A Multiplatform Parallel Approach for Lattice Sieving Algorithms

Michal Andrzejczak[1(✉)] and Kris Gaj[2]

[1] Military University of Technology, Warsaw, Poland
michal.andrzejczak@wat.edu.pl
[2] George Mason University, Fairfax, VA, USA
kgaj@gmu.edu

Abstract. Lattice sieving is currently the leading class of algorithms for solving the shortest vector problem over lattices. The computational difficulty of this problem is the basis for constructing secure post-quantum public-key cryptosystems based on lattices. In this paper, we present a novel massively parallel approach for solving the shortest vector problem using lattice sieving and hardware acceleration. We combine previously reported algorithms with a proper caching strategy and develop hardware architecture. The main advantage of the proposed approach is eliminating the overhead of the data transfer between a CPU and a hardware accelerator. The authors believe that this is the first such architecture reported in the literature to date and predict to achieve up to 8 times higher throughput when compared to a multi-core high-performance CPU. Presented methods can be adapted for other sieving algorithms hard to implement in FPGAs due to the communication and memory bottleneck.

Keywords: Lattice sieving · Hardware acceleration · Cryptography · Multi-platform parallel approach · Parallel algorithms

1 Introduction

Over the last decade, post-quantum cryptography (PQC) has emerged as one of the most important topics in the area of theoretical and applied cryptography. This new branch of cryptology is considered an answer to the threat of quantum computers. A full-scale quantum computer will be able to break popular public-key cryptosystems, such as RSA and ECDSA, using Shor's algorithm.

In 2016, the United States National Institute of Standards and Technology (NIST) announced the Post-Quantum Cryptography Standardization Process (NIST PQC), aimed at developing new cryptographic standards resistant to attacks involving quantum computers. In January 2019, 26 of these candidates (including results of a few mergers) advanced to Round 2.

The biggest group of submissions were lattice-based algorithms. The difficulty of breaking these cryptosystems relies on the complexity of some well-known and

M. Qiu (Ed.): ICA3PP 2020, LNCS 12452, pp. 661–680, 2020.
https://doi.org/10.1007/978-3-030-60245-1_45

computationally-hard problems regarding mathematical objects called lattices. One of these problems is the Shortest Vector Problem (SVP). Lattice sieving, which is a subject of this paper, is a family of algorithms that can be used to solve SVP (at least for relatively small to medium dimensions of lattices).

Recently, significant progress in lattice sieving has been made, especially due to Albrecht et al. [4]. Although multiple types of lattice sieving algorithms emerged in recent years, all of them share a single fundamental operation called vector reduction. As a result, an efficient acceleration of vector reduction is likely to work with most of the sieves and give them a significant boost in performance.

Sieving is a popular technique in cryptography. It was used previously, for example, for factoring large integers. However, it is a memory-intense method, so there exists a data transfer bottleneck disrupting any potential hardware acceleration, which is the biggest problem.

1.1 Contribution

To take full advantage of modern hardware parallel capabilities, we propose a modified approach to lattice sieving algorithms and present a massively parallel FPGA sieving accelerator. In the modified sieving algorithm, due to proper caching techniques on the hardware side, there is no data transfer bottleneck. Thus, the accelerator works with full performance, and a significant speed-up is achieved. In the end, the cost comparison for solving SVP instances with Amazon AWS is presented.

2 Mathematical Background

A *lattice* $\mathcal{L}$ is a discrete additive group generated by n linearly independent vectors $\mathbf{b_1}, \mathbf{b_2}, \ldots, \mathbf{b_n} \in \mathbb{R}^m$

$$\mathcal{L}(\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n) = \left\{ \sum x_i \mathbf{b}_i \mid x_i \in \mathbb{Z} \right\} \tag{1}$$

The vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n$ are called a *basis* of the lattice, and we define $\mathbf{B}$ as a $m \times n$ matrix consisting of basis vectors as columns. In this paper, we consider only the case of $m = n$.

The lattice, understood as a set of vectors, can be written as

$$\mathcal{L}(\mathbf{B}) = \{\mathbf{xB} \mid \mathbf{x} \in \mathbb{Z}^n\} \tag{2}$$

We define the length of a vector as the Euclidean norm $\|\mathbf{x}\| = \sqrt{\sum \mathbf{x}_i^2}$. The Shortest Vector Problem (SVP) aims at finding a linear combination of basis vectors with the shortest length possible. For a given basis $\mathbf{B} \in \mathbb{Z}^{n \times m}$, the shortest vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ is a vector for which

$$\forall \mathbf{x} \in \mathbb{Z}^n \quad \|\mathbf{v}\| \leq \|\mathbf{xB}\| \tag{3}$$

The shortest vector in a lattice is also called the first successive minimum and is denoted $\lambda_1(\mathcal{L})$. There are known estimates on boundaries of the length of the

shortest vector in a given lattice. For the most well-known SVP Challenge [1], a found vector $\mathbf{v}$ should be shorter than

$$\|\mathbf{v}\| \leq 1.05 \cdot \frac{(\Gamma(n/2+1))^{1/n}}{\sqrt{\pi}} \cdot (\det \mathcal{L})^{1/n}, \tag{4}$$

where Γ is Euler's gamma function, and det is determinant of the basis $\mathbf{B}$ generating the lattice $\mathcal{L}$.

If two different vectors $\mathbf{v}$, $\mathbf{u} \in \mathcal{L}(\mathbf{B})$ satisfy $\|\mathbf{v} \pm \mathbf{u}\| \geq \max(\|\mathbf{v}\|, \|\mathbf{u}\|)$, then $\mathbf{v}, \mathbf{u}$ are called *Gauss-reduced*. If every pair of two vectors $(\mathbf{v}, \mathbf{u})$ from the set $A \in \mathcal{L}(\mathbf{B})$ is Gauss-reduced, then the set A is called *pairwise-reduced*.

In this paper, we denote vectors as bold lowercase letters. Matrices are denoted as bold uppercase letters. Lattice points and vectors are used alternatively.

