving algorithm for the SVP," in *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*, ser. ICPP '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 590–599.

[38] A. Becker, L. Ducas, N. Gama, and T. Laarhoven, "New directions in nearest neighbor searching with applications to lattice sieving," in *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '16. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2016, pp. 10–24.

[39] A. Mariano, T. Laarhoven, and C. Bischof, "A parallel variant of LDSieve for the SVP on lattices," in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, March 2017, pp. 23–30.

[40] L. Ducas, M. Albrecht, E. Postlethwaite, G. Herold, E. Kirshanova, and M. Stevens, "The generalized sieve kernel," http://malb.io/discrete-subgroup/slides/2018-09-24-ducas.pdf, Sep 2018, presentation, accessed: 2018-10-07.

[41] H. Minkowski, "Allgemeine lehrsätze über die convexen polyeder," *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, vol. 2, pp. 198–220, 1897.

[42] E. Viterbo and E. Biglieri, "Computing the Voronoi cell of a lattice: the diamond-cutting algorithm," *IEEE Transactions on Information Theory*, vol. 42, no. 1, pp. 161–171, Jan 1996.

[43] D. Micciancio and P. Voulgaris, "A deterministic single exponential time algorithm for most lattice problems based on Voronoi cell computations," in *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, ser. STOC '10. New York, NY, USA: ACM, 2010, pp. 351–358.

[44] V. Shoup, “The Number Theory Library,” 2018, available at https://www.shoup.net/ntl/.

[45] The FPLLL development team, “fplll, a lattice reduction library,” 2016, available at https://github.com/fplll/fplll.

[46] U. of Zurich, “The plll lattice reduction library,” 2014, available at https://felix.fontein.de/plll/.

[47] A. Mariano and C. Bischof, “Enhancing the scalability and memory usage of HashSieve on multi-core CPUs,” in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Feb 2016, pp. 545–552.

[48] T. Laarhoven, “The HashSieve algorithm,” 2015, available at https://github.com/tmmlaarhoven/hashsieve.

Annex A

Optimizing the Memory Usage of Voronoi Cell-based Parallel Kernels for the Shortest Vector Problem on Lattices

Filipe Cabeleira
Artur Mariano
Gabriel Falcão
Instituto de Telecomunicações, Dept. of Electrical & Comp. Eng.
University of Coimbra
Coimbra, Portugal
filipe.cabeleira@co.it.pt, artur.mariano@co.it.pt, gff@co.it.pt

***Abstract*—In this paper, we also propose a parallel implementation of a Voronoi cell-based algorithm for the Shortest Vector Problem for both CPU and GPU architectures. Additionally, we present an algorithmic simplification with particular emphasis on significantly reducing the memory usage of the implementation. According to our tests, the parallel multi-core CPU implementation scales linearly with the number of cores used, and also benefits from simultaneous multi-threading, achieving a maximum speedup of $5.56\times$ for 8 threads. The parallel GPU implementation obtains speedups of $13.08\times$, compared with the sequential CPU implementation. The acceleration of this class of signal processing algorithms is a fundamental step in the evolution of post-quantum cryptanalysis. Currently, the best algorithms can take months to process for moderately low dimensions.**

***Index Terms*—Cryptography, Voronoi, Accelerators**

I. Introduction

MODERN cryptographic systems (such as RSA [1], ElGamal [2] and others) are based on hard mathematical problems, such as the factorization of large numbers and the computation of discrete logarithms. However, these systems were shown to be vulnerable, as they are feasible in the presence of quantum computers [3], [4], [5].

Given this vulnerability, new types of cryptosystems have been proposed since then, withstanding attacks even in the presence of quantum adversaries. Lattice-based cryptosystems are a very prominent type of cryptosystem for the so-called post-quantum era. They support advanced primitives, such as Fully Homomorphic Encryption, which allows for operations to be implemented on encrypted data, without having to decrypt it [6], [7]. They are also relatively efficient and easy to implement, and are believed to be secure in the presence of adversaries with quantum computers [5].

Lattice-based cryptosystems rely on problems such as the Shortest Vector Problem (SVP), the Closest Vector Problem (CVP) and their variants, as they cannot be solved exponentially faster on a quantum computer than on a traditional one.

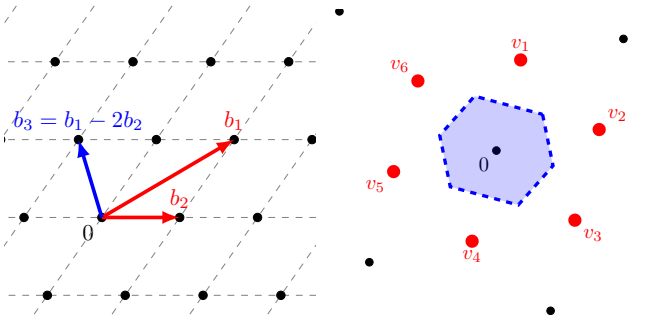
A lattice $\mathcal{L}$ in $\mathbb{R}^n$ is the subgroup formed by all integer linear combinations of a basis $\mathbf{B}$, a set of linearly independent vectors $\mathbf{b}_1$, ...,$\mathbf{b}_m$. It can be expressed by (1):

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

where $m \leq n$ is the *rank* of the lattice and if $m = n$, the lattice is of *full rank*.

Even though non-integer lattices are possible, integer lattices are normally used, which does not affect the hardness of problems and integers are easier to handle computationally.

Figure 1a shows an example of a lattice in $\mathbb{R}^2$, with its basis vectors ($\mathbf{b}_1$, $\mathbf{b}_2$) shown in red. Vector $\mathbf{b}_3$ is a linear combination of the basis vectors, and its Euclidean norm is smaller than $\mathbf{b}_1$. In this context, we will refer to this as a shorter vector, i.e. $\mathbf{b}_3$ is shorter than $\mathbf{b}_1$. The process of making lattice basis vectors shorter is called lattice basis reduction and is used in a multitude of lattice-based algorithms.



(a) A lattice in $\mathbb{R}^2$ and its basis ($\mathbf{b}_1$,$\mathbf{b}_2$) in red.

(b) A Voronoi cell in $\mathbb{R}^2$ (blue) and its relevant vectors (red).

Figure 1: Example of a lattice and a Voronoi cell.

The SVP consists in computing the shortest vector of a lattice in terms of its Euclidean norm. Similarly, the CVP consists in finding the lattice vector closest to a given arbitrary

vector (we will call these target vectors, and the closest lattice point to it, its solution vector).

