Sieving for Shortest Vectors in Ideal Lattices: a Practical Perspective

Joppe W. Bos[1], Michael Naehrig[2], and Joop van de Pol[3⋆]

[1] NXP Semiconductors, Leuven, Belgium
joppe.bos@nxp.com
[2] Microsoft Research, Redmond, USA
mnaehrig@microsoft.com
[3] Dept. Computer Science, University of Bristol, United Kingdom
joop.vandepol@bristol.ac.uk

Abstract. Lattice-based cryptography is a promising candidate for providing cryptographic functions that remain secure in a post-quantum era. The security of lattice-based schemes relies on the hardness of lattice problems such as the problem of finding short vectors in integral lattices. In this work, we propose a new variant of the parallel Gauss sieve algorithm which can compute such short vectors. Our algorithm combines favorable properties of previous approaches: all vectors in the global list remain pairwise Gauss reduced (reducing the list size and runtime) while this list is split up between the participating nodes (lowering the memory requirements per node). To demonstrate the benefits of this variant, we present an optimized implementation of our parallel algorithm, using commonly available vector instruction set extensions. We conducted experiments with lattices of dimensions 80, 88, and 96, and the results show that the new approach outperforms the previous parallel Gauss sieve algorithms. The implementation will be made available, and we hope that it will serve as a basis for additional experiments.
Almost all recent implementations of lattice-based cryptographic schemes use ideal lattices, and it is known that sieving algorithms can benefit from their additional algebraic structure. Namely, the list size can be reduced by considering vectors along with all their rotations. We show that ideal lattices allow more optimizations, which enhance the performance of sieving algorithms even further. We use the fast Fourier transform (FFT) to speed up the computation of inner products between a vector and the rotations of another. On a conventional, academic computer cluster, our algorithm solved an SVP ideal lattice challenge in a negacyclic ideal lattice of dimension 128 in less than nine days using 1024 cores. This is more than twice as fast as the recent computation by Ishiguro et al. that solved the same challenge on a cluster with the same computer architecture, but without using large shared memory. This indicates that the FFT version can lead to a practical performance increase compared to a naive version using only rotations. Our results shed additional light on the security of schemes which rely on the hardness of computing short vectors in this special setting.

1 Introduction

Since the late 1990s, there has been increasing interest in constructing cryptographic functions with security based on the computational hardness of lattice problems, initiated by works such as [1, 2, 25]. *Lattice-based cryptography* has become a promising source of cryptographic primitives for several reasons. One of them is its post-quantum potential, i.e. it enables to build efficient core cryptographic primitives that are believed to be secure against quantum computers (cf. the survey [36]). Another reason is that it allows to build primitives with new capabilities that are not possible with classical schemes: this includes fully homomorphic encryption [19] and multilinear maps [17].

⋆ Part of this work was done while the third author was an intern in the Cryptography Research group at Microsoft Research.

Many of the most recent lattice-based systems reduce their security to the hardness of the learning with errors (LWE) problem [46] or its ring variant, the ring-learning with errors (R-LWE) problem [33]. The latter provides an additional algebraic ring structure and is often preferred for efficiency, which is of particular importance for constructing (fully) homomorphic encryption schemes [9, 8]. In the R-LWE framework, the lattices that arise are ideal lattices corresponding to ideals in a polynomial ring. More precisely, they are ideals in the ring of integers of a cyclotomic number field. Arithmetic in such a ring is performed as arithmetic in $\mathbb{Z}[X]/(\Phi_m(X))$ where $\Phi_m(X)$ is the m-th cyclotomic polynomial of degree $n = \varphi(m)$. A popular choice is to use $m = 2^k$, for a non-zero integer k, since then $n = 2^{k-1}$ and $\Phi_{2^k}(X) = X^{2^{k-1}} + 1$. This polynomial is maximally sparse and polynomial arithmetic can be performed efficiently using the fast Fourier transform [32]. Many lattice-based constructions (e.g. [10, 50, 9, 8]) and most implementations (e.g. [32, 22, 23, 5, 42, 7, 8]) use this particular instance of a cyclotomic polynomial. Often, however, schemes are formulated in the more general setting and, in particular for implementations of homomorphic encryption, it can be useful to work with m not being a power of two [20]. See [34] for an overview and discussion of how to work in arbitrary cyclotomic rings.

The hardness of the R-LWE problem is related to the difficulty of finding a sufficiently short non-zero vector in a lattice associated to an ideal in a cyclotomic ring. For example, the distinguishing attack presented in [36] requires a short vector of a certain length to achieve a given distinguishing advantage. Hence, the security parameters of R-LWE-based cryptographic schemes are chosen such that polynomial run-time exponential-approximation algorithms such as the LLL algorithm [30] are unlikely to find such short vectors (cf. [51, 31]). The known deterministic algorithms which are capable of finding them, all have exponential running time. Typically, one distinguishes three types of algorithms to compute short vectors in lattices: Voronoi-cell based [37], enumeration [24, 43, 16], and probabilistic sieving algorithms (cf. Section 2.1 for a brief historical overview of sieving methods). In practice, the enumeration algorithms equipped with a technique called extreme pruning seem to perform best. However, these algorithms work by manipulating the Gram-Schmidt orthogonalization, which does not retain the additional structure of ideal lattices arising from cyclotomic rings due to projection. This prevents such algorithms from taking advantage of the additional structure. As sieving methods work with actual lattice vectors, they seem to be the only type of algorithm that can take advantage of this structure in order to find short vectors.

1.1 Contributions

In this paper we take a closer look at the Gauss sieve algorithm from a *practical* point of view. Our contributions are two-fold. Firstly, we propose yet another variant of the Gauss sieve algorithm which is inspired by the ideas from both the approach by Milde and Schneider [38] and Ishiguro et al. [26]. We keep the concept adopted by Milde and Schneider to split up the global list of vectors in the Gauss sieve across all compute instances and do not use the method of Ishiguro et al. to have all instances maintain a full copy of the global list. This evenly distributes the required storage space for the list vectors among the instances instead of duplicating it. We deviate from the Milde-Schneider approach how sampled and reduced vectors are treated. Instead of sampling vectors locally and propagating vectors through the system in a circular fashion, we sample new vectors and collect reduced vectors globally, to feed them into the system by broadcasting a batch of the same vectors to all computational units. This approach is closer to the one chosen by Ishiguro et al. and allows to ensure that

arbitrary pairs of vectors from the list are Gauss reduced after each round of computation; this lowers the global maximum list size and therefore the expected total runtime of the algorithm. It requires the computation rounds to be synchronized and all units to agree on a set of vectors that are added to the global list (in the form of adding them locally to the smallest list instance).

Secondly, we extend the approach taken by Schneider [47] and Ishiguro et al. [26] to use the additional ideal lattice structure of computing in the special setting of the ring $\mathbb{Z}[X]/(X^n+1)$ in our advantage. As done previously, we use the fact that one can represent the n rotations $X^i \cdot \mathbf{a}$ (of the same norm as $\mathbf{a}$) by only storing a single vector $\mathbf{a}$. This provides a factor n reduction in the storage requirement for list vectors for the same list size. Beyond that, we improve the efficiency of computing scalar products with these rotations, which are used to check the condition for two vectors to be Gauss-reduced. In Section 4, we show how all scalar products of one vector with all these n rotations of a second vector can be computed efficiently at once using the fast Fourier transform (FFT) (cf. [32] for the application in the cryptographic setting). This approach provides an improvement from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \ln n)$ operations over the naive approach of computing all scalar products separately. We also show that comparing the absolute values of n such scalar products to the square of the norms of the two original vectors is sufficient to decide whether all n^2 pairs of rotations of the first vector with rotations of the second vector are Gauss reduced or not. This decreases the number of comparisons from n^2 (when naively comparing all scalar products) to n. In Section 4.4 we point out that although our main target is the ring $\mathbb{Z}[X]/(X^n+1)$, our techniques also apply to other special rings such as $\mathbb{Z}[X]/(X^n-1)$ as used in the NTRU cryptosystem [25].

We have implemented our parallel Gauss sieve variant for general (non-ideal) lattices with additional stages of optimizations for negacyclic ideal lattices. The first stage uses vector rotations as in previous approaches and the second stage deploys all optimizations mentioned above, including using the FFT to compute scalar products. Our approach is suitable to be computed on single instruction, multiple data (SIMD) platforms and we have optimized our implementation by using commonly available vector instruction set extensions. The source code of our implementations will be made available and we hope that it will serve as a basis for other people to conduct more experiments.

The practical benefits of our parallel Gauss sieve variant are demonstrated by computing short vectors for generic lattices of dimensions 80, 88, and 96, using a variable number of nodes, clearly indicating the communication overhead. In the ideal lattice setting, we show the practical benefits of working in $\mathbb{Z}[X]/(X^{2^7}+1)$ of dimension 128 by computing the ring multiplication using Nussbaumer's symbolic FFT approach [40]. Again, this part of our implementation is tailored to computer architectures supporting vector instructions and we are able to speed up our implementation by a constant factor by computing the rotations, inner-products and the FFT algorithm using streaming SIMD extension (SSE) instructions.

The experiments with our parallel Gauss sieve algorithm (see Section 5) indicate that the implementation of the approach presented in this paper systematically outperforms the approach and implementation presented by Ishiguro et al. [26] when targeting both regular (non-ideal) and special (ideal) lattices. Besides this speed-up when using our new approach, our results also indicate that one can expect a speed-up of over a factor 25 when using the ideal lattice structure in dimension 128 and applying a parallel Gauss sieve algorithm which uses rotations. However, when using the FFT techniques presented in Section 4, an additional speed-up factor of at least 3.4 can be expected when computing short vectors in dimension 128. This shows that finding short vectors in ideal lattices can be done over 88 times faster

than finding short vectors in generic lattices: an implication, together with the significant reduction of the list size when solving the shortest vector problem in the ideal lattice setting, which should be taken into account when selecting parameters for cryptographic schemes for which security relies on the hardness of finding short vectors.

2 Preliminaries

Consider the Euclidean space $\mathbb{R}^n$ with its usual topology. We denote column vectors by bold letters, the inner product of two vectors by $\langle \mathbf{x}, \mathbf{y} \rangle$ and the corresponding Euclidean norm by $\|\mathbf{x}\| = \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle}$. For real numbers r, we define $\lfloor r \rceil = \lfloor r + \frac{1}{2} \rfloor$ to be the integer closest to r (rounded up if not unique).

A *lattice* L is a discrete subgroup of $\mathbb{R}^n$. We only consider full-rank, integral lattices $L \subset \mathbb{Z}^n$, represented by a basis $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$. The lattice $L = \mathcal{L}(B)$ consists of all linear combinations with integral coefficents of the basis vectors, i.e., $\mathcal{L}(B) = \{\sum_{i=1}^{n} \lambda_i \mathbf{b}_i : \lambda_i \in \mathbb{Z}\}$. We denote by $\lambda_1(L)$ the length of a shortest non-zero vector in L, i.e., $\lambda_1(L) = \min_{\mathbf{x} \in L \setminus \{\mathbf{0}\}} \|\mathbf{x}\|$.

