Hindawi
Scientific Programming
Volume 2021, Article ID 8852497, 13 pages
https://doi.org/10.1155/2021/8852497

*Research Article*

# Three Strategies for Improving Shortest Vector Enumeration Using GPUs

**Mohamed S. Esseissah, Ashraf Bhery, Sameh S. Daoud, and Hatem M. Bahig**

*Computer Science Division, Mathematics Department, Faculty of Science, Ain Shams University, Cairo, Egypt*

Correspondence should be addressed to Hatem M. Bahig; h.m.bahig@gmail.com

Received 8 August 2020; Revised 27 October 2020; Accepted 15 December 2020; Published 5 January 2021

Academic Editor: Jianping Gou



Hard Lattice problems are assumed to be one of the most promising problems for generating cryptosystems that are secure in quantum computing. The shortest vector problem (SVP) is one of the most famous lattice problems. In this paper, we present three improvements on GPU-based parallel algorithms for solving SVP using the classical enumeration and pruned enumeration. There are two improvements for preprocessing: we use a combination of randomization and the Gaussian heuristic to expect a better basis that leads rapidly to a shortest vector and we expect the level on which the exchanging data between CPU and GPU is optimized. In the third improvement, we improve GPU-based implementation by generating some points in GPU rather than in CPU. We used NVIDIA GeForce GPUs of type GTX 1060 6G. We achieved a significant improvement upon Hermans's improvement. The improvements speed up the pruned enumeration by a factor of almost 2.5 using a single GPU. Additionally, we provided an implementation for multi-GPUs by using two GPUs. The results showed that our algorithm of enumeration is scalable since the speedups achieved using two GPUs are almost faster than Hermans's improvement by a factor of almost 5. The improvements also provided a high speedup for the classical enumeration. The speedup achieved using our improvements and two GPUs on a challenge of dimension 60 is almost faster by factor 2 than Correia's parallel implementation using a dual-socket machine with 16 physical cores and simultaneous multithreading technology.

## 1. Introduction

A lattice $L$ is the set of all integer combination of $n$ linearly independent vectors $b_1, b_2, \ldots, b_n$ in $\mathbb{R}^m$. These vectors are known as a basis of the lattice. The most famous computational problems involving lattices are the shortest vector problem (SVP) and closet vector problem (CVP). Concretely, in SVP, given a basis $b_1, b_2, \ldots, b_n$, one is asked to output the shortest nonzero vector in the lattice, and in CVP, given a basis $b_1, b_2, \ldots, b_n$ and a target $t \in \mathbb{R}^m$, one is asked to output a lattice vector closest to $t$. SVP and CVP are NP-hard problems on the worst case [1, 2].

Due to the hardness of SVP and CVP, they are widely used in cryptography and other mathematical applications. More precisely, they are employed for creating public key cryptosystems that are known as lattice-based cryptosystems. Lattice-based cryptosystems have advantages over the earlier cryptosystem in terms of speed and hardness, where lattice-based cryptosystems can be implemented more faster than RSA and ECC cryptosystems at an equivalent security level [3, 4]. Moreover, lattice problems are known to be postquantum problems [5]. The most known lattice-based cryptosystems are GGH [6], NTRU [7], and recently LWE-based cryptosystems [8, 9]. Furthermore, the theory of lattices has more investigation in cryptography beyond creating new cryptosystems such as breaking some public key cryptosystems such as RSA and ECC.

The currently known algorithms for SVP and CVP can be classified into two categories:

(1) Approximation algorithms which return an approximation solution to either SVP or CVP: the first polynomial-time approximation algorithm for SVP was the celebrated Lenstra, Lenstra, and Lovász (LLL) basis reduction algorithm [10] which achieves an approximation factor $2^{\mathcal{O}(m)}$ exponential in the dimension $m$. Then, the idea of blockwise reduction appeared and several blockwise lattice reduction

algorithms [11–14] were proposed successively. Currently, the most used approximation algorithm for SVP is Blockwise Korkine–Zolotarev (BKZ) developed by Schnorr and Euchner [14]. LLL and BKZ are mainly used to reduce a basis in order to enhance the basis quality. In approximation algorithms, they are usually used along with a pruned enumeration [15]. Pruned enumeration is based on deleting subtrees that are unlikely to lead to a solution.

(2) Exact algorithms which return an exact solution for either SVP or CVP: they are based on the exhaustive search. Consequently, they are expensive in terms of running time, which is at least an exponential of the dimension of the lattice. Exact algorithms include classical enumeration [16], otherwise known as ENUM; also Voronoi cell [17] is known for SVP and CVP and Sieving [18] for SVP. There is no Sieving algorithm known to solve CVP instances.

Both categories are important and in fact are combined. First, lattice reduction approximation algorithms cannot output shortest vectors in high-dimensional lattices. They are used to find vectors that are sufficiently short to ensure an enumeration process to work efficiently. Second, the running time of enumeration algorithms depends on the quality of the input basis. So, preprocessing the input lattice basis using a lattice reduction approximation algorithm is an essential part of enumeration algorithms.

In this paper, we focus our work to the parallel lattice enumeration algorithms for solving SVP on Graphics Processing Unit (GPU). The classical algorithms for lattice enumeration were first proposed by Kannan [16] and by Fincke and Pohst [19]. Therefore, the enumeration is sometimes referred to as KFP algorithm. A distinguishing feature of these algorithms is that they require a polynomial-space complexity. Moreover, they have good asymptotic runtime, with the best theoretical algorithm (due to Kannan [16]), and further analyzed in [20], achieving $2^{\mathcal{O}(m\log m)}$ worst-case time complexity, where $m$ is the dimension of the lattice. Based on the enumeration algorithms presented by Kannan [16] and by Fincke and Pohst [19], Schnorr and Euchner [14] proposed another popular polynomial-space enumeration algorithm to search for the shortest vector of a lattice, which runs in exponential time, and the cost is no more than $2^{\mathcal{O}(m^2)}$ in the lattice dimension $m$. Although the complexity of the Schnorr–Euchner enumeration seems higher than that of Kannan's; it is in fact a widely used practical method. For example, it is a fundamental tool in the popular mathematical libraries NTL [21] and fpLLL [22].

Enumeration algorithms for solving SVP can be viewed as a depth-first search in a tree structure, going over all vectors in a specified search region deterministically. Typically, a basis reduction such as BKZ is performed first to improve the basis to one likely to yield a short vector via enumeration. The enumeration tree (conducted by classical enumeration algorithms) for solving SVP grows largely due to the large number of subtrees, which makes the researchers looking for an alternative approach to speed up the enumeration. Schnorr and Euchner [14] proposed a new strategy called the *pruned enumeration*. The main idea of the pruned enumeration is to prune subtrees of the search tree in which the probability of finding an exact solution is too small [23]. By using the pruned enumeration, exhaustive search is restricted to only a subset of the set of all possible solutions. Although, this may lead to some probability of missing the exact solution, but it gives a significant improvement in terms of the execution time. In [23], Gama et al. proposed the *extreme pruning* approach to solving SVP and approximate SVP and showed that it is possible to speed up the enumeration exponentially by randomizing the algorithm.

*1.1. Previous Works.* Dealing with SVP using parallel computing architecture is not a new trend. There are much works for that either for approximation algorithms such as LLL and BKZ or for exact algorithms such as Enumeration [16] and Sieving [18]. Backes and Wetzel [24] presented the first parallel implementation of LLL on multicores computer architecture. They have shown an improvement of about 3.2 times more than that in the traditional nonparallel algorithm. Afterward, the same authors introduced numbers of modifications [25] that increase the speedup of the algorithm in [24] by a factor 1.25. The enumeration for the shortest vector was considered in [26], achieving its improvement by making use of Field Programmable Gate Arrays (FPGAs). Dagdelen and Schnieder [27] have implemented a new parallel algorithm based on the modified version of KFP enumeration algorithm by a new algorithm invented by Schnorr and Euchner [14]. Dagdelen and Schnieder's parallel algorithm has achieved using multicores CPU a significant speedup of reaching up to factor 14 compared to the currently best sequential public implementation (fpLLL implementation [22]). Hermans et al. [28] presented a GPU implementation of enumeration using Nvidia GeForce GTX 280 GPU. It is notable that the authors with this cheap and simple specification of GPU compared to the current available GPUs could achieve an implementation five times faster than that in a single core of an Intel Core 2 Extreme QX9650 at 3 GHz. The enumeration algorithm with embedding the extreme pruning approach is exploited on cloud computing service providers in [29]. The cloud computing service provider has allowed the authors to test the shortest vectors of height dimensions, for example, 114 and 120 dimensions, providing them by many high-performance NVIDIA GPUs. Subsequently, Correia et al. [30] showed that the search tree of the enumeration-based CVP contains many symmetric branched. They proposed to avoid the redundant computation by exploring only one of the symmetric branches, which reduces the number of branches that are computed. In addition, they proposed another strategy that balances the workload among threads on multicores. These strategies are implemented on multicores improving the implementation of [27] by a factor of 35% to 60%.