3 Lattice Sieving

SVP is one of the best-known problems involving lattices. Due to its computational complexity, it can be used as a basis for the security of lattice-based cryptosystems. Lattice sieving is one of several approaches to solve SVP. It is not a single algorithm, but rather a class of algorithms. These algorithms are similar to one another and rely on a similar basic operation, but differ in terms of their time and space complexity.

The term "lattice sieving" was proposed in the pioneering work of Ajtai–Kumar–Sivakumar [2,3]. In 2001, these authors introduced a new randomized way for finding the shortest vector in an n-dimensional lattice by sieving sampled vectors.

The main idea was to sample a long list $L = \{\mathbf{w}_1, \ldots, \mathbf{w}_N\}$ of random lattice points, and compute all possible differences among points from this list L. As the algorithm progresses, during reduction, shorter and shorter vectors are discovered. By repeating this step many times, the shortest vector in the lattice is being found as a result of subtracting two vectors $\mathbf{v}_i - \mathbf{v}_j$.

The method proposed by Ajtai et al. is the main element of lattice sieving algorithms. Other algorithms differ mostly in the way of handling lattice vectors, grouping them, or using some techniques of prediction. However, the main idea is still to sample new random vectors and reduce them using those already accumulated.

3.1 The GaussSieve

In 2010, Micciancio and Voulgaris [11] presented two new algorithms: *ListSieve* with the time complexity $2^{3.199n}$ and the space complexity $2^{1.325n}$, and *GaussSieve*, able to find a solution in the running time $2^{0.52n}$, using memory space in the range of $2^{0.2n}$. The *GaussSieve* is shown below as Algorithm 1. The key idea is taken from Ataji's work and is based mostly on pairwise vector reduction. The GaussSieve starts with an empty list of lattice points L and an

Algorithm 1: GaussSieve($\mathbf{B}$) — algorithm that can compute the shortest vector. The *.pop*() operation returns the first vector from a given queue. *KleinSampler*() is a method for random sampling of new vectors. *GaussReduce* reduces vector by other vectors from the set L.

Data: $\mathbf{B}$ - lattice basis, c - maximum number of collisions, $\lambda_1(\mathbf{B})$- targeted norm
Result: $\mathbf{t} : \mathbf{t} \in \mathcal{L}(\mathbf{B}) \wedge \|\mathbf{t}\| \leq \lambda_1(\mathbf{B})$

```
1  begin
2      L ⟵ ∅, S ⟵ ∅, i ⟵ 0, t ⟵ KleinSampler(B)
3      while i < c and ||t|| > λ1(B) do
4          if S ≠ ∅ then
5              v_new ← S.pop()
6          else
7              v_new ← KleinSampler(B)
8          end if
9          v_new ← GaussReduce(v_new, L, S)
10         if ||v_new|| = 0 then
11             i ← i + 1
12         else
13             L ← L ∪ {v_new}
14             if ||v_new|| < ||t|| then
15                 t ← v_new
16             end if
17         end if
18     end while
19     return t
20 end
```

empty stack S. The stack is the first source of points to be processed in the next iteration of reduction. In the case of an empty stack, a new point is sampled using Klein's method for sampling random lattice points [9], with modifications and extensions from [7].

Next, a sampled lattice point $\mathbf{v}$ is pairwise reduced by every vector from the list L. The reduction method called *GaussReduce* returns vectors $\mathbf{u}, \mathbf{v}$ satisfying $max(\|\mathbf{u}\|, \|\mathbf{v}\|) \leq \|\mathbf{u} \pm \mathbf{v}\|$. This method is shown below as Algorithm 2. Thus, the list L is always Gauss reduced, so in the case of reducing a vector already on the list, the vector is moved to the stack. If the vector $\mathbf{v}$ is non-zero after reducing by the whole list, it is added to L. Otherwise, the number of collisions i is incremented. A collision occurs when the point is reduced to zero, which means that the same point has been sampled before. The algorithm stops when the number of collisions exceeds the given boundary c, or the shortest vector already found is at least as short as the targeted estimate.

By analyzing Algorithm 1 and Algorithm 2, the number of *Reduce()* calls can be approximated as k^2 for the case of k vectors.

Algorithm 2: GaussReduce(**p**, L, S)

```
Data: p - lattice vector, L - list of already reduced vectors, S - list of vectors to
      reduce
Result: p - reduced vector
1  begin
2      for v_i ∈ L do
3          Reduce(p, v_i)
4      end for
5      for v_i ∈ L do
6          if Reduce(v_i, p) then
7              L ← L\{v_i}
8              S.push(v_i − p)
9          end if
10     end for
11     return p
12 end
```

There have been many papers improving the complexity of presented algorithm and proposing modifications that speed up the computations by several orders of magnitude by applying additional techniques. However, the *GaussSieve* is still a part of newer methods and the vector reduction step is crucial for every lattice sieving algorithm.

3.2 Parallel Sieves

There have been several papers devoted to developing a parallel version of a lattice sieve. In addition to the *gsieve* and *fplll* libraries, Milde and Schneider [12] proposed a parallel version of the GaussSieve algorithm. Their main idea was to use several instances of the algorithm connected in a circular fashion. Each instance has its own queue Q, list L, and stack S. When a vector is reduced by one instance, it is moved to another one. The stacks contain vectors that were reduced in a given instance and need to pass the instance's list once more. During the algorithm, vectors in the instances lists are not always Gauss reduced.

Ishiguro et al. [8] modified the idea of the parallel execution of the *GaussSieve* algorithm. The stack is only one, global for all instances (threads). The execution of the algorithm is divided into three parts. In the first part, sampled vectors in the set V (new or from the stack) are reduced by vectors in the local instance lists. After reduction, the reduced vectors are compared. If any vector is different than before the reduction step, it is moved to the global stack. In the next step, sampled vectors are reduced by themselves. In the last step, vectors from the local lists are reduced by the sampled vectors. The procedure ends with moving vectors from the set V to local lists in instances. A new batch of vectors is sampled, and the procedure starts from the beginning. The advantage of this approach for parallel execution is that vectors in local lists are always pairwise reduced.

