Advanced Lattice Sieving on GPUs, with Tensor Cores

Léo Ducas[1] and Marc Stevens[1] and Wessel van Woerden[1]

CWI, Amsterdam, The Netherlands

Abstract. In this work, we study GPU implementations of various state-of-the-art sieving algorithms for lattices (Becker-Gama-Joux 2015, Becker-Ducas-Gama-Laarhoven 2016, Herold-Kirshanova 2017) inside the General Sieve Kernel (G6K, Albrecht *et al.* 2019). In particular, we extensively exploit the recently introduced Tensor Cores – originally designed for raytracing and machine learning – and demonstrate their fitness for the cryptanalytic task at hand. We also propose a new *dual-hash* technique for efficient detection of 'lift-worthy' pairs to accelerate a key ingredient of G6K: finding short lifted vectors.
We obtain new computational records, reaching dimension 180 for the SVP Darmstadt Challenge improving upon the previous record for dimension 155. This computation ran for 51.6 days on a server with 4 NVIDIA Turing GPUs and 1.5TB of RAM. This corresponds to a gain of about two orders of magnitude over previous records both in terms of wall-clock time and of energy efficiency.

Keywords: Lattice Sieving, Shortest Vector, G6K, Cryptanalysis, Challenges.

1 Introduction

Lattice reduction is a key tool in cryptanalysis at large, and is of course a central interest for the cryptanalysis of lattice-based cryptography. With the expected standardisation of lattice-based cryptosystems, the question of the precise performance of lattice reduction algorithms is becoming a critical one. The crux of the matter is the cost of solving the Shortest Vector Problem (SVP) with sieving algorithms. While even in the RAM model numerous questions remain regarding the precise cost of the fastest algorithms, one may also expect a significant gap between this model and practice, due to their high-memory requirements.

Lattice sieving algorithms [AKS01, NV08, MV10] are asymptotically superior to enumeration techniques [FP85, Kan83, SE94, GNR10], but this has only recently been shown in practice. Recent progress on sieving, both on its theoretical [Laa15, BGJ15, BDGL16, HKL18] and practical performances [FBB+14, Duc18, LM18, ADH+19], brought the cross-over point with enumeration as low as dimension 80. The work of Albrecht *et al.* at Eurocrypt 2019, named the General Sieve Kernel (G6K), set new TU Darmstadt SVP-records [SG10] on a single machine up to dimension 155, while before the highest record was at 152 using a cluster with multiple orders of magnitude more core-hours of computation.

Before scaling up to a cluster of computers, a natural step is to port cryptanalytic algorithms to Graphical Processing Units (GPUs); not only are GPUs far more efficient for certain parallel tasks, but their bandwidth/computation capacity ratio are already more representative of the difficulties to expect when scaling up beyond a single computational server. This step can therefore already teach us a great deal about how a cryptanalytic algorithm should scale in practice. The only GPU implementation of sieving so far [YKYC17] did not make use of advanced algorithmic techniques (such as the Nearest Neighbour Search techniques, Progressive Sieving or the Dimensions for Free technique [Laa15,LM18,Duc18]), and is therefore not very representative of the current state of the art.

An important consideration for assessing practical cryptanalysis is the direction of computation technologies, and one should in particular note the advent of *Tensor* architectures [JYP$^+$17], offering extreme performance for low-precision matrix multiplication. While this development has been mostly motivated by machine learning applications, the potential application for cryptanalytic algorithms must also be considered. Interestingly, such architectures are now also available on commodity GPUs (partly motivated by ray-tracing applications), and therefore accessible even with modest resources.

1.1 Contributions

The main contribution of this work is to show that lattice sieving, including the more complex and recent algorithmic improvements, can effectively be accelerated by GPUs. In particular, we show that the NVIDIA Tensor cores, only supporting specific low-precision computations, can be used efficiently for lattice sieving. We exhibit how the most computationally intensive parts of complex sieving algorithms can be executed in low-precision even in large dimensions.

We show and demonstrate by an implementation that the use of Tensor cores results in large efficiency gains for cryptanalytic attacks, both in hardware and energy costs. We present several new computational records, reaching dimension 180 for the TU Darmstadt SVP challenge record with a single high-end machine with 4 GPUs and 1.5TB RAM in 51.6 days. Not only did we break SVP-records significant faster, but also with $< 4\%$ of the energy cost compared to a CPU only attack. For instance, we solved dimension 176 using less time and with less than 2 times the overall energy cost compared to the previous record of dimension 155. Furthermore by re-computing data at appropriate points in our algorithms we reduced the memory usage per vector by 60% compared to the base G6K implementation with minimal computational overhead.

Our work also includes the first implementation of asymptotically best sieve (BDGL) from [BDGL16] inside the G6K framework, both for CPU-only (multi-threaded and `AVX2`-optimized) and with GPU acceleration. We use this to shed some light on the practicality of this algorithm. In particular we show that our CPU-only BDGL-sieve already improves over the previous record-holding sieve in dimensions as low as 95, but that this cross-over point lies much higher for our GPU accelerated sieve due to memory-bottleneck constraints.

One key feature of G6K is to also consider lifts of pairs even if such a pair is not necessarily reducible, so as to check whether such lifts are short; the more such pairs are lifted, the more dimensions for free one can hope for [Duc18, ADH+19]. Yet, Babai lifting of a vector has quadratic running time which makes it too expensive to apply to each pair. We introduce a filter based on dual vectors that detects whether pairs are worth lifting. With adequate pre-computation on each vector, filtering a pair for lifting can be made linear-time, fully parallelizable, and very suitable to implement on GPUs.

Open Source Code. Since the writing of this report, our CPU implementation of `bdgl` has been integrated in G6K, with further improvements, and we aim for long term maintenance.[1] The GPU implementations has also been made public, but with lower expectation of quality, documentation and maintenance.[2]

Acknowledgments. The authors would like to express their gratitude to Joe Rowell for his precious feedback and support on parts of our code.

The research of L. Ducas was supported by the European Union's H2020 Programme under PROMETHEUS project (grant 780701) and the ERC-StG-ARTICULATE project (no. 947821). W. van Woerden is funded by the ERC-ADG-ALGSTRONGCRYPTO project (no. 740972). The computational hardware enabling this research was acquired thanks to a Veni Innovational Research Grant from NWO under project number 639.021.645 and to the Google Security and Privacy Research Award awarded to M. Stevens.

2 Preliminaries

2.1 Lattices and the Shortest Vector Problem

Notation. Given a matrix $\mathbf{B} = (\mathbf{b}_0, \ldots, \mathbf{b}_{d-1}) \subset \mathbb{R}^d$ with linearly independent columns, we define the lattice generated by the basis $\mathbf{B}$ as $\mathcal{L}(\mathbf{B}) := \{\sum_i^d x_i \mathbf{b}_i : x_i \in \mathbb{Z}\}$. We denote the volume of the fundamental area $\mathbf{B} \cdot [0,1]^d$ by $\det(\mathcal{L}) := |\det(\mathbf{B})|$. Given a basis $\mathbf{B}$ we define π_i as the projections orthogonal to the span of $(\mathbf{b}_0, \ldots, \mathbf{b}_{i-1})$ and the Gram-Schmidt orthogonalisation as $\mathbf{B}^* = (\mathbf{b}_0^*, \ldots, \mathbf{b}_{d-1}^*)$ where $\mathbf{b}_i^* := \pi_i(\mathbf{b}_i)$. The projected sublattice $\mathcal{L}_{[l:r]}$ where $0 \le l < r \le d$ is defined as the lattice with basis $\mathbf{B}_{[l:r]} := (\pi_l(\mathbf{b}_l), \ldots, \pi_l(\mathbf{b}_{r-1}))$. Note that the Gram-Schmidt orthogonalisation of $\mathbf{B}_{[l:r]}$ is induced by $\mathbf{B}^*$ and equals $(\mathbf{b}_l^*, \ldots, \mathbf{b}_{r-1}^*)$; consequently $\det(\mathcal{L}_{[l:r]}) = \prod_{i=l}^{r-1} \|\mathbf{b}_i^*\|$. When working with the projected sublattice $\mathcal{L}_{[l:r]}$ and the associated basis $\mathbf{B}_{[l:r]}$ we say that we work in the *context* $[l:r]$.

[1] https://github.com/fplll/g6k/pull/61

[2] https://github.com/WvanWoerden/G6K-GPU-Tensor

The Shortest Vector Problem. The computationally hard problem on which lattice-based cryptography is based relates to the Shortest Vector Problem (SVP), which given a basis asks for a non-zero lattice vector of minimal length. More specifically, security depends on approximate versions of SVP, where we only try to find a non-zero lattice vector at most a factor $\mathrm{poly}(d)$ longer than the minimal length. However, via block reduction techniques like (D)BKZ [SE94, MW16] or slide reduction [GN08, ALNSD20], the approximate version can be reduced to a polynomial number of exact SVP instances in a lower dimension.

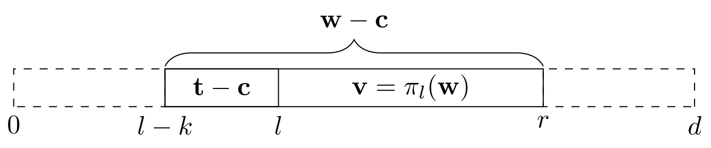
Definition 1 (Shortest Vector Problem (SVP)) *Given a basis* $\mathbf{B}$ *of a lattice* $\mathcal{L}$*, find a non-zero lattice vector* $\mathbf{v} \in \mathcal{L}$ *of minimal length* $\lambda_1(\mathcal{L}) := \min_{\mathbf{0} \neq \mathbf{w} \in \mathcal{L}} \|\mathbf{w}\|$.

For the purpose of cryptanalysis, SVP instances are typically assumed to be random, in the sense that they are distributed close to the Haar measure [GM03]. While the exact distribution is irrelevant, it is assumed for analysis that these instances follow the Gaussian Heuristic for 'nice' volumes K; which is widely verified to be true for lattices following the Haar measure.

Heuristic 1 (The Gaussian Heuristic (GH)) *Let* $K \subset \mathbb{R}^d$ *be a measurable body, then the number* $|K \cap \mathcal{L}|$ *of lattice points in* K *is approximately equal to* $\mathit{Vol}(K)/\det(\mathcal{L})$.

Note that the number of lattice points the Gaussian Heuristic indicates is exactly the expected number of lattice points in a random translation of K. When applying the Gaussian Heuristic to a d-dimensional ball of volume $\det(\mathcal{L})$ we obtain that the minimal length $\lambda_1(\mathcal{L})$ is approximately the radius of this ball, which asymptotically means that $\lambda_1(\mathcal{L}) \approx \sqrt{d/(2\pi e)} \cdot \det(\mathcal{L})^{1/d}$. For a lattice $\mathcal{L} \subset \mathbb{R}^d$ we denote this radius by $\mathrm{gh}(\mathcal{L})$, and to shorten notation we denote $\mathrm{gh}(l : r) := \mathrm{gh}(\mathcal{L}_{[l:r]})$. In practice for random lattices the minimal length deviates at most 5% from the predicted value starting around dimension 50, and even less in larger dimensions [GNR10, Che13]. Note that a ball of radius $\delta \cdot \mathrm{gh}(\mathcal{L})$ contains an exponential number of δ^d lattice vectors not much longer than the minimal length. We say that a list of lattice vectors *saturates* a volume K if it contains some significant ratio (say 50%) of the lattice vectors in $\mathcal{L} \cap K$ as predicted by the Gaussian Heuristic.

Lifting and dimensions for free. We discuss how to change context without increasing the length of vectors too much. Extending the context to the right (from $[l : r]$ to $[l : r + k]$) is merely following the inclusion $\mathcal{L}_{[l:r]} \subset \mathcal{L}_{[l:r+k]}$. Extending the context on the left is more involved. To *lift* a vector $\mathbf{v}$ from $\mathcal{L}_{[l:r]}$ to $\mathcal{L}_{[l-k:r]}$ for $0 \leq k \leq l$ we have to undo the projections away from $\mathbf{b}^*_{l-k}, \ldots, \mathbf{b}^*_{l-1}$. Such a lift is not unique, e.g., if $\mathbf{w} \in \mathcal{L}_{[l-k:r]}$ projects to $\mathbf{v}$, then so would the infinite number of lattice vectors $\mathbf{w} - \mathbf{c}$ with $\mathbf{c} \in \mathcal{L}_{[l-k:l]}$, and our goal is to find a rather short one.


$\mathbf{w} - \mathbf{c}$
$\mathbf{t} - \mathbf{c}$
$\mathbf{v} = \pi_l(\mathbf{w})$
0
$l - k$
l
r
d

Note that we can orthogonally decompose any lift as $\mathbf{w} - \mathbf{c} = (\mathbf{t} - \mathbf{c}) + \mathbf{v}$ with $\mathbf{t} \in \mathrm{span}(\mathcal{L}_{[l-k:l]}), \mathbf{c} \in \mathcal{L}_{[l-k:l]}$ and $\mathbf{v} \in \mathcal{L}_{[l:r]}$. So each lift has squared length $\|\mathbf{t} - \mathbf{c}\|^2 + \|\mathbf{v}\|^2$ and to minimize this we need to find a lattice vector $\mathbf{c} \in \mathcal{L}_{[l-k:l]}$ that lies close to $\mathbf{t}$. Note that even if we find a closest lattice point the added squared length $\|\mathbf{t} - \mathbf{c}\|^2$ is lower bounded by $\mathrm{dist}^2(\mathbf{t}, \mathcal{L}_{[l-k:l]})$. Instances for which this distance is very small are better known as δ-BDD (Bounded Distance Decoding) instances, where δ indicates the maximum distance of the target to the lattice.

Finding a close lattice point is at least as hard as finding a short vector, so for optimal lifts one would need the dimension k to stay small. E.g., for a 1-dimensional lattice the problem is equivalent to integer rounding. A famous polynomial time algorithm to find a somewhat close lattice point is Babai's nearest plane algorithm: lift in 1-dimensional steps $[l : r] \to [l - 1 : r] \to \cdots \to [l - k : r]$, greedily finding the closest lattice point in the 1-dimensional lattices $\mathbf{b}^*_{l-1}\mathbb{Z}, \ldots, \mathbf{b}^*_{l-k}\mathbb{Z}$. Babai's nearest plane algorithm finds a lattice point at squared distance at most $\frac{1}{4}\sum_{i=l-k}^{l-1} \|\mathbf{b}^*_i\|^2$, and always returns the closest lattice point for δ-BDD instances with $\delta \leq \frac{1}{2}\min_{l-k \leq i < l} \|\mathbf{b}^*_i\|$.

