JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 37, 1187-1209 (2021)
DOI: 10.6688/JISE.202109_37(5).0014

Parallelization on Gauss Sieve Algorithm over Ideal Lattice

PO-CHUN KUO[1], CHEN-MOU CHENG[2], WEN-DING LI[3] AND BO-YIN YANG[3]
[1]*Department of Electrical Engineering*
National Taiwan University
Taipei, 106 Taiwan
[2]*Graduate School of Natural Science and Technology*
Kanazawa University
Kanazawa, 920-1192 Japan
[3]*Institute of Information Science*
Academia Sinica
Taipei, 115 Taiwan
E-mail: {kbj; doug; thekev; by}@crypto.tw

Cryptanalysis of lattice-based cryptography is an important field in cryptography since lattice problems are among the most robust assumptions and have been used to construct a variety of cryptographic primitives. The security estimation model for concrete parameters is one of the most important topics in lattice-based cryptography. In this research, we focus on the Gauss Sieve algorithm proposed by Micciancio and Voulgaris, a heuristic lattice sieving algorithm for the central lattice problem, *shortest vector problem* (SVP). We propose a technique of *lifting* computations in prime-cyclotomic ideals into that in cyclic ideals. Lifting makes rotations easier to compute and reduces the complexity of inner products from $O(n^3)$ to $O(n^2)$. We implemented the Gauss Sieve on multi-GPU systems using two layers of parallelism in our framework, and achieved up to 55 times speed of previous results of dimension 96. We were able to solve SVP on ideal lattice in dimension up to 130, which is the highest dimension SVP instance solved by sieve algorithm so far. As a result, we are able to provide a better estimate of the complexity of solving central lattice problem.

Keywords: cryptography, parallel programming, lattice-based cryptography, sieving algorithm, gauss sieve, GPU, shortest vector problem, ideal lattices

1. INTRODUCTION

Lattice-based Cryptography Over the past two decades, lattice-based cryptosystems have attracted widespread interest from the cryptologic research community. Not only are they among the group of public-key cryptosystems (PKCs) that are potentially secure against threats from large quantum computers, but they also are the first to provide the constructions of many new cryptographic functionalities, *e.g.*, fully-homomorphic encryption [1]. Furthermore, in 1997 Ajtai and Dwork proved that some lattice problems possess worst-case to average-case reductions [2], which gave a strong guarantee on

Received September 23, 2019; accepted February 26, 2020.
Communicated by Meng Chang Chen.
* This work was supported by Ministry Of Science and Technology (MOST) Grants MOST 108-2221-E-001-008.

the security of lattice-based cryptosystems and inspired the construction of many cryptographic primitives [3]. A scheme is lattice-based scheme if their security is based on the lattice problem, *i.e.*, if the cryptosystems could be broken, the lattice problem could be solved. The lattice problem is a set of problems over lattice, and the central problem among lattice problem is *Shortest vector problem* (SVP).

Although many such constructions from general lattice are not efficient enough for daily use, ideal lattices have made both the keys shorter and running time faster, bringing many more new ideas closer to being practical. National Institute of Standards and Technology (NIST) announced the Post-Quantum Cryptography Standardization project, and the goal of this process is to select a number of acceptable candidate cryptosystems for standardization. In fact, most of the lattice based submissions for NIST used ideal lattice. Thus, estimating the hardness of the shortest vector problem over ideal lattice is the key to select concrete parameter for practical (ideal) lattice-based cryptosystems. However, it is not very clear to choose secure yet practical parameters for these schemes, and an accurate assessment of their security levels would be indispensable if we are to select suitable parameters for them.

Our Contribution

In this work, we implement the most practical variant of sieve algorithm on GPUs to solve SVP, using two layers of parallelism in our framework, and achieved up to 55 times speed of previous results in dimension 96. We use our implementation to estimate the hardness of SVP in order to select the concrete parameter for lattice-based cryptosystem. In this paper, we broaden the scope to include prime cyclotomic ideal lattices for the generality and contribute in the following ways.

- We propose and implement the first lattice sieving algorithm for a single machine with multiple GPUs. Our variant includes two carefully designed layers of parallelism, both inter-GPU and intra-GPU (Section 5).

- We show that by *lifting* lattice vectors generated by the polynomial $x^n + \cdots + 1$ into ones generated by $x^{n+1} - 1$, not only do inner products (the critical path of Gauss Sieve) speed up, some register rotation problems on GPUs are also mitigated (Section 3). Moreover, by heuristically applying *lazy rotation*, the complexity of reduction between two vectors with all their rotations goes down from $O(n^3)$ to $O(n^2)$ (only a constant times slower than the anticyclic lattice, cf. Section 3.3).

- We carefully crafted the reduction kernel to exploit both thread- and instruction-level parallelism (Section 6). Special care is taken with the layout of vectors in the register file, and some kernel-level heuristics are introduced that use the ideal lattice property.

- By incorporating these improvements into our implementation on GPUs, we were able to solve challenges of dimension 130 within 6583 GPU-hours, around 2^{55} GPU-cycles. This is the highest dimension SVP instance over ideal lattice has been solved by sieve algorithm so far.

- Our GPU implementation is 21.5 (resp. 55.8) times faster than a single-core CPU for general (resp. ideal) lattices. (Section 7.2)

- We provide a lower-bound complexity estimation for the SVP compared to the previous work (Section 7.4).

1.1 Background

Several algorithms have been proposed for the SVP or the approximate SVP. Exact algorithms include enumeration, sieving, and the ones based on Voronoi cells [4]. The Voronoi cell method, though having single exponential both in time and space complexity, proved to be impractical for higher dimensions (say, 100) [5]. On the other hand, lattice enumeration is an exhaustive search algorithm. The time complexity of lattice enumeration is $2^{O(n^2)}$ or $2^{O(n\log n)}$, where space complexity is polynomial [6, 7].

In general, lattice enumeration has remained the fastest approach to solve SVP in lower dimension, specifically, less than 150. In higher dimension, the running time of lattice enumeration algorithm is quite large because of its super-exponential time complexity. On the other hand, sieve algorithm is only single-exponential, so its running time grows much slower than that of lattice enumeration. Furthermore, the general approach is ill-suited for parallelizing on GPUs or similar wide vector architectures. For example, the speed-up achieved by Kuo *et al.* [7] is less than a factor of 10; the improvement in another work by Kuo *et al.* [8] is also around $10\times$. It is also unclear how to make use of the special structure of ideal lattices when using enumeration.

1.2 Sieving Algorithms

The first sieving algorithm was proposed by Ajtai, Kumar and Sivakumar in 2001 [9]. They proved that the time/space complexity is $2^{O(n)}$, which is asymptotically faster than enumeration algorithm. This property shows sieve algorithm will outperform enumeration in sufficiently high dimensions. Today, sieving algorithms are widely used to predict the hardness of SVP, for example, most of the lattice-based NIST submissions use sieving algorithm to estimate the security parameter for their cryptosystems [10].

As shown in Tables 1 and 2, the space is exponent of dimensions which is much worse than enumeration, but sieve algorithm has the advantage to operate on specific lattices, such as ideal lattices saving up to $\mathscr{O}(n)$ space. The sieve algorithm can be categorized into two groups: proven version and heuristics version, depending on whether or not heuristics are applied into the proof of algorithm. Besides the improvement of algorithm, Schneider adapted the ideal lattice into the sieve algorithm [11], and in turn reduce the space complexity in practice. Moreover, Laarhoven *et al.* incorporated locality-sensitive hashing into the algorithm [12, 13, 14, 15]. Instead of searching all the vector in the list, they group together near vectors using hash functions. Therefore, vectors are only reduced with geometrically more probable ones.

