The General Sieve Kernel and New Records in Lattice Reduction

Martin R. Albrecht[1], Léo Ducas[2], Gottfried Herold[3], Elena Kirshanova[3], Eamonn W. Postlethwaite[1], Marc Stevens[2⋆]

[1] Information Security Group, Royal Holloway, University of London
[2] Cryptology Group, CWI, Amsterdam, The Netherlands
[3] ENS Lyon

Abstract. We propose the General Sieve Kernel (G6K, pronounced /ʒe.si.ka/), an abstract stateful machine supporting a wide variety of lattice reduction strategies based on sieving algorithms. Using the basic instruction set of this abstract stateful machine, we first give concise formulations of previous sieving strategies from the literature and then propose new ones. We then also give a light variant of BKZ exploiting the features of our abstract stateful machine. This encapsulates several recent suggestions (Ducas at Eurocrypt 2018; Laarhoven and Mariano at PQCrypto 2018) to move beyond treating sieving as a blackbox SVP oracle and to utilise strong lattice reduction as preprocessing for sieving. Furthermore, we propose new tricks to minimise the sieving computation required for a given reduction quality with mechanisms such as recycling vectors between sieves, on-the-fly lifting and flexible insertions akin to Deep LLL and recent variants of Random Sampling Reduction.

Moreover, we provide a highly optimised, multi-threaded and tweakable implementation of this machine which we make open-source. We then illustrate the performance of this implementation of our sieving strategies by applying G6K to various lattice challenges. In particular, our approach allows us to solve previously unsolved instances of the Darmstadt SVP (151, 153, 155) and LWE (e.g. (75, 0.005)) challenges. Our solution for the SVP-151 challenge was found 400 times faster than the time reported for the SVP-150 challenge, the previous record. For exact SVP, we observe a performance crossover between G6K and FPLLL's state of the art implementation of enumeration at dimension 70.

Keywords: cryptanalysis, lattice reduction, sieving, SVP, LWE, BKZ.

⋆ The research of MA was supported by EPSRC grants EP/P009417/1, EP/S02087X/1 and by the European Union Horizon 2020 Research and Innovation Program Grant 780701; the research of LD was supported by a Veni Innovational Research Grant from NWO under project number 639.021.645 and by the European Union Horizon 2020 Research and Innovation Program Grant 780701; the research of EP was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1).

1 Introduction

Sieving algorithms have seen remarkable progress over the last few years. Briefly, these algorithms find a shortest vector in a lattice by considering exponentially many lattice vectors and searching for sums and differences that produce shorter vectors. Since the introduction of sieving algorithms in 2001 [AKS01], a long series of works, e.g. [MV10b, BGJ15, HK17], have proposed asymptotically faster variants; the asymptotically fastest of which has a heuristic time complexity of $2^{0.292d+o(d)}$, with d the dimension of the lattice [BDGL16].

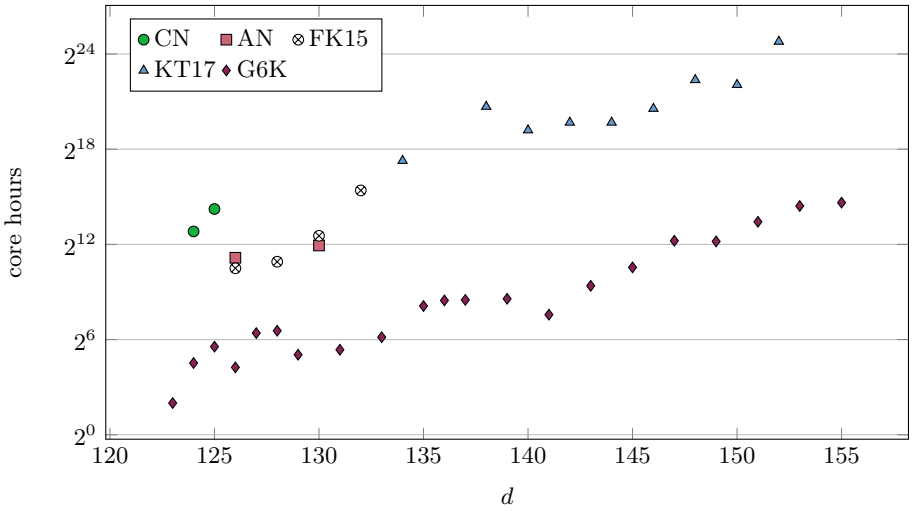
Such algorithms for finding short vectors are used in lattice reduction algorithms. These improve the "quality" of a lattice basis (see Section 2) and are used in the cryptanalysis of lattice-based cryptography.

On the other hand, lattice reduction libraries such as [dt18a, AWHT16] implement enumeration algorithms, which also find a shortest vector in a lattice. These algorithms perform an exhaustive search over all lattice points within a given target radius by exploiting the properties of projected sublattices. Enumeration has a worst-case time complexity of $d^{\frac{1}{2e}d+o(d)}$ [Kan83, HS07] but requires only polynomial memory.

While, with respect to running time, sieving already compares favourably in relatively low dimensions to simple enumeration[4] (Fincke–Pohst enumeration [FP85] without pruning), the Darmstadt Lattice Challenge Hall of Fame for both approximate SVP [SG10] and LWE [FY15] challenges has been dominated by results obtained using enumeration. Sieving has therefore not, so far, been competitive in practical dimensions when compared to state of the art enumeration with heavy preprocessing [Kan83, MW15] and (extreme) pruning [GNR10] as implemented in e.g. FPLLL/FPyLLL [dt18a, dt18b]. Here, "pruning" means to forego exploring the full search space in favour of focussing on likely candidates. The extreme pruning variant proceeds by further shrinking the search space, and rerandomising the input and restarting the search on failure. In this context "heavy preprocessing" means running strong lattice reduction, such as the BKZ algorithm [Sch87, CN11], which in turn runs enumeration in smaller dimensions, before performing the full enumeration. In short, enumeration currently beats sieving "in practice" despite having asymptotically worse running time. Thus [MW15], relying on the then state of the art, estimated the crossover point between sieving and enumeration for solving the Shortest Vector Problem (SVP) as dimension $d = 146$ (or in the thousands, assuming extreme pruning can be combined with heavy preprocessing without loss of performance).

Contribution. In this work, we report performance records for achieving various lattice reduction tasks using sieving. For exact-SVP, we are able to outperform the pruned enumeration of FPLLL/FPyLLL by dimension 70. For Darmstadt SVP challenges (1.05-Hermite-SVP) we solve previously unsolved challenges in dimensions $\{151, 153, 155\}$ (see Figure 1 and Table 2), and our running

[4] For example, the Gauss sieve implemented in FPLLL (`latsieve`) beats its *unpruned* SVP oracle (`fplll -a svp`) in dimension 50.



CN: Chen & Nguyen (HoF), BKZ+enum; AN: Aono & Nguyen (HoF), BKZ+enum; FK15: [FK15], RSR; KT17: [TKH18][5], RSR; G6K: `WorkOut` with `bgj1`-sieve (this work). "HoF" means data was extracted from the Darmstadt SVP Challenge Hall of Fame [SG10]. Raw data embedded.

Fig. 1: New Darmstadt SVP challenges records.

times are at least 400 times smaller than the previous records for comparable instances.

We also solved new instances $(n, \alpha) \in \{(40, 0.005), (50, 0.015), (55, 0.015), (60, 0.01), (65, 0.01), (75, 0.005)\}$ of the Darmstadt LWE challenge (see Table 3). For this, we adapted the strategy of [LN13], which consists of running one large enumeration after a BKZ tour of small enumerations, to G6K. This improves slightly upon the prediction of [ADPS16, AGVW17].

Our sieving performance is enabled by building on, generalising and extending previous works. In particular, the landscape of enumeration and sieving started to change recently with [Duc18a, LM18]. For example, [Duc18a] speculated that the crossover point, for solving SVP, between the SubSieve proposed there and pruned enumeration would be around $d = 90$ if combined with faster sieving than [MV10b]. A key ingredient for this performance gain was the realisation of several "dimensions for free" by utilising heavy preprocessing and Babai lifting (or size reduction) in said free dimensions. This may be viewed as a hybrid of pruned enumeration with sieving, and is enabled by strong lattice reduction preprocessing. In other words, we may consider these improvements as applying lessons learnt from enumeration to sieving algorithms. It is worth recalling here

[5] Their latest record (SVP-152) from Oct. 2018 is only reported in the HoF. It reports a computation time of 800K CPU-hours. According to personal communications with the authors, this translates to $36 \cdot 800\text{K} = 28.8\text{M}$ core-hours.

that the fastest enumeration algorithm relies on the input basis being quasi-HKZ reduced [Kan83], but prior to [Duc18a, LM18] sieving was largely oblivious to the quality of the input basis. Furthermore, both [Duc18a, LM18] suggest exploiting the fact that sieving algorithms hold a database of many short vectors, for example by recycling them in future sieving steps. Thus, instead of treating sieving as an SVP oracle outputting a single vector, they implicitly treat it as a stateful machine where the state comprises the current basis and a database of many relatively short vectors.

G6K, an abstract stateful machine. In this work, we embrace and push forward in this direction. After some preliminaries in Section 2, we propose the General Sieve Kernel (G6K, pronounced /ʒe.si.ka/) in Section 3, an abstract machine for running sieving algorithms, and driving lattice reduction. We define several basic instructions on this stateful machine that not only allow new sieving strategies to be simply expressed and easily prototyped, but also lend themselves to the easy inclusion and extension of previous works. For example, the progressive sieves from [Duc18a, LM18] can be concisely written as

$$\texttt{Reset}_{0,0,0},\ (\texttt{ER},\ \texttt{S})^{d},\ \texttt{I}_0$$

where `S` means to sieve, $\texttt{I}_0$ means to insert the shortest vector found into the basis, `ER` means to increase the sieving dimension and `Reset` initialises the machine.

Beyond formalising previous techniques, our machine provides new instructions, namely `EL`, which allows one to increase the sieving dimension "towards the left" (of the basis), and an insertion instruction `I` which is no longer terminal: it is possible to resieve after an insertion, contrary to [Duc18a]. These instructions increase the range of implementable strategies and we make heavy use of them to achieve the above results.

The General Sieve Kernel also introduces new tricks to further improve efficiency. First, all vectors encountered during the sieve can be lifted "on the fly" (as opposed to only the final set of vectors in [Duc18a]) offering a few extra dimensions for free and thus improved performance. Additionally, G6K keeps insertion candidates for many positions so as to allow a posteriori choices of the most reducing insertion, akin to Deep LLL [SE94] and the latest variants of Random Sampling Reduction (RSR) [TKH18], enabling stronger preprocessing.

Lattice reduction with G6K. Using these instructions, in Section 4 we then create reduction strategies for various tasks (SVP, BKZ-like reduction). These strategies encapsulate and extend the contributions and the suggestions made in [Duc18a, LM18], further exploiting the features of G6K. Using the instructions of our abstract stateful machine, our fundamental operation, named the `Pump`, may be written as

$$\texttt{Reset}_{\kappa,\kappa+\beta,\kappa+\beta},\ (\texttt{EL},\ \texttt{S})^{\beta-f},\ (\texttt{I},\ \texttt{S}^{s})^{\beta-f}.$$

While previous works mostly focus on recursive lattice reduction within sieving, we also explicitly treat and test utilising sieving within the BKZ algorithm.

Here, we report both negative and positive results. On the one hand, we report that, at least in our implementation, the elegant idea of a sliding-window sieve for BKZ [LM18] performs poorly and offer a discussion as to why. We also find that the strategy from [Duc18a], consisting of "overshooting" the block size β of BKZ by a small additive factor combined with "jumping" over the same number of indices in a BKZ tour, does not provide a beneficial quality vs. time trade-off. On the other hand, we find that for the second and following indices of a BKZ tour, cheaper sieving calls (involving less preprocessing) suffice and that opportunistically increasing the number of dimensions for free beyond the optimal values for solving SVP improves the quality vs. time trade-off. Thus, we vindicate the suggestion to move beyond treating sieving merely as an SVP oracle in BKZ.

Implementation. In Section 5, we then propose and describe an open-source, tweakable, multi-threaded, low-level optimised implementation of G6K, featuring several sieve variants [MV10b, BGJ15, HK17].[6] Our implementation is carefully optimised to support multiple cores in all time consuming operations, is highly parameterised and makes heavy use of the SimHash test [Cha02, FBB+15, Duc18a]. It combines a C++ kernel with a Python control module. Thus, our higher level algorithms are all implemented in Python for easy experimentation. Our implementation is written with a view towards being extensible and reusable and comes with documentation and tests. We consider hackable and usable software a contribution in its own right.

Performance and Records. Using and tuning our implementation of G6K then allows us to obtain the variety of performance records for solving lattice challenges as described above. We describe our approach in Section 6. There, we also describe our experiments for the aforementioned BKZ strategies.

Complementary information on the performance of our implementation is provided in appendices: Appendix A gives a feature-by-feature improvement report, and Appendix B assesses the parallelism efficiency.

Discussion. A natural question is how our results affect the security of lattice-based schemes, especially the NIST PQC candidates. Most candidates have been extremely conservative, and thus we do not expect the classical security claim of any scheme to be directly affected by our results. We note, however, that our results on BKZ substantiate further the prediction made in several analyses of NIST PQC candidates that the cost of the SVP oracle can be somewhat amortised in BKZ [PAA+17, Sec 4.2.6]. Thus, our results provide further evidence that the $8\,d \cdot C_{SVP}$ cost model [ACD+18] is an over-estimate,[7] but they nevertheless do not reach the lower bound given by the "core-hardness" estimates. However, we stress that our work justifies the generally conservative approach

[6] Our implementation is available at https://github.com/fplll/g6k/.

[7] Note that, in addition, this already follows in the enumeration regime from [LN13] which we adapt to the sieving regime in Section 6.

and we warn against security estimates based on a state of the art that is still in motion.