Lifting vectors to a larger context on the left increases their length. However under reasonable assumptions one can think of $\|\mathbf{b}^*_0\|, \ldots, \|\mathbf{b}^*_{d-1}\|$ as a decreasing sequence, which means that the minimal length over the extended context can be much larger than that of the original under the Gaussian Heuristic. So even though a vector becomes larger from lifting in the absolute sense, it can actually become shorter relatively to the context. Consequently lifting many short lattice vectors from $\mathcal{L}_{[l:d]}$ can result in finding a shortest vector in the full lattice $\mathcal{L}_{[0:d]} = \mathcal{L}$. Note that such a successful lift corresponds exactly to BDD instances, as the added length cannot be too large. When lifting a single-exponential number of short vectors, then l can be as large as $O(d/\log(d))$ [Duc18]. So for SVP algorithms that happen to find an exponential number of short vectors (instead of just a shortest), it suffices to run in a lower dimension; luckily lattice sieving algorithms do precisely that, essentially getting $O(d/\log(d))$ dimensions for free.

Lattice Sieving. Lattice sieving algorithms are among the current best asymptotic algorithms to solve SVP, running in single exponential time and memory. Sieving algorithms start with an exponentially large database of lattice vectors and try to find sums and differences of these vectors that are relatively short. These shorter combinations, which we call *reductions*, are inserted back into the database, possibly replacing longer vectors. The search for reductions is repeated until the database contains many short vectors, among which (hopefully) one of minimal length. As we do not know the exact length of the shortest vector a priori we need to fall back to alternative stopping conditions. In line with the dimensions for free technique explained before it makes sense to stop when the database saturates a ball with some *saturation radius* R, i.e., when the database contains a significant ratio of the short lattice vectors of length at most R. A simple sieving algorithm is summarized in Algorithm 1.

Provably solving SVP with lattice sieving leads to many technical problems like showing that we can actually find enough short combinations and in particular that they are new, i.e., they are not present in our database yet; unfortunately side-stepping these technicalities leads to high time and memory complexities [AKS01, MV10, PS09]. In contrast there are also sieving algorithms based mainly on the Gaussian and similar heuristics and these do fall in the practical regime. The first and simplest of these practical sieving algorithms by Nguyen and Vidick uses a database of $N = (4/3)^{d/2+o(d)} = 2^{0.2075d+o(d)}$ vectors and runs in time $N^{2+o(1)} = 2^{0.415d+o(d)}$ by repeatedly checking all pairs $\mathbf{v} \pm \mathbf{w}$ [NV08]. The database size of $(4/3)^{d/2+o(d)}$ is the minimal number of vectors that is needed in order to keep finding enough shorter pairs, and eventually saturate the ball of radius of $\sqrt{4/3} \cdot \mathrm{gh}(\mathcal{L})$. In a line of works [Laa15, BGJ15, BL16, BDGL16] the time complexity was gradually improved to $2^{0.292d+o(d)}$ by nearest neighbour searching techniques to find close pairs more efficiently. Instead of checking all pairs they first apply some bucketing strategy in which close vectors are more likely to fall into the same bucket. By only considering the somewhat-close pairs inside each bucket, the total number of checked pairs can be decreased. In order to lower the memory requirement of $2^{0.2075d+o(d)}$ one can also look at triplets of vectors in addition to pairs. This leads to a time-memory trade-off; lowering the memory cost while increasing the computational cost. The current best triple sieve with minimal memory $2^{0.1887d+o(d)}$ takes time $2^{0.3588d+o(d)}$ [HKL18].

2.2 The General Sieve Kernel

The General Sieve Kernel (G6K) [ADH+19] is a lattice reduction framework based on sieving algorithms that is designed to be 'stateful' instead of treating sieving as a black-box SVP oracle. This encompasses recent algorithmic progress like progressive sieving and dimensions for free. Besides an abstract state machine that allows to easily describe many reduction strategies, it also includes an

Algorithm 1: Lattice sieving algorithm.

```
Input  : A basis $\mathbf{B}$ of a lattice $\mathcal{L}$, list size $N$ and a saturation radius $R$.
Output: A list $L$ of short vectors saturating the ball of radius $R$.
1 Sample a list $L \subset \mathcal{L}$ of size $N$.
2 while $L$ does not saturate the ball of radius $R$ do
3     for every pair $\mathbf{v}, \mathbf{w} \in L$ do
4         if $\mathbf{v} - \mathbf{w} \notin L$ and $\|\mathbf{v} - \mathbf{w}\| < \max_{\mathbf{u} \in L} \|\mathbf{u}\|$ then
5             Replace a longest element of $L$ by $\mathbf{v} - \mathbf{w}$.
6 return $L$
```

open-source implementation that broke several new TU Darmstadt SVP Challenges [SG10] up to dimension 155. This implementation is multi-threaded and low-level optimized and includes many of the implementation tricks from the lattice sieving literature and some more. In this section we recall the state and instructions of G6K.

State. Naturally, the state includes a lattice basis $\mathbf{B} \in \mathbb{Z}^{d \times d}$ and its corresponding Gram-Schmidt basis $\tilde{\mathbf{B}}$. The current state keeps track of a *sieving context* $[l : r]$ and a *lifting context* $[\kappa : r]$. In the remainder of this work the sieving dimension will be denoted by $n := r - l$. There is a database L containing N lattice vectors from the sieving context. To conclude G6K also keeps track of good insertion candidates $\mathbf{i}_\kappa, \ldots, \mathbf{i}_l$ for the corresponding positions in the current lattice basis.

Instructions. In order to move between several contexts there are several instructions like Extend Right, Shrink Left and Extend Left. To avoid invalidating the database the vectors are lifted to the new context as explained in Section 2.1, keeping the lifted vectors somewhat short. The Insertion instruction inserts one of the insertion candidates back into the basis $\mathbf{B}$, replacing another basis vector, and the Gram-Schmidt basis $\tilde{\mathbf{B}}$ is updated correspondingly. By some carefully chosen transformations and by moving to the slightly smaller sieving context $[l + 1 : r]$ we can recycle most of the database after an insertion. We can also Shrink the database by throwing away the longest vectors or Grow it by sampling new (long) vectors. The Sieve instruction reduces vectors in the database until saturation of a ball of a given radius. G6K also allows for well-chosen vectors that are encountered during sieving to be lifted from the sieving context $[l : r]$ to hopefully short vectors in the lifting context $[\kappa : r]$, and storing the best insertion candidates. The Sieve instruction is agnostic about the sieving algorithm used, which allows to relatively easily implement and then compare sieving algorithms with each other, while letting G6K take care of global strategies.

Global Strategies. The implementation of G6K consists of a high level `Python` layer and a low-level `C++` layer. The earlier mentioned instructions can be called and parametrized from the `Python` layer, while the core implementation consists of highly optimized `C++` code. This allows one to quickly experiment with different global strategies. An important global strategy is known as the *pump up*: start in a small context of say $[d - 40 : d]$ and alternate the Extend Left, Grow and Sieve instructions until the context reaches a certain dimension (passed as a parameter). Note that the sieve in each dimension already starts with a database consisting of many relatively short vectors, thus taking significantly less iterations to complete. This technique is also known as *progressive sieving* [Duc18, LM18] and gives a significant practical speed-up. A full *pump* consists of a pump up followed by a *pump down*: repeat the Insertion instruction to improve the basis while making the context smaller again, and optionally combine this with the Sieve instruction to find better insertion candidates. To solve SVP-instances among other things G6K combines such pumps in a *workout*, which is a sequence of longer and longer pumps, until a short enough vector is found in the full context by lifting. Each pump improves the quality of the basis, which as a result lowers the expected length increase from lifting, making consequent pumps faster and simultaneously improving the probability to find a short vector in the full context.

G6K Sieve implementations. The current open-source implementation of G6K contains multiple sieving algorithms that implement the Sieve instruction. There are single-threaded implementations of the Nguyen–Vidick sieve (`nv`) [NV08] and Gauss sieve (`gauss`) [MV10], mostly for testing purposes. Furthermore G6K includes a fully multi-threaded and low-level optimized version of the Becker–Gama–Joux (BGJ) sieve with a single bucketing layer (`bgj1`) [BGJ15]. The filtering techniques from `bgj1` were also extended and used in a triple sieve implementation (`triple`) [BLS16, HK17]. This implementation considers both pairs and triples and its behaviour automatically adjusts based on the database size, allowing for a continuous time-memory trade-off between the (pair) sieve `bgj1` and a full triple sieve with minimal memory. Note that the asymptotically best sieve algorithm, which we will refer to as BDGL, has been implemented before [BDGL16, MLB17], but not inside of G6K.

Data representation. Given that lattice sieving uses an exponential number of vectors, it is of practical importance how much data is stored per vector in the database. G6K stores for each lattice vector $\mathbf{v} = \mathbf{B}\mathbf{x} \in \mathbb{R}^n$ the (16-bit integer) coordinates $\mathbf{x} \in \mathbb{Z}^n$ as well as the (32-bit floating-point) Gram-Schmidt representation $\mathbf{y} = (\langle \mathbf{v}, \mathbf{b}_i^* \rangle / \|\mathbf{b}_i^*\|)_i \in \mathbb{R}^n$ normalized by the Gaussian Heuristic of the current sieving context. The latter representation is used to quickly compute inner products between any two lattice vectors in the database. On top of that other preprocessed information is stored for each vector, like the corresponding lift target $\mathbf{t}$ in $\text{span}(\mathcal{L}_{[\kappa:l]})$, the squared length, a 256-bit SimHash (see [Cha02, FBB$^+$14, Duc18]) and a 64-bit hash as identifier. In order to sort the database on length, without having to move the entries around, there is also a lightweight database that only stores for each vector the length, a SimHash and the corresponding database index. A hash table keeps track of all hash identifiers, which are derived from the $\mathbf{x}$-coordinates, in order to quickly check for duplicates. All of this quickly adds up to a total of $\approx 2^{10}$ bytes per vector in a sieving dimension of $n = 128$.

3 Architecture

3.1 GPU Device Architecture

In this section we give a short summary of the NVIDIA Turing GPU architecture on which our implementations and experiments are based. During the write-up of this paper a new generation named Ampere was launched, doubling many of the performance metrics mentioned here.

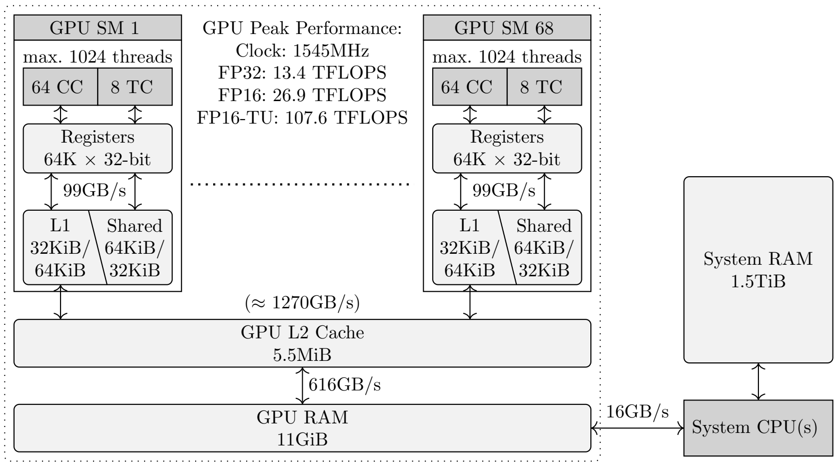


Fig. 1. Device architecture of the NVIDIA RTX 2080 Ti used in this work.

CUDA cores and memory. A NVIDIA GPU can have up to thousands of so-called *CUDA cores* organized into several execution units called Streaming Multiprocessors (*SM*). These SM use their many CUDA cores (e.g. 64) to service many more resident *threads* (e.g. 1024), in order to hide latencies of computation and memory operations. Threads are bundled per 32 in a *warp*, that follow the single-instruction multiple-data paradigm.

The execution of a GPU program, also called a *kernel*, consists out of multiple *blocks*, each consisting of some warps. Each individual block is executed on any available single SM. The GPU RAM, also called *global memory*, can be accessed by all cores. Global memory operations always pass through a GPU-wide L2 cache. In addition, each SM benefits from a individual L1 cache and offers an addressable *shared memory* that can only be used by threads in that block.

In this work we focus on the NVIDIA RTX2080 Ti that we used, whose architecture is depicted in Figure 1. While a high-end CPU with many cores can reach a performance in the order of a few tera floating point operations per second (TFLOPS), the RTX2080 Ti can achieve 13 TFLOPS for 32-bit floating point operations on its regular CUDA cores.

To implement GPU kernel functions for a NVIDIA GPU one can use CUDA [NBGS08, NVF20] which is an extension of the `C/C++` and `FORTRAN` programming languages. A kernel is executed by a specified number of threads grouped into blocks, all with the same code and input parameters. During execution each thread learns that it is thread t inside block b and one needs to use this information to distribute the work. For example when loading data from global memory we can let thread t read the t-th integer at an offset computed from b, because the requested memory inside each block is contiguous such a memory request can be executed very efficiently; such memory request are known as *coalescing* reads or writes and they are extremely important to obtain an efficient kernel.

Tensor cores. Driven by the machine learning domain there have been tremendous efforts in the past few years to speed up low-precision matrix multiplications. This lead to the so-called Tensor cores, that are now standard in high-end NVIDIA GPUs. Tensor cores are optimized for 4×4 matrix multiplication and also allow a trade-off between performance and precision. In particular we are interested in the 16-bit floating point format `fp16` with a 5-bit exponent and a 10-bit mantissa, for which the tensor cores obtain an $8\times$ speed-up over regular 32-bit operations on CUDA cores.

Efficiency. For cryptanalytic purposes it is not only important how many operations are needed to solve a problem instance, but also how cost effective these operations can be executed in hardware. The massively-parallel design of GPUs with many relatively simple cores results in large efficiency gains per FLOP compared to CPU designs with a few rather complex cores; both in initial hardware cost as in power efficiency.