The *Shortest vector problem (SVP)* is defined as follows: given a basis B of a lattice $L = \mathcal{L}(B)$, find a vector $\mathbf{v}$ in L such that $\|\mathbf{v}\| = \lambda_1(L)$. Generally there are two different kinds of algorithms to tackle this problem, approximation algorithms and exact algorithms. Exact algorithms run in exponential time, but provide an exact solution, whereas approximation algorithms generally run in polynomial time, but only provide some exponential approximation to the solution. These algorithms often complement each other, in the sense that an approximation algorithm is used to pre-process the lattice basis as preparation for the exact algorithm. Vice versa, exact algorithms are used as subroutines for the approximation algorithms, where they solve SVP instances in lower dimensional lattices.

A central notion for describing the Gauss sieve is that of Lagrange or Gauss reduction. Two vectors $\mathbf{x}$ and $\mathbf{y}$ are called *Gauss reduced* if $2 \cdot |\langle \mathbf{x}, \mathbf{y} \rangle| \leq \min\{\langle \mathbf{x}, \mathbf{x} \rangle, \langle \mathbf{y}, \mathbf{y} \rangle\}$. Equivalently, this means that $\|\mathbf{x} \pm \mathbf{y}\| \geq \max\{\|\mathbf{x}\|, \|\mathbf{y}\|\}$ and hence that the two-dimensional lattice spanned by the basis $\{\mathbf{x}, \mathbf{y}\}$ does not contain two linearly independent vectors $\mathbf{x}'$ and $\mathbf{y}'$ such that either $\|\mathbf{x}'\| < \min\{\|\mathbf{x}\|, \|\mathbf{y}\|\}$ or $\|\mathbf{x}'\| \leq \|\mathbf{y}'\| < \max\{\|\mathbf{x}\|, \|\mathbf{y}\|\}$. Any basis $\{\mathbf{a}, \mathbf{b}\}$ of a 2-dimensional lattice can be Gauss reduced by repeating the following two steps until $\mathbf{b}$ does not change: (1) If $\|\mathbf{b}\| \leq \|\mathbf{a}\|$, swap $\mathbf{a}$ and $\mathbf{b}$. (2) Replace $\mathbf{b}$ by $\mathbf{b} - \left\lfloor \frac{\langle \mathbf{a}, \mathbf{b} \rangle}{\langle \mathbf{a}, \mathbf{a} \rangle} \right\rceil \cdot \mathbf{a}$.

In the remainder of the paper, we use the following terminology: a vector $\mathbf{y}$ *can be reduced* with respect to a vector $\mathbf{x}$, if $2 \cdot |\langle \mathbf{x}, \mathbf{y} \rangle| > \langle \mathbf{x}, \mathbf{x} \rangle$. In this case, *reducing* $\mathbf{y}$ with respect to $\mathbf{x}$ means, replacing $\mathbf{y}$ by $\mathbf{y} - \left\lfloor \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\langle \mathbf{x}, \mathbf{x} \rangle} \right\rceil \cdot \mathbf{x}$. We sometimes also mean *reducing* $\mathbf{y}$ with respect to $\mathbf{x}$ to include checking the condition $2 \cdot |\langle \mathbf{x}, \mathbf{y} \rangle| > \langle \mathbf{x}, \mathbf{x} \rangle$, and if it is not satisfied, no further computation takes place and the vector $\mathbf{y}$ remains unchanged.

Ideal lattices are lattices that arise from ideals in a ring R. For example, consider the ring $R = \mathbb{Z}[X]/(f)$, where f is a monic polynomial of degree n, such that R consists of polynomials with degree at most $n - 1$. One way of associating a lattice to an ideal is via the so-called *coefficient embedding*. A polynomial in R can be identified with its coefficient vector, which is an element from $\mathbb{Z}^n$. Any ideal $I \subseteq R$ is an additive subgroup, and therefore the corresponding coefficient vectors form a lattice over $\mathbb{Z}^n$. Furthermore, an ideal is closed under multiplication with arbitrary ring elements $r \in R$. Hence, the ideal lattice inherits an additional algebraic structure from the ring multiplication.

If f is irreducible, then for any $v \in R$, the coefficient vectors corresponding to v, $X \cdot v$, ..., $X^{n-1} \cdot v \mod f$ are linearly independent and therefore together span a full-rank lattice. This

means that the lattice corresponding to the principal ideal generated by v can be represented by using only one element v. In comparison to representing the lattice without using the ideal structure or representing a general integral lattice in $\mathbb{Z}^n$, storage is reduced by a factor n.

2.1 A Brief Overview of Sieving Algorithms

Sieving algorithms for lattice problems date back to Ajtai, Kumar and Sivakumar (AKS) [3], who proposed the first randomized $2^{\mathcal{O}(n)}$ (in both time and space) algorithm to solve the shortest vector problem. At the core of this algorithm is a sieving procedure, which takes a list of vectors and returns a smaller list of shorter vectors. By generating an exponentially large list and iteratively applying this sieve, one obtains two vectors such that their difference is the shortest non-zero vector in the lattice.

A major obstacle to proving the correctness of sieving algorithms is the occurrence of collisions. A collision occurs when at some point during the algorithm, two different vectors get reduced to the same vector by the sieve step, which basically means that one vector is "lost". It is hard to bound the probability that collisions occur, and one way to surmount this obstacle is to make additional heuristic assumptions that allow to prove correctness. When running such an algorithm in practice, it is possible to verify these heuristics for practical parameters. In the following, we clearly distinguish between algorithms based on heuristics and algorithms that can be proven to be correct without any additional assumptions. Table 1 contains a summary of both types of algorithms.

Nguyen and Vidick [39] simplified the AKS sieve [3] and provided concrete constants for the $\mathcal{O}(n)$ term in the exponent. More importantly, they introduced a heuristic variant with much smaller complexity. Subsequent works change the sieving procedure, improving the asymptotic complexity of the heuristic variant of this particular algorithm [54, 55]. In the meantime, Micciancio and Voulgaris [53] proposed two new algorithms, the provable *List sieve* and its heuristic counterpart *Gauss sieve*. As opposed to the previous sieving algorithms, which start with a big list and reduce it in size as the algorithm progresses, these two algorithms iteratively build a list and use it to reduce the size of newly sampled vectors. List sieve only reduces new vectors with respect to the list, whereas Gauss sieve has an extra step during which it reduces all list vectors with respect to the newly sampled vectors as well. The maximum list size of List sieve can be bounded using an angle argument due to Kabatiansky and Levenshtein [27] and this is used to also bound the running time of the algorithm.

Table 1 lists the asymptotic runtimes and space complexities of several algorithms aimed at solving SVP. Even though they have a worse asymptotic runtime, simple enumeration algorithms due to Fincke and Pohst [14] (often based on the Schnorr-Euchner implementation [49]) using extreme pruning [16] perform best in practice. One reason is the fact that these methods are virtually memory-less, in the sense that they require only minimal storage. Kannan enumeration [28] recursively uses a stronger form of pre-processing, and has been shown by Hanrot and Stehlé [24] to achieve a better asymptotic runtime. The Voronoi-cell based algorithm by Micciancio and Voulgaris [37] has the best asymptotic runtime, but appears to perform the worst in practice. Finally, in recent work, Becker, Gama and Joux [6] proposed a new algorithm based on a decomposition approach using overlattices. This approach has been classified by some as a sieve. It appears to be the fastest heuristic algorithm that solves both SVP and CVP. It should be noted that all sieving-type algorithms of Table 1 are listed

Table 1. Asymptotic complexity of algorithms solving SVP provably (left) and heuristically (right), ignoring poly-log factors in the exponent.

Algorithm	Time	Memory
Fincke-Pohst enumeration [14]	$2^{\mathcal{O}(n^2)}$	poly(n)
Kannan enumeration [28, 24]	$n^{n/(2e)}$	poly(n)
AKS sieve [3, 53]	$2^{3.4n}$	$2^{1.97n}$
ListSieve [53]	$2^{3.199n}$	$2^{1.325n}$
ListSieve-birthday [44]	$2^{2.465n}$	$2^{1.233n}$
Voronoi-cell [37]	2^{2n}	2^{n}

Algorithm	Time	Memory
GaussSieve [53]	(?)	$2^{0.2075n}$
Nguyen-Vidick sieve [39]	$2^{0.415n}$	$2^{0.2075n}$
Two level sieve [54]	$2^{0.3816n}$	$2^{0.2557n}$
Three level sieve [55]	$2^{0.3778n}$	$2^{0.2833n}$
Decomposition [6]	$2^{0.3374n}$	$2^{0.2925n}$

according to their minimal asymptotic runtime, since in some cases, it is possible to optimize them for space requirements, resulting in less space used but more time required.

2.2 Gauss Sieve

In this paper we focus on Gauss sieve. The maximum list size of Gauss sieve can be bounded using the kissing number, but there is no provable bound on the running time of the algorithm, mostly because there is no bound on the number of collisions. Because all pairs of vectors need to be Gauss-reduced, the running time is at least quadratic in the list size, but practical experiments by Voulgaris and Micciancio suggest a running time of $2^{0.48n}$. Micciancio and Voulgaris also mentioned that their algorithms could make use of the structure of ideal lattices in certain cases, reducing the list size [53]. This concept was further explored by Schneider [47], who showed that the reduction in list size also results in a reduction of the running time. Experimentally, the Gauss sieve appears to be the fastest in practice, compared to other sieving algorithms.

Algorithm 1 outlines the Gauss sieve algorithm as described by Micciancio and Voulgaris [53] including some modifications. Specifically, this description contains some practical optimizations by Voulgaris that were also mentioned by Milde and Schneider [38] and are related to the reduction condition (e.g. the original condition $\|\mathbf{u} \pm \mathbf{w}\| > \|\mathbf{w}\|$ is replaced by $2 \cdot |\langle \mathbf{u}, \mathbf{w} \rangle| > \langle \mathbf{w}, \mathbf{w} \rangle$ since this can be computed more efficiently when the squared norms of the vectors are stored together with the vectors). The structure has been slightly adapted to reflect the current implementations in practice (e.g. [52] uses a linked list to represent the list of pairwise Gauss reduced vectors). The input consists of a basis $\mathbf{B}$, a target length μ and a maximum number of collisions c.