*1.2. Our Contribution.* In this paper, we present an improved parallel version of the pruned enumeration algorithm (approximation algorithm) of [14] that finds a shortest, nonzero vector in a lattice. In addition, we show

that the proposed improvements are efficient for the classical enumeration (exact algorithm). We use the CUDA framework of NVIDIA for implementing the algorithm on GPU. The results showed that our strategies improve Hermans's implementation [28] based on pruned enumeration by a factor of almost 5 using two GPUs. For classical enumeration, we compare our implementation to Correia's implementation [30]. The experimental results by solving a challenge of dimension 60 showed that our strategies based on two GPUs is twice faster than Correia's implementation based on 16 physical cores and simultaneous multithreading technology.

The GPU is considered as one of the most promising technology in the parallel computing field due to its intensive computation power. It speeded up the performance of many problems such as [31, 32]. A single GPU contains thousands of cores. Besides, GPU is developed with speeds far exceeding the development rate of CPUs. GPU from the hardware view is classified as a heterogeneous model, and it works along with the CPU; the input data must pass first through the CPU and then copied to the GPU. This operation is considered as the main factor which reduces the performance of the parallel computing on GPU. The frequent transmitting of data between CPU and GPU might affect the performance of our parallel computing. To avoid that we have to copy only the data that cannot be generated easily on GPU. In other words, GPU has a powerful computation, so generating data computationally might cost less than copying them. However, in some cases, the operation of generating data causes slowness in the execution time. The trade-off between the cost of copying data and generating data must be treated carefully as one of the important factors for improving the parallel computing in GPU.

In order to speed up the computation of solving SVP using the enumeration, we can make use of GPU, dividing the search tree of SVP to thousands of subtrees that can run independently. The search tree is divided into two parts: the first part runs on CPU and the second part runs on GPU. Those two parts are separated by a level $\alpha$ on which the subtrees are generated and sent to GPU to run. Each subtree is assigned to a thread in GPU. Actually, there are millions of subtrees that cannot be sent at once to GPU, so we may need many trips to GPU in order to cover all subtrees generated on level $\alpha$.

The technical core of our contribution is proposing three strategies for speeding up the running time of the classical enumeration and pruned enumeration. The first strategy is based on randomizing the basis. We perform randomization to get set of basis, for example, 100 basis. And then, we measure the quality of each basis by estimating the nodes of the enumeration tree of each basis. As suggested by Hanrot and Stehlé [20], the nodes of an enumeration tree can be estimated using the Gaussian heuristic. The basis that leads to fewer nodes is selected as the best basis. The second strategy is based on expecting the best level $\alpha$ on which the cost of transferring the data between CPU and GPU is very low, while the third strategy is based on gaining the improvement based on GPU, which can be carried out by generating some subtrees in GPU rather than in CPU. The main idea of the third strategy is similar to another one used for improving BDD Enumeration [33].

*1.3. Our Results.* Our work shows that we can improve Hermans et al.'s implementation [28] for pruned enumeration and Correia et al.'s implementation [30] for classical enumeration by applying some improvements such as starting with a better basis and choosing a good level for exchanging the data between GPU and CPU. Additionally, we propose to generate some subtrees with specific properties on GPU rather than CPU.

*1.4. Roadmap.* The rest of the paper is organized as follows. In Section 2, we introduce a background on lattice basis reduction. In Section 3, we show the classical enumeration algorithm for solving SVP. In Section 4, we describe the pruned enumeration algorithm for solving SVP. Section 5 is dedicated to show our contribution for improving the current parallel algorithm for solving SVP using GPU. In Section 6, we show the experimental results. Finally, conclusions and future work are shown in Section 7.

## 2. Preliminaries

In this section, we first define the lattice followed by a brief review of the needed background on lattice and lattice basis reduction.

Let $\mathbb{R}^m$ be the Euclidean $m$-dimensional space and $B = \{b_1, b_2, \ldots, b_n\}$ be a set of linearly independent vectors in $\mathbb{R}^m$. The lattice generated by $B$ is the set

$$L(B) = \{a_1 b_1 + \cdots + a_n b_n, \quad a_i \in \mathbb{Z}\} \tag{1}$$

of all linear integral combinations of $B$. The set $B$ is called a basis for the lattice $L(B)$. The integers $m$ and $n$ are called the dimension and rank of the lattice, respectively. If $n = m$, then $L(B)$ is called a full rank lattice. A lattice $L$ is said to be integral if it is contained in $\mathbb{Z}^m$, namely, any basis for $L$ only consists of integer vectors. In this paper, we assume that the basis vectors $\{b_1, b_2, \ldots, b_n\}$ for the lattice $L$ have integer entries, which is the usual case in the lattice-based cryptography.

Usually, for simplicity, a basis $B = \{b_1, b_2, \ldots, b_n\}$ is represented by a $m \times n$ matrix, with columns consisting of the vectors of a basis of lattice $L$. We define the determinant or volume of lattice $L$, denoted by $\det(L)$, as $\det(L) = \sqrt{\det(B^T B)}$. Since we search for a shortest vector, we must be aware that the basis of lattice is not unique. For any two distinct basis matrices $B$ and $C$, we say that $L(B) = L(C)$ if and only if $B$ and $C$ are related by unimodular matrix, i.e., $C = \mathrm{UB}$, where $U$ is an $m \times m$ matrix with integer entries and determinant $\det(U) = \pm 1$.

Such transformation of a unimodular matrix enables us to generate a new basis that is better than the original basis. Consequently, the question comes up is when a basis is "better" or more convenient than another. We can say that a basis for a lattice that is "as close to orthogonal as it can be" is the convenient one [34]; as a result, we can say that a closer basis to orthogonal is better. So, the initial step for getting a

shorter vector as a basis for a lattice is to generate an orthogonal basis; then, based on the orthogonal basis as well as what is so-called lattice reduction, we can generate a better basis. The lattice basis reduction is the procedure of finding a basis with vectors as short and nearly orthogonal. A famous method for generating an orthogonal basis for a lattice $L$ is the Gram–Schmidt orthogonalization method.

The Gram–Schmidt orthogonalization method is applied by transforming any set of vectors to another set of vectors which are pairwise orthogonal, and they span the same space in addition.

For a given lattice basis $\{b_1, b_2, \ldots, b_n\}$, we define the corresponding Gram–Schmidt orthogonalization basis $\{b_1^*, b_2^*, \ldots, b_n^*\}$ as follows:

$$\begin{aligned} b_1^* &= b_1, \\ b_i^* &= b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^*, \\ \text{where } \mu_{i,j} &= \frac{\langle b_i, b_j^* \rangle}{\left\| b_j^* \right\|^2}, \end{aligned} \tag{2}$$

where $\langle , \rangle$ denotes the inner production and  denotes Euclidean norm.

The Gram–Schmidt orthogonal basis vectors $\{b_1^*, b_2^*, \ldots, b_n^*\}$ may seem to be the best reduced basis vectors, but that is not true because they are not necessarily in the lattice. If the basis $\{b_1, b_2, \ldots, b_n\}$ consists of integer vectors, then the Gram–Schmidt orthogonal basis vectors $\{b_1^*, b_2^*, \ldots, b_n^*\}$ and the coefficients $\mu_{i,j}$ are rational [14]. However, the Gram–Schmidt orthogonal basis is helpful in the lattice reduction process; the lattice reduction can be computed easily if one has basis which is orthogonal or sufficiently close to orthogonal basis [34]. In 1982, Lenstra et al. [10] proposed a notion called LLL reduction and a polynomial-time reduction algorithm that compute an LLL-reduced basis from arbitrary basis of the same lattice. LLL algorithm is based on Gram–Schmidt orthogonal, and it aims to generate a new basis of lattice which is as close as possible to Gram–Schmidt orthogonal basis. Nowadays, the algorithm used for lattice reduction in practice is the BKZ algorithm. In this paper, we use the BKZ implementation in the NTL library of Shoup [21] to perform BKZ reduction.

