Lattice Enumeration on GPUs for fplll

Simon Pohmann, Marc Stevens, Jens Zumbrägel

March 20, 2021

The Kannan-Fincke-Pohst lattice enumeration algorithm is the classical method for solving the shortest vector problem in lattices. It is also a fundamental tool for most lattice reduction algorithms that provide speed-length tradeoffs. As this algorithm allows efficient parallel implementations, it is likely that implementing it on modern graphics processing units (GPUs) can significantly improve performance. We provide such an implementation that is compatible with the fplll lattice reduction library [fplll16] and achieves a considerable speedup in higher lattice dimensions, compared to current, multithreaded versions. For this, we use the CUDA technology that provides an abstract language for programming GPUs.

Keywords Lattice Enumeration, Shortest Vector, fplll, Cryptanalysis

1 Introduction

A lattice is a discrete free $\mathbb{Z}$-submodule of the d-dimensional Euclidean space. Lattices are usually considered together with the norm from this space, which then gives rise to interesting computational problems. The most fundamental is the shortest vector problem (SVP), which is to find a shortest nonzero lattice vector given a lattice basis. This problem is often used as a basis for cryptography, as it is conjectured to be extremely hard, even in approximative versions. In particular, the security of many promising candidates for post-quantum cryptography can be reduced to SVP. Therefore, a good understanding of its theoretical and practical hardness is important.

To solve this problem exactly, there are two main approaches: The classical method is the Kannan-Fincke-Pohst lattice enumeration [Kan83; FP85], which performs an exhaustive search of all lattice points of a bounded norm. In practice, this algorithm performs very well, also because many improvements have been introduced. However, the time complexity is super-exponential and therefore not asymptotically optimal, while the space complexity is only polynomial. The other approach is lattice sieving [AKS01], which yields exponential running times, at the cost of needing exponential space. A lot of work in this field has recently made this approach competitive. Nevertheless, lattice

enumeration is still widely used and for dimensions up to 70 the best known algorithm [Alb+19].

Due to the conjectured hardness of SVP, it suggests itself to use general-purpose GPU computing that has developed in the last decade. Graphics processing units (GPUs) are optimized for highly parallelized workloads and have higher computing power than CPUs for suitable algorithms. Therefore, they have been successfully used in various fields, like machine learning, physical simulations and optimization/search problems, also including cryptanalysis.

For example, an implementation of the sieving approach on GPUs has recently broken current SVP records [DSW21]. Because of the high memory requirements of sieving, this algorithm used 1.5 TB of system RAM on a GPU server. As current GPUs have at most up to 32 GB of on-card memory, this introduced the main bottleneck of system-GPU data transfers. On the other hand, for our implementation of the enumeration approach, using only on-card memory is sufficient. Therefore, it also runs on commodity hardware. Instead of the memory bottleneck, the enumeration algorithm leads to a lot of divergent program flow (i.e. different branching behaviour in concurrent execution threads) that is not well-suited for GPU architectures. Mitigating that problem is the main result of this work.

Contribution We provide an implementation of the lattice enumeration algorithm that is able to use the extensive parallelization capacities provided by GPUs and is able to achieve a speedup of up to 5 compared to a multithreaded state-of-the-art implementation [fplll16]. For this, we used the CUDA technology by Nvidia that provides a high-level language for programming Nvidia GPUs [Nic+08].

To design the algorithm, we focused on the view of lattice enumeration as a depth-first tree search, and designed an algorithm that is optimized for GPUs and the concrete structure of this tree. In general, tree algorithms are not optimal for GPU architectures, because they usually require irregular, non-local memory accesses and non-uniform branching. By using the properties of the enumeration tree however, we partly circumvent these problems and achieve better performance than in the case of DFS in general trees, as studied e.g. in [Jen+11].