The algorithm terminates either when a vector $\mathbf{v} \in \mathcal{L}(B)$ has been found such that $\|\mathbf{v}\| \leq \mu$ or when the number of collisions exceeds c. The motivation for this latter condition is that we do not have to specify a target length μ. One expects (heuristically) that when a short vector is found, it is found repeatedly (by subtracting different vectors), which results in many collisions (reducing to the zero vector). This reasoning seems valid in practice, the proportion of collisions appears to increase dramatically, once a shortest vector of the lattice is in the list. In practice, one typically adapts Algorithm 1 slightly such that the termination condition depends exclusively either on the target length μ or on the maximum number of collisions c. Note that Algorithm 1 makes use of a sorted linked-list L. In practice one could modify Algorithm 1 such that it uses an unsorted array instead. We choose to display the algorithm with a sorted linked list to ease the explanation.

Algorithm 1 The Gauss sieve algorithm [53]. Given a basis $\mathbf{B}$ and bounds $\mu, c > 0$, return a short vector $\mathbf{v}$. The algorithm terminates either when a vector $\mathbf{v}$ is found such that $\|\mathbf{v}\| \leq \mu$, in which case $\mathbf{v}$ is returned, or when the number of collisions is greater than or equal to c, in which case the smallest vector in the list is returned. The i-th vector from the sorted linked-list L is denoted by ℓ_i and the cardinality of L by $\#L$. The variable S is a stack.

```
1: function GaussSieve(B, μ, c)
2:     L ← {0}, S ← {}, K ← 0
3:     while K < c do
4:         v_new ← Sample(S), i ← 0
5:         while i < #L and ‖ℓ_i‖ ≤ ‖v_new‖ do
6:             if Reduce(v_new, ℓ_i) then
7:                 i ← −1
8:                 if ‖v_new‖ == 0 then
9:                     K ← K + 1
10:                    v_new ← Sample(S)
11:                end if
12:                if ‖v_new‖ ≤ μ then return v_new
13:                end if
14:            end if
15:            i ← i + 1
16:        end while
17:        if ‖v_new‖ > 0 then
18:            Insert v_new into L at position i
19:            i ← i + 1
20:        end if
21:        while i < #L and ‖ℓ_i‖ ≥ ‖v_new‖ do
22:            if Reduce(ℓ_i, v_new) then
23:                if ‖ℓ_i‖ == 0 then
24:                    Remove ℓ_i from L
25:                    i ← i − 1, K ← K + 1
26:                else
27:                    if ‖ℓ_i‖ ≤ μ then return ℓ_i
28:                    end if
29:                    Move ℓ_i from L to S
30:                end if
31:            end if
32:            i ← i + 1
33:        end while
34:    end while
35:    return the smallest ℓ_i from L
36: end function

37: function Sample(S)
38:     if S is empty then
39:         Sample v ≠ 0 using GPV [21]
40:     else
41:         Pop v from S
42:     end if
43:     return v
44: end function

45: function Reduce(u, w)
46:     if 2 · |⟨u, w⟩| > ⟨w, w⟩ then
47:         u ← u − ⌈⟨u,w⟩/⟨w,w⟩⌋ w
48:         return true
49:     end if
50:     return false
51: end function
```

The main approach of this algorithm is, given a Gauss reduced list of samples, to sample a new vector $\mathbf{v}_{\text{new}}$ and ensure this vector is Gauss reduced with respect to all vectors in the list. This is achieved by reducing $\mathbf{v}_{\text{new}}$ with respect to the vectors in the list that are shorter than itself and insert this new vector in the list. Finally, the (updated) $\mathbf{v}_{\text{new}}$ is used to reduce vectors in the list that are longer than itself. List vectors that are reduced by a new vector are moved to the stack (since we cannot guarantee that they are still Gauss reduced with respect to all other vectors in the list) and the sampling procedure takes vectors from the stack before sampling brand new ones (e.g. using GPV [21]).

3 Parallel Gauss Sieve

In order to tap into the vast wealth of modern computing power, it is preferable that algorithms can be computed on many computational cores concurrently. In this section we briefly examine previous attempts to compute the Gauss sieve in parallel and propose modifications resulting in our new variant of the algorithm.

Milde and Schneider [38] proposed to split up the problem into several "instances" and to connect each instance in a circular fashion. Each instance is computed on a separate computational unit and consists of a list, a stack and a queue (all private to this computational unit). Each instance acts as a local independent Gauss sieve, taking new vectors from the queue (filled by its neighboring instance), stack (filled by its local Gauss sieve algorithm) and the sampler in that order. However, instead of inserting vectors in the list once they are pairwise reduced with respect to the list, as outlined in Algorithm 1, they are sent on to the queue of the next instance. Only vectors that have made a full round across all instances are inserted. In this scenario, there is no explicit synchronization between processors (the instances). All communication consists of vectors being sent from one instance to the next, possibly in batches to reduce the communication overhead. In each round or iteration, a processor receives vectors from its predecessor, takes one vector and reduces it pairwise with each vector in its list and sends vectors to its successor if necessary. When testing this approach, Milde and Schneider found that it did not appear to scale well with the number of parallel instances [38]. The list sizes between instances would vary greatly and instances with small lists would sample lots of new vectors, which would in turn get stuck in the queues of instances with bigger lists (since these need to perform more work). Attempts to work around these traffic jams by relaxing the pairwise reducedness between instances caused the global list size to increase dramatically and ruined the efficiency. Another side-effect of this approach is that when a vector is inserted in a local list, it is Gauss reduced with respect to all other *list vectors*, not to the other vectors which are currently in the queue and stack of the other instances. The number of such vectors can be significant and this property increases the overall list size and hence the running time of the algorithm.

An approach which resolves this latter issue is due to Ishiguro et al. [26]. They propose to have a copy of the full global list available to each computational unit (instead of splitting it up across the units) and instead, process the vectors in batches. This process consists of three stages: reducing the new batch of vectors with respect to the global list, reducing the new batch of vectors with respect to each other and reducing the list vectors with respect to the batch vectors. If at any stage any vector is reduced, it is removed and moved to the (global) stack, whereas all vectors that "survived" the three stages comprise the new list. Both before and after these three stages, every pair of vectors in the list is guaranteed to be reduced. This results in a smaller overall list size, lowering the running time of the algorithm.

The obvious drawback of this approach is that all processors need to have access to the full list. Ishiguro et al. worked around this by using computers on the Amazon cluster that have a huge shared memory [26]. In experiments, this approach was used to solve a 128-dimensional ideal lattice challenge, using speed-ups that are discussed in Section 4. In practice, this means that up to a certain number of processors there is virtually no communication overhead, as everything is performed in shared memory. However, after exceeding this limit there needs to be communication to synchronize between the processors on different machines. Moreover, this approach might not scale well on conventional large compute clusters which are not equipped with shared memory between groups of nodes. Specifically, for each round, the batch

of sampled vectors must be distributed among all processors. If multiple processors make different changes to independent copies of the same batch vector, then with high probability these two copies will encounter each other in a later round and result in unnecessary collisions. There needs to be some communication to avoid this issue. Finally, any changes that are made to the list must be distributed among the processors as well.

3.1 A New Parallel Gauss Sieve Approach

We propose a modified version of the parallel Gauss sieve algorithm which combines the best of both previous approaches. Each processor maintains its own local list as in the approach by Milde and Schneider (reducing the storage per node), but the algorithm consists of synchronized rounds which ensure that all vectors in the union of all local lists are pairwise reduced as in the Ishiguro et al. approach (lowering the list size and therefore the run-time). All processors *collectively* sample a batch of new vectors. Each vector in the batch is reduced with respect to all vectors in the local list of each processor. Then each processor pairwise reduces a selection of the batch vectors with respect to the other batch vectors. Next, all processors communicate which sample vectors have survived globally, i.e. which vectors are pairwise Gauss reduced with respect to all list vectors and each other, and what the sizes of their local lists are. The processor with the smallest local list inserts the surviving batch vectors and the next round begins.

Algorithm 2 is an algorithmic description of this process. The steps that require communication are marked in red and are underlined. Let there be N nodes available for the parallel computation, and denote by N_i the i-th node for $0 \leq i < N$. Each node N_i locally maintains a list L_i, a list of samples Q_i, a list of reduced samples Q_i' and a stack S_i. The union $L = \bigcup_i L_i$ of the local lists can be viewed as the global list in the Gauss sieve from Algorithm 1 (note that Algorithm 2 does not require the lists to be sorted). At the start of each round, the collective sampling is done as follows. The first node N_0 compiles a list Q of k new samples by taking them from its stack or sampling them using the GPV algorithm [21]. The number of samples k is a parameter of the algorithm. It then broadcasts Q to all other nodes and each node N_i copies Q into their local sample list Q_i. Each node now locally compares all pairs of vectors from Q_i and L_i to see if they are pairwise reduced and reduces vectors wherever possible. Vectors from Q_i that are reduced are removed and stored in the reduced samples list Q_i', whereas vectors from L_i that are reduced are removed and stored in the local stack S_i. When all pairs have been checked, each node compares the surviving samples that are still in Q_i to each other. This is done in a structured way to split the work between all nodes: each node only checks a predetermined subset of Q against the rest of Q_i. Again, any sample that is reduced during this stage is removed from Q_i and added to Q_i'. Note that, although lines 9-14 and 17-22 of Algorithm 2 both list the reduce function twice, they require only a single inner product in practice, as they make the two comparisons $2 \cdot |\langle \mathbf{v}, \mathbf{l} \rangle| > \langle \mathbf{l}, \mathbf{l} \rangle$ and $2 \cdot |\langle \mathbf{l}, \mathbf{v} \rangle| > \langle \mathbf{v}, \mathbf{v} \rangle$ which share the same left-hand term.

