

Article

Optimizing Lattice Basis Reduction Algorithm on ARM V8 Processors

Ronghui Cao ¹, Julong Wang ¹, Liming Zheng ² , Jincheng Zhou ¹, Haodong Wang ¹, Tiaojie Xiao ³ and Chunye Gong ^{3,4,*}

¹ School of Computer and Communication Engineering, Changsha University of Science and Technology, Changsha 410073, China; caoronghui@hnu.edu.cn (R.C.); 0122wjl@gmail.com (J.W.); 22108041641@stu.csust.edu.cn (J.Z.); wang_tx0010@163.com (H.W.)

² College of Electronic Science, National University of Defense Technology, Changsha 410073, China; lmzheng@nudt.edu.cn

³ College of Computing, National University of Defense Technology, Changsha 410073, China; xiaotiaojie@nudt.edu.cn

⁴ National Supercomputer Center in Tianjin, Tianjin 300457, China

* Correspondence: gongchunye@nudt.edu.cn

Abstract: The LLL (Lenstra–Lenstra–Lovász) algorithm is an important method for lattice basis reduction and has broad applications in computer algebra, cryptography, number theory, and combinatorial optimization. However, current LLL algorithms face challenges such as inadequate adaptation to domestic supercomputers and low efficiency. To enhance the efficiency of the LLL algorithm in practical applications, this research focuses on parallel optimization of the LLL_FP (LLL double-precision floating-point type) algorithm from the NTL library on the domestic Tianhe supercomputer using the Phytium ARM V8 processor. The optimization begins with the vectorization of the Gram–Schmidt coefficient calculation and row transformation using the SIMD instruction set of the Phytium chip, which significantly improve computational efficiency. Further assembly-level optimization fully utilizes the low-level instructions of the Phytium processor, and this increases execution speed. In terms of memory access, data prefetch techniques were then employed to load necessary data in advance before computation. This will reduce cache misses and accelerate data processing. To further enhance performance, loop unrolling was applied to the core loop, which allows more operations per loop iteration. Experimental results show that the optimized LLL_FP algorithm achieves up to a 42% performance improvement, with a minimum improvement of 34% and an average improvement of 38% in single-core efficiency compared to the serial LLL_FP algorithm. This study provides a more efficient solution for large-scale lattice basis reduction and demonstrates the potential of the LLL algorithm in ARM V8 high-performance computing environments.

Keywords: lattice reduction; parallel optimization; Tianhe; ARM V8



Academic Editor: Christos Bouras

Received: 5 January 2025

Revised: 31 January 2025

Accepted: 5 February 2025

Published: 14 February 2025

Citation: Cao, R.; Wang, J.; Zheng, L.; Zhou, J.; Wang, H.; Xiao, T.; Gong, C. Optimizing Lattice Basis Reduction Algorithm on ARM V8 Processors. *Appl. Sci.* **2025**, *15*, 2021. <https://doi.org/10.3390/app15042021>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The LLL algorithm, proposed by Arjen Lenstra, Hendrik Lenstra, and László Lovász [1], is a landmark algorithm in computational mathematics and cryptography. The emergence of the LLL algorithm, along with Ajtai's pioneering work in 1996 [2], laid a solid foundation for the development of lattice cryptography. In recent years, Albrecht et al. [3] and Coppersmith et al. [4] have further emphasized the importance of in-depth studies of the LLL algorithm for the security assessment of modern cryptosystems. In the

fields of machine learning [5] and quantum computing [6], the LLL algorithm has also made significant contributions.

In the field of communication and signal processing, Hassibi and Vikalo [7] applied the LLL algorithm to the decoding of MIMO systems [8], significantly improving the performance of wireless communication systems. In the integer programming problem, Lovász and Scarf [9] demonstrated the functionality of the LLL algorithm. Dadush et al. [10] further extended its application in convex optimization problems, broadening the algorithm's application prospects in operations research. However, most of these applications use the basic functions of the LLL algorithm to solve application problems, without involving the different orthogonalization methods of the LLL algorithm or optimizing the algorithm itself. These studies present an important research direction for further improvement.

Many variants of the LLL algorithm have been developed, such as BKZ [11,12], progressive BKZ [13], L^2 [14], and other innovative algorithms [15]. These algorithms improve performance through various orthogonalisation methods, heuristic pruning, and block strategy techniques. Floating-point types are more suitable for large-scale computations, but rounding errors can affect the accuracy of high-dimensional lattices. The researchers solve this problem by using high-precision floating point, adaptive rounding, and multiple-precision libraries. Proposed Jacobi lattice-based statute algorithms [16,17] and greedy LLL algorithms [18] promote algorithm adaptation through orthogonalisation methods tightly coupled with architecture and floating-point-type optimization. Due to hardware platform differences, some foreign numerical libraries and optimized instruction sets are difficult to fully adapt. The Phytium 2000+ is a high-performance processor based on the ARMv8 architecture with a 64-bit design, used primarily in high-performance computing and servers. It offers strong computing power, multi-core support, and efficient memory management, while being compatible with both 32-bit and 64-bit ARM applications. The algorithm performance can be improved by using the SIMD instruction set and low-level assembly of domestic platforms [19]. It is important to conduct research on customizing LLL algorithms based on the domestic Tianhe supercomputer. Recent studies have explored optimizations specific to the ARM V8 architecture, focusing on SIMD instruction sets, vectorization, and multi-core parallelism, which significantly improve computational efficiency in lattice-based algorithms [20]. In comparison, optimization efforts for x86 architectures, which focus on leveraging advanced vector instructions and cache management, have been widely studied [21]. Similarly, GPU-based approaches for lattice algorithms emphasize the parallel processing capabilities of GPUs, offering substantial speedup for large-scale lattice problems [22].