On the other hand, the memory consumption of sieving eventually becomes a difficult issue for implementation, and could incur slowdowns due to memory access delays and bandwidth constraints. Though, it is not so clear that these difficulties are insurmountable, especially to an attacker having access to custom hardware. For example Kirchner claimed [Kir16] that simple sieving algorithms such as the Nguyen–Vidick sieve are implementable by a circuit with Area = Time = $2^{0.2075n+o(n)}$. Ducas further conjectured [Duc18b] that `bgj1` (a simplified version of [BGJ15]) can be implemented with Area = $2^{0.2075n+o(n)}$ and Time = $2^{0.142n+o(n)}$. More concretely, the algorithms that we have implemented mostly consider contiguous streams of data, making the use of disks instead of RAM plausibly not so penalising.

One may also argue that such an area requirement on its own is already unreasonable. Yet, such arguments should also account for what amount of wall-time is considered reasonable. For example, the walltime of a bruteforce search costing 2^{128} CPU-cycles on 2^{64} cores at 4GHz runs for 2^{64} cycles = 2^{32} seconds $\approx$ 134 years; larger walltimes with fewer cores can arguably be considered irrelevant for practical attacks.

2 Preliminaries

2.1 Notations and Basic Definitions

We start counting at zero. All vectors are denoted by bold lower case letters and are to be read as column vectors. Matrices are denoted by bold capital letters. We write a matrix $\mathbf{B}$ as $\mathbf{B} = (\mathbf{b}_0, \ldots, \mathbf{b}_{n-1})$ where $\mathbf{b}_i$ is the i-th column vector of $\mathbf{B}$. We may also denote $\mathbf{b}_i$ by $\mathbf{B}[i]$ and the j-th entry of $\mathbf{b}_i$ by $\mathbf{B}[i,j]$. If $\mathbf{B} \in \mathbb{R}^{d\times n}$ has full column rank n, the lattice $\mathcal{L}$ generated by the basis $\mathbf{B}$ is denoted by $\mathcal{L}(\mathbf{B}) = \{\mathbf{Bx} \,|\, \mathbf{x} \in \mathbb{Z}^n\}$. We denote by $(\mathbf{b}_0^*, \ldots, \mathbf{b}_{n-1}^*)$ the Gram–Schmidt orthogonalisation of the matrix $\mathbf{B} = (\mathbf{b}_0, \ldots, \mathbf{b}_{n-1})$. That is, we define

$$\mu_{i,j} = \frac{\langle \mathbf{b}_j^*, \mathbf{b}_i \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle} \qquad \text{and} \qquad \mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=0}^{i-1} \mu_{i,j} \cdot \mathbf{b}_j^*.$$

The process of updating $\mathbf{b}_i \leftarrow \mathbf{b}_i - \lfloor \mu_{ij} \rceil \mathbf{b}_j$, for $j \in \{i-1, \ldots, s\}$ with $0 \le s < i$, is known as "size reduction" or "Babai's Nearest Plane" algorithm. We also define $\mathbf{b}_i^\circ = \mathbf{b}_i^* / \langle \mathbf{b}_i^*, \mathbf{b}_i^* \rangle$ and extend this to $\mathbf{B}^\circ$ column wise. For $i \in \{0, \ldots, n-1\}$, we denote the projection orthogonally to the span of $(\mathbf{b}_0, \ldots, \mathbf{b}_{i-1})$ by π_i. For $0 \le \ell < r \le n$, we denote by $\mathbf{B}_{[\ell:r]}$ the local projected basis, $(\pi_\ell(\mathbf{b}_\ell), \ldots, \pi_\ell(\mathbf{b}_{r-1}))$, and when the basis is clear from context, by $\mathcal{L}_{[\ell:r]}$ the lattice generated by $\mathbf{B}_{[\ell:r]}$. We refer to the *left* (resp. the *right*) of a *context* $[\ell : r]$ and by "the context $[\ell : r]$" implicitly refer also to $\mathcal{L}_{[\ell:r]}$ and $\mathbf{B}_{[\ell:r]}$. More generally, we speak of the *left* (resp. the *right*) as a direction to refer to smaller (resp. larger) indices and of contexts becoming larger as $r - l$ grows.

The Euclidean norm of a vector $\mathbf{v}$ is denoted by $|\mathbf{v}|$. The volume of a lattice $\mathcal{L}(\mathbf{B})$ is $\mathrm{Vol}(\mathcal{L}(\mathbf{B})) = \prod_i |\mathbf{b}_i^*|$, an invariant of the lattice. The first minimum of a lattice $\mathcal{L}$ is the length of a shortest non-zero vector, denoted by $\lambda_1(\mathcal{L})$. We use the abbreviations $\mathrm{Vol}(\mathbf{B}) = \mathrm{Vol}(\mathcal{L}(\mathbf{B}))$ and $\lambda_1(\mathbf{B}) = \lambda_1(\mathcal{L}(\mathbf{B}))$.

2.2 Sieving, Lattice Reduction and Heuristics

Sieving algorithms build databases of lattice vectors, exponentially sized in the lattice dimension. In the simplest sieves, it is checked whether the sums or differences of any pair of database vectors is shorter than one of the summands or differands. More importantly for G6K as an abstract stateful machine is the property of sieving [NV08, MV10b] that, after sieving in some $\mathcal{L}$, this database contains a constant fraction, which we are able to set, of $\{\mathbf{w} \in \mathcal{L} \colon |\mathbf{w}| \leq R \cdot \mathrm{gh}(\mathcal{L})\}$. Here $\mathrm{gh}(\mathcal{L})$ is the expected length of the shortest vector of a lattice $\mathcal{L}$ (see Definition 2), and R is a small constant determined by the sieve (see Section 5.1). It is this information that G6K will leverage when changing context and inserting.

Lattice reduction is the process of taking a basis for a given $\mathcal{L}$ and finding subsequent bases of $\mathcal{L}$ with shorter and closer to orthogonal vectors. Two important notions of reduction are HKZ and BKZ-β reduction. The BKZ algorithm [SE94, CN11] takes as input a lattice basis of $\mathcal{L}$ and a block size β and outputs a BKZ-β reduced basis of $\mathcal{L}$.

Definition 1 (Hermite–Korkine–Zolotarev, Block-Korkine–Zolotarev). *A size-reduced basis $\mathbf{B} = (\mathbf{b}_0, \ldots, \mathbf{b}_{d-1})$ of a lattice $\mathcal{L}$ is Hermite–Korkine–Zolotarev (HKZ) reduced if $|\mathbf{b}_i^*| = \lambda_1(\mathcal{L}_{[i:d]}), \forall\, i < d$. It is Block-Korkine–Zolotarev with block size β (BKZ-β) reduced if $|\mathbf{b}_i^*| = \lambda_1(\mathcal{L}_{[i:\min\{i+\beta,d\}]}), \forall\, i < d$.*

Intuitively BKZ reduction requires that a given index in the basis is as short as possible when considering only a local projected sublattice, with the locality parameterised by β. The cost of BKZ increases with β. The LLL algorithm [LLL82] can be thought of as BKZ-2 and is often used as a cheap starting point for lattice reduction. Equally, HKZ reduction can be thought of as BKZ-d and is a strong notion of reduction.

The BKZ algorithm internally calls an SVP oracle in dimension $\leq \beta$, i.e. an algorithm that solves the Shortest Vector Problem (or an approximate variant of it) in dimension β.

The Gaussian heuristic predicts that the number, $|\mathcal{L} \cap \mathcal{B}|$, of lattice points inside a measurable body $\mathcal{B} \subset \mathbb{R}^n$ is approximately $\mathrm{Vol}(\mathcal{B})/\mathrm{Vol}(\mathcal{L})$. Applied to Euclidean n-balls, it leads to the following prediction of $\lambda_1(\mathcal{L})$ for a given $\mathcal{L}$.

Definition 2 (Gaussian Heuristic). *We denote by $\mathrm{gh}(\mathcal{L})$ the expected first minimum of a lattice $\mathcal{L}$ according to the Gaussian heuristic. For a full rank lattice $\mathcal{L} \subset \mathbb{R}^d$, it is given by*

$$\mathrm{gh}(\mathcal{L}) = \sqrt{d/2\pi e} \cdot \mathrm{Vol}(\mathcal{L})^{1/d}. \tag{1}$$

The quality of a basis after lattice reduction can be measured by a quantity called the root Hermite factor.

Definition 3 (Root Hermite Factor). *For a basis* $\mathbf{B}$ *of a* d*-dimensional lattice, the root Hermite factor is defined as*

$$\delta = \left(|\mathbf{b}_0| \,/\, \mathrm{Vol}\,(\mathbf{B})^{1/d} \right)^{1/d}. \tag{2}$$

For BKZ-β, the root Hermite factor is a well behaved quantity. For small blocksizes the root Hermite factor is experimentally calculated [GN08b] and for larger blocksizes [Che13] it follows the asymptotic formula

$$\delta(\beta)^{2(\beta-1)} = (\beta/(2\pi e))(\beta\pi)^{\frac{1}{\beta}}, \tag{3}$$

which tends towards 1. Finally we reproduce the Geometric Series Assumption (GSA) [Sch03] which, given β, heuristically determines the lengths of consecutive Gram–Schmidt basis vectors. It is reasonably accurate for $\beta > 50$ and $\beta \ll d$ [Ngu10, CN11, YD17].

Definition 4 (Geometric Series Assumption). *Let* $\mathbf{B}$ *be a BKZ-*β *reduced basis, then the Geometric Series Assumption states that* $|\mathbf{b}_i^*| \approx \delta(\beta)^{-2} \left|\mathbf{b}_{i-1}^*\right|$.

3 The General Sieve Kernel

3.1 Design Principles

In this section we propose the General Sieve Kernel (Version 1.0), an abstract machine supporting a wide variety of lattice reduction strategies based on sieving algorithms. It minimises the sieving computation effort for a given reduction quality by:

- offering a mechanism to recycle short vectors from one context to somewhat short vectors in an overlapping context, therefore already starting the sieve closer to completion. This formalises and generalises some of the ideas proposed in [Duc18a, LM18].
- being able to lift vectors to a larger context than the one currently considered. These vectors are considered for insertion at earlier positions. But as an extension to [Duc18a], which only lifted the final database of vectors, G6K is able to lift-and-compare all vectors encountered during the sieve. From this, we expect a few extra dimensions for free.[8]
- deferring the decision of where to insert a short vector until after the search effort. This is contrary to formal definitions of more standard reduction algorithms, e.g. BKZ or Slide [GN08a] reduction, and inspired by Deep LLL and recent RSR variants [TKH18].

[8] Lifting is somewhat more expensive than considering a pair of vectors. We are therefore careful to only lift a fraction of all considered vectors, namely only the considered vectors below a certain length of, say, $\sqrt{1.8} \cdot \mathrm{gh}(\mathcal{L}_{[\ell:r]})$.

The underlying computations per vector are reasonably cheap, typically linear or quadratic in the dimension of the vector currently being considered. The most critical operation, namely the SimHash test [Cha02, FBB+15, Duc18a] may be asymptotically sublinear or even polylogarithmic; in practice it consists of about a dozen x86 non-vectorised instructions for vectors of dimension roughly one hundred.

3.2 Vectors, Contexts and Insertion

All vectors considered by G6K live in one of the projected lattices $\mathcal{L}_{[\ell:r]}$ of a lattice $\mathcal{L}$. More specifically, they are represented in basis $\mathbf{B}_{[\ell:r]}$ as integral vectors $\mathbf{v} \in \mathbb{Z}^n$ where $n = r - \ell$, i.e. we have $\mathbf{w} = \mathbf{B}_{[\ell:r]} \cdot \mathbf{v}$ for some $\mathbf{w} \in \mathbb{R}^d$. Throughout, we may refer to the (projected) lattice vector $\mathbf{w}$ as the vector $\mathbf{v}$. It is convenient, and efficient, to also keep a representation, $\mathbf{v}^\circ \in \mathbb{R}^n$, of $\mathbf{w}$ in the orthonormalised basis $\mathbf{B}^\circ$. This conversion costs $O(n^2)$.

Below we list the three operations that *extend* or *shrink* a vector to the left or to the right.