1.3 Gauss Sieve

The Gauss Sieve algorithm was proposed by Micciancio and Voulgaris in [18]. The complexity is shown in Table 2, but it achieves $2^{0.41n}$ time complexity and $2^{0.21n}$ space complexity in practice. The main idea of the algorithm is to mutually reduce samples with a list of vectors by Gauss reduction. After Gauss reduction, the angle between any pair of

Table 1. Provable sieve algorithm.

Algorithm	Time Complexity	Space Complexity
AKS Sieve [9, 16, 17]	$2^{3.346n+o(n)}$	$2^{2.173n+o(n)}$
ListSieve [18]	$2^{3.199n+o(n)}$	$2^{1.325n+o(n)}$
Birthday ListSieve [19]	$2^{2.465n+o(n)}$	$2^{1.233n+o(n)}$
Quantum Sieve [20]	$2^{1.799n+o(n)}$	$2^{1.286n+o(n)}$

Table 2. Sieve algorithm with heuristic.

Algorithm	Time Complexity	Space Complexity
GaussSieve [18]	$2^{0.52n+o(n)}$	$2^{0.41n+o(n)}$
NV-Sieve [16]	$2^{0.415n+o(n)}$	$2^{0.208n+o(n)}$
Two-Level Sieve [21]	$2^{0.3836n+o(n)}$	$2^{0.2557n+o(n)}$
Three-Level Sieve [22]	$2^{0.3778n+o(n)}$	$2^{0.2833n+o(n)}$
Overlattice Sieve [23]	$2^{0.3774n+o(n)}$	$2^{0.2925n+o(n)}$
Triple Sieve with NNS [20]	$2^{0.265n+o(n)}$	$2^{0.265n+o(n)}$
Sieve with Hyperplane LSH [12]	$2^{0.3366n+o(n)}$	$2^{0.2075n+o(n)}$
Sieve with Cross-Polytope LSH [14]	$2^{0.298n+o(n)}$	$2^{0.298n+o(n)}$
Sieve with Spherical LSF [13]	$2^{0.292n+o(n)}$	$2^{0.292n+o(n)}$
Quantum Sieve with NNS [20]	$2^{0.265n+o(n)}$	$2^{0.265n+o(n)}$

two vectors is larger than sixty degree. By the Kabatiansky-Levenshtein theorem, one can bound the number of such vectors, and thus obtain the time complexity of the algorithm.

Why we CHOOSE the Gauss Sieve?

First, compared to previous approaches on multi-core shared memory system, the latency of memory access across multi-GPU is large. To reduce the communication cost, we need to choose the variant of sieve with low space complexity. In practice, Gauss Sieve and LDSieve [13] have the lowest space complexity. Second, as we shall see, the simplicity of the Gauss Sieve allow us to design multi-level parallel architecture and achieve high parallel efficiency.

1.4 Previous Result Related to Implementation

The Gauss Sieve algorithm has been implemented on CPU in [24] and some improvements have been proposed by Bos *et al.*, in [25]. The work of Ishiguro *et al.* improved and implemented the parallel version of the Gauss Sieve algorithm proposed by Schneider [11], and they also adapt to a specific ideal lattice called negacyclic ideal lattices. They solved a 128-dimensional SVP instance generated by the cyclotomic polynomial $x^{128}+1$ in Ideal Lattice Challenge from TU Darmstadt [26] using about 29,994 CPU hours, which is around 2^{58} CPU cycles. Later, Bos *et al.* proposed a different variant of the parallel Gauss Sieve algorithm which is more suited for high dimension lattice [25]. We will expound on their ideas in Section 5. Later, in 2015, Mariano *et al.* implemented the HashSieve on multi-core CPU, their implementation solved the SVP in dimension 96, in 17.38 hours, using 16 physical cores [27]. Following by [28], they further scale the HashSieve to 60 cores and achieve parallel efficiency of 71.2% comparing to [27].

2. PRELIMINARY

2.1 Definition and Notation

A *lattice* is a discrete additive group of all integer combinations of a basis $v_1, v_2, ..., v_m \in \mathbb{R}^n$, where $m \leq n$. In cryptography, integer lattices are often used, namely, the basis vectors are defined over $\mathbb{Z}^n$. The bases corresponding to a lattice are not unique. We use $\mathscr{L}(B)$ to denote the lattice spanned by the basis $B = [b_1, b_2, ..., b_m]$.

The first successive minimum $\lambda_1(\mathscr{L})$ is the length of the shortest nonzero vector of the lattice $\mathscr{L}$. The shortest vector problem, or SVP for short, asks for the shortest nonzero vector in a given lattice. The SVP is NP-hard under randomized reduction [29]. The approximation shortest vector problem (SVP$_\alpha$) asks for a short vector of length shorter than $\alpha\lambda_1(\mathscr{L})$, where $\alpha \geq 1$. For example, the SVP Challenge [26] asks for $\alpha \leq 1.05$.

Extending the idea into rings, we have *ideal lattices*, a special class of lattices. Consider an ideal of a ring $\mathscr{I} = \langle g \rangle \subseteq \mathbb{Z}[x]/f(x)$, where f is a monic irreducible polynomial of degree n, an ideal lattice is $\mathscr{L}(B) \in \mathbb{Z}^n$ such that $B = \{g \mod f : g \in \mathscr{I}\}$. The polynomial of a ring affects its structure and computation cost. Thus, cryptographers are concerned with four type of ideal lattices defined by the polynomial $f(x)$:

- *Cyclic* ideal lattice, with $f_{cyclic}(x) = x^n - 1$, are the simplest ones and easy to compute. However, since the polynomials are always divided by $x - 1$, this kind of ideal lattice does not guarantee the worst-case collision resistance.
- *Anti-cyclic* ideal lattice, with $f_{anti\text{-}cyclic}(x) = x^n + 1$, are also eligible for easy multiplication and convolution. Such polynomials are irreducible over $\mathbb{Z}$ if n is a power of 2. This kind of ideal lattice is commonly used in cryptography.
- *Prime-cyclotomic* ideal lattices, with $f_{prime\text{-}cyclotomic}(x) = x^{n-1} + x^{n-2} + \cdots + 1$, are the main type we focus on. If n is prime, $f_{prime\text{-}cyclotomic}(x)$ is irreducible.
- *Trinomial* ideal lattices, with $f_{trinomial}(x) = x^n + x^{n/2} + 1$ where $n/2$ is a power of three, are the ones considered in [24].

In ideal lattice, the vector $u = (u_0, u_1, \cdots, u_{n-1}) \in \mathbb{Z}^n$ also indicates a polynomial $u(x) = u_0 + u_1 x + \cdots + u_{n-1}x^{n-1} \in \mathbb{Z}[x]/f(x)$, the polynomial $x \cdot u(x)$ is still in the ideal. Thus, the vector corresponding to such polynomial is called the $(first)$ $rotation$ of u, denoted as $\mathbf{rot}(u)$. For example, consider $f(x) = x^n - 1$, the rotation of $u = (u_0, u_1, \cdots, u_{n-1})$ is $\mathbf{rot}(u) = (u_{n-1}, u_0, u_1, \cdots, u_{n-2})$.

2.2 Gauss Reduction

The core routine of the Gauss Sieve is Gauss reduction. Two vectors $u, v \in \mathscr{L}(B)$ satisfying $\|u \pm v\| \geq \max(\|u\|, \|v\|)$ are called *Gauss-reduced*. Given two arbitrary vectors u and v, we can reduce u with respect to v by $u \leftarrow u - \lfloor \frac{\langle u,v \rangle}{\langle v,v \rangle} \rceil v$. Thus, given two arbitrary

Algorithm 1 : Gauss reduction for general lattices: reduce list u by list v

```
Require: Lists $U$ and $V$
Ensure: Reduced list $U$ by list $V$
1: for each $u \in U$ do
2:     for each $v \in V$ do
3:         if $2 \cdot |\langle u,v\rangle| > \langle v,v\rangle$ then
4:             $u \leftarrow u - \lfloor \frac{\langle u,v\rangle}{\langle v,v\rangle} \rceil v$
5:             Mark $u$ as reduced.
6:         end if
7:     end for
8: end for
```

vectors u and v, we can convert them into Gauss-reduced ones by repetitively applying the reduction procedure alternatively, in a Euclidean algorithm-like manner, until the vectors no longer change. If any two vectors in a set are Gauss-reduced, it is *pairwise-reduced*.