At the end of this phase any two vectors from the set $(\bigcup_i L_i) \cup (\bigcap_i Q_i)$ are Gauss reduced. This is the case because this set contains exactly the *survivors* of the reduction phase, i.e. all vectors that could not be reduced with respect to any other vector in this set. The surviving sample vectors are the vectors in the set $\bigcap_i Q_i$ which need to be inserted into the global list $L = \bigcup_i L_i$, i.e. the collection of local lists, and it has to be decided into which local list L_i they are inserted. In order to determine this, the nodes collectively compute $\bigcap_i Q_i$, i.e. the set of original Q-vectors that have survived, and also which node holds the smallest list. The

Algorithm 2 Parallel Gauss sieve variant. Given a basis $\mathbf{B}$, length bound μ, and list of N nodes N_i, $0 \leq i < N$, return a short vector $\mathbf{v}$ with $\|\mathbf{v}\| \leq \mu$ as soon as it is found. Let (L_i, Q_i, Q'_i, S_i) respectively denote the list, sample list, non-survivor list and stack of node N_i. Any step requiring communication is colored in red and is underlined.

function GaussSieve($\mathbf{B}, \mu, \{N_i\}_{i=0}^{N}$)
  while short vector not found **do**
    N_0 samples a list Q of k vectors via Sample(S_0) (see Algorithm 1)
    N_0 broadcasts Q
    for i in {0,1,...,N-1} **do**
      $Q_i \leftarrow Q$
      $Q'_i \leftarrow \varnothing$, $S_i \leftarrow \varnothing$
      for $(\mathbf{v}, \mathbf{l}) \in Q_i \times L_i$ **do**
        if Reduce($\mathbf{v}, \mathbf{l}$) **then**
          Move $\mathbf{v}$ from Q_i to Q'_i
        end if
        if Reduce($\mathbf{l}, \mathbf{v}$) **then**
          Move $\mathbf{l}$ from L_i to S_i
        end if
      end for
      for $(\mathbf{v}, \mathbf{l}) \in Q_i \times Q_i$ **do**
        if Reduce($\mathbf{v}, \mathbf{l}$) **then**
          Move $\mathbf{v}$ from Q_i to Q'_i
        end if
        if Reduce($\mathbf{l}, \mathbf{v}$) **then**
          Move $\mathbf{l}$ from Q_i to Q'_i
        end if
      end for
    end for
    Compute $\bigcap_i Q_i$
    for $\mathbf{v} \in (\bigcap_i Q_i)$ **do**
      if $\|\mathbf{v}\| \leq \mu$ **then**
        return $\mathbf{v}$
      end if
    end for
    Compute index j such that $|L_j|$ is minimal
    $L_j \leftarrow L_j \cup (\bigcap_i Q_i)$
    for $\mathbf{v} \in Q \setminus (\bigcap_i Q_i) = \bigcup_i Q'_i$ **do**
      Add minimal representative of $\mathbf{v}$ to S_0
    end for
    for $\mathbf{v} \in \bigcup_i S_i$ **do**
      if $\|\mathbf{v}\| \leq \mu$ **then**
        return $\mathbf{v}$
      end if
    end for
    for i in {0,1,...,N-1} **do**
      N_i sends S_i to N_0
    end for
    $S_0 = S_0 \cup (\bigcup_i S_i)$
  end while
end function

first task requires the computation of a collective bitwise AND on a bitmask which lists the surviving vectors per node. The second requires a gathering of the list-sizes and a broadcast of the result (both are elementary standard operations which are available in any parallel computation library). The node with the smallest list inserts all the surviving vectors into its list, which requires *no additional communication* because the survivors are exactly those vectors from the original Q that were unchanged by all nodes.

The last step of the round is to gather all vectors that were reduced in this round in the global stack S_0 situated at the first node. The reduced vectors are the reduced samples in all the Q'_i, together with the reduced list vectors in the local stacks S_i. We do not propagate all vectors in $\bigcup_i Q'_i$ into the global stack for sampling. The reason is that this leads to unnecessary collisions. A vector $\mathbf{v}$ which is reduced at two different nodes N_i and N_j by two different list vectors $\mathbf{l}_i$ and $\mathbf{l}_j$, produces two vectors $\mathbf{v}_i \in Q'_i$ and $\mathbf{v}_j \in Q'_j$. If both of them are passed to the next rounds and are inserted into Q, it is likely that $\mathbf{v}_i$ will be reduced at node N_j by $\mathbf{l}_j$ and $\mathbf{v}_j$ at N_i by $\mathbf{l}_i$, leading to the same vector going back to the global stack at the end of the round. This behavior can be (and has been) observed in practice. Therefore, from all vectors that arise from the same sample vector by reduction with respect to different list vectors, we only propagate the vector with the minimal norm. We call this vector the *minimal representative* of $\mathbf{v}$. For each non-survivor from the original list Q, this requires a collective computation across all nodes of the vector with the minimal norm among the vectors in $\bigcup_i Q'_i$

that originate from the same vector in Q. The node that contains this *minimal representative* of the non-survivor sends it to the first node. Finally, the vectors in the local stacks S_i are all taken into the next round and each node sends the vectors from their stack S_i to the first node. This concludes the round.

4 Sieving in Ideal Lattices

Ideal lattices are lattices with additional structure, namely they are also ideals in a ring R. We restrict to lattices which correspond to ideals in the ring of integers of a cyclotomic number field, namely $R = \mathbb{Z}[X]/(\Phi_m(X))$, where $m = 2n$ is a power of 2 such that the cyclotomic polynomial is $\Phi_m(X) = X^n + 1$. Ideals are invariant under multiplication by arbitrary ring elements, in particular, if $a(X)$ is an element in an ideal, then the product $X \cdot a(X)$ also belongs to the ideal. This means that for any vector a in the ideal lattice, multiplying by powers of X yields lattice vectors $X^i \cdot a$ for all $i \in \mathbb{Z}$. Note that we allow negative exponents because $X^{-1} = -X^{n-1}$. An element $a \in R$ is of the form $a(X) = \sum_{i=0}^{n-1} a_{i-1} X^i$ and is given by its coefficient vector $\mathbf{a} = (a_0, a_1 \ldots, a_{n-1}) \in \mathbb{Z}^n$. The coefficient vectors of $X^i \cdot a(X) \mod (X^n + 1)$ are denoted by $X^i \cdot \mathbf{a}$ for $i \in \mathbb{Z}$, and we call these vectors the *rotations* of $\mathbf{a}$. Indeed, the polynomial $X \cdot a(X) \mod (X^n + 1)$ corresponds to a cyclic rotation of the coefficient vector $\mathbf{a}$, except for the sign of a_{n-1}: $X \cdot a(X) = -a_{n-1} + \sum_{i=1}^{n-1} a_{i-1} X^i \pmod{X^n + 1}$, i.e. $X \cdot \mathbf{a} = (-a_{n-1}, a_0, \ldots, a_{n-2})$. Conversely, rotating to the left corresponds to multiplying by $X^{-1} = -X^{n-1} \pmod{X^n + 1}$: $X^{-1} \cdot \mathbf{a} = -X^{n-1} \cdot \mathbf{a} = (a_1, \ldots, a_{n-1}, -a_0)$.

Schneider [47] and Ishiguro et al. [26] previously have used the fact that an ideal lattice contains all rotations of a lattice vector. Since a single stored vector actually represents its n rotations, the list storage is reduced by a factor n. However, these works do not seem to use the ring structure any further. In this section, we show how to use the relation between ring multiplication and the scalar products of rotations to further improve the efficiency of Gauss sieve for ideal lattices. We observe that for all n^2 possible pairs of rotations of one vector $\mathbf{a}$ with rotations of another vector $\mathbf{b}$, it can be checked whether they are Gauss reduced by only n comparisons using the n scalar products $\langle \mathbf{a}, X^i \cdot \mathbf{b} \rangle$. Next, we show that all n scalar products can be computed by a single ring product. Since in our case, the ring product is a negacyclic convolution, it can be computed via a fast Fourier transform (FFT) algorithm in $\mathcal{O}(n \ln n)$ arithmetic operations instead of $\mathcal{O}(n^2)$ for a naive, separate computation of the scalar products. We describe Nussbaumer's symbolic FFT algorithm [40] for such negacyclic convolutions, explain how it can be computed using vector instructions, and mention that similar techniques apply to other relevant settings such as NTRU lattices.

In the following lemma we collect a few useful identities for such negacyclic rotations, which can be easily proved by explicitly writing them out.

Lemma 1. *Let $a, b \in R = \mathbb{Z}[X]/(X^n + 1)$ for n a power of 2, let $\mathbf{a}, \mathbf{b}$ be their coefficient vectors, and let $i, j \in \mathbb{Z}$. Then with notation as above, we have:*

$$X^i \cdot (X^j \cdot \mathbf{a}) = X^{i+j} \cdot \mathbf{a}, \quad X^i \cdot (\mathbf{a} + \mathbf{b}) = X^i \cdot \mathbf{a} + X^i \cdot \mathbf{b}, \quad X^n \cdot \mathbf{a} = -\mathbf{a},$$
$$\langle X^i \cdot \mathbf{a}, X^i \cdot \mathbf{b} \rangle = \langle \mathbf{a}, \mathbf{b} \rangle, \quad \langle X^i \cdot \mathbf{a}, X^j \cdot \mathbf{b} \rangle = \langle \mathbf{a}, -X^{n-i+j} \mathbf{b} \rangle.$$

The next lemma shows that by computing the n scalar products $\langle \mathbf{a}, X^\ell \cdot \mathbf{b} \rangle$, $0 \le \ell < n$, one can easily check whether the two vectors $X^i \cdot \mathbf{a}$ and $X^j \cdot \mathbf{b}$ are Gauss reduced for any $i, j \in \mathbb{Z}$.

Lemma 2. *Let $a, b \in R = \mathbb{Z}[X]/(X^n + 1)$ with coefficient vectors $\mathbf{a}, \mathbf{b}$. If $2|\langle \mathbf{a}, X^\ell \cdot \mathbf{b} \rangle| \le \min\{\langle \mathbf{a}, \mathbf{a} \rangle, \langle \mathbf{b}, \mathbf{b} \rangle\}$ for all $0 \le \ell < n$, then $X^i \cdot \mathbf{a}$ and $X^j \cdot \mathbf{b}$ are Gauss reduced for all $i, j \in \mathbb{Z}$.*

Proof. The properties in Lemma 1 show that $|\langle X^i \cdot \mathbf{a}, X^j \cdot \mathbf{b}\rangle| = |\langle \mathbf{a}, X^{n-i+j}\mathbf{b}\rangle| = |\langle \mathbf{a}, X^\ell \mathbf{b}\rangle|$, where $\ell = (n - i + j) \mod n$, i.e. $0 \leq \ell < n$, as well as $\langle X^i \cdot \mathbf{a}, X^i \cdot \mathbf{a}\rangle = \langle \mathbf{a}, \mathbf{a}\rangle$ and $\langle X^j \cdot \mathbf{b}, X^j \cdot \mathbf{b}\rangle = \langle \mathbf{b}, \mathbf{b}\rangle$. Hence the lemma follows. □

If the condition in Lemma 2 is not satisfied, then either $\mathbf{a}$ or $\mathbf{b}$ can be reduced by some rotation of the other vector. Namely, if $2|\langle \mathbf{a}, X^\ell \cdot \mathbf{b}\rangle| > \langle \mathbf{b}, \mathbf{b}\rangle$, then $\mathbf{a}$ can be reduced by $X^\ell \cdot \mathbf{b}$ and if $2|\langle \mathbf{a}, X^\ell \cdot \mathbf{b}\rangle| > \langle \mathbf{a}, \mathbf{a}\rangle$, then $\mathbf{b}$ can be reduced by $X^{n-\ell} \cdot \mathbf{a}$. Note that by the proof of Lemma 2, this is equivalent to saying that, if $2|\langle \mathbf{a}, X^\ell \cdot \mathbf{b}\rangle| > \langle \mathbf{b}, \mathbf{b}\rangle$, then any rotation of $\mathbf{a}$ can be reduced by some rotation of $\mathbf{b}$. The same holds for $\mathbf{b}$ with the second condition. The function `ReduceRot` in Algorithm 3 can be used instead of `Reduce` in Algorithm 1 in the ideal lattice setting. When included in Algorithm 2, the checks of the inequalities in 9-14 and 17-22 can be combined as mentioned in Section 3.1 and require only n comparisons.