II. Prior Work

Voronoi cell-based algorithms received considerable less attention than other classes of SVP-solvers. To the best of our knowledge, there are no known Voronoi cell-based parallel GPU implementations. There is, however, substantial work on the parallelization of other types of lattice-related solvers, such as enumeration and sieving.

In regards to enumeration, Correia et al. proposed a parallel CPU implementation of the Schnorr-Euchner SE++ algorithm, achieving speedups up to $14\times$ for 16 threads [8]. Hermans et al. presented a parallel GPU (CUDA) implementation of the same enumeration algorithm, obtaining a speedup of $5\times$ on a GTX 280, compared to a sequential CPU implementation [9]. Kuo et al. proposed a CPU + GPU version of Hermans' et al. work, and were able to find the solution to the SVP for a lattice in dimension 114 in less than two days, using 8 NVIDIA GPUs [10].

Milde and Schneider's parallelized sieving algorithms, which scale (almost) linearly for small thread counts (up to 5) [11]. Ishiguro's et al. work is based on Milde and Schneider's, improving the efficiency of the algorithm for high thread counts [12]. Mariano et al. also proposed a parallel implementation of GaussSieve, achieving linear speedups up to 64 threads and generally improving upon the performance of previously published works [13]. Recently, Bos et al. [14] presented a parallel GaussSieve implementation, obtaining a speedup factor of 2 compared to Mariano's et al. work [13].

Mariano et al. also parallelized versions of the HashSieve and LDSieve algorithms, in [15] and in [16], respectively. The former reports speedups of up to $12\times$ with 16 threads, and the latter linear speedups for the same number of threads.

Our contribution is twofold: we decrease the memory usage of the CPU implementation, by reducing the list of relevant vectors from $2 \times (2^n - 1)$ to 1 (the shortest relevant vector); and we also propose parallel implementations of a Voronoi cell-based algorithm, for both CPU and GPU architectures, achieving linear speedups for the former, and further speedups (compared to the CPU version) for the latter.

III. Voronoi Cell Algorithm for the SVP

The Voronoi cell $\mathcal{V}$ of a lattice is, by definition, the set of all points closer to zero than any other lattice point (see (2)).

$$\mathcal{V}(\mathcal{L}) = \{\mathbf{x} \in \mathbb{R}^n : ||\mathbf{x}|| \leq ||\mathbf{x} - \mathbf{v}|| \quad \forall\ \mathbf{v} \in \mathcal{L}\}. \tag{2}$$

The vectors of the minimum set required to fully describe the Voronoi cell of a lattice are called Voronoi *relevant* vectors and this set has $2 \times (2^n - 1)$ vectors, at most. Figure 1b shows an example Voronoi cell of a lattice in $\mathbb{R}^2$, and its relevant vectors. The solution to the SVP is given by the shortest relevant vector.

The "Relevant Vectors" algorithm, by Agrell et al., is used to compute the relevant vectors of an arbitrary lattice, and runs on exponential time. The algorithm can be split into four steps, which we now describe briefly. For more detail, we refer the reader to [17]. First, the target vectors that will be later used by a CVP solver are generated. Second, the coordinate system of the input data is modified (we refer the reader to [17] for more details on the reasoning behind this step). Third, the lattice basis and the target vectors are fed into an enumeration-based CVP solver. This enumeration algorithm is based on Schnorr and Euchner's algorithm, and gets its name from the fact that it enumerates all lattice points inside a certain radius [18]. In this paper we adopt the nomenclature of [17], and refer to this step as the *decode* procedure. Finally, the output of the decode step is converted back to the original coordinate system and, if the result is valid, i.e. if it is indeed a relevant vector, stored in a list.

Function AllClosestPoints

Input: Matrix $\mathbf{M}$, matrix $\mathbf{H}$, matrix $\mathbf{Q}$, vector $\mathbf{s}$
Output: List of vectors $\mathbf{X}$

Compute $\mathbf{x} = \mathbf{s}\mathbf{Q}^T$;
$\mathbf{U}$ = Decode($\mathbf{H}$, $\mathbf{x}$);
Compute γ as the lowest value $||\mathbf{u}\mathbf{M} - \mathbf{s}||$ for all $\mathbf{u} \in \mathbf{U}$;
Compute $\mathbf{X}$ as all $\{\mathbf{u}\mathbf{M} : \mathbf{u} \in \mathbf{U}, ||\mathbf{u}\mathbf{M} - \mathbf{s}|| = \gamma\}$
return $\mathbf{X}$

Practically, it is desirable to start by reducing the basis, in order to increase performance and the numerical stability of the algorithm. This is accomplished by means of a Lenstra-Lenstra-Lovász (LLL) [19] or Block Korkine-Zolotareff (BKZ) [18], [20] reduction. The $\mathbf{s}_i$, $i = 1, ..., (2^n - 1)$ target vectors of the basis $\mathbf{M}$ are then generated and stored in list $\mathcal{TV}$, according to (3).

$$\mathcal{TV}(\mathbf{M}) = \left\{\mathbf{s} = \mathbf{z}\mathbf{M} : \mathbf{z} \in \{0, 1/2\}^n - \{\mathbf{0}\}\right\} \tag{3}$$

After the target vector generation and input data preprocessing stages, the decode procedure is then executed, resulting on the list of vectors $\mathbf{U}$, which is then processed according to (4), resulting in the list of vectors $\mathbf{X}$.

$$\begin{aligned} \gamma &= \min\left\{||\mathbf{u}\mathbf{M} - \mathbf{s}|| \text{ for all } \mathbf{u} \in \mathbf{U}\right\} \\ \mathbf{X} &= \left\{\mathbf{u}\mathbf{M} : \mathbf{u} \in \mathbf{U}, ||\mathbf{u}\mathbf{M} - \mathbf{s}|| = \gamma\right\} \end{aligned} \tag{4}$$

If the list $\mathbf{X}$ contains 2, and only 2 vectors, then the result of the decode procedure is valid, and the vectors are added to the list of Voronoi relevant vectors $\mathbf{N}$ (a valid result yields 2 vectors that are symmetric to each other and, therefore, have the same norm). In practice, because the vectors are symmetric and of equal norm, we store only one of them, thus requiring half the memory space for this list.

The pseudo-code of our baseline (CPU sequential) implementation of the algorithm is shown in Algorithm 1.