Algorithm 2 : Gauss reduction for ideal lattices: reduce list u by list v

```
Require: lists $U$, $V$ and number of rotations: $times$
Ensure: reduced list $U$ by list $V$, with all possible rotations
1: for each $u \in U$ do
2:     for each $v \in V$ do
3:         for $i \leftarrow 0$ to $times - 1$ do
4:             $w \leftarrow x^i v$
5:             for $j \leftarrow 0$ to $times - 1$ do
6:                 $(s,t) \leftarrow (x^j u, x^j w)$
7:                 $m \leftarrow \lfloor \langle s,t\rangle / \langle t,t\rangle \rceil$
8:                 if $m \neq 0$ then
9:                     $u \leftarrow s - mt$
10:                    mark $u$ as reduced.
11:                end if
12:            end for
13:        end for
14:    end for
15: end for
```

Algorithm 1 shows the pseudo-code for reducing the list U with the list V. Algorithm 2 is the ideal lattice counterpart.

In Algorithm 2, $times$ represents the number of possible rotations in the input lattice. In other words, $x^{times} = \pm 1$. As concrete examples, for anti-cyclic lattices, $times = n$; for prime cyclotomic lattices, $times = n+1$.

2.3 Prime Cyclotomic Rotation

First we state a nice property of anti-cyclic lattices.

Lemma 2.1 *[25] Let $a,b \in R = \mathbb{Z}[x]/(x^n+1)$ with coefficient vector a,b. If $2\|\langle a, x^l \cdot b\rangle\| \leq \min\{\langle a,a\rangle, \langle b,b\rangle\}$ for all $0 \leq l < n$, then $x^i \cdot a$ and $x^j \cdot b$ are Gauss-reduced for all $i, j \in \mathbb{Z}$.*

In contrast to the anti-cyclic case described in Lemma 2.1, prime cyclotomic lattices do not possess this property. Therefore, all the rotations of two vectors can contribute to the global status. Thus, the list size might be even smaller.

However, prime cyclotomic lattices may have some disadvantages. To illustrate the computational overhead of finding the norms of (all the) rotations of a vector, consider the vector $v = (5,4,3,2,1)$ in an ideal lattice generated by the polynomial $f(x) = x^5 + x^4 + x^3 + x^2 + x + 1$. The first rotation of v is

$$\mathbf{rot}(v) = (-1, 5-1, 4-1, 3-1, 2-1) = (-1,4,3,2,1).$$

Squaring and summing, we have the squared norm for $x \cdot v$:

$$\|\mathbf{rot}(v)\|^2 = (-1)^2 + 4^2 + 3^2 + 2^2 + 1^2 = 31.$$

Calculating norms like this can be slow, because only when the vector $\mathbf{rot}(v)$ is ready can we calculate the sum of squares. However, the value that is required in Gauss reduction is just $\|\mathbf{rot}(v)\|^2$, but not $\mathbf{rot}(v)$ per se. For processors with ADD, MUL and FMAD (fused multiply-add) instructions, it takes $2n$ operations to calculate the norm of an n-dimensional vector.

In this paper, we will see how to circumvent this by *lifting* a vector. By doing this, not only is the computation easier, but it also enables optimizations that are not possible without lifting.

2.4 CUDA Programming

Here we provide a minimalist CUDA programming introduction, including only relevant information that our implementation takes into consideration. For more details, please refer to the CUDA C Programming Guide [30].

Graphics processing units (GPUs) are high throughput, many-core architectures. Currently, the most widely used GPU development toolchain is CUDA by NVIDIA. CUDA supports writing fine-tuned programs for NVIDIA graphic cards. In this paper, we will especially focus on GPUs of the Maxwell architecture.

The CUDA programming model requires programmers to think in the *single instruction, multiple thread* (SIMT) programming model. The model exposes three key abstractions to programmers: a hierarchy of thread groups, shared memories, and barrier synchronization. Threads are first organized in blocks, and blocks are then organized in grids. A grid of GPU threads must run the same program (the kernel).

At the system level, blocks are independently dispatched to different processors. Since each block has a dedicated on-chip cache called the shared memory, threads within a block can only exchange data through the shared memory. However, this requires an explicit synchronization barrier that halts all the threads in a block, and thus can be a huge performance overhead for critical applications.

Fortunately, starting from the Kepler architecture, data exchange within a *warp* can be done using the *warp shuffle* instructions without any explicit synchronization barrier.

A *warp*, consisting of 32 consecutive threads, is the smallest batch that can be scheduled and issued at once by a processor. For example, using the warp shuffle instructions, summing different values from threads with in a warp can be done relatively fast through the *parallel reduction* paradigm. If the threads in a warp are executing different instructions – most likely because of different branch conditions – severe *warp divergence* can occur, drastically lowering the warp utilization.

3. LIFTING IDEAL LATTICES

We now develop the properties for prime cyclic lattices and see how they can facilitate computation.

3.1 Lifting Prime Cyclotomic Polynomials

The idea behind lifting lattices is to supplement vectors with a bit of redundant information to ease computation. Specifically, we will express an n-dimensional vector with an $(n+1)$-dimensional one. Let $\mathscr{L}$ be a lattice generated by $x^n + x^{n-1} + \cdots + 1$, and $\overline{\mathscr{L}}$ by $x^{n+1} - 1$. We wish to seek a way to connect the two lattices according to the following criteria:

- The conversion of vectors between the two lattices is simple.
- The rotation of vectors must be preserving, so that the complicated rotation in $\mathscr{L}$ can be done instead cyclically in $\overline{\mathscr{L}}$.

Technically speaking, we are looking for simple ring homomorphisms between $\mathbb{F}[x]/(x^n + x^{n-1} + \cdots + 1)$ and $\mathbb{F}[x]/(x^{n+1} - 1)$.

An intuitive clue to accomplish this comes from the observation that the polynomial $x^{n+1} - 1$ factorizes as

$$x^{n+1} - 1 = (x-1)(x^n + x^{n-1} + \cdots + 1).$$

This suggests we connect u and its *lift* $\bar{u}$ by thinking of $\bar{u}$ as reduced modulo $x^n + x^{n-1} + \cdots + 1$:

$$u \equiv \bar{u} \pmod{x^n + x^{n-1} + \cdots + 1}.$$

Note that this choice also preserves rotation.

As an example, lifting directly $u = (1,2,3,4,5)$ in a lattice generated by $x^4 + x^3 + x^2 + x + 1$ gives $\bar{u} = (1,2,3,4,5,0)$ in a lattice generated by $x^5 - 1$. This is not the only way to lift u. Another possibility is $\bar{u}' = (2,3,4,5,6,1)$, since $(2-1, 3-1, 4-1, 5-1, 6-1) = (1,2,3,4,5)$. In general, to lift any u, we can choose p arbitrarily and lift u as $\bar{u} = (u_0 + p, u_1 + p, \cdots, u_n + p, p)$.

In the following section, we denote the lifted lattice with a bar on the top of the symbol of the lattice. For example, $\bar{u}$ is a lift of the vector u and $\bar{L}$ is a lift of the lattice L.

3.2 Norms and Inner Products

During the Gauss reduction, we are especially interested in the norms and inner products of rotations of vectors. Let us see how to derive these quantities for the underlying lattice directly, without converting from the lifted lattice.