## 3. Classical Enumeration Algorithm

In order to solve SVP, enumeration algorithms are usually employed, since they are the most efficient algorithms for currently realistic dimensions. The standard enumeration algorithm usually attributed to [16] and by Fincke and Pohst [19]. In [14], Schnorr and Euchner presented a zig-zag strategy for enumerating the lattice vectors to make the algorithm have a better performance in practice. This algorithm (pseudocode can be found in Algorithm 1) can be described as follows: given as input a basis $B = \{b_1, b_2, \ldots, b_n\}$, the Gram–Schmidt coefficients $\mu_{i,j}$, $1 \le i, j \le n$, the quadratic norm of the Gram–Schmidt orthogonalization $\|b_1^*\|^2$, $\|b_2^*\|^2$, …, $\|b_n^*\|^2$ of $B$, and an initial bound $A \in \mathbb{R} > 0$, and its goal is to find a coefficient vector $u \in \mathbb{Z}^n$ satisfying the equation

$$\left\| \sum_{i=1}^{n} u_i \cdot b_i \right\| = \min_{x \in \mathbb{Z}^n} \left\| \sum_{i=1}^{n} x_i \cdot b_i \right\|. \tag{3}$$

The search space of the original enumeration algorithm is the set of all coefficients vectors $u \in \mathbb{Z}^n$ that satisfy $\| \sum_{i=1}^{n} u_i \cdot b_i \| \le A$. Those linear combination are organized in a tree structure. Leafs of the tree contain full linear combinations, whereas inner nodes contain partly filled vectors. The search for the tree leaf that determines a nonzero shortest vector is performed in a depth-first search order [16]. The most important part of the enumeration is cutting off parts of the tree, i.e., the strategy in which the subtrees are explored and subtrees that do not lead to a shorter vector are eliminated. This strategy is based on the computation of an intermediate squared norm $l_k$, where $k$ is a level of the tree. The following lemma shows how it is computed.

**Lemma 1** (see [34]). *Let $b_1, b_2, \ldots, b_n$ be a basis for the lattice $L$, $b_1^*, b_2^*, \ldots, b_n^*$ be Gram–Schmidt orthogonal basis and $\mu_{i,j}$ be the Gram–Schmidt coefficients for $1 \le i, j \le n$. Let $A \in \mathbb{R} > 0$ be an upper bound for the Euclidean norm of the lattice $L$. Let $v = \sum_{i=1}^{n} b_i v_i \in L$ such that $\|v\|^2 \le A$; then, for $1 \le k \le n$,*

$$\sum_{i=k}^{n} \left( v_i + \sum_{j=i+1}^{n} \mu_{j,i} v_j \right)^2 \left\| b_i^* \right\|^2 \le A. \tag{4}$$

Let $l_k = \sum_{i=k}^{n} (v_i + \sum_{j=i+1}^{n} \mu_{j,i} v_j)^2 \|b_i^*\|^2$ and $c_k = \sum_{j=k+1}^{n} \mu_{j,k} v_j$. Consequently, the inequality can be reduced to the following formula:

$$l_k = l_{k+1} + (v_k + c_k)^2 \left\| b_k^* \right\|^2 \le A. \tag{5}$$

## 4. Pruned Enumeration Algorithm

In order to decrease the execution time of Schnorr and Euchner's algorithm, Schnorr and Hörner suggested a modification to enumeration in 1995 [15], called pruning. The pruned enumeration is based on deleting subtrees that are unlikely to lead to a solution. The main idea of the pruned enumeration is to prune subtrees of the search tree in which the probability of finding an exact solution is too small [23]. By using the pruned enumeration, exhaustive search is restricted to only a subset of the set of all possible solutions. Although, this may lead to some probability of missing the exact solution, but it gives a significant improvement in terms of the execution time. In 2010, Gama et al. proposed the enumeration with extreme pruning [23], which prunes a much higher number of nodes of the enumeration tree, thus significantly reducing the execution time of the algorithm. Although the extreme pruning speeds up significantly the search tree, the probability of missing the solution is very high. The probability to find a solution is only 0.1%. However, according to Gama et al., the probability can be increased to reach more than 90% if we perform the enumeration almost 1000 times on different bases. In this paper, we focus on the pruned enumeration and propose an efficient

approach for speeding up the search using the pruned enumeration.

The pruned enumeration is based on using a bound $A_k$ for each level $k$ rather than using the bound $A$ for all levels. Mathematically, in the pruned enumeration, we replace the inequality $l_k < A$, at each level $k$ by $l_k < A_k$, where $A_k$ is the bound at level $k$. The bound $A_k$ at level $k$ is driven from the bound $A$ using a function called a bounding function. A well-chosen bounding function helps significantly in enumerating rapidly the tree and at the same time leads to a solution. Actually, there are many bounding functions; the famous one is the linear bounding function proposed by Schooner and Euchner [14], which is $A_k = A \cdot \min(1, (1.05)k/n)$. The pruned enumeration algorithm is similar to Algorithm 1 except that we compute $A_k$ before Step 3 and replace $A$ by $A_k$ in Step 6 of Algorithm 1.

## 5. The Proposed GPU-Based Pruned Enumeration

In this section, we propose a GPU-based pruned enumeration algorithm (Algorithm 4) that searches for the shortest vector in a lattice. This algorithm is an improved version of the GPU algorithm described by Hermans et al. [28], which is in turn based on the enumeration algorithm ENUM proposed by Schnorr and Euchner [14]. GPU-based pruned enumeration algorithm can be easily adapted to be a GPU-based classical enumeration algorithm just by eliminating the pruning condition. The improvements are done by introducing three strategies for speeding up the running time of the pruned enumeration Algorithm 2. The first strategy, presented in Section 5.1.1, is used to improve the quality of the lattice basis by finding a better basis that speeds up the enumeration. The second strategy is based on expecting the best level $\alpha$ on which the cost of transferring the data between CPU and GPU is very low and is presented in Section 5.1.2. The third strategy, presented in Section 5.2, is based on gaining the improvement based on GPU, which can be carried out by generating some subtrees in GPU rather than in CPU.

The pseudocode of the proposed GPU-based pruned enumeration algorithm is shown in Algorithm 4. It consists of three different phases: the preprocessing (Steps 3 and 4), sequential part (Steps 6–14 and Steps 30–34), and parallel part (Steps 15–29). In Step 3, we call Algorithm 2 to apply the strategy for choosing the better basis. In Step 4, we apply the second strategy by choosing a good level $\alpha$ for exchanging the data between GPU and CPU. The sequential part is executed on CPU. It works as follows: first, we generate GPU-points on CPU using Algorithm 3; then, the generated GPU points are sent to run on GPU. Once the execution of GPU points on GPU is terminated, Algorithm 3 is used again to generate another GPU point for the next launch of GPU. The parallel part runs on GPU. This part processes in the same way as the parallel part of the parallel algorithm in [28], except that when a subtree enumeration is terminated (Step 16), its adjacent subtree corresponding to a neighbor point is generated on GPU to run on the same thread. If the current neighbor point is cut off just as it is generated (Step 21), the subtree enumeration is canceled and a new start point is assigned to the current thread (Step 26). GPU points in the set $G$ complete their subtree enumerations on threads in GPU. In the same way as in [28], a subtree enumeration in GPU is performed in a zig-zag pattern and depth-first search order, starting at level $\alpha$ moving towards the leaf level (at level 0).

*5.1. Preprocessing Strategies.* In this section, we propose two preprocessing strategies for choosing a better basis and a better level $\alpha$ in Algorithm 4. We use a combination of randomization and the Gaussian heuristic in Section 5.1.1 to expect a better basis that leads rapidly to a shortest vector; also, in Section 5.1.2, we expect the level on which the exchanging data between CPU and GPU is optimized. In [23], Gama et al. used randomization to increase the success probability of finding a shorter vector by generating many different basis for a lattice $L$, while our strategy in Section 5.1.1 aims to reducing the number of branches of search tree by finding a basis that is corresponding to a smaller search tree, i.e., the search tree that has a fewer branches.

