Check for updates

ACM DIGITAL LIBRARY | Association for Computing Machinery | acm open

Latest updates: https://dl.acm.org/doi/10.1145/3732365.3732420

RESEARCH-ARTICLE

# Loop Vectorization Optimization Technique for System Calls

LILI LIU

JINYANG YAO

WENBO LIU

CHAOWEI ZHAO

YINGYING LI

JINLONG XU

View all

PDF Download
3732365.3732420.pdf
24 January 2026
Total Citations: 0
Total Downloads: 325

**Published:** 21 February 2025

Citation in BibTeX format

CNSSE 2025: 2025 5th International Conference on Computer Network Security and Software Engineering
*February 21 - 23, 2025*
*Qingdao, China*

CNSSE '25: Proceedings of the 2025 5th International Conference on Computer Network Security and Software Engineering (February 2025)
https://doi.org/10.1145/3732365.3732420
ISBN: 9798400713613

# Loop Vectorization Optimization Technique for System Calls

Lili Liu
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan, China
liull_lili@163.com

Jinyang Yao
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan, China
yaojy1024@126.com

Wenbo Liu
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan, China
liuwenbo060412@126.com

Chaowei Zhao
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan, China
542819222@qq.com

Yingying Li
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan, China
ieulyy@163.com

Jinlong Xu
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan, China
longkaizh@126.com

Ping Zhang
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan, China
13674910726@139.com

Bo Zhao
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan, China
zhaob07@tsinghua.org.cn

## Abstract

With the continuous advancement of modern processor architectures, SIMD (Single Instruction Multiple Data) vectorization has become one of the key techniques for improving program performance. Loop vectorization, as a core technique of SIMD optimization, can significantly enhance the execution efficiency of loop code with data parallelism. However, in practical applications, many loops contain system calls (such as printf), which often introduce side effects or uncertainties in control flow. This leads traditional compilers to adopt a conservative strategy during loop vectorization, avoiding vectorizing the entire loop. While this strategy helps prevent potential execution errors, it also leaves many loops with vectorization potential under-optimized. To address this issue, this paper proposes a loop vectorization optimization technique for loops containing system calls. By introducing directive-based guidance and appropriate handling mechanisms, the proposed method allows the compiler to identify and effectively process loops with system calls, enabling successful vectorization. Specifically, programmers can guide the compiler to selectively vectorize these loops as needed, maximizing performance improvements while ensuring program correctness. By precisely controlling which parts of the code can be vectorized and which must preserve their original execution order, this method effectively solves the problem of non-vectorizable loops with system calls, significantly improving program execution efficiency. To evaluate the effectiveness of the proposed method, we designed 24 test cases containing system calls. Experimental results show that, the highest optimization level (O3) of the compiler, the proposed method achieved an average speedup of 1.4677×. Further analysis indicates that as the ratio of computational instructions to system call instructions increases, the speedup generally rises, despite some fluctuations.

This work is licensed under a Creative Commons Attribution International 4.0 License.

*CNSSE 2025, Qingdao, China*
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1361-3/2025/02
https://doi.org/10.1145/3732365.3732420

## CCS Concepts

• **Software and its engineering** → Software notations and tools; Compilers.

## Keywords

Loop vectorization, OpenMP, System Calls, SIMD

**ACM Reference Format:**
Lili Liu, Jinyang Yao, Wenbo Liu, Chaowei Zhao, Yingying Li, Jinlong Xu, Ping Zhang, and Bo Zhao. 2025. Loop Vectorization Optimization Technique for System Calls. In *2025 5th International Conference on Computer Network Security and Software Engineering (CNSSE 2025), February 21–23, 2025, Qingdao, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3732365.3732420

## 1 Introduction

With the continuous advancement of computing hardware, especially the widespread use of SIMD technology, modern processors can process multiple data elements in parallel within a single clock cycle, significantly improving the execution efficiency of data-intensive applications [1]. SIMD architecture encapsulates multiple data operations into a single instruction, allowing the same operation to be performed on multiple data items within the same instruction cycle. This provides great potential for enhancing the performance of computation-heavy tasks [2]. Currently, SIMD technology is widely used in fields such as digital signal

processing [3], big data analysis [4], artificial intelligence [5], and high-performance computing [6]. Against this background, loop vectorization technology, as an optimization technique, makes full use of the advantages of SIMD hardware. Its goal is to convert traditional scalar code into vectorized code suitable for parallel execution on SIMD architectures. Through loop vectorization, multiple loop iterations in a program can be processed in parallel, which significantly improves the overall execution efficiency of the program, especially in large-scale data processing application scenarios, showing significant performance advantages. [7].

Although loop vectorization can bring significant performance improvements in many applications, it is challenging to implement, especially in code blocks involving system calls. System calls serve as the interface between the program and the operating system and perform a variety of important functions such as resource management, process control, and I/O operations. Because the results of system calls and their side effects are usually unpredictable, compilers face significant difficulties in data flow analysis and dependency inference. System calls often involve accessing or affecting external resources, such as the state of the file system, network resources, or external devices, and these factors further increase the difficulty of compiler control during loop vectorization. This leads compilers to typically abandon vectorization optimization and adopt a more conservative strategy when dealing with loops containing system calls.