Based on the above background, this study focuses on the LLL algorithm based on double-precision-type Gram–Schmidt orthogonalisation in the NTL library. The main innovations of this paper include the following: SIMD vectorization and assembly optimization of the Gram–Schmidt orthogonalisation method, the row transformation part of the LLL_FP algorithm for double-precision floating point types on the architecture of the Phytium processor. In terms of memory access, optimization techniques such as data prefetch and loop unrolling [23] are used to multiplex the registers, which further improves the performance of the algorithm significantly. These optimization strategies improve the efficiency of the algorithm by up to 42% in specific cases, and provide a solid foundation for the applications of the LLL algorithm in high-performance computing environments.

This paper is organized as follows: Section 2 introduces the technical background of the LLL algorithm and related algorithmic theories. Section 3 analyses the structure of algorithm in detail and describes the optimization strategy used. Section 4 presents the experimental results and provides an in-depth analysis of the performance of the optimized

algorithm. Section 5 offers a discussion on how the presented methods could be adapted to future architectures. Section 6 concludes the research results.

2. Technical Background

2.1. Foundation of Lattice

Let n be a positive integer and the lattice L be a discrete, additive subgroup of $\mathbb{R}^n$, defined as

$$L = \{Bz\} \quad (1)$$

where z is an n -dimensional integer vector and B is an $m \times n$ ($m \geq n$) matrix, called a lattice generating matrix.

Let $B = [b_1, b_2, \dots, b_n]$, where $b_1, b_2, \dots, b_n$ are linearly independent columns. They form the basis of L . For example, matrix

$$B = \begin{bmatrix} b_1 & b_2 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 1 & 0 \end{bmatrix} \quad (2)$$

A lattice can have many lattice bases. For example, matrix

$$C = \begin{bmatrix} c_1 & c_2 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ -1 & 1 \end{bmatrix} \quad (3)$$

Since the lattice L can have multiple bases, some bases are preferable to others. We can expect the short basis vectors to be nearly orthogonal. c_1 and c_2 are shorter basis vectors than b_1 and b_2 , and they are more orthogonal than b_1 and b_2 . The short basis is called a reduced basis.

2.2. The Gram-Schmidt Process

Gram-Schmidt orthogonality is a method for constructing a set of orthogonal vectors from a set of linearly independent vectors. It iteratively builds vectors that are orthogonal to all the vectors that have already been constructed, as follows:

$$\begin{aligned} a_1^* &= a_1, \\ a_2^* &= a_2 - \frac{a_2^T a_1^*}{(a_1^*)^T a_1^*} a_1^*, \\ &\vdots \\ a_n^* &= a_n - \frac{a_n^T a_{n-1}^*}{(a_{n-1}^*)^T a_{n-1}^*} a_{n-1}^* - \dots - \frac{a_n^T a_1^*}{(a_1^*)^T a_1^*} a_1^*. \end{aligned} \quad (4)$$

2.3. LLL Algorithm

The conventional LLL algorithm first uses Gram-Schmidt orthogonality with an $m \times n$ matrix B :

$$\begin{aligned}
 B &= [b_1 \ \cdots \ b_n] \\
 &= [\tilde{b}_1 \ \cdots \ \tilde{b}_n] \begin{bmatrix} 1 & \cdots & u_{1,n} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix} \\
 &= \begin{bmatrix} \frac{\tilde{b}_1}{\|\tilde{b}_1\|_2} & \cdots & \frac{\tilde{b}_n}{\|\tilde{b}_n\|_2} \end{bmatrix} \begin{bmatrix} \|\tilde{b}_1\|_2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \|\tilde{b}_n\|_2 \end{bmatrix} \begin{bmatrix} 1 & \cdots & u_{1,n} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix} \\
 &= QDU,
 \end{aligned} \tag{5}$$

where Q is an orthogonal matrix, D is a diagonal matrix, and $U = [u_{i,j}]$ is an upper triangular matrix with a unit diagonal. If the decomposition satisfies the two convention conditions

$$u_{i,j} \leq \frac{1}{2}, \quad 1 \leq i < j \leq n \tag{6}$$

$$w\|\tilde{b}_{i-1}\|_2^2 \geq \|\tilde{b}_i\|_2^2 + u_{i-1,i}^2\|\tilde{b}_{i-1}\|_2^2, \quad 2 \leq i \leq n \tag{7}$$

where $1/4 < w < 1$, then the basis formed by the columns of B is called a reduced basis.

2.4. LLL_FP Algorithm

NTL (Number Theory Library) is a C++ number theory library for efficiently working with integers and finite fields. The NTL library when it comes to the LLL algorithm has different types of variants like FP(double), QP(quad_float), XD(xdouble), and an RR type. These are all variants of the original LLL algorithm. FP stands for double precision. This paper chooses to optimize the LLL_FP algorithm, which can be used in conjunction with optimization methods such as SIMD vectorisation to speed up the algorithm with a small loss in accuracy.

The computational kernel of the LLL_FP algorithm consists of depth insertion, row transformation, and Gram–Schmidt coefficient computation. The Gram–Schmidt computation consists of Kahan summation [24] and inner product computation.

The Deep Insertion strategy improves the selection of row swaps in the LLL_FP algorithm. The algorithm not only checks whether the length ratio of the current row to the immediately preceding row vector satisfies the required orthogonality or normalization conditions, but also advances several rows to find the optimal insertion point for better control of the length ratio.

The LLL_FP algorithm reduces the basis vectors by progressively making the matrix satisfy the LLL condition through matrix row transformations. Each iteration of the algorithm determines the need for row transformations based on the value of the Gram–Schmidt coefficient. If the coefficient exceeds a threshold, the relevant rows are adjusted to bring the matrix closer to the ideal state. The row transformation is both the core operation of adjusting the matrix and the basis for determining row swaps or insertions.