As anecdotal evidence we compare the acquisition cost, energy usage and theoretical peak performance of the CPU and GPU in the new server we used for our experiments: the Intel Xeon Gold 6248 launched in 2019 and the NVIDIA RTX2080 Ti launched in 2018 respectively. The CPU has a price of about €2500 and a TDP of 150 Watt, while the GPU is priced at about €1000 and has a TDP of 260 Watt. For 32-bit floating point operations the peak performance is given by 3.2 TFLOPS[3] and 13.45 TFLOPS for the CPU and GPU respectively, making the GPU a factor 2.4 better per Watt and 10.5 better per Euro spend on acquisition. For general 16-bit floating point operations these number double for the GPU, while the CPU obtains no extra speed-up (one actually has to convert the data back to 32-bit). When considering the specialized Tensor cores with 16-bit precision the GPU has a theoretical peak performance of 107.6 TFLOPS, improving by a factor 19.4 per Watt and a factor 84 per Euro spend on acquisition compared to the CPU.

3.2 Sieve Design

The great efficiency of the GPU is only of use if the state-of-the-art algorithms are compatible with the massively-parallel architecture and the specific low-precision operations of the Tensor cores. To show this we extended the lattice sieving implementation of G6K. We will focus our main discussion on the sieving part, as the other G6K instructions are asymptotically irrelevant and relatively straightforward to accelerate on a GPU (which we also did).

All of our CPU multi-threaded and GPU-powered sieve implementations follow a similar design (cf. Fig. 2) consisting out of three sequential phases: bucketing, reducing and result insertion. We call the execution of this triplet an *iteration* and these iterations are repeated until the desired saturation is achieved. Note that our sieves are not 'queued' sieves such as the Gauss-Sieve of [MV10] and the

[3] With 64 FLOP per core per cycle using two `AVX`-512 FMA units and a maximal clock frequency of 2500MHz when using `AVX`-512 on all 20 cores.

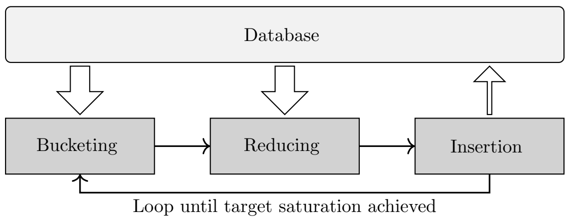


Fig. 2. High level diagram of the implemented Sieving process.

previous record setting `triple_sieve`; this relaxation aligns with the batched nature of GPU processing and allows to implement an asymptotically optimal BDGL-like sieve [BDGL16], without major memory overhead.

Bucketing. During the bucketing phase, the database is subdivided in several buckets $B_1, \ldots, B_m \subset L$, each containing relatively close vectors. We do not necessarily bucket our full database, as some vectors might be too large to be interesting for the reduction phase in the first few iterations. For each bucket we collect the database indices of the included vectors. For the sieves we consider, these buckets can geometrically be interpreted as spherical caps or cones with for each bucket B_k an explicit or implicit *bucket center* $\mathbf{c}_k \in \mathbb{R}^n$ indicating its direction. For each included vector $\mathbf{v} \in B_k$, we also store the inner product $\langle \mathbf{c}_k, \mathbf{v} \rangle$ with the bucket center, which is obtained freely from the bucketing process. Note that a vector may be included in several buckets, something which we tightly control by the *multi-bucket* parameter, whose value we will denote by M. The optimal amount of buckets m and the expected number of vectors in a bucket differs for each of our bucketing implementations. In Section 4, we further exhibit our different bucketing implementations and compare their performance and quality.

Reducing. During the reduction phase, we try to find all close pairs of lattice vectors inside each bucket, i.e., at distance at most some *length bound* ℓ. Using negation, we orient the vectors inside a bucket into the direction of the bucket center based on the earlier computed inner product $\langle \mathbf{c}_k, \mathbf{v}_i \rangle$. In case the bucketing center $\mathbf{c}_k$ is itself a lattice vector (as can be the case for BGJ-like sieves, but not for BDGL), it is also interesting to check if $\mathbf{c}_k - \mathbf{v}_i - \mathbf{v}_j$ is a short lattice vector, leading to a triple reduction [HK17].

For each bucket B_k, we compute all pairwise inner products $\langle \mathbf{v}_i, \mathbf{v}_j \rangle$ for $\mathbf{v}_i, \mathbf{v}_j \in B_k$. Together with the already computed lengths $\|\mathbf{v}_i\|, \|\mathbf{v}_j\|, \|\mathbf{c}_k\|$ and inner products $\langle \mathbf{c}_k, \mathbf{v}_i \rangle, \langle \mathbf{c}_k, \mathbf{b}_j \rangle$ we can then efficiently decide if $\mathbf{v}_i - \mathbf{v}_j$ or $\mathbf{c}_k - \mathbf{v}_i - \mathbf{v}_j$ is short. Note that we compute the length of both the pair and the triple essentially from a single inner product computation. We return the indices of pairs and triplets that result in a vector of length at most the length bound ℓ, together with the length of the new vector. In Section 5 we further discuss the reduction phase, and in Appendix B and exhibit implementation details of our reduction kernel on the GPU using low-precision Tensor cores.

The number of inner products we have to compute per bucket grows quadraticly in the bucket size $|B_k|$, while the number of buckets only decreases linearly in the bucket size. Therefore, one would in principle want many buckets that are rather small and of high quality, improving the probability that a checked pair actually gives a reduction. For a fixed bucketing algorithm more buckets generally increase the cost of the bucketing phase, while decreasing the cost of the reduction phase due to smaller bucket sizes. We try to balance the cost of these phases to obtain optimal performance.

Next to finding short vectors in the sieving context we also want to find pairs that lift to short vectors in the larger lifting context. Unfortunately it is too costly to just lift all pairs as this has a cost of at least $\Theta((l-\kappa)^2)$ per pair. In Section 6 we introduce a filter based on dual vectors that can be computed efficiently for each pair given a bit of pre-computed data per vector. The few pairs that survive this filter are more likely to lift to a short vector and we only lift those pairs.

Result insertion. After the sieving part we have a list of tuples with indices and the corresponding length of the new vector they represent. The hash identifier of the new vector can efficiently be recomputed by linearity of the hash function and we check for duplicates in our current database. For all non-duplicate vectors we then compute their $\mathbf{x}$-representation. After all new entries are created they are inserted back in the database, replacing entries of greater length.

3.3 Data storage and movement

Recall from Section 2.2 that G6K stores quite some data per vector such as the coefficients $\mathbf{x}$ in terms of the basis, a Gram-Schmidt representation $\mathbf{y}$, the lift target $\mathbf{t}$, a SimHash, and more. Theoretically we could remove all data except the $\mathbf{x}$-representation and compute all other information on-the-fly. However, as most of this other information has a cost of $\Theta(n^2)$ to compute from the $\mathbf{x}$-representation this would mean a significant computational overhead, for example increasing the cost of an inner product from $\Theta(n)$ to $\Theta(n^2)$. Also given the limited amount of performance a CPU has compared to a GPU we certainly want to minimize the amount of such overhead for the CPU. By recomputing at some well chosen points on the GPU, our accelerated sieves minimize this overhead, while only storing the $\mathbf{x}$-representation, length and a hash identifier per vector, leading to an approximately 60% reduction in storage compared to the base G6K implementation. As a result we can sieve in significantly larger dimensions with the same amount of system RAM.

While GPUs have an enormous amount of computational power, the memory bandwidth between the database in system RAM and the GPU's RAM is severely limited. These are so imbalanced that one can only reach theoretical peak performance with Tensor cores if every byte that is transferred to the GPU is used in at least 2^{13} computations. A direct result is that reducing in small buckets is (up to some threshold) bandwidth limited. Growing the bucket size in this regime would not increase the wall-clock time of the reduction phase, while at the same time considering more pairs. So larger buckets are preferred, in our hardware for a single active GPU the threshold seems to be around a bucket size of 2^{14}, matching the 2^{13} computations per byte ratio. Because in our hardware each pair of GPUs share their connection to the CPU, halving the bandwidth for each, the threshold grows to around 2^{15} when using all GPUs simultaneously. The added benefit of large buckets is that the conversion from the $\mathbf{x}$-representation to the $\mathbf{y}$-representation, which can be done directly on the GPU, is negligible compared to computing the many pairwise inner products. To further limit the movement of data we only return indices instead of a full vector; if we find a short pair $\mathbf{v}_i - \mathbf{v}_j$ on the GPU we only return i, j and $\|\mathbf{v}_i - \mathbf{v}_j\|^2$. The new $\mathbf{x}$-representation and hash identifier can efficiently (in $O(n)$) be computed on the CPU directly from the database.

4 Bucketing

The difference between different lattice sieve algorithms mainly lies in their bucketing method. These methods differ in their time complexity and their performance in catching close pairs. In this section we exhibit a Tensor-GPU accelerated bucketing implementation `triple_gpu` similar to `bgj1` and `triple` inspired by [BGJ15, HK17], and two optimized implementations of the asymptotically best known bucketing algorithm [BDGL16], one for CPU making use of `AVX2` (`bdgl`) and one for GPU (`bdgl_gpu`). After this we show the practical performance difference between these bucketing methods.

4.1 BGJ-like bucketing (`triple_gpu`)

The bucketing method used in `bgj1` and `triple` is based on spherical caps directed by explicit bucket centers that are also lattice points. To start the bucketing phase we first choose some bucket centers $\mathbf{b}_1, \ldots, \mathbf{b}_m$ from the database; preferably the directions of these vectors are somewhat uniformly distributed over the sphere. Then each vector $\mathbf{v} \in L$ in our database is associated to bucket $B_{k_\mathbf{v}}$ with

$$k_\mathbf{v} = \operatorname*{arg\,max}_{1 \le k' \le m} \left| \left\langle \frac{\mathbf{b}_{k'}}{\|\mathbf{b}_{k'}\|}, \mathbf{v} \right\rangle \right| .$$

We relax this condition somewhat by the multi bucket parameter M, to associate a vector to the best M buckets. In this we differ from the original versions of `bgj1` and `triple` [BGJ15, HK17, ADH+19] in that they use a fixed filtering threshold on the angle $|\langle \mathbf{b}_k / \|\mathbf{b}_k\|, \mathbf{v}/\|\mathbf{v}\| \rangle|$. As a result our buckets do not exactly match spherical caps, but they should still resemble them; in particular such a change does not affect the asymptotic analysis. We chose for this alternation as this fixes the amount of buckets per vector, which reduced some communication overhead in our highly parallel GPU implementations.

In each iteration the new bucket centers are chosen, normalized and stored once on each GPU. Then we stream our whole database $\mathbf{v}_1, \ldots, \mathbf{v}_N$ through the

GPUs and try to return for each vector the indices of the M closest normalized bucket vectors and their corresponding inner products $\langle \mathbf{v}_i, \mathbf{b}_k \rangle$. For efficiency reasons the bucket centers are distributed over 16 threads and each thread stores only the best encountered bucket for each vector. Then we return the buckets from the best $M \leq 16$ threads, which are not necessarily the best M buckets overall. The main computational part of computing the pairwise inner products is similar to the Tensor-GPU implementation for reducing, and we refer to Appendix B for further implementation details.

The cost of bucketing is $O(N \cdot n)$ per bucket. Assuming that the buckets are of similar size $|B_k| \approx M \cdot N/m$ the cost to reduce is $O(\frac{M \cdot N}{m} \cdot n)$ per bucket. To balance these costs for an optimal runtime one should choose $m \sim M \cdot \sqrt{N}$ buckets per iteration. For the regular 2-sieve strategy with an asymptotic memory usage of $N = (4/3)^{n/2+o(n)} = 2^{0.208n+o(n)}$ this leads to a total complexity of $2^{0.349n+o(n)}$ using as little as $2^{0.037n+o(n)}$ iterations. Note that in low dimensions we might prefer a lower number of buckets to achieve the minimum required bucket size to reach peak efficiency during the reduction phase.

4.2 BDGL-like bucketing (`bdgl` and `bdgl_gpu`)

The asymptotically optimal bucketing method from [BDGL16] is similar to `bgj1` as in that it is based on spherical caps. The difference is that in contrast to `bgj1` the bucket centers are not arbitrary but structured, allowing to find the best bucket without having to compute the inner product with each individual bucket center.

Following [BDGL16], such a bucketing strategy would look as follows. First we split the dimension n into k smaller blocks (say, $k = 2, 3$ or 4 in practice) of similar dimensions $n_1, \ldots, n_k$ that sum up to n. In order to randomize this splitting over different iterations one first applies a random orthonormal transformation $\mathbf{Q}$ to each input vector. Then the set C of bucket centers is constructed as a direct product of random local bucket centers, i.e., $C = C_1 \times \pm C_2 \cdots \times \pm C_k$ with $C_b \subset \mathbb{R}^{n_b}$. Note that for a vector $\mathbf{v}$ we only have to pick the closest local bucket centers to find the closest global bucket center, implicitly considering $m = 2^{k-1} \prod_b |C_b|$ bucket centers at the cost of only $\sum_b |C_b| \approx O(m^{1/k})$ inner products. By sorting the local inner products we can also efficiently find all bucket centers within a certain angle or say the closest M bucket centers. With similar reasons as for `triple_gpu` we always return the closest M bucket centers for each vector instead of a fixed threshold based on the angle. While for a fixed number of buckets m we can expect some performance loss compared to `bgj1` as the bucket centers are not perfectly random, this does not influence the asymptotics.[4]

To optimize the parameters we again balance the cost of bucketing and reducing. Note that for $k = 1$ we essentially obtain `bgj1` with buckets of size $O(N^{1/2})$ and a time complexity of $2^{0.349n+o(n)}$. For $k = 2$ or $k = 3$ the buckets become smaller of size $O(N^{1/3})$ and $O(N^{1/4})$ respectively and of higher quality,

[4] The analysis of [BDGL16, Theorem 5.1] shows this is up to a sub-exponential loss.

leading to a time complexity of $2^{0.3294n+o(n)}$ and $2^{0.3198n+o(n)}$ respectively. By letting k slowly grow, e.g., $k = O(\log(n))$ there will only be a sub-exponential $2^{o(n)}$ number of vectors in each bucket, leading to the best known time complexity of $2^{0.292n+o(n)}$. Note however that a lot of sub-exponential factors might be hidden inside this $o(n)$, and thus for practical dimensions a rather small value of k might give best results.

We will take several liberties with the above strategy to address practical efficiency consideration and fine-tune the algorithm. For example, for a pure CPU implementation we may prefer to make the average bucket size somewhat larger than the $\approx N^{1/(k+1)}$ vectors that the theory prescribes; this will improve cache re-use when searching for reducible pairs inside buckets. In our GPU implementation, we make this average bucket size even larger, to prevent memory bottlenecks in the reduction phase.