In [5], Bos et al. combined ideas from [12] and [8]. As in Milde and Schneider, each node maintains its own local list, but the rounds are synchronized between nodes. During synchronization, the vectors are ensured to be pairwise reduced as in the Ishiguro et al. approach.

Yang et al. [13] proposed a parallel architecture for *GaussSieve* on GPUs. A single GPU executes a parallel approach proposed by Ishiguro et al. Communication and data flow between multiple GPUs is performed by adopting ideas from Bos et al.

Every paper listed above targeted a multi-thread or multi-device implementation, but due to FPGAs structure, some ideas might be also adapted to hardware.

As for FPGAs, there is no publicly available paper about hardware implementation of lattice sieving. FPGAs have been used for solving SVP, but using a class of enumeration algorithms. In 2010, Detrey et al. proposed an FPGA accelerator for the Kannan-Fincke-Pohst enumeration algorithm (KFP) [6]. For a 64-dimensional lattice, they obtained an implementation faster by a factor of 2.12, compared to a multi-core CPU, using an FPGA device with a comparable cost (Intel Core 2 Quad Q9550 vs. Xilinx Virtex-5 SXT 35). For a software implementation, the fplll library was used.

4 Hardware Acceleration of Vector Reduction

Lattices used in cryptography are usually high-dimensional. The hardest problem solved in the SVP Challenge, as of April 2020, is for a 157-dimensional lattice [1]. This dimension is still significantly smaller than dimensions of lattices used in the post-quantum cryptography public-key encryption schemes submitted to the NIST Standardization Process. However, it is still a challenge, similar to RSA-Challenge, to solve an as big problem as possible. Thus, a hardware acceleration might help to find solutions for higher dimensions in a shorter time.

Algorithm 3 describes a common way of implementing the *Reduce* function in software. This method is dependant on the lattice dimension, which affects the dot product computation and the update of the vector's value. The number of multiplications increases proportionally to the dimension. A standard modern CPU requires more time to perform the computations as the lattice dimension increases. However, both affected operations are highly parallelizable. Almost all multiplications can be performed concurrently by utilizing low-level parallelism. Thus, specialized hardware can be competitive even for modern CPUs capable of performing vectorized instructions and can offer a higher level of parallelism.

In this section, we present a new hardware architecture for lattice vector reduction. The analysis is performed step by step, line by line, and the corresponding hardware is proposed.

4.1 Computing a Vector Product

Computing the product of two vectors and an update of the vector's value are the two most time-consuming operations during reduction, but there is a chance for

Algorithm 3: Reduce($\mathbf{v}$, $\mathbf{u}$) – vector reduction. The return value is *true* or $false$, depending on whether reduction occurs or not.

Data: $\mathbf{v}, \mathbf{u}$ - lattice vectors
Result: *true* or $false$

```
1  begin
2      dot = Σ v_i · u_i
3      if 2 · |dot| ≤ ||u||^2 then
4          return false
5      else
6          q = ⌊ dot / ||u||^2 ⌉
7          for i = 0; i < n; i++ do
8              v_i -= q · u_i
9          end for
10         ||v||^2 += q^2 · ||u||^2 − 2 · q · dot
11         return true
12     end if
13 end
```

FPGAs to accelerate these computations with massive parallelism. The proposed hardware circuit for obtaining a vector product is shown in Fig. 1. The first step is a multiplication of corresponding coefficients. This multiplication is performed in one clock cycle, even for a very large lattice. After executing the multiplication step, the results are moved to an addition tree, consisting of $\lceil \log_2(n) \rceil$ addition layers. For the majority of FPGAs, the critical path for addition is shorter than for multiplication. Thus, it is possible to perform more than one addition in a single clock cycle without negatively affecting the maximum clock frequency. Let β denote the number of additions performed in one clock cycle, with a shorter latency path than multiplication. This parameter depends on an FPGA vendor and device family. For our target device, $\beta = 4$ addition layers are executed in one clock cycle, and the latency of the addition tree for an n-dimensional vector is $\lceil \log_2(n)/\beta \rceil$ clock cycles.

The proposed design also offers an option for the pipelined execution. It is possible to feed new vectors to registers $\mathbf{v}$ and $\mathbf{u}$ in each clock cycle, reaching the highest possible performance for a given set of vectors. The total latency required for computing the vector product is $1 + \lceil \log_2(n)/\beta \rceil$ cycles. Using this approach, the maximum level of parallelism is achieved.

4.2 Division with Rounding to the Nearest Integer

The next operation performed in the proposed accelerator for the *Reduce* function is a division with rounding to the nearest integer. The division involves the computed vector product, *dot*, and the square of the norm of the second vector $\|\mathbf{u}\|^2$. Instead of performing normal division, we take advantage of the fact that the result of the division is rounded to the nearest integer in a limited range,

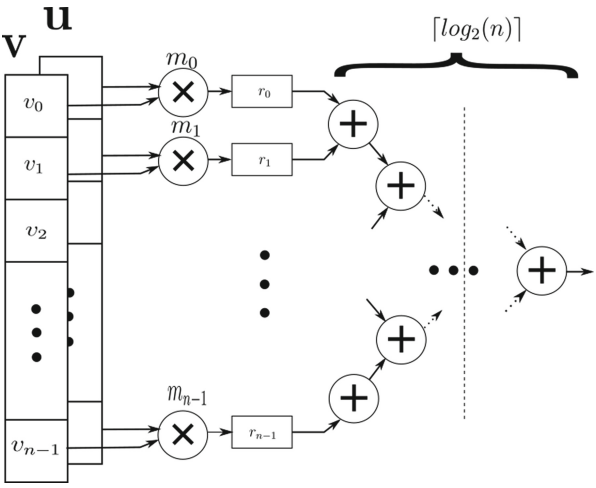

Fig. 1. Hardware module for the pipelined vector product computation. **v** and **u** are input vectors stored in registers.

so several conditions can be checked instead of performing a real division. The comparisons being made are listed in Table 1 and are easily executed in hardware by using simple shifts, additions, and subtractions.