Next we describe the main strategies used to perform the parallelization of the algorithm.

IV. Voronoi Cell Parallelization

We parallelized Algorithm 1 with OpenMP compiler directives for the CPU, and OpenACC compiler directives for the

Algorithm 1: Relevant Vectors

Input: Basis matrix $\mathbf{B}$
Output: Relevant Vectors $\mathbf{N}$

1 $\mathbf{M}$ = Reduce($\mathbf{B}$); /* for example, using the LLL algorithm */
2 [$\mathbf{Q}$, $\mathbf{R}$] = QR decomposition of $\mathbf{M}$;
3 $\mathbf{G} = \mathbf{R}^T$;
4 $\mathbf{H} = \mathbf{G}^{-1}$;
5 $\mathbf{N} = \varnothing$;
6 **forall** *vectors* $\mathbf{s} \in \mathcal{TV}$ **do**
7 $\mathbf{X}$ = AllClosestPoints($\mathbf{M}$, $\mathbf{H}$, $\mathbf{Q}$, $\mathbf{s}$);
8 **if** $|\mathbf{X}| = 2$ **then**
9 $\mathbf{N} = \mathbf{N} \bigcup \{2\mathbf{x} - 2\mathbf{s} : \mathbf{x} \in \mathbf{X}\}$;
10 **return** $\mathbf{N}$

CPU and GPU. These directives were applied to the main loop of the algorithm (line 6 of Algorithm 1). Each iteration of the main loop, where target vector generation and decoding take place, is completely independent from one another, allowing threads to run concurrently without data races and, therefore, the need for synchronization. Due to the fact that the workload may be unbalanced, i.e. not every decode takes the same amount of time, we used OpenMP's dynamic scheduler.

The memory usage of the Voronoi algorithm is very low, with the exception of the matrix that holds the relevant vectors. This structure grows exponentially with the dimension of the lattice basis. Given the fact that we are merely interested in the solution to the SVP, we implemented a mechanism to store only the shortest relevant vector found. This introduces the need for some synchronization between threads but, in our tests, it had a negligible effect on the performance of the algorithm, but with the benefit of decreasing the memory footprint. This was achieved with OpenMP's critical region and is shown in Algorithm 2 (note that the data management clauses – shared, private, etc – are not shown for legibility purposes). This memory optimization could not be applied to the GPU implementation due to the fact that, so far, OpenACC does not provide a critical construct. The memory savings achieved with this approach can be quantified by (5).

$$\text{Memory Savings} = \frac{8 \times 2^n \times n}{4 \times n} = 2 \times 2^n \tag{5}$$

OpenACC is similar to CUDA in regards to GPU thread management. The control is provided to the user by means of the number of gangs, workers and vector length, which are equivalent to CUDA's number of blocks, warp size and threads per block, respectively. In our implementation, addressed in Algorithm 3, we used a vector length L (threads per block) of 128 and the number of gangs T (number of blocks) depends on the size of the problem, given by (6).

$$T = \left\lceil \frac{\text{Number of Target Vectors}}{L} \right\rceil \tag{6}$$

This launch configuration results in a kernel where each thread decodes a single target vector (see Figure 2). However, this

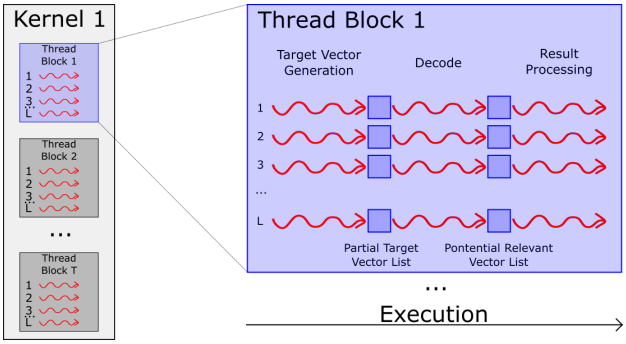


Figure 2: Execution flow of the OpenACC kernel for blocks processing L threads in parallel.

Algorithm 2: OpenMP Parallel Relevant Vectors

Input: Basis matrix $\mathbf{B}$
Output: Relevant Vectors $\mathbf{N}$

1 $\mathbf{M}$ = Reduce($\mathbf{B}$); /* for example, using the LLL algorithm */
2 [$\mathbf{Q}$, $\mathbf{R}$] = QR decomposition of $\mathbf{M}$;
3 $\mathbf{G} = \mathbf{R}^T$;
4 $\mathbf{H} = \mathbf{G}^{-1}$;
5 $\mathbf{N} = \varnothing$;
6 min_norm = ∞;
7 #pragma omp parallel for
8 **forall** *vectors* $\mathbf{s} \in \mathcal{M}$ **do**
9 $\mathbf{X}$ = AllClosestPoints($\mathbf{M}$, $\mathbf{H}$, $\mathbf{Q}$, $\mathbf{s}$);
10 #pragma omp critical
11 **if** $||2\mathbf{x} - 2\mathbf{s}|| <$ min_norm **then**
12 min_norm = $||2\mathbf{x} - 2\mathbf{s}||$;
13 $\mathbf{N} = \{2\mathbf{x} - 2\mathbf{s} : \mathbf{x} \in \mathbf{X}\}$;
14 **return** $\mathbf{N}$

also means that some structures must be private to each thread, thus increasing the memory usage of the OpenACC implementation. Note that, although the list that holding the relevant vectors is not as small as in the OpenMP implementation, it did not mean much, memory-wise, for the dimensions we tested.

Besides the parallelization of the algorithm, and in regards to memory management, we use a single large array instead of an array of arrays for storing each matrix. This requires the use of special indexing notation, but decreases allocation and deallocation time, and improves memory locality.

V. Experimental Results

The tests shown in this section were carried out in the machines detailed in Table I. Machine A is running Ubuntu 16.04 x86_64, with kernel 4.13. Machine B is running Ubuntu 17.10 x86_64, with kernel 4.13. The clock frequency in parenthesis represents the maximum attainable frequency using Turbo Boost technology. SMT stands for simultaneous multi-threading and HT stands for hyper-threading. L1 cache is split between instruction (i) and data (d) cache. All programs were compiled with the `-O3` and `-march=native` optimization

Algorithm 3: OpenACC Parallel Relevant Vectors

Input: Basis matrix $\mathbf{B}$
Output: Relevant Vectors $\mathbf{N}$

```
1  M = Reduce(B);   /* for example, using the LLL
     algorithm */
2  [Q, R] = QR decomposition of M;
3  G = R^T;
4  H = G^-1;
5  N = ∅;
6  min_norm = ∞;
7  #pragma acc parallel loop independent
8  forall vectors s ∈ M do
9      X = AllClosestPoints(M, H, Q, s);
10     if |X| = 2 then
11         N = N ⋃ {2x − 2s : x ∈ X};
12 return N
```

Table I: Apparatus: Machine A was used for the GPU and Machine B was used for the CPU runs.