The Gaussian Heuristic provides an estimate on the number of lattice points inside a *nice enough* set. More specifically, it says that, for a given lattice $L$ and a set $S$, the number of points in $S \cap L$ is approximately $|\mathrm{vol}(S)/\det(L)|$. In [20], Hanrot Stehlé noticed (and verified in [23]) that the number of nodes at depth $k$ could be estimated from the Gaussian heuristic as follows:

$$H_k = \frac{1}{2} \cdot \frac{V_k(A)}{\prod_{i=n+1-k}^{n} \left\| b_i^* \right\|} = \frac{1}{2} \cdot \frac{V_k(A)}{\mathrm{vol}(\pi_k(L))}, \tag{6}$$

where $V_k(A)$ denotes the volume of the $k$-dimensional ball with radius $A$.

Recall that the volume of the $k$-dimensional ball can be calculated by

$$V_k(A) = A^k \cdot \frac{\pi^{k/2}}{\Gamma((k/2)+1)}, \tag{7}$$

where $\Gamma$ denotes the gamma function [35].

From equation (7) and the volume of a unit sphere in dimension $n$, we can heuristically bound the number of nodes at depth $k$ in the Schnorr–Euchner Algorithm 2 by setting an initial bound $A$ of the lattice $L$ as follows:

$$A = \frac{\Gamma(n/2+1)^{1/n}}{\sqrt{\pi}} \cdot \det(L)^{1/n}. \tag{8}$$

*5.1.1. Strategy for Selecting Better Lattice Basis.* A basis is better if it leads more rapidly to the solution, i.e., if the number of nodes of the search tree corresponding to a basis is smaller; we propose to find a better basis by generating many basis using a randomization strategy which is carried out by multiplying the original basis by a random small unimodular matrix as shown in [6]. Then, we calculate the corresponding nodes for each basis using the Gaussian

(1) **Input**: Gram–Shmidt Coefficients $\mu_{i,j}$, for $1 \le i, j \le n$, $\|b_1^*\|^2, \|b_2^*\|^2, \ldots, \|b_n^*\|^2$ and $A$
(2) $v \longleftarrow (1, 0, \ldots, 0)$, $l \longleftarrow (0, 0, \ldots, 0)$, $c \longleftarrow (0, 0, \ldots, 0)$, $\Delta \longleftarrow (0, 0, \ldots, 0)$, $\delta \longleftarrow (1, 0, \ldots, 0)$, $u \longleftarrow (1, 0, \ldots, 0)$, $s = 1$ and $v_{\min} \longleftarrow (1, 0, \ldots, 0)$
(3) $k = 1$
(4) **While** $k \le n$ **do**
(5) $\quad l_k = l_{k+1} + (v_k + c_k)^2 \|b_k^*\|^2$
(6) $\quad$ **if** $l_k < A$ **then**
(7) $\quad\quad$ **if** $k > 1$ **then**
(8) $\quad\quad\quad k = k - 1$
(9) $\quad\quad\quad c_k = \sum_{i=k+1}^{n} v_i \mu_{i,k}$, $v_k = u_k = \lceil -c_k \rfloor$, $\Delta_k = 0$
(10) $\quad\quad\quad$ **if** $v_k > -c_k$ **then**
(11) $\quad\quad\quad\quad \delta_k = -1$
(12) $\quad\quad\quad$ **else**
(13) $\quad\quad\quad\quad \delta_k = 1$
(14) $\quad\quad\quad$ **end if**
(15) $\quad\quad$ **else**
(16) $\quad\quad\quad A \longleftarrow l_k$, $v_{\min} \longleftarrow v_k$
(17) $\quad\quad$ **end if**
(18) $\quad$ **else**
(19) $\quad\quad k = k + 1$, $s := \max(s, k)$
(20) $\quad\quad$ **if** $k < s$ **then** $\Delta_k = -\Delta_k$
(21) $\quad\quad$ **if** $\Delta_k \delta_k \ge 0$ **then** $v_k = u_k + \Delta_k$
(22) $\quad$ **end if**
(23) **end while**
(24) **Output**: $v_{\min}$ such that $v = \sum_{i=1}^{n} v_i b_i$ is the shortest vector

Algorithm 1: Schnorr and Euchner's algorithm of enumeration [14].

(1) **Input**: the basis $b_1, b_2, \ldots, b_n$
(2) rand = 1, $h = H = 0$, and $c_1, c_2, \ldots, c_n$
(3) Compute a bound $A$ using equation (8)
(4) **while** rand $\le 100$ **do**
(5) $\quad$ Randomize the basis $b_1, b_2, \ldots, b_n$
(6) $\quad$ Reduce the basis $b_1, b_2, \ldots, b_n$ using pruned BKZ
(7) $\quad R \longleftarrow$ the minimum norm of the reduced basis of $b_1, b_2, \ldots, b_n$
(8) $\quad$ **if** $R < A$ **then**
(9) $\quad\quad A \longleftarrow R$
(10) $\quad$ **end if**
(11) $\quad$ Compute $H$ using the bound $A$ and equation (6) $\triangleright H$ is the number of nodes
(12) $\quad$ **if** rand == 1 **then**
(13) $\quad\quad h \longleftarrow H + 1$
(14) $\quad$ **end if**
(15) $\quad$ **if** $H < h$ **then**
(16) $\quad\quad$ **for** $i \longleftarrow 1$ to $n$ **do**
(17) $\quad\quad\quad c_i \longleftarrow b_i$
(18) $\quad\quad$ **end for**
(19) $\quad\quad h \longleftarrow H$
(20) $\quad$ **end if**
(21) $\quad$ rand $\longleftarrow$ rand + 1
(22) **end while**
(23) **Output**: the basis $c_1, c_2, \ldots, c_n$

Algorithm 2: Generation of a better basis.

heuristic as in equation (6). Algorithm 2 describes this strategy, where the variable rand refers to the number of generated basis.

*5.1.2. Strategy for Determining Level $\alpha$.* Choosing the level $\alpha$ can be considered a challenging task for implementing the enumeration algorithm efficiently on GPU because the level

(1) **Input**: Gram–Shmidt coefficients $\mu_{i,j}$ for $1 \le i, j \le n$, $\|b_1^*\|^2, \|b_2^*\|^2, \ldots, \|b_n^*\|^2$ and $A$
(2) $v \longleftarrow (1, 0, \ldots, 0)$, $l \longleftarrow (0, 0, \ldots, 0)$, $c \longleftarrow (0, 0, \ldots, 0)$, $\Delta \longleftarrow (0, 0, \ldots, 0)$, $\delta \longleftarrow (1, 0, \ldots, 0)$, $u \longleftarrow (1, 0, \ldots, 0)$, $s = 1$, $v_{\min} \longleftarrow (1, 0, \ldots, 0)$, $\alpha$ and $N$
(3) $\mathrm{SV} = \varnothing$, $S\Delta = \varnothing$
(4) $t = n$
(5) **while** $t \le n$ **do**
(6) $\quad l_t = l_{t+1} + (v_t + c_t)^2 \|b_t^*\|^2$
(7) $\quad$ **if** $l_t \le A$ **then**
(8) $\quad\quad$ **if** $t == \alpha$ **then**
(9) $\quad\quad\quad \mathrm{SV} = \mathrm{SV} \cup \{v_\alpha, v_{\mathrm{beta}+1}, \ldots, v_n\}$
(10) $\quad\quad\quad S\Delta = S\Delta \cup \{\Delta_\alpha, \Delta_{\mathrm{beta}+1}, \ldots, \Delta_n\}$
(11) $\quad\quad\quad$ **if** $|SV| \ge N$ **then**
(12) $\quad\quad\quad\quad$ **return**
(13) $\quad\quad\quad$ **end if**
(14) $\quad\quad\quad$ **while** $l_t < A$ **do**
(15) $\quad\quad\quad\quad$ update $\Delta_t$ and $v_t$
(16) $\quad\quad\quad\quad l_t = l_t + (v_t + c_t)^2 \|b_t^*\|^2$
(17) $\quad\quad\quad$ **end while**
(18) $\quad\quad$ **else**
(19) $\quad\quad\quad$ **if** $t > 1$ **then**
(20) $\quad\quad\quad\quad t = t - 1$
(21) $\quad\quad\quad\quad c_t = \sum_{i=t+1}^{n} v_i \mu_{i,t}$, $v_t = u_t = \lceil -c_t \rfloor$, $\Delta_t = 0$
(22) $\quad\quad\quad\quad$ **if** $v_t > -c_t$ **then**
(23) $\quad\quad\quad\quad\quad \delta_t = -1$
(24) $\quad\quad\quad\quad$ **else**
(25) $\quad\quad\quad\quad\quad \delta_t = 1$
(26) $\quad\quad\quad\quad$ **end if**
(27) $\quad\quad\quad$ **else**
(28) $\quad\quad\quad\quad A \longleftarrow l_t$, $v_{\min} \longleftarrow v_t$
(29) $\quad\quad\quad$ **end if**
(30) $\quad\quad$ **end if**
(31) $\quad$ **else**
(32) $\quad\quad$ select $v_t$ new value using zig-zag pattern
(33) $\quad$ **end if**
(34) **end while**
(35) **Output**: the sets of GPU points SV and $S\Delta$