The Kahan Summation algorithm is an algorithm used to reduce the rounding error in summation operations. It is mainly used for summing a series of floating-point numbers to prevent the accumulation of errors caused by the limitations of floating point precision in the summation operation. In standard floating point summations, accumulated rounding errors can result in a loss of precision. Kahan summation reduces the effect of cumulative error by using an additional variable, called a compensator, to track the partial error, and incorporate it into the next addition operation. During each addition operation, the current

value is added to the compensator, and the result is stored as a new compensator. Then, the compensator is added to the next value, and the compensator is updated. Finally, all the values are summed, and the final compensator is added to obtain the final summation result. The Kahan algorithm effectively reduces the error of the sum result caused by rounding errors by mitigating the sum error in each iteration. This makes the algorithm more accurate for calculating the sum of floating point numbers, especially when the number of accumulation operations is large or the precision of floating point numbers is low.

The Gram–Schmidt coefficient calculation is the core step of the Gram–Schmidt orthogonalisation (Algorithm 1), based on the Schmidt orthogonalisation formula and combined with Kahan summation to achieve error compensation. Its main function is to compute the orthogonal vectors for the k row of the input matrix B with guaranteed accuracy. Specifically, it calculates the inner product of the k row and the previous $k-1$ row, determines whether high accuracy is required based on the value of the inner product and the length of the vectors, accumulates the error to correct the inner product, generates the Gram–Schmidt coefficients, and updates the buffer to obtain the length of the k th row after orthogonalisation. By calculating the inner product and correcting the error, the procedure ensures that the generated set of orthogonal vectors satisfies orthogonality and length normality, thus realising the orthogonal basis transform of the vector space. This procedure is often called in the LLL_FP algorithm to maintain the orthogonal vector space structure, to support subsequent truncation and swapping operations, and finally, to correct the error by Kahan summation.

Algorithm 1 Calculation of Gram–Schmitt coefficient.

```

1: function COMPUTEGS( $B, B_1, \mu, b, c, k, bound, st, buf$ )
2:   Set number of columns  $n$  of  $B$ 
3:   if  $st < k$  then
4:     Initialize  $buf$ 
5:   end if
6:   for each row  $j = st$  to  $k - 1$  do
7:     InnerProduct  $s$ 
8:     if  $b[k]$  far greater than  $b[j]$  then
9:       if  $s$  is not satisfies precision then
10:        Precisely compute  $s$ 
11:      else
12:         $s$ 
13:      end if
14:    else
15:       $s$ 
16:    end if
17:    for summing terms do
18:      Update  $t1$ 
19:    end for
20:    Compute  $\mu[k][j]$ 
21:  end for
22:  if low precision then
23:    Use Kahan summation for  $c[k]$ 
24:  else
25:    Ordinary summation for  $c[k]$ 
26:  end if
27: end function

```

The parameters are the large integer-type lattice basis matrix B , the floating-point-type matrix B_1 , the floating-point-type array μ for the Gram–Schmidt orthogonalization

coefficients, the squared norm matrix b for the rows of $B1$, the squared norm matrix c for the Gram–Schmidt orthogonal vectors, the intermediate variable matrix buf , and $t1$ as an intermediate variable.

By combining the above steps, the flowchart of LLL_FP algorithm in Figure 1 and Algorithm 2 are obtained. In the algorithm flowchart, the steps of the algorithm start with the initialization of the variables, followed by the calculation of the Gram–Schmidt coefficient, reduction, deep insertion, and finally, line exchange and advancement. If the condition is met, it proceeds to the next iteration of the loop; if the condition is not met, the algorithm terminates.

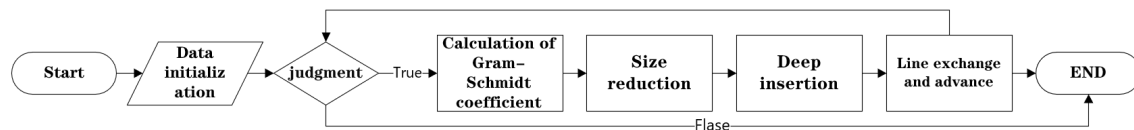


Figure 1. LLL_FP algorithm flow.

Algorithm 2 LLL_FP Algorithm

```

1: function LLL_FP
2:   Initialize loop parameters  $init\_k$  and  $rst$ 
3:   for  $k = init\_k \dots m$  do
4:     Calculation of Gram–Schmidt coefficient
5:     for  $j = rst - 1 \dots 1$  do
6:       if the condition (6) is not satisfied then
7:         Row transformation
8:       end if
9:     end for
10:    if deep insertion condition is met then
11:      Perform deep insertion
12:    end if
13:    if condition (7) is not satisfied then
14:      Perform row swap
15:    end if
16:  end for
17:  return  $m$ 
18: end function
  
```

In Algorithm 2, the process starts with initializing the loop parameters $init_k$ and rst in line 2. Gram–Schmidt coefficients are computed in a loop at line 4 for subsequent size reduction operations. This is followed by size reduction at lines 5 to 8, aiming to make each vector of the matrix smaller and orthogonal. Deep insertion occurs at lines 10 to 12. If it is found that the length of the current row is too large or does not meet the required orthogonality or normalization conditions, the row needs to be inserted into a more appropriate position. Finally, line exchange and advancement are carried out. These operations continue until the conditions fail to be satisfied, at which point the algorithm terminates.

3. Algorithm Analysis and Optimization

3.1. Algorithm Analysis

The LLL_FP algorithm of the NTL library is efficient but has weak parallelism and low memory usage. As shown in Figure 2, the Gram–Schmidt coefficient computation is a hot function of the LLL_FP algorithm according to the performance analysis tool. It needs to recompute the inner product of each vector with all previous vectors in each loop. This inner product overhead takes up a large amount of time with many loop iterations. It is worth noting that the LLL_FP algorithm determines which inter-row transformations are needed in each main loop iteration by calculating the Gram–Schmidt coefficient values. In some special cases, the row transformations may also become hotspot functions. Therefore, when faced with the original LLL_FP algorithm, SIMD vectorization and assembly optimization can be performed on these two parts under the Tianhe supercomputing architecture to increase parallelism. In terms of memory access, data prefetch, loop unrolling, and other techniques can be used to further improve the overall performance of the algorithm.