2 Preliminaries

2.1 Lattices

A *lattice* is a discrete subgroup $L \subseteq \mathbb{R}^n$. It follows that each lattice is of the form $L = b_1\mathbb{Z} + ... + b_m\mathbb{Z} = \mathrm{rowsp}_{\mathbb{Z}}(B)$ for linearly independent b_i resp. a matrix $B \in \mathbb{R}^{m\times n}$ with rows b_i. To be consistent with the convention that the b_i are the rows of B (as opposed to columns), we also interpret lattice points as row vectors. These b_i are called *basis* of the lattice L, and are in general not unique. In particular, two bases (given as their matrices) B and B' generate the same lattice, if and only if $B' = UB$ for some unimodular $U \in \mathbb{Z}^{m\times m}$ holds. The number m is called the *rank* of the lattice. Usually, we will consider full-rank lattices, i.e. $m = n$.

For a given lattice L, there is a (non-unique) nonzero vector $v \in L$ of smallest Euclidean norm. This norm is denoted by $\lambda(L) := \min_{v \in L \setminus \{0\}} \|v\|$. Now consider the so-called shortest vector problem (SVP): Given a basis $B \in \mathbb{R}^{n \times n}$ that generates a full-rank lattice L, compute a point $x \in L \setminus \{0\}$ such that $\|x\| = \lambda(L)$. To solve this problem exactly, there are mainly two approaches: The lattice enumeration algorithm, as described in Section 3, and lattice sieve algorithms [AKS01; Alb+19].

2.2 Gram-Schmidt orthogonalization

Given a matrix $B \in \mathbb{R}^{n \times n}$ with rows $b_1, ..., b_n$, the Gram-Schmidt process yields the projections b_i^* of b_i onto $\langle b_1, ..., b_{i-1} \rangle^\perp$. Denoting by B^* the matrix whose rows are the b_i^*, this results in $B = \mu B^*$ where μ is a lower triangle matrix with a diagonal of 1s. Furthermore, for $i \neq j$ the vectors $b_i^* \perp b_j^*$ are perpendicular, so with $D = \mathrm{diag}(\|b_1^*\|, ..., \|b_n^*\|)$ we have the decomposition

$$B = \mu D B' \quad \text{where } B' = D^{-1} B^* \text{ is orthonormal}$$

The SVP problem is invariant under linear isometries, i.e. given a lattice L generated by a basis B and an orthonormal matrix S, we have $\lambda(LS) = \lambda(L)$ (here $LS = \{xS \mid x \in L\}$ is the lattice that results from applying S to each of the lattice points, so it is generated by BS). Additionally, a solution to SVP in LS can be easily transformed into a solution to SVP in L and vice versa. Hence, to solve SVP it suffices to consider the lattice generated by μD, as the B' is orthonormal. The fact that this is a lower triangle matrix will be useful to formulate the calculations in a very concise way.

2.3 The CUDA programming model

CUDA is a language extension of C++ that allows writing code for execution on GPUs [Nic+08]. The only difference between a CUDA application and a standard C++ one is that the former can define and call "kernels", which are similar to functions but are executed on the GPU. As GPUs are optimized for heavily parallel algorithms, starting a kernel usually means starting thousands of GPU threads, all executing the same code in the kernel. These threads are grouped into the following units:

Warp The smallest unit of threads; on all current architectures, a warp consists of 32 threads. In CUDA, a thread is just a logical concept, and the hardware works directly with warps. As a result, all threads in a warp classically share a program counter, so they execute the same instruction at same time.

There are two important consequences impacting the performance: If threads in the same warp take different paths during conditional code execution, these paths are executed sequentially, and all threads that did not take the current path are idle. Therefore, for high throughput, it is essential to avoid divergent code within a warp. The second point deals with memory access. If threads within a warp access sequential words in memory, all of them can be done by the memory controller in one step (called "coalesced" memory access). These coalesced memory accesses are crucial for avoiding huge memory latencies.

Block Up to 1024 threads can be grouped in a block. Within a block, it is possible to use barrier-style synchronization. Except in very recent versions of CUDA, it is impossible to synchronize threads between blocks (however, atomics are available). Apart from this, one can allocate so-called "shared memory" that can be accessed by all threads within a block. Shared memory is scarce (up to 100KB per block), but accesses are significantly faster than the RAM-like global memory.

Grid The grid is the logical collection of all blocks that are started for the current kernel.

3 Approach

3.1 Lattice Enumeration

In this section, we describe the Kannan-Fincke-Pohst enumeration algorithm [Kan83; FP85], as it is used in [fplll16].

