Speeding up lattice sieve with Xeon Phi coprocessor

Anja Becker,* and Dusan Kostic

EPFL, Lausanne, Switzerland
anja.becker@epfl.ch, dusan.kostic@epfl.ch

Abstract. Major substep in a lattice sieve algorithm which solves the Euclidean shortest vector problem (SVP) is the computation of sums and Euclidean norms of many vector pairs. Finding a solution to the SVP is the foundation of an attack against many lattice based crypto systems. We optimize the main subfunction of a sieve for the regular main processor and for the co-processor to speed up the algorithm in total. Furthermore, we show that the co-processor can provide a significant performance improvement for highly parallel tasks in the lattice sieve. Four-fold speed up achieved, compared to the CPU, indicates that co-processors are a viable choice for implementation of lattice sieve algorithms.

Keywords: Xeon Phi, vector, norm, lattice, sieve, shortest vector problem

1 Introduction

The security of the lattice-based cryptography relies on two well known problems: the shortest vector problem (SVP) and the closest vector problem (CVP). The SVP is a problem of finding a shortest non-zero lattice vector given a basis of the lattice and the CVP is defined as finding the lattice vector which is closest to the given arbitrary lattice vector and given a lattice basis. We consider Euclidean lattices and Euclidean vector norms and in the rest of the paper we will omit the prefix Euclidean for brevity. Both problems are known to be NP-hard[1] to solve exactly [1, 14] and also NP-hard to approximate [7, 17] within at least constant factors.

The time complexity of known algorithms that find the *exact* solution to the shortest vector problem (SVP) or closest vector problem (CVP) are at least exponential in the dimension of the lattice. These algorithms also serve as subroutines for strong polynomial time *approximation* algorithms which have been shown to be of great use as a cryptanalytic tool. In cryptology, they were used for a long time, first through a direct approach as in [12] and then more indirectly using Coppersmith's small roots algorithms [4, 5]. Algorithms for the exact problem enable us to choose appropriate parameters for cryptographic schemes.

A *shortest* vector can be found by enumeration [6, 22, 13], sieving [2, 20, 19, 23, 3, 8, 24, 15, 16] or the Voronoi-cell algorithm [18]. Enumeration uses a negligible amount of memory and its running time is between $2^{\tilde{\mathcal{O}}(m)}$ and $2^{\tilde{\mathcal{O}}(m^2)}$ depending on the amount and quality of the preprocessing. Probabilistic sieving algorithms, as well as the deterministic Voronoi-cell algorithm are simply exponential in time and memory.

Our contribution. The basic step in sieving algorithms for SVP is to search for pairs of lattice vectors for which the norm of the sum or difference is smaller than a given bound, the so-called *neighboring* vectors . This is the dominating part of sieving algorithms in

* This work was supported by the Swiss National Science Foundation under grant numbers 200021-126368 and 200020-153113.

[1] Under randomized reductions in the case of SVP.

terms of running time and thus it is important to do it as efficiently as possible. In this report we focus on optimizing the searching procedure both for the regular main CPU and for the Intel Xeon Phi co-processor, and show that the co-processor is suitable for this task.

2 Basic background

In this section we briefly present the necessary facts about Euclidean lattices, sieving algorithms for solving the SVP and the lattice sieving algorithm based on the decomposition approach.

Euclidean lattices. A lattice $\mathcal{L}$ of dimension $s \leq m$ is a discrete subgroup of $\mathbb{R}^m$ that spans an s-dimensional linear subspace. A lattice can be described as the set of all integer combinations $\{\sum_{i=1}^{s} \alpha_i \boldsymbol{b}_i \,|\, \alpha_i \in \mathbb{Z}\}$ of s linearly independent vectors $\boldsymbol{b}_i$ in $\mathbb{R}^m$. Such vectors $\boldsymbol{b}_1, .., \boldsymbol{b}_s$ are called a basis of $\mathcal{L}$. The problem to find a shortest non-zero lattice vector from a given basis is called the shortest vector problem (SVP). A second problem is the closest vector problem (CVP), in which one needs to find a lattice vector closest to an arbitrary vector in $\mathbb{R}^m$ given a basis.

