CHAPTER 4

Advanced Lattice Sieving on GPUs, with Tensor Cores

This chapter is an extended version of the joint work 'Advanced lattice sieving on GPUs, with tensor cores', with Léo Ducas and Marc Stevens, published at Eurocrypt 2021.

4.1 Introduction

Lattice sieving algorithms [AKS01; NV08; MV10; HPS11a] are asymptotically superior to enumeration techniques [FP85; Kan83; SE94; GNR10], but this has only recently been shown in practice. Progress on sieving, both on its theoretical [Laa15; BGJ15; BDGL16; HKL18] and practical performances [Fit+14; Duc18; LM18; Alb+19], brought the cross-over point with enumeration as low as dimension 80. Practical performance is often measured by solving TU Darmstadt SVP Challenges, given a random challenge lattice, one has to find a lattice vector of length $1.05 \cdot \mathrm{gh}(\mathcal{L})$, i.e. a factor 1.05 above the expected length of the shortest vector. The work of M. R. Albrecht et al. at Eurocrypt 2019 [Alb+19], 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 enumeration techniques on a large cluster with multiple orders of magnitude more computational resources. They achieved this with an asymptotically non-optimal sieving algorithm based on [BGJ15; HK17], which with sufficient memory runs in time $2^{0.349n+o(n)}$.

While it is now clear that, in terms of time, sieving beats enumeration not only asymptotically, but also in practice, there are still two big open questions in the development of practical lattice sieving algorithms.

Firstly, can the asymptotically optimal sieve [BDGL16], running in time $2^{0.292n+o(n)}$, be made practical and improve on the current record-holding sieve in feasible dimensions? While concrete parameters for lattice-based cryptography schemes are based on the asymptotic complexity of this sieve, there has to date been no practically (or even asymptotically) efficient implementation of it. The asymptotic analysis of the algorithm hides some potentially large polynomial and subexponential factors, which are poorly understood from a concrete perspective. Additionally, a naive implementation leads to subexponential time, or even exponential space overhead factors, that make the algorithm uncompetitive [MLB17] in feasible dimensions. Another interesting question is how the properties of this sieve interact with parallelization efforts that are necessary for large scale attacks.

This brings us to the second open question: can these advanced sieving algorithms scale efficiently beyond a shared-memory architecture, e.g., to a large cluster of computers? The large number of lattice vectors and their interactions makes lattice sieving algorithms non-trivial to parallelize. Before scaling up to a cluster of computers, a natural intermediate 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 and 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.

4.1.1 Related work

We give here a short survey of past efforts on the parallelization of lattice sieving algorithms.

In 2011, Milde and Schneider [MS11] gave the first parallel implementation of the Gauss Sieve. The global list is distributed over several independent Gauss Sieve instances that are connected in a cycle. A vector is passed along and reduced with each local list and vice versa, before being inserted. As a result the vectors in the distributed list are nearly pairwise reduced; nearly because the lists could have changed in the meantime. This method scales almost linearly up to about 5 threads, but the efficiency quickly degrades above that due to the many non pairwise reduced vectors in the distributed list.

This problem was tackled by Ishiguro et al. [IKMT14] in 2014: by storing a full copy of the list in shared-memory on each node, they make sure that all vectors in the distributed list are pairwise reduced. The resulting implementation scales well to hundreds of threads and, by also abusing the ideal structure, was used to solve a TU Darmstadt Ideal SVP challenge in dimension 128. However, because each node has to store the full list, the maximum list size, and thus the largest achievable dimension, is limited by the smallest node.

Also in 2014, Mariano et al. [MDB14] implemented a parallel version of the List Sieve algorithm, with super-linear scalability at least up to 32 threads. The claimed super-linear speed-up is possible due to relaxations to the List Sieve properties made by the parallel variant. Around the same time Mariano et al. [MTB14] proposed a fine-grained parallel shared-memory implementation of the Gauss Sieve that achieves almost linear speed-up at least up to 64 cores, and achieves similar run times as Ishiguro et al.

Later that year Bos et al. [BNP17] presented another distributed implementation of the Gauss Sieve similar to the works of Milde et al., and Ishiguro et al., solving the scalability problems of the former in a different way, while maintaining the use of fully distributed lists.

In 2017, both Ishiguro et al.' and Bos et al.' parallel Gauss Sieve implementations were adapted to a (multi-)GPU setting by Yang et al. [YKYC17], solving a TU Darmstadt Ideal SVP challenge in dimension 130 on 8 GPUs in 824 hours.

The first parallel implementations for more advanced and asymptotically faster bucketed sieves such as the Hash Sieve [Laa15] and BDGL Sieve [BDGL16] were introduced by Mariano et al. in 2015 [MBL15] and 2017 [MLB17] respectively. Both implementations use a queued model similar to the Gauss Sieve, which in combination with the bucketing techniques leads, both asymptotically and in practice, to a much higher memory usage, making them infeasible for dimensions above 100.

In 2019, a large combined effort of Albrecht, Ducas, Herold, Kirshanova, Postlethwaite, and Stevens [Alb+19] introduced the open-source General Sieve Kernel (G6K), which combined many of the state-of-the art advanced techniques such as Progressive Sieving and Dimensions for Free into a single implementation. G6K assumes a single node shared-memory model, and contains several sieving variants, from simple List and Gauss Sieves to parallel variants of [BGJ15] and [HK17]. The authors also set a new TU Darmstadt SVP challenge record at dimension 155 in about 14 days using 72 CPU cores. For more information about G6K see Section 3.5.

4.1.2 Contributions

The main contribution of this work is to answer the two open questions (partially). We show that lattice sieving, including the more advanced 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 only 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.

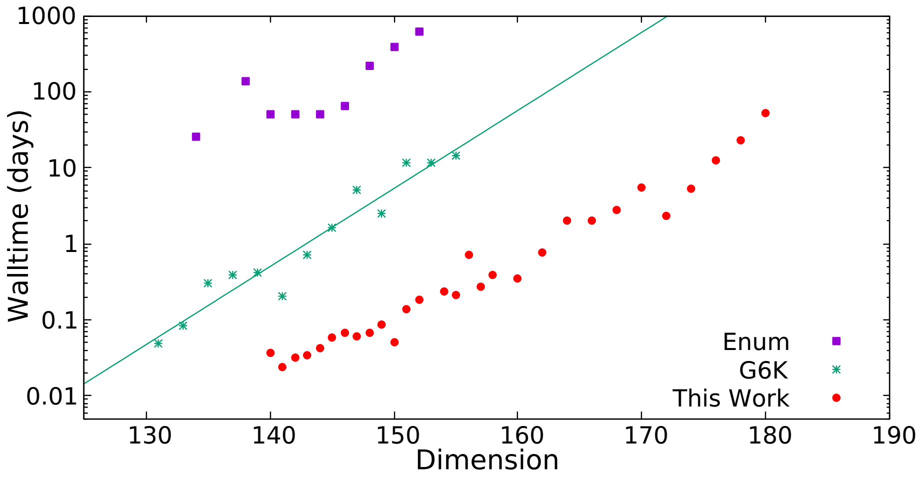


Figure 4.1: Solved TU Darmstadt lattice challenges measured in wall-clock time[1], as reported on the challenge website [SG10]. Details on the used hardware are also reported there. In short, the enumeration records used large clusters of computers, while the sieving records used a single (large) server.

Secondly, we introduce the first practical implementation of the asymptotically best BDGL Sieve from [BDGL16] inside the G6K framework, both for CPU-only (multi-threaded and `AVX2`-optimized) and with GPU acceleration. In contrast to [MLB17] our implementation does not use significant extra memory, and can therefore be used in much higher dimensions. We use this to shed some light on the practicality of this algorithm. In particular we show that our CPU-only BDGL Sieve variant already improves over the previous record-holding sieve in sieving dimensions as low as 90, with a $5\times$ speed-up around dimension 120. Furthermore, we show that this cross-over point lies much higher for our GPU accelerated sieve due to memory-bottleneck constraints.