- Extend Right (inclusion) $\mathrm{er} : \mathcal{L}_{[\ell:r]} \to \mathcal{L}_{[\ell:r+1]}$

$$(v_0, \dots v_{n-1}) \mapsto (v_0, \dots v_{n-1}, 0)$$
$$(v_0^\circ, \dots v_{n-1}^\circ) \mapsto (v_0^\circ, \dots v_{n-1}^\circ, 0)$$

- Shrink Left (projection) $\mathrm{sl} : \mathcal{L}_{[\ell:r]} \to \mathcal{L}_{[\ell+1:r]}$

$$(v_0, \dots v_{n-1}) \mapsto (v_1, \dots v_{n-1})$$
$$(v_0^\circ, \dots v_{n-1}^\circ) \mapsto (v_1^\circ, \dots v_{n-1}^\circ)$$

- Extend Left (Babai-lift) $\mathrm{el} : \mathcal{L}_{[\ell:r]} \to \mathcal{L}_{[\ell-1:r]}$

$$(v_0, \dots, v_{n-1}) \mapsto (-\lfloor c \rceil, v_0, \dots, v_{n-1})$$
$$(v_0^\circ, \dots, v_{n-1}^\circ) \mapsto ((c - \lfloor c \rceil) \cdot \left|\mathbf{b}_{\ell-1}^*\right|, v_0^\circ, \dots v_{n-1}^\circ),$$
$$\text{where } c = \sum_{j=0}^{n-1} \mu_{\ell-1,\ell+j} \cdot v_j.$$

These operations maintain, somewhat, the shortness of vectors. Indeed, by abuse of notation, letting $|\mathbf{v}|$ represent $|\mathbf{w}|$,

$$|\mathrm{er}(\mathbf{v})| = |\mathbf{v}|, |\mathrm{sl}(\mathbf{v})| \approx \sqrt{(r-\ell-1)/(r-\ell)} \cdot |\mathbf{v}|, |\mathrm{el}(\mathbf{v})|^2 \leq |\mathbf{v}|^2 + \left|\mathbf{b}_{\ell-1}^*\right|^2 / 4.$$

More properly, "shortness" should be considered relative to the Gaussian heuristic of a context, $\mathrm{gh}(\mathcal{L}_{[\ell:r]})$. For BKZ-$\beta$ reduced bases, and growing in accuracy as $r - \ell \to \infty$,

$$\frac{\mathrm{gh}(\mathcal{L}_{[\ell:r]})}{\mathrm{gh}(\mathcal{L}_{[\ell:r+1]})} \text{ and } \frac{\mathrm{gh}(\mathcal{L}_{[\ell:r]})}{\mathrm{gh}(\mathcal{L}_{[\ell+1:r]})} \approx \delta(\beta), \frac{\mathrm{gh}(\mathcal{L}_{[\ell:r]})}{\mathrm{gh}(\mathcal{L}_{[\ell-1:r]})} \approx \delta(\beta)^{-1}.$$

We may then calculate an approximate growth factor, relative to the Gaussian heuristics of the contexts, for each of the three operations

$$\frac{|\mathrm{er}(\mathbf{v})| \cdot \mathrm{gh}(\mathcal{L}_{[\ell:r]})}{|\mathbf{v}| \cdot \mathrm{gh}(\mathcal{L}_{[\ell:r+1]})} \approx \delta(\beta), \frac{|\mathrm{sl}(\mathbf{v})| \cdot \mathrm{gh}(\mathcal{L}_{[\ell:r]})}{|\mathbf{v}| \cdot \mathrm{gh}(\mathcal{L}_{[\ell+1:r]})} \approx \sqrt{\frac{r-\ell-1}{r-\ell}} \cdot \delta(\beta),$$
$$\frac{|\mathrm{el}(\mathbf{v})| \cdot \mathrm{gh}(\mathcal{L}_{[\ell:r]})}{|\mathbf{v}| \cdot \mathrm{gh}(\mathcal{L}_{[\ell-1:r]})} \leq \delta(\beta)^{-1} \left(1 + \frac{\left|\mathbf{b}^*_{\ell-1}\right|^2}{4 \cdot |\mathbf{v}|^2}\right)^{1/2}.$$

While it would seem natural to also define a Shrink Right operation, we have not found a geometrically meaningful way of doing so. Moreover, we neither have any algorithmic purpose for it.

Insertion. Performing an insertion (the elementary lattice reduction operation) of a vector is less straightforward. For $i \leq \ell < r$, $n' = r - i$, $n = r - \ell$ an insertion of a vector $\mathbf{v}$ at position i is a local change of basis making $\mathbf{w} = \mathbf{B}_{[i:r]} \cdot \mathbf{v}$ the first vector of the new local projected basis, i.e. applying a unimodular matrix $\mathbf{U} \in \mathbb{Z}^{n' \times n'}$ to $\mathbf{B}_{[i:r]}$ such that $(\mathbf{B}_{[i:r]} \cdot \mathbf{U})[0] = \mathbf{w}$. While doing so, we would like to recycle a database of vectors currently living in the context $[\ell : r]$.

In the case $i = \ell$, this causes no difficulties, and one could apply any change of basis $\mathbf{U}$ to the database. But to exploit dimensions for free, we will typically have $i < \ell$, which is more delicate. If we can ensure that

$$\mathrm{Span}((\mathbf{B} \cdot \mathbf{U})_{[i:\ell+1]}) = \mathrm{Span}(\mathbf{B}_{[i:\ell]} \cup \{\mathbf{w}\}) \tag{4}$$

then one can simply project all the database vectors orthogonally to $\mathbf{w}$, to end up with a database in a new smaller context $[\ell+1 : r]$. If it holds that $\mathbf{v}[j] = \pm 1$ for some $j \in \{\ell, \ldots, r-1\}$ an appropriate matrix $\mathbf{U}$ can be constructed as

$$\mathbf{U} = \left(\begin{array}{c|cc} \mathbf{v} & \mathbf{I}_{j \times j} & \mathbf{0} \\ & \mathbf{0} & \mathbf{0} \\ & \mathbf{0} & \mathbf{I}_{n'-j-1 \times n'-j-1} \end{array}\right). \tag{5}$$

However, it is important that the local projected bases remain somewhat reduced. If not, numerical stability issues may occur. Moreover, the condition that the short vector $\mathbf{v}$ contains a ± 1 in the context $[\ell : r]$ is often not satisfied without sufficient reduction. While we must be careful to not alter the vector space inside the sieving context, we can nevertheless perform a full size reduction (upper triangular matrix $\mathbf{T}$ with unit diagonal) on the whole of $\mathbf{B}_{[i:r]}$, as well as two local LLL reductions $\mathbf{U}_\mathrm{L}$ and $\mathbf{U}_\mathrm{R}$ on $\mathbf{B}_{[i:\ell+1]}$ and $\mathbf{B}_{[\ell+1:r]}$.

$$\mathbf{U}' = \mathbf{U} \cdot \mathbf{T} \cdot \begin{pmatrix} \mathbf{U}_\mathrm{L} & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_\mathrm{R} \end{pmatrix}. \tag{6}$$

Note that $\mathrm{Span}((\mathbf{B} \cdot \mathbf{U}')_{[i:\ell+1]}) = \mathrm{Span}((\mathbf{B} \cdot \mathbf{U})_{[i:\ell+1]})$, so that condition (4) is preserved.

3.3 G6K: a Stateful Machine

The General Sieve Kernel is defined by the following internal states and instructions.

State

- A lattice basis $\mathbf{B} \in \mathbb{Z}^{d \times d}$, updated each time an insert is made (Section 3.2). Associated with it is its Gram–Schmidt Orthonormalisation basis $\mathbf{B}^\circ$.
- Positions $0 \leq \kappa \leq \ell \leq r \leq d$. We refer to the context $[\ell : r]$ as the *sieving context*, and $[\kappa : r]$ as the *lifting context*. We define $n = r - \ell$ (the sieving dimension).
- A database db of N vectors in $\mathcal{L}_{[\ell:r]}$ (preferably short).
- Insertion candidates $\mathbf{c}_\kappa, \ldots, \mathbf{c}_\ell$ where $\mathbf{c}_i \in \mathcal{L}_{[i:r]}$ or $\mathbf{c}_i = \perp$.

Instructions

- Initialisation ($\texttt{Init}_{\mathbf{B}}$): initialise the machine with a basis $\mathbf{B} \in \mathbb{Z}^{d \times d}$.
- Reset ($\texttt{Reset}_{\kappa,\ell,r}$): empty database, and set (κ, ℓ, r).
- Sieve (`S`): run some chosen sieving algorithm. During execution of the algorithm, well chosen visited vectors are lifted from $\mathcal{L}_{[\ell:r]}$ to $\mathcal{L}_{[\kappa:r]}$ (by iterating el just on these vectors). If such a lift improves (i.e. is shorter than) the best insertion candidate $\mathbf{c}_i$ at position i, then it replaces $\mathbf{c}_i$. We call this optional[9] feature *on-the-fly lifting*.
- Extend Right, Shrink Left, Extend Left (`ER`, `SL`, `EL`): increase or decrease ℓ or r and apply er, sl or el to each vector of the database. All three operations maintain the insertion candidates (except for `EL` which drops $\mathbf{c}_\ell$).
- Insert (`I`): choose the best insertion candidate $\mathbf{c}_i$ for $\kappa \leq i \leq \ell$, according to a score function, and insert it at position i. The sieving context changes to $[\ell + 1 : r]$ and the database is updated as described in Section 3.2. If no insertion candidate is deemed suitable, then we simply run `SL` so as to ensure that the sieving context will end up as expected.[10] When we write $\texttt{I}_i$, we mean that insertion is only considered at position i.
- Grow or Shrink ($\texttt{Resize}_N$): change the database to a given size N. When shrinking, remove the longest vectors from the database. When growing, sample new vectors (using some unspecified sampling algorithm[11]). Typically, we will not explicate the calls to these operations, and assume that calling a sieve includes resizing the database to the appropriate size, for example $N = O(\sqrt{4/3}^n)$ for the 2-sieves of [NV08, MV10b, BGJ15].

Our implementation of this machine offers more functionality, such as the ability to monitor its state and therefore the behaviour of the internal sieve algorithm, and to tune the underlying algorithms.

[9] The alternative being to only consider the vectors of the final database for lifting.
[10] Note that sl can be viewed as the trivial insertion of the vector $v_\kappa = (1, 0, \ldots, 0)$.
[11] When possible we prefer to sample by summing random pairs of vectors from the database.

4 Reduction Algorithms using G6K

Equipped with this abstract machine, we can now reformulate, improve and generalise strategies for lattice reduction with sieving algorithms. In the following we will assume that the underlying sieve algorithm has a time complexity proportional to C^n, with n the dimension of the SVP instance, and we also define $C' = 1/(1-1/C)$. This second constant approximates the multiplicative overhead $\sum_{i=1}^{n} C^i/C^n$ encountered on iterating sieves in dimensions 1 to n. Note that this overhead grows when C decreases. More concretely, depending on the sieve, C can range from $4/3$ down to $\sqrt{3/2}$, giving $C' = 4$ up to $C' \approx 5.45$.

4.1 The Pump

In this section we propose a sequence of instructions called the `Pump`. They encompass the progressive-sieving strategy proposed in [Duc18a, LM18] as well as the dimensions for free and multi-insertion tricks of [Duc18a]. The original progressive-sieving strategy can be written as

$$\texttt{Reset}_{0,0,0},\ (\texttt{ER},\ \texttt{S})^{d},\ \texttt{I}_0. \tag{7}$$

Similarly, a $\texttt{SubSieve}_f$ which attempts a partial HKZ reduction using sieving with f dimensions for free can be written as

$$\texttt{SubSieve}_f : \texttt{Reset}_{0,f,f},\ (\texttt{ER},\ \texttt{S})^{d-f},\ \texttt{I}_0, \texttt{I}_1, \ldots, \texttt{I}_{d-f-1}.^{12} \tag{8}$$

We note that due to the newly introduced `EL` operation, it is also possible to perform the progressive-sieving right to left

$$\texttt{Reset}_{0,d,d}, (\texttt{EL},\ \texttt{S})^{d-f},\ \texttt{I}_0, \texttt{I}_1, \ldots, \texttt{I}_{d-f-1}. \tag{9}$$

Perhaps surprisingly, experimentally the left variant of progressive-sieving performs substantially better. In combination with certain sieving methods, the right variant even fails completely, this will be discussed in more detail in Section 4.5.

To arrive at `Pump`, note first that G6K maintains insertion candidates at many positions. We can therefore relax the insertion positions of (9) and choose those that appear to be optimal. The choice of insertion position is discussed in Section 4.4.

Secondly, due to on-the-fly lifting, we note that the sequence (9) considers many more insertion candidates for the first insertion than for subsequent insertions. Moreover, we noticed that after several insertions, the database contained vectors much longer than recent inserts. By sieving also during the "descent phase", i.e. when inserting and shrinking the sieve context, we remedy this imbalance and expect to obtain a more strongly reduced basis, ideally obtaining an HKZ-reduced context.

[12] This sequence refers to $\texttt{SubSieve}^{+}(\mathcal{L}, f)$ with Sieve being progressive [Duc18a].

In summary, we define the parameterised $\texttt{Pump}_{\kappa,f,\beta,s}$ as the following sequence

$$\texttt{Pump}_{\kappa,f,\beta,s} : \texttt{Reset}_{\kappa,\kappa+\beta,\kappa+\beta}, \overbrace{(\texttt{EL}, \texttt{S})^{\beta-f}}^{\text{pump-up}}, \overbrace{(\texttt{I}, \texttt{S}^s)^{\beta-f}}^{\text{pump-down}}. \tag{10}$$

where $0 \leq \kappa \leq \kappa + \beta \leq d$, $0 \leq f \leq \beta$, and where $s \in \{0, 1\}$ controls whether we sieve during pump-down. One may expect the cost of these extra sieves to be close to a multiplicative factor of 2, but experimentally the factor can reach 3 for certain sieves (e.g. `bgj1`), as more collisions[13] seem to occur during the descent phase. This feature is mostly useful for weaker reduction tasks such as BKZ, see `PumpNJumpBKZTour` below.

4.2 SVP

To solve the shortest vector problem on the full lattice, starting from an LLL-reduced basis $\mathbf{B}$, we proceed as in [Duc18a], that is, we iterate $\texttt{Pump}_{0,f,d,s}$ for decreasing values of f. While only the last `Pump` delivers the shortest vector, the previous iterations provide a strongly reduced basis (near HKZ-reduced), which allows more dimensions for free to be achieved. We expect to obtain further dimensions for free due to on-the-fly lifting.

Similarly, for solving SVP in context $[\kappa : \kappa + \beta]$ (e.g. as a block inside BKZ), we instead make iterative calls to $\texttt{Pump}_{\kappa,f,\beta,s}$.

Note that we can decrease f in larger increments than 1 to balance the cost of the basis reduction effort and the search for the shortest vector itself. Indeed, with increments of 1, the overhead factor C' for $C = \sqrt{3/2}$ is $C' \approx 4.45$. Decreasing f by 2 gives an overhead of $C' = 1/(1 - C^{-2}) = 3$ and by 3 gives $C' = 1/(1 - C^{-3}) \approx 2.19$. Such speed-ups are worth losing 1 or 2 dimensions for free.