Sieving algorithms to solve SVP. Asymptotic complexity classification identifies sieving as the most efficient method to solve the exact SVP in high dimensions. Lattice sieving algorithms usually start with the list of exponentially many large lattice vectors (obtained by enumeration, e.g. Schnorr-Euchner algorithm [22]), and iteratively combine these vectors, for instance replacing vectors by their difference with other close vectors in the list, when they are shorter. The algorithm goes on, while the norm of the vectors in the list decreases, until the list contains the shortest lattice vector, or at least, a very good approximation. Sieving algorithms differ by the strategy to find pairs of vectors close to each other in a list of arbitrary lattice vectors. Simple algorithms essentially test all pairs of vectors using a quadratic approach to detect close vectors.

In [3] the authors proposed a new heuristic algorithm for solving the exact SVP and CVP for n-dimensional lattices, namely the decomposition approach (Alg.1 in [3]). The new algorithm follows the method of lattice sieving. The main part of the algorithm (function *Merge by collision* - Alg.2 in [3]) accepts a list of vectors as an input, searches for *neighboring* vectors, and outputs a new list. Before the search, input list of vectors is divided into buckets and buckets are grouped into pairs, such that each bucket corresponds to exactly one other bucket (see [3] for details). In the search routine, which we call *FindNeighbors*, we go through each of the bucket pairs and test all the vector pairs from those corresponding buckets. Two vectors which form a vector pair pass the test (are neighbors) if the norm of their sum is lower than a given bound and in that case the summation vector is added to the output list. *Merge by collision* is the most time consuming part of the algorithm and speeding it up would yield a significant performance gain for the whole algorithm. More precisely, in this report we aim at improving the performance of the subroutine *FindNeigbors* which is presented in more details in the following sections.

3 Intel Xeon Phi coprocessor and optimization tools

In the following section we present a high level description of the Intel Xeon Phi coprocessor architecture and programming model [9–11]. Intel Many Integrated Core (MIC) architecture combines many Intel processor cores onto a single chip. The first product

based on this architecture was Xeon Phi Knights Ferry in 2010, after which Intel developed two more generations, namely the Knights Corner (KNC) and the Knights Landing (KNL).

The Xeon Phi coprocessors comprise up to 61 in-order single-issue processor cores running at 1 to 1.3 GHz. Each individual core supports 4 hardware threads, resulting in up to 244 threads per device. The cores are connected with an on-die bidirectional interconnect network. One of the most important features of the Xeon Phi are the Vector Processing Units (VPUs). There is one 512-bit wide VPU on each of the cores. It contains 128 vector registers of 512 bits each, resulting in 32 such vector registers per thread context. Intel AVX-512 extension of the Instruction Set Architecture offers vector instructions which can utilize the VPUs.

Each core of the Xeon Phi has 32 KB L1 data cache with 64 B cache line size, 32 KB L1 instruction cache, and a 512 KB L2 cache. All the L2 caches are connected with each other, effectively creating a shared last level cache of up to 32 MB, depending on the number of cores. Xeon Phi also supports up to 16 GB of RAM memory. The coprocessor is connected to the host system through a PCI Express system interface.

There are two execution modes: *native* and *offload* mode. In the *native* mode, the coprocessor is treated as a standalone multicore computer running a Linux μOS. The binary of the application is built on the host system copied to the coprocessor along with the data and then executed. In the *offload* mode, the compiler on the host system generates a heterogeneous binary to run on the host, while selected regions of the code are offloaded to the coprocessor during runtime. Offload can be done in a synchronous way in which case the host code will wait until the offloaded computation is finished and the data is available, or in an asynchronous way when the host code will continue with the execution and collect the data from the coprocessor at some later point in time.

Optimization methods for Xeon Phi can be summarized as follows:

- Utilization of powerful VPUs which can work with 16 single-precision elements simultaneously (or 8 double-precision elements) in SIMD mode. Compiler can *vectorize* the source code by itself, but in some more complex cases the code should be *vectorized* with explicit intrinsic functions.
- Utilization of all available hardware threads - it is important to use more than one thread per core (preferably 4) because the latency of arithmetic vector instructions is 4 cycles and it is not possible to issue instructions from the same thread context in consecutive cycles.
- Proper memory alignment - the best possible performance with vector instructions can be achieved only if the memory is properly aligned.
- Reduce the need for synchronization between threads as much as possible - compared to the general purpose CPUs, synchronization on the Xeon Phi is a bigger issue, since there are many more threads (e.g. 240 vs 32).