Suppose $\bar{u} = (\bar{u}_0, \bar{u}_1, \cdots, \bar{u}_{n-1}, \bar{u}_n)$ in $\overline{\mathscr{L}}$. We first reduce $\bar{u}$ modulo the polynomial $x^n + x^{n-1} + \cdots + 1$ to get its underlying form:

$$\begin{aligned} u &= (\bar{u}_0 - \bar{u}_n, \bar{u}_1 - \bar{u}_n, \cdots, \bar{u}_{n-1} - \bar{u}_n) \\ &= (\bar{u}_0 - p, \bar{u}_1 - p, \cdots, \bar{u}_{n-1} - p). \end{aligned}$$

Here, we rewrite $\bar{u}_n$ as p interchangeably, since $\bar{u}_n$ acts as a *pivot* for the vector. We can now calculate the norm of u:

$$\begin{aligned} \langle u, u \rangle^2 &= \sum_{i=0}^{n-1} (\bar{u}_i - \bar{u}_n)^2 \\ &= \sum_{i=0}^{n} (\bar{u}_i - \bar{u}_n)^2 \qquad \text{since } \bar{u}_n - \bar{u}_n = 0. \\ &= \sum_{i=0}^{n} \bar{u}_i^2 - 2\bar{u}_n \sum_{i=0}^{n} \bar{u}_i + (n+1)\bar{u}_n^2 \\ &= \boxed{\langle \bar{u}, \bar{u} \rangle^2} - 2p \boxed{\sum_{i=0}^{n} \bar{u}_i} + (n+1)p^2, \end{aligned}$$

where the boxed terms remain constant throughout all cyclic rotations of $\bar{u}$, and thus can be saved beforehand. Note that we do not need to know what u is at all.

Similarly, the inner product of two vectors u and v is

$$\langle u, v \rangle^2 = \langle \bar{u}, \bar{v} \rangle^2 - p \sum_{i=0}^{n} \bar{v}_i - q \sum_{i=0}^{n} \bar{u}_i + (n+1)pq,$$

where q is the pivot of $\bar{v}$.

3.2.1 Simplifying formulae

Although these formulae may look intimidating, we can always simplify them by choosing the "right" pivot. If we set $\sum_{i=0}^{n} \bar{u}_i = 0$, and solve for p:

$$\begin{aligned} 0 &= \bar{u}_0 + \bar{u}_1 + \cdots + \bar{u}_{n-1} + p \qquad \text{rewrite } \bar{u}_n \text{ as } p \\ &= (u_0 + p) + (u_1 + p) + \cdots + (u_{n-1} + p) + p \\ &= (u_0 + u_1 + \cdots + u_{n-1}) + (n+1)p, \end{aligned}$$

we can choose the pivot as

$$p = -\frac{\sum_{i=0}^{n-1} u_i}{n+1}.$$

This is our standard way to lift a vector.

Algorithm 3 : Gauss reduction between two lists for prime cyclotomic lattices (lifted)

```
Require: Lifted lists $\overline{U}$ and $\overline{V}$
Ensure: Reduced, lifted list $\overline{U}$ by list $\overline{V}$
 1: for each $\bar{u} \in \overline{U}$ do
 2:     for each $\bar{v} \in \overline{V}$ do
 3:         for $i \leftarrow 0$ to $n$ do
 4:             $\bar{w} \leftarrow x^i \bar{v}$
 5:             $\langle \bar{w}, \bar{w} \rangle \leftarrow \langle \bar{v}, \bar{v} \rangle + (n+1)\bar{v}_{n-i}^2$
 6:             for $j \leftarrow 0$ to $n$ do
 7:                 Calculate $\langle \bar{u}, \bar{w} \rangle$.
 8:                 $\langle s, t \rangle \leftarrow \langle \bar{u}, \bar{w} \rangle + (n+1)\bar{u}_{n-j}\bar{w}_{n-j}$
 9:                 $\langle t, t \rangle \leftarrow \langle \bar{w}, \bar{w} \rangle + (n+1)\bar{w}_{n-j}^2$
10:                 $m \leftarrow \lfloor \langle s, t \rangle / \langle t, t \rangle \rceil$
11:                 if $m \neq 0$ then
12:                     $(\bar{s}, \bar{t}) \leftarrow (x^j \bar{u}, x^j \bar{w})$
13:                     $\bar{u} \leftarrow \bar{s} - m\bar{t}$
14:                     Mark $\bar{u}$ as reduced.
15:                 end if
16:             end for
17:         end for
18:     end for
19: end for
```

Carrying out the same procedure for v, we can now write the inner product succinctly:

$$\langle u, v \rangle^2 = \langle \bar{u}, \bar{v} \rangle^2 + (n+1)pq.$$

Lifting in this manner, we amend Algorithm 2 into Algorithm 3. Note that the underlying vectors s and t are no longer needed on line 12. Since we do not have to track and update $\sum u_i$ anymore, simplifying in this manner eases some computational burden and memory overhead in the innermost loop for GPUs. However, integer vectors are now represented by floating points, which may lead to error accumulation after several rounds. We choose to rectify these vectors by unlifting and rounding the numbers when they are taken out from the stack for later rounds. We could also eliminate the $n+1$ by "normalizing" and dividing vectors by $\sqrt{n+1}$, but this is less intuitive.

3.3 Lazy Rotation

We now address two GPU performance bottlenecks in Algorithm 3, and provide two kernel-level heuristics to solve these problems.

First, on lines 12-14, whenever $\bar{u}$ is reduced, it is assigned as the difference of two rotated vectors $\bar{s}$ and $m\bar{t}$. However, such register indexing, unlike on CPUs, can cause spills on GPUs. Since $\bar{s} - m\bar{t} = x^j(\bar{u} - m\bar{w})$, we can instead write $\bar{u} \leftarrow \bar{u} - m\bar{w}$ and choose to rotate $\bar{u}$ back *lazily* after the kernel finishes. Now the lazy version of $\bar{u}$, however, may be representing a vector much longer than it should. To prevent reducing with a lazy $\bar{u}$ in later rounds, we need to keep track of the current correct norm of u. This is done on lines 14–16 in Algorithm 4. Second, because u may have changed in the previous round, $\langle u,w\rangle$ must be recalculated on line 7. To avoid recalculating $\langle u,w\rangle$ repeatedly, observe that the probability of reducing u more than once is not high in the innermost loop. We can keep track of the best m so far, moving the entire if statement on lines 11–14 out and after the for loop.

Algorithm 4 : Gauss reduction between two lists for prime cyclotomic lattices (lifted, with lazy rotation)

```
Require: Lifted lists $\overline{U}$ and $\overline{V}$
Ensure: Reduced, lifted list $\overline{U}$ by list $\overline{V}$
1: for each $\bar{u} \in \overline{U}$ do
2:     $norm \leftarrow \langle \bar{u},\bar{u}\rangle + (n+1)\bar{u}_n^2$
3:     for each $\bar{v} \in \overline{V}$ do
4:         for $i \leftarrow 0$ to $n$ do
5:             $\bar{w} \leftarrow x^i \bar{v}$
6:             $\langle \bar{w},\bar{w}\rangle \leftarrow \langle \bar{v},\bar{v}\rangle + (n+1)\bar{v}_{n-i}^2$
7:             Calculate $\langle \bar{u},\bar{w}\rangle$.
8:             for $j \leftarrow 0$ to $n$ do
9:                 $\langle s,s\rangle \leftarrow \langle \bar{u},\bar{u}\rangle + (n+1)\bar{u}_{n-j}^2$
10:                $\langle s,t\rangle \leftarrow \langle \bar{u},\bar{w}\rangle + (n+1)\bar{u}_{n-j}\bar{w}_{n-j}$
11:                $\langle t,t\rangle \leftarrow \langle \bar{w},\bar{w}\rangle + (n+1)\bar{w}_{n-j}^2$
12:                $m \leftarrow \lfloor \langle s,t\rangle / \langle t,t\rangle \rceil$
13:                $norm_{new} \leftarrow \langle s,s\rangle - 2m\langle s,t\rangle + m^2\langle t,t\rangle$
14:                if $norm_{new} < norm$ then
15:                    $m_{best} \leftarrow m$
16:                    $norm \leftarrow norm_{new}$
17:                end if
18:            end for
19:            if $m_{best} \neq 0$ then
20:                $\bar{u} \leftarrow \bar{u} - m_{best}\bar{w}$
21:                $\langle \bar{u},\bar{u}\rangle \leftarrow \langle \bar{u},\bar{u}\rangle - 2m_{best}\langle \bar{u},\bar{w}\rangle + m_{best}^2\langle \bar{w},\bar{w}\rangle$
22:                Mark $\bar{u}$ as reduced.
23:            end if
24:        end for
25:    end for
26: end for
```

Applying these two heuristics, we now reach Algorithm 4. This amended algorithm is more efficient because (1) the body of the most inner loop runs in constant time, thus reducing the complexity to calculate all inner products of two vectors from $O(n^3)$ to $O(n^2)$, and (2) the need to rotate $\bar{u}$ is completely eliminated.