Given an n-dimensional lattice L with basis $b_1, ..., b_n$, consider the projections π_k onto the space $\langle b_1, ..., b_k \rangle^\perp$. Then the idea of lattice enumeration is to begin with the origin $0 \in \pi_n L = \{0\}$ and repeatedly expand points of norm $\leq r$ in $\pi_{k+1} L$ to multiple points of norm $\leq r$ in $\pi_k L$.

For $p \in \pi_{k+1} L$ of norm $\|p\| \leq r$ we call all points $p' \in \pi_k L$ of norm $\|p'\| \leq r$ with $\pi_{k+1} p' = p$ the *children* of p. As the projections do not increase the norm of vectors, have that for $p' \in \pi_k L$ with norm $\|p'\| \leq r$, also $\pi_{k+1} p' \in \pi_{k+1} L$ is of norm $\leq r$. Therefore, considering the children of each point in $\pi_{k+1} L$ of norm $\leq r$ yield exactly all points in $\pi_k L$ of norm $\leq r$.

As the use of "children" already indicates, this defines a tree which we call the enumeration tree. This enumeration tree has root 0 and each maximal path has length $n + 1$. Additionally, the leaf nodes are exactly given by the lattice points of norm $\leq r$. The standard method is now to perform a depth-first search on this tree, and return the shortest nonzero leaf node that was encountered.

Therefore, the fundamental operation of the algorithm is the calculation of all children points. Usually, instead of storing the points $p \in \pi_k L$, the coefficients w.r.t. the projected basis $\pi_k b_i$ are stored. In this case, on tree level $n - k$, each point $p \in \pi_k L$ has a representation $p = \pi_k \sum_{i=k+1}^{n} x_i b_i,\ x_i \in \mathbb{Z}$, so the coefficients for $b_1, ..., b_k$ can be chosen to be zero. It follows that the children p' of a point $p = \pi_k \sum_{i=k+1}^{n} x_i b_i$ are characterized by their coefficient x_k, so they are of the form

$$p' = \pi_{k-1} \sum_{i=k}^{n} x_i b_i \quad \text{where} \quad \|b_k^*\| \left| x_k + \sum_{i=k+1}^{n} x_i \mu_{ki} \right| \leq \sqrt{r^2 - \|p\|^2},\ x_k \in \mathbb{Z},$$

which can be easily seen by calculating $\|p'\|$

$$\begin{aligned}
&\Big\|\pi_{k-1}\sum_{i=k}^{n} x_i b_i\Big\|^2 = \Big\|b_k^* \frac{1}{\langle b_k^*, b_k^*\rangle}\langle b_k^*, \sum_{i=k}^{n} x_i b_i\rangle + \pi_k \sum_{i=k}^{n} x_i b_i\Big\|^2 \\
&= \frac{\|b_k^*\|^2}{\langle b_k^*, b_k^*\rangle^2}\langle b_k^*, \sum_{i=k}^{n} x_i b_i\rangle^2 + \Big\|\pi_k \sum_{i=k}^{n} x_i b_i\Big\|^2 \\
&= \|b_k^*\|^2\Big(\sum_{i=k}^{n} x_i \mu_{ki}\Big)^2 + \Big\|\pi_k \sum_{i=k+1}^{n} x_i b_i\Big\|^2 = \|b_k^*\|^2\Big(x_k + \sum_{i=k+1}^{n} x_i \mu_{ki}\Big)^2 + \|p\|^2.
\end{aligned}$$

Iterating over all children is therefore equivalent to iterate over all integers x_k between

$$-\sum_{i=k+1}^{n} x_i\mu_{ki} - \frac{\sqrt{r^2 - \|p\|^2}}{\|b_k^*\|} \;\le\; x_k \;\le\; -\sum_{i=k+1}^{n} x_i\mu_{ki} + \frac{\sqrt{r^2 - \|p\|^2}}{\|b_k^*\|}$$

This directly shows that each node has at most $2r/\|b_k^*\|$ children, giving a running time of $\exp(O(n^2))$ on an LLL-reduced basis with $r = \|b_1\| \in O(2^{n/2})\|b_k^*\|$. A more thorough analysis shows that under the right reduction assumptions on the input basis, the enumeration algorithm has a running time of $2^{O(n \log n)}$ [Kan83; HS07].