Table 1. Conditions checked for the rounded division with the restricted result range

Division result	Value	Condition
$<0; 0.5)$	0	$2 \cdot \lvert dot \rvert < \lVert \mathbf{u} \rVert^2$
$<0.5; 1.5)$	1	$\lVert \mathbf{u} \rVert^2 \leq 2 \cdot \lvert dot \rvert < 3 \cdot \lVert \mathbf{u} \rVert^2$
$<1.5; 2.5)$	2	$3 \cdot \lVert \mathbf{u} \rVert^2 \leq 2 \cdot \lvert dot \rvert < 5 \cdot \lVert \mathbf{u} \rVert^2$
$<2.5; 3.5)$	3	$5 \cdot \lVert \mathbf{u} \rVert^2 \leq 2 \cdot \lvert dot \rvert < 7 \cdot \lVert \mathbf{u} \rVert^2$
$<3.5; 4.5)$	4	$7 \cdot \lVert \mathbf{u} \rVert^2 \leq 2 \cdot \lvert dot \rvert < 9 \cdot \lVert \mathbf{u} \rVert^2$

The full range of possible results is not covered. The selection of the results range is based on statistical data and the chosen assumption. Assuming that sampled vectors provided to the sieving algorithm are no longer than x times the approximate shortest vector, the rounded division will never generate a result bigger than x. Based on experiments, $x = 4$ is sufficient to accept all sampled vectors. The division in the selected range is necessary to make the comparison with CPU implementations more accurate and avoid rare events when the vector is required to be reduced again, which may lead to data desynchronization in the accelerator.

4.3 Update of Vector Values

Having all the necessary values, it is possible to update the reduced vector element and its norm. These two operations can be performed in parallel.

The element update function simply subtracts the product $q \cdot \mathbf{u}$ from $\mathbf{v}$. This operation can also be executed in parallel. In the first step, the products $q \cdot u_i$ are calculated. They are then subtracted from v_i in the second step. Each step is executed in a separate clock cycle to decrease the critical path's length and obtain a higher maximum clock frequency.

In hardware, the norm update function is executed in three steps, taking one clock cycle each. At first, $2 \cdot dot$ and $q \cdot \|\mathbf{u}\|$ are computed. Secondly, the multiplication by q is applied to both partial results. At the end, the subtraction and addition operations are performed.

4.4 Reduce Module

The described above parts were used to develop the entire *Reduce* algorithm. The hardware block diagram combining previously discussed steps is shown in Fig. 2.

With additional shift registers required to store data for further steps of the algorithm, it is possible to start computations for a new vector pair in each clock cycle, utilizing pipeline properties of used building blocks, and increasing the total performance.

The latency for one pair of vectors depends only on the dimension of a lattice. For an n-dimensional lattice, the latency $f_{cl}(n)$ equals exactly

$$f_{cl}(n) = \left\lceil \frac{\log_2(n)}{\beta} \right\rceil + 5 \tag{5}$$

clock cycles. This is also the number of pairs of vectors being processed in the module concurrently. Therefore, for 200 MHz clock frequency, the pipelined version can perform up to 200,000,000 vector reductions per second, and the branched version can perform twice as many reductions. This calculation does not include the communication overhead, so the practical performance for a standard approach will be lower.

5 Caching Approach to Lattice Sieving for Multi-platform Environment

In the previous section, the hardware accelerator for the vector reduction was introduced. The next step is to develop a way of using this accelerator to increase the performance of any sieve. Due to the communication overhead, a simple call to the accelerator for every occurrence of the reduce operation will not give any speedup. The data transfer takes almost 90% of the total execution time, and the performance is lower than on standard CPU. Moreover, it is not possible to run the entire algorithm on an FPGA due to its lack of sufficiently large memory to

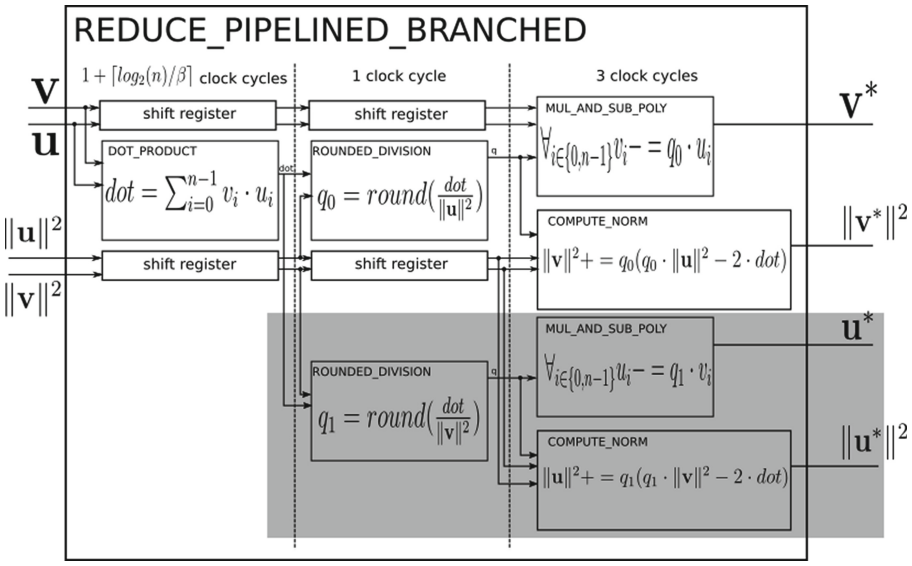

Fig. 2. The architecture of the reduce accelerator with pipelining and branching. The shaded part denotes logic implemented in the branched version and omitted in the standard implementation.

perform standalone sieving on FPGAs. Thus, in this section, a caching approach for lattice sieving algorithms in a multi-platform environment is presented. Our modification allows eliminating the communication delays, omitting the memory limitations, and fully utilizing the proposed parallel architecture for lattice sieving by combining previously reported methods with caching techniques. The proposed techniques will also work for other kinds of sieves.

A software/hardware approach is considered, where only a part of computations is performed in FPGAs, the rest of an algorithm is executed on CPU, and the majority of necessary data is stored on CPU. Currently, there are several approaches to combining CPUs with FPGAs. Thus, the calculations are not focused on any particular solution, but rather on a universal approach, applicable to each practical realization of the system combining both device types.