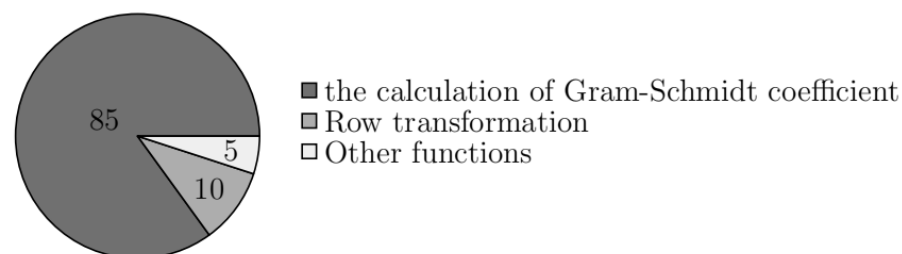


Figure 2. Proportion of function calls.

As parallel computing becomes more popular, especially for supercomputing platforms, tuning LLL_FP to better support multicore parallelism, GPU acceleration, and distributed computing can significantly increase algorithmic performance.

3.2. Simd Vectorization

SIMD (Single Instruction Multiple Data) is a computing technology that uses a single instruction to process multiple streams of data simultaneously. It can be understood as changing the original calculation of one value at a time to calculate multiple values in parallel. The Phytium 2000+ chip is based on the ARM V8 architecture and supports the NEON SIMD instruction set, which can process multiple data elements in parallel using 64-bit and 128-bit vector registers. This makes it suitable for the LLL_FP algorithm.

The inner product calculation in the iteration of the Gram–Schmidt coefficient calculation is an operation of significant importance. The Kahan summation is the key step in correcting each calculation error. Therefore, the focus of this paper is on vectorizing the inner product calculation and Kahan summation in each Gram–Schmidt coefficient calculation. In the case of cycle ranges that are powers of two, the inner product calculation and Kahan summation can be directly handled in parallel. However, for cycle ranges outside the power of two, ordinary serial computation is employed.

The optimization of Gram–Schmidt coefficient calculation is shown in Algorithm 3. SIMD vectorization was applied to the inner product computation in line 7, the Gram–Schmidt coefficient computation in line 20, and the Kahan computation in line 23.

Algorithm 3 Optimization of Gram–Schmidt coefficient calculation.

```

1: function COMPUTEGS(B, B1, mu, b, c, k, bound, st, buf)
2:   Set number of columns n of B
3:   if st < k then
4:     Initialize buf
5:   end if
6:   for each row j = st to k − 1 do
7:     SIMD vectorization InnerProduct s
8:     if b[k] far greater than b[j] then
9:       if s does not satisfy precision then
10:        Precisely compute s
11:       else
12:        s
13:       end if
14:     else
15:       s
16:     end if
17:     for summing terms do
18:       Update t1
19:     end for
20:     Use SIMD vectorization to compute mu[k][j]
21:   end for
22:   if low precision then
23:     Use SIMD vectorization for Kahan summation for c[k]
24:   else
25:     Ordinary summation for c[k]
26:   end if
27: end function

```

The floating point part of the row transformation is optimised to perform multi-step addition, subtraction, and multiplication operations simultaneously. Load the data in the floating point section into the appropriate place in the floating point registers. Use addition, subtraction, and multiplication instructions to reduce the number of instructions and optimise line conversion operations.

3.3. Assembly Optimization

After analyzing the vectorization part of SIMD, it was found that the inner product operation and the floating point operation of row transformation in the calculation of the Gram–Schmidt coefficient have complex instructions and low efficiency. These sections will be optimized for deeper assembly in order to improve performance.

Figure 3 shows the core assembly code for inner product computation, which processes two elements per iteration. This will reduce the number of loops and the overhead of jump instructions. The sequential execution of instructions for inner-product computation is interleaved, and allowing the processor to execute multiple independent instructions at the same time. This will improve instruction-level parallelism. Register reuse will fully utilize the registers and reduce the number of memory access operations. The use of the `faddp` instruction, which directly sum the values of each channel in the pair vector register in line 9, reduces the additional overhead associated with the sum operation.

1	mov	x0,	%[s]	
2	dup	v0.2d,	x0	
3	0:			
4	ld1	v1.2d,	[%[a]],	#16
5	ld1	v2.2d,	[%[b]],	#16
6	fmla	v0.2d,	v1.2d,	v2.2d
7	subs	[%[neonlen]],	[%[neonlen]],	#1
8	bgt	0b		
9	faddp	d1,	v0.2d	
10	fmov	%[s],	d1	

Figure 3. Inner-product calculation assembly optimization.

Figure 4 shows the core part of the floating point operation of the row transformation. The multiplication operation and addition operation are used directly to determine the positive and negative values of the Gram–Schmidt coefficient. A more streamlined **fmla** operation is used to integrate multiplication and addition in line 6, which reduces the use of instructions and speeds up the execution.

1	mov	x0,	%[s]	
2	dup	v0.2d,	x0	
3	0:			
4	ld1	v1.2d,	[%[a]]	
5	ld1	v2.2d,	[%[b]]	
6	fmla	v1.2d,	v0.2d,	v2.2d
7	st1	v1.2d,	[%a]	
8	add	[%a],	[%a],	#16
9	add	[%b],	[%b],	#16
10	subs	[%[neonlen]],	[%[neonlen]],	#1
11	bgt	0b		

Figure 4. Rowtransform assembly optimization.