We therefore define `WorkOut` as the following sequence of `Pump`

$$\begin{aligned}\texttt{WorkOut}_{\kappa,\beta,f,f^+,s} : &\texttt{Pump}_{\kappa,\beta-f^+,\beta,s}, \texttt{Pump}_{\kappa,\beta-2f^+,\beta,s}, \\ &\texttt{Pump}_{\kappa,\beta-3f^+,\beta,s}, \ \ldots \ \texttt{Pump}_{\kappa,f,\beta,s},\end{aligned} \tag{11}$$

where f^+ is the increment mentioned above. From experiments on exact-SVP and SVP-challenges, we found it worthwhile to deactivate sieving in the descent phase ($s = 0$), though activating it ($s = 1$) is preferable in other contexts, or to use less memory at a larger time cost. Similarly, for certain tasks (e.g. the SVP challenges, i.e. 1.05-Hermite-SVP) we found the optimal increment, f^+, to be 2 or 3. This parameter also drives a time-memory trade-off; setting f^+ to 1 saves on memory by allowing for a larger f, but at a noticeable cost in time.

For solving exact-SVP, it is not clear when to stop this process because we are never certain that a vector is indeed the shortest vector of a lattice (except maybe by running a very costly non-pruned enumeration). In these cases, one should

[13] A collision is when a new vector $\mathbf{v}$ to be inserted in the database equals $\pm\mathbf{v}_2$ for some $\mathbf{v}_2$ already present in the database.

therefore guess, from experimental data, a good number f of dimensions for free. Note that it is is rarely critical to achieve exact-SVP, and lattice reduction algorithms such as BKZ tolerate approximations.

In some cases, such as the Darmstadt SVP Challenge, we do not have to solve exact-SVP, but rather find a vector of a prescribed norm, near the Gaussian heuristic. In this case we do not need to predetermine f and simply iterate the `Pump` until satisfaction. As a consequence, we also add an extra option to the `Pump` to allow early aborts when it finds a satisfying candidate $\mathbf{c}_\kappa$. In practice we observe significant savings from this feature, i.e. we observe the `Pump` aborting before reaching its topmost dimension, or at the beginning of the descent phase.

4.3 BKZ

Having determined the appropriate parameters f, f^+, s for solving SVP-β (made implicit in the following), the naïve implementation of BKZ is given by the following program

$$\begin{aligned}\texttt{NaiveTour}_\beta : \texttt{WorkOut}_{0,\beta},\ \texttt{WorkOut}_{1,\beta+1},\ \dots \\ \texttt{WorkOut}_{d-\beta,d},\ \dots,\ \texttt{WorkOut}_{d-1,d}.\end{aligned} \tag{12}$$

Several strategies to amortise the cost of sieving inside BKZ were suggested in [Duc18a, LM18]. These aimed to reduce the cost of a tour of BKZ-β below d (or $d-\beta$) times the cost of SVP in dimension β. Again, these strategies are implementable as a sequence of G6K instructions.

Namely, the sliding-window strategy of [LM18] can be expressed as

$$\texttt{SlidingWindowTour}_\beta : \texttt{Reset}_{0,0,0},\ (\texttt{ER},\ \texttt{S})^\beta,\ (\texttt{I}_\ell,\ \texttt{S},\ \texttt{ER},\ \texttt{S})^{d-\beta},\ (\texttt{I}_\ell,\ \texttt{S})^\beta. \tag{13}$$

It is also possible to combine this strategy with the dimensions for free of [Duc18a]. However, there are two caveats. First, it relies on extend right, which is currently problematic in our implementation of G6K, see Section 4.5. Secondly, even if this issue is solved, we remark that inside a BKZ tour it is preferable to run LLL on the full basis periodically. From the sandpile point of view [MV10a, HPS11], not doing so implies that a "bump" accumulates at the right of the reduced blocks, as we try to push the sand to the right. We see no clear strategies to recycle the vectors of a block when calling a full LLL.

Alternatively, [Duc18a] identified two other potential amortisations. First, it is noted that the `WorkOut` (or even just a `Pump`) in a block $[\kappa : \kappa+\beta]$ leaves the next block $[\kappa+1 : \kappa+\beta+1]$ already quite well reduced. It may therefore not be necessary to do a full `WorkOut`, but simply run the last `Pump` of this `WorkOut`, therefore saving up to a factor of C' in the running time.

The second suggestion of [Duc18a] consists of overshooting the blocksize β, so that a `Pump` in dimension $\beta' > \beta$ attempts to HKZ reduce a larger block. In particular for parameter j, let $\beta' = \beta + j - 1$ and after a $\texttt{Pump}_{\kappa,f,\beta'}$ *jump* by j blocks. This decreases the number of calls to the `Pump` to d/j and may also slightly improve the quality of the reduction, but increases the cost of the `Pump`

calls by a factor of C^{j-1}. It is argued that such a strategy could give a speed-up factor ranging from 2.2 to 3.6 for a fixed basis reduction quality. In this case we therefore perform the following sequence

$$\texttt{PumpNJumpTour}_{\beta',f,j} : \texttt{Pump}_{0,f,\beta'},\ \texttt{Pump}_{j,f,\beta'},\ \texttt{Pump}_{2j,f,\beta'},\dots \tag{14}$$

We alter the version above to allow for more opportunism. Since choosing f to almost certainly solve exact-SVP in blocks is costly, we instead embrace the idea of achieving the most basis reduction from a given sieving context. Extending the lift context makes the lift operation more expensive, but gives more insertion candidates, and therefore gives a new trade-off that can optimised over. To this end note that while $\texttt{Pump}_{\kappa',f+\kappa-\kappa',\beta+\kappa-\kappa'}$ for $\kappa' < \kappa$ takes more dimensions for free than $\texttt{Pump}_{\kappa,f,\beta}$, it still provides the same insertion candidates, $\mathbf{c}_\kappa, \dots, \mathbf{c}_{\kappa+f}$. It also provides new insertion candidates $\mathbf{c}_{\kappa'}, \dots, \mathbf{c}_{\kappa-1}$. This is because the sieving contexts do not shrink. Therefore, provided we take care in the first few blocks, the quality cannot decrease. To achieve this start with Pumps with no dimensions for free and move the sieving context right until the desired number of dimensions for free is attained, then continue as before. Set $f' > f$, $\beta = \beta + f' - f$ (i.e. the value that fixes the sieve context sizes) and $\beta' = \beta + j - 1$ and perform

$$\begin{aligned}\texttt{PumpNJumpTour}_{\beta',f',j} : \texttt{Pump}_{0,0,\beta'-f'},\ \texttt{Pump}_{0,j,\beta'-f'+j},\dots,\ \texttt{Pump}_{0,f',\beta'},\\ \texttt{Pump}_{j,f',\beta'},\ \texttt{Pump}_{2j,f',\beta'},\dots\end{aligned} \tag{15}$$

4.4 Scoring for Inserts

The issue of deciding where in a basis to insert given candidates throughout reduction has already been discussed in [TKH18], in the context of the SVP Challenges. Until the actual shortest vector is found, these insertions have the purpose of improving the basis quality. Inserting at an early position may degrade quality at later positions, because we do not know a priori how inserting $\mathbf{c}_i$ will affect $\mathbf{B}_{[\ell:r]}$ for $i \leq \ell < r$. Therefore one must find a good trade-off between making long lasting yet weak improvements at early positions, and strong yet fragile improvements at later positions.

One way to achieve this is to use the scoring proposed in [TKH18], a function over the whole basis which measures the global effect of each potential insert, i.e. checking exactly how inserting $\mathbf{c}_i$ affects the $\mathbf{B}_{[\ell:r]}$. We use a simplified variant of this scoring which scores the improvement of each potential insert according to the following local condition

$$\varsigma(i) = \begin{cases} 0, & \text{if } \mathbf{c}_i = \perp \\ \theta^{-i} \cdot |\mathbf{b}_i^*|^2 \,/\, |\mathbf{c}_i|^2, & \text{otherwise} \end{cases} \tag{16}$$

for some constant $\theta \geq 1$ and take the maximum over the valid indices. Setting $\theta = 1$ corresponds to always choosing the "most improving" candidate, while setting θ quite large (say 10) corresponds to always inserting at the earliest position.

To optimise θ, we ran $\texttt{WorkOut}_{0,d,f}$ for $f = 30$ and $d = 110$, measured $\gamma = \mathrm{gh}(\mathcal{L})/\mathrm{gh}(\mathcal{L}_{[f:d]})$, and chose $\theta = 1.04$ which minimised this quantity γ. We recall [Duc18a] that γ must be below a certain threshold to guarantee the success of exact-SVP in dimension d with f dimensions for free.

The optimal value of θ may differ depending on other parameters, e.g. dimension, approximation factor, and the context, e.g. exact-SVP, 1.05-Hermite-SVP, BKZ, and the question of optimising insertion strategies requires more theoretical and experimental attention. We hope that our open source implementation will ease such future research.

4.5 Issue with Extend Right

As mentioned earlier, our current implementation does not support the `ER` operation very well. In more detail, the issue is that after running a sieve in the context $[\ell : r]$, and applying `ER`, the vectors in the database are padded with 0 to be defined over the context $[\ell : r + 1]$; geometrically, these vectors remain in the context $[\ell : r]$, and so will all their potential combinations considered by the sieve. While we do add some fresh vectors to increase the database size, the fraction of those fresh vectors in the database is rather small: $1 - \sqrt{3/4} \approx 13\%$. This alone seems to slow down the Gauss sieve when used in right progressive-sieving compared to left progressive-sieving.

The situation is even worse with the faster sieves we implemented. Indeed, apart from the reference Gauss sieve, our sieves are not guaranteed to maintain the full-rankness of the database. The reason is, for performance purposes, we relaxed the replacement condition. In the standard Gauss sieve, $\mathbf{x} \pm \mathbf{y}$ may only replace $\mathbf{x}$ or $\mathbf{y}$ if it is shorter. We relax this and allow $\mathbf{x} \pm \mathbf{y}$ to replace the current largest vector $\mathbf{z}$ in the database. Fresh vectors are much longer than the recycled ones, therefore they are quickly replaced by combinations of recycled vectors, effectively meaning there is little representation of the newly introduced basis vector after an `ER`.

While we tried to implement countermeasures to avoid losing rank, they had a noticeable impact on performance, and were not robust. For this work, we therefore avoid the use of extend right, as reductions based on extend left already perform well. We leave it as an open problem to develop appropriate variants of fast sieve algorithms that avoid this issue.

5 Implementation details

5.1 Sieving

We implemented several variants of sieving, namely: a Gauss sieve [MV10b], a relaxation of the Nguyen–Vidick sieve [NV08], a restriction of the Becker–Gama–Joux sieve [BGJ15] and a 3-sieve [BLS16, HK17]. All exploit the SimHash speed-up [Cha02, FBB+15, Duc18a].

The first two were mostly implemented for reference and testing purposes, and therefore are not multi-threaded. Nevertheless, we fall back to Gauss sieve in

small dimensions for efficiency and robustness; as discussed earlier, Gauss sieve is immune to loss of rank, which we sometimes experienced with other sieves in small dimensions (say, $n < 50$), even when not using extend right.

The termination condition for the sieves follows [Duc18a], namely, they stop when we have obtained a given ratio of the expected number of vectors of norm less than $R \cdot \mathrm{gh}(\mathcal{L}_{[\ell:r]})$. The saturation radius is dictated by the asymptotics of the algorithm at hand, namely, R is such that the sieve uses a database of $N = O(R^n)$ vectors. In particular $R = \sqrt{4/3}$ for all implemented sieves, except for the 3-sieve for which one can choose $R^2 \in \left[3\sqrt{3}/4, 4/3\right] \approx [1.299, 1.333]$.

Nguyen–Vidick Sieve (`nv`) and Gauss Sieve (`Gauss`) The Nguyen–Vidick sieve finds pairs of vectors $(\mathbf{v}_1, \mathbf{v}_2)$ from the database, whose sum or difference gives a shorter vector, i.e. $|\mathbf{v}_1 \pm \mathbf{v}_2| < \max\{|\mathbf{v}| : \mathbf{v} \in \mathtt{db}\}$. Once such a pair is found, the longest vector from the database gets replaced by $\mathbf{v}_1 \pm \mathbf{v}_2$. The size of the database is a priori fixed to the asymptotic heuristic minimum $2^{0.2075n+o(n)}$ required to find enough such pairs. The running time of the Nguyen–Vidick sieve is quadratic in the database size.

The Gauss sieve algorithm, similar to the Nguyen–Vidick sieve, searches for pairs with a short sum, but the replacement and the order in which we process the database vectors, differ. More precisely, the database now is (implicitly) divided into two parts, the so-called "list" part and the "queue" part. This separation is encoded in the ordering, with the list part being the first τ vectors. Both parts are kept separately sorted. The list part has the property that the shortness of $\mathbf{v}_1 \pm \mathbf{v}_2$ has been checked for all pairs of vectors $\mathbf{v}_1, \mathbf{v}_2$ in the list. We then only check pairs $(\mathbf{v}_1, \mathbf{v}_2)$, where $\mathbf{v}_1$ comes from the queue part and $\mathbf{v}_2$ from the list part. As opposed to Nguyen–Vidick sieve, once a reduction is found, the longer vector from the pair $(\mathbf{v}_1, \mathbf{v}_2)$ gets replaced by $\mathbf{v}_1 \pm \mathbf{v}_2$, not the longest in the database. In the case where the list vector $\mathbf{v}_2$ gets replaced, the result of the reduction $\mathbf{v}_1 \pm \mathbf{v}_2$ is put into the "queue" part and the search is continued with the same "queue" vector $\mathbf{v}_1$. Otherwise, if the queue vector $\mathbf{v}_1$ was the longest and is replaced, we restart comparing $\mathbf{v}_1$ with all list vectors. A vector is moved from the "queue" to the "list" part once no reduction with the "list" vectors can be found. Asymptotically, the running time and the database size for the Gauss sieve is the same as for the Nguyen–Vidick sieve, but it performs better in practice.