5.1 Data Transfer Costs

The biggest issue with current algorithms is the data transfer cost. Even the largest FPGAs are not able to store all required data to run sieving standalone for currently attacked dimensions. Thus, only a hybrid solution is considered. However, with only a part of the algorithm being executed on the FPGA side, some data is required to be exchanged between both sides. In lattice sieving, the transferred data will consist mostly of lattice points. For a simple vector expressing the lattice point, its size depends on the dimension of the lattice.

In the presented accelerator, each vector element is stored in 16 bits. It can be extended to 32 bits if needed, but due to our experiments on reduced lattices, 16 bits is sufficient. Additionally, the squared value of a vector length is also stored in another 32 bits. Thus, the number of bits $f_{nb}(n)$ required for a simple n-dimensional vector is expressed as:

$$f_{nb}(n) = n \cdot 16 + 32 = (n + 2) \cdot 16 \tag{6}$$

This number also matches the number of bits required for one vector transfer in any direction between CPU and FPGA. The communication time depends on the size of data and on the width of a data bus. The commonly used data buses are $w = \{32, 64, 128, 256\}$-bits wide and are able to deliver a new data in every FPGA clock cycle. The data transfer latency f_{tl} for one vector is expressed as the number of clock cycles and can be obtained from the equation:

$$f_{tl}(n, w) = \left\lceil \frac{f_{nb}(n)}{w} \right\rceil = \left\lceil \frac{(n + 2) \cdot 16}{w} \right\rceil \tag{7}$$

5.2 Reducing Newly Sampled Vectors by a Set

In large lattice dimensions, the total required memory is significantly larger than the memory available in any FPGA device. For dimensions larger than 85, the accelerator must cooperate with CPU during the reduction of the newly sampled vectors due to memory limitations. The vectors will be processed in smaller sets, and an efficient way to manage the data transfer is required.

In this approach, every new vector is used for reduction at least $2 \cdot |L|$ times. Assuming that the set L is going to be divided into smaller sets L_i, capable of fitting in FPGA memory, the data transfer costs may reduce or even eliminate the acquired acceleration.

In the most basic approach, the newly sampled vector $\mathbf{v}_n$ is reduced by the set L_i that fits FPGA's memory. In the first step, $\mathbf{v}_n$ is reduced sequentially by elements from L_i, while the reduction of elements from L_i by $\mathbf{v}_n$ in the second step is executed in parallel. Elements used in the first step of the reduction are replaced by other elements from L. This approach is not efficient due to the data transfer requirements, and several changes must be made to achieve the best performance.

5.3 On-the-Fly Reduction

It is not necessary to wait until all data is available on the FPGA side. The designed algorithm should take advantage of the fact that reductions may start right after sending the first two vectors. Every new vector will be reduced by those transferred so far, and the communication will happen in the background. This approach will allow to reduce the combined time of computations and data transfers.

The gains from the on-the-fly reduction depend on the approach for sieving. Applying ideas from Bos et al. [5] or Milde and Schneider [12] will require a

different data transfer schedule and will be affected differently by the continuous memory transfer. The ideal algorithm should allow avoiding any data transfer costs.

5.4 Maximizing Performance with the Proper Schedule of Operations

To efficiently accelerate any sieving with FPGAs (or any other devices), the aforementioned elements must be included in the algorithm's design.

Taking ideas from literature for parallel sieve, let us divide the *GaussSieve* execution into three parts, as proposed by Ishiguro et al. [8] and extend it to meet our requirements.

The algorithm will operate on a set S of newly sampled vectors, instead of only one vector. The first part is the reduction of the set S by already reduced vectors in the list L. A data transfer latency for one lattice vector depends on the lattice dimensions and the data bus width w, as shown in Eq. 7. Thus, to avoid the data transfer overhead, one reduced lattice vector should be processed during the exact time required for a new one to be transferred. This can be done by extending the size of the set S from one to $k = f_{tl}(n, w)$. Then, taking into account the pipelining capabilities of the reduce function accelerator, during the first reduction, after k clock cycles, the accelerator should be able to start processing a new vector. The algorithm can take advantage of the pipelined execution of instructions due to the lack of any data dependency between vectors from the set S and an already used vector from the list L. The state of registers during the first step of sieving is visualized in Fig. 3, 4 and 5. The number of reductions in the first step is equal to $k \cdot |L|$, and this is also the number of clock cycles spent on computations. The communication cost will include only sending first $k+1$ vectors, where the remaining vectors will be transferred during computations. The FPGA latency will be then:

$$f_{el}^{p1}(n, w) = k^2 + k \cdot |L| + f_{cl}(n) \tag{8}$$

In the second step, elements from the set S are reduced by themselves. The set is already in FPGA memory, so there is no transfer overhead. The accelerator is going to execute the normal *GaussSieve* algorithm. Without the transfer overhead, the latency of computations is

$$f_{el}^{p2} = \frac{k^2}{2} \cdot f_{cl}(n) + \frac{k^2}{2} + f_{cl}(n) \tag{9}$$

During the second stage computations, a new batch S' of sampled vectors can be transferred to FPGA. The number of clock cycles required to transfer a new data consisting of k vectors is expressed as $k \cdot f_{tl}(n, w) = f_{tl}(n, w)^2 = k^2$, and is smaller than the computational latency of the second step.

In the last step, all vectors from the list L are reduced by vectors from the set S, which is already in the FPGA memory. Each vector $\mathbf{v} \in L$ is going to be reduced by k vectors, and reductions may be performed in parallel.

Again, there is no communication overhead. If any reduction occurs, the lattice vector is transferred back to CPU in the background. Otherwise if no reduction happens, there is no need for moving a given vector back to CPU. The latency of the third step is then

$$f_{el}^{p3} = k \cdot |L| + f_{cl}(n) \tag{10}$$

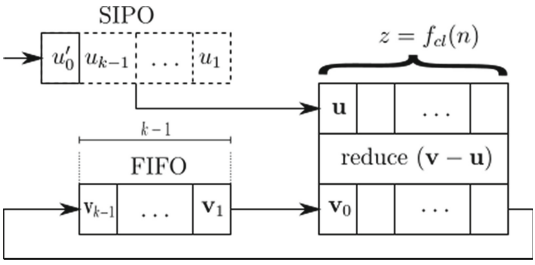

Fig. 3. The state of the accelerator and registers after the first clock cycle. The first part of the next vector $\mathbf{u}'$ is loaded, while the remaining parts of the SIPO unit contain parts of the previously loaded $\mathbf{u}$. The first vector $\mathbf{v}_0$ from the internal set is delivered to the reduce module to be reduced by the vector $\mathbf{u}$. FIFO contains $k-1$ elements and is smaller by one element than the SIPO unit.