Furthermore, we optimize the construction of the local bucket centers $\mathbf{c} \in C_i$ to allow for a fast computation of the local inner products $\langle \mathbf{c}, \mathbf{v} \rangle$. While [BDGL16] choose the local bucket centers C_i uniformly at random, we apply some extra structure to compute each inner product with a vector $\mathbf{v}$ in time $O(\log(n_i))$ instead of $O(n_i)$. The main idea is to use the (Fast) Hadamard Transform $\mathcal{H}$ on say $32 \le n_i$ coefficients of $\mathbf{v}$. Note that this computes the inner product between $\mathbf{v}$ and 32 orthogonal ternary vectors, which implicitly form the bucket centers, using only $32 \log_2(32)$ additions or subtractions. To obtain more than 32 different buckets we permute and negate coefficients of $\mathbf{v}$ in a pseudo-random way before applying $\mathcal{H}$ again. This strategy can be heavily optimized both for CPU using the vectorized `AVX2` instruction set (`bdgl`) and for GPU by using special warp-wide instructions (`bdgl_gpu`). In particular this allows a CPU core to compute an inner product every 1.3 to 1.6 cycles for $17 \le n_i \le 128$. For further implementation details we refer to Appendix A.

Since the writing of this report, our CPU implementation of `bdgl` has been integrated in G6K, with further improvements.[5] As it may be of independant interest, the `AVX2` bucketer is also provided as a standalone program.[6]

4.3 Quality Comparison

In this section we compare the practical bucketing quality of the BGJ- and BDGL-like bucketing methods we implemented. More specifically, we consider `triple_gpu`, `1-bdgl_gpu` and `2-bdgl_gpu` where the latter two are instances of `bdgl_gpu` with $k = 1$ and $k = 2$ blocks respectively. Their quality is compared to the idealized theoretical performance of `bgj1` with uniformly distributed bucket centers.[7] For `triple_gpu`, we follow the Gaussian Heuristic and sample bucket centers whose directions are uniformly distributed. As a result the quality difference between `triple_gpu` and the idealized version highlights the quality loss

[5] https://github.com/fplll/g6k/pull/61

[6] https://github.com/lducas/AVX2-BDGL-bucketer

[7] Volumes of caps and wedges for predicting the idealized behavior where extracted from [AGPS19], and more specifically https://github.com/jschanck/eprint-2019-1161/blob/main/probabilities.py.

resulting from our implementation decisions. Recall that compared to `bgj1` the main difference is that for every vector we return the M closest bucket centers instead of using a fixed threshold for each bucket. Also these are not exactly the M closest bucket centers, as we first distribute the buckets over 16 threads and only store a single close bucket per thread. For our `bdgl_gpu` implementation the buckets are distributed over 32 threads and we add to this that the bucket centers are not random but somewhat structured by the Hadamard construction.

To compare the geometric quality of bucketing implementations, we measure how uniform vectors are distributed over the buckets and how many close pairs end up in at least one common bucket. The first measure is important as the reduction cost does not depend on the square of the average bucket size $\left(\frac{1}{m}\sum_{k=1}^{m}|B_k|\right)^2$, which is fixed, but on the average of the squared bucket size $\frac{1}{m}\sum_{k=1}^{m}|B_k|^2$, which is only minimal if the vectors are equally distributed over the buckets. For all our experiments we observed at most an overhead of 0.2% compared to perfectly equal bucket sizes and thus we will further ignore this part of the quality assessment. To measure the second part efficiently we sample 2^{20} close unit pairs $(\mathbf{x}, \mathbf{y}) \in \mathcal{S}^n \times \mathcal{S}^n$ uniformly at random such that $\langle \mathbf{x}, \mathbf{y} \rangle = \pm\frac{1}{2}$. Then we count the number of pairs that have at least 1 bucket in common, possibly over multiple iterations. We run these experiments with parameters that are representative for practical runs. In particular we consider (sieving) dimensions up to $n = 144$ and a database size of $N = 3.2 \cdot 2^{0.2075n}$ to compute the number of buckets given the desired average bucket size and the multi-bucket parameter M. Note that we specifically consider the geometric quality of these bucketing implementations for equivalent parameters and not the cost of the bucketing itself.

To compare the bucketing quality between the different methods and the idealized case we first consider the experimental results in graphs **a.** and **b.** of Figure 3. Note that the bucketing methods `triple_gpu` and `1-bdgl_gpu` obtain extremely similar results overall, showing that the structured Hadamard construction is competitive with fully random bucket centers. We see a slight degradation of 5% to 20% for `triple_gpu` with respect to the idealized case as a result of not using a fixed threshold. We do however see this gap decreasing when M grows to 4 or 8, indicating that these two methods of assigning the buckets become more similar for a larger multi-bucket parameter. At $M = 16$ we see a sudden degradation for `triple_gpu` which exactly coincides with the fact that the buckets are distributed over 16 threads and we only store the closest bucket per thread. The quality loss of `2-bdgl_gpu` seems to be between 15% and 36% in the relevant dimensions, which is quite significant but reasonable given a loss potentially as large as sub-exponential [BDGL16, Theorem 5.1].

Now we focus our attention on graph **c.** of Figure 3 to consider the influence of the average bucket size on the quality. We observe that increasing the average bucket size reduces the bucketing quality; many small buckets have a better quality than a few large ones. This is unsurprising as the asymptotically optimal BDGL sieve aims for high quality buckets of small size. Although our `k-bdgl_gpu` bucketing method has no problem with efficiently generating many

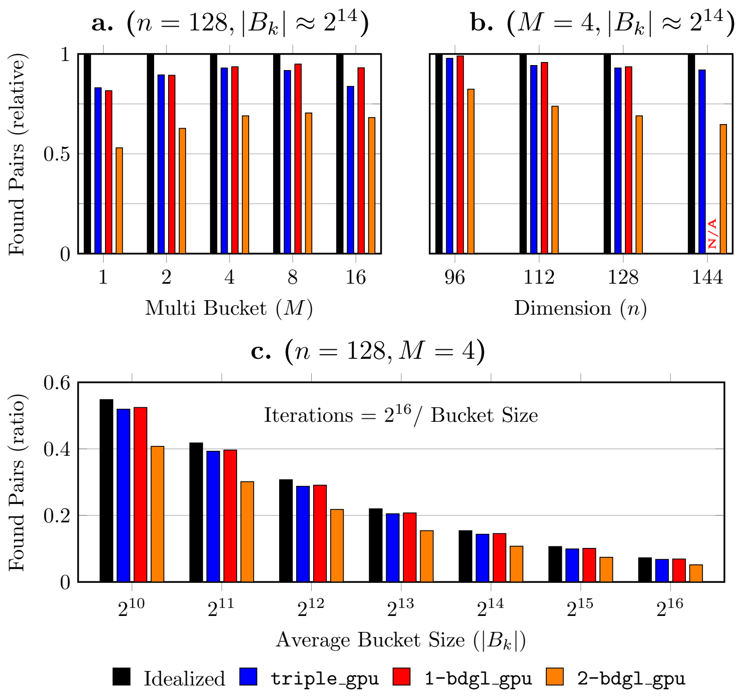


Fig. 3. Bucketing Quality Comparison. We sampled 2^{20} pairs $\mathbf{v}, \mathbf{w}$ of unit vectors such that $|\langle \mathbf{v}, \mathbf{w} \rangle| = 0.5$ and we measured how many fell into at least 1 common bucket. The number of buckets is computed based on the desired average bucket size $|B_k|$, the multi-bucket parameter M, and a representative database size of $N = 3.2 \cdot 2^{0.2075n}$. The found pairs in **a.** and **b.** are normalized w.r.t. idealized theoretical performance of `bgj1` (perfectly random spherical caps). For **c.** the number of applied iterations is varied such that the total reduction cost is fixed.

small buckets, the reduction phase cannot efficiently process small buckets due to memory bottlenecks. This is the main trade-off of (our implementation of) GPU acceleration, requiring a bucket size of 2^{15} versus e.g. 2^{10} leads to a potential loss factor of 7 to 8 as shown by this graph. For `triple_gpu` this gives no major problems as for the relevant dimensions $n \geq 130$ the optimal bucket sizes are large enough. However `2-bdgl_gpu` should become faster than `bgj1` exactly by considering many smaller buckets of size $N^{1/3}$ instead of $N^{1/2}$, and a minimum bucket size of 2^{15} shifts the practical cross-over point above dimension 130, and potentially much higher.

5 Reducing with Tensor cores

Together with bucketing, the most computationally intensive part of sieving algorithms is that of finding reducing pairs or triples inside a bucket. We consider a bucket of s vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s \in \mathbb{R}^n$ with bucket center $\mathbf{c}$. Only the $\mathbf{x}$-representations are send to the GPU and there they are converted to the 16-bit Gram-Schmidt representations $\mathbf{y}_1, \ldots, \mathbf{y}_s$ and $\mathbf{y_c}$ that are necessary to quickly compute inner products. Together with the pre-computed squared lengths $\|\mathbf{y}_1\|^2$, $\ldots$, $\|\mathbf{y}_s\|^2$ and inner products $\langle \mathbf{y_c}, \mathbf{y}_1\rangle, \ldots, \langle \mathbf{y_c}, \mathbf{y}_s\rangle$, the goal is to find all pairs $\mathbf{y}_i - \mathbf{y}_j$ or triples $\mathbf{y_c} - \mathbf{y}_i - \mathbf{y}_j$ of length at most some bound ℓ. A simple derivation shows that this is the case if and only if

$$\text{for } \textbf{pairs:} \qquad \langle \mathbf{y}_i, \mathbf{y}_j \rangle \geq \frac{\|\mathbf{y}_i\|^2 + \|\mathbf{y}_j\|^2 - \ell^2}{2}, \text{ or}$$

$$\text{for } \textbf{triples:} \qquad \langle \mathbf{y}_i, \mathbf{y}_j \rangle \leq -\frac{\|\mathbf{y_c}\|^2 + \|\mathbf{y}_i\|^2 + \|\mathbf{y}_j\|^2 - \ell^2 - 2\langle \mathbf{y_c}, \mathbf{y}_i \rangle - 2\langle \mathbf{c}, \mathbf{y}_j \rangle}{2}.$$

And thus we need to compute all pairwise inner products $\langle \mathbf{y}_i, \mathbf{y}_j \rangle$. If we consider the matrix $\mathbf{Y} := [\mathbf{y}_1, \ldots, \mathbf{y}_s] \in \mathbb{R}^{n \times s}$ then computing all pairwise inner products is essentially the same as computing one half of the matrix product $\mathbf{Y}^t\mathbf{Y}$.

Many decades have been spend optimizing (parallel) matrix multiplication for CPUs, and this has also been a prime optimization target for GPUs. As a result we now have heavily parallelized and low-level optimized BLAS (Basic Linear Algebra Subprograms) libraries for matrix multiplication (among other things). For NVIDIA GPUs close to optimal performance can often be obtained using the proprietary cuBLAS library, or the open-source, but slightly less optimal CUTLASS library. Nevertheless the BLAS functionality is not perfectly adapted to our goal. Computing and storing the matrix $\mathbf{Y}^t\mathbf{Y}$ would require multiple gigabytes of space. Streaming the result $\mathbf{Y}^t\mathbf{Y}$ to global memory takes more time than the computation itself. Indeed computing $\mathbf{Y}^t\mathbf{Y}$ using cuBLAS does not exceed 47 TFLOPS for $n \leq 160$, and this will be even lower when also filtering the results.

For high performance, in our implementation we combined the matrix multiplication with result filtering. We made sure to only return the few indices of pairs that give an actual reduction to global memory; filtering the results locally while the computed inner products are still in registers. Nevertheless the data-movement design, e.g. how we efficiently stream the vectors $\mathbf{y}_i$ into the registers of the SMs, is heavily inspired by CUTLASS and cuBLAS. To maximize memory read throughput, we had to go around the dedicated CUDA tensor API and reverse engineer the internal representation to obtain double the read throughput. Further implementation details are discussed in Appendix B.

Efficiency. To measure the efficiency of our Tensor-accelerated GPU kernel we did two experiments: the first experiment runs only the kernel with all (converted) data already present in global memory on the GPU, while the second experiment emulates the practical efficiency by including all overhead. This overhead consists of obtaining the vectors from the database, sending them to the GPU, converting them to the appropriate representation, running the reduction kernel, recomputing the length of the resulting close pairs, and retrieving the results from the GPU. Each experiment processed a total of 2^{28} vectors of dimension 160 in a pipelined manner on a single NVIDIA RTX 2080 Ti GPU and

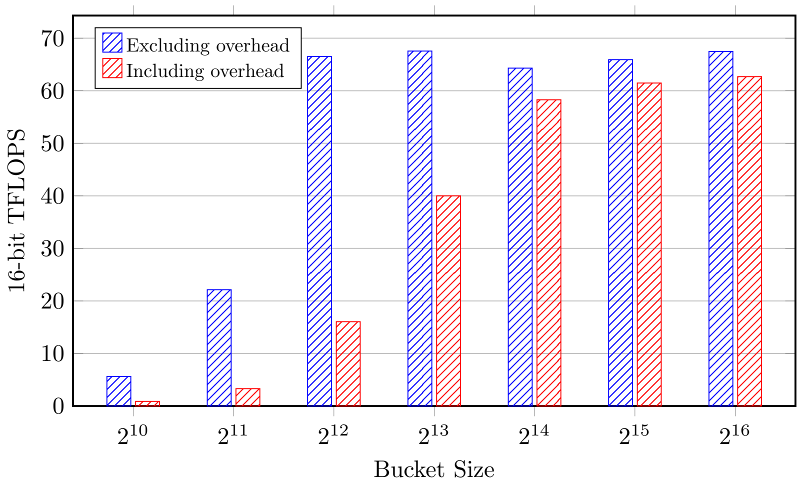


Fig. 4. Efficiency of the reduction GPU kernel for different bucket sizes on a RTX 2080 Ti, only counting the $2n$ FLOPS per inner product. The overhead includes obtaining the vectors from the database, sending them to the GPU, conversions, recomputing length at higher precision, and retrieving the results from the GPU in a pipelined manner.

with a representative number of 10 CPU threads. We only counted the $2n$ 16-bit floating point operations per inner product and not any of the operations necessary to transfer data or to filter and process the results. The theoretical limit for this GPU when only using Tensor cores and continuously running at boost clock speeds is 107 TFLOPS, something which is unrealistic in practice.