Becker–Gama–Joux Sieve (`bgj1`) The sieve algorithm from [BGJ15] accelerates the Nguyen–Vidick sieve [NV08] from $2^{0.415n+o(n)}$ down to $2^{0.311n+o(n)}$ by using locality sensitive filters, while keeping the memory consumption to its bare minimum for a 2-sieve, namely $2^{0.2075n+o(n)}$.

This optimal complexity is reached using recursive filtering, however we only implemented a variant of this algorithm with a single level of filtration (hence the name `bgj1`). We leave it to future work to implement the full algorithm and determine when the second level of filtration becomes interesting.

We briefly describe our simplified version. The algorithm finds neighbouring pairs in the database by successively filling buckets according to a filtering rule, and doing all pairwise tests inside a bucket. Concretely, it chooses a uniform direction $\mathbf{d} \in \mathbb{R}^n$, $|\mathbf{d}| = 1$, and puts in the bucket all database vectors taking (up to sign) a small angle with $\mathbf{d}$, namely all $\mathbf{v}$ such that $|\langle \mathbf{v}, \mathbf{d} \rangle| > \alpha \cdot |\mathbf{v}|$.

We choose α so that the size of the buckets is about the square root of the size of the database (asymptotically, $\alpha^2 \to 1 - \sqrt{3/4} \approx 0.366^2$). This choice balances the cost of populating the bucket (through testing the filtering condition) and exploring inside the bucket (checking for pairwise reductions). Both cost $O(N) = 2^{0.2075n+o(n)}$; though in practice we found it faster to make the buckets slightly larger, namely around $3.2\sqrt{N}$. Also note that we can apply a SimHash pre-filtering before actually computing the inner product $\langle \mathbf{v}, \mathbf{d} \rangle$, but using a larger threshold for the bucketing pre-filter than for the reduction pre-filter.

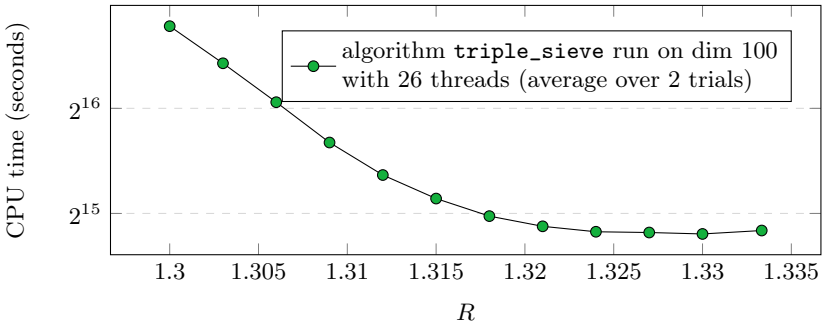
Following the heuristic arguments from the literature, and in particular the wedge volume formula [BDGL16, Lemma 2.2], we conclude that this sieve succeeds after about $(2/\sqrt{3} - 1/3)^{-n/2} \approx 2^{0.142n+o(n)}$ buckets, for a total complexity of $2^{0.349n+o(n)}$.

3-sieve (`triple_sieve`) In its original versions [BLS16, HK17], the 3-sieve algorithm aims to reduce memory consumption at the cost of a potential increase in the running time. The 3-sieve algorithm searches not for pairs, but for *triples* of vectors, whose sum gives a shorter vector (hence, the name 3-sieve). Clearly, for a fixed size list of vectors, there are more possible triples than pairs and, therefore, we can start with a shorter list and still find enough reductions. However, a (naïve) search now costs three iterations over the list. To speed-up the naïve search, we can apply filtering techniques similar to the ones used for `bgj1`. In particular, the 3-sieve algorithm with filtering described in [HK17] requires memory $2^{0.1788n+o(n)}$ and runs in time $2^{0.396n+o(n)}$.

For any vector $\mathbf{x}$ from the database, the 3-sieve algorithm of [HK17] filters the database by collecting all vectors $\mathbf{v}$ with a large enough inner product $|\langle \mathbf{x}, \mathbf{v} \rangle|$. For all pairs of these collected vectors $(\mathbf{v}_1, \mathbf{v}_2)$, 3-sieve checks if $|\mathbf{x} \pm \mathbf{v}_1 \pm \mathbf{v}_2|$ gives a short(er) vector. Such an inner product test, as in `bgj1`, helps to identify "promising" vectors which are likely to result in a length reduction. The only subtlety lies in the fact that in order for a triple to give a reduction, the vectors $\mathbf{x}, \mathbf{v}_1, \mathbf{v}_2$ should be far apart, not close to each other as in 2-sieve. We handle this by adjusting the inner product test and choosing the $\pm$ signs appropriately.

The version of the 3-sieve implemented in G6K splits the database into "list" and "queue" parts in the same way as the Gauss sieve above. Further, it combines 2- and 3-sieves. Notice that the filtering process of 3-sieve is basically the same as bucketing in `bgj1`, with a bucket centre defined by a database[14] vector $\mathbf{x}$. When processing the bucket, we check not only whether a pair $(\mathbf{v}_1, \mathbf{v}_2)$ from

[14] This relies on the fact that we do not use recursive filtering in `bgj1`: the asymptotically optimal choice from [BGJ15] mandates choosing the buckets centres in a structured way, which is not compatible with choosing them as `db` elements.



The X-axis is the parameter R such that the database size is set to $3.2 \cdot R^{n/2}$. In particular, the right-most point corresponds to the size of a database set to $3.2 \cdot (4/3)^{n/2}$; for the left-most point this value is set to $3.2 \cdot (3\sqrt{3}/4)^{n/2}$. Raw data embedded.

Fig. 2: Time-memory trade-off for our implementation of the 3-sieve algorithm.

the bucket gives a shorter vector, but also whether a triple $(\mathbf{x}, \mathbf{v}_1, \mathbf{v}_2)$ may. This additional check has no noticeable impact on performance (we know in which case we potentially are from the signs of the scalar products alone), but has the potential to find more shorter vectors.

As a result, in this combined version of the sieve, we can find more reductions than in 2-sieve if we keep the same database size as for 2-sieve. In such a memory regime, most of the reductions will come from 2-reductions. Setting a smaller database makes the algorithm look for more 3-reductions as 2-reductions become less likely.

As `triple_sieve` finds more reductions than `bgj1` with the same database sizes, we may decrease the size of the database and check how the running time degrades. The results of these experiments are shown in Figure 2. The leftmost point corresponds to minimal memory regime for 3-sieve, namely when the database size is set to $2^{0.1788n+o(n)}$, while the rightmost point is for the `bgj1` memory regime, that is the database size is set to $2^{0.2075n+o(n)}$. It turns out that on moderate dimensions (i.e. 80–110), `triple_sieve` performs slightly better if the database size is a bit less than $2^{0.2075n+o(n)}$. Furthermore, these experiments are consistent with theoretical results on the high memory regime for 3-sieve: in [HKL18] it was proven that the running time of 3-sieve quickly drops down if allowed slightly more memory, as Figure 2 shows.

5.2 The Three Layers: `C++ / Cython / Python`

Our implementation consists of three layers.

C++11. The lowest level routines are implemented in C++11. In particular, at this level we define a Siever class which realises G6K for all sieves considered in this work: Gauss, NV, BGJ1 and 3-sieve. The general design is similar to FPLLL where algorithms are objects operating on matrices and Gram–Schmidt

objects. In particular, different sieves are realised as methods on the same object (and thus the same database) allowing the caller to pick which sieve to run in a given situation. For example, in small dimensions it is beneficial to run the Gauss sieve and this design decision allows the database to be reused between different sieves. Our C++ layer does not depend on any third party libraries (except pthreads). On the other hand, our C++ layer is relatively low level.

Cython. Cython is a glue language for interfacing between CPython (the C implementation of the Python programming language) and C/C++. We use Cython for this exact purpose. Our Cython layer is relatively thin, mainly making our C++ objects available to the Python layer and translating to and from FPyLLL data structures [dt18b]. The most notable exception is that we implemented the base-change computation of the *insert* instruction `I` (equations (5) and (6)) in Cython instead of C++. The reason being that we call LLL on the lifting context when inserting (the Cython function `split_lll`) which is realised by calling FPyLLL. That is, while our C++ layer has no external dependencies, the Cython layer depends on FPyLLL.

Python. All our high level algorithms are implemented in (C)Python (2). Our code does not use the functional-style abstractions from Section 3, but a more traditional object-oriented approach where methods are called on objects which hold the state. We do provide some syntactic sugar, though, permitting a user to construct new instructions from basic instructions in a function-composition style similar to the notation in Section 3. Nevertheless, this simplified abstraction is not able to fully exploit all the features of our implementation, and significant savings may be achieved by using the full expressivity of our library.

5.3 Vector Representation and Data Structures

The data structures of G6K have been designed for the high performance of sieving operations, where we have tried to minimise memory usage where possible. For high performance we retain the following information about each vector $\mathbf{v}$ as an *entry* `e` in the sieve database `db`:

- `e.x`: the vector $\mathbf{v}$ itself as 16-bit integer coordinates in base $\mathbf{B}_{[\ell:r]}$;
- `e.yr`: a 32-bit floating point vector to efficiently compute $\langle \mathbf{v}, \mathbf{v}_2 \rangle$, this is a renormalised version of $\mathbf{v}^\circ$;
- `e.cv` (compressed vector): a 256-bit SimHash of $\mathbf{v}$;
- `e.uid` (unique identifier): a 64-bit hash of $\mathbf{v}$;
- `e.len`: the squared length $|\mathbf{v}|$ as a 64-bit floating point number.

The entire database `db` is stored contiguously in memory, although unordered. This memory is preallocated for maximum database size within each `Pump`, to avoid additional memory usage caused by reallocations of the database whenever it grows.

To be able to quickly determine whether a potential new vector is already in the database we additionally maintain a C++ `unorderedset` (i.e. a hash table)

`uiddb` containing all 64-bit hashes `uid` of the vectors in `db`.[15] This hash $\texttt{uid} = H(\texttt{x})$ of $\texttt{x}$ is simply computed as the inner product of $\texttt{x}$ with a global random vector in the ring $\mathbb{Z}/2^{64}\mathbb{Z}$, which has the additional benefit that $H(\texttt{x}_1 \pm \texttt{x}_2)$ can be computed more efficiently as $H(\texttt{x}_1) \pm H(\texttt{x}_2)$. This allows us to cheaply discard collisions without even having to compute $\texttt{x}_1 \pm \texttt{x}_2$.

To maintain a sorted database we utilise a compressed database `cdb` that only stores the 256-bit SimHash, 32-bit floating point length, and the 32-bit `db`-index of each vector. This requires only 40 bytes per vector and everything is also stored contiguously in memory. It is optimised for traversing the database in order of increasing length and applying the SimHash as a prefilter, since accessing the full entry in `db` only occurs a fraction of the time.

For the multi-threaded `bgj1`-sieve, the compressed database `cdb` is maintained generally sorted in order of increasing length. Initially `cdb` is sorted, then during sieving vectors are replaced one-by-one starting from the back of `cdb`. It is only resorted when a certain fraction of entries have been replaced. Since we only insert a new vector if its length is below the minimum-length of the range of to-be-replaced vectors in `cdb`, this approach ensures that we always replace the largest vector in `db`. In the sieve variants that split the database into queue and list ranges, we regularly sort the individual ranges. In our multi-threaded `triple_sieve`, the vectors removed during a replacement are chosen iteratively from the backs of the two ranges.

Most sieving operations use buckets that are filled based on locality-sensitive filters. In `bgj1`, we use the same datastructure as `cdb` for the buckets, and thus copy those compressed entries in contiguous memory reserved for that bucket. For `triple_sieve`, we also store information about the actual scalar product $\langle \mathbf{x}, \mathbf{v} \rangle$ of the bucket elements $\mathbf{v}$ with the bucket centre $\mathbf{x}$ inside the bucket.

5.4 Multi-threading

G6K is able to efficiently use multi-threading for nearly all operations; a detailed efficiency report is to be found in Appendix B. Global per-entry operations such as `EL`, `ER`, `SL` and `I`-postprocessing are simply distributed over all available threads in the global threadpool.

During multi-threaded sieving we guarantee all write operations to entries in `db`, `cdb` and the best lift database to be executed in a thread-safe manner using atomic operations and write locks. (The actual locking strategies differ per implementation.) We always perform all heavy computations before locking and let each thread locally buffer pending writes and execute these writes in batches to avoid bottlenecks in exclusive access of these global resources.

Threads reading entries in `db` and `cdb` do not use locking and can thus potentially read partially overwritten entries. While this may result in some wasted computations, no faulty vectors will be inserted in the `db`: for every new vector we completely recompute its full entry `e` from `e.x` including its length and verify

[15] This `unorderedset` is in fact split into many parts to eliminate most blocking locks during a multi-threaded sieve.

Machine	CPUs	base freq.	cores	threads	HTC*	RAM
L	4xIntel Xeon E7-8860v4	2.2Ghz	72	72	No	512GiB
S	2xIntel Xeon Gold 6138	2.0Ghz	40	80	Yes	256GiB
C	2xIntel Xeon E5-2650v3	2.3Ghz	20	40	Yes	256GiB
A	2xIntel Xeon E5-2690v4	2.6Ghz	28	56	Yes	256GiB

* HTC: Hyperthreading-capable.

Table 1: Details of the machines used for experiments.

it is actually shorter than the length of the to-be-replaced vector before actually replacing it.

Safely resorting `cdb` during sieving is the most complicated, since threads do not block on reading `cdb`. Our implementations in G6K resolve this as follows. We let one thread resort `cdb` and use locking to prevent any insertions (or concurrent resorting) by other threads. We keep the old `cdb` untouched as a shadow copy for other threads, while computing a new sorted version that we then atomically publish. Afterwards, other threads will then eventually switch to the newer version. Insertions are always performed using `cdb` and never using a shadow copy, even if e.g. a thread is still using a shadow copy for its main operations, e.g. when building a bucket.