[1]Comparing wall-clock times between distinct hardware is in general not a good practice, but the different nature of CPUs and GPUs makes it hard to fairly

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; Alb+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

The CPU implementation of the BDGL Sieve has been integrated in G6K, with further improvements, and we aim for long term maintenance.[2]. The GPU implementations has also been made public, but with lower expectation of quality, documentation and maintenance.[3]

4.1.3 Prerequisites

The reader of this chapter is assumed to be familiar with the contents of Chapter 3, in particular lattice sieving, bucketing techniques, Dimensions for Free and G6K.

4.2 GPU architecture and sieve design

In this section we explain the architecture of a GPU, the global design of our new sieve implementations and how we reduce memory usage and movement.

4.2.1 GPU device architecture

We give a short summary of the NVIDIA Turing GPU architecture on which our implementations and experiments are based. Since this

compare them on other metrics. As further motivation the estimated energy usage of our server was only a factor 2 higher than the one used for the G6K records.

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

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

work was done a new generation named Ampere was launched, doubling many of the performance metrics mentioned here.

CUDA cores and memory

An 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, but is relatively slow. Global memory operations always pass through a GPU-wide L2 cache. In addition, each SM benefits from an individual L1 cache and offers a fast addressable *shared memory* that can only be used by threads in that block. The fastest type of memory are the *registers*, which are local to each thread, and which act as the input and output of computational operations.

To implement GPU kernel functions for an 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

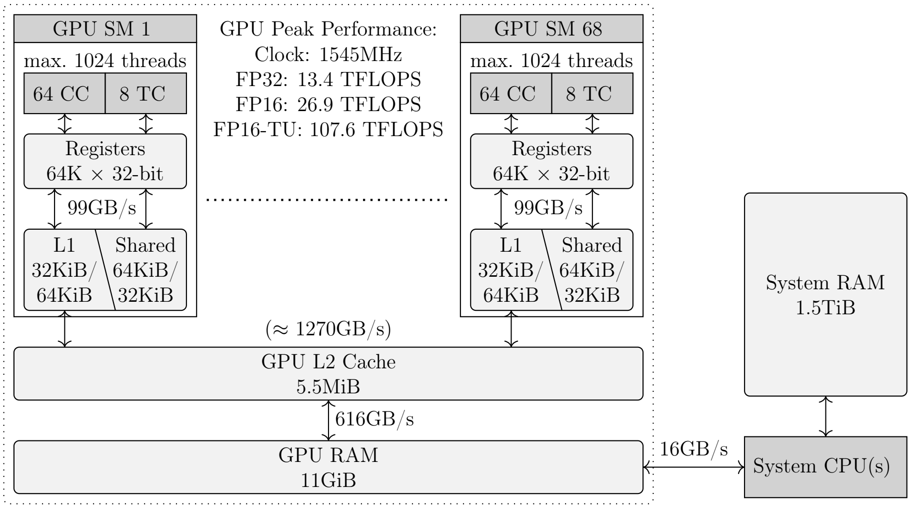


Figure 4.2: Device architecture of the NVIDIA RTX 2080 Ti and the system used in this work.

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.

RTX2080 Ti details. In this work we used the NVIDIA RTX2080 Ti with 68 Streaming Multiprocessors, whose architecture is depicted in Figure 4.2. There are 64 cores in each SM, that can service up to 1024 resident threads. There is a large register file of 65536×32-bit registers, where up to 256 registers can be assigned to a thread. Each SM has a dedicated 96KiB memory that is split into a private L1 cache and shared memory (either 32-64 KiB or 64-32 KiB split). The Turing architecture is the 2nd NVIDIA GPU generation to feature special Tensor units, eight per SM.

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. The specialized Tensor cores enable the RTX2080 Ti to theoretically reach in the order of 107 `fp16`-TFLOPS, improving by two orders of magnitude on a single high-end CPU.

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. Showing that these simple cores can be used effectively also shortens the gap to design low-cost and highly efficient customized hardware for lattice sieving.

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 had a price of about €2500 and has a TDP of 150 Watt, while the GPU was priced at about €1000 and has a TDP of 260 Watt. For 32-bit floating point operations the theoretical peak performance is given by 3.2 TFLOPS[4] 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.

4.2.2 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

[4]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.

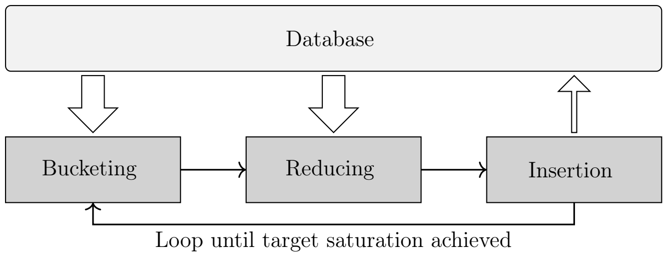


Figure 4.3: High level diagram of the implemented Sieving process.

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. 4.3) 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 previous record setting `bgj1` and `hk3` sieves; 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. How the subdivision is done is precisely where sieving variants differ, and how different (asymptotic) time complexities are achieved. 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 precisely 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 4.3, 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 4.4 we further discuss the reduction phase and 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 4.5 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.

4.2.3 Data storage and movement

Recall from Section 3.5 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 one does consider 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 incidental 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.

4.3 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 `hk3` 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.

4.3.1 BGJ-like bucketing (`triple_gpu`)

Recall from Section 3.3 that the bucketing method used in `bgj1` and `hk3` is based on spherical caps, directed by explicit bucket centers that are also lattice points. Inspired by this we start the bucketing phase by first choosing 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 we associate each vector $\mathbf{v} \in L$ in our database to the bucket $B_{k_\mathbf{v}}$ where

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

In this we differ from the original versions of `bgj1` and `hk3` [BGJ15; HK17; Alb+19] in that they use a fixed filtering threshold on the angle $|\langle \mathbf{b}_k/\|\mathbf{b}_k\|, \mathbf{v}/\|\mathbf{v}\| \rangle|$. We further relax this condition somewhat by the multi bucket parameter M, to associate a vector to the best M buckets. As a result our buckets do not exactly match spherical caps, but they should still resemble them; in particular such a change does only affect the asymptotic analysis up to subexponential factors. We chose for this alternation as this fixes the amount of buckets associated to each 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 $M \le 16$ 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 Section 4.4.1 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.

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

Recall from Section 3.3 that the asymptotically optimal bucketing method from [BDGL16] improves over the `bgj1` and `hk3` sieves by using structured bucket centers. One splits the dimension n into k smaller blocks of similar dimensions $n_1, \ldots, n_k$ that sum up to n, and (after some orthonormal transformation) the set C of bucket centers is a direct product $C = C_1 \times \pm C_2 \cdots \times \pm C_k$ of local bucket centers. 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.

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 $\tilde{O}(N^{1/2})$ and a time complexity of $2^{0.349n+o(n)}$. For $k = 2$ or $k = 3$ the optimal buckets become smaller of size $\tilde{O}(N^{1/3})$ and $\tilde{O}(N^{1/4})$ respectively and of higher quality, 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., $k = \Theta(\log(n))$ there will only be a subexponential $2^{o(n)}$ number of vectors in each bucket, leading to the best known time complexity of $2^{0.292n+o(n)}$.

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 $32 \leq n_i$ 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 $17 \leq n_i \leq 128$. We first explain implementation details 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.

