A vectorized, cache efficient LLL implementation

Artur Mariano, Fábio Correia, and Christian Bischof

Institute for Scientific Computing
Technische Universität Darmstadt
Darmstadt, Germany
artur.mariano@sc.tu-darmstadt.de

Abstract. This paper proposes a vectorized, cache efficient implementation of a floating-point version of the Lenstra-Lenstra-Lovász (LLL) algorithm, which is a key algorithm in many fields of computer science. We propose a re-arrangement of the data structures in LLL, which exposes parallelism and enables vectorization. We show that in one kernel, 128-bit SIMD vectorization works better than 256-bit, while in another kernel it is the other way around. In high lattice dimensions, this re-arrangement renders the implementation more cache friendly, thereby further increasing performance. Our floating-point LLL implementation is slightly slower than the implementation in the Number Theory Library (NTL) without vectorization, but 10% faster when vectorized, for lattices that require exhaustive computation with multi-precision. For larger lattices, we obtain a speedup factor of 35% over a non-vectorized implementation.

1 Introduction

Lattices are discrete subgroups of the m-dimensional Euclidean space $\mathbb{R}^m$, with a strong periodicity property. A lattice $\mathcal{L}$ generated by a basis $\mathbf{B}$, a set of linearly independent vectors $\mathbf{b}_1$,...,$\mathbf{b}_n$ in $\mathbb{R}^m$, is denoted by:

$$\mathcal{L}(\mathbf{B}) = \{\mathbf{x} \in \mathbb{R}^m : \mathbf{x} = \sum_{i=1}^{n} u_i \mathbf{b}_i,\ \mathbf{u} \in \mathbb{Z}^n\}. \tag{1}$$

where n is the rank of the lattice. When $n = m$, the lattice is said to be of full rank. When m is at least 2, each lattice has infinitely many different bases.

Lattice basis reduction is the process of transforming a given lattice basis $\mathbf{B}$ into another lattice basis $\mathbf{B}'$, whose vectors are shorter and more orthogonal than those of $\mathbf{B}$ and where $\mathbf{B}$ and $\mathbf{B}'$ generate the same lattice, i.e., $\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{B}')$. While there is not a formal definition of lattice reduction, the goal of lattice reduction algorithms is to yield a *nearly orthogonal* basis.

The Lenstra-Lenstra-Lovász (LLL) algorithm was the first tractable algorithm to reduce bases [4]. It lays the foundation for many algorithms for problems on lattices. LLL has applications in many fields in computer science, ranging from integer programming to cryptanalysis [5]. Although many core concepts in the theory of lattices are already well understood, many questions regarding the performance of lattice algorithms are still under investigation. Studying the performance potential of lattice algorithms, including LLL, is of great relevance, as this determines, for example, the potential of attacks to lattice-based cryptosystems.

The original LLL algorithm was described with rational arithmetic, which was soon realized to be overly expensive. In a breakthrough result, Schnorr et al. [6] published a floating-point version of LLL that offers good practical performance and moderate stability. Since then, various improvements of LLL's stability and its algorithmic performance have been proposed e.g. [2, 3, 5]. For instance, in 2008, Backes et al. proposed a shared-memory parallel LLL implementation, with moderate scalability. In a follow-up paper, Backes et al. achieved an improved speedup factor of 3x for 4 threads and a bit over 4x for 8 threads [1]. However, to our knowledge, there are no studies regarding the vectorization of LLL and the impact of its data structures and kernels on cache locality.

Our contributions. In this paper, we propose a re-arrangement of the data structures in LLL that leverages both cache locality and enables SIMD vectorization. The re-arrangement of the data structures offers an immediate gain in cache locality, while the width of the SIMD vectorization should be chosen based on the pattern of computation in the kernel. We show that our floating-point LLL implementation is slower than NTL's[1], but outperforms it by 10% when employing our optimizations on bases that require multi-precision support. Moreover, the performance boost of our optimizations increases with the lattice dimension.