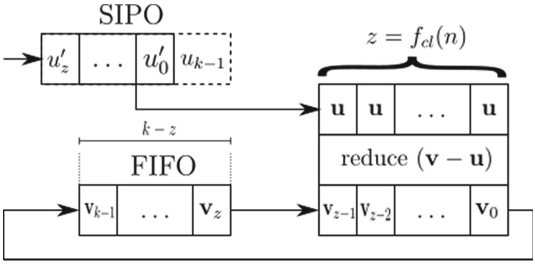

Fig. 4. The state of the accelerator and registers after $z = f_{cl}(n)$ clock cycles. The reduction of the first vector is finished and the vector $\mathbf{v}_0$ is going to be put in the FIFO queue. In every part of the reduce module, the same vector $\mathbf{u}$ is used for reduction. Only z parts from k parts of the new vector $\mathbf{u}'$ had been transferred so far. The FIFO queue contains $k - z$ elements.

By adding all the three steps together, it is possible to compute the latency of adding k new vectors to the list L of already Gauss-reduced vectors. The latency for the first execution will be then

$$f_{el}(n, w) = k^2 + 2 \cdot k \cdot |L| + \frac{k^2}{2} \cdot f_{cl}(n) + k^2 + 3 \cdot f_{cl}(n) \tag{11}$$

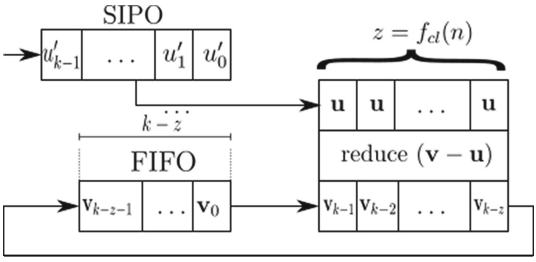

Fig. 5. The state of the accelerator and registers after k clock cycles. The entire new vector $\mathbf{u}'$ is on the FPGA side and will be used in the next run. Only v_i remaining in the `reduce` module had not been reduced by u so far and will be reduced in the next z clock cycles.

As the new batch S' of sampled vectors is transferred during the second step, for every next execution, the cost of data transfer can be omitted and then the final latency becomes:

$$f_{el}(n, w) = k^2 + 2 \cdot k \cdot |L| + \frac{k^2}{2} \cdot f_{cl}(n) + 3 \cdot f_{cl}(n) \tag{12}$$

Compared to CPU, the expected acceleration can be computed as

$$A = \frac{2 \cdot |L| + k}{P_{CPU}(n)} \cdot \frac{H}{k + 2 \cdot |L| + \frac{k}{2} \cdot f_{cl}(n) + \frac{3 \cdot f_{cl}(n)}{k}} \tag{13}$$

where the $P_{CPU}(n)$ is the performance of CPU for n dimensional lattice, expressed as the maximum number of reduce operations per second, and H is the maximum clock frequency of the hardware accelerator.

To determine the acceleration for the targeted platforms, we first measured the performance of software implementation. We took advantage of the *fplll* library, that was used as a basis of *g6k* code for computing the best result in the TU Darmstadt SVP Challenge (as of July 2020, dimensions from 153 to 170). Thus, the sieving operations implemented in *fplll* were used for constructing experiments aimed at measuring performance of software sieving.

In the designed experiment, a large set of vectors is sampled and pairwise reduced. Only the reduction time is measured. The size of the set is large enough to exceed the processor's cache memory, which allows us to measure the performance in a real scenario.

For the experiments, an Amazon Web Service `c5n.18xlarge` instance equipped with 72-core, a 3.0 GHz Intel Xeon Platinum processor was used.

In Fig. 6, an expected acceleration for the targeted lattice dimensions is presented, as the combined visualisation of Eq. 13 and obtained experimental data. For dimensions being currently considered in the SVP challenge (dimensions between 155 and 160), the expected acceleration from the proposed FPGA accelerator, compared to one CPU core, is around 30x. As for FPGA, clock frequency

was set to 200 MHz, and in our algorithm, there is no visible difference between the considered data bus widths. The number of elements in L was set to 10,000.

The accelerator almost always performs the pipelined vector reduction. Only in a small part of the second stage, the reductions are not pipelined. The communication bottleneck has been completely eliminated. Almost the maximum theoretical performance of the proposed accelerator (i.e., the performance without taking into account the communication overhead) has been achieved with this approach. Moreover, the accelerator can be adapted to other parallel sieves and work with other devices. It is possible to use the proposed accelerator in the parallel implementation of *g6k* as one of the devices performing the basic step of sieving, a vector reduction.

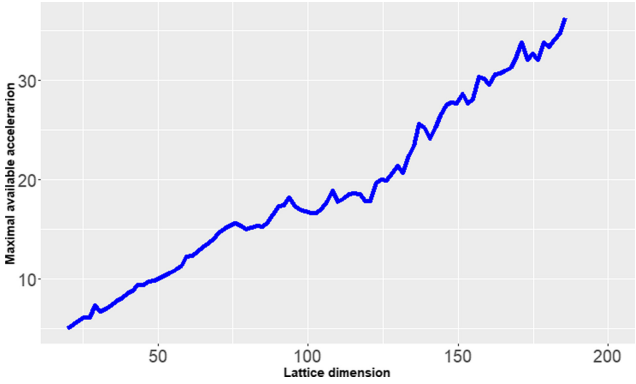

Fig. 6. Expected acceleration offered by one accelerator deployed on FPGA.