The results of these experiments are displayed in Figure 4. We see that the kernel itself reaches around 65 TFLOPS starting at a bucket size of at least 2^{12}. When including the overhead we see that the performance is significantly limited below a bucket size of 2^{13} which can fully be explained by CPU-GPU memory-bottlenecks. For bucket sizes of at least 2^{14} we see that the overhead becomes reasonably small. We observed that this threshold moves to 2^{15} when using multiple GPUs, because in our hardware the CPU-GPU bandwidth is shared per pair of GPUs.

Precision. The main drawback of the high performance of the tensor cores is that the operations are at low precision. Because the runtime of sieving algorithms is dominated by computing pairwise inner products to find reductions or for bucketing (in case of `triple_gpu`) we focus our attention on this part. Other operations like converting between representations are computationally insignificant and can easily be executed by regular CUDA cores at higher precisions. As the GPU is used as a filter to find (extremely) likely candidates for reduction, we can tolerate some relative error, say up to 2^{-7} in the computed inner product, at the loss of more false positives or missed candidates. Furthermore it

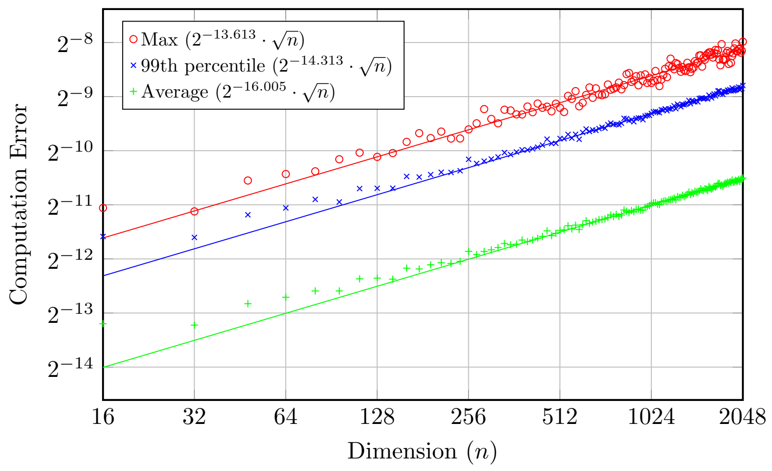


Fig. 5. Computation error $|S - \hat{S}|$ observed in dimension n over 16384 sampled pairs of unit vectors $\mathbf{y}, \mathbf{y}'$ that satisfy $S := \langle \mathbf{y}, \mathbf{y}' \rangle \approx 0.5$.

is acceptable for our purposes if say 1% of the close vectors are missed because of even larger errors. In Appendix C we show under a reasonable randomized error model that problems due to precision are insignificant up to dimensions as large as $n = 2048$. This is also confirmed by practical experiments as shown in Figure 5.

6 Filtering Lifts with Dual Hash

Let us recall the principle of the 'dimensions for free' trick [Duc18]; by lifting many short vectors in the sieving context $[l : r]$ we can recover a short(est) vector in some larger context $[l - k : r]$ for $k > 0$. The sieving implementation G6K [ADH+19] puts extra emphasis on this by lifting any short pair it encounters while reducing a bucket, even when this vector is not short enough to be added to the database. Note that G6K first filters on the length in the sieving context because lifting has a significant cost of $O(n \cdot k + k^2)$ per pair. The $O(n \cdot k)$ part to compute the corresponding target $\mathbf{t}_i - \mathbf{t}_j \in \mathbb{R}^k$ in the context $[l - k : l]$ can be amortized to $O(k)$ over all pairs by pre-computing $\mathbf{t}_1, \ldots, \mathbf{t}_s$, leaving a cost of $O(k^2)$ for the Babai nearest plane algorithm.

We went for a stronger filter with an emphasis on the extra length added by the lifting. Most short vectors will lift to rather large vectors, as by the Gaussian Heuristic we can expect an extra length of $\mathrm{gh}(l - k : l) \gg \mathrm{gh}(l - k : r)$. For the few lifts that we are actually interested in we expect an extra length of only $\delta \cdot \mathrm{gh}(l - k : l)$, for some $0 < \delta < 1$ (say $\delta \in [0.1, 0.5]$ in practice). This means that we need to catch those pairs $\mathbf{t}_i - \mathbf{t}_j$ that lie exceptionally close to the lattice $[l - k : l]$, also known as BDD instances.

More abstractly we need a filter that quickly checks if pairs are (exceptionally) close over the *torus* $\mathbb{R}^k/\mathcal{L}$. Constructing such a filter directly for this rather complex torus and our practical parameters seems to require at least quadratic time like Babai's nearest plane algorithm. Instead we introduce a *dual hash* to move the problem to the much simpler but possibly higher dimensional torus $\mathbb{R}^h/\mathbb{Z}^h$. More specifically, we will use inner products with short dual vectors to build a BDD distinguisher in the spirit of the so-called dual attack on LWE given in [MR09] (the general idea can be traced back at least to [AR05]). This is however done in a different regime, where the shortest dual vectors are very easy to find (given the small dimension of the considered lattice); we will also carefully select a subset of those dual vectors to optimize the fidelity of our filter. Recall that the dual of a lattice $\mathcal{L}$ is defined as $\mathcal{L}^* := \{\mathbf{w} \in \operatorname{span}(\mathcal{L}) : \langle \mathbf{w}, \mathbf{v} \rangle \in \mathbb{Z} \text{ for all } \mathbf{v} \in \mathcal{L}\}$.

Definition 1 (Dual hash). *For a lattice $\mathcal{L} \subset \mathbb{R}^k$, $h \geq k$ and a full (row-rank) matrix $\mathbf{D} \in \mathbb{R}^{h \times k}$ with rows in the dual $\mathcal{L}^*$, we define the dual hash*

$$\begin{aligned} \mathcal{H}_{\mathbf{D}} : \mathbb{R}^k/\mathcal{L} &\to \mathbb{R}^h/\mathbb{Z}^h, \\ \mathbf{t} &\mapsto \mathbf{D}\mathbf{t}. \end{aligned}$$

The dual hash relates distances in $\mathbb{R}^k/\mathcal{L}$ to those in $\mathbb{R}^h/\mathbb{Z}^h$.

Lemma 2. *Let $\mathcal{L} \subset \mathbb{R}^k$ be a lattice with some dual hash $\mathcal{H}_{\mathbf{D}}$. Then for any $\mathbf{t} \in \mathbb{R}^k$ we have*

$$\operatorname{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}), \mathbb{Z}^h) \leq \sigma_1(\mathbf{D}) \cdot \operatorname{dist}(\mathbf{t}, \mathcal{L}),$$

where $\sigma_1(\mathbf{D})$ denotes the largest singular value of $\mathbf{D}$.

Proof. Let $\mathbf{x} \in \mathcal{L}$ such that $\|\mathbf{x} - \mathbf{t}\| = \operatorname{dist}(\mathbf{t}, \mathcal{L})$. By definition we have $\mathbf{D}\mathbf{x} \in \mathbb{Z}^h$ and thus $\mathcal{H}_{\mathbf{D}}(\mathbf{t} - \mathbf{x}) \equiv \mathcal{H}_{\mathbf{D}}(\mathbf{t})$. We conclude by noting that $\operatorname{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t} - \mathbf{x}), \mathbb{Z}^h) \leq \|\mathbf{D}(\mathbf{t} - \mathbf{x})\| \leq \sigma_1(\mathbf{D})\, \|\mathbf{t} - \mathbf{x}\|$.

So if a target $\mathbf{t}$ lies very close to the lattice then $\mathcal{H}_{\mathbf{D}}(\mathbf{t})$ lies very close to $\mathbb{Z}^h$. We can use this to define a filter that passes through BDD instances.

Definition 3 (Filter). *Let $\mathcal{L} \subset \mathbb{R}^k$ be a lattice with some dual hash $\mathcal{H}_{\mathbf{D}}$. For a hash bound H we define the filter function*

$$\mathcal{F}_{D,H} : \mathbf{t} \mapsto \begin{cases} 1, & \textit{if } \operatorname{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}), \mathbb{Z}^h) \leq H, \\ 0, & \textit{else.} \end{cases}$$

Note that computing the filter has a cost of $O(h \cdot k)$ for computing $\mathbf{D}\mathbf{t}$ for $\mathbf{D} \in \mathbb{R}^{h \times k}$ followed by a cost of $O(h)$ for computing $\operatorname{dist}(\mathbf{D}\mathbf{t}, \mathbb{Z}^h)$ using simple coordinate-wise rounding. Given that $h \geq k$, computing the filter is certainly not cheaper than ordinary lifting, which is the opposite of our goal. However this changes when applying the filter to all pairs $\mathbf{t}_i - \mathbf{t}_j$ with $1 \leq i < j \leq h$. We can pre-compute $\mathbf{D}\mathbf{t}_1, \ldots, \mathbf{D}\mathbf{t}_s$ once, which gives a negligible overhead for large buckets, and then compute $\mathbf{D}(\mathbf{t}_i - \mathbf{t}_j)$ by linearity, lowering the total cost to $O(h)$ per pair.

6.1 Dual Hash Analysis

We further analyse the dual hash filter and try to understand the correlation between the distance $\operatorname{dist}(\mathbf{t}, \mathcal{L})$ and the dual hash $\mathcal{H}_{\mathbf{D}}(\mathbf{t})$. In fact we consider two regimes, the preserved and unpreserved regime. Consider a target $\mathbf{t} \in \mathbb{R}^k$ and let $\mathbf{x}$ be a closest vector in $\mathcal{L}$ to $\mathbf{t}$. We will say that we are in the preserved regime whenever $\mathbf{D}(\mathbf{t} - \mathbf{x}) \in [-\frac{1}{2}, \frac{1}{2}]^h$ (i.e., $\mathbf{Dx}$ remains a closest vector of $\mathbf{Dt}$ among $\mathbb{Z}^h$), in which case it holds that $\|\mathbf{D}(\mathbf{t}-\mathbf{x})\|_2 = \operatorname{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}), \mathbb{Z}^h)$. In the general case, we only have the inequality $\|\mathbf{D}(\mathbf{t}-\mathbf{x})\|_2 \geq \operatorname{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}), \mathbb{Z}^h)$. For the relevant parameters, the BDD instances we are interested in will fall almost surely in the preserved regime, while most of the instances we wish to discard quickly will fall in the unpreserved regime.

Preserved Regime. We have that $\|\mathbf{D}(\mathbf{t}-\mathbf{x})\|_2 = \operatorname{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}), \mathbb{Z}^h)$, and therefore Lemma 2 can be complemented with a lower bound as follows:

$$\sigma_k(\mathbf{D}) \cdot \operatorname{dist}(\mathbf{t}, \mathcal{L}) \leq \operatorname{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}), \mathbb{Z}^h) \leq \sigma_1(\mathbf{D}) \cdot \operatorname{dist}(\mathbf{t}, \mathcal{L}).$$

Setting a conservative hash bound based on the above upper bound leads to false positives of distance at most $\sigma_1(\mathbf{D})/\sigma_k(\mathbf{D})$ further away than the targeted BDD distance. This is a worst-case view, however, and we are more interested in the average behavior. We will assume without loss of generality that $\mathbf{x} = 0$, such that $\operatorname{dist}(\mathbf{t}, \mathcal{L}) = \|\mathbf{t}\|$. To analyse what properties play a role in this correlation we assume that $\mathbf{t}$ is spherically distributed for some fixed length $\|\mathbf{t}\|$. Suppose that $\mathbf{D}^{\mathbf{t}}\mathbf{D}$ has eigenvalues $\sigma_1^2, \ldots, \sigma_k^2$ with corresponding normalized (orthogonal) eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$. We can equivalently assume that $\mathbf{t} = \sum_{i=1}^k t_i \mathbf{v}_i$ with $(t_1, \ldots, t_k)/\|\mathbf{t}\|$ uniformly distributed over the sphere. Computing the expectation and variation we see

$$\mathbb{E}[\|\mathbf{Dt}\|^2] = \mathbb{E}\left[\sum_{i=1}^k t_i^2 \cdot \sigma_i^2\right] = \sum_{i=1}^k \sigma_i^2 \cdot \mathbb{E}[t_i^2] = \|t\|^2 \cdot \frac{1}{k}\sum_{i=1}^k \sigma_i^2$$

$$\operatorname{Var}\left[\|\mathbf{Dt}\|^2\right] = \frac{\|t\|^4}{(k/2+1)}\left(\frac{1}{k} \cdot \sum_{i=1}^k \sigma_i^4 - \left(\frac{1}{k}\sum_{i=1}^k \sigma_i^2\right)^2\right).$$

So instead of the worst case bounds from Lemma 2, $\operatorname{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}), \mathbb{Z}^h)$ is more or less close to $\sqrt{\frac{1}{k}\sum_{i=1}^k \sigma_i^2} \cdot \|\mathbf{t}\|$.

Unpreserved Regime. In this regime $\operatorname{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}), \mathbb{Z}^h)$ is not really a useful metric, as there will seemingly be no relation with $\|\mathbf{D}(\mathbf{t}-\mathbf{x})\|_2$. Note that we can expect this regime to mostly contain targets that lie rather far from the lattice, i.e., these are targets we want to not pass our filter. Therefore it is interesting to analyse how many (false) positives we can expect from this regime.

Inspired by practical observations, we analyse these positives from the heuristic assumption in this regime that every $\mathbf{Dt}$ is just uniformly distributed over

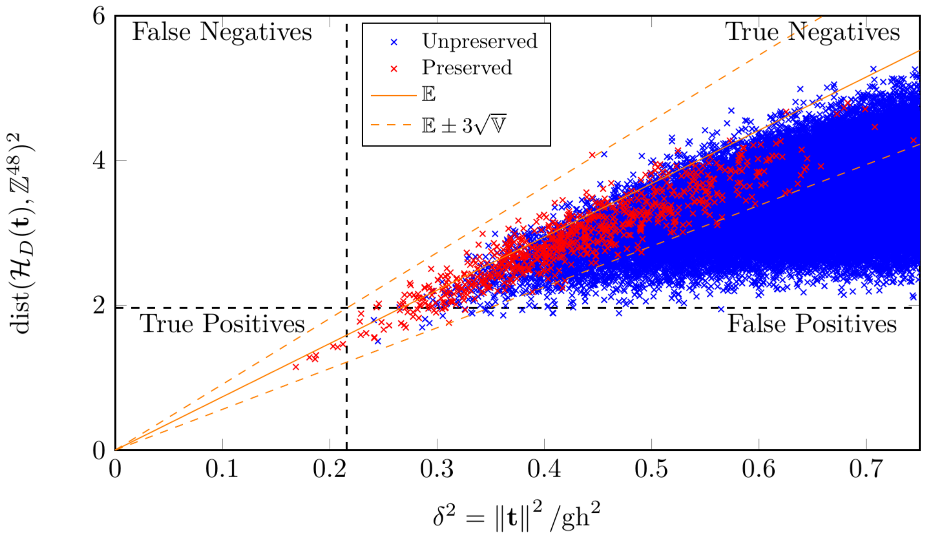


Fig. 6. Dual hash filter correlation on the context $[14 : 30]$ for a reduced 160-dimensional lattice using 48 dual vectors. The BDD-bound was computed with a representative squared length bound of 1.44 and the 2^{20} targets are uniformly sampled over the Voronoi cell around 0.