Notation. Vectors and matrices are written in bold face, vectors are written in lower-case, and matrices in upper-case, as in vector $\mathbf{v}$ and matrix $\mathbf{M}$. The ith coordinate of a vector $\mathbf{v}$ is denoted by $\mathbf{v}_i$. $\langle \mathbf{v}, \mathbf{p} \rangle$ denotes the inner product of two vectors $\mathbf{v}$ and $\mathbf{p}$. The Euclidean norm of $\mathbf{v}$ is given by $||\mathbf{v}||$. $\mathbf{v}$ is called a *zero vector* if $||\mathbf{v}|| = 0$. v' denotes the floating point value of an exact value v. $\lceil x \rfloor$ rounds x to the nearest integer. A detailed description of the Gram Schmidt (GS) orthogonalization, which is essential in LLL reduction, can be found in Section 1.2.2 of [7].

2 A LLL floating-point implementation

We implemented a variant of the floating-point LLL algorithm proposed by Schnorr et al., described in [6]. For space reasons, we often point to Algorithm L^{3}FP in [6] throughout the text, instead of replicating its description in this paper. Floating-point LLL implementations are more practical than exact versions, but errors might occur, and a mechanism to correct them must be in place. The input of the algorithm is the lattice basis and a reduction parameter δ, which defines the extent of the reduction. The algorithm works with both *exact* and *approximate* versions of the basis. An exact copy of the basis is always available

[1] www.shoup.net/ntl/

in the algorithm, since errors in the basis change the lattice and can not be corrected, unless a copy of the exact basis is kept.

The algorithm starts at stage $k = 2$, by computing the Gram Schmidt orthogonalization, which starts with the computation of the inner product between two vectors. If the precision loss is too high, the exact dot product has to be computed, for which we use the exact version of the basis. The Gram Schmidt orthogonalization outputs the approximate values of one row of the coefficient vectors of the orthogonal basis, μ_k, and the square norm of the corresponding orthogonal vector.

The next step is a size reduction procedure of the vector b_k with all vectors b_j, for $j = k-1, ..., 1$ (step 3 of L^{3}FP in [6]), if the size reduction is possible. This procedure consists in subtracting the coordinates of one vector by another, whose coordinates are multiplied by a constant i.e. ($b_k = b_k - \lceil \mu_{k,j} \rfloor \times b_j$). If $|\mu_{k,j}| > 1/2$ holds true, it is possible to perform a size reduction. If the reduction takes place, we approximate the k-th row of the basis.

Finally, the reduced vector will be swapped with its predecessors unless the Lovász condition holds (step 4 of L^{3}FP in [6]). This condition ensures that successive vectors are at least δ times bigger than their respective predecessor. The described process is repeated for each vector in the basis, until all vectors are LLL-reduced. Once this condition is verified, an LLL-reduced basis with δ is returned. To improve the numerical stability and the performance of the algorithm, we modified it as follows:

1. As in the NTL implementation, we replaced the 50% precision loss test of [6] by another, which tolerates a loss of up to 15% in the computation of the inner products.
2. Unlike L^{3}FP in [6] (cf. line 5, step 2), we check whether the values fit into a `double` data type.If they do, we use `double`s to compute the dot product as operations are more efficient than on `xdouble`s.
3. If a given basis vector b_k can be reduced, Schnorr et al. test whether the precision loss is too high. If so, the algorithm tries to reduce b_k again. However, in our implementation, b_k is always reduced again, even when the precision loss is low. This is also how the algorithm is implemented in NTL. In addition, we also recompute the Gram Schmidt orthogonalization the first time b_k is reduced, since errors may occur that are hard to correct at a later stage.

2.1 Multi-precision and Data structures

LLL requires multi-precision capability to handle large numbers that may be present in lattices, including most lattices available from the SVP challenge[2]. A viable option to implement multi-precision is the GNU Multiple Precision Arithmetic Library (GMP) library. NTL can be compiled with either its own multi-precision module or with GMP. The LLL function in NTL is considerably faster with its own multi-precision module than with GMP, presumably because

[2] www.latticechallenge.org/svp-challenge/

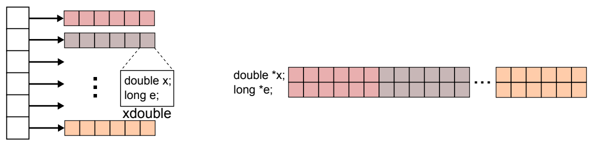

Fig. 1. Original (left side) and re-arranged (right side) data structures.

memory can be handled much more efficiently (e.g. auxiliary variables for conversion are not needed) in the multi-precision module. In our implementation, we used GMP to store exact values.