3.4. Data Prefetch and Loop Unrolling

Before data prefetch and loop unrolling, the specific parameters are determined according to the hardware characteristics of the Phytium FT-2000+ processor, the performance requirements of the algorithm, and the actual performance test results. As shown in Table 1, three combinations are proposed to determine the optimal parameters for data prefetch and loop unrolling based on the cache sizes of the Phytium FT-2000+ at all levels. The unit of data prefetch is the number of bytes, and the unit of loop unrolling is the number of iterations. Taking combination 1 as an example, the instruction prefetch is 64 bytes, and the loop unrolling factor is two.

Table 1. Parameter combination.

Combination	Data Prefetch	Loop Unrolling
1	64	2
2	128	4
3	256	8

By comparing the performance of the three combinations, the optimal parameters with combination 1 on the domestic Tianhe supercomputer Phytium FT-2000+ processor are determined.

Compared with Figure 3, Figure 5 adds **prfm** prefetch instructions in lines 4 and 5, which reduce memory access waiting time and improve data loading efficiency. The **ld1** (load) and **fmla** (multiplication and addition) operations are performed in the loop from line 6 to line 11 to increase computational density and reduce the overhead of loop control

instructions. By prefetching data and increasing computational density, more efficient utilization of memory bandwidth improves overall computing efficiency.

1	mov	x0,	%[s]
2	dup	v0.2d,	x0
3	0:		
4	prfm	pldl1keep,	[%a], 64]
5	prfm	pldl1keep,	[%b], 64]
6	ld1	v1.2d,	[%a]], #16
7	ld1	v2.2d,	[%b]], #16
8	fmla	v0.2d,	v1.2d, v2.2d
9	ld1	v1.2d,	[%a]], #16
10	ld1	v2.2d,	[%b]], #16
11	fmla	v0.2d,	v1.2d, v2.2d
12	subs	[%neonlen],	[%neonlen], #1
13	bgt	0b	
14	faddp	d1,	v0.2d
15	fmov	[%s],	d1

Figure 5. Inner-product calculation prefetch+ loop unroll.

Compared with Figure 4, Figure 6 also adds prefetch instructions in lines 4 and 5. At the same time, the data offset and calculations from lines 6 to 17 are unlooped twice. The neonlen is a quarter of the length of the original data.

1	mov	x0,	%[s]
2	dup	v0.2d,	x0
3	0:		
4	prfm	pldl1keep,	[%a], 64]
5	prfm	pldl1keep,	[%b], 64]
6	ld1	v1.2d,	[%a]]
7	ld1	v2.2d,	[%b]]
8	fmla	v1.2d,	v0.2d, v2.2d
9	st1	v1.2d,	[%a]
10	add	[%a],	[%a], #16
11	add	[%b],	[%b], #16
12	ld1	v1.2d,	[%a]]
13	ld1	v2.2d,	[%b]]
14	fmla	v1.2d,	v0.2d, v2.2d
15	st1	v1.2d,	[%a]
16	add	[%a],	[%a], #16
17	add	[%b],	[%b], #16
18	subs	[%neonlen],	[%neonlen], #1
19	bgt	0b	

Figure 6. Rowtransform prefetch+ loop unroll.

4. Experimental Results and Performance Analysis

4.1. Experimental Environment

The LLL_FP algorithm in this paper was run on the Phytium FT-2000+ processor, which is based on the 64-bit version of the ARMv8 architecture (aarch64) of the Tianhe supercomputer from the National Supercomputing Center in Tianjin. The L1 cache was 2 MiB, the L2 cache was 256 MiB, SIMD was 128 bits, and the compiler used was GCC 9.3.0. The NTL library version 11.5.1 was used. The compilation parameters were set to -O3. The FT-2000+ processor was chosen as the research platform due to its widespread use in domestic supercomputing centers, its compatibility with the ARMv8 architecture, and its advanced hardware features, such as large cache size and support for SIMD instructions, which provide an ideal foundation for testing and optimizing lattice reduction algorithms. The following Table 2 describes the specific environment configurations.

Table 2. Environment configuration.

Hardware environment	CPU	Phytium FT-2000+
	Arch	Aarch64
	SIMD	128 bits
	L1 cache	2 MiB
	L2 cache	256 MiB
Software environment	Compiler	GCC9.3.0
	NTL	11.5.1

4.2. Correctness Verification

4.2.1. Check the Lattice Vector Correctness

In the context of lattice-based algorithms, the core of the correctness check is to verify that the basis vectors after the protocol still generate the same lattice space as the original lattice. Formally, we need to prove that the prescribed matrix B' satisfies the following relationship with the original lattice matrix B :

$$B' = BU \quad (8)$$

where U is the transformation matrix.

4.2.2. Residual Check

The residual matrix R is defined as the difference between the result of the transformation of the original lattice matrix B by the integer matrix U and the prescribed basis matrix B' :

$$R = BU - B' \quad (9)$$

We use the Frobenius norm to measure the size of the residual matrix R .

$$\|R\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |r_{ij}|^2} \quad (10)$$

where m is the number of rows in the matrix, n is the number of columns in the matrix, and r_{ij} is the element of row i and column j in the matrix.

4.2.3. Data Validation

Since this paper optimizes the NTL library, it only needs to check whether the matrix B' generated by it is consistent. After validation, the results for the LLL_FP and the optimized LLL_FP algorithms coincided for lattices of order 10, 20, ..., 50.

4.2.4. Hadamard Ratio

The Hadamard Ratio (HR) is an important metric used to measure the quality of lattice transformation in lattice specification algorithms. It is defined as follows:

$$HR = \left(\frac{\det(B)}{\prod_{i=1}^n \|b_i\|} \right)^{1/n} \quad (11)$$

where B is the lattice base matrix, $\det(B)$ is the determinant of the lattice base matrix B , and $\|b_i\|$ is the Euclidean norm of the lattice base vector b_i . After validation, the HR for the LLL_FP and the optimized LLL_FP algorithms coincided for lattices of order 10, 20, ..., 50.