The partial center sums

From this description we see that the major work is computing the sum $\sum_{i=k+1}^{n} x_i\mu_{ki}$, often called `center` as it is the center of the interval from which to choose x_k. By keeping track of all the partial sums $\sum_{i=k}^{n} x_i\mu_{li}$ for $l < k$, the center is always available, and updating the partial sums requires $n - k$ multiplications on tree level $n - k$.

Decrease enumeration bound

When finding any leaf node in the enumeration tree, this node corresponds to a lattice point $x \in L$. If $x \neq 0$, we know that there is a nonzero lattice point of norm $\le \|x\|$ in the lattice, so to find the shortest one, it suffices now to search only the points of norm $\le \|x\|$. In other words, we can potentially decrease the enumeration bound r by $r := \min\{r, \|x\|\}$ (there is some complication because of rounding errors during floating point arithmetic, see [PS08]), which can significantly reduce the size of the tree. Therefore, finding leaf nodes as early as possible is important for a fast algorithm. The resulting routine is shown in Algorithm 1.

3.2 Parallelization of the Lattice Enumeration

The main difficulty of implementing the enumeration algorithm efficiently on GPUs is the fact that nodes in the enumeration tree have greatly varying degree, so subtrees may have completely different size and structure. This introduces a lot of branching and makes it hard to evenly distribute work on the threads.

Algorithm 1 Find tree node children

Input: parent coefficients $x_{k+1}, ..., x_n$, parent norm $\|p\|^2$, partial center sums $\sum_i x_i \mu_{li}$ for $l < k+1$, matrix $(\mu_{ij})_{ij}$

Output: coefficients $x_k^{(i)}, ..., x_n^{(i)}$ and norms of the children $\pi_{k-1} \sum_n x_n^{(i)} b_n$ of $\pi_k \sum_n x_n b_n$

Set center $:= \sum_{i=k+1}^{n} x_i \mu_{kj}$
Set $x_0 = \lfloor \text{center} \rceil$
Set $\delta = 1$ if center $\geq x_0$, otherwise $\delta = -1$
for all $x \in \{x_0, x_0 + \delta, x_0 - \delta, x_0 + 2\delta, ...\}$ **do**
  If $(x - \text{center})^2 \|b_k^*\|^2 + \|p\|^2 > r^2$, exit
  Otherwise, yield the point p' with coefficients $(x, x_{k+1}, ..., x_n)$ and norm $\|p'\|^2 = \|p\|^2 + (x - \text{center})^2 \|b_k^*\|^2$
end for

These problems especially occur in the following "naive" approach: Enumerate all points on a certain tree level on the host, and then assign each GPU thread one of these points and let them enumerate the corresponding subtree. Nevertheless, this is still the main idea of our approach. However, we try to counter the problems by assigning a subtree not to a thread, but to a warp and using a work-stealing queue to distribute work among warps.

3.3 Subtree enumeration within a warp

The main idea for the thread cooperation within a warp is to let every thread expand the children of an assigned node, but not recurse into the corresponding subtrees. Instead, all the created new children nodes are then written to memory and are assigned to potentially different threads in the next step.

This prevents threads whose subtrees have different size to diverge and having to wait for the longer one. Additionally, having a list of nodes whose subtrees must still be searched also allows us to pick nodes that will be processed in the next step. This way, we ensure that all threads always work on nodes on the same tree level, which allows coalesced memory access, given a correct memory layout of the data.

The caveat of this approach is of course that it requires frequent memory accesses to load/store the tree nodes. The latency introduced by this is the main factor limiting performance. To at least reduce it, we apply the node shuffling not at each tree level, but only at every k-th tree level, for a constant k (in experiments, $k = 3$ has yielded the best results). In some more detail, this is described in Algorithm 2. For finding the coefficients of the points children$^k(\{x_i\})$ in the algorithm, an adaption of the efficient (but branching) recursive enumeration procedure from fplll is used. It also uses the previously calculated partial center sum values from Section 3.1.