Algorithm 3: Generation of GPU points.

$\alpha$ that is very close to the root increases the computation load on GPU. However, it reduces the possible points on that level. On the contrary, the level $\alpha$ that is very far from the root mitigates the computation load on GPU, but it increases massively the number of possible points sent to GPU, which means that the number of communications between GPU and CPU would be increased substantially. Consequently, the main challenge is how to strike a balance between the computation load on GPU and the number of communications between CPU and GPU. In other words, the challenge is how we can determine the best choice of level $\alpha$ that combines the two objectives of mitigating the computation load and the number of communications between CPU and GPU at the same time. In this section, we propose a strategy for this challenge. The strategy we use is based on using Gaussian heuristic to compute the expected nodes on the candidate levels. And then by approximating time taken for running a node on GPU, we can expect the execution time for implementing all nodes of enumeration for each level by multiplying the number of nodes by the time approximate for running a node on GPU. Finally, we choose the best choice for level $\alpha$ which is corresponding to the smallest expected execution time. For example, assume that we have a lattice $L$ of dimension 100 and ten candidate levels from $\alpha = 31$ to $\alpha = 40$. Firstly, we calculate the expected number of nodes $H_\alpha$ on each level $\alpha$ using equation (6). Then, we run the GPU's kernel for each candidate level for a short time, and by using a CUDA utility called *nvprof*, we can get the average time taken for executing GPU's kernel. Then, by dividing this time by the number of nodes per GPU's kernel, we get an approximate execution time for a node. Assume that the execution times for a node are $\mathrm{Time}_{31}, \ldots, \mathrm{Time}_{40}$, corresponding to levels $\alpha = 31, \ldots, \alpha = 40$, respectively. Now, we can expect the execution time for a level $\alpha$ by multiplying the approximate time taken for executing a node by the number of nodes for this level. More precisely, let $T_{31}, \ldots, T_{40}$ denote the expected executions times for all nodes on levels $\alpha = 31, \ldots, \alpha = 40$, respectively. Then, $T_{31}, \ldots, T_{40}$ are computed as follows: $T_{31} = H_{31}^* \mathrm{Time}_{31}, \ldots, T_{40} = H_{40}^* \mathrm{Time}_{40}$. The level corresponding to the smallest value among $T_{31}, \ldots, T_{40}$ is selected as the best level for generating the GPU points.

(1) **Input**: basis $b_1, b_2, \ldots, b_n$, GSIZE and a small integer $q \approx 5000$
(2) Compute the quadratic norm of the Gram–Schmidt orthogonalization basis and coefficients $\|b_1^*\|^2, \|b_2^*\|^2, \ldots, \|b_n^*\|^2$ and $\mu_{i,j}$ for $1 \le i, j \le n$
(3) Generate a better basis using Algorithm 2
(4) Determine the best choice of level $\alpha$ using the strategy shown in Section 5.1.2, $G = \varnothing$ and $R = \varnothing$
(5) **while** true **do**
(6) $T$ = GSIZE − $|R|$ ▷ denotes the number of elements
(7) Generate $T$ GPU-points $p_1, \ldots, p_N$ on level $\alpha$ using Algorithm 3
(8) $G = G \cup \{p_1, \ldots, p_N\}$
(9) **if** $|G| \le q$ **then**
(10) Enumerate the GPU-points in $G$ on CPU
(11) break;
(12) **end if**
(13) Copy $G$ to GPU memories
(14) Start the GPU stage
(15) **for each** thread in GPU **do**
(16) **if** the current enumeration corresponding to a start-point $p$ is cut off or reaches level $\alpha$ **then**
(17) Produce a $g$ neighbor-point of the start-point $p$
(18) Calculate $l_\alpha$ for the neighbor-point $g$
(19) **if** $l_\alpha \le A$ **then**
(20) The enumeration corresponding to $g$ is launched on the current thread
(21) **else**
(22) **if** this start-point $p$ is the last point in $G$ **then**
(23) Stop enumeration on GPU and return to CPU
(24) **end if**
(25) Set a flag that indicates that the start-point $p$ terminates its corresponding enumeration
(26) Select a new start-point and launch its corresponding enumeration on the current thread
(27) **end if**
(28) **end if**
(29) **end for each**
(30) Stop the GPU stage
(31) Copy start points that terminate their enumerations in GPU to $R$
(32) $G = G - R$
(33) **end while**
(34) **Output**: a shorter vector $v = (v_1, v_2, \ldots, v_n)$

Algorithm 4: The proposed GPU-based pruned enumeration.

*5.2. Strategy for Improving Uses of GPU.* The overhead of transmitting data between the CPU and GPU can be mitigated by decreasing the number of GPU points sent to GPU. In order to achieve that we propose to classify the GPU points into two types such that the transferring operation is needed only for the first types, while for the second type, the GPU points are generated on GPU rather than CPU. Let us call the first type of GPU points the start points and the second types of GPU points the neighbor points (Figure 1). Start points represent GPU points that cannot be generated depending on its adjacent point without moving subsequently up in the tree. While neighbor points are GPU points that can be easily expected based on the values of their adjacent points without having to move in the tree. For example, assume that we have five points $p_1$, $p_2$, $p_3$, $p_4$, and $p_5$ on the level $\alpha$. Let $l_{p_1\alpha}$, $l_{p_2\alpha}$, $l_{p_3\alpha}$, $l_{p_4\alpha}$, and $l_{p_5\alpha}$ be the corresponding values of $p_1$, $p_2$, $p_3$, $p_4$, and $p_5$, respectively. The point $p_1$ is sent to GPU if the inequality $l_{p_1\alpha} \le A$ holds, otherwise it is cut off. As depicted in Figure 1, since $l_{p_1\alpha} \le A$, then the next value of $\Delta_\alpha$ can be calculated easily (Step 20 of Algorithm 1). Subsequently, as in Step 21 of Algorithm 1, the factor $v_{p_2\alpha}$ is calculated, so we can say that the point $p_2$ is a neighbor point since it is generated just by using the integers $\Delta_\alpha$ and $v_\alpha$ of its adjacent point $p_1$. Similarly, we can generate the neighbor point $p_3$. Now, we are at the point $p_4$, and since $l_{p_4\alpha} > A$, then the enumeration algorithm moves up, so the factor $v_{p_5\alpha}$ of the point $p_5$ will be calculated using Step 9 of Algorithm 1, which depends on the values of $v_k$ for $n \le k \le \alpha - 1$. Consequently, point $p_5$ cannot be generated without revealing those values of $v_k$. In other words, from point $p_4$, we have to move in the tree until we reach the point $p_5$ and then those values will be revealed. Thus, we can say that the point $p_5$ is a start point.