The CPU implementation of `bdgl` has been integrated in G6K.[5] As it may be of independent interest, the `AVX2` bucketer is also provided as a stand-alone program.[6]

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 $\mathbf{v} = (v_1, \dots, v_{n/k})$ is distributed over the 32 threads by storing v_i in thread $i \pmod{32} \in \{1, \dots, 32\}$. 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

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

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

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 $i \in [1, \dots, 32]$ stores the best of the buckets $i, i + 32, i + 64, \dots$ 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. We will see in Section 4.3.3 that for small values of M this does not influence the bucketing quality that much.

4.3.3 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 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 direct

[7]Volumes of caps and wedges for predicting the idealized behavior where extracted from [AGPS20], and more specifically `https://github.com/jschanck/eprint-2019-1161/blob/main/probabilities.py`.

product and Hadamard construction. In particular the product construction has an inherent subexponential quality loss, which is further analysed in [Duc22].

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 4.4. 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

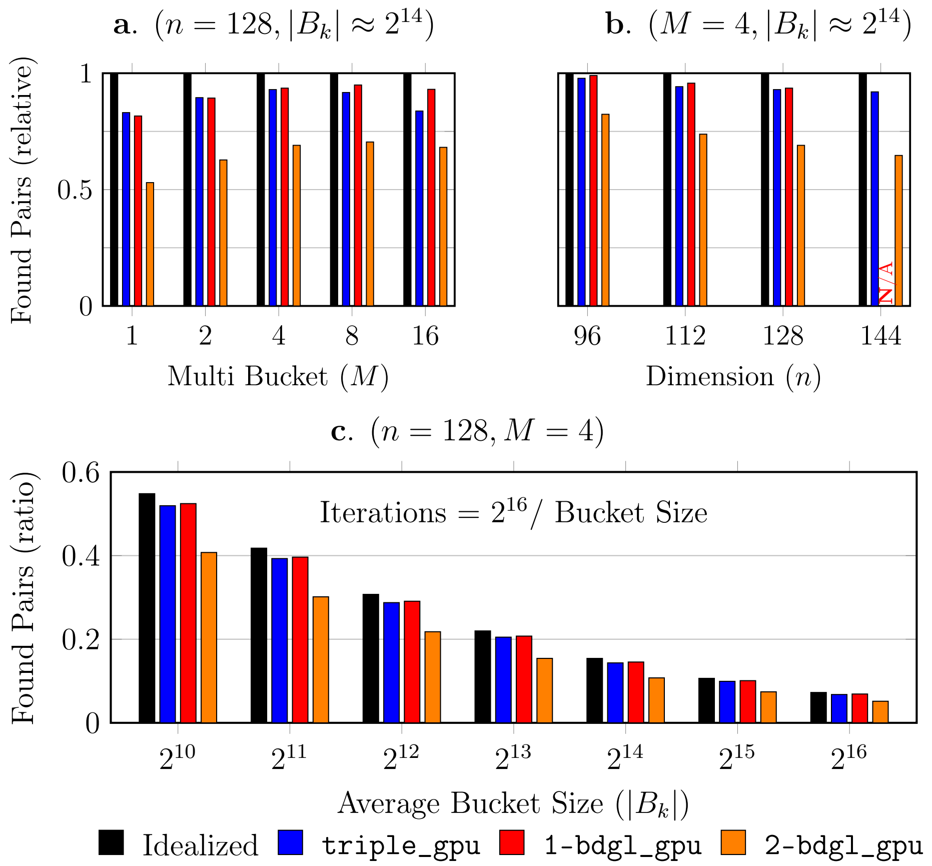


Figure 4.4: Bucketing Quality Comparison. We sampled 2^{20} pairs $\mathbf{v}, \mathbf{w}$ of unit vectors such that $|\langle \mathbf{v}, \mathbf{w} \rangle| = 0.5$ and we measured how many fell into at least 1 common bucket. The number of buckets is computed based on the desired average bucket size $|B_k|$, 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.

loss potentially as large as subexponential [BDGL16, Theorem 5.1].

Now we focus our attention on graph **c.** of Figure 4.4 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 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.

4.4 Reducing

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 $\mathbf{v}_1, \dots, \mathbf{v}_s \in \mathbb{R}^n$ with bucket center $\mathbf{c}$. Only the $\mathbf{x}$-representations are sent to the GPU and there they are converted to the 16-bit Gram-Schmidt representations $\mathbf{y}_1, \dots, \mathbf{y}_s$ and $\mathbf{y_c}$ that are necessary to quickly compute inner products. Together with the pre-computed squared lengths $\|\mathbf{y}_1\|^2, \dots, \|\mathbf{y}_s\|^2$ and inner products $\langle \mathbf{y_c}, \mathbf{y}_1 \rangle, \dots, \langle \mathbf{y_c}, \mathbf{y}_s \rangle$, 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

for **pairs:**

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

for **triples:**

$$\langle \mathbf{y}_i, \mathbf{y}_j \rangle \leq -\frac{\|\mathbf{y_c}\|^2 + \|\mathbf{y}_i\|^2 + \|\mathbf{y}_j\|^2 - \ell^2 - 2\langle \mathbf{y_c}, \mathbf{y}_i \rangle - 2\langle \mathbf{y_c}, \mathbf{y}_j \rangle}{2}.$$

And thus we need to compute all pairwise inner products $\langle \mathbf{y}_i, \mathbf{y}_j \rangle$. If we consider the matrix $\mathbf{Y} := [\mathbf{y}_1, \dots, \mathbf{y}_s] \in \mathbb{R}^{n \times s}$ 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 [NVI], or the open-source, but slightly less optimal CUTLASS library [Ker+22]. 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.

4.4.1 Reduction kernel

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 4.6. 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 4.2) 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

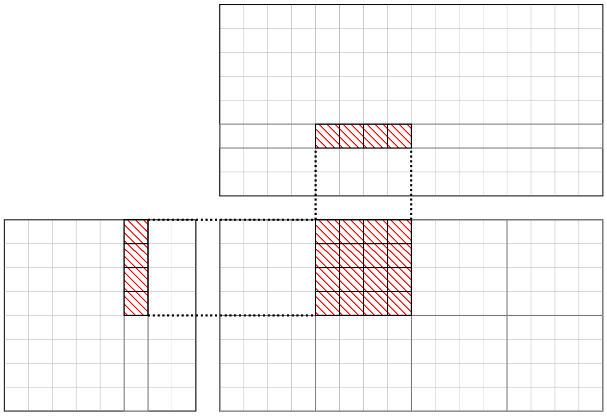


Figure 4.5: Example of the computation on a 64×64 sub-block done by a single warp. Each cell is a 16×16 matrix fragment. The full 128×256 block is processed by 8 warps in parallel.

matrix sub-block (see Figure 4.5). 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 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.

No 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; Fit+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

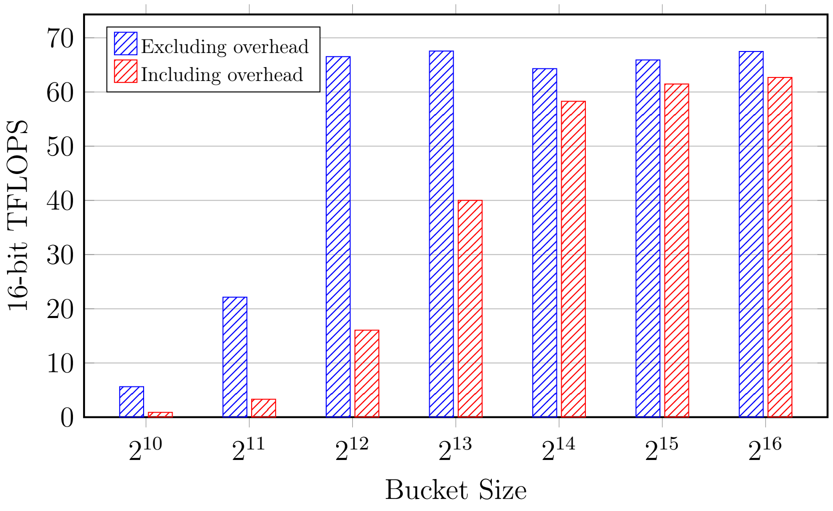


Figure 4.6: Efficiency of the reduction GPU kernel for different bucket sizes on a RTX 2080 Ti, only counting the $2n$ FLOPS per inner product. The overhead includes obtaining the vectors from the database, sending them to the GPU, conversions, recomputing length at higher precision, and retrieving the results from the GPU in a pipelined manner.

false negatives of the SimHash the database size needs to be a larger, which hinders our focus on minimizing RAM usage.