4.3. Performance Analysis

This paper implements the optimized LLL_FP algorithm in the NTL library. This section will compare the performance of different optimization methods, focusing on the LLL_FP algorithm as a whole, as well as its core calculation components: the Gram–Schmidt coefficient and row transformation. For clarity, this paper uses different descriptors to represent various implementation methods.

In the single-core case, the data size interval was selected as [1000, 3000], with a step size of 500. This resulted in five scales—1000, 1500, 2000, 2500, and 3000—which were used for performance testing. The tests were then run 10 times for each scale to obtain their average values. For ease of description, Table 3 of this paper uses different descriptive symbols to indicate different implementations.

Table 3. Method implementation.

Implementation Method	Descriptor
Inner product calculation	InnerProduct
Optimize inner product calculation	Opt InnerProduct
Calculation of Gram–Schmidt coefficient	ComputeGS
Optimize calculation of Gram–Schmidt coefficient	Opt ComputeGS
Row transformation	RowTransform
Optimize row transformation	Opt RowTransform
Floating-point lattice reduction algorithm	LLL_FP
Optimized floating point lattice reduction algorithm	Opt LLL_FP

4.3.1. Different Methods to Optimize the Effect

Figure 7 shows the performance comparison of different optimization methods. All three optimization methods outperform the serial LLL_FP algorithm. Among them, the assembly optimization method achieves the best results, closely followed by SIMD vectorization. The combination of instruction prefetch and loop unrolling marginally outperforms the serial LLL_FP algorithm. The results indicate that SIMD vectorization and assembly optimization effectively improve parallelism, as discussed in the algorithm analysis section. Additionally, techniques such as data prefetch and loop unrolling enhance overall performance by optimizing memory access.

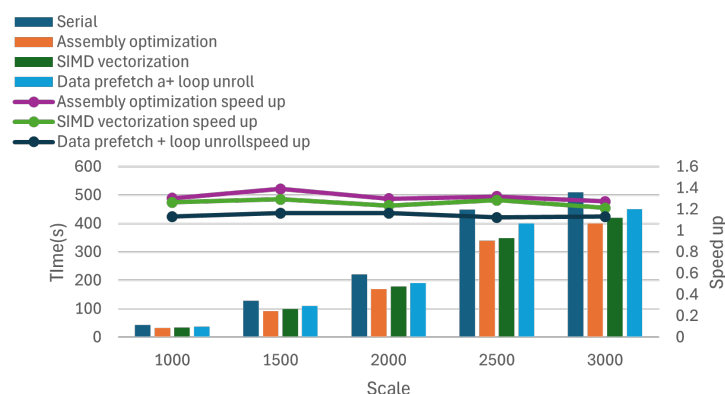


Figure 7. Performance comparison of different methods.

4.3.2. Optimization Effect of Inner Product Calculation

Figure 8 presents the performance comparison of the standard and optimized inner product calculations at different scales. The maximum acceleration ratio is 1.37, the minimum acceleration ratio is 1.23, and the average speedup ratio is 1.33.

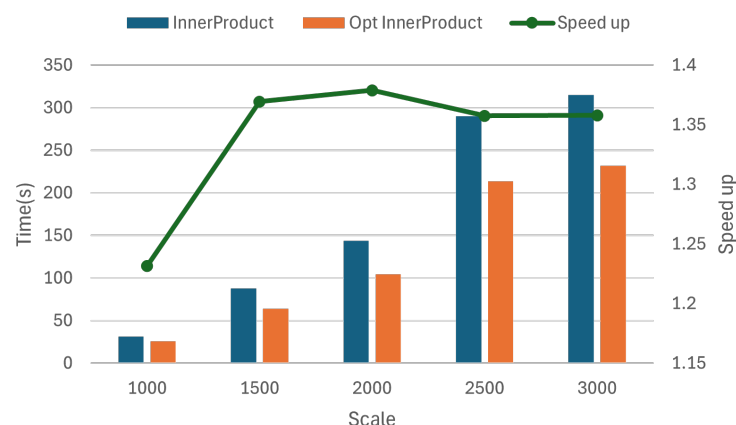


Figure 8. Performance comparison of inner product.

4.3.3. Optimization Effect in Computing Gram–Schmidt Coefficient

Figure 9 presents the performance comparison of the standard and optimized Gram–Schmidt coefficient calculations at different scales. The maximum speedup ratio is 1.50, the minimum speedup ratio is 1.38, and the average speedup ratio is 1.42.

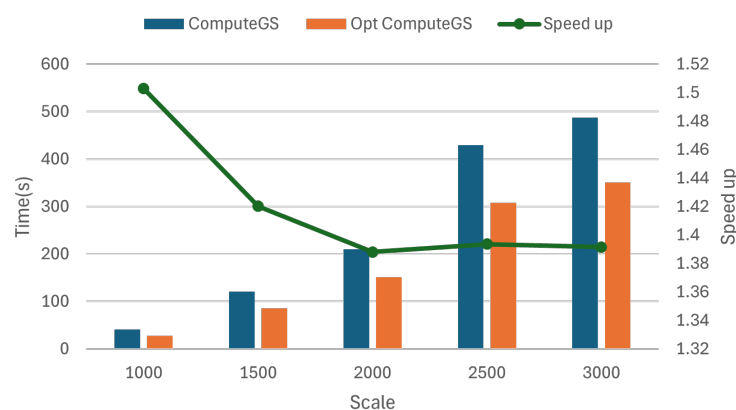


Figure 9. Performance comparison of ComputeGS.

4.3.4. Optimization Effect of Row Transform.

Figure 10 shows the performance comparison between the standard and optimized RowTransform at different scales. Due to their relatively small impact, the maximum speed-up ratio reaches 1.22, the minimum speedup ratio is 1.11, and the average speedup ratio is 1.15.