*5.2.1. Generating GPU-Point Algorithm.* The improvement of strategy in Section 5.2 is mainly applied in Steps 7–18 of Algorithm 3 and Steps 16–21 of Algorithm 4. We showed the workflow of Algorithm 4. Now, we will show how Algorithm 3 is implemented. Algorithm 3 is implemented in CPU in the same way as in [28]. It starts by initializing the value of $v$ at the root level by $(1, 0, 0, \ldots, 0)$ (Step 2). At the root level (top level), the only revealed value is $v_n = 1$; the other values $v_{n-1}, \ldots, v_\alpha$ will be revealed during traversing the tree in the

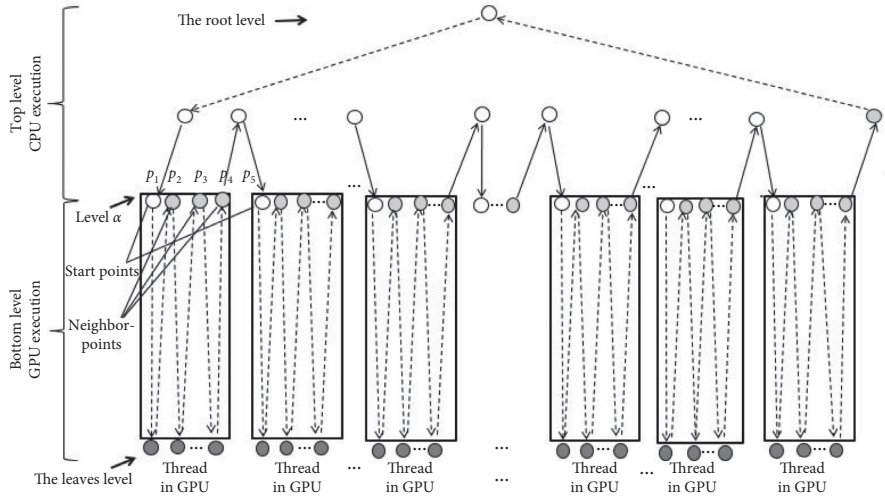


Figure 1: Our parallel enumeration.

zig-zag pattern and depth-first search order towards the level $\alpha$. When the enumeration reaches the level $\alpha$, the search order is changed to be a breadth-first search order as long as $l_\alpha \leq A$. The difference between our Algorithm 3 and the algorithm for generating GPU points in [28] is that we generate only the start points. In fact, for each GPU point $p$ at level $\alpha$, we have two cases:

(1) $l_\alpha \leq A$ (Step 7): in this case, the GPU point $p$ is accepted and added to the set $G$ (the set of GPU point sent subsequently to GPU); then, the tree is enumerated in the breadth-first search order for revealing the next point. However, in this case, the next point is considered as the neighbor point, so there is no need to be added to $G$, and it can be generated easily afterward in GPU using Steps 16–18 of Algorithm 4. For each GPU point $p \in G$, we store the values $(v_\alpha, v_{\alpha+1}, \ldots, v_n)$ and $(\Delta_\alpha, \Delta_{\alpha+1}, \ldots, \Delta_n)$, and these values are used to initialize the data of the GPU point in GPU.

(2) $l_\alpha > A$, in this case, the subtree corresponding to the point $p$ is cut off and the enumeration moves up to the parent point using the zig-zag pattern (Steps 20–26 of Algorithm 3). Thus, the enumeration algorithm continues moving up and down in the depth-first search order until it returns back to the level $\alpha$ with a new value $l_\alpha < A$. Then, the corresponding point is taken as a new GPU point, and we use the same procedure as in case (1).

## 6. Experimental Results

In this section, we present the results obtained by implementing the proposed strategies. In addition, some analysis were conducted to show the high performance of GPU achieved by using improvements of Sections 5.1.2 and 5.2. Our implementations of pruned enumeration and classical enumeration are compared to Hermans's pruned enumeration [28] and Correia's enumeration [30], respectively. This section is organized as follows. In Section 6.1, we present the platform specification and the dataset of our implementation. In Section 6.2, we describe in detail the implementation results showing the execution time achieved by using each proposed strategy for pruned enumeration, and we also compare our implementation results to Hermans's implementation results. In Section 6.3, we show the implementation results for classical enumeration using the proposed strategies. Then, we compare our GPU execution time to CPU multicores execution time of Correia's implementation.

*6.1. Platform Specification and Dataset.* In order to examine our algorithm's improvements, we use two Nvidia GeForce GTX 1060 GPUs for the parallel execution with a CPU of Intel Core 2 Quad at 2.83 GHz for the sequential execution such that each GPU runs along with a separated core of CPU. The used software is CUDA 9.1 platform installed on Ubuntu Linux 16.10. Regarding dataset of our implementation, we distinguish between two types of dataset: the dataset for pruned enumeration and the dataset for classical enumeration. The dataset for pruned enumeration is generated using the NTL package, which is considered as one of the famous available mathematical packages for lattice. The use of NTL package allows us to generate more samples for each dimension and to conduct more examinations on our improvements. We generate five samples for each dimension, where a sample is a basis of a matrix of $n$ columns; the bit size of each element of each row is set to 64 bits. The dataset for classical enumeration is a single sample provided from SVP-Challenge (http://www.latticechallenge.org/svp-challenge/)for dimension 60. The dimension $n$ of the lattice is set to be varied between 80 and 110 for the pruned enumeration, while it is set to 60 for the classical enumeration. Level $\alpha$ on which we create the GPU points is set depending on the second strategy. In general, it ranges from $n - 20$ to $n - 42$ for the pruned enumeration, and it is set to $n - 14$ for the classical enumeration. The transferring GPU points from CPU to GPU for each launch of GPU's kernel are around 393,216 GPU points. Occupancy in GPU is

considered one of the main factors for obtaining full use of GPU, so we must set carefully the number of threads per block which plays an important role in boosting the implantation process of GPU. Our implementation tests show that the thread block of size 128 may be a good choice for our Nvidia card, and it yields to high computational performance.

The traditional procedure for improving the quality of the lattice basis is to use a reduction method before implementing the enumeration algorithm, e.g., LLL or BKZ reduction. Actually, we use BKZ reduction in our implementation. For pruned enumeration, we set BKZ block size to 30, 35, 35, and 45 for dimensions 80, 90, 100, and 110, respectively, while for classical enumeration, we set BKZ block size to 20 for dimension 60. There are two libraries available for lattice reduction: the NTL library of Shoup [21] and fpLLL library of Albrecht et al. [22]. The fpLLL library does not offer BKZ. Therefore, for our experiments, we use the NTL library.

*6.2. Implementation of Pruned Enumeration.* In this section, we present in detail the implementation process and the results obtained by implementing each of our proposed strategies for pruned enumeration. We compare our pruned enumeration implementation results to Hermans's pruned enumeration. For each dimension, we run our code and Hermans's code on the same five samples:

(1) The implementation of the strategy based on generating GPU points on GPU (Section 5.2): for examining this strategy, we compare our parallel GPU-based implementation against Hermans's parallel GPU-based implementation [28], where the platform specification and the dataset used in the two implementations are the same. According to the experimental results in Table 1, the improvement gained by using this strategy is around 1.3.

(2) The implementation of the strategy based on optimizing the selection of level $\alpha$: for each sample in dimension $n$, we compute the expected nodes on each level based on equation (6), and then by calculating the approximate time taken for executing GPU's kernel for nodes as illustrated in Section 5.1.2, we calculate the expected execution time. Finally, we select the best one among the candidate levels. For example, for a sample of dimension $n = 100$, as in Table 2, we can see that level 34 is expected to optimize the execution time. While in Table 3, level 32 is expected to optimize the execution time for $n = 90$.

(3) The implementation of the strategy based on boosting the basis quality using the randomization (Section 5.1.1): for each dimension $n$, we randomize the original basis 100 times; in each time, we conduct a BKZ reduction and using a Gaussian heuristic to obtain the expected nodes for each randomization. The result obtained is corresponding to the basis of fewer nodes. This strategy speeds up the execution time by a factor of almost 1.5, as shown in Table 1.