3.4 Generalizing Lifting

The regularity of terms in the quotient polynomial $f(x)$ plays an important role in the computation of rotations. Consider a cyclotomic polynomial $p(x)$. There might exist another low degree polynomial $r(x)$ such that $p(x)r(x) = x^n \pm 1$. This suggests we promote a vector with dimension $\deg(p(x))$ into dimension $\deg(p(x)) + \deg(r(x))$, thus lowering the computation cost. The same technique of choosing the right pivots can be applied as well.

For example, the next unsolved ideal lattice challenge is dimension 132. One of the ideal lattices in the challenge is generated by the polynomial $f(x) = x^{132} - x^{130} + x^{128} - \cdots + x^4 - x^2 + 1$. Since $(x^2+1)f(x) = x^{134} + 1$, we can convert this lattice into a 134-dimensional anti-cyclic lattice with two pivots, one for the $+1$ terms and the other for -1 terms.

4. APPLICATION TOWARDS CRYPTOSYSTEM CONSTRUCTION

The lifting technique not only accelerates the sieve algorithm, but also can be applied to construct ideal lattice-based cryptosystem. One main design choice in constructing a lattice-based cryptosystem is choosing mathematical structure, *i.e.*, a secure and efficient ideal ring. For example, NTRU uses $f(x) = x^n - 1$ which is very efficient but can not be proven to be secure. In the proved secure scenario, the ring $f(x) = x^n + 1$ where n is power of 2, is the typical one use in [31, 32]. In this paper, we consider the prime-cyclotomic ring $f(x) = x^{n-1} + x^{n-2} + \cdots + 1$ where n is prime, and this had been proved secure under NTRUEnc by Yu *et al.* [33]. The benefit to select over the ring is: for any degree less than n, the number of ring $f(x) = x^{2^m} + 1$ for any $2^m \leq n$ is $\lg n$, but the number of prime cyclotomic is $\Theta(n/\lg n)$. Thus, it is more flexible to choose the dimension for required security level. To compute over this ring, we suggest use our lifting technique and follow the method proposed by Bernstein *et al.* [34]. They proposed a variant of NTRU cryptosystem which is based on the ideal generated by $x^n + x + 1$, and they use combination of Karatsuba and Toom algorithm to reduce the number of multiplications. Lifting also works in for this encryption scheme since the $x^n - 1 = (x-1)(x^{n-1} + x^{n-2} + \cdots + 1)$. After finishing the combined Karatsuba and Toom multiplications in the lifted ring, an extra $(n-1)$ subtractions is applied to the ring element to reduce into the original ring.

5. PARALLELIZATION

Let us now look at our parallel variant of Gauss Sieve for a single machine with multiple GPUs. Two layers of parallelization naturally arise in this setting: the workload should first be split (1) across different GPUs, then (2) to different processors within a GPU. These two layers of architectures differ in communication cost. Broadcasting data from the host memory across all the GPUs through PCIe is much more expensive than broadcasting data from the on-chip memory to different processors within a single GPU.

We carefully design these two layers in hope to mitigate communication overhead. Specifically, we view each GPU as an independent sieve (inner layer), and all the GPUs cooperate as a complete parallel sieve (outer layer). In the following subsections, we will see (1) how the problem is divided into independent sub-sieves on different GPUs, so that each sub-sieve acts as a blackbox, ordinary Gauss Sieve, and (2) how the sub-sieve is designed to maximize GPU power.

5.1 Outer Layer

To distribute the work among the GPUs on a single machine, we first recall the work by Bos *et al*. [25], which was originally designed for computer clusters. In their work, each node acts as an independent Gauss Sieve, maintaining its own local list while reducing the same batch of samples broadcast over all the nodes. These nodes communicate only at the end of each iteration, putting any sample that is ever reduced in any of the nodes to the stack. The advantage of this approach is that the long, local lists are never completely moved out of the nodes; only a limited amount of reduced vectors and samples are involved in communication. Communication cost is thus small. Here, we adapt their method to a machine with multiple GPUs using the following analogy: A cluster is to the machine what a node is to the GPU. As a result, a GPU now works as if it were a node, having its own local list, and communication is done on the host. At each iteration, all the GPUs are given the same batch of samples, either newly generated or from the stack. Each GPU then first reduces its samples mutually with its local list, using the method described in Subsection 5.2. Next, for each sample, if ever reduced in one GPU, the host compares and chooses the shortest "representative", putting it to the stack. Reduced vectors from local lists are also put on the stack. Last, the "surviving" samples are appended to the shortest local list. The vectors on the stack become the input for later rounds.

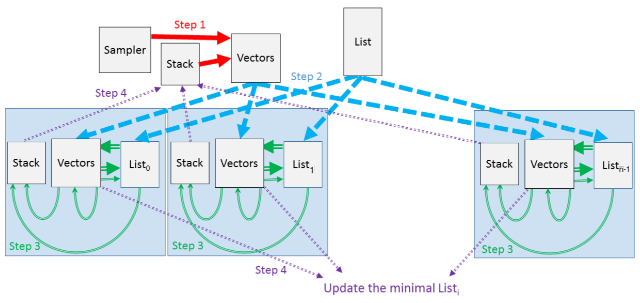


Fig. 1. Illustration of the outer layer parallel: Step 1, the vectors are sampled from the Sampler or the Stack; Step 2, copy the vectors to each GPU and divide the List into n partitions, each partition is store in a GPU. Step 3, reduces the vectors mutually with its local list, this described in next subsection; Step 4, gather the partitions of List and update the minimal list.

5.2 Inner Layer

The inner layer is a modified version of Ishiguro *et al*.'s idea. As mentioned in the previous subsection, each GPU can be thought of as an independent sieve, reducing its local list with a batch of samples. First, the local list is reduced with the samples. Then, such samples are mutually reduced with each other. Finally, the samples are reduced with the local list. If any vector is ever reduced during any step, it is marked and later collected on the stack. As showed by Ishiguro *et al*., any pair of surviving vectors in the local list remains reduced during the process.

Since these three steps share the same pattern – they all reduce one list with another, the same GPU kernel can be used. See Algorithm 1 for general lattices and Algorithm 2 for ideal lattices. To reduce list A with list B, the kernel takes as inputs list A and list B, and in-place outputs the reduced list A. In the kernel, list A is sliced into adequate chunks and distributed to different processors, while list B is broadcast to all processors. The kernel is crafted with care to ensure high throughput, as will be described in the next section.