6 New Lattice Reduction Records

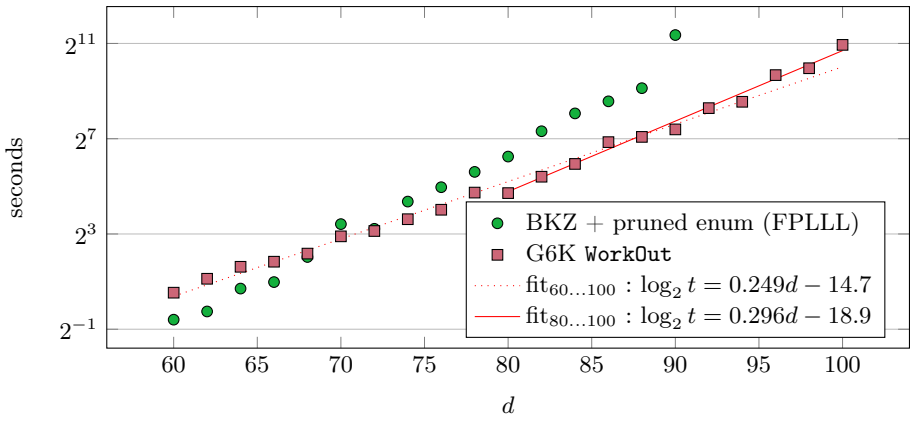
The experiments reported in this section are based on `bgj1`-sieving, except those on BKZ and LWE which are based on `triple_sieve`, in the high memory regime ($N = \Theta((4/3)^{n/2})$). The switch occurred when improvements to the latter made it faster than the former (especially with pump-down sieve, $s = 1$). While it seemed wasteful to rerun all the experiments, we nevertheless now recommend `triple_sieve` over `bgj1` for optimal performance within our library. The details of the machines used for our various experiments are given in Table 1.

6.1 Exact-SVP

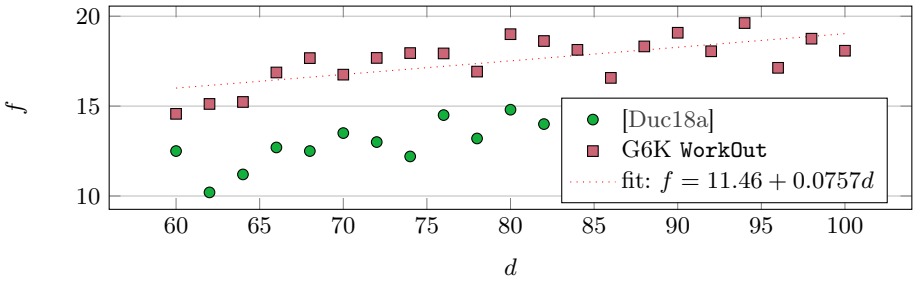
We first report on the efficiency of our implementation of G6K's `WorkOut` ($s = 0$, $f^+ = 1$) for solving exact-SVP. The comparison with pruned enumeration is given in Figure 3a. While fitted curves are provided, we highlight that they are significantly below asymptotic predictions of $2^{0.349d+o(d)}$ for `bgj1` and thus unreliable for extrapolation.[16] Based on these experiments, we report a crossover with enumeration around dimension 70. Note that we significantly outperform the guesstimates of a crossover at dimension 90 made in [Duc18a].

[16] This mismatch with theory can be explained by various kinds of overheads, but mostly by the dimensions for free trick: as $f = \Theta(d/\log d)$ is quasilinear, the slope will only very slowly converge to the asymptotic prediction.

(a) Average time in seconds for solving exact SVP.



(b) Average number of dimensions for free when solving exact SVP.



The running time was averaged over 60 trials (12 trials on random bases of 5 different lattices from [SG10]). Each instance was monothreaded, but ran in parallel (20thread/20cores, not hyperthreaded) on machine C. Raw data embedded.

Fig. 3: Performance for exact-SVP.

While our improved speed compared to [Duc18a] is mostly due to having implemented a faster sieving algorithm, the new features of G6K also contribute to this improved efficiency (see Appendix A for a detailed comparison). In particular the on-the-fly lifting strategy offers a few extra dimensions for free as depicted in Figure 3b. That is, our new implementation is not only faster but also consumes less memory.

6.2 1.05-Hermite-SVP (a.k.a. Darmstadt SVP Challenges)

The detailed performance of our implementation for solving Darmstadt SVP Challenges is given in Table 2. For some challenges, we also continued the experiments until no shorter vectors were found, hoping to have solved exact-SVP on those instances. We also compare the running time of our experiments with prior works in Figure 1. We warn the reader that the experiments of Table 2

New solutions to 1.05-Hermite-SVP

SVP dim	Norm	Hermite factor	Sieve max dim	Wall time	Total CPU time	Memory usage	Machine
155	3165	1.00803	127	$14d$ $16h$	$1056d$	†246 GiB	L
153	3192	1.02102	123	$11d$ $15h$	$911d$	†139 GiB	S
151	3233	1.04411	124	$11d$ $19h$	$457.5d$	†160 GiB	C
149	3030	0.98506	117	$60h$ $7m$	$4.66kh$	†59 GiB	S
147	3175	1.03863	118	$123h$ $29m$	$4.79kh$	67.0 GiB	C
145	3175	1.04267	114	$39h$ $3m$	$1496h$	37.7 GiB	C
143	3159	1.04498	110	$17h$ $23m$	$669h$	21.3 GiB	C
141	3138	1.04851	105	$4h$ $59m$	$190h$	10.6 GiB	C
139	3111	1.04303	108	$9h$ $56m$	$380h$	16.2 GiB	C
137	3093	1.04472	107	$9h$ $26m$	$362h$	14.1 GiB	C
136	3090	1.04937	108	$9h$ $16m$	$354h$	16.2 GiB	C
135	3076	1.04968	108	$7h$ $21m$	$277.4h$	16.1 GiB	C
133	3031	1.04133	103	$1h$ $59m$	$71.7h$	8.0 GiB	C
131	2959	1.02362	100	$1h$ $11m$	$41.5h$	5.3 GiB	C
129	2988	1.03813	98	$54m$	$33.2h$	4.2 GiB	C
128	3006	1.04815	102	$2h$ $32m$	$94.9h$	7.6 GiB	C
127	2972	1.04244	101	$2h$ $17m$	$85.0h$	6. GiB	C
126	2980	1.04976	100	$31m$	$19.2h$	5.6 GiB	C
125	2948	1.04393	99	$1h$ $18m$	$47.6h$	5.2 GiB	C
124	2937	1.04032	98	$39m$	$23.9h$	4.4 GiB	C
123	2950	1.04994	93	$7m$	$4.0h$	2.2 GiB	C

New candidate solutions to Exact-SVP

SVP dim	Norm	Hermite factor	Sieve max dim	Wall time	Total CPU time	Memory usage	M.$^+$
136	2934	0.99621	112	$18h$ $29m^*$	$704h$	28.5 GiB	C
135	2958	1.00920	108	$6h$ $26m^*$	$244h$	16.2 GiB	C
133	2909	0.99940	103	$2h$ $59m^*$	$112.4h$	12.1 GiB	C
131	2904	1.00465	108	$7h$ $51m^*$	$302.6h$	16.1 GiB	C
129	2875	0.99878	106	$5.2h^*$	$199.3h$	12.0 GiB	C

*: Continued from previously reduced basis for the 1.05-Hermite-SVP solution.
†: Not measured, estimate

Table 2: Performances on Darmstadt SVP challenges

are rather heterogeneous – different machines, different software versions, and different parametrisations were used – and therefore discourage extrapolations. Moreover the design decisions below and the probabilistic nature of the algorithm explain the non monotonic time and space requirements.

The parameters were optimised towards speed by trial-and-error on many smaller instances ($d \approx 100$). More specifically we ran `WorkOut` with parameters $f = 16 + d/12$, $f^+ = 3$, $s = 1$; choosing $f^+ = 1$ or 2 would cost more time and less memory.[17] The loop was set to exit as soon as a vector of the desired length was found, and if it reached the minimal value of f, it would repeat this largest `Pump` until success (this repetition rarely happened more than three times). The sieve max-dim column reports the actual dimension $d - f_{\text{last}}$ of the last `Pump`.

6.3 BKZ

To test `PumpNJumpTour` we compare its quality vs. time performance against BKZ 2.0 [CN11] in FPyLLL and against `NaiveTour` (see Figure 4). We generate random q-ary lattice bases, of dimension 180 with 90 equations modulo $q = 2^{30}$. We prereduce the bases using one FPyLLL BKZ tour for each blocksize from 20 to 59 and then report the cumulative time taken by further progressive tours of several BKZ variants.

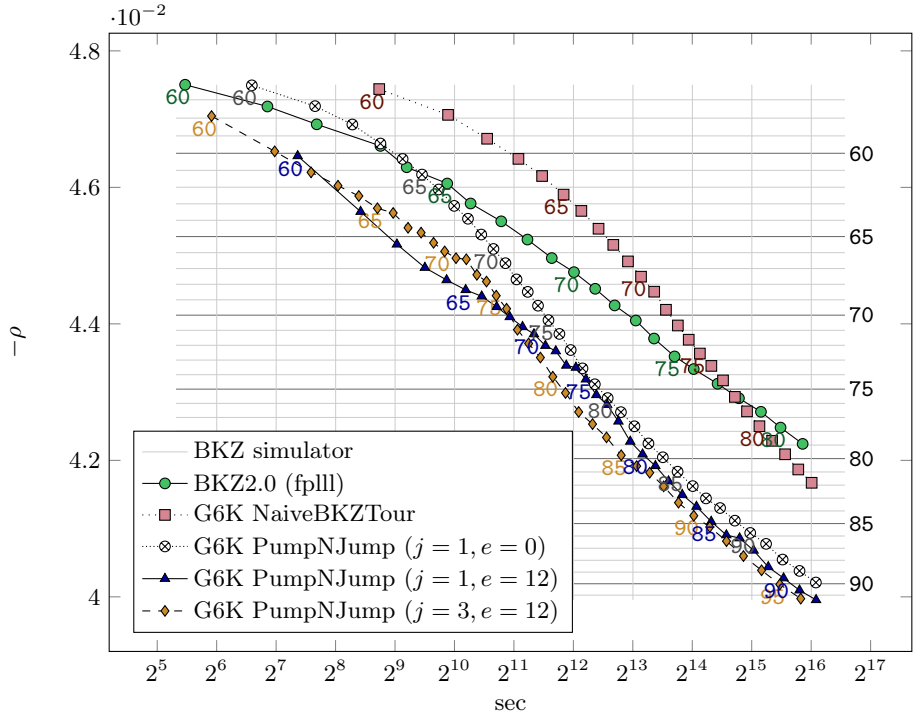
Contrary to exact-SVP, we find it beneficial for the running time to activate sieving during pump-down for all G6K based BKZ experiments. We further find that `triple_sieve` is noticeably faster than `bgj1`; it seems that the former suffers fewer collisions than the latter when sieving during the pump-down phase.

For all G6K based BKZ experiments we choose the number of dimensions for free following the experimental fit of Figure 3b, that is $f = 11.5 + 0.075\beta$. We also introduce a parameter $e = f' - f$ to concretise the more opportunistic `PumpNJumpTour` variant discussed at the end of Section 4.3.

To measure quality we use an averaged quality measurement, namely, the slope metric of FPyLLL. This slope, ρ, is a least squares fit of the $\log |\mathbf{b}_i^*|^2$. For comparison this metric is preferable to the typical root Hermite factor as it displays much less variance. In the GSA model, the slope ρ relates to the root Hermite factor by $\delta = \exp(-\rho/4)$. We also provide the predictions for progressive tours given by the BKZ simulator of [CN11, Wal16]. We note that the simulator is optimistic compared to even the most "textbook" variants, BKZ2.0 and `NaiveTour`, a phenomenon already documented in [YD17, **?**].

Conclusion. These experiments confirm that it is possible to outperform a naïve application of an SVP-β oracle to obtain a quality equivalent to BKZ-β in less time. Indeed, $\texttt{PumpNJumpTour}_{\beta,f,1}$ is about 4 times faster than $\texttt{NaiveTour}_{\beta,f}$ for the same reduction quality. Furthermore, the opportunistic variant with $e = 12$ gives even better quality per time, and also only requires a smaller β for the same quality, therefore decreasing memory consumption. These experiments also suggest that jumps $j > 1$ are not beneficial, they require similar running time per quality, but with a larger memory consumption.

[17] The number $f = 16 + d/12$ of dimensions for free is only meant to be a local approximation, as we asymptotically expect $f = \Theta(d/\log d)$ even for $O(1)$-approx-SVP [Duc18a].



The time and slope are averaged over 8 instances for each algorithm. Each instance was monothreaded, but ran in parallel (40threads/40cores, not hyper-threaded) on machine S. We label the point by β for all multiples of 5. Raw data embedded.

Fig. 4: Performance of BKZ-like algorithms.

6.4 LWE

The Darmstadt LWE challenges [FY15] are labeled by (n, α), where n denotes the dimension of the secret in $\mathbb{Z}_q$, for some q, and α is a noise rate. Concretely the challenges are given as $(\mathbf{A}, \mathbf{b})$ where $\mathbf{As} + \mathbf{e} \equiv \mathbf{b} \mod q$ with $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ for some m, $\mathbf{s} \in \mathbb{Z}_q^n, \mathbf{e} \in \mathbb{Z}^m$ and $\mathbf{b} \in \mathbb{Z}_q^m$. Each entry of $\mathbf{e}$, the error, is sampled independently from the Gaussian distribution over the integers with mean $\mu = 0$ and standard deviation $\sigma = \alpha \cdot q$, while the entries of $\mathbf{A}$ and $\mathbf{s}$ are sampled independently and uniformly from $\mathbb{Z}_q$. Both q and m are constant for a given n, but increase with n.