Algorithm 2 Basic intra-warp enumeration
Input: subtree root R (with data required for Alg. 1), matrix $(\mu_{ij})_{ij}$
Output: Coeffients $x_1, ..., x_n$ and norm $\| \sum_n x_n b_n \|$ of shortest nonzero leaf vector in the subtree spanned by R

```
Init buffer with single node R on level 0
while node buffer is not empty do
  l := deepest tree level for which there are ≥ 32 nodes
  If such a l does not exist, use highest level with ≠ 0 nodes
  Assign one node x_i on level l to each thread i ∈ {0, ..., 31}
  if l is leaf level of enumeration tree then
    Thread i computes children^k({x_i}) using Alg. 1 recursively
    If one of these is ≠ 0 and shorter than the current optimal solution, update it
  else
    Thread i computes children^k({x_i}) using Alg. 1 recursively
    Store their coefficients and norms
    Calculate new partial center sums with a parallelized matrix-matrix multiplication
  end if
end while
```

3.4 Parameters

To balance the cost induced by memory accesses, the percentage of cases in which there are not enough tasks for all threads and the time threads of the same warp have to wait for each other, we have introduced algorithm parameters that control the thresholds used in the algorithm:

$k =$ `dimensions_per_level` As described above, this is the amount of enumeration tree levels that are expanded using Algorithm 1, before resulting nodes are written into the buffer. The name `dimensions_per_level` refers not to the levels of the enumeration tree, but to the levels of the "compacted" enumeration tree, in which each level corresponds to k levels of the original tree. From this perspective, the children$^k(\{x_i\})$ are the direct children of x_i in the compacted tree.

$T =$ `max_subtree_paths` During the calculation of children$^k(\{x_i\})$ using Algorithm 1, if more than T leaves resp. root-leaf-paths of the induced small subtree are found, the thread stops, and processing on the node will continue later. This can prevent threads from waiting for the computation on a very big subtree, but different subtree structures can still cause suspended threads.

$q =$ `min_active_parents_percentage` This parameter is used to determine on which level nodes should be processed in the current step. Using the deepest level with ≥ 32 nodes as in Algorithm 2 does not work, as we store a reference to its parent for each node. Therefore, a node cannot be deleted before all its children are, otherwise, the parent reference would be invalid.

Algorithm 3 Improved intra-warp enumeration
Input: list of subtree roots, whose subtrees are to be searched, matrix $(\mu_{ij})_{ij}$
Output: Coeffients $x_1, ..., x_n$ and norm $\|\sum_n x_n b_n\|$ of shortest nonzero leaf vector in any of the subtrees spanned by the given roots

---

Start $\lceil N_2/32 \rceil$ warps, with 32 threads each, that each execute the following:
Atomically (w.r.t. all warps), get next N_1 root nodes $R_1, ..., R_{N_1}$ from the input list
if No root nodes are left in the list **then**
  Terminate the current warp; if all warps are done, the algorithm is finished
end if
Init the (shared by the threads in a warp) node buffer with $R_1, ..., R_{N_1}$ on level 0
Set the current level $l := 0$
while node buffer is not empty **do**
  while the last 32 nodes in the buffer on level l contain more than $32 \cdot q$ unfinished ones **and** new children points fit into the buffer **do**
    Assign one node x_i on level l to each thread $i \in \{0, ..., 31\}$
    Thread i computes children$^k(\{x_i\})$ using Alg. 1 recursively (possibly resume an old computation); if their number exceeds T, stop and update the buffer, so that Alg. 1 can be resumed later
    if l is the leaf level of the enumeration tree **then**
      If one of these is $\neq 0$ and shorter than the current optimal solution, update it
    else
      Store their coefficients and norms
      Calculate new partial center sums with a matrix-matrix multiplication
    end if
  end while
  if children points have been added to the buffer during the above loop **then**
    Process the newly generated children points in the next step, i.e. set $l := l + 1$
  else
    Delete finished nodes among the last 32 ones on level l from the buffer
    Note that here, there are no children that might refer to the deleted nodes
    if the buffer contains no nodes on level l **then**
      Go one level up, i.e. $l := l - 1$
    end if
  end if
end while
Goto the beginning

Therefore, in each step we process the deepest level with enough nodes. If now after processing a level, the fraction of nodes that are not finished falls below q, we completely process all nodes in the buffer below the current level, and then delete the finished nodes.