The extended exponent double precision data type (xdouble), allows to represent floating point numbers with the same precision as a double, but with a much larger exponent. It is implemented as a class, where two instance variables are used, a double x and a long e, to store the mantissa and the exponent, respectively. For any given number in the form $x \times b^e$, x denotes the mantissa, b the base and e the exponent.

The data structures of our *base* implementation consist of 2-dimensional arrays, of either xdoubles for floating point arithmetic (GS coefficients μ and the approximate basis $\mathbf{B}'$, or the GMP mpz_t data type for exact arithmetic (exact basis $\mathbf{B}$), for matrices. For vectors, we used 1-dimensional arrays containing xdoubles (square norms of the GS vectors - no vectors with exact precision are needed). In addition, two xdouble arrays are used to store the square norms of the approximated basis vectors and the result of $\mu_{k,i}c_i$ (line 8, step 2 of L^{3}FP).

2.2 Data structure re-organization and vectorization

We now describe two core modifications of our LLL implementation, which improve its performance. Figure 1 shows the re-arrangement of the data structure to store the approximate version lattice basis. On the left side, we store an array of N pointers to other arrays, each of which has N elements. Each element is stored as a xdouble object, which is a struct of two elements (a double and a long). On the right side, we show the data structure re-arranged. This re-arrangement results in immediate performance boost, as it is more cache friendly.

As the original data structures are an array of structs (AoS), cache locality is low. With the re-arrangement, multiple vectors are brought to cache with two accesses (arrays *x and *e). A vector in dimension N has N coordinates of 16 bytes each (8 bytes for the long and 8 bytes for the double). Therefore, accessing array *x brings 8 elements to each L1 cache line, assuming a 64 bytes L1 cache line size. This is also true for cache lines in L2, thus reducing memory access latency in comparison to the original implementation.

We store the GS coefficients μ in an identical data structure, although it has the shape of a lower triangular matrix. The re-arrangement is similar, as shown in Figure 2. The major difference is index calculation. In the new format, $\mu_{i,j}$ is accessed at the index $(i \times (i-1)/2) + j$, thereby incurring in a slight overhead.

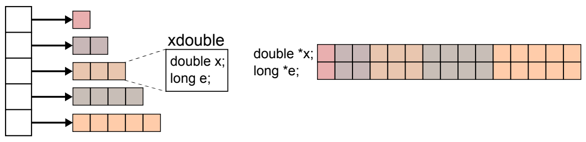

Fig. 2. Original (left side) and re-arranged (right side) data structures.

These re-arrangements also allow one to vectorize (i) the *dot product* between two vectors when they fit in doubles (cf. line 5, step 2 of L^3FP in [6]) and (ii) the add and multiply (*AddMul*) (cf. line 8 of the same source). Note that when vectors do not fit into doubles, no vectorization is used in (i), as this kernel represents a tiny percentage of the overall execution time. For (ii), we were able to partially vectorize the operation, as it is performed exclusively with xdoubles. We split the kernel in two steps. First, we multiply the elements (xdoubles) of one array by the corresponding elements of a second array, which has no dependencies and can be vectorized. In particular, the mantissas are multiplied by one another and the exponents are summed up, and both operations are vectorized. Then, we sum up the partial multiplications. However, there is a case statement in the sum which impedes vectorization.

3 Experiments

As mentioned before, we used NTL's implementation of LLL as a reference implementation. We note that NTL's implementation is faster than our base implementation due to two main reasons: (1) NTL uses its own multi-precision module, which is more efficient than GMP (which we used in our implementation), and (2) NTL's LLL implementation is more efficient than ours in terms of Gram Schmidt computations. However, our main goal is to propose optimizations that can be applied to any LLL implementation (including NTL's).

Throughout this section, we refer to our implementation as either (i) *base* implementation, for the non-optimized, implementation, (ii) *optimized/OPT*, for the version with the data structures re-arranged or (iii) *vectorized/VEC* for the version with re-arranged data structures and vectorization enabled.

We used random Goldstein-Mayer lattice bases, available on the SVP challenge website, for which we ran 50 seeds for each dimension. For tests with Ajtai lattices from the Lattice challenge[3], we run tests on a single seed for each dimension, as no lattice generator is available. The test platform has two Intel E5-2698v3 chips at 2,3 GHz, each of which has 16 cores. Each core has 32 KB of L1 instruction and data cache (a cache line has 64 bytes). L2 caches have 256 KB and are not shared. The L3 cache is shared among all the cores, and has 40 MB. The machine has 756 GBs of RAM.