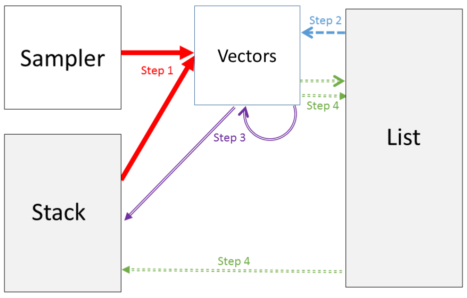


Fig. 2. Illustration of the inner layer parallel: Step 1: the vectors are sampled from the Sampler or the Stack. Note that, this step is done by outer layer of parallelization of our framework. Step 2: reduce the vectors by the vector in the List. Step 3: reduce the vectors mutually by themselves and move the vector into stack if it is reduces; Step 4: reduce the vectors in the List by vectors, and move the vector into stack if it is reduces.

6. IMPLEMENTATION

In this section, we will first see how common performance tuning techniques can be applied to our algorithm. This includes thread- and instruction-level parallelism. Next, we point out more kernel optimization tricks. Finally, we describe two more heuristics that can significantly improve the execution time.

6.1 Vector Layout

On GPUs, each thread has a physical register number limit; depending on how many resources each thread requires, each processor also has a runtime limit for thread numbers. For example, consider the kernel for $n = 100$. On a Maxwell GPU, each thread can use up to 255 registers. If we put both $\bar{u}$ and $\bar{w}$ in one thread, we need $2 \times (100+1) = 202$ registers. Although fewer than 255, this is still so much that the processors can only schedule a few threads, limiting thread-level parallelism. At the other extreme, if we spread a vector across too many threads, the overhead of parallel reduction to calculate inner products collectively will take over.

Empirically, we choose to spread a vector across 4 threads. For our target dimension 130, this means each thread takes $\lceil 130/4 \rceil = 33$ elements, with the extra two elements padded with 0. This choice not only reduces register pressure, but also makes the vector length a multiple of 4, which is essential for cache line alignment. To this end, parallel reductions are needed both on line 7 to collectively sum inner products, and before line 17 (after the for loop) to agree on the best m. To make the code more readable, we use the CUB library [35] for block load and store in the kernel.

6.2 Instruction-Level Parallelism

Yet another commonly applied trick to increase GPU utilization is to exploit instruction-level parallelism. The idea is to issue independent instructions at once to increase the pipeline usage. However, since the algorithm is very much inherently dependent from line to line, the direct implementation will run very slowly. We do not overlap two independent copies of kernel at the same time, because the register usage is immediately multiplied by two. Instead, we unroll the loop on line 4 with an empirical factor of 8 to facilitate register reuse on line 7. This technique is possible only if the lattice is lifted, since a lattice point is represented in its cyclic form.

Next, we identify two new heuristics due to loop unrolling. First, at the end of each 8th iteration, we choose the best m_{best} from eight possible m_{best}'s. Second, since the prime n is never a multiple of 8, empirically we just omit the remainder of the unrolled loop.

6.3 More Kernel Optimizations

- The rotation on line 5 is tricky because vectors are padded with zero. Therefore, the last thread that contains a vector would have to deal with these zeros. Actually, the padded vector $\bar{v}$ is first stored in the shared memory, then rotated one by one at each iteration. More specifically, at the end of one iteration, the first padded zero is replaced with the next "right" element, and at the start of the next iteration, the vector $\bar{w}$ is read at the "right" offset.
- The vectors in the lists $\overline{U}$ and $\overline{V}$ are loaded in bulks and put in a shared-memory buffer to increase global memory throughput.
- In practice, we choose the first element of a lifted vector as its pivot and rotate reversely. This transforms the index $n - j$ on lines 9–10 to an easier j.
- To ensure high kernel throughput, we empirically tune all the parameters mentioned in the above sections as well as kernel launch parameters, although it is not feasible to try all possible combinations.

6.4 On Faster Convolution

The question naturally arises: why not use FFT or the Karatsuba algorithm to calculate inner products? The reasons are:

- The Karatsuba algorithm reduces the number of multiplications, while adding a lot more additions. On GPUs, however, FMAD is fastest.
- If on line 5, we use them to produce results for all i's simultaneously, there will not be enough register to hold both the results and all the intermediate values during computation.
- The dimension is not a power of 2, which makes the convolution more difficult to be designed efficiently.

There are several techniques to convert non-power-of-2 DFT's into convolutions or FFTs of the same or larger dimensions, *e.g.* Cooley–Tukey FFT algorithm. The best approach we are aware of is Devil's convolution [36], but this is not easily applicable on GPUs.

6.5 Heuristics

Besides the heuristics for kernel optimization, we also applied two heuristics to speed up in conjunction with the techniques above.

First, as already mentioned in Voulgaris' implementation [37], the lists in Step 1 are sorted so that only longer vectors (before rotation) are reduced with shorter ones. We also tried to see if this can be applied to Steps 2 and 3. Empirically we found that it is not as effective, probably because vectors are shorter during Steps 2 and 3; they are less likely to be reduced. We also use the CUB library to sort data on GPUs. Second, empirically we choose to iterate the innermost loop over only the first 16 values of j (line 8). This is because the rotations of prime cyclotomic vectors have larger norms. The expansion factor for prime cyclotomic lattices is discussed in [11].

7. RESULTS

For our experiments, we use a total of eight NVIDIA GeForce GTX TITAN X graphics cards. Four of these cards are installed on the main machine, while the other four are installed on a PCIe extension box.

7.1 Parallel Efficiency

We use the bases from the Ideal Lattice Challenge [26]. Since for dimension n, the prime cyclotomic polynomial has index $n+1$, as an example, we choose the basis for dimension 126 from the file `ideallatticedim126index127seed0.txt`. The input bases for Gauss Sieve are first reduced by BKZ with block size 30 and $\delta = 0.99$.

Here we show the parallel efficiency of the outer layer of our parallel framework in Fig. 3. The *(parallel) efficiency* for N GPUs is defined in [25] as

$$E = \frac{\text{runtime for } N \text{ GPUs}}{N \cdot \text{runtime for 1 GPU}}.$$

For the dimension 108, the efficiency is $74\%, 72\%, 55\%$ and 45% for 2, 4, 6 and 8 GPUs, respectively. However, the dimension is so low that the efficiency is quite low as the number of GPUs exceeds 6.

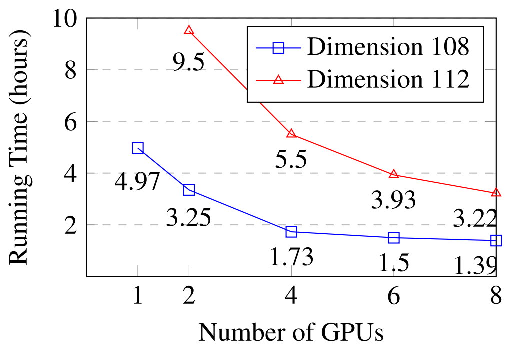


Fig. 3. Parallel Efficiency: the numbers of GPUs versus running time and the exact running time is labelled below the node.

On the other hand, for dimension 112, the efficiency scales better with the number of GPUs. However, we do not yet have the running time for one GPU. If we base the efficiency on 2 GPUs, then the efficiency is the 86%, 81% and 74% for 4, 6 and 8 GPUs, respectively. We believe that in high enough dimensions, the efficiency of 8 GPUs will be more than 70%.

7.2 Ideal Lattices versus General Lattices

For general lattices, we use the bases from the SVP Challenge [38]. Our single-GPU implementation takes 9.3 hours to solve the challenge of dimension 96. In contrast, the implementation from [24] requires 200 CPU-hour. That is, our single-GPU implementation on general lattices is 21.5 times faster than the CPU version.

For ideal lattices, our 4-GPU implementation requires 5 minutes to solve the challenge of dimension 96 and our single-GPU implementation requires 8.6 minutes. In contrast, the implementation from [24] requires 8 CPU-hours. That is, our 4-GPU (resp. single-GPU) implementation on general lattices is 96 (resp. 55.8) times faster than the CPU version. Note that the polynomial we use is prime-cyclotomic ($x^{96}+x^{95}+\cdots+1$), which is more complicated than the trinomial polynomial ($x^{96}+x^{48}+1$) used by [24].