Machine	A (GPU)	B (CPU)
CPU	Intel Core i3 6100	Intel Core i7 740QM
Clock frequency	3.70 GHz	1.73 GHz (2.93 GHz)
Cores	2	4
SMT	Yes (w/HT, 4 threads)	Yes (w/HT, 8 threads)
L1 Cache	32 kB i + 32 kB d	32 kB i + 32 kB d
L2 Cache	256 kB	256 kB
L3 Cache	3 MB	6 MB
RAM	8 GB	8 GB
GPU	NVIDIA GeForce 1060 GTX	—
GPU Clock rate	1759 MHz	—
GPU RAM	6 GB	—
OpenMP Compiler	—	g++ 7.2.0
OpenACC Compiler	PGI Compiler Suite 18.4	PGI Compiler Suite 18.4

flag. The OpenACC GPU program was compiled with the `-acc` flag, in order to enable OpenACC directives, and the `-ta=tesla:cc60` flag, to generate code for a GPU with compute capability 6.0. The compilation of the CPU implementation using OpenACC is similar, with exception of the `-ta=multicore` flag, to generate code for a multi-core CPU instead.

The bases used in our tests were generated using the SVP-Challenge's lattice basis generator (https://www.latticechallenge.org/svp-challenge/), compiled with NTL version 9.3 (https://www.shoup.net/ntl/). Unless otherwise stated, we carried out 10 runs per dimension (seeds 0 through 9), and the results shown correspond to the arithmetic average of those 10 runs. The bases were all reduced with NTL's LLL algorithm.

As previously mentioned, the OpenACC implementation uses more memory than OpenMP's, and it would be infeasible to run a single thread per target vector for dimensions 20 or higher. Instead, each thread decodes two target vectors. Moreover, the memory usage of the original implementation, for dimension $n = 20$, is 168 MBytes for the structure that holds the target vectors, while the optimized version drops this usage to 80 Bytes.

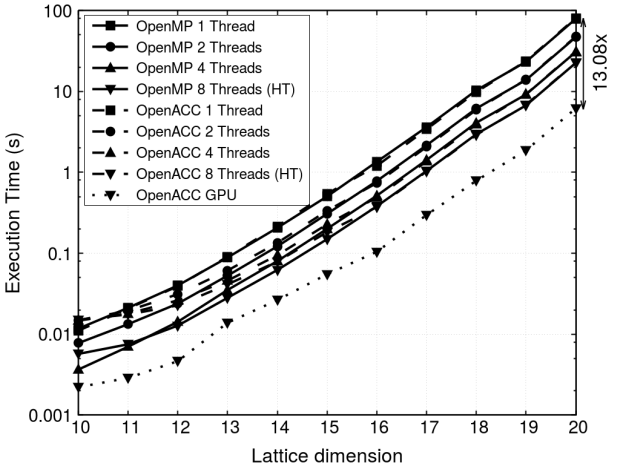


Figure 3: OpenMP and OpenACC CPU implementation using 1, 2, 4 and 8 threads on Machine B (Turbo Boost On), and OpenACC GPU implementation on Machine A, lattice dimensions 10 to 20.

Regarding scalability, and due to the fact that the effect of Turbo Boost varies with the number of threads used, we also tested our implementations with Turbo Boost turned off, achieving linear speedups for higher dimensions (i.e. higher workloads, where the overhead of thread creation is several orders of magnitude lower than actual computation time), for both OpenMP and OpenACC CPU implementations. Figure 3 shows the execution times of the OpenMP and OpenACC implementations, with Turbo Boost enabled for all CPU runs.

The GPU implementation is able to outperform both the OpenMP and OpenACC CPU ones for all dimensions shown, with a maximum speedup of $13.08\times$ for dimension 20, when compared against the sequential CPU run. The execution times of the OpenMP and OpenACC CPU implementations are virtually identical for higher dimensions.

VI. Conclusions

The proposed parallel Voronoi cell-based approach with an algorithmic simplification that reduces the memory usage scales linearly with the number of CPU and GPU cores used.

The reported speedups and the quick learning curve of the proposed OpenACC-based solution show that high-level parallelization targeted for GPUs is easily within reach of the signal processing community to engage in the world of HPC for dealing with relevant compute-intensive problems.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[2] T. El Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Proceedings of CRYPTO 84 on Advances in Cryptology*. New York, NY, USA: Springer-Verlag New York, Inc., 1985, pp. 10–18.

[3] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, ser. SFCS '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 124–134.

[4] ——, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997.

[5] D. J. Bernstein, J. Buchmann, and E. Dahmen, *Post Quantum Cryptography*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[6] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," in *Foundations of Secure Computation*, R. J. Lipton, D. P. Dobkin, and A. K. Jones, Eds. Orlando, FL, USA: Academic Press, Inc., 1978, pp. 160–179.

[7] A. Pedrouzo-Ulloa, J. R. Troncoso-Pastoriza, and F. Prez-Gonzlez, "Multivariate lattices for encrypted image processing," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, April 2015, pp. 1707–1711.

[8] F. Correia, A. Mariano, A. Proença, C. Bischof, and E. Agrell, "Parallel improved Schnorr-Euchner enumeration SE++ for the CVP and SVP," in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Feb 2016, pp. 596–603.

[9] J. Hermans, M. Schneider, J. Buchmann, F. Vercauteren, and B. Preneel, "Parallel shortest lattice vector enumeration on graphics cards," in *Proceedings of the Third International Conference on Cryptology in Africa*, ser. AFRICACRYPT'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 52–68.

[10] P.-C. Kuo, M. Schneider, O. Dagdelen, J. Reichelt, J. Buchmann, C.-M. Cheng, and B.-Y. Yang, "Extreme Enumeration on GPU and in Clouds: How Many Dollars You Need to Break SVP Challenges," in *Proc. of the 13th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Berlin: Springer-Verlag, 2011, pp. 176–191.