N_1 = `initial_nodes_per_group` This is the number of subtree roots the buffer is initialized with in the first step; as opposed to Algorithm 2, this may be greater than 1.

N_2 = `thread_count` The total number of CUDA threads that will be started.

Including these additional ideas yields Algorithm 3.
In experiments, the following set of parameters has yielded the best results:

k	T	q	N_1	N_2
3	50	0.5	8	$32 \cdot 256$

4 Performance

The following benchmark was done on a machine with an Intel core i7-7700K CPU and a GeForce GTX 1080 Ti GPU. As a comparison for the CUDA implementation, we use the multithreaded enumeration algorithm from the fplll library [fplll16] running on all 8 (logical) cores the CPU offers. For each dimension, four knapsack matrices with a uniform 350-bit column were used as lattice, and the graph shows the median of the running time of both implementations. The matrices can also be reproduced using the tool latticegen from the fplll library (via `latticegen -randseed $s r $dim 350` for $s \in \{0, 1, 2, 3\}$). The results are displayed in Figure 1.

4.1 Pruning

Usually, lattice enumeration is used as a subroutine in the BKZ or similar algorithms [SE94]. These provide speed-length tradeoffs by using the enumeration on sublattices or projections of the lattice of smaller dimension. By choosing the dimension of the enumerated lattices appropriately, working with lattices of much greater dimension is possible. A technique that can significantly reduce the enumeration time is to work with a reduced enumeration radius, risking that no lattice point within the bound exists [GNR10; CN11]. If no lattice point was found by the enumeration, the basis is randomized and the enumeration is applied again. This reduces the "denseness" of the enumeration tree, which leads to more branching during the algorithm. If the dimension of the enumerated lattice is high enough however (corresponds to the BKZ block size), the CUDA algorithm has a similar advantage as in the benchmark without pruning. The results seen in Table 1 are obtained on the same system as above, by applying the fplll BKZ implementation with block size 53 and standard fplll pruning parameters to a random, 100-dimensional, 300-bit knapsack matrix (i.e. `latticegen r 100 300 |`