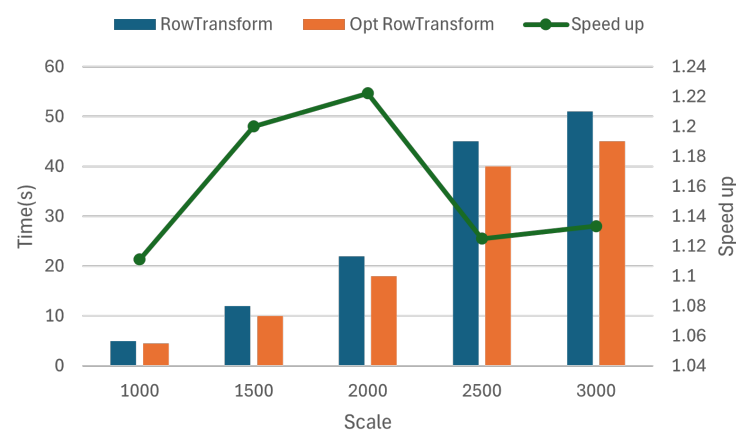


Figure 10. Performance comparison of rowtransform.

4.3.5. Overall Optimization Effect of LLL_FP Algorithm

Figure 11 shows the performance comparison between LLL_FP and the optimized LLL_FP at different scales, where the maximum speedup ratio is 1.42, the minimum speedup ratio is 1.34, and the average speedup ratio is 1.38.

Overall, the optimized LLL_FP algorithm and its sub-methods presented in this paper outperform the LLL_FP algorithm and its sub-methods in the NTL library. The results demonstrate that the optimizations applied to the LLL_FP algorithm in this paper, including SIMD vectorization, assembly optimization, data prefetch, and loop unrolling, are effective.

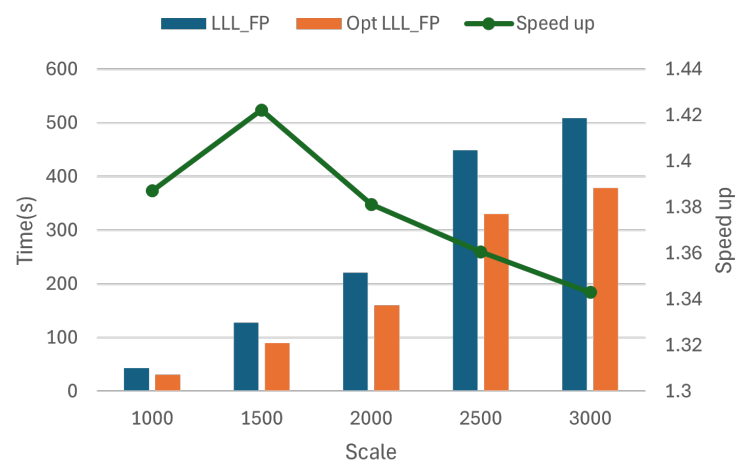


Figure 11. Performance comparison of LLL_FP.

5. Discussion

In the context of the rapid development of current technologies, the method proposed in this paper exhibits strong adaptability to future architectures. With the ongoing development of ARM V9 and RISC-V architectures, which are gradually becoming mainstream choices in high-performance computing, they offer many advanced features. The ARM V9 architecture introduces enhanced security features, expanded SIMD instruction sets, and more efficient multi-core support, all of which offer great potential for algorithm optimization based on this architecture. Meanwhile, the RISC-V architecture, being open-source, provides flexible customization options, allowing optimizations tailored to specific application requirements, opening up more possibilities for lattice reduction algorithm research.

The LLL_FP algorithm proposed in this paper, by optimizing parallel computing, memory access, and SIMD instruction utilization, can fully leverage the advanced features of the ARM V9 and RISC-V architectures. Especially in terms of multi-core and SIMD support, the optimization strategies of this algorithm can be easily ported to these new architectures for enhanced computational performance. Additionally, due to the open-source and customizable nature of ARM V9 and RISC-V, researchers can further optimize algorithms according to specific needs, further improving performance. Therefore, the method proposed in this paper is not only suitable for existing platforms but also will be adaptable to the challenges and opportunities presented by future architectures.

6. Conclusions

In this paper, the LLL_FP algorithm in the NTL library is optimized for the domestic Tianhe supercomputer with an ARMv8 processor. By focusing on the core computational tasks of the algorithm, such as the calculation of the Gram–Schmidt coefficient and row transformation, vectorization and inline assembly are introduced to enhance the performance of the algorithm. Optimization techniques, including data prefetch and loop

unrolling, are applied based on the cache size of the Phytium FT-2000+, and improve the performance of the lattice basis reduction. The optimized LLL_FP algorithm outperforms the original version and provides a new reference for applying lattice reduction algorithms in high-performance computing environments. This study demonstrates the feasibility and effectiveness of optimizing lattice reduction algorithms on domestic supercomputers.

Furthermore, the optimized LLL_FP algorithm has significant practical impacts in domains such as cryptography, wireless communication, and computational number theory, where lattice reduction plays a critical role. The performance improvements enable faster and more efficient computations for cryptographic key attacks, decoding in MIMO systems, and solving integer programming problems. By extending the algorithm to other fields, such as artificial intelligence for feature selection and machine learning model compression, it can potentially open new opportunities for research and application. This study highlights the importance of optimizing algorithms not only for specific hardware platforms but also for addressing broader challenges across multiple disciplines.

Future work will explore cross-platform optimization strategies for other lattice reduction algorithms in the NTL library and investigate artificial intelligence-based optimization technologies [25]. We plan to promote and apply these methods in the National Center for Supercomputing in Tianjin to enhance the practical capabilities of the algorithm and support the needs of various fields.

Author Contributions: Conceptualization, C.G. and J.W.; methodology, C.G. and R.C.; software, C.G.; validation, J.W.; formal analysis, J.W.; data curation, J.W.; writing—original draft preparation, J.W.; writing—review and editing, J.W., J.Z., L.Z., T.X. and H.W.; visualization, R.C.; visualization, C.G.; supervision, C.G.; project administration, C.G. and J.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was financed by the National Natural Science Foundation of Key Program and the National Natural Science Foundation of Youth Program (Nos. 62032023, 61902411, and 42104078).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in the study are included in the article; further inquiries can be directed to the corresponding author.