[11] B. Milde and M. Schneider, "A parallel implementation of gausssieve for the shortest vector problem in lattices," in *Proc. of the 11th International Conference on Parallel Computing Technologies (PaCT)*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 452–458.

[12] T. Ishiguro, S. Kiyomoto, Y. Miyake, and T. Takagi, "Parallel gauss sieve algorithm: Solving the SVP challenge over a 128-dimensional ideal lattice," in *Proc. of the 17th International Conference on Public-Key Cryptography (PKC), Vol. 8383*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 411–428.

[13] A. Mariano, S. Timnat, and C. Bischof, "Lock-free GaussSieve for linear speedups in parallel high performance SVP calculation," in *IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, Oct 2014, pp. 278–285.

[14] J. W. Bos, M. Naehrig, and J. V. D. Pol, "Sieving for shortest vectors in ideal lattices: a practical perspective," *International Journal of Applied Cryptography*, vol. 3, no. 4, pp. 313–329, 2017.

[15] A. Mariano, C. Bischof, and T. Laarhoven, "Parallel (probable) lock-free hash sieve: A practical sieving algorithm for the SVP," in *Proc. of the IEEE 44th International Conference on Parallel Processing (ICPP)*, Washington, DC, USA, 2015, pp. 590–599.

[16] A. Mariano, T. Laarhoven, and C. Bischof, "A parallel variant of LDSieve for the SVP on lattices," in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, March 2017, pp. 23–30.

[17] E. Agrell, T. Eriksson, A. Vardy, and K. Zeger, "Closest point search in lattices," *IEEE Transactions on Information Theory*, vol. 48, no. 8, pp. 2201–2214, Aug 2002.

[18] C. P. Schnorr and M. Euchner, "Lattice basis reduction: Improved practical algorithms and solving subset sum problems," *Math. Program.*, vol. 66, no. 2, pp. 181–199, Sep. 1994.

[19] A. K. Lenstra, H. W. Lenstra, and L. Lovász, "Factoring polynomials with rational coefficients," *Mathematische Annalen*, vol. 261, no. 4, pp. 515–534, Dec 1982.

[20] C. P. Schnorr, "A hierarchy of polynomial time lattice basis reduction algorithms," *Theoretical Computer Science*, vol. 53, no. 2, pp. 201 – 224, 1987.

B

Annex B

Energy-efficient lattice-based cryptanalysis attacks on low-power embedded GPUs

Filipe Cabeleira, Artur Mariano, Gabriel Falcao, *Senior Member, IEEE*

***Abstract*—Post-quantum cryptography, that is the cryptosystems that will withstand attacks with quantum computers, require a complete re-design of the current systems we know. At the same time, the energetic cost of developing new systems can be leveraged by the support of low-power embedded GPU platforms. Lattice-based algorithms have gained special interest for several reasons, including their simplicity and capability to perform operations on encrypted data (Fully Homomorphic Encryption). In this paper we exploit massive thread-level parallelism on embedded CPUs and GPUs from Nvidia, in one algorithm used in lattice-based cryptography. The developed OpenMP and CUDA-based kernels allow exploiting parallel regions of the code more conveniently on either the ARM CPU or the GPU, respectively. We report 3.7 to 5.3W of processing power for the Nvidia embedded processors, while achieving a relatively high throughput, with considerably higher energy efficiency, compared to the other processors used.**

***Index Terms*—Cryptography, Voronoi cell, Embedded processors, Parallel processing, Low-power, Energy saving**

I. Introduction

ALGORITHMS tailored for quantum architectures have shown vulnerabilities in modern cryptographic systems, such as RSA [1], ElGamal [2] and others. Specifically, in 1994 Shor proposed an algorithm designed for quantum computers that made the factorization of large numbers and computation of discrete logarithms feasible, thus rendering insecure the cryptosystems that base their security on these types of hard mathematical problem [3], [4].

Motivated by this discovery, the scientific community endeavored to design and study quantum-immune alternatives to current cryptographic systems. Lattice-based cryptosystems are especially interesting in this context, not only because they are believed to remain secure in the presence of quantum attacker, but also due to the fact that they are relatively simple to implement and efficient to use, and also support advanced cryptographic primitives, such as Fully Homomorphic Encryption, which allows some operations to be applied directly to the encrypted data [5], [6].

Lattices are all the integer linear combinations of a given lattice basis. A basis in $\mathbb{R}^n$ consists of the set of linearly independent vectors $\mathbf{b}_1$, ...,$\mathbf{b}_m$, formalized in (1).

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

where $m \leq n$ is the *rank* of the lattice. If $m = n$, the lattice is of *full rank*. Even though non-integer lattices are possible, integer lattices are normally used given that they are easier to handle computationally, while the hardness of the problem remains unchanged.

Lattice-based cryptosystems also base their security on hard mathematical problems, such as the Shortest Vector Problem (SVP), the Closest Vector Problem (CVP) and others. As the name suggests, the former consists in finding the shortest, non-zero lattice vector and the latter in finding the lattice point closest to a given vector, usually called a *target* vector.

In this paper, we show that some lattice-based cryptography algorithms are tailored to massively parallel architectures, while we provide hints that low-power processors are of extreme importance as they exhibit better energy efficiency for the same input data.

II. Related work

There are several types of algorithms to solve lattice-related problems, like enumeration, sieving and Voronoi cell-based. The first two have been extensively studied in recent years, and several variants of each have been proposed.

In regards to enumeration, various improvements of the early methods proposed by Pohst [7] and Kannan [8] have been proposed, such as the work of Schnorr and Euchner [9], improved upon by Agrell et al. [10]. Ghasemmehdi and Agrell further improved the latter [11]. Gama et al. published, in 2010, the enumeration with extreme pruning, greatly improving the execution time of the algorithm at the expense of a small degree of uncertainty of the result. In addition, parallel versions have also been published, such as the one presented by Dagdelen and Schneider, based on Schnorr and Euchner's work [12].

Sieving has also been subject to a similar level of attention, following the first published sieving algorithm by Ajtai, Kumar and Sivakumar, in 2001 [13], know as AKS, after its authors. Nguyen and Vidick proposed an heuristic version of AKS, further improving the algorithm [14]. Other variants of sieving were proposed, such as ListSieve and its heuristic variant, GaussSieve, by Micciancio and Voulgaris [15], Laarhoven's HashSieve [16] and Becker's et al. LDSieve [17]. Similarly to enumeration, parallel versions of the various sieving variants were published.