4.1 Computing n Scalar Products by a Single Ring Product

Given two elements $a, b \in R$, we can use FFT-based methods to compute their polynomial ring product

$$c(X) = a(X) \cdot b(X) \mod (X^n + 1).$$

In this subsection, we describe the relation of this product to the scalar products of the rotations of the elements a and b. For $a, b, c \in R$, let $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$, $\mathbf{b} = (b_0, b_1, \ldots, b_{n-1})$ and $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$ be their coefficient vectors. Furthermore, define the reflex polynomial $b^{(R)}(X)$ of $b(X)$ as

$$b^{(R)}(X) = X^{n-1} \cdot b(X^{-1}),$$

and let $\mathbf{b}^{(R)} = (b_{n-1}, b_{n-2}, \ldots, b_0)$ be its coefficient vector. The coefficients c_i of the ring product are given by the following equations involving scalar products of $\mathbf{a}$ with rotations of $\mathbf{b}^{(R)}$:

$$\begin{array}{lllll}
c_0 & = a_0b_0 - a_1b_{n-1} - \cdots - a_{n-1}b_1 & = \langle \mathbf{a}, (b_0, -b_{n-1}, \ldots, -b_1)\rangle & = \langle \mathbf{a}, -X \cdot \mathbf{b}^{(R)}\rangle, \\
c_1 & = a_0b_1 + a_1b_0 - a_2b_{n-1} - \cdots - a_{n-1}b_2 & = \langle \mathbf{a}, (b_1, b_0, -b_{n-1}, \ldots, -b_2)\rangle & = \langle \mathbf{a}, -X^2 \cdot \mathbf{b}^{(R)}\rangle, \\
\vdots & \vdots & \vdots & \vdots \\
c_{n-1} & = a_0b_{n-1} + a_1b_{n-2} \cdots + a_{n-1}b_0 & = \langle \mathbf{a}, (b_{n-1}, b_{n-2}, \ldots, b_0)\rangle & = \langle \mathbf{a}, \mathbf{b}^{(R)}\rangle.
\end{array}$$

In general, this means that we get $c_i = \langle \mathbf{a}, -X^{i+1}\mathbf{b}^{(R)}\rangle$, $0 \leq i < n$. If we replace b by $-b^{(R)}(X)$ and instead compute the ring product c as

$$c(X) = a(X) \cdot (-b^{(R)}(X)) \mod (X^n + 1),$$

we obtain in the coefficients of c the scalar products $c_i = \langle \mathbf{a}, X^{i+1} \cdot \mathbf{b}\rangle$ because $(b^{(R)})^{(R)}(X) = b(X)$. Now the last scalar product is $c_{n-1} = \langle \mathbf{a}, -\mathbf{b}\rangle$. Using one of the properties described above and computing c as the product with $-X \cdot b^{(R)}(X)$ instead, we get that

$$c(X) = a(X) \cdot (-X \cdot b^{(R)}(X)) \mod (X^n + 1)$$

has coefficients $c_i = \langle \mathbf{a}, X^i \cdot \mathbf{b}\rangle$ because the product $c(X) = (Xa(X)) \cdot (-b^{(R)}(X)) \mod (X^n + 1)$ has coefficients $c_i = \langle X \cdot \mathbf{a}, X^{i+1} \cdot \mathbf{b}\rangle = \langle \mathbf{a}, X^i \cdot \mathbf{b}\rangle$. So we have proved the following lemma.

Algorithm 3 Algorithms for pairwise reducing $(\mathbf{a}, X^i \cdot \mathbf{b})$ for all $i \in \{0, \ldots, n-1\}$ The function `ReduceFFT` assumes as additional input the precomputed values $\hat{a} = \mathrm{FFT}(\mathbf{a})$ and $\hat{b}^{(R)} = \mathrm{FFT}(-X \cdot \mathbf{b}^{(R)})$.

function ReduceRot($\mathbf{a}, \mathbf{b}$)
  for $i = 0, \ldots, n-1$ **do**
    if $2 \cdot |\langle \mathbf{a}, X^i\mathbf{b} \rangle| > \langle \mathbf{b}, \mathbf{b} \rangle$ **then**
      $\mathbf{a} \leftarrow \mathbf{a} - \left\lceil \frac{\langle \mathbf{a}, X^i\mathbf{b} \rangle}{\langle \mathbf{b}, \mathbf{b} \rangle} \right\rfloor X^i\mathbf{b}$
      return true
    end if
  end for
  return false
end function

function ReduceFFT($(\mathbf{a}, \hat{a}), (\mathbf{b}, \hat{b}^{(R)})$)
  $z \leftarrow \mathrm{FFT}^{-1}(\hat{a} \odot \hat{b}^{(R)})$
  for $i = 0, \ldots, n-1$ **do**
    if $2 \cdot |z_i| > \langle \mathbf{b}, \mathbf{b} \rangle$ **then**
      $\mathbf{a} \leftarrow \mathbf{a} - \left\lceil \frac{z_i}{\langle \mathbf{b}, \mathbf{b} \rangle} \right\rfloor X^{n-i+1} \cdot \mathbf{b}$
      return true
    end if
  end for
  return false
end function

Lemma 3. *Let $\mathbf{a}, \mathbf{b} \in \mathbb{Z}^n$ be two coefficient vectors corresponding to the ring elements $a, b \in R = \mathbb{Z}[X]/(X^n + 1)$. Let*

$$c(X) = a(X) \cdot (-X \cdot b^{(R)}(X)) \mod (X^n + 1)$$

and let $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1}) \in \mathbb{Z}^n$ be its coefficient vector. Then $c_i = \langle \mathbf{a}, X^i \cdot \mathbf{b} \rangle$ for $0 \leq i < n$.

This means that on input of two vectors $\mathbf{a}$ and $\mathbf{b}$ corresponding to polynomials $a(X)$ and $b(X)$ in R, by reordering the coefficients of b and adjusting the signs to get the polynomial $-X \cdot b^{(R)}(X)$ and then computing the ring product $c(X) = a(X) \cdot (-X \cdot b^{(R)}(X)) \mod (X^n+1)$ using an FFT-algorithm in $\mathcal{O}(n(\ln n)(\ln \ln n))$ bit operations (see the approach described in Section 4.2), one obtains all scalar products $c_i = \langle \mathbf{a}, X^i \cdot \mathbf{b} \rangle$ of $\mathbf{a}$ with the rotations of $\mathbf{b}$.

4.2 Nussbaumer's Algorithm for Negacyclic Convolutions

The product of two polynomials in R, i.e. a product $a(X)b(X) \mod (X^n + 1)$ where n is a power of 2, is a negacyclic convolution and we use Nussbaumer's symbolic algorithm described in [40] to compute it; see [29, Exercise 4.6.4.59] and [13, Section 9.5.7] for more details and explanations. In this section, we describe the algorithm and how we use it in the reduction steps of the sieving algorithm. The algorithm recursively reduces the negacyclic convolution of vectors of length n, by a re-organization of coefficients, to additions, subtractions and negacyclic convolutions of shorter vectors.

The key observation is that the ring extension $R = \mathbb{Z}[X]/(X^n+1)$ of $\mathbb{Z}$ can be decomposed into two extensions as follows. Let $n = 2^k = s \cdot r$ where $s \mid r$, e.g. $s = 2^{\lfloor k/2 \rfloor}$ and $r = 2^{\lceil k/2 \rceil}$. Then

$$R \cong S = T[X]/(X^s - Z), \text{ where } T = \mathbb{Z}[Z]/(Z^r + 1).$$

Note that $Z^{r/s}$ is an s-th root of -1 in T and $X^s = Z$ in S. This isomorphism is expressed simply by a re-ordering of coefficients. An element $a(X) = \sum_{i=0}^{n-1} a_{i-1}X^i \in R$ can be written as an element of S by replacing all powers X^s by Z, which gives

$$a(X) = \sum_{i=0}^{s-1} A_i(Z)X^i, \text{ with } A_i(Z) = a_i + a_{i+s}Z + \cdots + a_{i+s(r-1)}Z^{r-1} \in T.$$

Nussbaumer's algorithm for computing the negacyclic convolution of two polynomials $a, b \in R$ begins by interpreting the coefficients of a, b as the sequences of elements $A_0, \ldots, A_{s-1} \in T$ and $B_0, \ldots, B_{s-1} \in T$ as above. The result $a(X)b(X) \mod (X^n + 1)$ can then be computed as a cyclic convolution of the sequences $A = (A_0, A_1, \ldots A_{s-1}, 0, \ldots, 0)$ and $B = (B_0, B_1, \ldots B_{s-1}, 0, \ldots, 0)$ of length $2s$. The products of the A_i with the B_j for $1 \leq i, j < s$ are computed in the ring T and are thus negacyclic convolutions of sequences of length r.

The cyclic convolution can be computed by a symbolic fast Fourier transform algorithm (FFT) using the $2s$-th root of unity $Z^{r/s}$. This means that one computes the discrete Fourier transforms (DFT) $\tilde{A} = (\tilde{A}_0, \ldots, \tilde{A}_{2s-1}), \tilde{B} = (\tilde{B}_0, \ldots, \tilde{B}_{2s-1})$ of the sequences of length $2s$ above, carries out the coefficient-wise multiplications $\tilde{C}_i = \tilde{A}_i \tilde{B}_i$, $0 \leq i < 2s$ by negacyclic convolutions of length r in T, and computes the inverse DFT of $\tilde{C} = (\tilde{C}_0, \ldots, \tilde{C}_{2s-1})$ (using $Z^{r/s}$) to obtain the result $C = (C_0, \ldots, C_{2s-1})$. The final result $c(X) = a(X)b(X) \mod (X^n + 1)$ is given by

$$c(X) = \sum_{i=0}^{2s-1} C_i(X^s)X^i \mod (X^n + 1) \in R$$

and can be computed by s additions and subtractions in T.

In summary, Nussbaumer's algorithm can be divided into three steps: first, the two polynomials a, b are re-ordered and split up into sequences A, B of length $2s$ (half of the entries being zero), each of which is converted to its DFT. The resulting sequences $\tilde{A}, \tilde{B}$ of length $2s$ are then multiplied coefficient-wise via negacyclic convolutions, resulting in a sequence $\tilde{C}$. Finally, this sequence is converted back from the DFT representation and unified into a single polynomial that corresponds to the negacyclic convolution of the two original polynomials. The second step of computing negacyclic convolutions of shorter sequences can be expanded recursively by splitting each of the $2s$ coefficients of each sequence into $2s'$ pairs themselves, resulting in even smaller negacyclic convolutions that need to be computed. This is equivalent to expanding the conversions in the first and third steps to account for the levels of recursion and applying step two to all $4ss'$ sequence elements.