4 Detailed optimization of the FindNeighbors step

The algorithm we aim at improving is the *Merge by collision* outlined in Alg. 1, more precisely the subfunction called *FindNeighbors*. Let $\mathcal{L}$ be a lattice of dimension n, then a coset $\mathcal{C}$ of $\mathcal{L}$ is a translation of $\mathcal{L}$ by a vector $x \in \text{span}(\mathcal{L})$, namely the set $\mathcal{C} = \{x + v | v \in \mathcal{L}\}$. Bounded coset $\mathcal{C}$ is a coset such that for all vectors $v \in \mathcal{C}$ norm of v is lower than a given bound R. Algorithm 1 is given a list of vectors which belong to the bounded coset $\mathcal{C}_{in}$, testing morphism ϕ, and bound R. Based on the morphism ϕ, vectors from input coset are organized into k buckets. Buckets are then paired into $k/2$ bucket pairs such that the sum of two vectors from two corresponding buckets belongs to the output coset (without the restriction on the norm - not bounded). *FindNeighbors* algorithm then runs

Algorithm 1 Merge by collision

Input: Bounded coset C_{in}, Testing morphism ϕ, Radius bound R
Output: Bounded coset C_{out}
1: Reorganize vectors in C_{in} into k buckets indexed by values of ϕ
2: Group buckets into $k/2$ pairs of buckets (A_i, B_i) s.t. for every bucket pair (A_j, B_j) for all $u \in A_j, v \in B_j$ we have $\phi(u) = -\phi(v)$
3: $C_{out} \leftarrow FindNeighbors(A_i, B_i, R)$
4: **return** C_{out}

Algorithm 2 Find Neighbors

Input: Pairs of vector buckets (A_i, B_i), vector hashes, bound R
Output: List of vectors such that their norm is less than the bound
1: **for each** pair of buckets (A_i, B_i) **do**
2: **for all** vectors $\boldsymbol{u} \in$ bucket A_i **do**
3: **for all** vectors $\boldsymbol{v} \in$ bucket B_i **do**
4: **if** norm$(\boldsymbol{u} + \boldsymbol{v}) \leq$ bound **then**
5: add vector $\boldsymbol{u} + \boldsymbol{v}$ to the output list uniquely
6: **return** output list of vectors

through all the potentially good vector pairs, tests if the norm of the summation vector is lower than the bound R and adds the ones which pass the test to the output bounded coset $\mathcal{C}_{out}$. Potentially good vector pairs are formed from vectors from two corresponding buckets. The outline of the *FindNeighbors* algorithm is given in Alg. 2.

Note: This is a high level description of the *Merge by collision* algorithm. We omit the details because they are not necessary for understanding the algorithm we are optimizing in this report - *FindNeighbors*. For details on how testing morphism is chosen and how vectors are divided into buckets refer to [3].

Since the bucket pairs are completely independent, they can be processed in parallel. Parallel *FindNeighbors* algorithm is described in Alg. 3. For each vector pair from the corresponding buckets we do three steps: compute the squared norm of the sum of the vectors and compare it with the bound, check if the pair is already added to the output (we use a hash table), and add the pair to the output list.

Although the processing of the bucket pairs is independent, there are two steps in the parallel algorithm when the synchronization of the threads is necessary. Those steps are labelled as *critical* in Alg. 3 (lines 7 and 9).

Uniqueness - hash table. We want a list of unique vectors at the output of the algorithm and to achieve this we use a hash table. Before adding the vector pair to the output list we check if it is already in the hash table and if it is we avoid putting it in the output again. Alongside with the vectors at the input of the algorithm we also have vector hashes. When we want to check if the vector pair is unique we sum the hashes of the two vectors and check if the resulting sum of hashes is already in the table. Since there is a single output list and single hash table, shared among all the threads, access to the list and the table has to be properly synchronized.