Acknowledgments: We are very thankful for all the editors and reviewers who helped us improve this manuscript.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Lenstra, A.K.; Lenstra, H.W.; Lovász, L. Factoring polynomials with rational coefficients. *Math. Comput.* **1982**, *39*, 607–620. [\[CrossRef\]](#)
2. Ajtai, M. Generating hard instances of lattice problems. In Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, Philadelphia, PA, USA, 22–24 May 1996; pp. 99–108.
3. 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 *Annual International Conference on the Theory and Applications of Cryptographic Techniques*; Springer International Publishing: Cham, Switzerland, 2019; pp. 717–746.
4. Coppersmith, D. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *J. Cryptol.* **1997**, *10*, 233–260. [\[CrossRef\]](#)
5. Cheng, Y.; Diakonikolas, I.; Ge, R.; Woodruff, D.P. Faster algorithms for high-dimensional robust covariance estimation. In Proceedings of the Conference on Learning Theory, PMLR, Phoenix, AZ, USA, 25–28 June 2019; pp. 727–757.
6. Eisenträger, K.; Hallgren, S.; Kitaev, A.; Song, F. A quantum algorithm for computing the unit group of an arbitrary degree number field. In Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing, New York, NY, USA, 1–3 June 2014; pp. 293–302.

7. Hassibi, B.; Vikalo, H. On the sphere-decoding algorithm I. Expected complexity. *IEEE Trans. Signal Process.* **2005**, *53*, 2806–2818. [\[CrossRef\]](#)
8. Chockalingam, A.; Rajan, B.S. *Large MIMO Systems*; Cambridge University Press: Cambridge, UK, 2014.
9. Lovász, L.; Scarf, H.E. The generalized basis reduction algorithm. *Math. Oper. Res.* **1992**, *17*, 751–764. [\[CrossRef\]](#)
10. Dadush, D.; Végh, L.A.; Zambelli, G. Geometric rescaling algorithms for submodular function minimization. *Math. Oper. Res.* **2021**, *46*, 1081–1108. [\[CrossRef\]](#)
11. Schnorr, C.P.; Euchner, M. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.* **1994**, *66*, 181–199. [\[CrossRef\]](#)
12. Lyu, S.; Ling, C. Boosted KZ and LLL algorithms. *IEEE Trans. Signal Process.* **2017**, *65*, 4784–4796. [\[CrossRef\]](#)
13. Aono, Y.; Wang, Y.; Hayashi, T.; Takagi, T. Improved progressive BKZ algorithms and their precise cost estimation by sharp simulator. In Proceedings of the Advances in Cryptology-EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, 8–12 May 2016; Proceedings, Part I 35; Springer: Berlin/Heidelberg, Germany, 2016; pp. 789–819.
14. Neumaier, A.; Stehlé, D. Faster LLL-type reduction of lattice bases. In Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation, Waterloo, ON, Canada, 19–22 July 2016; pp. 373–380.
15. Luo, Y.; Qiao, S. A parallel LLL algorithm. In Proceedings of the Fourth International C* Conference on Computer Science and Software Engineering, Montreal, QC, Canada, 16–18 May 2011; pp. 93–101.
16. Jeremic, F.; Qiao, S. A Parallel Jacobi-Type Lattice Basis Reduction Algorithm. *Int. J. Numer. Anal. Model. Ser. B* **2014**, *5*, 1–12.
17. Tian, Z.; Qiao, S. An enhanced Jacobi method for lattice-reduction-aided MIMO detection. In Proceedings of the 2013 IEEE China Summit and International Conference on Signal and Information Processing, Beijing, China, 6–10 July 2013; IEEE: Piscataway, NJ, USA, 2013; pp. 39–43.
18. Wen, Q.; Ma, X. Efficient greedy LLL algorithms for lattice decoding. *IEEE Trans. Wirel. Commun.* **2016**, *15*, 3560–3572. [\[CrossRef\]](#)
19. Gu, B.; Qiu, J.; Chi, X. Performance evaluation analysis and empirical research of parallel computing software for heterogeneous systems. *Front. Data Comput.* **2024**, *6*, 116–126.
20. Park, T.; Seo, H.; Kim, J.; Park, H.; Kim, H. Efficient Parallel Implementation of Matrix Multiplication for Lattice-Based Cryptography on Modern ARM Processor. *Secur. Commun. Netw.* **2018**, *2018*, 7012056. [\[CrossRef\]](#)
21. Hassan, S.A.; Mahmoud, M.M.; Hemeida, A.M.; Saber, M.A. Effective implementation of matrix-vector multiplication on Intel's AVX multicore processor. *Computer Languages. Syst. Struct.* **2018**, *51*, 158–175.
22. Lee, W.K.; Seo, H.; Zhang, Z.; Hwang, S.O. TensorCrypto: High throughput acceleration of lattice-based cryptography using tensor core on GPU. *IEEE Access* **2022**, *10*, 20616–20632. [\[CrossRef\]](#)
23. Gao, W.; Xu, J.; Sun, H.; Li, M. Research on cycle optimization technology for SIMD vectorization. *J. Inf. Eng. Univ.* **2016**, *17*, 496–503.
24. Higham, N.J. The accuracy of floating point summation. *SIAM J. Sci. Comput.* **1993**, *14*, 783–799. [\[CrossRef\]](#)
25. Gong, C.; Chen, X.; Lv, S.; Liu, J.; Yang, B.; Wang, Q.; Bao, W.; Pang, Y.; Sun, Y. An efficient image to column algorithm for convolutional neural networks. In Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN), Shenzhen, China, 18–22 July 2021; pp. 1–8. [\[CrossRef\]](#)

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.