For a vector $\mathbf{a}$ corresponding to $a \in R$, we denote the DFT representation of $\mathbf{a}$ after the combined computations of the conversions described above by $\mathrm{FFT}(\mathbf{a})$. For two such vectors $\mathbf{a}, \mathbf{b}$, the pairwise negacyclic convolution of these DFT representations in step two above is denoted by $\mathrm{FFT}(\mathbf{a}) \odot \mathrm{FFT}(\mathbf{b})$. We denote the representation achieved by computing the inverse DFT of a vector $\hat{c}$ in step three by $\mathrm{FFT}^{-1}(\hat{c})$. Furthermore, let $\hat{a} = \mathrm{FFT}(\mathbf{a})$ and $\hat{b}^{(R)} = \mathrm{FFT}(-X \cdot \mathbf{b}^{(R)})$.

During the (possibly recursive) conversion in step one (when computing $\hat{a}$ and $\hat{b}^{(R)}$), the two polynomials do not interact. Therefore, this step only needs to be performed once and the result can be stored for each sampled vector in the Gauss sieve (significantly decreasing the cost of computing these n inner-products at the cost of additional memory). Then, when two vectors are compared, it only remains to compute the pairwise negacyclic convolutions ($\hat{c} = \hat{a} \odot \hat{b}^{(R)}$) and the (possibly recursive) reverse transformation ($\mathrm{FFT}^{-1}(\hat{c})$). As a result, these two latter steps account for most of the computation time during the algorithm. Given functionality which can compute such a DFT, this can be used in practice to speed-up the reduction step in the Gauss sieve algorithm as outlined in the function `ReduceFFT` in Algorithm 3 and can be used instead of `Reduce` in Algorithm 1 in the ideal lattice setting.

4.3 Nussbaumer's Algorithm Using SSE instructions

We assume that $\hat{a} = \mathrm{FFT}(\mathbf{a})$ and $\hat{b}^{(R)} = \mathrm{FFT}(-X \cdot \mathbf{b}^{(R)})$ are precomputed. Hence, the two most costly steps of Nussbaumer's algorithm are the negacyclic convolutions $\tilde{A}_i \tilde{B}_i \mod (Z^r + 1)$ and the inverse FFT transformations. We implemented Nussbaumer's algorithm completely using SSE instructions to target dimension 128 using two levels of recursion. In this case $n = n_1 = 128 = 2^{k_1} = s_1 \cdot r_1$, where $k_1 = 7$, $s_1 = 2^{\lfloor 7/2 \rfloor} = 8$ and $r_1 = 2^{\lceil 7/2 \rceil} = 16$. For the second level of recursion, we have $n_2 = 16 = 2^{k_2} = s_2 \cdot r_2$, where $k_2 = 4$, $s_2 = 2^{\lfloor 4/2 \rfloor} = 4$ and $r_2 = 2^{\lceil 4/2 \rceil} = 4$. As a result, step two computes $(2s_1)(2s_2) = 16 \cdot 8 = 128$ negacyclic convolutions in dimension $r_2 = 4$, which corresponds to the following computation:

$$
\begin{aligned}
z_0 &= x_0y_0 - x_1y_3 - x_2y_2 - x_3y_1, \\
z_1 &= x_0y_1 + x_1y_0 - x_2y_3 - x_3y_2, \\
z_2 &= x_0y_2 + x_1y_1 + x_2y_0 - x_3y_3, \\
z_3 &= x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0,
\end{aligned}
$$

where $\mathbf{x} = (x_0, x_1, x_2, x_3)$ and $\mathbf{y} = (y_0, y_1, y_2, y_3)$ are one of the 128 pairs and $\mathbf{z} = (z_0, z_1, z_2, z_3)$ is our desired output.

We assume that the size of each vector coefficient is less than 2^{16} such that all coefficients can be stored in 16 bits. This eases adaptation in our vector instruction based implementation. The same assumption has also been made in previous implementations of the parallel Gauss sieve [26] and does not seem to pose any restrictions on the dimensions which are considered in practice nowadays. Moreover, we assume that the entries of the DFT representations after the transformation of step one fit in 16 bits. The main reason for this restriction is that in the second step two of these entries are multiplied together and such values are added which can be done efficiently using the SSE2 "multiply and add packed integers" instruction `PMADDWD` (available as the C-intrinsic `_mm_madd_epi16`). This instruction takes as input two 4-way SIMD 32-bit registers $((a_0, a_1), (a_2, a_3), (a_4, a_5), (a_6, a_7))$ and $((b_0, b_1), (b_2, b_3), (b_4, b_5), (b_6, b_7))$, where the a_i, b_i are 16-bit values, two of which are stored together in one 32-bit register. It then computes (c_0, c_1, c_2, c_3) such that

$$
\begin{aligned}
c_0 &= (a_0 \cdot b_0) + (a_1 \cdot b_1), \\
c_1 &= (a_2 \cdot b_2) + (a_3 \cdot b_3), \\
c_2 &= (a_4 \cdot b_4) + (a_5 \cdot b_5), \\
c_3 &= (a_6 \cdot b_6) + (a_7 \cdot b_7).
\end{aligned}
$$

I.e. it multiplies the 16-bit integers stored in a 32-bit word and adds these results. Hence, the results are stored as 32-bit entries. This restriction is not a problem since we expect that the inner-products of the list vectors get smaller while the algorithm progresses. We always double-check the inner-product with a regular routine without any size restrictions to catch false-positives: e.g. in case the inner-product of a candidate overflows. This does not increase the computational time noticeably since a reduction of a list vector happens only very infrequently (relative to the total number of computed inner-products).

For each of the pairs $(\mathbf{x}, \mathbf{y})$ we store the following vectors in an SSE register:

$$
\begin{aligned}
\mathbf{X} &= (x_0, x_1, x_2, x_3, x_0, x_1, x_2, x_3), \\
\mathbf{Y}_1 &= (y_0, -y_3, -y_2, -y_1, y_1, y_0, -y_3, -y_2), \\
\mathbf{Y}_2 &= (y_2, y_1, y_0, -y_3, y_3, y_2, y_1, y_0).
\end{aligned}
$$

Now we obtain $\mathbf{z}$ by computing the "multiply and add packed integers" on the pairs $(\mathbf{X}, \mathbf{Y}_1)$ and $(\mathbf{X}, \mathbf{Y}_2)$ after performing the appropriate unpack routines. From this point on, the DFT is stored in SSE registers with 32 bits per entry. Note that the storage of the DFT is asymmetric for $\mathbf{x}$ and $\mathbf{y}$. In our implementation, we generally store the DFT of the sampled vectors in the form of $\mathbf{X}$ and the DFT's of list vectors in the form of $\mathbf{Y}_1$ and $\mathbf{Y}_2$. This way, we always have one representation of each type when comparing sample vectors to the list. The third step is implemented along the lines of [29, Exercise 4.6.4.59]. Negacyclic rotations are performed by using appropriate left and right shifts combined with additions and subtractions.

4.4 A Note on NTRU Lattices

The NTRU cryptosystem [25], published in 1998 long before the appearance of schemes based on R-LWE, uses a similar ring structure. The original NTRU setting works with the ring $R = \mathbb{Z}[X]/(X^n - 1)$. An ideal in R containing an element $a \in R$ also contains all elements $X^i \cdot a(X)$. The corresponding vectors are cyclic rotations of $\mathbf{a}$ and this time there is no sign flip, i.e. $X^n \cdot \mathbf{a} = \mathbf{a}$ and the other properties in Lemma 1 hold in this case as well. The analog of Lemma 3 holds accordingly: Let $\mathbf{a}, \mathbf{b} \in \mathbb{Z}^n$ with corresponding $a, b \in R$ and let $c(X) = a(X) \cdot (X \cdot b^{(R)}(X)) \mod (X^n - 1)$, then $c_i = \langle \mathbf{a}, X^i \cdot \mathbf{b} \rangle$ for $i \in \{0, \ldots, n-1\}$. This means that one can also obtain n scalar products for the price of a single ring product.

Multiplication in R corresponds to the standard cyclic convolution of vectors, and can therefore be carried out with an FFT algorithm. If n is chosen to be a power of 2, we can use Nussbaumer's algorithm for cyclic convolution which recursively calls cyclic and negacyclic convolutions as described above. An implementation using SSE instructions can be done similarly to what we have described in Section 4.3. But in light of Gentry's attack [18] on composite degrees, the parameter n is usually chosen to be prime. In this case, the work by Rader [45] shows that the DFT can still be computed in $\mathcal{O}(n \ln n)$ digit operations. For example, the first approach described by Rader is to separate the computation of the 0-th coefficient from the others and transform the relevant sequences into sequences of length $n - 1$. If $n - 1$ is highly composite or even a power of 2, the DFT can then be computed by convolutions of length $n - 1$ on a permutation of the original sequence and a sequence of corresponding roots of unity. If $n - 1$ itself has large prime factors, a zero-padding approach to some dimension $n' > n$ might be more suitable. A different approach is using discrete weighted transforms as outlined by Crandall and Fagin [12].

5 Experimental results

In order to assess the viability of the algorithmic techniques presented in this work, we created an implementation which allows to be executed concurrently using the message passing interface standard [35]. In this section, we summarize the experiments we ran and discuss the results. First, we discuss experiments with our new variant of the parallel Gauss sieve algorithm (see Section 3.1) in order to establish how it scales when increasing the number of nodes. Next, we consider the ideal lattice variant using the FFT approach (see Section 4) and compare this to an ideal lattice variant using rotations only as well as to the regular (non-ideal) algorithm. All experiments were run on the BlueCrystal Phase 2 cluster of the Advanced Computing Research Centre at the University of Bristol. The nodes in the cluster are equipped with two 2.8 GHz quad-core Intel Harpertown E5462 processors with 8 GB

Table 2. Times in seconds and scaling of our parallel Gauss sieve variant for different dimensions. For a varying number of cores C we state the speedup S with respect to the 8-core setting and the efficiency $E = S \cdot 8/C$.

C	80			88			96		
	t	S	E	t	S	E	t	S	E
8	1999	-	-	31038	-	-	469037	-	-
16	1076	1.9	.95	16834	1.8	.90	264329	1.8	.90
32	661	3.0	.75	10169	3.0	.75	128893	3.6	.90
64	329	6.1	.76	5041	6.1	.76	74700	6.3	.79
96				3493	8.8	.73	55788	8.4	.70
128				2927	10.6	.66	39005	12.0	.75
160				2497	12.4	.62	33011	14.2	.71
192				2096	14.8	.62	26190	17.9	.75
224				2061	15.1	.53	23490	19.9	.71
256				1819	17.1	.53	21223	22.1	.69

RAM per node (1GB per core). The nodes are connected via a QLogic Infinipath high-speed network.