Note: The construction of the hash table and the function used to insert the elements in the table (simple sum of vector hashes) is completely taken from the implementation of the algorithms provided by the authors of [3] in order to maintain compatibility between two implementations.

Output format. Furthermore, we do not store the summation vectors in the output list but only indices of the involved vectors, and the summation vectors are constructed afterwords from the indices. Storing only the indices has two advantages. Firstly, the

Algorithm 3 Parallel FindNeighbors

Input: Pairs of vector buckets (A_i, B_i), vector hashes, bound R
Output: List of vector's indices fulfilling the requirements

```
1: Initialize hash table
2: for each pair of buckets (A_i, B_i) in parallel do
3:     for all vectors u ∈ bucket A_i do
4:         for all vectors v ∈ bucket B_i do
5:             norm_sq ← compute_squared_norm_of_the_sum(u, v)
6:             if norm_sq ≤ R then
7:                 CRITICAL{ unique ← check if vector pair exists in the hash table }
8:                 if unique then
9:                     CRITICAL{ add indices of vectors u and v to the output list }
10: return output list of vector indices
```

required amount of memory is smaller because we store only two indices, compared to n vector elements if we want to store the summation vectors. Secondly, performance is improved because we have smaller number of memory writes (2 compared to n, for each vector pair) and we do not need to compute the summation vector but just the squared norm of the sum of given vectors. Computing the resulting summation vectors from the indices, after the *FindNeighbors* function, requires negligible amount of time compared to the *FindNeighbors*, because the size of the output list is approximately of the same size as the input list and *FindNeighbors* algorithm runs in time almost quadratic in the size of the input list. The size of the output list and the number of operations in *FindNeighbors* depend on the choice of parameters in *Merge by collision* algorithm, which provides a trade-off between running time and required memory. In practical implementation of the algorithm the parameters are chosen such that the size of the output and the input list is comparable.

5 Implementation of the FindNeighbors for the Xeon Phi

Implementations of the parallel *FindNeighbors* algorithm for regular CPU and for Xeon Phi coprocessor follow the same procedure and optimization techniques. The main differences are in the number of threads and in computing the squared norm of the sum of two vectors. The algorithm is implemented in C++ programming language and for parallel execution we used OpenMP [21]. OpenMP *parallel for* directive is a work sharing directive and it specifies that the iterations of the loop will be executed by a team of threads in parallel. We use this directive on the outer loop of the parallel *FindNeighbors* algorithm and thus each pair of buckets is processed by one of the available threads.

Vectorization. The most straightforward way to compute the squared norm of the sum of two n-dimensional vectors is to go through the vectors element by element, sum them, square the sum and add the square to a total sum. Since Intel Xeon Phi has powerful VPUs, we can substantially improve the performance of this function by using vector instructions. Xeon Phi has a 512 bits wide VPU and can work with vectors of 16 single-precision elements at the same time in SIMD mode (AVX512 data types and instruction set). Computing the squared norm with vector AVX512 intrinsics is outlined in the pseudo-code Alg. 4. Data type *m512* denotes the AVX512 data type which is a register 512 bits wide and in our case represents 16 single-precision elements (floats). Instructions starting with *vec* denote vector instructions, for example *vec_load* loads 512 bits from main memory into the register, while *vec_add* sums two vectors element wise and stores the result into the third vector.

Algorithm 4 Norm squared

Input: Two vectors a and b, vector dimension n
Output: Squared norm of the sum of the input vectors

```
1: m512 total_sum ← vec_init_zeroes()
2: for i = 0; i < n; i += 16 do
3:     m512 rega ← vec_load(a[i])
4:     m512 regb ← vec_load(b[i])
5:     m512 tmp ← vec_add(rega, regb)
6:     tmp ← vec_mul(tmp, tmp)
7:     total_sum ← vec_add(total_sum, tmp)
8: result ← vec_reduce(total_sum) // Sum of all the vector elements
9: return result
```