Combining these two cases, the speed-up from using the property of prime-cyclotomic ideal lattices is $\dfrac{9.3\text{ hrs}}{8.6\text{ mins}} = 64.9$ in dimension 96. Applying the complexity estimation from [18], we estimate the ratio to be $\dfrac{9.3\text{ hrs}\cdot 2^{0.52\cdot 30}}{2734\text{ hrs}} = 169$ in dimension 126 and $\dfrac{9.3\text{ hrs}\cdot 2^{0.52\cdot 34}}{6583\text{ hrs}} = 297$ in dimension 130.

In contrast, [24] shows that the speed-up ratio of using the property of anti-cyclic ideal lattices is around 600 in dimension 128. This gives an evidence that the SVP over prime-cyclotomic ideal lattices is harder than over anti-cyclic ideal lattices by a factor of around 2.

7.3 Chronological Behavior

The chronological behavior of the sieving algorithms is studied intensively in [18, 39]. We can observe the same behavior in Fig. 4 (a) shows that the size of the list grows

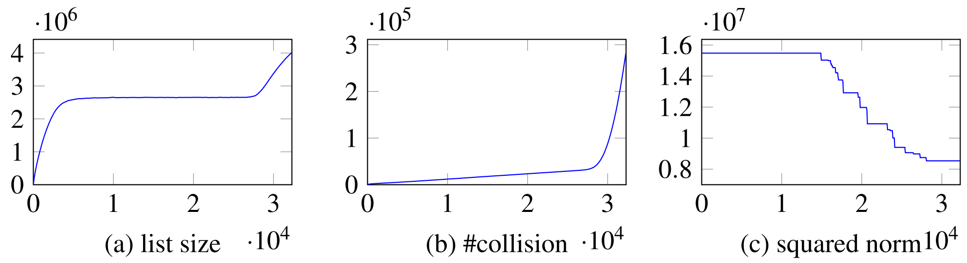


Fig. 4. Behavior of Gauss Sieve for dimension 126 with 8 GPUs: (a) the list size versus iteration; (b) the number of collisions versus iteration; (c) the squared norm of current shortest vector versus iteration.

very fast in the beginning, later on it reaches a plateau, and finally when the shortest vector is found, it grows rapidly again; (b) shows the number of collision grows almost linearly but goes up very fast after the shortest vector is found; (c) shows the squared norm of the current shortest vector. The norm starts to drop half-way, and keeps descending until the shortest vector is found. One possible improvement is to reduce the basis by the current founded short vector, as in the work [40]. However, since the very first "shorter" vector only shows up half-way, the speed-up ratio by this method is limited by 2.

Our result in Table 3 is the fastest implementation of the Gauss Sieve algorithm so far. A rough space usage estimation is $2 \times 4 \times \mathit{ListSize} \times \mathit{Dimension}$. The factor 4 is due to the data type, 4-byte float, and the 2 is due to an extra buffer for sorting on the device. Therefore, it requires around 0.37, 2.59 and 4.35 GB of memory for dimension 112, 126 and 130, respectively.

Table 3. Results of ideal lattice challenge.

Dimension	112	126	130
Number of vectors	444,341	2,759,903	4,490,083
Running Time (GPU-hours)	32	2,734	6,583

7.4 Hardness Estimation

Finally, Fig. 5 compares our results with previous works. Obviously, our results are below the estimation of [41]. Even more, the slope of ours is flatter than theirs, which means that there is an exponential speed-up. Some of the data from the SVP and Ideal SVP Challenge is computed using an accelerated random sampling algorithm [40], but in higher dimensions (say, higher than 136), our fitting curve is also below their results. This might imply that the running time of the Gauss Sieve algorithm grows quite slowly.

Fitting our data using least-square regression, we have $y = 2^{0.435x-31.8}$, as depicted in Fig. 5. To resist attacks using super powerful special-purpose hardware, our conservative model of SVP hardness in ideal lattices, with approximation factor 1.05, is

$$time(SVP^{Ideal}_{\alpha=1.05}) = 2^{0.43n-50}(\text{seconds}).$$

However, we emphasize that the space complexity of the sieve algorithm is exponential, but estimation models of [41] and [42] are based on BKZ or BKZ 2.0, which requires

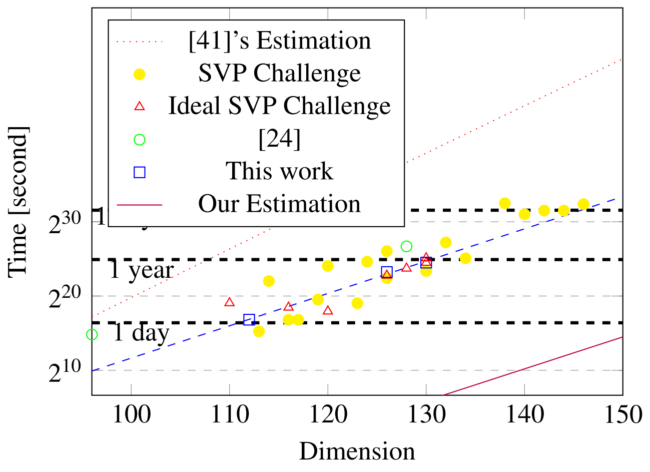


Fig. 5. Comparison of time: the dimension of SVP challenge versus the running time.

only polynomial space. More precisely, our implementation requires $2^{0.19n+7.3}$ bytes of memory.

8. CONCLUSION

In this work, we propose the lifting technique for prime-cyclotomic ideal lattices, which accelerates the Gauss Sieve algorithm. Moreover, by applying a sequence of transformations described in Section 3, the cost of reducing two vectors with all of their rotations decreases from $O(n^3)$ to $O(n^2)$. We also designed and implemented Gauss Sieve that includes these technique both on a single GPU and on multiple GPUs. Our implementation is more than 21.5 (resp. 55.8) times faster than the best prior known result on a single CPU core for general (resp. ideal) lattice. Finally, we give a reasonable model to estimate the running time of solving SVP in ideal lattices. Although our model requires an exponential space due to the nature of sieving algorithms, it suggests a bound much lower than that provided by the previous model [41].

REFERENCES

1. C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, 2009, pp. 169-178.
2. M. Ajtai and C. Dwork, "A public-key cryptosystem with worst-case/average-case equivalence," in *Proceedings of Annual ACM Symposium on Theory of Computing*, 1997, pp. 284-293.
3. R. E. Bansarkhani, D. Cabarcas, P.-C. Kuo, P. Schmidt, and M. Schneider, "A selection of recent lattice-based signature and encryption schemes," *Tatra Mountains Mathematical Publications*, Vol. 53, 2012, pp. 81-102.