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 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 4.6. 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.

4.4.2 Tensor cores and 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 precision.

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 is acceptable for our purposes if say 1% of the close vectors are missed because of even larger errors. We consider the case that the vectors are represented using the `fp16` format. In addition we also assume that the inner product is accumulated in 16-bit precision[8].

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). Computing the inner product of two vectors using Tensor cores leads to errors in two places, first there

[8]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.

is some *representation error* when representing the vectors in 16-bit precision, and secondly some *computation error* accumulates during the inner product computation.

Representation error

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

$$\|\mathbf{y} - \hat{\mathbf{y}}\| \leq \max\{2^{-11}, 2^{-25} \cdot \sqrt{n}\}. \tag{4.1}$$

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}.

Computation error

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 [Bla+20], 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

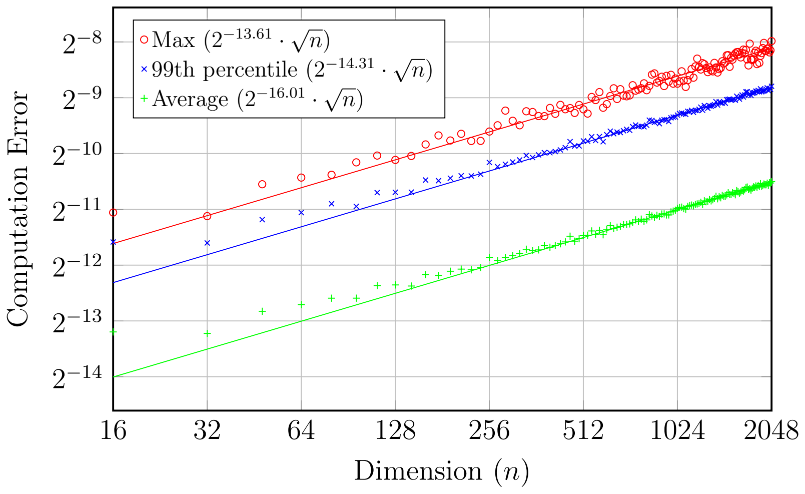


Figure 4.7: Computation error $|S - \hat{S}|$ observed in dimension n over 16384 sampled pairs of unit vectors $\mathbf{y}, \mathbf{y}'$ that satisfy $S := \langle \mathbf{y}, \mathbf{y}' \rangle \approx 0.5$.

models. We see in Figure 4.7 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}.

4.5 A dual hash for BDD

Let us recall the principle of the 'dimensions for free' trick [Duc18]; by lifting many short vectors from the sieving context $\mathcal{L}_{[l:r)}$ of dimension $n := r - l$ we can recover a short(est) vector in some larger context $\mathcal{L}_{[l-k:r)}$ for $k > 0$. The sieving implementation G6K [Alb+19] puts extra emphasis on this by lifting many vectors it encounters while reducing a bucket, even when these reduced vectors are not short enough to be added to the database. Recall that for lifting a vector we have to solve an (approx)-CVP instance w.r.t. the *lifting context* $\mathcal{L}_{[l-k:l)}$. By applying the appropriate isometry we identify $\mathcal{L}_{[l-k:l)}$ in this section with a full rank lattice in $\mathbb{R}^k$ (this coincides with the $\mathbf{y}$-

representation that G6K already stores). Lifting using Babai's nearest plane algorithm has a significant cost of $\Theta(n \cdot k + k^2)$ per vector, and thus it would be inefficient to lift all reduction pairs encountered. Therefore the G6K implementation first filters on their length in the *sieving context*, which is already being computed, and only lifts them when they are short enough. The $O(n \cdot k)$ part of the cost, to compute the corresponding target $\mathbf{t}_i - \mathbf{t}_j \in \mathbb{R}^k$, can be amortized to $O(k)$ over all pairs by pre-computing $\mathbf{t}_1, \ldots, \mathbf{t}_s$, leaving the cost of $\Theta(k^2)$ for the Babai nearest plane algorithm w.r.t. $\mathbf{B}_{[l-k:l)}$. In theory by pruning and early aborting the lifting process the cost of $\Theta(k^2)$ could be reduced somewhat more.

To minimize the overhead from the lifting process we want it to be less costly than the reduction part that has a cost of about $O(n)$ per pair. Thus the filter should have an acceptance rate of at most $O(n/k^2)$. A squared length filter of 1.8 on the sieving length, leads to an exponentially small acceptance rate in the sieving dimension $n = r - l$. However assuming BGJ1 bucketing and input vectors of squared length $\frac{4}{3}$ the acceptance rate would be about $0.9754^{n+o(n)}$; giving a concrete acceptance rate of order 10^{-2} even in sieving dimensions as large as $n = 128$.

When using GPUs we have an additional problem that we cannot run the lifting computation inline in the reduction kernel, as this would diminish the overall performance by using more registers and shared memory for storing the targets, and because the remaining threads in a warp would also have to wait for this. Therefore one would need to return all the pairs of indices that pass the filter and store them temporarily in global memory. With an acceptance rate in the order of 10^{-2} this would imply tens of millions of results for the usual bucket sizes, taking more than ten times the storage of the input vectors. This would become a bottleneck in practice, even ignoring the reduction kernel overhead of saving those results in the first place.

As an alternative we introduce 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 $\mathrm{gh}(\mathcal{L}_{[l-k:l)}) \gg \mathrm{gh}(\mathcal{L}_{[l-k:r)})$. For the few lifts that we are actually interested in we expect an extra length of only $\delta \cdot \mathrm{gh}(\mathcal{L}_{[l-k:l)})$, 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 $\mathcal{L}_{[l-k:l)}$, also known as δ-BDD instances, and their share decreases exponentially as δ^k, assuming the targets are uniform over $\mathrm{span}(\mathcal{L}_{\mathrm{bdd}})/\mathcal{L}_{\mathrm{bdd}}$, where $\mathcal{L}_{\mathrm{bdd}} := \mathcal{L}_{[l-k:l)}$. E.g., for reasonable parameters $\delta = 0.5$ and $k = 24$ this amounts to only $6 \cdot 10^{-8}$ of all pairs.

So we want to filter δ-BDD instances over the k-dimensional lattice $\mathcal{L}_{\mathrm{bdd}}$. More abstractly we need a filter that quickly checks if pairs are (exceptionally) close over the *torus* $\mathrm{span}(\mathcal{L}_{\mathrm{bdd}})/\mathcal{L}_{\mathrm{bdd}} \cong \mathbb{R}^k/\mathcal{L}_{\mathrm{bdd}}$. 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 $\mathbb{R}^h/\mathbb{Z}^h \cong \left[-\frac{1}{2}, \frac{1}{2}\right)^h$. 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 k 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 $\mathcal{L}_{\mathrm{bdd}}$ is defined as $\mathcal{L}_{\mathrm{bdd}}^* := \{\mathbf{w} \in \mathrm{span}(\mathcal{L}_{\mathrm{bdd}}) : \langle \mathbf{w}, \mathbf{v} \rangle \in \mathbb{Z} \text{ for all } \mathbf{v} \in \mathcal{L}_{\mathrm{bdd}}\}$.