Algorithm 5 Modified function for Xeon Phi

Input: Two buckets of vectors, vector hashes, bound
Output: List of vector's indices fulfilling the requirements

```
1: Initialize hash table
2: k = vector_dimension/16     // m512 registers can store 16 floats, so k is the number of
3:                                registers required for each lattice vector
4: for i = 0; i < sizeof(left bucket); i += 4 do
5:     m512 vec_1[k], vec_2[k], vec_3[k] vec_4[k]
6:     Load 4 vectors into registers
7:     for all vectors v ∈ right bucket do
8:         normsq1 ← compute_squared_norm_of_the_sum(vec_1, v)
9:         normsq2 ← compute_squared_norm_of_the_sum(vec_2, v)
10:        normsq3 ← compute_squared_norm_of_the_sum(vec_3, v)
11:        normsq4 ← compute_squared_norm_of_the_sum(vec_4, v)
12:        Check if squared norms are less than the bound and add indices to output uniquely
13: return output list of vectors
```

Registers. Since Xeon Phi has plenty of 512-bit registers, more precisely 32 per thread, we slightly modified the algorithm such that it uses available registers more efficiently. The modified algorithm is presented in Alg. 5. In the beginning of the outer loop we load four vectors from the left bucket into registers. For each vector we need 4, 5, or 6, 512-bit registers depending on the lattice dimension. If we store the vector in registers, we avoid reading the vector from memory in each iteration of the inner loop which improves performance. By storing 4 lattice vectors in the registers before entering the inner loop we actually improve the ratio of number of memory operations versus number of arithmetic operations in the loop, since for every load of vector $\boldsymbol{v}$ from the right bucket we now compute 4 norms, compared to 1 norm when we store only one vector. Memory instructions have a **bigger latency** than arithmetic instructions, so with more arithmetic instructions the compiler reorders them in such a way that the **latency** of memory read/write instructions is **neutralized**. We tested the function with storing different number of vectors in registers (from 1 to 5 vectors) and the best performance is achieved with storing 4 of them.

Critical regions. As already mentioned, there is a need for two thread synchronization points in the algorithm - *critical* sections in Alg. 3. OpenMP *critical* keyword provides a way to specify a region of code that should be executed by only one thread at a time. We can use this approach in both cases - checking the hash table and writing indices to the output. However, **critical regions significantly degrade the performance** of the algorithm, especially on the Xeon Phi where we have 240 threads competing to

enter the critical regions which results in many threads waiting for the region to be free and not doing any useful work.

OpenMP locks. Besides *critical* regions, OpenMP provides API for software *locks* as another method of thread synchronization. We virtually **divide the hash table into several parts** and use OpenMP *locks* to control access to every part of the table. In this way we avoid locking the whole hash table when a thread needs to use it and allow more threads to use different parts of the table simultaneously, hence lowering the probability that a thread will wait to access the hash table.

OpenMP atomic. The second synchronization point is writing to the output list which is shared among all the threads. We solved this problem by maintaining a **shared counter** which indicates the next free location in the output list. When a thread needs to store the result to the output list it reads the counter value, increments it and writes to the list. Reading and incrementing of the counter has to be done atomically - without the interruption by some other thread, but writing to the list can be done simultaneously by multiple threads. OpenMP provides the *atomic* directive which specifies that a memory location must be updated atomically, without letting multiple threads attempt to write to it. This construction is much more efficient than critical regions because it uses the CPU specific atomic machine instructions and thus avoids the expensive software synchronization primitives.

Memory alignment. In order to achieve the maximum efficiency of vector instructions we have to **properly align the memory**. For AVX512 instructions data should be aligned on the boundary of 64 bytes. This is easily achieved with Intel's intrinsics for aligned memory allocation. Furthermore, since lattice dimension, and consequently the size of the vectors, is not always a multiple of 16, we have to introduce padding. Even though the lattice dimension is n, we fix the allocated size of the vectors to the next value greater or equal to n which is divisible by 16, and fill the padding with zeros, if it exists. Implementation with memory alignment has around 50 percent better performance than the one without the alignment.