Table 1: Average execution time in seconds for our implementations and Hermans's implementation [28].

| Hermans et al. [28] with | Dimension | | | |
|---|---|---|---|---|
| | 80 | 90 | 100 | 110 |
| I1: Sections 5.2, 5.1.1, and 5.1.2 | 287 | 1450 | 4876 | 30,726 |
| I2: Sections 5.2 and 5.1.1 | 365 | 2177 | 6305 | 40,836 |
| I3: Sections 5.2 | 242 | 1764 | 9360 | 59,482 |
| I4: — | 297 | 2398 | 12,125 | 77,982 |
| Speedup of I1 compared to I4 | 1.03 | 1.65 | 2.49 | 2.54 |
| Speedup of I2 compared to I4 | 0.81 | 1.1 | 1.92 | 1.91 |
| Speedup of I2 compared to I3 | 0.66 | 0.81 | 1.48 | 1.46 |
| Speedup of I3 compared to I4 | 1.23 | 1.36 | 1.30 | 1.31 |

The results of Table 1 and Figure 2 show that a combination of the three strategies improves the execution time of Hermans's pruned enumeration by a factor of almost 2.5.

Finally, we show the scalability of our implementation of the strategies by using two GPUs. The results of Table 4 show that the achieved improvements using two GPUs are almost twice faster than the improvements of a single GPU. More precisely, two GPUs speed up the execution time of Hermans's pruned enumeration by a factor of almost 5 (Figure 2).

*6.3. Implementation of Classical Enumeration.* In this section, we present the results of implementing our strategies on classical enumeration. We compared our implementation results to Correia's implementation results [30]. For dimension 60, we used a single sample obtained from SVP challenge. We got the execution time of Correia's implementation from [30].

The results depicted in Figure 3 show that the proposed strategies are also efficient for classical enumeration. A challenge of dimension 60 can be solved in 45 seconds and 24 seconds using a single GPU and two GPUs, respectively. While a dual-socket machine with 16 physical cores and simultaneous multithreading technology [30] solved the same challenge of dimension 60 in 47 seconds. Thus, our improvements with two GPUs are twice faster than 16 physical cores provided by simultaneous multithreading technology for the challenge of dimension 60.

*6.4. GPU Performance Analysis.* Improvements by using some features of GPU have already been applied in Hermans et al. [28] such as using registers and shared memory [36]. Although they improved the execution time by reducing the latency, they affected the occupancy [36] since the sizes of registers and shared memory are very limited. In general, using of registers and shared memory achieves improvements of the GPU performance. We also improved the efficiency of accessing the global memory by applying coalescing access of the global memory [36].

In this section, we analyze the performance of our strategies using GPU factors. The known factors of GPU for evaluating the performance are [36] occupancy, divergence, global load efficiency, and global store efficiency. The last two factors measure the efficiency of

Table 2: An example of selecting level $\alpha$ on which GPU points are generated, for dimension $n = 100$.

| Level | Expected nodes | Execution time for 393,216 nodes | Expected time | Real time |
|---|---|---|---|---|
| 27 | 52,967,138 | 312 | 42,027 | 31,209 |
| ... | ... | ... | ... | ... |
| 31 | 121,978,431 | 58 | 17,992 | 7,966 |
| 32 | 35,750,903 | 45 | 4,091 | 6,603 |
| 33 | 167,652,863 | 30 | 12,850 | 5,657 |
| 34 | 46,247,260 | 25 | 2,940 | 4,895 |
| 35 | 216,875,236 | 20 | 11,030 | 4,746 |
| 36 | 625,052,451 | 12 | 20,092 | 4508 |
| 37 | 3,663,833,922 | 9.86 | 91,871 | 5,233 |
| 38 | 1,044,016,727 | 4.15 | 11,620 | 5,384 |
| 39 | 5,351,034,447 | 2,42 | 32,932 | 5,872 |

Table 3: An example of selecting level $\alpha$ on which GPU points are generated, for dimension $n = 90$.

| Level | Expected nodes | Execution time for 393,216 nodes | Expected time | Real time |
|---|---|---|---|---|
| 27 | 367,594,493 | 14.32 | 13,386 | 850 |
| 28 | 159,673,404 | 8.17 | 3,317 | 495 |
| 29 | 946,459,138 | 4.27 | 10,277 | 493 |
| 30 | 330,757,526 | 2.46 | 2,069 | 489 |
| 31 | 1,727,309,500 | 1.66 | 7,292 | 504 |
| 32 | 535,645,634 | 1.16 | 1,081 | 497 |
| 33 | 2,659,143,869 | 0.865 | 5,849 | 613 |
| 34 | 1,085,745,445 | 0.526 | 1,725 | 722 |
| 35 | 27,763,817,440 | 0.301 | 21,252 | 885 |

Table 4: Average execution time in seconds of our improvements using a single GPU and two GPUs.

| Hermans et al. [28] with Sections 5.2, 5.1.1, and 5.1.2 using | Dimensions | | | |
|---|---|---|---|---|
| | 80 | 90 | 100 | 110 |
| Single GPU | 287 | 1450 | 4876 | 30,726 |
| Two GPUs | 143 | 684 | 2469 | 15512 |
| Speedup of two GPUs compared to a single GPU | 2.01 | 2.12 | 1.97 | 1.98 |

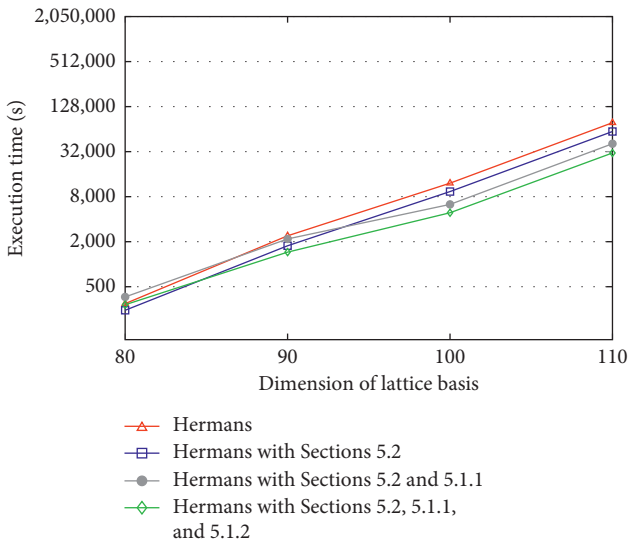


Figure 2: Average execution time in seconds for Hermans et al. [28] and our improvements.

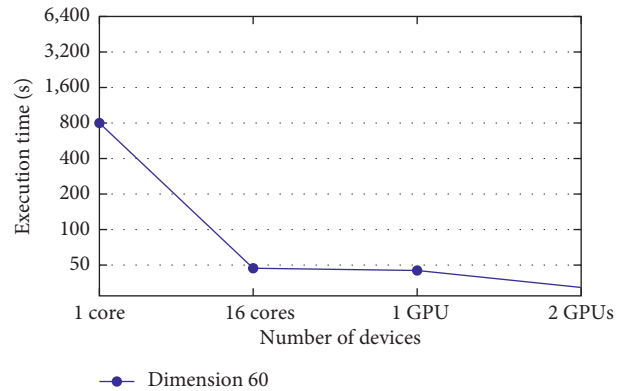


Figure 3: Average execution time in seconds for our implementations using 1 GPU and 2 GPUs and Correia's implementation [30].

using the global memory: the reading and writing efficiencies. We show that our strategies based on GPU likes "selecting better level $\alpha$" and "generating GPU points on

Table 5: GPU performance analysis with strategy of Section 5.1.2 for dim. 90.

| Performance factor | Level $\alpha$ | | | |
|---|---|---|---|---|
| | 64 | 58 | 52 | 46 |
| Execution time (s) | 1029 | 621 | 2090 | 6216 |
| Number of launching GPU's kernels | 63 | 616 | 8202 | 25145 |
| Global load efficiency (%) | 26 | 26 | 27 | 29 |
| Global store efficiency (%) | 19 | 21 | 80 | 95 |
| Branch efficiency (%) | 81 | 82 | 88 | 94 |
| Occupancy (%) | 25 | 25 | 25 | 25 |

Table 6: GPU performance analysis with strategies of Sections 5.1.2 and 5.2 for dim. 90.

| Performance factor | Level $\alpha$ | | | |
|---|---|---|---|---|
| | 64 | 58 | 52 | 46 |
| Execution time (s) | 955 | 497 | 1809 | 5362 |
| Number of launching GPU's kernel | 21 | 402 | 5757 | 17104 |
| Global load efficiency (%) | 29 | 29 | 30 | 31 |
| Global store efficiency (%) | 14 | 20 | 58 | 61 |
| Branch efficiency (%) | 78 | 81 | 87 | 92 |
| Occupancy (%) | 25 | 25 | 25 | 25 |

GPU" improve significantly the performance of GPU. As in Table 5, "selecting of level $\alpha$" plays a key role for improving the performance of GPU. It reduces substantially the number of lunching GPU's kernel. However, it affects some factors of GPU performance, for example, global store efficiency and branch efficiency. As in Table 5, global store efficiency decreases as level $\alpha$ moves up because when level $\alpha$ moves up, more variables are needed to be stored in registers. Since the registers are very limited, additional data are split over to a free space on the global memory. As a result, the global store efficiency is affected.