$[-\frac{1}{2}, \frac{1}{2}]^h$ modulo $\mathbb{Z}^h$. Then we can ask the question how probable it is that $\|\mathbf{Dt}\|_2 = \mathrm{dist}(\mathbf{Dt}, \mathbb{Z}^h) \leq H$; i.e., that the target passes the filter. This is equivalent to the volume of the intersection of an h-dimensional ball with radius H and the hypercube $[-\frac{1}{2}, \frac{1}{2}]^h$. We can bound this by just the volume of the ball, which is quite tight if H is not too large. Therefore we would expect in this regime a false positive rate bounded by $H^h \cdot \frac{\pi^{h/2}}{\Gamma(h/2+1)}$. Note that this only depends on the filter threshold H and the number of dual vectors h and not on the specific matrix $\mathbf{D}$.

Choosing a dual hash. We will shortly discuss how to pick the dual hash matrix $\mathbf{D} \in \mathbb{R}^{h \times k}$. The goal is to obtain a filter with a good correlation, i.e., a good trade-off between the positive-rate and the number of false negatives. As the computational cost mostly depends on the number of dual vectors h we will try to optimize $\mathbf{D}$ for a fixed h.

In the preserved regime we see that the variation is minimized if all singular values are equal, so we want $\mathbf{D}$ to be well conditioned in the sense that all singular values are somewhat the same. For the unpreserved regime we want the filter bound H to be small, which means we want $\sum_{i=1}^{k} \sigma_i(D)^2$ to be small (together with the variance); this can be achieved by working with short dual vectors.

To summarize we want to find a set of short dual vectors to form the dual hash such that $\mathbf{D}$ is well conditioned. One initial method is to just pick the h

shortest dual vectors (modulo sign). This definitely satisfies the needs of the unpreserved regime, but the conditioning of the resulting matrix is often not that great. Given a list of short dual vectors we can greedily try to improve the conditioning of $\mathbf{D}$ by replacing some of the (row) vectors from the list. A good continuous metric to measure if all singular values are somewhat the same is $\mathrm{Tr}(\mathbf{D}^t\mathbf{D})/\det(\mathbf{D}^t\mathbf{D})^{1/k}$. From experiments we can conclude that this greedy method to improve the filter works really well. For example with the parameters as in Figure 6, picking the 48 shortest dual vectors leads to a positive rate of $1.3 \cdot 10^{-4}$ for a false negative rate of 1%; using the greedy construction improves the positive rate down to $1.4 \cdot 10^{-5}$ for the same false negative rate. The additional overhead of the greedy method is negligible and easily won back from allowing a lower number of dual vectors h.

6.2 Implementation

Given a list of pre-computed $\mathbf{Dt}_1, \ldots, \mathbf{Dt}_m$ we want to use the GPU to efficiently compute $\mathrm{dist}(\mathbf{D}(\mathbf{t}_i - \mathbf{t}_j), \mathbb{Z}^h)$ for all $i < j$. As usual the actual implementation requires some trade-offs to significantly improve performance. Given that the dimension of the dual hash seems to have more impact than the precision of the values we choose for an 8-bit integer representation for the dual hash coordinates in $[-1/2, 1/2)$ by dividing it in 256 equally sized intervals. The added benefit of this representation is that the $\bmod \mathbb{Z}$ operations are implicitly handled by integer overflow. Both CUDA and Tensor cores have special instructions and very good performance for 8-bit arithmetic, even when using 32 bits to accumulate inner products. We refer to Appendix D for more implementation details on computing all pairwise dual hash filters using CUDA cores; we also discuss how one could adapt it for Tensor cores.

Choosing the parameters. To use the dual hash in practice as a filter we need to decide on what context to use it and what the threshold should be. Applying the dual hash to the full lift context $[\kappa : l]$ would fail to return short vectors for positions $l' > \kappa$, which are also needed to improve the quality of the basis. Therefore we apply the dual hash to a subcontext $[f : l]$ (the lift-filter context) of the lift context $[\kappa : l]$. If a vector is short in the context $[l' : r]$ for some $l' < f$ then we can also expect it to be short in the filter context, and therefore to be catched by our filter.

We also need to decide on a distance threshold. Let $\mathbf{v}$ be a lattice vector in the sieving context $[l : r]$ of length R. We can assume that $R \geq \ell$ as otherwise the vector would already be inserted (and always lifted) in the sieving database. Suppose that $\mathbf{v}$ lifts to a short vector with length at most $\ell_{l'}$ in some context $[l' : r]$ for $\kappa \leq l' \leq f$. This corresponds to a target $\mathbf{t}$ at distance at most

$$\mathrm{dist}(\mathbf{t}, \mathcal{L}_{[l':l]}))^2 \leq \ell_{l'}^2 - \ell^2.$$

in the context $[l' : l]$. Although we cannot know what the length of $\mathbf{t}$ would be in the filter context we can expect this to be close to $\sqrt{\frac{l-f}{l-l'}}\,\|\mathbf{t}\|$ by the Gaussian

Heuristic. Therefore setting the filter length bound to

$$F_{l'} := \sqrt{\frac{l-f}{l-l'}\left(\ell_{l'}^2 - \ell^2\right)}$$

allows a significant part of the short lifts in the context $[l' : r]$ through the filter. Note that most of the pairs we lift are much larger on the sieving part, and thus have to be even shorter in the filter context; definitely passing the above filter length bound. We conclude by setting the filter to aim for a length of at most $F := \max_{\kappa \leq l' \leq f}\{F_{l'}\}$.

Given the filter length bound we could immediately apply Lemma 2 to obtain a threshold for the dual hash that guarantees that our filter has no false negatives. However as usual there is a trade-off between the number of false negatives and the positive rate of the filter. For our purposes we set the bound at the expectation plus 3 standard deviations in the preserved regime to prevent most false negatives. For a more precise bound under a fixed false negative ratio one could fall back to Monte-Carlo sampling methods as we do not know of a closed form formula for the distribution. Figure 6 shows the effectiveness of the dual hash filter based on realistic parameters as encountered during a 130-dimensional pump on a 160-dimensional lattice. The pre-processing of the basis consisted of a workout with pumps up to dimension 128.

7 Sieving in practice

7.1 Comparison

We compare several of our sieve implementations. Although our BDGL-like implementations `bdgl` and `bdgl_gpu` will eventually be faster than the BGJ-like implementations `triple` by G6K and `triple_gpu` by us, the cross-over point could be outside of practical dimensions. For the comparison we run a pump up to dimension 120 and 140 for CPU and GPU respectively in a lattice of dimension 160 that has been pre-processed by a workout up to dimension 118 and 138. In Figure 7 we display the wall-clock time taken for each Sieve during the pump up. All our GPU implementations use a multi-bucket parameter of 4, which should give a balanced comparison based on Figure 3. Any on-the-fly lifting or dual hash techniques are disabled. For the remaining parameters we refer to the next Section 7.2. The cross-over point between our `3-bdgl` and record-holding `triple` sieve from [AGPS19] is already in a sieving dimension of 94, and our speed-up grows to a speed-up of 2.7 in dimension 120. This shows that for CPU implementations BDGL is already extremely practical. For `2-bdgl_gpu` and `triple_gpu` the cross-over point lies above dimension 140, and given the extrapolations we expect them to cross in dimension $n \approx 149$. The large minimum bucket size shifts the cross-over point by more than 50 dimensions. In this light, it did not appear pertinent to implement `3-bdgl_gpu`, which, while being asymptotically faster, would cross-over even later.

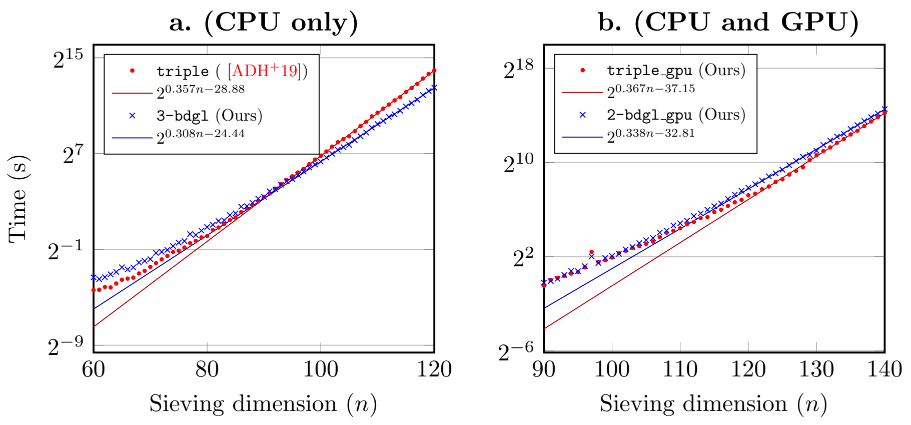


Fig. 7. Comparison of different sieve implementations from [ADH+19] and from this work. We ran a single pump up in a 160-dimensional lattice to a sieving dimension of 120 and 140 for CPU only and GPU accelerated respectively. The timings give the amount of time spend in each sieving dimension before reaching a saturation of 37.5% with a database size of $2.77 \cdot 2^{0.2075n}$. The fitting is obtained by a linear least-squares regression on the last 20 dimensions in log-space.

7.2 SVP Parameter Tuning

There are many parameters in our implementation that can be tuned for optimal performance with respect to memory and time complexity. We will focus on `triple_gpu` as we have shown it to be the fastest implementation in practical sieving dimensions $n \leq 150$. As low level parameters, such as minimum bucket sizes for GPUs, are discussed earlier, here we discuss the higher level parameters to solve 1.05-approxSVP for a lattice of dimension d.

Given the large amount of computational power available with the 4 GPUs, we can potentially solve lattice 1.05-approxSVP up to dimension 180 in reasonable time on a single machine. The main limiting factor at that point is the available memory, in our case 1.5 TiB RAM. We have spent significant efforts aiming to reduce the memory footprint of our G6K-GPU implementation, such as maintaining only basis coordinates, length and a hash of each vector in our database. Many parameters can be safely tweaked in certain regions without significantly affecting time complexity, hence we focus more on suitable values that limit memory usage.

To increase dimensions-for-free, and thus decrease memory usage, we enabled DownSieve for all workouts for a stronger preprocessing. We found that with DownSieve on, a larger PreferLeftInsert is more benificiary. I.e., prefer to insert even a slightly improved b'_i into the basis over a more significantly improved b'_{i+1}.

Another main parameter affecting memory use is the constant factor in database size, normally chosen as 3.2 in G6K [ADH+19]. We opted to reduce this to 2.77, resulting in $\mathsf{DBSize}(d) = 2.77 \times (4/3)^{(d/2)}$ for sieve dimension d, and compensate by also reducing $\mathsf{SaturationRatio}$ from .5 to .375.

Additionally, we introduced a database size limit by setting an experimentally-verified target dimensions-for-free $\mathsf{TD4F}(n) = \lfloor n/\log(n) \rfloor$, and limiting the database size to $\mathsf{DBSizeLimit}(n) = \mathsf{DBSize}(n - \mathsf{T4DF}(n))$. This means that the database size limit does not affect sieving up to the target dimensions-for-free. However, for unlucky cases, we allow G6K workouts of up to 4 dimensions larger without further increasing the database size. Because `triple_gpu` also considers triples we can be certain that saturation will still be reached.

As discussed before, we use DualHash lifting: starting from a sieving dimension of 106 in the filter context $[l - 24, l]$ using 32 dual vectors. To reduce memory overhead from storing buckets and results (before insertion), we set $\mathsf{MultiBucket} = 2$. Thus, our main parameters are:

$$
\begin{array}{ll}
\mathsf{TD4F}(n) = \lfloor n/\log(n) \rfloor, & \mathsf{MaxSieveDim}(n) = n - \mathsf{TD4F}(n) + 4, \\
\mathsf{DBSize}(d) = 2.77 \times (4/3)^{(d/2)}, & \mathsf{DBSizeLimit}(n) = \mathsf{DBSize}(n - \mathsf{T4DF}(n)), \\
\mathsf{SaturationRadius} = 4/3, & \mathsf{SaturationRatio} = .375, \\
\mathsf{DualHashMinDim} = 106, & \mathsf{DualHashDim} = 24, \quad \mathsf{DualHashVecs} = 32, \\
\mathsf{PreferLeftInsert} = 1.2, & \mathsf{DownSieve} = \mathsf{True}, \\
\mathsf{MultiBucket} = 2 & \mathsf{Sieve} = \texttt{triple\_gpu},
\end{array}
$$

7.3 New SVP Records

With the parameters tuned as discussed above, we have solved several Darmstadt Lattice 1.05-approxSVP Challenges for lattices with dimension in the range of 158 till 180 (all with seed=0). Details about the effort and results for each challenge are presented in Table 1.

With a new top record of the 1.05-approxSVP challenges with dimension 180, we improve significantly upon the last record of dimension 155 by [ADH+19]. Note that this last record was achieved on a single large machine with 72 CPU cores in 14 days and 16 hours, where we were able to find an even shorter vector of length $0.9842 \cdot \mathrm{gh}$ in about 5 hours ($68\times$ faster). Also we can improve this record from 155 by no less than 21 dimensions by solving lattice 1.05-approxSVP for dimension 176 on our 4-GPU machine in less wall-clock time: 12 days and 11 hours.