The code was compiled with GNU g++ 4.8.4. We compiled the code with the -O2 optimization flag, since it was slightly better than -O3.

[3] http://www.latticechallenge.org/

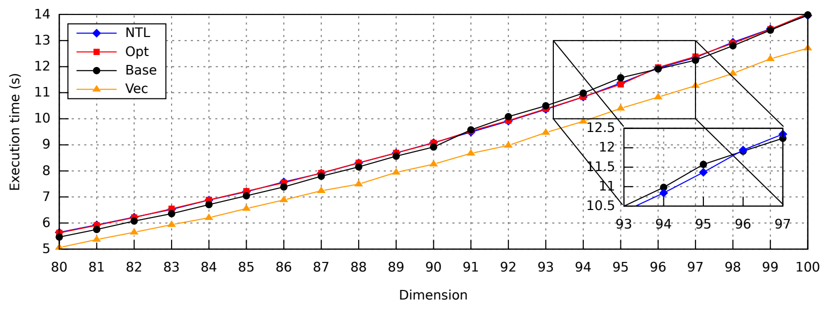

Fig. 3. Execution time of our LLL implementation and NTL's, for lattices from the SVP challenge. Note the zoom-in section for Base and OPT, between dimensions 93-97.

3.1 Goldstein-Mayer lattices (low dimensions)

In this section, we show the benchmarks that were carried out for Goldstein-Mayer lattices that we obtained from the SVP challenge. We used the lattice generator to generate 50 lattices with seeds 1-50 and thus have a statistical significant result. We run our LLL implementation and NTL's implementation for lattices in dimensions 80-100. The performance of our *base* implementation is comparable to NTL's (it is about 3% slower), as shown in Figure 3 (note the zoom-in section where the performance difference is accentuated). We did not extend the benchmarks to higher dimensions as the pattern seems to be fairly stable and higher dimensions require large chunks of time to be tested (dimension 100 required about 14 seconds $\times$ 50 seeds = $\approx$12 minutes, and dimension 150 would require about 3.5 hours).

We note that our *optimized* (OPT) version does not perform necessarily better than the *base* version. We believe that this happens because the lattices we tested are too small to exhibit enough cache locality gains to outweigh the overhead incurred in this version. To prove this, we measured the cache misses of our implementation at the L1 level cache[4], as shown in Figure 4. As the figure shows, our OPT version incurs much fewer cache misses than the base version, and in particular, the difference increases with higher lattice dimensions. Ideally, we would test lattices in dimension 500-1000 but these dimensions would be impractical to solve on this type of lattices. In the next section, we test lattices in dimensions 500-800, on a kind of lattices that requires far less time.

Our *vectorized* (VEC) version obtains speedups of 9-11% over the base version. This version incurs, somewhat surprisingly, more cache misses than the other versions. We believe that this happens because, as performance increases, more memory accesses are performed within the same timespan, thus shortening the window opportunity for efficient prefetching.

With 256-bit SIMD vectorization, we could obtain a theoretical maximum speedup of 4x (as we vectorize 8-byte `doubles`) and with 128-bit SIMD vectoriza-

[4] A complete version of this paper extends this analysis.

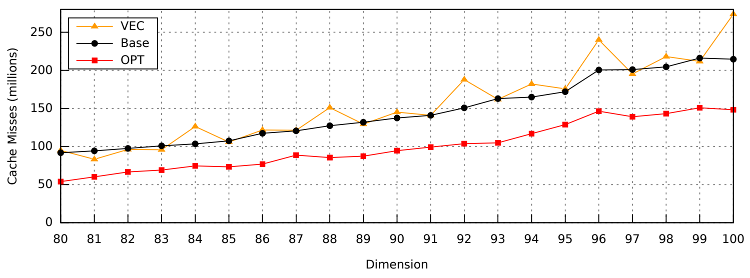

Fig. 4. L1 cache misses (in millions) of our implementation on Goldstein-Mayer lattices.

tion we could obtain a theoretical maximum speedup of 2x, for the same reason (but for 8-byte longs). Thus, in theory, we could achieve an overall speedup of 19.5%, as the *dot product* loop takes approximately 16% of the execution time of the base version (for a lattice in dimension 100), for which we used 256-bit SIMD registers, while the *AddMul* loop takes approximately 31%, for which we used 128-bit SIMD registers[5]. A 11% speedup is in our view a good result, as the maximum number of vectorized elements is N (in this case 100, at most), which is not sufficient to achieve the full potential of vectorization. For 256-bit SIMD vectorization we used AVX2, while for 128-bit SIMD vectorization SSE 4.2 was used.