Some of the algorithms (e.g., [5]) allow immediately reducing both processed vectors ($\mathbf{v}-\mathbf{u}$ and $\mathbf{u}-\mathbf{v}$) in the next consecutive steps. In that case, a branched version of the accelerator can be used. Then, the third scenario changes, reducing the execution only to the two first steps. In the branched version, the first and the last steps are computed at a time. Thus, the total acceleration can increase by around 1.5 times.

6 Multiple Parallel Instances of the Accelerator in One FPGA

During the second stage, a new batch of sampled vectors is transferred to an FPGA. However, communication requires less time than the computations during the second stage. After all k vectors are transmitted, the data bus waits unused until the third stage of the algorithm. The clock latency for the data transfer $C_T(n)$ is k^2 cycles, whereas the computational latency $C_C(n)$ is $k^2 \cdot f_{cl}(n) + k^2 + f_{cl}(n))$ cycles. Then, the ratio $\frac{C_C(n)}{C_T(n)}$ indicates how many times

the computations are longer. A difference between the two times can be used to send several other sets S to other accelerators implemented in the same FPGA. The maximum number of accelerators working in parallel is expressed then as:

$$\begin{aligned} \frac{C_C(n)}{C_T(n)} &= \frac{k^2 \cdot f_{cl}(n) + k^2 + f_{cl}(n)}{k^2} \\ &= f_{cl}(n) + 1 + \frac{f_{cl}(n)}{k^2} \\ &\approx f_{cl}(n) + 1 \end{aligned} \tag{14}$$

The term $\frac{f_{cl}(n)}{k^2}$ for large dimensions is always lower than 1 and can be omitted. Then, for the targeted dimensions, $n > 64$, the computations are $f_{cl}(n) + 1$ times longer than the communication. This number is also the maximum number of accelerators working in parallel with the full performance each. It is possible to connect more accelerator instances, but some of them will have to wait until all new sets S' are transferred. The other way to maximize the performance and avoid data transfer bottlenecks is to extend the computation latency f_{cl}.

In Fig. 7, a schedule representing the execution of the algorithm using several accelerators working in parallel is presented. The number of accelerators is denoted as σ, and every accelerator is denoted as A_i. The execution of the first and the second stage starts at the same time in every accelerator. Vectors used for reduction are everywhere the same. The last stage differs. Every next accelerator starts sieving $k + f_{cl}(n)$ clock cycles after the previous one and takes as an input an output vector from the previous unit. The $k + f_{cl}(n)$ clock cycles are required for processing the first vector by an accelerator and pushing it further. The last accelerator starts working $(\sigma - 1) \cdot (k + f_{cl}(n))$ clock cycles after the first one. This approach is different from the first stage because the reduction $v_i - u_j$ is performed instead of $u_j - v_i$, where vectors v_i are loaded from CPU.

The inbound transmission is divided into two parts. In the first and the last stage, the previously reduced vectors, marked by small rectangles, are transferred to FPGA. In the second stage, a new batch of sampled vectors divided into σ sets, marked as a bigger rectangle, is transferred.

The outbound transmission starts in the third stage and can proceed until the end of the second stage. At first, the Gauss-reduced sets S_i of vectors from the second stage are sent back to CPU. As for results from the third stage, only shortened vectors are pushed back to CPU. It is hard to estimate the number of shortened vectors. To avoid data loses, the output FIFO queue should have a large enough memory available. Fortunately, there is only one FIFO queue, receiving data from the last accelerator.

The performance of multiple accelerators, implemented in one FPGA, scales with their number. For σ accelerators, the performance will be $\approx \sigma$ times higher compared to a single one. The acceleration is not exactly σ times better due to the $(\sigma - 1) \cdot (k + f_{cl}(n))$ clock cycles delay for the last module and proportionally less for other modules in the third step.

After the entire vector is transferred, this vector may be saved to one of the internal FIFO queues, currently not used by the reduce module, or directly

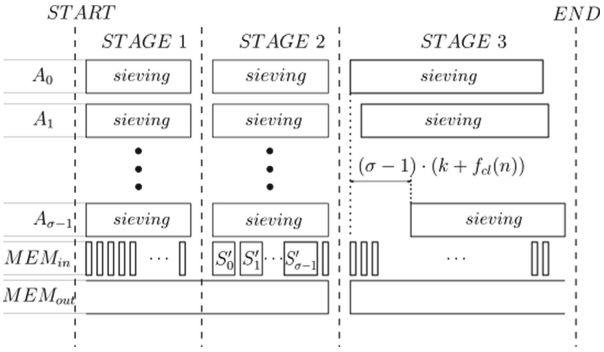

Fig. 7. The activity diagram for multiple accelerators in one FPGA. A_i denotes an i-th instance of the accelerator in FPGA, and σ is the number of accelerators. k is the size of sampled vectors sets S and S'. S is currently used, where S' will be used in the next run. $f_{cl}(n)$ is a function representing the reduction latency for n-dimensional vectors.

provided as one of the input vectors for reduction. Vectors from CPU are saved in queues only in the second stage of the algorithm. When the reduction is applied, there are two options. In the first and the second stage, vectors are always written back to the currently used internal queue. In the second and the third stage, vectors that were shortened during the reduction are placed in the output FIFO queue. In the second step, all vectors are transferred back to CPU, but in the third stage, only the reduced vectors are moved back. If vectors stay the same (i.e., there was no reduction), then they are overwritten in FPGA.

A multi-core version consists of several instances connected into a chain of accelerators. An output from one element is connected to the input of the next element. The last instance is responsible for the data transfer back to CPU.

6.1 Final Results

To measure the highest possible acceleration, we tried to fit as many instances as possible in one FPGA. The final number for 160-dimensional lattices is 20 accelerators working in parallel. This number is bigger than the boundary in Eq. 14, so we extended the f_{cl} latency to 12 to achieve higher clock frequency (150 MHz) and this way mitigated the second stage data transfer bottleneck. The design was described in the VHDL language and verified in simulation. Our code passes all stages of the FPGA design process. However, the actual run would be too long to be attempted with the current equipment and algorithm for a 160-dimensional lattice. Then, we used a proposed in the literature method for comparing cross-platform implementations [10], and we cost-compared our estimated results using two Amazon AWS instances: *f1.2xlarge* equipped with Xilinx FPGAs and *c5.18xlarge* aforementioned Intel Xeon. The results are presented in Table 2. The FPGA-based AWS instance can solve an equivalent problem for only 6% of the CPU-based instance price.