Definition 71 (Dual hash). *For a full rank lattice $\mathcal{L} \subset \mathbb{R}^k$, dual hash length $h \geq k$ 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*

$$\mathcal{H}_{\mathbf{D}} : \mathbb{R}^k/\mathcal{L} \to \mathbb{R}^h/\mathbb{Z}^h \cong \left[-\frac{1}{2}, \frac{1}{2}\right)^h,$$
$$\mathbf{t} \mapsto \mathbf{D}\mathbf{t}.$$

The dual hash relates distances in $\mathbb{R}^k/\mathcal{L}$ to those in $\mathbb{R}^h/\mathbb{Z}^h$. Note that the distance $\mathrm{dist}(\mathbf{t}', \mathbb{Z}^h)$ is well defined for any $\mathbf{t}' \in \mathbb{R}^h/\mathbb{Z}^h$.

Lemma 72. *Let $\mathcal{L} \subset \mathbb{R}^k$ be a full rank lattice with some dual hash $\mathcal{H}_{\mathbf{D}}$ of length h. Then for any $\mathbf{t} \in \mathbb{R}^k$ we have*

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

where $\sigma_1(\mathbf{D})$ denotes the largest singular value of $\mathbf{D} \in \mathbb{R}^{h \times k}$.

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 $\mathcal{H}_{\mathbf{D}}(\mathbf{t} - \mathbf{x}) \equiv \mathcal{H}_{\mathbf{D}}(\mathbf{t}) \bmod \mathbb{Z}^h$. 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 73 (Filter). *Let $\mathcal{L} \subset \mathbb{R}^k$ be a full rank lattice with some dual hash $\mathcal{H}_{\mathbf{D}}$. For a hash bound H we define the filter function*

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

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.

4.5.1 An average-case analysis of the dual hash

Lemma 72 allows us to choose a dual hash bound H such that our filter $\mathcal{F}_{D,H}$ returns 1 for all δ-BDD instances. Such a worst-case bound is far from practical behaviour, thereby giving a much looser filter than needed. In this section we introduce an average-case analysis, both for which bound to pick and for the positive rate of our filter. We assume that the targets are uniformly distributed, e.g., uniform over $\mathbb{R}^k/\mathcal{L}_{\text{bdd}}$, more specifically unless otherwise stated we assume without loss of generality (given that the dual hash filter only depends on the coset and not the particular representative) that the targets are uniform over the Voronoi cell $\mathcal{V}(\mathcal{L}_{\text{bdd}}) \subset \mathbb{R}^k$.

The dual hash bound

We discuss what bound H to pick for the dual hash filter $\mathcal{F}_{\mathbf{D},H}$, such that we detect those targets $\mathbf{t}$ that lie at some close distance

$\mathrm{dist}(\mathcal{L}_{\mathrm{bdd}}, \mathbf{t}) \leq R := \delta \cdot \lambda_1(\mathcal{L}_{\mathrm{bdd}})$ of the lattice. We consider the unique decoding regime, i.e., when $\delta < \frac{1}{2}$. In this regime we can assume without loss of generality that the normalized targets $\mathbf{t}/\|\mathbf{t}\|$ are uniformly distributed over the sphere. Suppose that $\mathbf{D}^\top \mathbf{D}$ has eigenvalues $\sigma_1^2, \ldots, \sigma_k^2$ with corresponding normalized (orthogonal) eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$. By a change of basis 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 we see

$$\begin{aligned}\mathbb{E}[\mathrm{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}), \mathbb{Z}^h)^2] \leq \mathbb{E}[\|\mathbf{D}\mathbf{t}\|^2] &= \mathbb{E}\left[\sum_{i=1}^k t_i^2 \cdot \sigma_i^2\right] \\ &= \sum_{i=1}^k \sigma_i^2 \cdot \mathbb{E}[t_i^2] = \|\mathbf{t}\|^2 \cdot \frac{1}{k}\sum_{i=1}^k \sigma_i^2\end{aligned}$$

So instead of the worst case bounds from Lemma 72, the average-case expected distance $\mathrm{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}), \mathbb{Z}^h)$ is bounded by $\sqrt{\frac{1}{k}\sum_{i=1}^k \sigma_i^2} \cdot \|\mathbf{t}\|$. Note that if the latter bound is relatively small then we can also expect that all coordinates of $\mathbf{D}\mathbf{t}$ lie in $[-\frac{1}{2}, \frac{1}{2}]$, and thus the first inequality is tight. If we explicitly assume that $\mathrm{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}), \mathbb{Z}^h) = \|\mathbf{D}\mathbf{t}\|$ in this regime we can also compute the variation which equals

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

We see that when all eigenvalues $\sigma_1^2, \ldots, \sigma_k^2$ are equal the variance is 0, and more generally the more balanced they are the better. For the rest of the section we assume that the dual hash bound for targets at distance R is set to $R \cdot \sqrt{\frac{1}{k}\sum_{i=1}^k \sigma_i^2}$, accepting that we only detect say half of such targets. Note that the bound is based on targets at distance exactly R, the targets that lie closer are even more probable to pass the filter. By adding some multiple of the variance to this bound one could obtain more quantative values for the false negative rate using Chebyshev's inequality. For a random lattice most targets are still uniquely decodable even for δ somewhat larger than $\frac{1}{2}$, and thus we can still use the above bound as a good estimate.

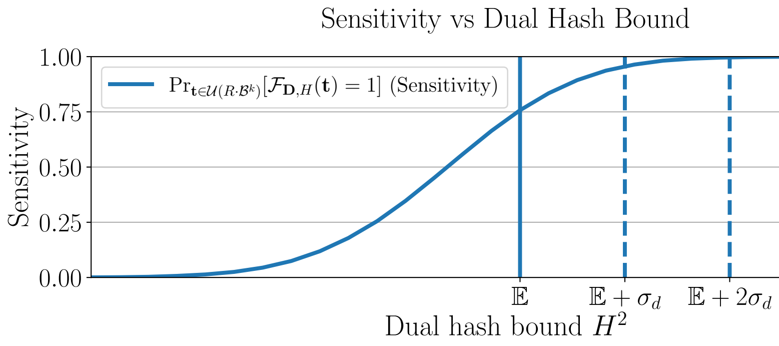


Figure 4.8: Experimental results of the true positive rate of the dual hash filter $\mathcal{F}_{\mathbf{D},H}$ in $\mathcal{L}_{\mathrm{bdd}}$ of dimension $k=24$ with $h=64$ dual vectors. The 2^{17} targets were sampled uniformly in a ball of radius $R := 0.5\cdot\mathrm{gh}(\mathcal{L}_{\mathrm{bdd}})$. With a bound of $H^2=\mathbb{E} := R\cdot\frac{1}{k}\operatorname{Tr}(\mathbf{D}^\top\mathbf{D})$ the true positive rate is 72.8%, and at 1 and 2 standard deviations higher it is respectively 94.8% and 99.5%.