5.1 Implementation of the FindNeighbors on CPU

As already mentioned, most of the optimization techniques used for the Xeon Phi can also be implemented for a regular CPU. Intel CPUs also have vector processing units and can work with vectors of 128 or 256 bits in length viewed as 4 or 8 single-precision elements respectively (SSE and AVX data types and instruction set). Computing the squared norm with SSE and AVX intrinsics follows the same procedure as presented in Alg. 4, but obviously with four and two times more iterations, since vector registers are shorter.The second big difference between CPU and Xeon Phi is in the number of available vector registers (e.g. 16 per core on CPU vs 128 per core on Phi). Because of the low number of vector registers on CPUs, the optimization step denoted as *Registers* in previous section does not give a performance improvement, on the contrary, when we store some of the vectors in the registers before the inner loop, the application is even slower. Other optimization steps from the previous section apply equally to the CPU implementation. Critical regions are replaced with OpenMP locks and atomic instructions. Data in memory is properly aligned on the boundary of 16 bytes for SSE, and 32 bytes for AVX instructions and padding is inserted if needed.

6 Experimental results

The functions as well as the testing environment is implemented in C++ and for parallelization we used OpenMP. The source code is compiled with Intel C++ Compiler

(ICPC) version 14.0.2. The testing platform consists of a dual socket motherboard with two Intel Xeon E5-2650 processors and one Intel Xeon Phi 5110P coprocessor. The specifications of the processors are shown in Table 1. In all the experiments, application on CPU was tested with 32 threads and on Xeon Phi with 240 threads.

	2 x Intel Xeon E5-2650	Intel Xeon Phi 5110P
# of Cores	16	60
# of Threads	32	240
Frequency	2 GHz	1.053 GHz
Cache size	40 MB	30 MB
VPU width	256 bits	512 bits

Table 1. CPU and Xeon Phi specifications

All the functions previously described are tested on the same data set and the results are presented bellow. The data set consists of 1920 buckets which gives 960 bucket pairs and each bucket has 3200 lattice vectors. The expected number of output vectors per bucket pair is 1000, which in total gives almost 1 million vectors. The size of this data set is chosen such that it almost fills the available RAM memory on the Xeon Phi. When the lattice dimension is increased, the size of the buckets stays relatively the same, but the number of buckets increases exponentially in the dimension. For clarity, we present the performance results for one batch of computations (1920 buckets) for different lattice dimensions.

In Table 2 are presented performance results of the function running on the CPU. There are three different implementations, one using the Intel AVX instruction set, one using the SSE instruction set, and one without any vector intrinsic instructions. Execution times are in seconds and speed-ups are calculated between versions without vectorization and with AVX vectorization. It can be clearly seen that the algorithm benefits significantly from the use of intrinsics. The code is compiled with the default level of optimization in which the compiler tries to vectorize the code even when explicit intrinsics are not used. From the results it is clear that the compiler is not able to match the performance of manually vectorized code. Implementation with SSE instructions achieved the average improvement factor of 2.8, and AVX implementation further speed-up of 1.4 times, resulting in total speed-up of approximately 4 times of AVX version compared to the initial implementation without intrinsics.

Dimension	NOVEC[s]	SSE[s]	AVX[s]	NOVEC/AVX speed-up
64	21.9089	7.16444	5.35586	4.09
80	27.0982	9.77734	6.53369	4.15
96	32.9169	12.3163	9.24105	3.56

Table 2. CPU implementation results

Table 3 shows the execution times of the implementation with critical regions and the implementation with OpenMP locks and atomics, on Xeon Phi. The results show that critical regions have a big impact on performance of the Phi, but also that this can be circumvented by a smart use of locks and atomic directive. When critical regions

are replaced with locks and atomics, the execution time of the function drops from approximately 12 to 1.5 seconds, giving a speed-up of around 8 times for all the tested lattice dimensions. On the other hand, implementation on the CPU does not benefit from locks and atomics as much as the Xeon Phi. Average speed-up on CPU among the different lattice dimensions is around 1.2 (20 percent improvement). This discrepancy between speed-ups on CPU and Xeon Phi are as expected considering that the algorithm on Phi runs with many more threads which collide in the critical regions.