Voronoi cell-based algorithms, however, have not been studied as much, and the only relevant contributions are those of Agrell et al. [10] and, later in 2010, Micciancio and Voulgaris [18]. The former uses the enumeration routine presented in the same paper to compute the Voronoi relevant vectors of a lattice.

Thanks

III. Overview of the Voronoi cell-based algorithm

The work carried out in this paper is based upon the "Relevant Vectors" algorithm, presented by Agrell et al. in [10] that, as mentioned previously, is used to compute the Voronoi relevant vectors of a lattice. The set of all points closer to zero than any other lattice point forms the Voronoi cell of a lattice (see (2)).

$$\mathcal{V}(\mathcal{L}) = \{\mathbf{x} \in \mathbb{R}^n : ||\mathbf{x}|| \leq ||\mathbf{x} - \mathbf{v}|| \quad \forall\ \mathbf{v} \in \mathcal{L}\}. \tag{2}$$

The Voronoi relevant vectors consist of the minimum set of vectors necessary to completely describe the Voronoi cell of a lattice. This set has, at most, $2 \times (2^n - 1)$ vectors.

A brief description of the "Relevant Vectors" algorithm is now presented, given that its working principle is not the main focus of this work. For more details, we refer the reader to [10].

This algorithm can be split into four main steps, starting with the generation of the target vectors, that will later be passed on to the enumeration-based CVP solver. Next, the coordinate system of the input data is modified into the format required by the enumeration-based solver, followed by the execution of said solver on the modified data – the "decode" stage, following the nomenclature of [10]. Finally, the results are converted back into the original coordinate system and stored in a list if its valid.

In practice, we start by applying a reduction algorithm to the lattice basis, in order to improve the performance and numerical stability of the algorithm. This can be done with a Lenstra-Lenstra-Lovász (LLL) [19] or a Block Korkine-Zolotareff (BKZ) [20] reduction, for example. The list $\mathcal{TV}$ holds the $\mathbf{s}_i$, $i = 1, ..., (2^n - 1)$ target vectors of the lattice basis $\mathbf{M}$, given by (3).

$$\mathcal{TV}(\mathbf{M}) = \left\{\mathbf{s} = \mathbf{zM} : \mathbf{z} \in \{0, 1/2\}^n - \{\mathbf{0}\}\right\} \tag{3}$$

Afterwards, the decode procedure is executed, generating the list of vectors $\mathbf{U}$, that are then processed according to (4), yielding the list of vectors $\mathbf{X}$.

$$\begin{aligned} \gamma &= \min\left\{||\mathbf{uM} - \mathbf{s}|| \text{ for all } \mathbf{u} \in \mathbf{U}\right\} \\ \mathbf{X} &= \left\{\mathbf{uM} : \mathbf{u} \in \mathbf{U}, ||\mathbf{uM} - \mathbf{s}|| = \gamma\right\} \end{aligned} \tag{4}$$

If this result is valid, i.e. list $\mathbf{X}$ holds 2, and only 2 vectors, the result is stored in the relevant vector list $\mathbf{N}$. Given that a valid result consists in 2 vectors that are symmetric to each other – and, therefore, have the same norm – we only store one of them, thus decreasing the memory required for this list by half.

The pseudo-code of our baseline (CPU sequential) implementation of the algorithm is shown in Algorithm 1.

Function AllClosestPoints

Input: Matrix $\mathbf{M}$, matrix $\mathbf{H}$, matrix $\mathbf{Q}$, vector $\mathbf{s}$
Output: List of vectors $\mathbf{X}$

Compute $\mathbf{x} = \mathbf{sQ}^T$;
$\mathbf{U}$ = Decode($\mathbf{H}$, $\mathbf{x}$);
Compute γ as the lowest value $||\mathbf{uM} - \mathbf{s}||$ for all $\mathbf{u} \in \mathbf{U}$;
Compute $\mathbf{X}$ as all $\{\mathbf{uM} : \mathbf{u} \in \mathbf{U}, ||\mathbf{uM} - \mathbf{s}|| = \gamma\}$
return $\mathbf{X}$

Algorithm 1: Relevant Vectors

Input: Basis matrix $\mathbf{B}$
Output: Relevant Vectors $\mathbf{N}$

1 $\mathbf{M}$ = Reduce($\mathbf{B}$); /* for example, using the LLL algorithm */
2 [$\mathbf{Q}$, $\mathbf{R}$] = QR decomposition of $\mathbf{M}$;
3 $\mathbf{G} = \mathbf{R}^T$;
4 $\mathbf{H} = \mathbf{G}^{-1}$;
5 $\mathbf{N} = \varnothing$;
6 **forall** *vectors* $\mathbf{s} \in \mathcal{TV}$ **do**
7 $\mathbf{X}$ = AllClosestPoints($\mathbf{M}$, $\mathbf{H}$, $\mathbf{Q}$, $\mathbf{s}$);
8 **if** $|\mathbf{X}| = 2$ **then**
9 $\mathbf{N} = \mathbf{N} \bigcup \{2\mathbf{x} - 2\mathbf{s} : \mathbf{x} \in \mathbf{X}\}$;
10 **return** $\mathbf{N}$

IV. Parallel Decoding Approach

Given the independence of the iterations that make up the main loop of the algorithm (line 6 of Algorithm 1), where the generation and decoding of target vectors takes place, the various threads can execute the iterations of the loop in a completely concurrent fashion, without the need for synchronization.

Implementation-wise, OpenMP compiler directives are used to generate parallel code suitable for the CPU and CUDA is used for the GPU. Because each decode may take different amounts of time to complete, OpenMP's dynamic scheduler is used so that each thread gets assigned a new iteration as soon as they finish their current one. Regarding the GPU, since the decode procedure itself is sequential in nature (despite the various parallel enumeration algorithms available, the one described in [10] is sequential), each GPU thread is assigned a target vector to decode.

In order to better utilize the GPU's resources, several CUDA-specific optimizations were implemented, starting with asynchronous data transfers for the matrices whose values remain unchanged after their initial computation. Furthermore, and for the same reason, these matrices are stored in constant memory. For all other structures (including auxiliary ones, private to each thread), a single, large memory allocation is made prior to the kernel launch, in order to avoid several, expensive memory requests. Afterwards, pointer arithmetic is used to properly utilize this memory.