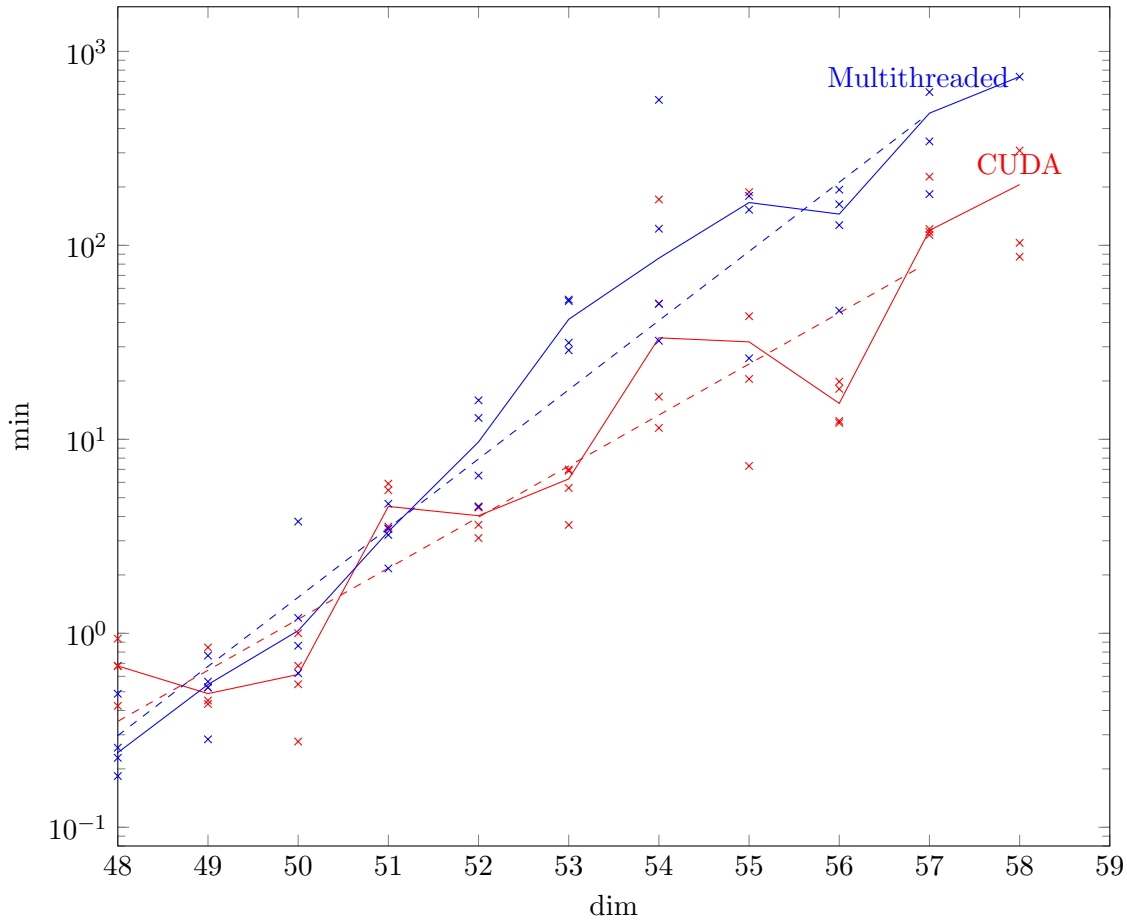

Figure 1: Performance of CUDA enumeration (red) and multithreaded enumeration (blue), the dashed lines are exponential regression curves; some data points are clipped

	Time (min)	Executed BKZ tours	Time per tour (min)
Multithreaded	1462, 1190	7, 2	209, 595
CUDA	769, 630	6, 4	128, 158

Table 1: Performance of both enumerations as part of BKZ, with pruning; data for different matrices are comma-separated

`./fplll -a bkz -b 53`). Note that during a single BKZ tour, enumerations on different block sizes are performed (usually, the block size decreases at the end of the tour). Also, the more tours have been executed, the more reduced the lattice is, so later tours may be faster than earlier ones. An exemplary running time comparison over multiple BKZ tours on one lattice can be seen in Figure 2. The random seed of 3 was chosen to use different lattices from Table 1.