Positive rate

We try to understand the positive rate p_{pos} of the filter, e.g., the ratio of targets $\mathbf{t}\sim\mathbb{R}^k/\mathcal{L}_{\mathrm{bdd}}$ for which $\mathcal{F}_{\mathbf{D},H}(\mathbf{t})=1$. It seems hard to determine the exact positive rate, as multiple factors play are role. Still we can get some good estimate by splitting the analysis in two cases: the *preserved* and the *unpreserved* regime. Consider a target $\mathbf{t}\in\mathbb{R}^k$ and let $\mathbf{x}$ be a closest vector in $\mathcal{L}_{\mathrm{bdd}}$ 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 $\|\mathbf{D}(\mathbf{t}-\mathbf{x})\|_2=\mathrm{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}),\mathbb{Z}^h)$. The unpreserved regime considers those targets for which there is no equality, i.e., $\|\mathbf{D}(\mathbf{t}-\mathbf{x})\|_2 > \mathrm{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}),\mathbb{Z}^h)$.

Preserved Regime. We assumed without loss of generality that the targets $\mathbf{t}$ are uniform over the Voronoi cell $\mathcal{V}(\mathcal{L}_{\mathrm{bdd}})$, such that their closest vector is $\mathbf{0}$. In the preserved regime we have the equality $\mathrm{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}),\mathbb{Z}^h)=\|\mathbf{Dt}\|_2$, and thus $\mathcal{F}_{\mathbf{D},H}(\mathbf{t})=1$ if and only if $\|\mathbf{Dt}\|_2\leq$

H. The positive rate p_{pres} is then given by

$$p_{\text{pres}} = \Pr_{\mathbf{t}\sim\mathcal{U}(\mathcal{V}(\mathcal{L}_{\text{bdd}}))}\left[\|\mathbf{Dt}\|_2 \leq H\right].$$

We proceed in the same way as earlier by picking applying a basis transformation using the normalized eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ of $\mathbf{D}^\top\mathbf{D}$. Let $\mathbf{V} := [\mathbf{v}_1; \ldots; \mathbf{v}_k]$, then we have

$$p_{\text{pres}} = \Pr_{\mathbf{t}\sim V\cdot\mathcal{U}(\mathcal{V}(\mathcal{L}_{\text{bdd}}))}\left[\sum_{i=1}^{k} t_i^2 \cdot \sigma_i^2 \leq H^2\right] = \frac{\operatorname{vol}(V \cdot \mathcal{V}(\mathcal{L}_{\text{bdd}}) \cap \mathcal{E}_{H,\sigma_1^2,\ldots,\sigma_k^2})}{\operatorname{vol}(V \cdot \mathcal{V}(\mathcal{L}_{\text{bdd}}))},$$

where $\mathcal{E}_{H,\sigma_1^2,\ldots,\sigma_k^2}$ is the ellipsoid defined by $\{\mathbf{t} \in \mathbb{R}^k : \sum_{i=1}^{k} t_i^2 \cdot \sigma_i^2 \leq H^2\}$. We can simply bound the volume in the numerator by that of the ellipsoid, and note that the volume of the denominator equals the determinant $\det(\mathcal{L}_{\text{bdd}})$. So we conclude that

$$p_{\text{pres}} \leq \frac{\operatorname{vol}(\mathcal{E}_{H,\sigma_1^2,\ldots,\sigma_k^2})}{\operatorname{vol}(\mathcal{V}(\mathcal{L}_{\text{bdd}}))} = \frac{\operatorname{vol}(\mathcal{B}^k) \cdot H^k}{\det(\mathcal{L}_{\text{bdd}}) \cdot \prod_{i=1}^{k} \sigma_i}.$$

Note that the inequality is only provably tight when $H/\sigma_k \leq \frac{1}{2}\lambda_1(\mathcal{L}_{\text{bdd}})$, where σ_k^2 is the smallest eigenvalue, but again for random lattices we can expect the estimate to also be reasonably close also for larger bounds.

Unpreserved Regime. In the unpreserved regime $\operatorname{dist}(\mathcal{H}_{\mathbf{D}}(\mathbf{t}), \mathbb{Z}^h)$ is not really a useful metric, as there will seemingly be no clear relation with $\|\mathbf{D}(\mathbf{t} - \mathbf{x})\|_2$. Inspired by practical observations, we analyse these positives from the heuristic assumption in this regime that every $\mathbf{Dt}$ is uniformly distributed over $[-\frac{1}{2}, \frac{1}{2})^h$ modulo $\mathbb{Z}^h$. Then we can ask the question how probable it is that $\|\mathbf{Dt}\|_2 = \operatorname{dist}(\mathbf{Dt}, \mathbb{Z}^h) \leq H$; i.e., that the target passes the filter even though it is not close to the lattice. 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,

$$\begin{aligned} p_{\text{unpr}} &:= \Pr_{\mathbf{v}\in\mathcal{U}[-\frac{1}{2},\frac{1}{2})^h}\left[\mathbf{v} \in H \cdot \mathcal{B}^h\right] \\ &= \Pr_{\mathbf{v}\in\mathcal{U}(H\cdot\mathcal{B}^h)}\left[\mathbf{v} \in [-\tfrac{1}{2}, \tfrac{1}{2})^h\right] \cdot \operatorname{vol}(H \cdot \mathcal{B}^h) \leq H^h \cdot \operatorname{vol}\mathcal{B}^h. \end{aligned}$$

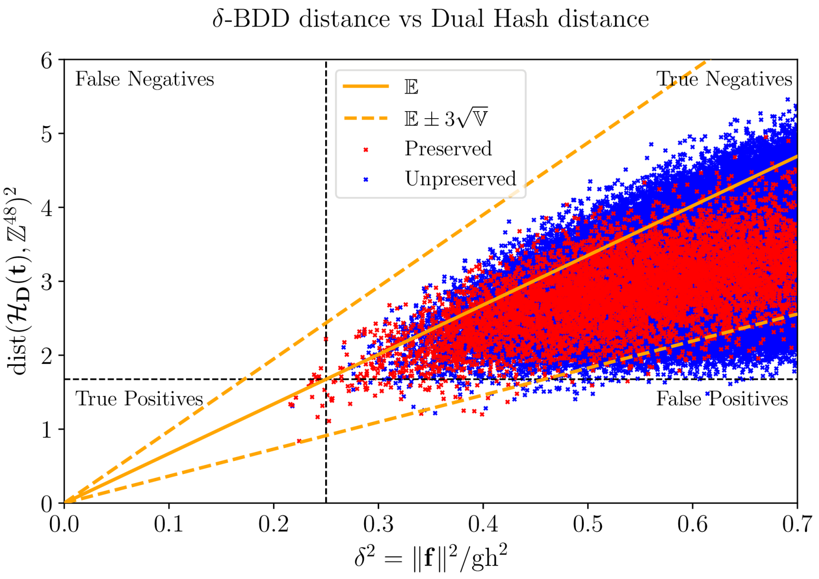


Figure 4.9: Dual hash filter correlation on the context $\mathcal{L}_{[14:30)}$ for a reduced 160-dimensional lattice using 48 dual vectors. The BDD-bound was computed with a representative squared length bound of 1.44 and the 2^{20} targets are uniformly sampled over the Voronoi cell around 0.