CUDA provides GPU thread management control to the user by means of the number of blocks, warp size and threads per block. Empirically, we selected $L = 128$ threads per block, as that was the value that performed best. As mentioned previously, each thread decodes a single target vector and, as such, the number of blocks T depends on the dimension n of the problem, and is given by (5).

$$T = \left\lceil \frac{\text{Number of Target Vectors}}{L} \right\rceil \tag{5}$$

Common to both the CPU and GPU implementations is the

use of a single large array for storing each matrix, as opposed to using an array of arrays. While this requires the use of special indexing notation, memory operations are optimized, by decreasing allocation and deallocation time, and improving memory locality, as each matrix is that way guaranteed to be continuous in RAM.

V. Experimental Results

Table I presents the apparatus regarding the different parallel computing platforms used under the context of this work.

TABLE I: Apparatus: Machine A was used for the GPU and Machine B was used for the CPU runs. Machine C (Jetson TX2) was used for both CPU and GPU runs.

Machine	A (GPU)	B (CPU)	C
Sockets	1	2	1
CPU	Intel Core i3 6100	Intel Xeon E5-2660v4	ARM Cortex-A57
Clock frequency	3.70 GHz	2.00 GHz	2.00 GHz
Cores per Socket	2	14	4
SMT	Yes (w/HT, 4 threads)	Yes (w/HT, 28 threads)	No
L1 Cache	32 kB i + 32 kB d	448 kB i + 448 kB d	48 kB i + 32 kB d
L2 Cache	256 kB	3.5 MB	2 MB
L3 Cache	3 MB	35 MB	—
RAM	8 GB	128 GB	8 GB[1]
GPU	NVIDIA GeForce 1060 GTX	—	Pascal GPU
GPU Clock rate	1759 MHz	—	1300 MHz
GPU RAM	6 GB	—	8 GB[1]
OpenMP Compiler	—	g++ 7.2.0	—
CUDA Compiler	CUDA Toolkit v9.1	—	JetPack v3.3

[1] Memory shared between the CPU and the GPU.

Although Machine B's CPU supports Turbo Boost technology, providing higher frequencies of operation (up to 3.2 GHz), in order for the results to be directly comparable with the other machines, all tests were conducted with this technology disabled. For the same reason, even though Machine B's CPU also provides hyper-threading (HT) – Intel's proprietary simultaneous multi-threading (SMT) implementation – only physical cores are used (Machine A is only used for GPU tests). L1 cache consists of instruction (i) and data (d) cache. All applications used in this paper were compiled with the `-O3` and `-march=native` optimization flags. The CUDA application was compiled with the `-arch=sm_61` flag, for Machine A's GPU and with the `-arch=sm_62` flag, for Machine C's GPU, in order to generate code for a GPU with compute capability 6.1 and 6.2, respectively.

The lattice basis used in this paper were obtained using the SVP-Challenge's generator, available at https://www.latticechallenge.org/svp-challenge/. The results shown in this work correspond to the arithmetic average of 10 bases per dimension. Basis reduction was accomplished by means of NTL's LLL algorithm. The power measurements are obtained, in the case of Machine A, by means of the `nvidia-smi` utility. Regarding Machine C, those measurements are recorded using the power monitor built-in to the Jetson TX2.

The goal of this work is to analyze not only the execution of the implementations on the different architectures, but also their energy consumption. Figure 1 compares the execution times of the parallel CPU implementation of the algorithm (Machines B and C).

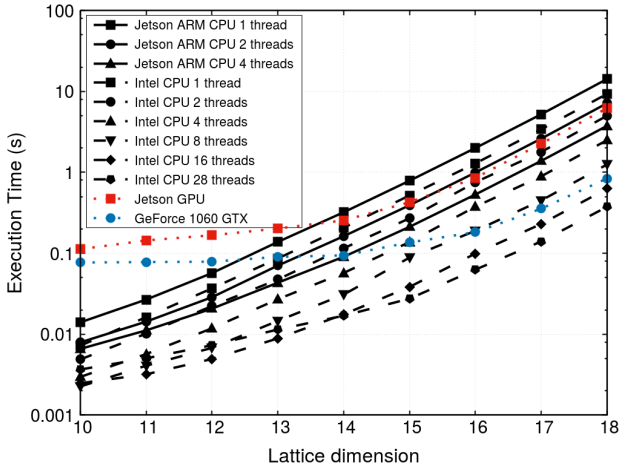


Fig. 1: Execution times of the Jetson ARM CPU (using 1, 2 and 4 threads), Intel CPU (using 1, 2, 4, 8, 16 and 28 threads), Jetson GPU and the GeForce GPU, lattice dimensions 10 to 18.

Given the independence of the iterations that make up the loop of the algorithm, both CPUs show a linear scalability of the algorithm, with the Intel CPU being roughly 53% faster than the ARM CPU, for an equivalent thread count. The Jetson GPU surpasses both the sequential ARM CPU and the sequential Intel CPU, reporting speedups of 2.3× and 1.5×, respectively. Similarly, the GeForce GPU shows a speedup of 17× and 11.1×, compared to the sequential Intel CPU and sequential ARM CPU, respectively. Naturally, the GeForce GPU outperforms the Jetson GPU with a speedup of 7.4×. However, while the CUDA implementation fares better than the sequential CPU implementation, regardless of the GPU used, the results obtained show that, for the algorithm at hand, a high-thread count CPU provides the best performance in regards the execution time.

Figure 2 compares the execution times of the parallel GPU implementation (Machines A and C), against the sequential CPU one (Machines B and C), and presents the measured power draw for both GPUs. Table II summarizes all the results obtained, and presents the power and energy statistics of the various processors used in this work.