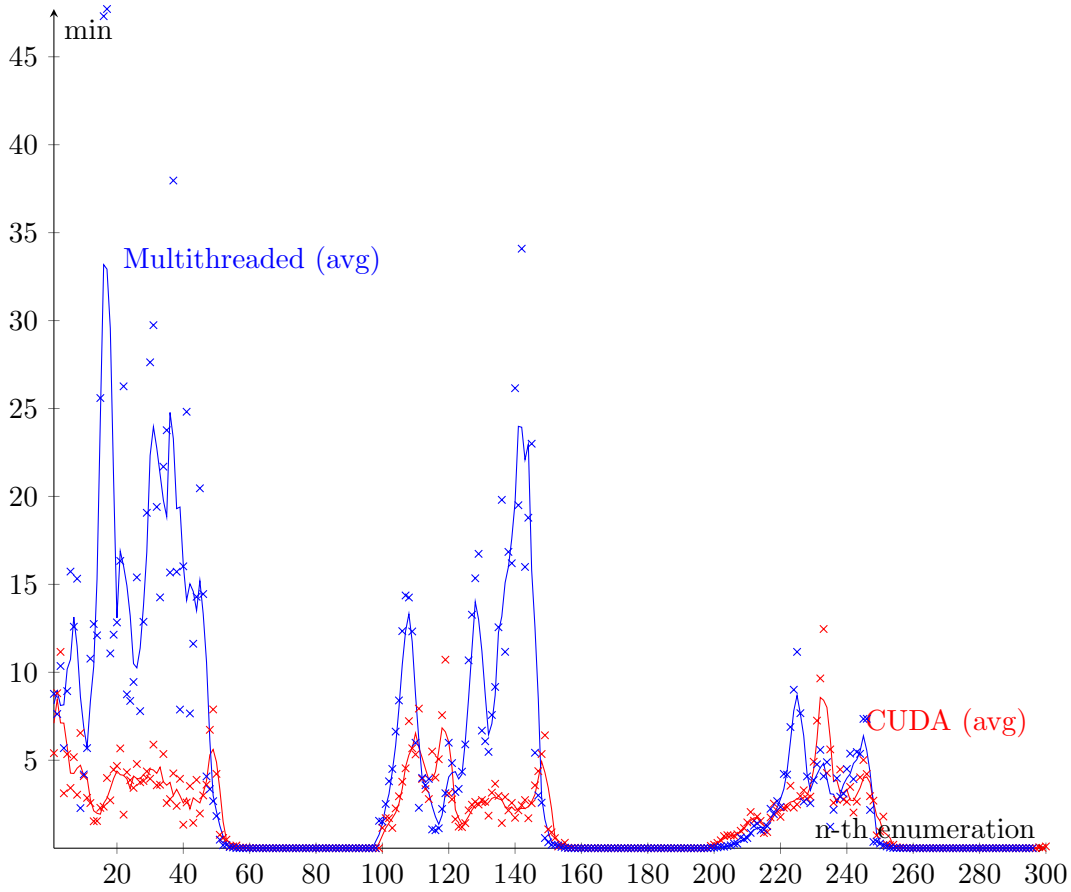

Figure 2: Performance of enumerations during one BKZ execution with block size 53 (on lattice from `./latticegen -randseed 3 r 100 300`); the averages are taken over five consecutive values

5 Source Code

At github.com/FeanorTheElf/fplll-cuda-enumeration the source code can currently be found. It is possible that it will be moved to the fplll organization github.com/fplll soon.

References

[Kan83] Ravi Kannan. "Improved Algorithms for Integer Programming and Related Lattice Problems". In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. STOC '83. New York, NY, USA: Association for Computing Machinery, 1983, pp. 193–206.

[FP85] U. Fincke and M. Pohst. "Improved methods for calculating vectors of short length in a lattice, including a complexity analysis". In: *Math. Comp.* 44 (170) (1985), pp. 463–471.

[SE94] C. P. Schnorr and M. Euchner. "Lattice basis reduction: Improved practical algorithms and solving subset sum problems". In: *Mathematical Programming* 66 (1994), pp. 181–199.

[AKS01] Miklós Ajtai, Ravi Kumar, and D. Sivakumar. "A Sieve Algorithm for the Shortest Lattice Vector Problem". In: *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*. STOC '01. Hersonissos, Greece: Association for Computing Machinery, 2001, pp. 601–610.

[HS07] Guillaume Hanrot and Damien Stehlé. "Improved Analysis of Kannan's Shortest Lattice Vector Algorithm". In: *Advances in Cryptology - CRYPTO 2007*. Ed. by Alfred Menezes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 170–186.

[Nic+08] John Nickolls et al. "Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For?" In: *Queue* 6 (Mar. 2008), pp. 40–53.

[PS08] Xavier Pujol and Damien Stehlé. "Rigorous and Efficient Short Lattice Vectors Enumeration". In: *Advances in Cryptology - ASIACRYPT 2008*. Ed. by Josef Pieprzyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 390–405.

[GNR10] Nicolas Gama, Phong Q. Nguyen, and Oded Regev. "Lattice Enumeration Using Extreme Pruning". In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 257–278.

[CN11] Yuanmi Chen and Phong Q. Nguyen. "BKZ 2.0: Better Lattice Security Estimates". In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–20.

[Jen+11] John Jenkins et al. “Lessons Learned from Exploring the Backtracking Paradigm on the GPU”. In: *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*. Euro-Par’11. Bordeaux, France: Springer-Verlag, 2011, pp. 425–437.

[fplll16] The FPLLL development team. “fplll, a lattice reduction library”. 2016. URL: `https://github.com/fplll/fplll`.

[Alb+19] Martin R. Albrecht et al. “The General Sieve Kernel and New Records in Lattice Reduction”. In: *Advances in Cryptology – EUROCRYPT 2019*. Ed. by Yuval Ishai and Vincent Rijmen. Cham: Springer International Publishing, 2019, pp. 717–746.

[DSW21] Léo Ducas, Marc Stevens, and Wessel van Woerden. *Advanced Lattice Sieving on GPUs, with Tensor Cores*. Cryptology ePrint Archive, Report 2021/141. `https://eprint.iacr.org/2021/141`. 2021.