For a small bound H the probability $\Pr_{\mathbf{v}\in\mathcal{U}(H\cdot\mathcal{B}^h)}\left[\mathbf{v} \in [-\frac{1}{2}, \frac{1}{2})^h\right]$ is close to 1 and thus the bound is rather tight. For somewhat larger values of H we can estimate $\Pr_{\mathbf{v}\in\mathcal{U}(H\cdot\mathcal{B}^h)}\left[\mathbf{v} \in [-\frac{1}{2}, \frac{1}{2})^h\right]$ by heuristically assuming independence between the coordinates. Recall that if $\mathbf{v} \sim \mathcal{U}(\mathcal{B}^h)$, then each coefficient $\mathbf{v}_i^2$ follows a $\mathrm{Beta}(\frac{1}{2}, \frac{h+1}{2})$ distribution with CDF $I_x(\frac{1}{2}, \frac{h+1}{2})$, leading to the heuristic estimate

$$p_{\mathrm{unpr}} \approx I_{\frac{1}{4H^2}}(\tfrac{1}{2}, \tfrac{h+1}{2})^h \cdot H^h \cdot \mathrm{vol}\,\mathcal{B}^h.$$

Alternatively one can use Monte-Carlo sampling to estimate p_{unpr}. The number of samples (under a fixed error variance) is inversely proportional to the event probability, and thus directly estimating p_{unpr} for low values of H is too costly. Luckily for small values of H the probability $\Pr_{\mathbf{v}\in\mathcal{U}(H\cdot\mathcal{B}^h)}\left[\mathbf{v} \in [-\frac{1}{2}, \frac{1}{2})^h\right]$ is large and thus we can estimate that probability instead, leading to an efficient Monte-Carlo estimate both for small and large values of H. To determine in which regime we

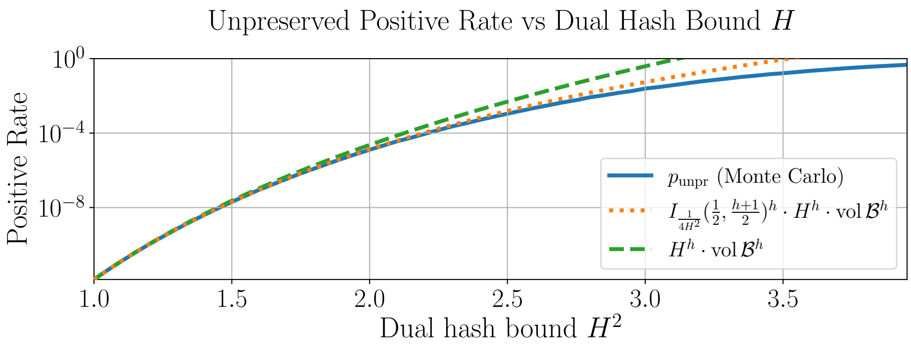
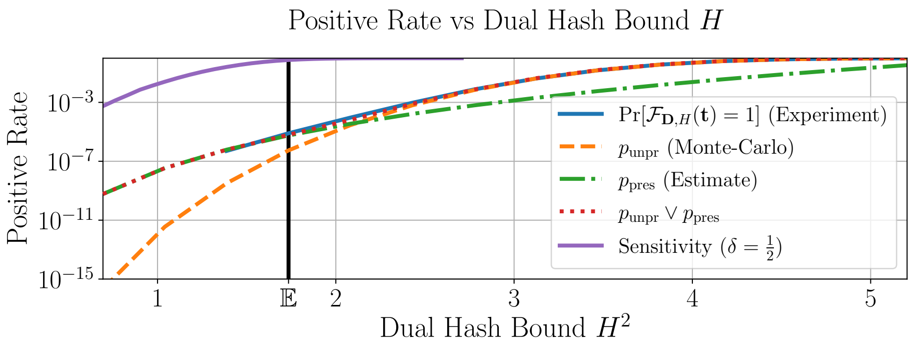


Figure 4.10: The unpreserved positive rate p_{unpr} for $h = 48$.



are one could use the heuristic estimate for $\Pr_{\mathbf{v} \in \mathcal{U}(H \cdot \mathcal{B}^h)} \left[\mathbf{v} \in [-\frac{1}{2}, \frac{1}{2})^h\right]$ presented earlier. Note that p_{unpr} only depends on the filter threshold H and the number of dual vectors h and not on the specific matrix $\mathbf{D}$. We could thus precompute p_{unpr} for relevant parameters instead of computing it on the fly.

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 sensitivity. We fix the target distance to some $0 < R \leq \frac{1}{2}\lambda_1(\mathcal{L}_{\text{bdd}})$, and as the computational cost mostly depends on the number of dual vectors h we will try to optimize $\mathbf{D}$ for a fixed number of dual vectors h.

To simplify the overall optimization we set the dual hash bound to $H := R \cdot \sqrt{\frac{1}{k}\sum_{i=1}^{k}\sigma_i^2}$, such that the false negative rate is expected to be reasonable for uniform targets at distance at most R. We can now focus on minimizing the positive rate.

In the preserved regime the (bound on the) positive rate is proportional to $H^k / \prod_{i=1}^{k}\sigma_i$. Note that $\mathrm{Tr}(\mathbf{D}^\top\mathbf{D}) = \sum_{i=1}^{k}\sigma_i^2$ and $\det(\mathbf{D}^\top\mathbf{D}) = \prod_{i=1}^{k}\sigma_i^2$, the positive rate in the preserved regime is thus propertional to

$$p_{\mathrm{pres}} \sim \frac{(\frac{1}{k}\mathrm{Tr}(\mathbf{D}^\top\mathbf{D}))^{k/2}}{\sqrt{\det(\mathbf{D}^\top\mathbf{D})}},$$

which is a well known conditioning metric for matrices. To optimize this one can think of picking h dual vectors that are of about the same length and somewhat orthogonal. For the unpreserved regime the positive rate is mostly proportional to H^k, which means we want $\mathrm{Tr}(\mathbf{D}^\top\mathbf{D})^{k/2}$ to be small; 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 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.

From experiments we can conclude that this greedy method to improve the filter works really well. For example with the parameters as in Figure 4.9, 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 in somewhat large dimensions and easily won back from allowing a lower number of dual vectors h.

4.5.2 Application and results

We shortly discuss how one would use the introduced dual hash techniques in practice for solving SVP challenges more efficiently. Recall that for such challenges we are given a lattice $\mathcal{L}$ of dimension d and we try to find a short vector of norm at most $1.05 \cdot \mathrm{gh}\,\mathcal{L}$. We do this by sieving in some smaller context $\mathcal{L}_{[l:d)}$ (for progressively decreasing l), and by lifting vectors from this context to the full lattice $\mathcal{L}_{[0:d)} = \mathcal{L}$, hoping to encounter a short enough vector. The goal of the dual hash is to efficiently filter pairs of vectors which difference lies close to the lattice $\mathcal{L}_{[0:l)}$, thereby increasing the number of dimensions for free l.

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 $\mathcal{L}_{[0:l)}$ might fail to return short vectors for positions $l' > 0$, which are also needed to improve the quality of the basis. Therefore we apply the dual hash to the smaller *filter context* $\mathcal{L}_{\mathrm{bdd}} := \mathcal{L}_{[f:l)}$. If a vector is short in the context $\mathcal{L}_{[l':r)}$ 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 $\mathcal{L}_{[l:r)}$ of length R. We can assume that $R \geq \ell$, where ℓ is the sieving length bound, 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 $\mathcal{L}_{[l':r)}$ for $\kappa \leq l' \leq f$. This corresponds to a target $\mathbf{t}$ at distance at most

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

from the lattice $\mathcal{L}_{[l':l)}$. 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

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

allows a significant part of the short lifts in the context $\mathcal{L}_{[l':r)}$ 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 72 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 4.9 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.

4.5.3 Implementation details

Given a list of pre-computed $\mathbf{Dt}_1, \ldots, \mathbf{Dt}_m$ we want to use the GPU to efficiently compute $\operatorname{dist}(\mathbf{D}(\mathbf{t}_i - \mathbf{t}_j), \mathbb{Z}^h)$ 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.

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

Dual hash 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 difference 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 $\mathbb{Z}^4$.

Dual hash 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 could use the fact that $\text{dist}(x, \mathbb{Z})^2 \approx (1 - \cos(x \cdot 2\pi))/16$ 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 $\mathbf{D}\mathbf{t}_i$ and $\mathbf{D}\mathbf{t}_j$ respectively. Rewriting using trigonometric identities we get:

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

If we pre-compute that values $(\cos((\mathbf{D}\mathbf{t}_i)_l \cdot 2\pi))_l$ and $(\sin((\mathbf{D}\mathbf{t}_i)_l \cdot 2\pi))_l$ 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. In the end we did not implement this adaptation as the pairwise dual hash computations were not a bottleneck, and thus improving it would only result in a minor overall speed-up.