3.2 Ajtai lattices (high dimensions)

For lattices from the SVP challenge, LLL is only practical below dimension 200, as we mentioned in the previous subsection. Ajtai lattices from the Lattice challenge allows one to carry out benchmarks with larger lattice dimensions, as this kind of lattices contain far smaller numbers and LLL-reduces them much faster. Note that lattices from the SVP challenge have numbers with over 300 digits, while lattices from the Lattice challenge have numbers with no more than 3 or 4 digits.

Figure 5 compares our implementation against NTL's, for lattices between dimension 500 and 800. NTL is approximately 2x faster than our base implementation, as (i) NTL saves more GS computations in higher lattice dimensions and (ii) converting data types from/to GMP, which we use, incurs increasing overhead with the lattice dimension. However, the key point in this subsection is not to show how our implementation compares to NTL, but what performance gain can be attained when optimizing it. As the figure shows, we obtain a 6% speedup by simply switching to the optimized (aka with re-organized data structures) version. This backs up our claim that re-organizing the data structures delivers higher gains for higher lattice dimensions, as the experiments in the last subsection were only done for lattices up to dimension 100.

[5] As the number of elements that are vectorized in the loop decreases, there may not be 4 elements, which are necessary to use 256-bit SIMD.

In addition, we obtain a speedup of as much as 35% (from which 6% is obtained from the data structures re-arrangement). For the vectorization, we could achieve a theoretical speedup of 36.9%, as the *dot product* loop takes approximately 26.4% of the execution time of the base version (for a lattice in dimension 100), for which we used 256-bit SIMD registers, while the *AddMul* loop takes approximately 60.6%, for which we used 128-bit SIMD registers. The overall speedup of 35% (29%, if the speedup from the re-arrangement is deducted) is thus closer to the maximum possible speedup of 36.9%, which backs up our claim that the vectorization benefit increases with the lattice dimension.

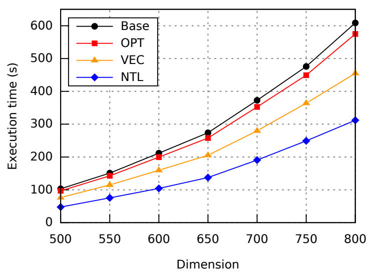

Fig. 5. Execution time of our LLL implementation and NTL's LLL implementation, for Ajtai lattices from the Lattice challenge.

4 Conclusions

Although a comprehensive body of work pertaining to LLL has been published in the last decades, there are no studies regarding the vectorization of LLL and the impact of its data structures and kernels on cache locality. In this paper, we fill this gap in knowledge. We propose a re-organization of the data structures in the algorithm, which enables the vectorization of two computationally expensive kernels. We show that (i) our data structure re-arrangement increases performance with the lattice dimension, (ii) vectorizing the *dot product* and *AddMul* kernels can achieve as much as 35% speedup on larger lattices and (iii) our implementation is as much as 10% more efficient than NTL's on smaller lattices.

References

1. W. Backes and S. Wetzel. Improving the Parallel Schnorr-Euchner LLL Algorithm. ICA3PP'11, pages 27–39, 2011.
2. H. Koy and C. P. Schnorr. *Cryptography and Lattices: International Conference*, chapter Segment LLL-Reduction of Lattice Bases, pages 67–80. 2001.
3. H. Koy and C. P. Schnorr. *Cryptography and Lattices: International Conference*, chapter Segment LLL-Reduction with Floating Point Orthogonalization, pages 81–96. 2001.
4. A. Lenstra, H. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.
5. P. Q. Nguên and D. Stehlé. *EUROCRYPT'05*, chapter Floating-Point LLL Revisited, pages 215–233.
6. C. Schnorr and et al. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In *Math. Programming*, pages 181–191, 1993.
7. D. Stehlé. Floating-point LLL: theoretical and practical aspects. In *The LLL Algorithm - Survey and Applications*, pages 179–213. 2010.