Dimension	Critical regions[s]	Locks and atomic[s]	Speed-up factor
64	12.5277	1.50102	8.3
80	13.0306	1.69802	7.6
96	13.4984	1.71858	7.8

Table 3. OpenMP locks and atomic vs. critical regions on Xeon Phi

Figure 1 shows the execution time for three test cases on Xeon Phi. Blue bars represent the test case without preloading of lattice vectors into registers before the innermost loop of the algorithm 3. Orange and gray bars represent the execution times of the modified algorithm such that 1 or 4 lattice vectors are loaded into registers before the innermost loop. The pseudo code of the modification with preloading of 4 vectors is shown in algorithm 5, and functions with 1, 2, and 3 vectors preloaded are implemented in the same fashion. Performance gradually improves with increase of the number of preloaded vectors, achieving the peak performance at 4 vectors. Loading only one vector before the inner loop yields a performance improvement factor of around 1.7 when compared to the version without preloading, while working with four vectors at the same time gives a speed-up of almost 7 times.

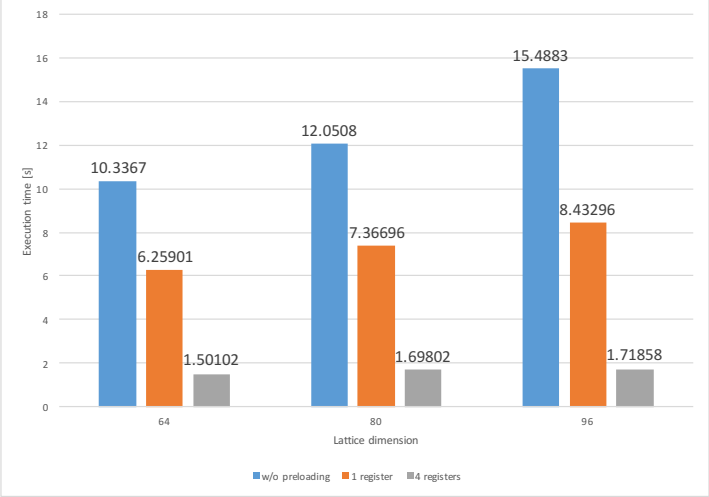

Fig. 1. Xeon Phi results with and without preloading of vectors

Finally, Figure 2 shows the comparison between the best running times achieved on CPU and Xeon Phi. For each dimension, blue and orange bars show the execution time and green boxes show the speed-up of Xeon Phi over the CPU implementation. In

both implementations the memory is aligned, and locks and atomics are used instead of critical regions. CPU implementation uses AVX intrinsics for explicit vectorization and runs with 32 threads. Xeon Phi implementation uses AVX512 intrinsics, runs with 240 threads, and 4 lattice vectors are preloaded before the inner loop. The results show a speed-up of 3.57, 3.84, and 5.37 for lattice dimensions 64, 80, and 96 respectively. The speed-up improves when the lattice dimension increases, which can be explained by the fact that increasing the dimension by 16, the number of additional instructions on CPU is twice the number of additional instructions on Xeon Phi, due to the VPUs' width (512 bits on Xeon Phi and 256 bits on CPU). It should be noted that the speed-up will not increase infinitely with the increase of the dimension. The theoretical maximum speed-up which could be achieved on the Xeon Phi compared to the CPU is approximately 8. However, we can conclude that the higher the lattice dimension, the speed-up on Phi is closer to the theoretical value.

Fig. 2. Performance results, CPU and Xeon Phi

7 Conclusion

In this paper we presented the efficient implementation of the *FindNeighbors* subroutine, which is the dominant part of the lattice sieve algorithm based on decomposition approach. In addition to optimization of the routine for regular CPUs, we presented highly optimized implementation aimed for execution on Intel's many-core coprocessor, Xeon Phi. In order to fully exploit the resources of the Xeon Phi, several optimization tools should be used, as explained in the previous sections. The power of the Xeon Phi comes from many processor cores (e.g. 60 in our device), and vector processing units which are 512 bits wide and allow us to work with 16 single-precision elements at the same time. Although compilers usually try to vectorize the source code automatically, this is not always an easy task for them, and in those cases developer needs to use appropriate instruction set extension, such as AVX or AVX512, and explicitly implement the vectorization. Experimental results showed that for *FindNeighbors* algorithm Xeon Phi 5110P outperforms two Xeon E5-2650 processors by a factor of around 4 for the tested lattice dimensions, with tendency to improve the speed-up factor in higher dimensions.