Table 6 shows that the strategy of "generating GPU points on GPU" decreases the time consumed for launching GPU's kernel and transferring data between CPU and GPU. However, with this strategy, we may lose coalescing access to some data, and that is the reason why the global store efficiency decreases. Additionally, in order to generate GPU points on GPU, we need to add some branches to our implementation. Consequently, the branch efficiency is affected.

## 7. Conclusions and Future Work

In this paper, we proposed three strategies for improving the pruned enumeration for solving SVP. A strategy based on generating GPU points on GPU, a strategy based on improving the quality of basis by using a randomization approach, and a strategy based on expecting the best level on which we generate the GPU points. We showed that a significant speeding up could be gained by using those strategies altogether. The experimental results showed that a speeding up by a factor of almost 2.5 and 5 was achieved using a single GPU and two GPUs, respectively. Additionally, we showed that two GPUs are twice faster than 16 physical cores provided by simultaneous multithreading technology. For future work, we plan to implement these strategies using multi-GPUs with multicores.

## Data Availability

The data used to support the finding of the study are available from the first author upon request.

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## References

[1] M. Ajtai, "The shortest vector problem in L2 is NP-hard for randomized reductions," in *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pp. 10–19, ACM, Dallas, TX, USA, May 1998.

[2] P. van Emde Boas, "Another NP-complete problem and the complexity of computing short vectors in a lattice," Tecnical Report, Department of Mathmatics, University of Amsterdam, Amsterdam, Netherlands, 1981.

[3] T. Güneysu, V. Lyubashevsky, and T. Pöppelmann, "Practical lattice-based cryptography: a signature scheme for embedded systems," in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 530–547, Springer, Worcester, MA, USA, August 2012.

[4] H. James, T. Pöppelmann, M. O'neill, E. O'sullivan, and T. Güneysu, "Practical lattice-based digital signature schemes," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 3, pp. 1–24, 2015.

[5] O. Regev, "Lattice-based cryptography," in *Proceedings of the Annual International Cryptology Conference*, pp. 131–141, Springer, Santa Barbara, CA, USA, August 2006.

[6] O. Goldreich, S. Goldwasser, and S. Halevi, "Public-key cryptosystems from lattice reduction problems," in *Proceedings of the Annual International Cryptology Conference*, pp. 112–131, Springer, Santa Barbara, CA, USA, August 1997.

[7] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: a ring-based public key cryptosystem," in *Proceedings of the International Algorithmic Number Theory Symposium*, pp. 267–288, Springer, Portland, OR, USA, June 1998.

[8] R. Lindner and C. Peikert, "Better key sizes (and attacks) for LWE-based encryption," in *Proceedings of the Cryptographers Track at the RSA Conference*, pp. 319–339, Springer, San Francisco, CA, USA, February 2011.

[9] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing, STOC '05*, pp. 84–93, ACM, Baltimore, MD, USA, May 2005.

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

[11] N. Gama, N. Howgrave-Graham, H. Koy, and P. Q. Nguyen, "Rankin's constant and blockwise lattice reduction," in *Proceedings of the Annual International Cryptology Conference CRYPTO 2006*, pp. 112–130, Springer, Barbara, CA, USA, August 2006.

[12] N. Gama and P. Q. Nguyen, "Finding short lattice vectors within mordell's inequality," in *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, pp. 207–216, Victoria, Canada, May 2008.

[13] D. Micciancio and M. Walter, "Practical, predictable lattice basis reduction," in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 820–849, Springer, Vienna, Austria, May 2016.

[14] C.-P. Schnorr and M. Euchner, "Lattice basis reduction: improved practical algorithms and solving subset sum problems," in *Proceedings of the International Symposium on Fundamentals of Computation Theory*, pp. 68–85, Springer, Gosen, Germany, September 1991.

[15] C.-P. Schnorr and H. H. Hörner, "Attacking the Chor-Rivest cryptosystem by improved lattice reduction," in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 1–12, Springer, Saint-Malo, France, May 1995.

[16] R. Kannan, "Improved algorithms for integer programming and related lattice problems," in *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pp. 193–206, ACM, New York, NY, USA, December 1983.

[17] D. Micciancio and P. Voulgaris, "A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations," *SIAM Journal on Computing*, vol. 42, no. 3, pp. 1364–1391, 2013.

[18] 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*, pp. 601–610, ACM, Crete, Greece, July 2001.

[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] G. Hanrot and D. Stehlé, "Improved analysis of Kannan's shortest lattice vector algorithm," in *Proceedings of the 27th Annual International Cryptology Conference on Advances in Cryptology*, pp. 170–186, Springer, Santa Barbara, CA, USA, August 2007.

[21] V. Shoup, *NTL: A Library for Doing Number Theory*, http://www.shoup.net/ntl/, 2001.

[22] M. Albrecht, B. Shi, D. Cadé, X. Pujol, and D. Stehlé, "fpLLL-4.0, a floating-point LLL implementation," 2017, http://perso.ens-lyon.fr/damien.stehle.

[23] N. Gama, P. Q. Nguyen, and O. Regev, "Lattice enumeration using extreme pruning," in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques EUROCRYPT 2010*, pp. 257–278, Springer, French Riviera, France, May 2010.

[24] W. Backes and S. Wetzel, "Parallel lattice basis reduction using a multi-threaded Schnorr-Euchner LLL algorithm," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pp. 960–973, Delft, The Netherlands, August 2009.

[25] B. Werner and S. Wetzel, "Improving the parallel Schnorr-Euchner LLL algorithm," in *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing ICA3PP*, pp. 27–39, Melbourne, Australia, October 2011.

[26] J. Detrey, G. Hanrot, X. Pujol, and D. Stehlé, "Accelerating lattice reduction with FPGAs," *Lecture Notes in Computer Science*, Springer, Berlin, Germany, pp. 124–143, 2010.

[27] Ö. Dagdelen and M. Schneider, "Parallel enumeration of shortest lattice vectors," in *Proceedings of the 16th International Euro-Par Conference on Parallel Processing Euro-Par 2010*, pp. 211–222, Ischia, Italy, August 2010.

[28] J. Hermans, M. Schneider, J. A. 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 AFRICACRYPT 2010*, vol. 10, pp. 52–68, Stellenbosch, South Africa, May 2010.

[29] P.-C. Kuo, M. Schneider, Ö. Dagdelen et al., "Extreme enumeration on GPU and in clouds," in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems–CHES 2011*, pp. 176–191, Nara, Japan, September 2011.

[30] F. Correia, A. Mariano, A. Proenca, C. Bischof, and E. Agrell, "Parallel improved Schnorr-Euchner enumeration se++ for the CVP and SVP," in *Proceedings of the 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 596–603, IEEE, Crete, Greece, February 2016.

[31] H. Bahig and K. A. AbdElbari, "A fast GPU-based hybrid algorithm for addition chains," *Cluster Computing*, vol. 21, no. 12, pp. 2001–2011, 2018.

[32] W. Dai, Y. Doröz, and B. Sunar, "Accelerating NTRU based homomorphic encryption using gpus," in *Proceedings of the 2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, Waltham, MA, USA, September 2014.

[33] M. S. Esseissah, A. Bhery, and H. Bahig, "Improving BDD enumeration for LWE problem using GPU," *IEEE Access*, vol. 8, pp. 19737–19749, 2019.

[34] S. D. Galbraith, *Mathematics of Public Key Cryptography*, Cambridge University Press, Cambridge, UK, 2012.

[35] J. Hoffstein, J. Pipher, and J. H. Silverman, *An Introduction to Mathematical Cryptography*, Springer, Berlin, Germany, 2014.

[36] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*, John Wiley & Sons, Hoboken, NJ, USA, 2014.