4.6 New records

4.6.1 Comparison

We compare several of our sieve implementations experimentally. Although our BDGL-like implementations `bdgl` and `bdgl_gpu` will eventually be faster than the BGJ-like implementations `triple` by G6K[9] 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 4.11 we display the wall-clock time taken for each Sieve operation during the pump up. All our GPU implementations use a multi-bucket parameter of 4, which should give a balanced comparison based on Figure 4.4. Any on-the-fly lifting or dual hash techniques are disabled. For the remaining parameters we refer to the next Section 4.6.2.

CPU sieves

We observe in Figure 4.11 that our BDGL implementations `2-bdgl` and `3-bdgl` are extremely practical. They both improve upon the record-holding `triple` sieve from dimension ±85 and onwards. The speed-up of `2-bdgl` and `3-bdgl` over `triple` increases to a factor of 5.3× and 3.1× respectively in dimension 120[10]. We also see that while

[9]The `triple` sieve in G6K is in the meantime renamed to `hk3` after [HK17].

[10]In the original work we reported a speed-up of 2.7× for `3-bdgl`. By optimising the cache locality of the reduction phase this was improved to 3.1×. These optimisations were much more effective for the larger buckets of `2-bdgl`, resulting

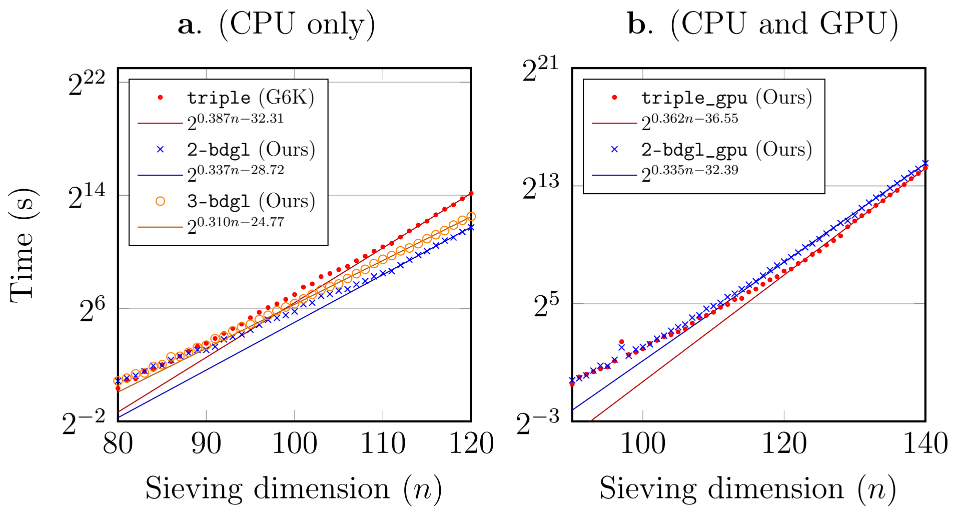


Figure 4.11: Comparison of different sieve implementations from [Alb+19] and this work. We ran a single pump up in a well-reduced 160-dimensional lattice to a sieving dimension of 120 and 140 for CPU only and GPU accelerated respectively. The timings give the amount of time spend in each sieving dimension. All experiments used a saturation of 37.5% with a database size of $2.77 \cdot 2^{0.2075n}$. All 40 CPU cores and 4 GPU's were used. The fitting is obtained by a linear least-squares regression on the last 10 dimensions in log-space.

`3-bdgl` is asymptotically faster than `2-bdgl`, this is not concretely the case below dimension 120. Given the extrapolations we expect the cross-over around dimension 146.

Given the large speed-up of the `2-bdgl` sieve over the record-holding `triple` sieve we also tried to resolve the 155-dimensional Darmstadt 1.05-approxSVP record challenge. We solved the challenge (with seed 1) with `2-bdgl` in 51.7 hours on 40 cpu cores (machine specifications as in Table 4.1), or about 2069 cpu hours, compared to a reported 1056 cpu *days* for the `triple` sieve. This is more than a factor 12× speed-up (ignoring cpu core differences). Part of the speed-up (about a factor 4×) can be explained by a few more dimensions for free (34 vs 28). It is unclear if we got lucky or if this increase can be attributed to the algorithm.

in the final 5.3× speed-up.

GPU sieves

Firstly, we observe that the GPU accelerated sieves are significantly faster than the CPU-only sieves. In dimension 120 `triple_gpu` is a tremendous factor 121× faster than `triple`. Secondly, we observe a different behaviour of the asymptotically superior sieve `2-bdgl_gpu` versus `triple_gpu`. While for the CPU-only version the cross-over between the two was already in dimensions around 85, for the GPU version the cross-over lies in a dimension greater than 140. Following the extrapolations, we only expect them to cross in dimension ± 154. The main reason for this is that the BDGL sieve obtains its speed-up from having many small buckets, but for the GPU implementation there is a large minimum bucket size to prevent memory bottlenecks in the reduction phase. The large minimum bucket size shifts the cross-over point by more than 70 dimensions. In this light, it did not appear pertinent to implement `3-bdgl_gpu`, which, while being asymptotically faster, would cross-over even later.

4.6.2 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.

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

Figure 4.12: Main parameters used for the record runs.

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 [Alb+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 SaturationRatio from .5 to .375.

Additionally, we introduced a database size limit by setting an experimentally-verified target $\mathsf{TD4F}(n) = \lfloor n/\log(n) \rfloor$ for the number of dimensions-for-free, 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 MultiBucket = 2. Our main parameters are summarized in Figure 4.12.

4.6.3 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

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

Dimensions				Norm	Cost		
dim	TD4F	D4F	MSD	Norm/GH	FLOP	Walltime	Mem GiB
158	31	29	129	1.04329	$2^{62.1}$	9h 16m	89
160	31	33	127	1.02302	$2^{61.8}$	8h 24m	88
162	31	31	131	1.04220	$2^{63.2}$	18h 32m	156
164	32	28	136	1.04368	$2^{64,8}$	2d 01h	179
166	32	30	136	1.03969	$2^{64.8}$	2d 01h	234
168	32	31	137	1.04946	$2^{65.3}$	2d 18h	318
170	33	31	139	1.04594	$2^{66.3}$	5d 11h	364
172	33	35	137	1.04582	$2^{65.0}$	2d 09h	364
174	33	35	139	1.04913	$2^{66.3}$	5d 06h	518
176	34	33	143	1.04412	$2^{67.5}$	12d 11h	806
178	34	32	146	1.02725	$2^{68.6}$	22d 18h	1060
180	34	30	150	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 4.1: Darmstadt Lattice 1.05-approxSVP Challenge results

the effort and results for each challenge are presented in Table 4.1.

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 [Alb+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× 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 Darm-

(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)

Figure 4.13: A solution to the Darmstadt Lattice 1.05-approxSVP Challenge in dimension 180 with seed 0.

		Wattage		Total usage	
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 4.2: Power use comparison for records of dimension 155 (G6K) and 176 (ours).

stadt Lattice 1.05-approxSVP Challenge dimension 180 with seed 0 in Figure 4.13.

4.6.4 Remarks

Power use

To compare power efficiency of our new record computation for dimension 176 with the previous record computation for dimension 155, we estimated the power use as shown in Table 4.2 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 4.2 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 the measured memory usage in Table 4.1, we estimate that our implementation requires about 416 Bytes per vector including the amortized overheads, 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.