Since these devices are in the same price range and power consumption range, we can conclude that Xeon Phi is a promising platform for highly parallel applications which can utilize the vector processing units, such as the algorithm studied in this paper, but one still has to work a bit and manually optimize the code to efficiently utilize available resources.

References

1. M. Ajtai. The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract). In *Proceedings of the 30th annual ACM symposium on Theory of computing*, STOC '98, pages 10–19, New York, NY, USA, 1998. ACM Press.
2. M. Ajtai, R. Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of the 33rd annual ACM symposium on Theory of computing*, STOC 2001, pages 601–610, New York, NY, USA, 2001. ACM, ACM Press.
3. A. Becker, N. Gama, and A. Joux. A sieve algorithm based on overlattices. *LMS Journal of Computation and Mathematics*, 17:49–70, 2014.
4. D. Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In U. Maurer, editor, *Advances in Cryptology – EUROCRYPT 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin Heidelberg, 1996.
5. D. Coppersmith. Finding a small root of a univariate modular equation. In M. Darnell, editor, *Advances in Cryptology – EUROCRYPT 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 155–165. Springer Berlin Heidelberg, 1996.
6. U. Dieter. How to calculate shortest vectors in a lattice. *Mathematics of Computation*, 29(131):827–833, 1975.
7. I. Dinur, G. Kindler, and S. Safra. Approximating-cvp to within almost-polynomial factors is np-hard. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS '98)*, pages 99–109, Washington, DC, USA, 1998. IEEE Computer Society.
8. G. Hanrot, X. Pujol, and D. Stehlé. Algorithms for the shortest and closest lattice vector problems. In *Coding and Cryptology - Third International Workshop, IWCC 2011, Qingdao, China, May 30-June 3, 2011. Proceedings*, pages 159–190, 2011.
9. Intel. Intel xeon phi coprocessor developer's quick start guide.
10. Intel. Intel xeon phi coprocessor instruction set architecture reference manual. https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf.
11. Intel. Intel xeon phi coprocessor system software developers guide. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf.
12. A. Joux and J. Stern. Lattice reduction: A toolbox for the cryptanalyst. *Journal of Cryptology*, 11(3):161–185, 1998.
13. R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the 15th annual ACM symposium on Theory of computing*, STOC '83, pages 193–206, New York, NY, USA, 1983. ACM Press.
14. R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.
15. T. Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In R. Gennaro and M. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9215 of *Lecture Notes in Computer Science*, pages 3–22. Springer Berlin Heidelberg, 2015.
16. T. Laarhoven and B. de Weger. Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. *IACR Cryptology ePrint Archive*, 2015:211, 2015.
17. D. Micciancio. The shortest vector in a lattice is hard to approximate to within some constant. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, FOCS '98, pages 92–, Washington, DC, USA, 1998. IEEE Computer Society.
18. D. Micciancio and P. Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on Voronoi cell computations. In *Proceedings of the 42nd ACM*

symposium on Theory of computing, STOC '10, pages 351–358, New York, NY, USA, 2010. ACM.

19. D. Micciancio and P. Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms (SODA '10)*, pages 1468–1480. ACM/SIAM, Society for Industrial and Applied Mathematics, 2010.
20. P. Q. Nguyen and T. Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2):181–207, 2008.
21. OpenMP Architecture Review Board. OpenMP API version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.
22. K.-P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66:181–199, 1994.
23. X. Wang, M. Liu, C. Tian, and J. Bi. Improved Nguyen-Vidick heuristic sieve algorithm for shortest vector problem. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 1–9, New York, NY, USA, 2011. ACM.
24. F. Zhang, Y. Pan, and G. Hu. A Three-Level Sieve Algorithm for the Shortest Vector Problem. In T. Lange, K. Lauter, and P. Lisonek, editors, *SAC 2013 - 20th International Conference on Selected Areas in Cryptography*, Lecture Notes in Computer Science, pages 29–47, Burnaby, Canada, 2013. Springer Berlin Heidelberg.