Recently there have been some heuristic improvements to the (serial) GaussSieve [15]. These serial improvements should be fully applicable to our parallel (non-ideal) variant, whereas only the improvements to the sampler can be combined with our DFT-based approach for ideal lattices.

5.1 The New Parallel Gauss Sieve Variant

To benchmark our modified parallel Gauss sieve variant, we ran experiments on Goldstein-Mayer lattices from the SVP challenge [48] of dimensions 80, 88, and 96. To allow for easier comparison with the results from Ishiguro et al. [26], we also used BKZ [49] with block size 30 (as implemented in the fplll library [11]) to preprocess the basis before applying the algorithm. We repeated these experiments using a varying number C of cores, and measured the time t_C in seconds that it took for these C cores to find a vector with norm smaller than the desired bound, i.e. less than 1.05 times the Gaussian heuristic. As the baseline we use 8 cores, as a single node in the cluster contains 8 cores, which corresponds to the "shared memory" case. For each higher number of cores, we computed the *speedup* $S = t_C/t_8$ and the *efficiency* $E = S \cdot 8/C$. Table 2 contains the results for these experiments and Figure 1 shows the plot of the time needed to find a short vector when varying the number of cores C.

In order to assess the performance of our implementation, we compare against the state-of-the-art results presented by Ishiguro et al. [26]. They present results when finding short vectors using their version of the parallel Gauss sieve algorithm for random and ideal lattices on the AmazonEC2 cluster. One "instance" in this cluster contains two Intel Xeon E5-2670 processors (containing two times eight cores and which supports running 32 threads in parallel). We note that this is an unfair comparison, since we use a different (older) computer architecture, have only two times four cores per node and the cluster we used does not support large shared memory, but we think such an exercise can be interesting nonetheless. It requires 0.9 hours to find a short vector in a random 80-dimensional lattice when running on a single AmazonEC2 instance [26]. This can be compared to our runtime of 0.3 hours when using 16 cores (see Table 2). Ishiguro et al. report a run-time of 200 instance hours for finding a short vector in a 96-dimensional random lattice. Our variant of the parallel Gauss sieve algorithm can do this without shared memory in 73.4 hours on 16 cores. These comparisons are not straightforward,

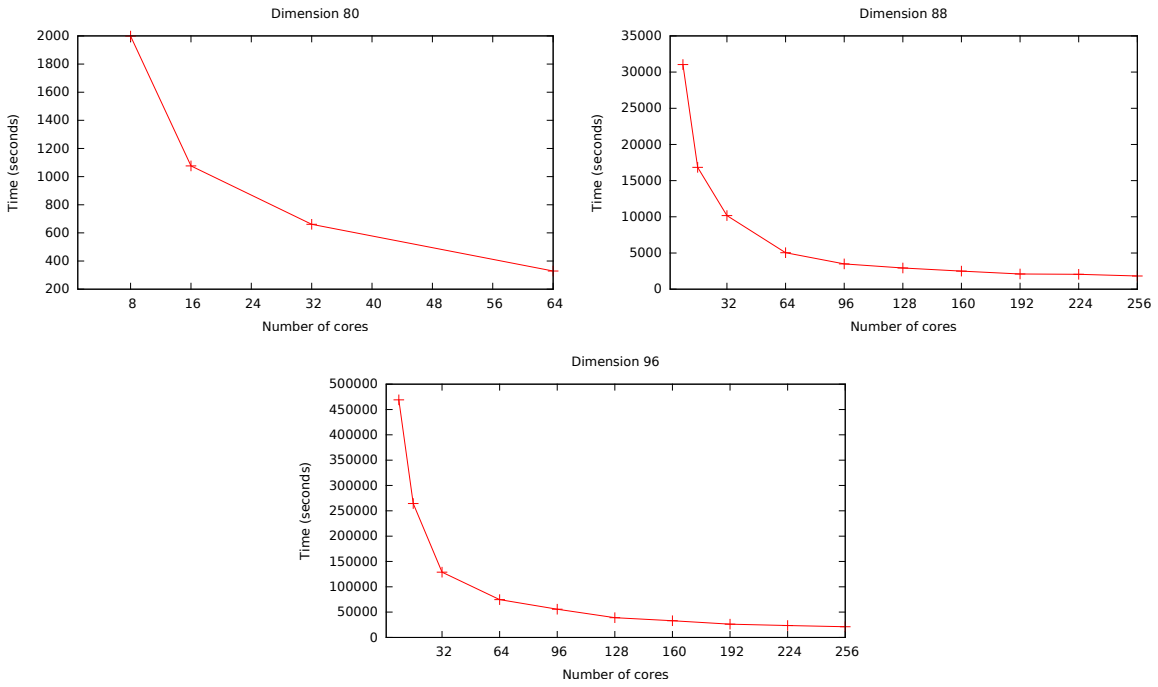

Fig. 1. The time in seconds needed to find a short vector when varying the number of cores for dimensions 80, 88, and 96 when using our parallel Gauss sieve variant.

because our program terminates when a vector with appropriate norm is found (t_{GH}), whereas previous timings terminate once sufficiently many collisions have occurred (t_{Coll}). However, our experiments suggest that the difference between t_{GH} and t_{Coll} becomes smaller as the dimension increases, with $t_{\mathrm{Coll}}/t_{\mathrm{GH}} \leq 1.5$ for the dimensions listed here. Both this and the difference of the computer architecture used can not solely explain this difference in runtime and these performance numbers highlight the potential of our approach which shows a speed-up of up to a factor three, even with recent improvements to the shared-memory approach [4].

Consider the dimension 96 results from Table 2. When using four nodes (32 cores) the communication overhead between the nodes is relatively low: we see a factor 3.6 reduction in wall-clock time compared to the single-node setting, while we can hope for at most a factor four speed-up. This communication overhead becomes more prominent when using much more compute nodes. When running on 256 cores we still get a factor 22.1 reduction in wall-clock time while we can hope for a maximum factor of 32. We expect the communication overhead to reduce significantly when running the experiments on clusters with a large shared memory like the AmazonEC2 cluster.

5.2 The Ideal Lattice Setting

To benchmark the ideal Gauss sieve variant, we ran experiments on a power of two cyclotomic lattice of dimension 128 from the ideal lattice challenge [41]. As before, we used BKZ with block size 30 to preprocess the basis, which is consistent with the work of Ishiguro et al. [26]. We aim to measure the difference between our regular parallel variant using no ideal-specific techniques (see Section 3.1), a version which additionally uses information about the rotations (see the function `ReduceRot` in Algorithm 3 and Lemma 1) and a version which uses our

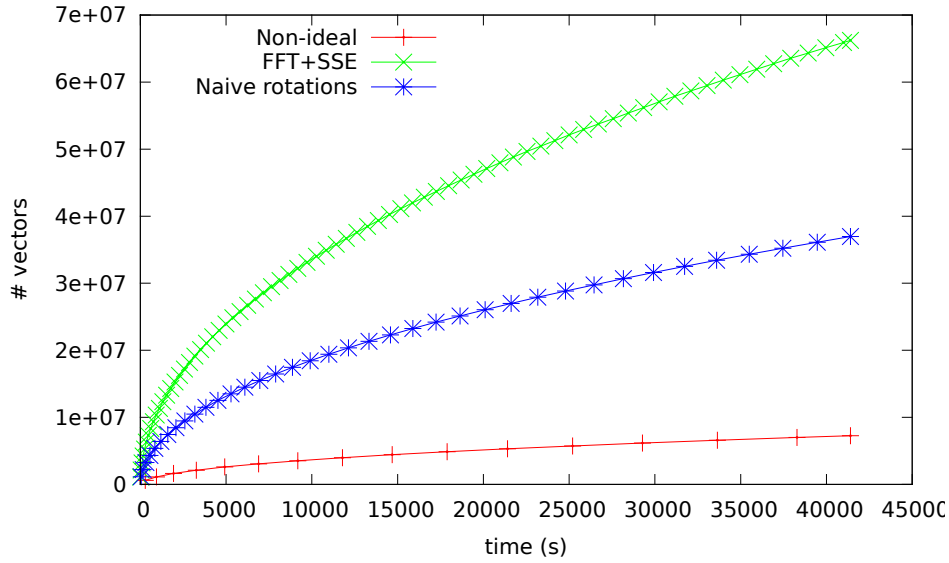

Fig. 2. The number of vectors in the list against the running time in a power of two cyclotomic ideal lattice of dimension 128 for three parallel Gauss sieve variants.

FFT approach using SSE instructions (see Section 4.3). While fixing the number of compute cores to 64, we measure how many vectors are in the (global) list after a certain time. This characteristic is useful since the size of list provides information on how far the algorithm has proceeded.

Figure 2 shows the results of these experiments. Note that for the versions that use ideal-specific techniques, the "list" only stores one vector for each 128 vectors in the system, because all the rotations of vectors are taken into account as well. The number of vectors printed in the graph is the complete number of vectors in the system which includes rotations, i.e., the total number of vectors that are all pairwise Gauss reduced. Another important observation is that both algorithms that use the ideal speed-ups are performing functionally *exactly the same* operations on vectors in terms of which vectors are being reduced. This means that from the same random sampled vectors, these algorithms will construct identical lists. This makes a comparison between the two ideal-lattice approaches more meaningful.

Figure 2 clearly shows the advantage of the approaches which use the ideal structure over the regular algorithm. While the regular approach has accumulated a list-size of $7.3 \cdot 10^6$ after 41 400 seconds, this many vectors are represented by the approach which uses rotations after 1605 seconds (25.8 times faster) and the FFT approach only requires 470 seconds (88.1 times faster). Hence, the FFT approach outperforms the ideal approach using rotations only by a factor 3.4. Interestingly, the gap between the two ideal approaches seems to widen as the number of vectors goes up. This may be attributed to the fact that when fewer vectors are in the system, a relatively larger proportion of the time is spent on converting vectors using FFT, whereas later on the dominating cost is the reverse FFT operation. After eleven and a half hours (41 400 seconds) the version using only rotations had completed 19 428 rounds, and had 288 783 vectors in its system, representing $288\,783 \cdot 128$ vectors in total, whereas the version using FFT achieved the same in about three hours and twenty minutes, or 12 000 seconds, which corresponds to a speed-up of a factor 3.75.

