Memory-Optimized Voronoi Cell-based Parallel Kernels for the Shortest Vector Problem on Lattices

Filipe Cabeleira, Artur Mariano*, and Gabriel Falcao
Instituto de Telecomunicações, Dept. of Electrical & Comp. Eng., University of Coimbra, Portugal
**U. Minho, HASLab - INESC TEC, Portugal*
{filipe.cabeleira, gff}@co.it.pt, artur.miguel@gmail.com

***Abstract*—In this paper we 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 factorization of large integers is 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]. 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.

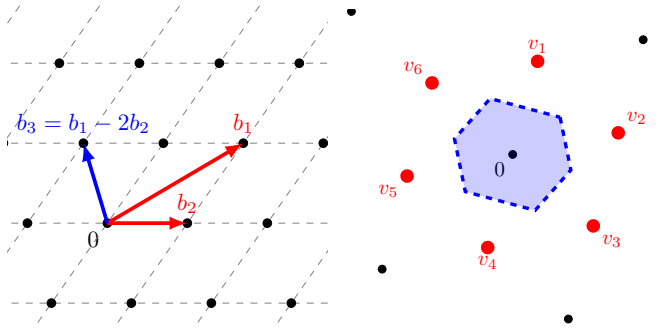
A lattice $\mathcal{L}$ in $\mathbb{R}^n$ is the subgroup formed by all integer linear combinations of a basis $\mathbf{B}$, a set of linearly independent vectors $\mathbf{b}_1$, ...,$\mathbf{b}_m$. It can be expressed by (1):

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

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 non-zero 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. Two of the most known works in this subject are those of Agrell et al. [7] and Micciancio and Voulgaris [8]. 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 [9]. Hermans et

978-9-0827-9703-9/19/$31.00 ©2019 IEEE

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 [10]. 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 [11].

Milde and Schneider's parallelized sieving algorithms, which scale (almost) linearly for small thread counts (up to 5) [12]. Ishiguro's et al. work is based on Milde and Schneider's, improving the efficiency of the algorithm for high thread counts [13]. 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 [14]. Recently, Bos et al. [15] presented a parallel GaussSieve implementation, obtaining a speedup factor of 2 compared to Mariano's et al. previous work [14]. The GuassSieve algorithm was also parallelized on GPUs, with some success [16].

Mariano et al. also parallelized versions of the HashSieve and LDSieve algorithms, in [17] and in [18], 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 (we also consider the border to be part of this set), as defined in (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. A lattice vector $\mathbf{r}$ is relevant if $\mathcal{V}$ and $\mathcal{V} + \mathbf{r}$ share a non-empty boundary. Figure 1b shows an example Voronoi cell of a lattice in $\mathbb{R}^2$, and its relevant vectors. A solution to the SVP is given by a 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 [7]. 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 [7] 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 [19]. In this paper we adopt the nomenclature of [7], 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 $\mathbf{X}$

1 Compute $\mathbf{x}$ = $\mathbf{s}\mathbf{Q}^T$;
2 $\mathbf{U}$ = Decode($\mathbf{H}$, $\mathbf{x}$);
3 Compute γ as the lowest value $||\mathbf{u}\mathbf{M} - \mathbf{s}||$ for all $\mathbf{u} \in \mathbf{U}$;
4 Compute $\mathbf{X}$ as all $\{\mathbf{u}\mathbf{M} : \mathbf{u} \in \mathbf{U}, ||\mathbf{u}\mathbf{M} - \mathbf{s}|| = \gamma\}$
5 **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) [20] or Block Korkine-Zolotareff (BKZ) [19], [21] 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 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.