As proof we present our short vector for Darmstadt Lattice 1.05-approxSVP Challenge dimension 180 with seed 0:

(68, 33, -261, 11, 101, 354, -48, -398, 196, -84, 217, 319, -137, -157, -29, 304, -14, 312, 28,
-240, -347, -6, -153, -35, -214, 67, -565, 91, 365, 382, -168, 152, 30, 42, -12, -14, -230, 54,
304, 51, 398, 380, 76, -111, 437, 374, -554, -171, -90, -92, 564, 32, 217, 60, -107, 475,
-290, -326, -224, -218, 27, -271, 12, 200, 463, -365, 119, -431, 92, 450, 58, 183, 342, 82,
-144, 77, -95, -62, -245, 171, 169, -106, -330, 236, 194, 41, -84, -297, 567, 58, 553, 279,
260, 140, -141, -30, -183, -448, -112, 45, 135, -260, -261, 1, -105, 507, 105, -414, -161,
-9, -337, -287, 431, 92, -91, 350, -376, -75, 11, -249, 119, -172, -351, 410, 97, -320, -270,
223, -287, 97, 235, 242, 279, -222, 384, -95, 501, 317, 167, -130, -103, 441, 424, 25, 187,
-128, -9, -90, 328, -107, -132, -81, 2, 94, -326, -109, 465, 49, -30, 345, 125, -114, 909, 180,
-5, -112, 190, 182, -65, -291, -83, 445, -68, -318, -18, -732, -241, 246, -34, 299)

(T)D4F = target/actual dimensions for free
MSD = actual maximum sieving dimension
FLOP = # bucketing + reduction core floating point operations

dim	TD4F	D4F	MSD	Norm	Norm/GH	FLOP	Walltime	Mem GiB
158	31	29	129	3303	1.04329	$2^{62.1}$	9h 16m	89
160	31	33	127	3261	1.02302	$2^{61.8}$	8h 24m	88
162	31	31	131	3341	1.04220	$2^{63.2}$	18h 32m	156
164	32	28	136	3362	1.04368	$2^{64,8}$	2d 01h	179
166	32	30	136	3375	1.03969	$2^{64.8}$	2d 01h	234
168	32	31	137	3424	1.04946	$2^{65.3}$	2d 18h	318
170	33	31	139	3435	1.04594	$2^{66.3}$	5d 11h	364
172	33	35	137	3455	1.04582	$2^{65.0}$	2d 09h	364
174	33	35	139	3482	1.04913	$2^{66.3}$	5d 06h	518
176	34	33	143	3487	1.04412	$2^{67.5}$	12d 11h	806
178	34	32	146	3447	1.02725	$2^{68.6}$	22d 18h	1060
180	34	30	150	3509	1.04003	$2^{69.9}$	51d 14h	1443

Machine specification:
2× Intel Xeon Gold 6248 (20C/40T @ 2.5-3.9GHz)
4× Gigabyte RTX 2080 TI (4352C @ 1.5-1.8GHz)
1.5 TiB RAM (2666 MHz)
Average load: 40 CPU threads @ 93%, 4 GPUs @ 79%/1530MHz/242Watt

Table 1. Darmstadt Lattice 1.05-approxSVP Challenge results

dim	time	CPU+GPU only	system	CPU+GPU only	system
155	352 h	560 W	720 W	197 kWh	254 kWh
176	229 h	1268 W	1428 W	379 kWh	427 kWh

Table 2. Power use comparison for records of dimension 155 (G6K) and 176 (ours).

7.4 Remarks

Power use. To compare power efficiency of our new record computation for dimension 176 with the previous record computation for dimension 155 regarding power usage, we estimated power use as shown in Table 2 as follows. Their dimension 155 computation ran for 352 hours on 4 CPUs (Intel Xeon E7-8860V4) that have a TDP of 140 Watt each. Our dimension 176 computation ran for 299 hours on 2 CPUs (Intel Xeon Gold 6248) with a TDP of 150 Watt each, and 4 GPUs that typically used 242 Watt as measured through the `nvidia-smi` tool. For both systems we approximate other system power usage covering motherboard, RAM and disk as about 160W.

Note in Table 2 that while solving the challenge for dimension 176 is about two orders of magnitude harder compared to dimension 155, we spent less than a factor 2 more in electricity.

Memory use. From these, we estimate that our implementation requires about 416 Bytes per vector for dimensions higher than 137.Hence, sieving up to dimension 146 could still fit within our 1.5 TiB of available RAM, which allowed us to solve the lattice challenge of dimension 180.

References

[ADH$^+$19] Martin R Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W Postlethwaite, and Marc Stevens, *The general sieve kernel and new records in lattice reduction*, Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2019, pp. 717–746.

[AGPS19] Martin R Albrecht, Vlad Gheorghiu, Eamonn W Postlethwaite, and John M Schanck, *Estimating quantum speedups for lattice sieves*, Tech. report, Cryptology ePrint Archive, Report 2019/1161, 2019.

[AKS01] Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar, *A sieve algorithm for the shortest lattice vector problem*, Proceedings of the thirty-third annual ACM symposium on Theory of computing, 2001, pp. 601–610.

[ALNSD20] Divesh Aggarwal, Jianwei Li, Phong Q Nguyen, and Noah Stephens-Davidowitz, *Slide reduction, revisited—filling the gaps in svp approximation*, Annual International Cryptology Conference, Springer, 2020, pp. 274–295.

[AR05] Dorit Aharonov and Oded Regev, *Lattice problems in NP $\cap$ coNP*, Journal of the ACM (JACM) **52** (2005), no. 5, 749–765.

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven, *New directions in nearest neighbor searching with applications to lattice sieving*, Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms, SIAM, 2016, pp. 10–24.

[BGJ15] Anja Becker, Nicolas Gama, and Antoine Joux, *Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search.*, IACR Cryptol. ePrint Arch. **2015** (2015), 522.

[BHL$^+$19] Pierre Blanchard, Nicholas J Higham, Florent Lopez, Theo Mary, and Srikara Pranesh, *Mixed precision block fused multiply-add: Error analysis and application to gpu tensor cores.*

[BL16] Anja Becker and Thijs Laarhoven, *Efficient (ideal) lattice sieving using cross-polytope lsh*, International Conference on Cryptology in Africa, Springer, 2016, pp. 3–23.

[BLS16] Shi Bai, Thijs Laarhoven, and Damien Stehlé, *Tuple lattice sieving*, LMS Journal of Computation and Mathematics **19** (2016), no. A, 146–162.

[Cha02] Moses S Charikar, *Similarity estimation techniques from rounding algorithms*, Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, 2002, pp. 380–388.

[Che13] Yuanmi Chen, *Réduction de réseau et sécurité concrete du chiffrement completement homomorphe*, Ph.D. thesis, Paris 7, 2013.

[Duc18] Léo Ducas, *Shortest vector from lattice sieving: a few dimensions for free*, Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2018, pp. 125–145.

[FBB$^+$14] Robert Fitzpatrick, Christian Bischof, Johannes Buchmann, Özgür Dagdelen, Florian Göpfert, Artur Mariano, and Bo-Yin Yang, *Tuning gausssieve for speed*, International Conference on Cryptology and Information Security in Latin America, Springer, 2014, pp. 288–305.

[FP85] Ulrich Fincke and Michael Pohst, *Improved methods for calculating vectors of short length in a lattice, including a complexity analysis*, Mathematics of computation **44** (1985), no. 170, 463–471.

[GM03] Daniel Goldstein and Andrew Mayer, *On the equidistribution of hecke points*, Forum Mathematicum, vol. 15, De Gruyter, 2003, pp. 165–189.

[GN08] Nicolas Gama and Phong Q Nguyen, *Finding short lattice vectors within mordell's inequality*, Proceedings of the fortieth annual ACM symposium on Theory of computing, 2008, pp. 207–216.

[GNR10] Nicolas Gama, Phong Q Nguyen, and Oded Regev, *Lattice enumeration using extreme pruning*, Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2010, pp. 257–278.

[HK17] Gottfried Herold and Elena Kirshanova, *Improved algorithms for the approximate k-list problem in euclidean norm*, IACR International Workshop on Public Key Cryptography, Springer, 2017, pp. 16–40.

[HKL18] Gottfried Herold, Elena Kirshanova, and Thijs Laarhoven, *Speed-ups and time–memory trade-offs for tuple lattice sieving*, IACR International Workshop on Public Key Cryptography, Springer, 2018, pp. 407–436.

[HM19] Nicholas J Higham and Theo Mary, *A new approach to probabilistic rounding error analysis*, SIAM Journal on Scientific Computing **41** (2019), no. 5, A2815–A2835.

[JYP$^+$17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al., *In-datacenter performance analysis of a tensor processing unit*, Proceedings of the 44th Annual International Symposium on Computer Architecture, 2017, pp. 1–12.

[Kan83] Ravi Kannan, *Improved algorithms for integer programming and related lattice problems*, Proceedings of the fifteenth annual ACM symposium on Theory of computing, 1983, pp. 193–206.

[Laa15] Thijs Laarhoven, *Sieving for shortest vectors in lattices using angular locality-sensitive hashing*, Annual Cryptology Conference, Springer, 2015, pp. 3–22.

[LM18] Thijs Laarhoven and Artur Mariano, *Progressive lattice sieving*, International Conference on Post-Quantum Cryptography, Springer, 2018, pp. 292–311.

[MLB17] Artur Mariano, Thijs Laarhoven, and Christian Bischof, *A parallel variant of ldsieve for the svp on lattices*, 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), IEEE, 2017, pp. 23–30.

[MR09] Daniele Micciancio and Oded Regev, *Lattice-based cryptography*, Post-quantum cryptography, Springer, 2009, pp. 147–191.

[MV10] Daniele Micciancio and Panagiotis Voulgaris, *Faster exponential time algorithms for the shortest vector problem*, Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms, SIAM, 2010, pp. 1468–1480.

[MW16] Daniele Micciancio and Michael Walter, *Practical, predictable lattice basis reduction*, Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2016, pp. 820–849.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron, *Scalable parallel programming with cuda*, Queue **6** (2008), no. 2, 40–53.

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

[NVF20] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek, *Cuda, release: 10.2.89*, 2020.

[PS09] Xavier Pujol and Damien Stehlé, *Solving the shortest lattice vector problem in time 22.465 n.*, IACR Cryptol. ePrint Arch. **2009** (2009), 605.

[SE94] Claus-Peter Schnorr and Martin Euchner, *Lattice basis reduction: Improved practical algorithms and solving subset sum problems*, Mathematical programming **66** (1994), no. 1-3, 181–199.

[SG10] Michael Schneider and Nicolas Gama, *Darmstadt SVP Challenges*, https://www.latticechallenge.org/svp-challenge/index.php, 2010, Accessed: 06-10-2020.

[YKYC17] Shang-Yi Yang, Po-Chun Kuo, Bo-Yin Yang, and Chen-Mou Cheng, *Gauss sieve algorithm on gpus*, Cryptographers' Track at the RSA Conference, Springer, 2017, pp. 39–57.

8 Supplementary Materials

A BDGL-like bucketing - Implementation Details

The theoretical version of [BDGL16] picks the local bucket centers C_i uniformly at random over the sphere, which requires one to fully store $C_i = \{\mathbf{c}_1, \dots, \mathbf{c}_{m'}\}$ and to compute km' generic inner products per vector to bucket, where $m' = m^{1/k} \cdot 2^{-(k-1)/k}$. One can certainly decrease bucketing costs by making k larger, however, this affects the geometric quality of the bucketing: for the overall performances of the sieve, it is therefore worthwhile optimizing the local bucketing step. Our approach is to not let the set C_i be entirely random, but in fact only be defined implicitly, in way that allows very fast bucketing. We first explain our method specifically targeted for the `AVX2` CPU instruction set, and then we discuss how to similarly implement it for the GPU.

Bucketing on `AVX2` CPU Cores. For the sake of this discussion, let us fix the dimension of the local blocks to $\frac{n}{k} = 48$. We represent the input vector $\mathbf{v} \in \mathbb{R}^{48}$ using 16-bits fixed-point precision; so that we can fit $\mathbf{v}$ within 3 `AVX2` registers. We then construct on-the-fly a pseudo-random sequence of $m'' = m'/32$ permutations π_i acting over the 48 coordinates of $\mathbf{v}$. For brevity these permutations also include pseudo-random negations. The permutation are constructed by sequential updates as $\pi_i = \tau_i \circ \pi_{i-1}$.

The pseudo-randomness is obtained simply by iterating *a single round* of `AES-NI` using the seed both for fixed key material and initial value; each round produces 128 pseudo-random bits used for a permutation update (the choice for AES is due to hardware-acceleration). The permutation update τ_i is constructed from this pseudo-random bits by combining `AVX2` bytewise `shuffle` instructions within each `AVX2` registers (chosen pseudo-randomly among a few predefined shuffles), and controlled swaps between pairs of `AVX2` registers (directly constructed from the pseudo-randomness using bitwise operations).

Finally, we also rely on the Fast Hadamard transform $\mathcal{H} : \mathbb{R}^{48} \to \mathbb{R}^{32}$ over the first 32 coordinates of a permuted vector $\mathbf{w} = \pi(\mathbf{v}) \in \mathbb{R}^{48}$. We note that each output of $\mathcal{H}(\pi(\mathbf{v}))$ for a random permutation implicitly corresponds to an inner product $\langle \mathbf{v}, \mathbf{c} \rangle$ for some ternary vector $\mathbf{c} \in \{-1, 0, 1\}^{48}$ of hamming weight 32. Because the output of the Hadamard transform $\mathcal{H}$ is also in the form of `AVX2` registers, extracting (the index of) $\arg\max_{\mathbf{c} \in S} \langle \mathbf{c}, \mathbf{v} \rangle$ can also be done in a vectorized and on-the-fly manner.

For large m', we can bucket a vector $\mathbf{v}$ in less than $1.6 \cdot m'$ many CPU cycles. Another advantage is that this whole procedure fits within register memory: there is no need to access the vectors $\mathbf{c} \in S$ from memory. This therefore leaves all the memory bandwidth and cache storage to concurrent processes working on other tasks.

Bucketing on GPU cores. We also created a GPU accelerated version `bdgl_gpu` based on the ideas from the CPU implementation. We describe some implementation details of the local bucketing step.

Recall that 32 threads are grouped in a warp, following the single-instruction multiple-data paradigm. One could interpret instructions executed by a warp as `AVX2` instructions on very wide (1024 bit) registers. We let each warp process multiple vectors at the same time and each vector $\mathbf{v} = (v_1, \ldots, v_{n/k})$ is distributed over the 32 threads by storing v_i in thread $i \pmod{32} \in \{1, \ldots, 32\}$. Processing multiple vectors at the same time allows to amortize the cost of generating the appropriate randomness and to hide data latencies.