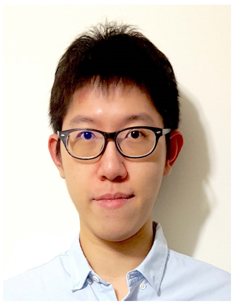
4. D. Micciancio and P. Voulgaris, "A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations," in *Proceedings of the 42nd ACM Symposium on Theory of Computing*, 2010, pp. 351-358.
5. T. Laarhoven, "Finding closest lattice vectors using approximate voronoi cells," Cryptology ePrint Archive, 2016, Report 2016/888, https://eprint.iacr.org/2016/888.
6. N. Gama, P. Q. Nguyen, and O. Regev, "Lattice enumeration using extreme pruning," in *Proceedings of EUROCRYPT*, 2010, pp. 257-278.
7. P. Kuo, M. Schneider, Ö. Dagdelen, J. Reichelt, J. A. Buchmann, C. Cheng, and B. Yang, "Extreme enumeration on GPU and in clouds – how many dollars you need to break SVP challenges," in *Proceedings of the 13th International Workshop on Cryptographic Hardware and Embedded Systems*, 2011, pp. 176-191.
8. P.-C. Kuo and C.-M. Cheng, "Lattice-based cryptanalysis-how to estimate the security parameter of lattice-based cryptosystem," in *Proceedings of IEEE International Conference on Consumer Electronics*, 2014, pp. 53-54.
9. M. Ajtai, R. Kumar, and D. Sivakumar, "A sieve algorithm for the shortest lattice vector problem," in *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, 2001, pp. 601-610.
10. M. R. Albrecht, B. R. Curtis, A. Deo, A. Davidson, R. Player, E. Postlethwaite, F. Virdia, and T. Wunderer, "Estimate all the LWE, NTRU schemes," https://estimate-all-the-lwe-ntru-schemes.github.io, 2018.
11. M. Schneider, "Sieving for shortest vectors in ideal lattices," in *Proceedings of AFRICACRYPT*, 2013, pp. 375-391.
12. T. Laarhoven, "Sieving for shortest vectors in lattices using angular locality-sensitive hashing," in *Proceedings of the 35th Annual Cryptology Conference*, Part 1, 2015, pp. 3-22.
13. A. Becker, L. Ducas, N. Gama, and T. Laarhoven, "New directions in nearest neighbor searching with applications to lattice sieving," in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2016, pp. 10-24.
14. A. Becker and T. Laarhoven, "Efficient (ideal) lattice sieving using cross-polytope LSH," in *Proceedings of AFRICACRYPT*, 2016, pp. 3-23.
15. T. Laarhoven and B. de Weger, "Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing," in *Proceedings of the 4th International Conference on Cryptology and Information Security*, 2015, pp. 101-118.
16. P. Q. Nguyen and T. Vidick, "Sieve algorithms for the shortest vector problem are practical," *Journal of Mathematical Cryptology*, Vol. 2, 2008, pp. 181-207.
17. G. Hanrot, X. Pujol, and D. Stehlé, "Algorithms for the shortest and closest lattice vector problems," in *Proceedings of the 3rd International Workshop on Coding and Cryptology*, 2011, pp. 159-190.
18. D. Micciancio and P. Voulgaris, "Faster exponential time algorithms for the shortest vector problem," in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2010, pp. 1468-1480.
19. X. Pujol and D. Stehle, "Solving the shortest lattice vector problem in time $2^{2.465n}$," Cryptology ePrint Archive, Report 2009/605, 2009, http://eprint.iacr.org/.
20. T. Laarhoven, M. Mosca, and J. van de Pol, "Finding shortest lattice vectors faster using quantum search," *Designs, Codes and Cryptography*, Vol. 77, 2015, pp. 375-400.

21. F. Zhang, Y. Pan, and G. Hu, “A three-level sieve algorithm for the shortest vector problem,” in *Proceedings of the 20th International Conference on Selected Areas in Cryptography*, 2013, pp. 29-47.
22. X. Wang, M. Liu, C. Tian, and J. Bi, “Improved nguyen-vidick heuristic sieve algorithm for shortest vector problem,” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011, pp. 1-9.
23. A. Becker, N. Gama, and A. Joux, “Solving shortest and closest vector problems: The decomposition approach,” *IACR Cryptology ePrint Archive*, Vol. 2013, 2013, p. 685.
24. 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 IACR International Conference on Practice and Theory of Public-Key Cryptography*, 2014, pp. 411-428.
25. J. W. Bos, M. Naehrig, and J. van de Pol, “Sieving for shortest vectors in ideal lattices: a practical perspective,” *IACR Cryptology ePrint Archive*, Vol. 2014, 2014, p. 880.
26. “Idea Lattice challenge,” http://www.latticechallenge.org/ideallattice-challenge/.
27. A. Mariano, C. H. Bischof, and T. Laarhoven, “Parallel (probable) lock-free hash sieve: A practical sieving algorithm for the SVP,” in *Proceedings of the 44th International Conference on Parallel Processing*, 2015, pp. 590-599.
28. A. Mariano and C. H. Bischof, “Enhancing the scalability and memory usage of hashsieve on multi-core cpus,” in *Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2016, pp. 545-552.
29. M. Ajtai, “The shortest vector problem in l_2 is np-hard for randomized reductions,” *Electronic Colloquium on Computational Complexity*, Vol. 4, 1997, No. TR97-047.
30. “CUDA C programming guide 7.5,” http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2015.
31. E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, “Post-quantum key exchange - A new hope,” in *Proceedings of the 25th USENIX Security Symposium*, 2016, pp. 327-343.
32. J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, and D. Stehlé, “CRYSTALS - kyber: a cca-secure module-lattice-based KEM,” *IACR Cryptology ePrint Archive*, Vol. 2017, 2017, p. 634.
33. Y. Yu, G. Xu, and X. Wang, “Provably secure NTRU instances over prime cyclotomic rings,” in *Proceedings of the 20th International Conference on Practice and Theory in Public-Key Cryptography*, Part 1, 2017, pp. 409-434.
34. D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal, “NTRU prime,” *IACR Cryptology ePrint Archive*, Vol. 2016, 2016, p. 461.
35. D. N. C. Merrill, “The CUB Library,” http://nvlabs.github.io/cub/.
36. R. E. Crandall, *Topics in Advanced Scientific Computation*, 1996, p. 340.
37. P. Voulgaris, “Gauss sieve implementation by panagiotis voulgaris,” https://cseweb.ucsd.edu/pvoulgar/impl.html.
38. “Lattice challenge,” http://www.latticechallenge.org/svp-challenge/.
39. M. Schneider, “Analysis of gauss-sieve for solving the shortest vector problem in lattices,” in *Proceedings of the 5th International Workshop on Algorithms and Computation*, 2011, pp. 89-97.

40. M. Fukase and K. Kashiwabara, "An accelerated algorithm for solving SVP based on statistical analysis," *Journal of Information Processing*, Vol. 23, 2015, pp. 67-80.
41. R. Lindner and C. Peikert, "Better key sizes (and attacks) for lwe-based encryption," in *Proceedings of the Cryptographers' Track at the RSA Conference*, 2011, pp. 319-339.
42. 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*, 2011, pp. 1-20.

Po-Chun Kuo received his Ph.D., MS and BS degrees in Electrical Engineering from National Taiwan University in 2020, 2011 and 2010, respectively. He is currently the owner of Byzantine Lab in Taiwan and was a Research Assistant in Academia Sinica in Taiwan. His research interests include cryptography, high-performance computing, blockchain and graph theory. His main research activities focus on the lattice-based cryptography and consensus algorithm.

Chen-Mou Cheng received his BS and MS in Electrical Engineering from National Taiwan University in 1996 and 1998, respectively, and his Ph.D. in Computer Science from Harvard University in 2007. He joined the Department of Electrical Engineering of National Taiwan University in 2007, where he is currently an Associate Professor. His main research area is in cryptographic hardware and embedded systems (CHES), as well as electronic system-level (ESL) design. Currently, his main research activities focus on the design and analysis of efficient algorithms to solve several important problems arising from cryptology, as well as the development and implementation of these algorithms on massively parallel computers. These problems include solving systems of polynomial equations over finite fields, integer factorization, elliptic-curve discrete logarithm, and lattice reduction.



Wen-Ding Li received his BS and MS degrees in Electrical Engineering from National Taiwan University. He is now a Research Assistant at Research Center for Information Technology Innovation, Academia Sinica, Taiwan. His research interests include high performance computing, symbolic computation and cryptography.

Bo-Yin Yang received his Ph.D. in Mathematics from Massachusetts Institute of Technology in 1991. After teaching mathematics at Tamkang University in Taiwan, he started working with cryptography in 2002. Eventually moved to the Institute of Information Science at Academia Sinica in 2006. He is known for his work on efficient crypto implementations, algebraic cryptanalysis, and post-quantum public-key cryptography.