Our method for solving LWE is via embedding $\mathbf{e}$ into a uSVP instance [Kan87, BG14] but using the success condition originally given in [ADPS16] and experimentally justified in [AGVW17]. We also use the embedding coefficient $t = 1$ following [ADPS16, AGVW17]. We choose the minimal β such that after BKZ-β reduction, $|\pi_{d-\beta}(\mathbf{e})| < \mathrm{gh}(\mathcal{L}_{d-\beta,d})$. Therefore $\pi_{d-\beta}(\mathbf{e})$ will be inserted at index $d - \eta$. It is shown in [AGVW17] that size reduction (here lifting) is then enough

(n, α)	Estimated (β, η, d)	Successful (β, ν, ν')	CPU time	Wall time	M.
$(65, 0.010)$	$(108, 137, 244)$	$(112, 124, 120)$	$2553h$	$60h$	A
$(55, 0.015)$	$(106, 135, 219)$	$(110, 125, 103)$	$2198h$	$34h\ 50m$	S
$(40, 0.030)$	$(102, 133, 179)$	$(108, 120, 111)$	$1116h$	$17h\ 43m$	S
$(75, 0.005)$	$(88, 118, 252)$	$(88, 112, 107)^{\ddagger}$	$591h$	$12h\ 26m$	S
$(60, 0.010)$	$(92, 122, 222)$	$(94, 112, 106)^{\dagger}$	$579h$	$11h\ 59m$	S
$(50, 0.015)$	$(87, 118, 194)$	$(81, 111, 95)$	$8h\ 36m$	$1h\ 23m$	S

†: There was also a failed search after $\beta = 90$, with $\nu = 115$.
‡: There was also a failed search after $\beta = 84$, with $\nu = 115$.

Table 3: Performances on Darmstadt LWE challenges.

to lift $\pi_{d-\beta}(\mathbf{e})$ to $\mathbf{e}$. The success condition presented in [ADPS16] is

$$\sqrt{\beta} \cdot \sigma < \delta(\beta)^{2\beta - d} \cdot \mathrm{Vol}\left(\mathcal{L}\right)^{1/d}. \tag{17}$$

There is no a priori reason why the β used for BKZ reduction and the dimension of the SVP call (the last full block in some BKZ-β tour) which first finds $\pi_{d-\beta}(\mathbf{e})$, should be equal. For enumeration based algorithms it is customary to run one large enumeration after the smaller enumerations inside BKZ, see [LN13]. To apply this to sieving we alter the above inequality to allow a "decoupling" of these quantities and then balance the expected total time cost.

Let β continue to denote the BKZ block size and η denote the dimension of an SVP call on the lattice $\mathcal{L}_{d-\eta,d}$. We obtain the following inequality

$$\sqrt{\eta} \cdot \sigma < \delta(\eta)^{\eta} \cdot \delta(\beta)^{\eta - d} \cdot \mathrm{Vol}\left(\mathcal{L}\right)^{1/d}. \tag{18}$$

The left hand side is an approximation of the length $\pi_{d-\eta}(\mathbf{e})$ and the right hand side an approximation of the Gaussian heuristic of $\mathcal{L}_{[d-\eta:d]}$. Indeed

$$\mathrm{gh}(\mathcal{L}_{[d-\eta:d]}) = \sqrt{\eta/2\pi e} \cdot \mathrm{Vol}\left(\mathcal{L}_{[d-\eta:d]}\right)^{1/\eta} = \sqrt{\eta/2\pi e} \cdot \left(\prod_{i=d-\eta}^{d-1} |\mathbf{b}_i^*| \right)^{1/\eta}, \tag{19}$$

and further $\delta(\eta)^{\eta} \sim \sqrt{\eta/2\pi e}$ in increasing η. By combining the GSA and the estimate the root Hermite factor gives for $|\mathbf{b}_0|$, (18) may be derived from (19).

Implemented strategy and performance. To solve LWE instances in practice we implemented code which returns triples (β, η, d) that satisfy (18), and choose the number of LWE samples accordingly. We then run `PumpNJumpTour` with $s = 1, j = 1, e = 12$ and `triple_sieve` as the underlying sieve, and increase β progressively (choosing f according to the SVP prediction). After each tour,

we measure the walltime T elapsed since the beginning of the reduction, and predict the maximal dimension ν reachable by pumping up within time T. We predict whether we expect to find the projected short vector in this pump (ignoring on-the-fly lifting), following the reasoning of [Duc18a]. That is, we check the inequality

$$\sqrt{\nu} \cdot \sigma \leq \sqrt{4/3} \cdot \mathrm{gh}(\mathcal{L}_{[d-\nu:d]}). \tag{20}$$

If this condition is satisfied, we proceed with searching for the LWE solution with this `Pump` (κ, $f = \nu - \kappa$, $\beta = d - \kappa$, $s = 0$)[18] otherwise, we continue BKZ reduction with larger β. If this search is triggered but fails, we also go back to reducing the basis with progressive BKZ, and reset the timer T. The search may also succeed before reaching the pump dimension ν, in which case we denote by ν' the dimension at which it stops.

Details of the six new Darmstadt LWE records are in Table 3. It should be noted that the CPU-time/walltime ratio can be quite far from e.g. 80, the number of threads on machine S. This is because parallelism only kicks in for sieves in large dimensions (see Appendix B), while the walltimes of some of the computations were dominated by BKZ tours with medium blocksizes. One could tailor the parameterisation to improve the walltime further, but this would be in vain as we are mostly interested in the more difficult instances, which suffer very little from this issue.

Acknowledgements

We thank Kenny Paterson for discussing a previous version of this draft. We also thank Pierre Karpman for running some of our experiments.

References

[ACD+18] Martin R. Albrecht, Benjamin R. Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W. Postlethwaite, Fernando Virdia, and Thomas Wunderer, *Estimate all the LWE, NTRU schemes!*, SCN 18 (Dario Catalano and Roberto De Prisco, eds.), LNCS, vol. 11035, Springer, Heidelberg, September 2018, pp. 351–367.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe, *Post-quantum key exchange—a new hope*, 25th USENIX Security Symposium (USENIX Security 16) (Austin, TX), USENIX Association, 2016, pp. 327–343.

[AGVW17] Martin R. Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer, *Revisiting the expected cost of solving uSVP and applications to LWE*, ASIACRYPT 2017, Part I (Tsuyoshi Takagi and Thomas Peyrin, eds.), LNCS, vol. 10624, Springer, Heidelberg, December 2017, pp. 297–322.

[18] One could choose $\kappa = 0$ to be entirely sure not to miss the solution during the lifting phase, but this increases the cost of lifting. Instead, we can choose κ such that $\sqrt{\kappa}\sigma < \mathrm{gh}(\mathcal{L}_{[d-\kappa:d]})$, with a small margin of, say, five dimensions.

[AKS01] Miklós Ajtai, Ravi Kumar, and D. Sivakumar, *A sieve algorithm for the shortest lattice vector problem*, 33rd ACM STOC, ACM Press, July 2001, pp. 601–610.

[AWHT16] Yoshinori Aono, Yuntao Wang, Takuya Hayashi, and Tsuyoshi Takagi, *Improved progressive BKZ algorithms and their precise cost estimation by sharp simulator*, EUROCRYPT 2016, Part I (Marc Fischlin and Jean-Sébastien Coron, eds.), LNCS, vol. 9665, Springer, Heidelberg, May 2016, pp. 789–819.

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven, *New directions in nearest neighbor searching with applications to lattice sieving*, 27th SODA (Robert Krauthgamer, ed.), ACM-SIAM, January 2016, pp. 10–24.

[BG14] Shi Bai and Steven D. Galbraith, *Lattice decoding attacks on binary LWE*, ACISP 14 (Willy Susilo and Yi Mu, eds.), LNCS, vol. 8544, Springer, Heidelberg, July 2014, pp. 322–337.

[BGJ15] Anja Becker, Nicolas Gama, and Antoine Joux, *Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search*, Cryptology ePrint Archive, Report 2015/522, 2015, http://eprint.iacr.org/2015/522.

[BLS16] Shi Bai, Thijs Laarhoven, and Damien Stehle, *Tuple lattice sieving*, Cryptology ePrint Archive, Report 2016/713, 2016, http://eprint.iacr.org/2016/713.

[Cha02] Moses Charikar, *Similarity estimation techniques from rounding algorithms*, 34th ACM STOC, ACM Press, May 2002, pp. 380–388.

[Che13] Yuanmi Chen, *Réduction de réseau et sécurité concrète du chiffrement complètement homomorphe*, Ph.D. thesis, Paris 7, 2013, Thèse de doctorat dirigée par Nguyen, Phong-Quang Informatique Paris 7 2013, p. 1 vol. (133 p.).

[CN11] Yuanmi Chen and Phong Q. Nguyen, *BKZ 2.0: Better lattice security estimates*, ASIACRYPT 2011 (Dong Hoon Lee and Xiaoyun Wang, eds.), LNCS, vol. 7073, Springer, Heidelberg, December 2011, pp. 1–20.

[dt18a] The FPLLL development team, *FPLLL, a lattice reduction library*, Available at https://github.com/fplll/fplll, 2018.

[dt18b] The FPyLLL development team, *FPyLLL, a lattice reduction library*, Available at https://github.com/fplll/fpylll, 2018.

[Duc18a] Léo Ducas, *Shortest vector from lattice sieving: A few dimensions for free*, EUROCRYPT 2018, Part I (Jesper Buus Nielsen and Vincent Rijmen, eds.), LNCS, vol. 10820, Springer, Heidelberg, April / May 2018, pp. 125–145.

[Duc18b] Léo Ducas, *Shortest Vector from Lattice Sieving: a Few Dimensions for Free (talk)*, https://eurocrypt.iacr.org/2018/Slides/Monday/TrackB/01-01.pdf, April 2018.

[FBB+15] Robert Fitzpatrick, Christian H. Bischof, Johannes Buchmann, Özgür Dagdelen, Florian Göpfert, Artur Mariano, and Bo-Yin Yang, *Tuning GaussSieve for speed*, LATINCRYPT 2014 (Diego F. Aranha and Alfred Menezes, eds.), LNCS, vol. 8895, Springer, Heidelberg, September 2015, pp. 288–305.

[FK15] Masaharu Fukase and Kenji Kashiwabara, *An accelerated algorithm for solving SVP based on statistical analysis*, JIP **23** (2015), no. 1, 67–80.

[FP85] U. Fincke and M. Pohst, *Improved methods for calculating vectors of short length in a lattice, including a complexity analysis*, Mathematics of Computation **44** (1985), no. 170, 463–463.

[FY15] Florian Göpfert F and A Yakkundimath, *Darmstadt LWE Challenges*, https://www.latticechallenge.org/lwe_challenge/challenge.php, 2015, Accessed: 15-08-2018.

[GN08a] Nicolas Gama and Phong Q. Nguyen, *Finding short lattice vectors within Mordell's inequality*, 40th ACM STOC (Richard E. Ladner and Cynthia Dwork, eds.), ACM Press, May 2008, pp. 207–216.

[GN08b] ———, *Predicting lattice reduction*, EUROCRYPT 2008 (Nigel P. Smart, ed.), LNCS, vol. 4965, Springer, Heidelberg, April 2008, pp. 31–51.

[GNR10] Nicolas Gama, Phong Q. Nguyen, and Oded Regev, *Lattice enumeration using extreme pruning*, EUROCRYPT 2010 (Henri Gilbert, ed.), LNCS, vol. 6110, Springer, Heidelberg, May / June 2010, pp. 257–278.

[HK17] Gottfried Herold and Elena Kirshanova, *Improved algorithms for the approximate k-list problem in euclidean norm*, PKC 2017, Part I (Serge Fehr, ed.), LNCS, vol. 10174, Springer, Heidelberg, March 2017, pp. 16–40.

[HKL18] Gottfried Herold, Elena Kirshanova, and Thijs Laarhoven, *Speed-ups and time-memory trade-offs for tuple lattice sieving*, PKC 2018, Part I (Michel Abdalla and Ricardo Dahab, eds.), LNCS, vol. 10769, Springer, Heidelberg, March 2018, pp. 407–436.

[HPS11] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé, *Analyzing blockwise lattice algorithms using dynamical systems*, CRYPTO 2011 (Phillip Rogaway, ed.), LNCS, vol. 6841, Springer, Heidelberg, August 2011, pp. 447–464.

[HS07] Guillaume Hanrot and Damien Stehlé, *Improved analysis of kannan's shortest lattice vector algorithm*, CRYPTO 2007 (Alfred Menezes, ed.), LNCS, vol. 4622, Springer, Heidelberg, August 2007, pp. 170–186.

[Kan83] Ravi Kannan, *Improved algorithms for integer programming and related lattice problems*, 15th ACM STOC, ACM Press, April 1983, pp. 193–206.

[Kan87] Ravi Kannan, *Minkowski's convex body theorem and integer programming*, Mathematics of Operations Research **12** (1987), no. 3, 415–440.

[Kir16] Paul Kirchner, *Re: Sieving vs. enumeration*, https://groups.google.com/forum/#!msg/cryptanalytic-algorithms/BoSRLOuHIjM/wAkZQlwRAgAJ,, May 2016.

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Ĺovasz, *Factoring polynomials with rational coefficients*, Mathematische Annalen **261** (1982), no. 4, 515–534.

[LM18] Thijs Laarhoven and Artur Mariano, *Progressive lattice sieving*, Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018 (Tanja Lange and Rainer Steinwandt, eds.), Springer, Heidelberg, 2018, pp. 292–311.

[LN13] Mingjie Liu and Phong Q. Nguyen, *Solving BDD by enumeration: An update*, CT-RSA 2013 (Ed Dawson, ed.), LNCS, vol. 7779, Springer, Heidelberg, February / March 2013, pp. 293–309.

[MV10a] Manfred G. Madritsch and Brigitte Vallée, *Modelling the LLL algorithm by sandpiles*, LATIN 2010 (Alejandro López-Ortiz, ed.), LNCS, vol. 6034, Springer, Heidelberg, April 2010, pp. 267–281.