The table shows that execution times of the high-power processors is lower than that of the equivalent low-power one. However, the gap between the power requirements of those processors is greater than the difference in execution times. As previously mentioned, the speedup of the GeForce GPU vs the Jetson GPU is 7.4× but the measured power required by the former is 13.2× greater than that of the latter, thus making the Jetson GPU to consume less energy to perform the same calculations. Similarly, the sequential CPU implementation on the Intel CPU is 1.53× faster than the ARM CPU, but also requires 5.68× more power, resulting in a much higher energy

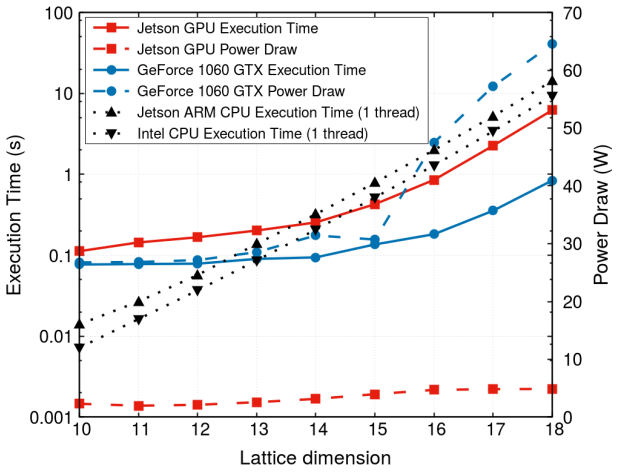


Fig. 2: Execution times and measured power draw of GPU implementations of the algorithm, compared against the sequential CPU one, lattice dimensions 10 to 18.

TABLE II: Comparison of the performance of Machine A's GPU, Machine B's CPU and Machine C's CPU and GPU. Results for dimension 18.

Average results	Execution Time (s)	Measured Power Draw (W)	Peak Power Draw (W)	Energy Used (J)
Jetson ARM CPU (1 thread)	14.247	**3.695**	**7.5**[*]	52.643
Jetson ARM CPU (4 threads)	3.759	5.329	**7.5**[*]	**20.032**
Jetson GPU	6.206	4.884	**7.5**[*]	30.310
Intel CPU (1 thread)	9.311	21[#]	105	195.531
Intel CPU (28 threads)	**0.375**	210[#]	210[†]	78.750
GeForce 1060 GTX	0.840	64.505	120	54.184

[*] Peak power of the entire Jetson TX2 board.
[#] Power measurements not available for the processor. Assuming 20% of peak power draw (taken from [21]) for 1 thread and 100% of peak power draw for 28 threads.
[†] Recall that Machine B is a two socket computer, 28 threads correspond to 14 cores running on each CPU.

consumption. As with previous cases, fully utilizing the CPUs causes the Intel CPU to be 10× faster than the ARM CPU but with a significantly higher power draw.

Of all the processors used, the ARM CPU running 4 threads is the most efficient, consuming the least amount of energy, while still being the third best in regards to execution time.

VI. Conclusions

The proposed parallel CPU implementation of the Voronoi cell-based algorithm scales linearly with the number of threads used. The GPU implementation provides a significant improvement to the sequential CPU implementation, but a high-thread count CPU still manages to outperform the GPU.

The low-power board used in this work reported slower execution times than the high-power processors but, given its significantly lower power draw, proved to be the most efficient of the tested machines in regards to energy consumption. As such, this result shows that low-power embedded systems are capable of efficiently providing the necessary performance even to the most compute-intensive problems.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.
[2] T. El Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Proceedings of CRYPTO 84 on Advances in Cryptology*. New York, NY, USA: Springer-Verlag New York, Inc., 1985, pp. 10–18.
[3] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, ser. SFCS '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 124–134.
[4] ——, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997.
[5] D. J. Bernstein, J. Buchmann, and E. Dahmen, *Post Quantum Cryptography*, 1st ed. Springer Publishing Company, Incorporated, 2009.
[6] P. Martins, L. Sousa, and A. Mariano, "A survey on fully homomorphic encryption: An engineering perspective," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 83:1–83:33, Dec. 2017.
[7] M. Pohst, "On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications," *SIGSAM Bull.*, vol. 15, no. 1, pp. 37–44, Feb. 1981.
[8] R. Kannan, "Improved algorithms for integer programming and related lattice problems," in *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '83. New York, NY, USA: ACM, 1983, pp. 193–206.
[9] C. P. Schnorr and M. Euchner, "Lattice basis reduction: Improved practical algorithms and solving subset sum problems," *Math. Program.*, vol. 66, no. 2, pp. 181–199, Sep. 1994.
[10] E. Agrell, T. Eriksson *et al.*, "Closest point search in lattices," *IEEE Transactions on Information Theory*, vol. 48, no. 8, pp. 2201–2214, Aug 2002.
[11] A. Ghasemmehdi and E. Agrell, "Faster recursions in sphere decoding," *IEEE Transactions on Information Theory*, vol. 57, no. 6, pp. 3530–3536, June 2011.
[12] Ö. Dagdelen and M. Schneider, "Parallel enumeration of shortest lattice vectors," in *Euro-Par 2010 - Parallel Processing*, P. D'Ambra, M. Guarracino, and D. Talia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 211–222.
[13] M. Ajtai, R. Kumar, and D. Sivakumar, "A sieve algorithm for the shortest lattice vector problem," in *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, ser. STOC '01. New York, NY, USA: ACM, 2001, pp. 601–610.
[14] P. Nguyen and T. Vidick, "Sieve algorithms for the shortest vector problem are practical," *Journal of Mathematical Cryptology*, vol. 2, no. 2, p. 181207, Jul 2008.
[15] D. Micciancio and P. Voulgaris, "Faster exponential time algorithms for the shortest vector problem," in *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '10. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2010, pp. 1468–1480.
[16] T. Laarhoven, "Sieving for shortest vectors in lattices using angular locality-sensitive hashing," in *Advances in Cryptology – CRYPTO 2015*, R. Gennaro and M. Robshaw, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 3–22.
[17] A. Becker, L. Ducas *et al.*, "New directions in nearest neighbor searching with applications to lattice sieving," in *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '16. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2016, pp. 10–24.
[18] D. Micciancio and P. Voulgaris, "A deterministic single exponential time algorithm for most lattice problems based on Voronoi cell computations," in *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, ser. STOC '10. New York, NY, USA: ACM, 2010, pp. 351–358.

[19] A. K. Lenstra, H. W. Lenstra, and L. Lovász, "Factoring polynomials with rational coefficients," *Mathematische Annalen*, vol. 261, no. 4, pp. 515–534, Dec 1982.

[20] C. P. Schnorr, "A hierarchy of polynomial time lattice basis reduction algorithms," *Theoretical Computer Science*, vol. 53, no. 2, pp. 201 – 224, 1987.

[21] "Intel Xeon Processor E5 v4 Product Family Thermal Mechanical Specification and Design Guide," https://www.intel.com/content/dam/www/public/us/en/documents/guides/xeon-e5-v4-thermal-guide.pdf, accessed: 2019-02-12.