Algorithm 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 $\mathbf{M}$;
$\mathbf{G} = \mathbf{R}^T$;
$\mathbf{H} = \mathbf{G}^{-1}$;
$\mathbf{N} = \varnothing$;
forall *vectors* $\mathbf{s} \in \mathcal{TV}$ **do**
  $\mathbf{X}$ = AllClosestPoints($\mathbf{M}$, $\mathbf{H}$, $\mathbf{Q}$, $\mathbf{s}$);
  if $|\mathbf{X}| = 2$ **then**
    $\mathbf{N} = \mathbf{N} \bigcup \{2\mathbf{x} - 2\mathbf{s} : \mathbf{x} \in \mathbf{X}\}$;
return $\mathbf{N}$

The memory usage of the Voronoi algorithm is significantly 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 depend on lattice dimension n and can be quantified by (5).

$$\begin{aligned} \text{Memory Savings} &= \frac{8 \times (2^n - 1) \times n}{4 \times n} \\ &= 2 \times (2^n - 1) \\ &= 2^{n+1} - 2 \end{aligned} \tag{5}$$

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 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.

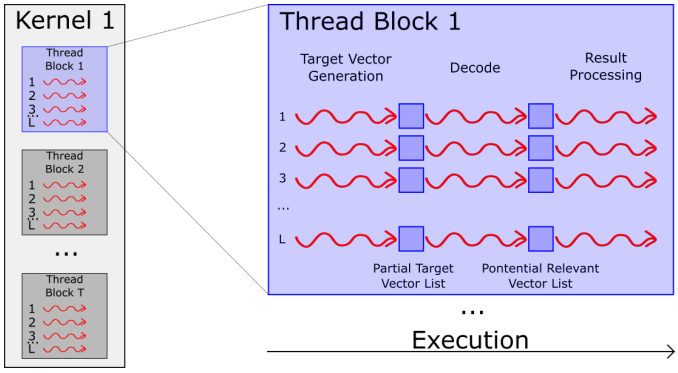


Figure 2: Execution flow of the OpenACC kernel for blocks processing L threads in parallel.

Algorithm 2: OpenMP Parallel Relevant Vectors

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

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

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 flag. The OpenACC GPU program was compiled with the `-acc` flag, in order to enable OpenACC directives, and

Algorithm 3: OpenACC Parallel Relevant Vectors

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

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

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. Note that, for the GPU implementation, enough memory must be allocated for the possibility that all target vectors yield relevant vectors, due to the fact that allocated memory remains unchanged during the main loop of the algorithm.

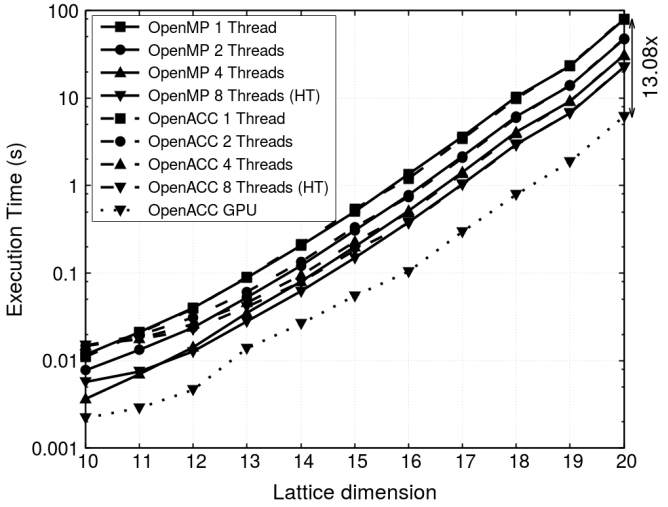


Figure 3: OpenMP and OpenACC CPU implementation using 1, 2, 4 and 8 threads on Machine B (Turbo Boost On), and OpenACC GPU implementation on Machine A, lattice dimensions 10 to 20.

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.

VII. Acknowledgment

This research work was supported by Instituto de Telecomunicações (IT) and Fundação para a Ciência e a Tecnologia (FCT) under projects UID/EEA/50008/2019 and

PTDC/EEI-HAC/30485/2017. Artur Mariano is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Projektnummer 382285730.

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] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford, CA, USA, 2009, aAI3382729.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] S.-Y. Yang, P.-C. Kuo, B.-Y. Yang, and C.-M. Cheng, "Gauss Sieve Algorithm on GPUs," in *Topics in Cryptology – CT-RSA 2017*, H. Handschuh, Ed. Cham: Springer International Publishing, 2017, pp. 39–57.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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