[MV10b] Daniele Micciancio and Panagiotis Voulgaris, *Faster exponential time algorithms for the shortest vector problem*, 21st SODA (Moses Charika, ed.), ACM-SIAM, January 2010, pp. 1468–1480.

[MW15] Daniele Micciancio and Michael Walter, *Fast lattice point enumeration with minimal overhead*, 26th SODA (Piotr Indyk, ed.), ACM-SIAM, January 2015, pp. 276–294.

[Ngu10] Phong Q. Nguyen, *The lll algorithm: Survey and applications*, ch. Hermite's Constant and Lattice Algorithms, pp. 19–69, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[NV08] Phong Q. Nguyen and Thomas Vidick, *Sieve algorithms for the shortest vector problem are practical*, J. Mathematical Cryptology **2** (2008), no. 2, 181–207.

[PAA+17] Thomas Poppelmann, Erdem Alkim, Roberto Avanzi, Joppe Bos, Leo Ducas, Antonio de la Piedra, Peter Schwabe, and Douglas Stebila, *Newhope*, Tech. report, National Institute of Standards and Technology, 2017, available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[Sch87] Claus-Peter Schnorr, *A hierarchy of polynomial time lattice basis reduction algorithms*, Theor. Comput. Sci. **53** (1987), 201–224.

[Sch03] Claus Peter Schnorr, *Lattice reduction by random sampling and birthday methods*, STACS 2003 (Berlin, Heidelberg) (Helmut Alt and Michel Habib, eds.), Springer Berlin Heidelberg, 2003, pp. 145–156.

[SE94] C. P. Schnorr and M. Euchner, *Lattice basis reduction: Improved practical algorithms and solving subset sum problems*, Mathematical Programming **66** (1994), no. 1, 181–199.

[SG10] Michael Schneider and Nicolas Gama, *Darmstadt SVP Challenges*, https://www.latticechallenge.org/svp-challenge/index.php, 2010, Accessed: 17-08-2018.

[TKH18] Tadanori Teruya, Kenji Kashiwabara, and Goichiro Hanaoka, *Fast lattice basis reduction suitable for massive parallelization and its application to the shortest vector problem*, PKC 2018, Part I (Michel Abdalla and Ricardo Dahab, eds.), LNCS, vol. 10769, Springer, Heidelberg, March 2018, pp. 437–460.

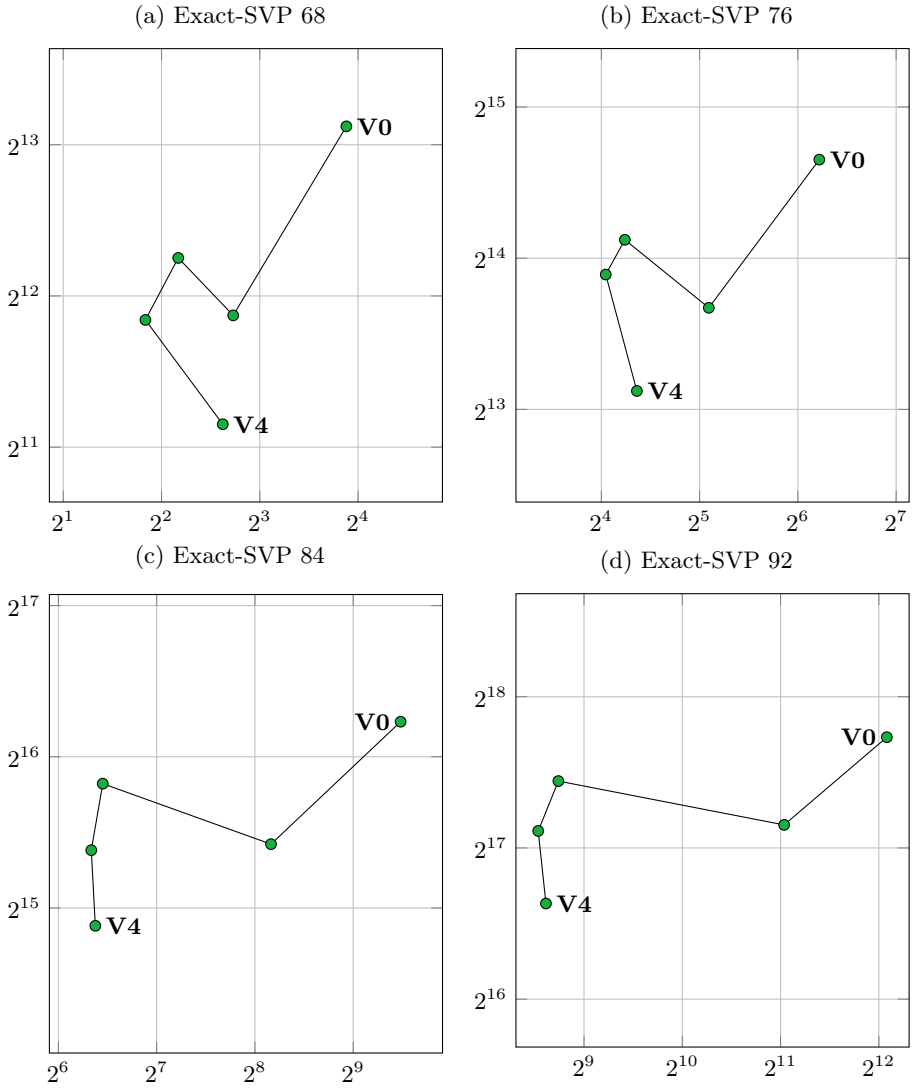
[Wal16] Michael Walter, *Sage implementation of Chen and Nguyen's BKZ simulator*, http://pub.ist.ac.at/~mwalter/src/sim_bkz.sage, 2016.

[YD17] Yang Yu and Léo Ducas, *Second order statistical behavior of LLL and BKZ*, SAC 2017 (Carlisle Adams and Jan Camenisch, eds.), LNCS, vol. 10719, Springer, Heidelberg, August 2017, pp. 3–22.

A Details of the Efficiency Improvements

In this appendix, we report on the separate contributions of each of our new features to improving the efficiency of finding a solution for exact-SVP, by activating them one at a time. The comparison is depicted in Figure 5. Below, we describe all variants considered for this comparison. We caution the reader of the somewhat noisy nature of these experiments, as many parameters can influence the time-memory trade-off.

V0: Baseline reimplementation of [Duc18a]. Our starting point is a reimplementation of the iterated `SubSieve`$^+$ of [Duc18a] within G6K (or more precisely, an algorithm as close as possible given the G6K instructions) using none of the new features introduced in this work. It is already faster than that reported for the original implementation of [Duc18a] for dimension 80 by a factor of about $2^{0.9}$, using the same machine as [Duc18a] (Intel Core i7-4790 @3.6Ghz, 4 cores). Unfortunately, [Duc18a] does not report on the size of the database.



Comparison of running time (x-axis, seconds) and memory consumption (y-axis, $|db|$) of the 5 variants V0 to V4 described above. Data are averaged over 40 samples. Each instance was monothreaded, but ran in parallel (40threads/40cores, not hyper-threaded) on machine S. Raw data embedded.

Fig. 5: Comparison of the variants V0 to V4 for increasing dimension.

V1: Enabling the `pump`. We switch to our new high level algorithms, essentially replacing the right-progressive sieve by a left-progressive sieve. Another change is the use of flexible insertion policy. This modification gives a substantial improvement on both time (factor ≈ 2) and memory consumption (factor $\approx 2^{0.5}$).

V2: Switching to 3-sieve [HK17]. We switch from the Gauss sieve to the 3-sieve[19], while still using the same asymptotic amount of memory. This decreases the asymptotic running time from $2^{0.415n+o(n)}$ to $2^{0.349n+o(n)}$. In practice, and with the default parameterisation, this costs a bit more memory (a factor $\approx 2^{0.3}$): while our Gauss sieve automatically adjusts its database size, the 3-sieve uses a predetermined size. As expected, this saves substantial amounts of time, namely a factor $\approx 2^{2.3}$ in dimension 92, and this saving grows with dimension.

V3: Activating on-the-fly lifting. We activate the on-the-fly lifting functionality within G6K. This feature saves a substantial amount of memory ($\approx 2^{0.5}$) at no cost to the running time (or even a minor gain).

V4: Activating sieve during `pump-down`. We activate the pump during the descent phase of the pump ($s = 1$). This feature saves a substantial amount of memory ($\approx 2^{0.5}$) at a minor cost to the running time.[20]

B Parallelism Efficiency

To compare the parallel performance of our implementation, we consider our core sieving algorithm by running a full `pump` and deactivating dimensions for free and on-the-fly lifting. We experiment with two types of parallelism, namely "job parallelism" (independent tasks for each thread, denoted "j") and "thread parallelism" (all threads collaborating on a single task, denoted "t"). Experiments with hyper-threading are marked with a $*$. The purpose of the "job parallelism" experiments is to assess whether resources other than CPU-cycles are limiting, such as RAM or cache bandwidth. The purpose of the "thread parallelism" experiments is to assess how much time is spent idle by threads and the relationship between walltime and CPU-time. For a clearer comparison, we deactivated "frequency-scaling" which is used by most multi-core CPUs for power and heat management.

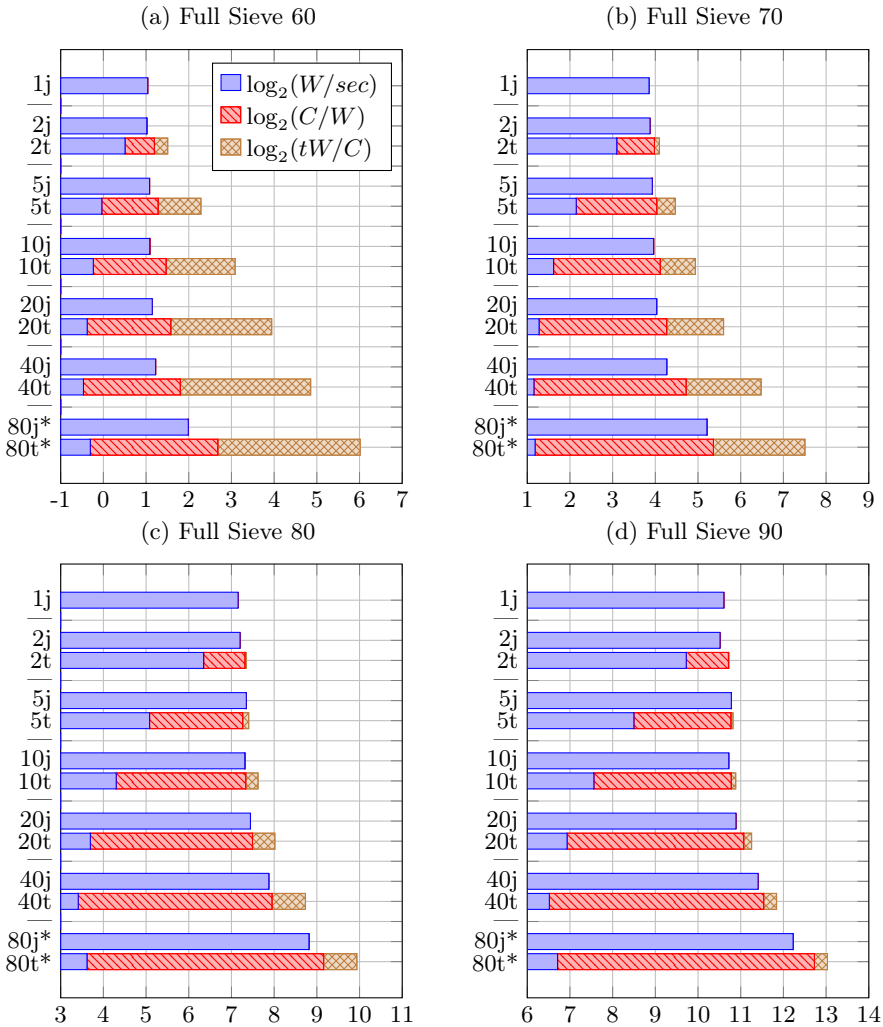
Our experiments are reported in Figure 6. The blue bars correspond to the walltime in a logarithmic scale. The blue and red bars together correspond to CPU-time. The brown bars represent idleness, i.e. the cumulative time spent by threads doing nothing.

We note that for the "thread parallel" experiments, sources of idleness (and thus sublinear performance improvements) are: threads waiting for a lock to be released and time spent in an non-parallel routine. We can also observe computational inefficiency, i.e. the amount of extra operations required by thread-parallelism. It is expressed by the difference between the blue bar for (say) 10j with the blue + red bar for 10t.

[19] Since writing the main body of this report, improvements to the implementation of the 3-sieve have allowed it to slightly outperform `bgj1`, but not sufficiently to warrant a rerun of all our experiments.

[20] As mentionned earlier, this feature seems to cost more time when combined with the `bgj1` sieve than with the 3-sieve.

First, observing job parallelism, we note RAM or cache bandwidth does not seem to be an issue, as we observe only minor slowdowns when increasing job parallelism. We also note that hyperthreading is not completely useless: the walltime increases by a factor slightly less than two between experiments 40j and 80j*.



Legend: W = walltime, C = CPU-time, and t = threads per job. The bars are "stacked", e.g. the right end of the red bar corresponds to $\log_2(W/\text{sec})+\log(C/W) = \log_2(C/\text{sec})$. These experiments were run on machine S, and averaged over 5 samples.

Fig. 6: Parallel performance.

As for thread parallelism, in small dimensions, there is a noticeable amount of computational inefficiency, and a huge amount of idleness. Nevertheless, these inefficiencies seem to fade away with increasing dimension.

Thus, if the task at hand allows it, it is clearly preferable to choose job parallelism over thread parallelism in small dimensions. However, it should be noted that the memory requirement grows with the number of jobs, not with the number of threads: in large dimensions one may be forced to opt for thread parallelism even if it incurs a small slowdown.