Table 2. The normalized cost comparison for GaussSieve executed on CPU and FPGAs. The performance of one core is used as a reference value to compute the acceleration for multiple cores. The total acceleration refers to the acceleration obtained by fully utilizing a device, and it denotes a number of cores multiplied by their acceleration, which is equivalent to the number of CPU cores that matches the same performance. The normalized acceleration compares FPGA designs to a multi-core CPU. The price per acceleration is in row E. This price is compared to the price for CPU in row F.

No	Device	CPU	FPGA
A	# of cores	72	20
B	Acceleration per core	1	30
C	Total acceleration (A · B)	72	600
D	Normalized acceleration	1	8.32
E	AWS price ($/h)	3.05	1.65
F	Price per acceleration (E/D)	3.05	0.20
G	Compared to CPU (F/F.CPU)	1	0.06

6.2 Comparison to Other Results

It is hard to compare cross-platform implementations. Looking only at the performance, the presented implementation achieves more than 8x speed-up compared to a 72-core CPU for a 160-dimensional lattice, so the implementation has the performance of around 576 CPU cores. The [13] achieved 21.5x acceleration for a 96-dimensional lattice when compared to [8] (2x CPUs with 8 cores), so it has the performance of around 344 cores (in a lower dimension). And the cost of power consumption will be very likely probably lower for FPGA when compared to GPU.

7 Conclusions

This paper introduces a new approach to lattice sieving by using a massively parallel FPGA design to accelerate the most common operation in every lattice sieving algorithm – vector reduction. As an example, the *GaussSieve* algorithm was accelerated. The acceleration is possible only with the proposed modification to parallel versions of sieving algorithms. The modification is devoted to eliminating the communication overhead between the specialized circuit, implemented in FPGA, and the CPU, running the rest of the algorithm, by using a caching strategy. The acceleration depends on the lattice dimension and increases linearly as a function of that dimension. For the targeted 160-dimensional lattice, the proposed solution is estimated to achieve 8.32 better performance compared to CPU. The results were obtained from FPGA simulation and CPU experiments. Comparing the cost of solving the SVP problem in AWS, the presented architecture will require only 6% of the CPU-based costs. Our project is also the first attempt reported to date to accelerate lattice sieving with specialized hardware.

The proposed hardware accelerator can be used directly for almost any lattice sieve performing a vector reduction operation. In this paper, the GaussSieve algorithm was investigated as an example algorithm. The parallel hardware architecture with the proposed caching strategy can be adapted to other GaussSieve modifications reported in the literature [5,8,12], as well as for other lattice sieving algorithms with a better complexity. As a part of future work, the adoption of the presented solution to algorithms other than GaussSieve will be explored. Additionally, an application of the proposed solution to other algorithms hard to implement in FPGAs due to the communication and memory bottleneck will be investigated.

References

1. SVP Challenge. https://www.latticechallenge.org/svp-challenge/
2. Ajtai, M., Kumar, R., Sivakumar, D.: An overview of the sieve algorithm for the shortest lattice vector problem. In: Silverman, J.H. (ed.) CaLC 2001. LNCS, vol. 2146, pp. 1–3. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44670-2_1
3. Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing, STOC 2001, pp. 601–610. ACM Press (2001). https://doi.org/10.1145/380752.380857. http://portal.acm.org/citation.cfm?doid=380752.380857
4. Albrecht, M.R., Ducas, L., Herold, G., Kirshanova, E., Postlethwaite, E.W., Stevens, M.: The general sieve kernel and new records in lattice reduction. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11477, pp. 717–746. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17656-3_25
5. Bos, J.W., Naehrig, M., van de Pol, J.: Sieving for shortest vectors in ideal lattices: a practical perspective. Int. J. Appl. Cryptol. **3**(4), 313–329 (2017). https://doi.org/10.1504/IJACT.2017.089353
6. Detrey, J., Hanrot, G., Pujol, X., Stehlé, D.: Accelerating lattice reduction with FPGAs. In: Abdalla, M., Barreto, P.S.L.M. (eds.) LATINCRYPT 2010. LNCS, vol. 6212, pp. 124–143. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14712-8_8
7. Gentry, C., Peikert, C., Vaikuntanathan, V.: Trapdoors for hard lattices and new cryptographic constructions. In: Proceedings of the Fourtieth Annual ACM Symposium on Theory of Computing, STOC 2008, p. 197. ACM Press (2008). https://doi.org/10.1145/1374376.1374407. http://dl.acm.org/citation.cfm?doid=1374376.1374407
8. Ishiguro, T., Kiyomoto, S., Miyake, Y., Takagi, T.: Parallel gauss sieve algorithm: solving the SVP challenge over a 128-dimensional ideal lattice. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 411–428. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54631-0_24
9. Klein, P.: Finding the closest lattice vector when it's unusually close. In: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2000, pp. 937–941. Society for Industrial and Applied Mathematics, USA (2000)

10. Kuo, P.-C., et al.: Extreme enumeration on GPU and in clouds. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 176–191. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23951-9_12
11. Micciancio, D., Voulgaris, P.: Faster exponential time algorithms for the shortest vector problem. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1468–1480. Society for Industrial and Applied Mathematics. https://epubs.siam.org/doi/10.1137/1.9781611973075.119
12. Milde, B., Schneider, M.: A parallel implementation of GaussSieve for the shortest vector problem in lattices. In: Malyshkin, V. (ed.) PaCT 2011. LNCS, vol. 6873, pp. 452–458. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23178-0_40
13. Yang, S.-Y., Kuo, P.-C., Yang, B.-Y., Cheng, C.-M.: Gauss Sieve algorithm on GPUs. In: Handschuh, H. (ed.) CT-RSA 2017. LNCS, vol. 10159, pp. 39–57. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52153-4_3