Finally, we applied our FFT implementation to the same SVP ideal lattice challenge of dimension 128 [41] that Ishiguro et al. [26] solved. For this, we used the BlueCrystal Phase 3 cluster of the Advanced Computing Research Centre at the University of Bristol. The central processing units are identical to the ones used in [26], which makes a comparison of the two approaches more meaningful. However, we did not have access to large shared memory pools among the computing nodes. Again, we used BKZ with block size 30 to preprocess the basis obtained from the ideal lattice challenge. We ran our software on 64 nodes (1024 cores) and found a short vector after 750 478 seconds wall-clock time (8.69 days). The short vector found is a rotation of the short vector presented in [26]. In [26], this short vector was obtained after computing 14.88 days on 1344 cores (where each core had a significant amount of memory in order to duplicate the entire list among all cores). Our computation required 24.4 core years, is thus more than twice as fast as the computation carried out in [26], and shows the practical advantages of our approach and implementation on conventional compute clusters.

6 Discussion

What should the reader take away from these results? Lattice cryptanalysis is a complex subject, with different types of algorithms that are combined to find short vectors in lattices. The most recent results of the SVP challenge [48] suggest that in currently tractable dimensions, a combination of BKZ and (some variant of) enumeration appears to give the best results. However, it is not possible to reproduce and verify most of those results, because too many details of the experiments are not public: including, and most importantly, in most cases the source code used. This makes comparing the different algorithms and their results difficult.

In this paper, we have shown that for sieving algorithms such as the Gauss sieve, there are additional techniques to speed up the search for short vectors in cyclic and negacyclic lattices. Given the current (unverified) performance details by various results on the SVP challenge [48], it seems that our speed-ups are not enough to defeat other SVP-solvers such as enumeration in tractable dimensions. However, cryptographic applications are not designed to utilize such relatively small dimensions. The asymptotic run-times of the various algorithms for solving the shortest vector problem (see Table 1) show that sieving-based algorithms will eventually become faster than enumeration based approaches. This paper aims to lower this cross-over point in the power of two cyclotomic ideal case by making use of this special structure; something which does not appear to aid the enumeration based algorithms. Studying when this cross-over point occurs is an important question for the security assessment of lattice-based cryptographic schemes and currently remains an open question. We hope this paper aids future research to pinpoint this cross-over point in order to determine which type of algorithm is most suitable to find short vectors for dimensions considered in lattice-based cryptography nowadays.

Acknowledgements

The third author's work has been supported in part by EPSRC via grant EP/I03126X. We thank Nigel Smart for his comments on an earlier version of the paper.

References

1. M. Ajtai. Generating hard instances of lattice problems (extended abstract). In G. L. Miller, editor, *STOC*, pages 99–108. ACM, 1996.

2. M. Ajtai and C. Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In F. T. Leighton and P. W. Shor, editors, *STOC*, pages 284–293. ACM, 1997.
3. M. Ajtai, R. Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In J. S. Vitter, P. G. Spirakis, and M. Yannakakis, editors, *STOC*, pages 601–610. ACM, 2001.
4. S. T. Artur Mariano and C. Bischof. Lock-free gausssieve for linear speedups in parallel high performance svp calculation. Cryptology ePrint Archive, Report 2014/775, 2014. `http://eprint.iacr.org/`.
5. R. E. Bansarkhani and J. Buchmann. Improvement and efficient implementation of a lattice-based signature scheme. In T. Lange, K. Lauter, and P. Lisonek, editors, *Selected Areas in Cryptography*, volume 8282 of *LNCS*, pages 48–67. Springer, 2013.
6. A. Becker, N. Gama, and A. Joux. A sieve algorithm based on overlattices. *LMS Journal of Computation and Mathematics*, 17:49–70, 1 2014.
7. J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. Cryptology ePrint Archive, Report 2014/599, 2014. `http://eprint.iacr.org/`.
8. J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In M. Stam, editor, *IMA Cryptography and Coding*, volume 8308 of *LNCS*, pages 45–64. Springer, 2013.
9. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In S. Goldwasser, editor, *Innovations in Theoretical Computer Science*, pages 309–325. ACM, 2012.
10. Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In P. Rogaway, editor, *CRYPTO*, volume 6841 of *LNCS*, pages 505–524. Springer, 2011.
11. D. Cadé, X. Pujol, and D. Stehlé. fplll library, version 4.0.4. Available at `http://perso.ens-lyon.fr/damien.stehle/fplll/`, 2013.
12. R. Crandall and B. Fagin. Discrete weighted transforms and large-integer arithmetic. *Mathematics of Computation*, 62(205):305–324, 1994.
13. R. Crandall and C. Pomerance. *Prime numbers: a computational perspective*. Springer, second edition, 2005.
14. U. Fincke and M. Pohst. Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, 44(170):pp. 463–471, 1985.
15. R. Fitzpatrick, C. Bischof, J. Buchmann, O. Dagdelen, F. Gopfert, A. Mariano, and B.-Y. Yang. Tuning gausssieve for speed. Cryptology ePrint Archive, Report 2014/788, 2014. `http://eprint.iacr.org/`.
16. N. Gama, P. Q. Nguyen, and O. Regev. Lattice enumeration using extreme pruning. In H. Gilbert, editor, *EUROCRYPT*, volume 6110 of *LNCS*, pages 257–278. Springer, 2010.
17. S. Garg, C. Gentry, and S. Halevi. Candidate multilinear maps from ideal lattices. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *LNCS*, pages 1–17. Springer, 2013.
18. C. Gentry. Key recovery and message attacks on NTRU-composite. In B. Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *LNCS*, pages 182–194. Springer, 2001.
19. C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.
20. C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO*, volume 7417 of *LNCS*, pages 850–867. Springer, 2012.
21. C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *STOC*, pages 197–206. ACM, 2008.
22. N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. A. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In E. Prouff and P. Schaumont, editors, *CHES*, volume 7428 of *LNCS*, pages 512–529. Springer, 2012.
23. T. Güneysu, T. Oder, T. Pöppelmann, and P. Schwabe. Software speed records for lattice-based signatures. In P. Gaborit, editor, *Post-Quantum Cryptography*, volume 7932 of *LNCS*, pages 67–82. Springer-Verlag, 2013.
24. G. Hanrot and D. Stehlé. Improved analysis of Kannan's shortest lattice vector algorithm. In A. Menezes, editor, *CRYPTO*, volume 4622 of *LNCS*, pages 170–186. Springer, 2007.
25. J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In J. Buhler, editor, *ANTS-III*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998.
26. T. Ishiguro, S. Kiyomoto, Y. Miyake, and T. Takagi. Parallel Gauss sieve algorithm: Solving the SVP challenge over a 128-dimensional ideal lattice. In H. Krawczyk, editor, *PKC*, volume 8383 of *LNCS*, pages 411–428. Springer, 2014.

27. G. A. Kabatiansky and V. I. Levenshtein. On bounds for packings on a sphere and in space. *Problemy Peredachi Informatsii*, 14(1):3–25, 1978.
28. R. Kannan. Improved algorithms for integer programming and related lattice problems. In *STOC*, pages 193–206. ACM, 1983.
29. D. E. Knuth. *Seminumerical Algorithms*. The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts, USA, 3rd edition, 1997.
30. A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.
31. T. Lepoint and M. Naehrig. A comparison of the homomorphic encryption schemes FV and YASHE. In D. Pointcheval and D. Vergnaud, editors, *AFRICACRYPT*, volume 8469 of *LNCS*, pages 318–335. Springer, 2014.
32. V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. SWIFFT: A modest proposal for FFT hashing. In K. Nyberg, editor, *FSE*, volume 5086 of *LNCS*, pages 54–72. Springer, 2008.
33. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In H. Gilbert, editor, *EUROCRYPT*, volume 6110 of *LNCS*, pages 1–23. Springer, 2010.
34. V. Lyubashevsky, C. Peikert, and O. Regev. A toolkit for ring-LWE cryptography. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *LNCS*, pages 35–54. Springer, 2013.
35. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0. Standard 3, Message Passing Interface Forum, 2012.
36. D. Micciancio and O. Regev. Lattice-based cryptography. In D. J. Bernstein and J. Buchmann, editors, *Post-quantum Cryptography*. Springer, 2008.
37. D. Micciancio and P. Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations. In L. J. Schulman, editor, *STOC*, pages 351–358. ACM, 2010.
38. B. Milde and M. Schneider. A parallel implementation of GaussSieve for the shortest vector problem in lattices. In V. Malyshkin, editor, *PaCT*, volume 6873 of *LNCS*, pages 452–458. Springer, 2011.
39. P. Q. Nguyen and T. Vidick. Sieve algorithms for the shortest vector problem are practical. *J. Mathematical Cryptology*, 2(2):181–207, 2008.
40. H. J. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(2):205–215, 1980.
41. T. Plantard and M. Schneider. Ideal lattice challenge. `http://latticechallenge.org/ideallattice-challenge/index.php`, 2012.
42. T. Pöppelmann and T. Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In T. Lange, K. Lauter, and P. Lisonek, editors, *Selected Areas in Cryptography*, volume 8282 of *LNCS*, pages 68–85. Springer, 2013.
43. X. Pujol and D. Stehlé. Rigorous and efficient short lattice vectors enumeration. In J. Pieprzyk, editor, *ASIACRYPT*, volume 5350 of *LNCS*, pages 390–405. Springer, 2008.
44. 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/`.
45. C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968.
46. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In H. N. Gabow and R. Fagin, editors, *STOC*, pages 84–93. ACM, 2005.
47. M. Schneider. Sieving for shortest vectors in ideal lattices. In A. Youssef, A. Nitaj, and A. E. Hassanien, editors, *AFRICACRYPT*, volume 7918 of *LNCS*, pages 375–391. Springer, 2013.
48. M. Schneider and N. Gama. SVP challenge. `http://latticechallenge.org/svp-challenge/index.php`, 2010.
49. C. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(1-3):181–199, 1994.
50. D. Stehlé and R. Steinfeld. Making ntru as secure as worst-case problems over ideal lattices. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *LNCS*, pages 27–47. Springer, 2011.
51. J. van de Pol and N. P. Smart. Estimating key sizes for high dimensional lattice-based systems. In M. Stam, editor, *IMA Cryptography and Coding*, volume 8308 of *LNCS*, pages 290–303. Springer, 2013.
52. P. Voulgaris. gsieve library. Available at `http://cseweb.ucsd.edu/~pvoulgar/impl.html`, 2011.
53. P. Voulgaris and D. Micciancio. Faster exponential time algorithms for the shortest vector problem. *Electronic Colloquium on Computational Complexity (ECCC)*, 16:65, 2009.
54. X. Wang, M. Liu, C. Tian, and J. Bi. Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. In B. S. N. Cheung, L. C. K. Hui, R. S. Sandhu, and D. S. Wong, editors, *ASIACCS*, pages 1–9. ACM, 2011.

55. F. Zhang, Y. Pan, and G. Hu. A three-level sieve algorithm for the shortest vector problem. In T. Lange, K. Lauter, and P. Lisonek, editors, *Selected Areas in Cryptography*, volume 8282 of *LNCS*, pages 29–47. Springer, 2013.