Depending on the dimension we use a Hadamard transform over the first 32 or 64 coefficients, extracting 32 or 64 implicit inner products at a time respectively. Similar to `AVX2` we have the `__shfl_sync` instruction to move data between threads in a warp. For example the `__shfl_xor_sync` instruction exchanges data between all threads i and `XOR` (i, c) for some value c. We used this to implement the Fast Hadamard Transform with a logarithmic number of such exchanges in a straightforward way.

The pseudo-randomness is obtained from the cuRAND library. The total permutation consists out of three parts. First we try to fully permute the coefficients inside the Hadamard region by doing random swaps using the `__shfl_xor_sync` instruction and a coin-flip over several rounds. Note that exchanging threads also need to do a shared coin-flip, but only once per round for all vectors that are processed at the same time. The second step consists out of random sign flips of coefficients inside the Hadamard region. Lastly coefficients from outside the Hadamard region are randomly swapped with coefficients inside the region, this happens locally on each thread.

To prevent many branches and a high overhead each thread only stores the best encountered bucket for each vector it processes. So thread $i \in [1, \ldots, 32]$ stores the best of the buckets $i, i + 32, i + 64, \ldots$ for each vector. Of these 32 results we then take the best M buckets. This is a performance trade-off and as a result we do not necessarily obtain the best M buckets overall.

B Reduction Kernel - Implementation Details

In this section we discuss some implementation details of our Tensor-GPU kernel to find reductions. For bucketing methods as `triple_gpu` the implementation is similar. For performance results see Figure 4. We consider a bucket of s vectors $\mathbf{Y} := [\mathbf{y}_1, \ldots, \mathbf{y}_s] \in \mathbb{R}^{n \times s}$ and we need to filter pairs based on their inner product. Note that we essentially want to compute one half of the matrix $\mathbf{Y}^t\mathbf{Y}$.

Fragments. When working with Tensor cores a fundamental building block is a fragment: a 16×16 `fp16` matrix that is distributed over a warp, each thread storing 8 of the 256 values. CUDA allows a matrix multiply and accumulate operation $\mathbf{C} = \mathbf{C} + \mathbf{AB}$ with fragments $\mathbf{A}, \mathbf{B}, \mathbf{C}$ accelerated by the Tensor cores.

Note that the accumulation fragments that contain the resulting inner products are distributed over the threads in a black-box manner which is not specified by NVIDIA; each thread knows some inner product values but not to which pairs they belong. The official solution is to first store the results back to shared memory using a special CUDA instruction, but this severely degrades performance.

We reverse engineered the distribution so we can immediately process the inner products while they are still stored in registers, something which might break in future hardware or CUDA versions. Additionally we also used this to improve the loading of fragments from global memory. Before starting the expensive reduction kernel in which every SM loads fragments of $\mathbf{Y}$ from global memory we first reorder the storage of $\mathbf{Y}$ such that all 8 `fp16` coefficients that are stored on a single thread are stored in a consecutive 128 bits range, allowing to load it with a single coalescing instruction directly into the correct local registers, instead of 4 different non-coalescing 32-bit load instructions without this reordering.

Data movement. Optimizing a GPU kernel is all about data movement. Given the memory hierarchy (see Figure 1) with different capacities, bandwidths and latencies the main challenge is to get the data into the registers in time to run the appropriate computations. By reusing data locally as much as possible we can improve the ratio of computations versus memory that is moved.

Each block computes a row of $\mathbf{Y}^t\mathbf{Y}$ of 128 vectors wide, using 8 warps, i.e., 256 threads in total. As a result each coefficient that is loaded from $\mathbf{Y}$ is used for 128 multiply and add operations, enough to prevent a significant memory bottleneck between global memory and each SM. Inside each row we process a matrix block of 128×256 at a time, where each warp takes care of $4 \times 4 = 16$ fragments in a 64×64 matrix sub-block (see Figure 8). After processing a matrix block we shift to the next one, until we exceed the diagonal; we only want to compute half of the symmetric result matrix.

When loading a row of fragments of $\mathbf{Y}$ from global memory all 8 warps in a block work together to only fetch each element once, the values are temporarily stored in what we call *cache registers.* From here these elements are stored in shared memory after which all warps are synced such that we can be sure the shared memory contains the correct elements. Then each warp separately loads the 4 vertical and 4 horizontal fragments that it needs to compute its 64×64 matrix sub-block from shared memory to what we call the *compute registers.*

Loading data from shared and global memory involves significant latencies from tens to hundreds of cycles. During this time we need to make sure our kernel is still computing with data that is already in registers. For example while data is being loaded from global memory we can do computations with the data in our shared memory (with a much lower latency). A well know technique to further hide latencies is double buffering in which we double the amount of registers and shared memory we mentioned before. While one half of the compute registers is used for computation the other half is obtaining new data from the shared memory; continuously alternating their roles to hide the shared memory latency. Note that all these techniques are limited by the amount of registers and shared memory we have, so they require a careful balancing. The double buffering technique is also used between the CPU and GPU to simultaneously transfer new bucket vectors and old results during kernel executions.

Processing the results. While computing the inner products is the computational intensive part of the reduction kernel filtering out the results also involves

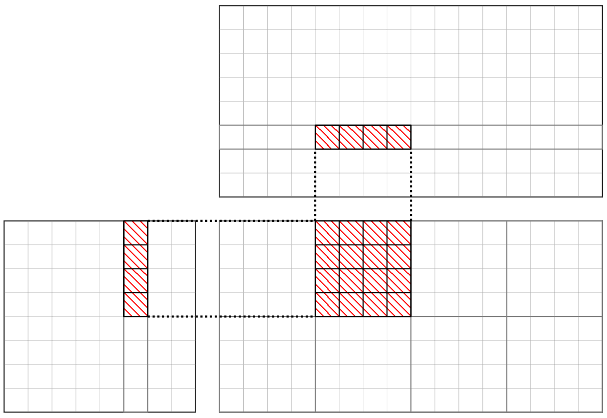


Fig. 8. Example of the computation on a 64×64 sub-block done by a single warp. Each cell is a 16×16 matrix fragment. The full 128×256 block is processed by 8 warps in parallel.

some overhead. Especially since this has to happen using regular CUDA cores, which are relatively slow compared to Tensor cores. Also filtering results involves branches, something which can heavily degrade performance when threads in a warp diverge in which branch they take.

For the filtering of pairs we have to compare the value of the computed inner product with $(\|\mathbf{y}_i\|^2 + \|\mathbf{y}_j\|^2 - \ell^2)/2$. To avoid having to compute this value completely we pre-compute for each vector the value $\frac{1}{2}\|\mathbf{y}_i\|^2 - \frac{1}{4}\ell^2$ such that for each pair the comparison value can be computed by a single addition. Similarly when checking for triples we pre-compute the value $\frac{1}{4}\ell^2 - \frac{1}{4}\|\mathbf{b}\|^2 - \|\mathbf{y}_i\|^2 + \langle \mathbf{b}, \mathbf{y}_i \rangle$ for each vector. Note that after this pre-computation we do not even have to pass the length bound ℓ to the kernel as this is already incorporated in the pre-computed values.

SimHash filter. We quickly discuss why for our Tensor-GPU implementation we did not choose to make use of the so-called SimHash filter (see [Cha02, FBB+14,Duc18]) based on fast binary operations that has been used successfully to speed-up sieve implementations for the CPU. We do note that the Tensor cores have support for the necessary binary computations. Our reasons are as follows. Firstly the speed-up from using a SimHash filter is less with respect to the 16-bit GPU computations than with respect to the 32-bit CPU computations. Secondly post-processing the passing pairs and triplets on the GPU (recomputing their lengths) has to happen in higher precision on relatively slow CUDA cores, something which can quickly become a bottleneck with a low fidelity filter such as the SimHash. And lastly to compensate for false negatives of the SimHash the database size needs to be a larger, which hinders our focus on minimizing RAM usage.

C Tensor cores and precision

The main drawback of the Tensor cores is the low-precision at which computations are executed. In particular we consider the case that the vectors are represented using the `fp16` format. In this section we show that for our purpose of computing pairwise inner products this raises no significant problems. By first normalizing the vectors we can assume for analysis that they are unit vectors (a normalization close to unit length is actually done in our implementation).

We assume the worst-case, namely that we also accumulate the inner products in 16-bit precision. Although the Tensor cores have an option to use 32-bit accumulate, which severely increases the precision, this is capped at half speed in some of the consumer GPUs. As the GPU is more used as a filter to find (extremely) likely candidates for reduction, we can tolerate some error, say up to 2^{-7} in the computed inner product between two unit vectors, at the loss of more false positives or missed candidates. Furthermore it is acceptable for our purposes if say 1% of the close vectors are missed because of even larger errors. Computing the inner product of two vectors using Tensor cores leads to errors in two places, first there is some error when representing the vectors in 16-bit precision, and secondly some error accumulates during the inner product computation.

First we show that the representation error between a unit vector $\mathbf{y} = (y_1, \ldots, y_n) \in \mathbb{R}^n$ of unit length and its closest 16-bit representative $\hat{\mathbf{y}}$ is small. With a 5 bit exponent and 10 bit mantissa we have $|y_i - \hat{y}_i| \leq \max\{|y_i| \cdot 2^{-11}, 2^{-25}\}$. For the full unit vector this gives an error bound of

$$\|\mathbf{y} - \hat{\mathbf{y}}\| \leq \max\{2^{-11}, 2^{-25} \cdot \sqrt{n}\}. \tag{1}$$

Note that for the scope of lattice sieving the dimension n is at most a few hundred, and thus we can safely assume a fixed error bound of 2^{-11} independent of the dimension. By the Cauchy-Schwarz inequality the representation error contributes at most $\max\{2^{-10}, 2^{-24} \cdot \sqrt{n}\}$ to the eventual pairwise inner product between unit vectors and is thus small enough compared to the tolerated error of 2^{-7}.

Next we have to consider the computation of the inner product. For each combined multiplication and addition there can be some small error at most μ; we have $\mu \leq 2^{-12}$ in the $[-1, 1]$ range. The main problem is that these errors can accumulate resulting in a worst case error of the form $\mu \cdot n$, which could already be problematic for say $n \geq 32$. For Tensor cores n can be replaced by $n/4$ because the Tensor cores act on 4×4 blocks and the internal computation is at a higher precision [BHL+19], but this still gives a worst-case bound that is too large for our regime.

These worst-case bounds assume that the error at each operation is both maximal and of the same sign, something that is not observed in practice. A common heuristic for an average-case analysis is the assumption that the error is equally likely to be positive as negative, which improves the accumulated error from $\mu \cdot n$ to $O(\mu\sqrt{n})$ with high probability. See [HM19] for probabilistic bounds under such error models. We see in Figure 5 that this heuristic randomized analysis closely matches the experimentally observed growth $\Theta(\sqrt{n})$ of the computation error.

We conclude that when accepting some inaccuracy in a small ratio of inner products the 16-bit Tensor cores contribute effectively in dimensions as large as 2^{11}.

D Dual hash - Implementation Details

Given a list of pre-computed $\mathbf{Dt}_1, \ldots, \mathbf{Dt}_m$ we want to use the GPU to efficiently compute $\mathrm{dist}(\mathbf{D}(\mathbf{t}_i - \mathbf{t}_j), \mathbb{Z}^h)^2$ for all $i < j$, where the coordinates use an 8-bit integer representation. We focus on the core computation as the for the memory movement one can apply a similar strategy as for the reduction kernel (see Appendix B).

Implementation for CUDA cores. Although CUDA cores are flexible we have to pack 4 of the 8-bit values together in a 32-bit register $a = a_3|a_2|a_1|a_0$ and apply special operations to obtain optimal 8-bit arithmetic performance. First we need to take the coordinate-wise different and then take the modulo to get back in the interval $[-1/2, 1/2)$. By representing these values using signed integers the modulo is equivalent to integer overflow. CUDA has an operation to take the coordinate-wise difference, however this actually decodes into multiple instructions. Therefore as a trade-off we just take the 32-bit integer different between a and b, ignoring off by one errors in each coordinate due to a possible carry bit. For computing the length we can use the relatively new operation `__dp4a(a,b,c)` that computes the inner product between the packed a and b and accumulates this in a 32-bit integer c. So in only two cycles we can compute the squared distance over 4 coordinates modulo $\mathbb{Z}^4$.

Implementation for Tensor cores. Although Tensor cores can be up to 4 times faster than the CUDA cores for similar precision they can basically do only a single thing well: pairwise inner products, i.e., a matrix product. So to use the Tensor cores effectively we need to convert this computation to that of a matrix product. We quickly discuss how one could potentially make such an adjustment.

We use the fact that $\mathrm{dist}(x, \mathbb{Z})^2 \approx (1 - \cos(x \cdot 2\pi))/16$ for $x \in [-\frac{1}{4}, \frac{1}{4}] + \mathbb{Z}$. For the remaining range the value of $(1 - \cos(x \cdot 2\pi)/16$ is somewhat smaller, but in the BDD-regime we are working these values still seem large enough to reject bad vectors. So using this similarity we want to compute

$$d_{ij} := \sum_{l=1}^{h} \frac{1 - \cos((\mathbf{D}(\mathbf{t}_i - \mathbf{t}_j))_l \cdot 2\pi)}{16}.$$

from some pre-computed values depending on $\mathbf{Dt}_i$ and $\mathbf{Dt}_j$ respectively. Rewriting using trigonometric identities we get:

$$d_{ij} = \frac{m}{16} - \frac{1}{16}\left(\sum_{l=1}^{m} \cos(2\pi\mathbf{Dt}_i)\cos(2\pi\mathbf{Dt}_j) + \sin(2\pi\mathbf{Dt}_i)\sin(2\pi\mathbf{Dt}_i)\right).$$

If we pre-compute that values $(\cos((\mathbf{Dt}_i)_l \cdot 2\pi))_l$ and $(\sin((\mathbf{Dt}_i)_l \cdot 2\pi))_l$ we can compute the above sum by computing two h-dimensional (pairwise) inner products which Tensor cores can do efficiently. Again we can use 8-bit representations and computing the inner product (with 32-bit accumulate) is up to 4 times faster on Tensor cores compared to CUDA cores. However here we need to compute two h-dimensional inner products instead of one, but this is compensated by not having to compute the coordinate-wise difference first. So one could expect up to 4 times the performance compared to CUDA cores.