Although this conservative strategy can ensure the correctness of the program, it often leads to the loss of optimization opportunities for those loops that can be safely vectorized. This approach fails to take full advantage of the parallel computing power of modern hardware, especially on hardware platforms that support the SIMD instruction set. [8] [9] [10]. In some computationally intensive applications, this lack of optimized processing can significantly affect the efficiency of program execution, resulting in wasted hardware resources and failure to take full advantage of the processor's parallel computing power. More critically, in many cases, some system calls (e.g., a simple printf) do not cause any side effects, or their side effects have no real impact on loop parallelism and data consistency. For these scenarios, the compiler can determine, based on the programmer's guidance, which system calls are safe, thus allowing these loops to be vectorized for optimization. If a vectorization strategy specific to system call functions can be implemented in such scenarios, the full potential of the hardware can be exploited to improve computational performance without sacrificing program correctness.

To address this problem, this paper proposes a loop vectorization method based on a guidance statement control mechanism, which aims to enhance the compiler's vectorization ability to handle loops containing system calls. By introducing explicit guidance statements in the source code, the programmer can specify which loops can be safely optimized for vectorization, thus enabling the compiler to be more flexible in deciding whether to vectorize or not when system calls are encountered. This approach not only provides higher optimization freedom for the compiler, but also provides more control for the developer, thus enabling efficient parallel computation even in complex application scenarios. We have also designed a special processing mechanism for system call vectorization that maximizes the effect of loop body vectorization without sacrificing the correctness and safety of system calls. We implemented the method in gcc-12.3.0 and designed 24 test cases containing system calls to evaluate the method. The experimental results show that the method achieves an average speedup ratio of 1.4677 times compared to the highest optimization level (O3) of the compiler. As the ratio of the number of computed instructions to the number of system call instructions increases, the acceleration ratio shows an overall increasing trend, although there are some fluctuations.

## 2 Related Work

In the field of modern high-performance computing, SIMD parallelism optimization technique has become one of the key means to improve the efficiency of program execution. Especially when dealing with intensive computational tasks, how to enable programs to better execute parallel computations by taking full advantage of SIMD architecture has become a key research direction of compiler optimization. Loop vectorization is the focus of compiler optimization, which makes full use of the SIMD instruction set in hardware by converting the execution of an entire loop into multiple data-parallel vector operations.

Allen et al [11] first proposed a loop-based automatic vectorization method that treats the entire array as a vector unit by manipulating the iteration space of inner loops. Through dependency analysis, they can transform multiple statements between different iterations which do not form dependency loops with each other into vectorized form. With the development of loop vectorization techniques, academics began to explore the potential of outer loop vectorization. Bik et al [12] and Hampton et al [13] proposed loop swapping techniques to address the problem of costly or unachievable vectorization for inner loops with dependency loops, reductio ad absurdum operations, or discontinuity of array references and loop indexes, by swapping outer loops to the innermost position to achieve vectorization. To more fully utilize the data parallelism between outer loop iterations or in linear codes, Nuzman [14] proposed a new loop-aware super word-length parallelism (SLP) approach that combines loop vectorization with SLP vectorization to exploit the vectorization opportunities between iterations.

In the study of loop vectorization, some work has been done for the case of function calls, aiming to achieve efficient vectorization while maintaining computational performance. However, despite the progress in the vectorization of regular function calls, the treatment of system call functions is still insufficient. System call functions usually involve resource management at the operating system level, and their function bodies cannot be inlined. This makes them more complex challenges in the vectorization process. While most of the current research focuses on optimizing the vectorization of inline functions, the feasibility and efficiency of vectorization for system call functions still face significant difficulties due to their complex execution flow and external dependencies. Therefore, despite the accumulation of techniques for the vectorization of traditional function calls, how to effectively deal with the vectorization of system-called functions is still a research topic that needs to be solved in depth.

Kandiah et al. [15] proposed an SPMD programming model called Parsimony, aiming to achieve high computational performance by efficiently utilizing the CPU's SIMD/vector units, while maintaining compatibility with standard programming models, languages, and compiler toolchains. This approach effectively vectorizes loops containing function calls by handling them through inlining. However, it cannot vectorize system call functions that cannot be inlined. Moll et al [16] proposed an if-conversion algorithm called "partial linearization" and implemented it on RV (region vectorizer). The algorithm effectively avoids the problem of executing multiple objectives in a SIMD program by performing if-conversion on non-divergent branches, thus improving the utilization of SIMD. Meanwhile, the algorithm can handle discrete control flows such as break and return, provides good support for complex cases such as containing function calls, and has been successfully applied to loop vectorization containing system call functions such as printf. However, in nested loops, this method still has some problems in handling cases containing system call functions, resulting in incorrect output results. Rapaport et al [17] proposed CHOR$\sigma$uS (C Higher-Order Vector Semantics), a lightweight static C extension that allows programmers to represent computations as composable vector operations applied to scalar kernels. Full-function vectorization is achieved by using the map and fold functions to represent vector operations. However, the method cannot handle system call functions without function bodies. Tian et al [18] proposed a technique for compiling C/C++ SIMD extensions on multicore SIMD processors aimed at vectorization of functions and loops. The method introduces a new set of high-level C/C++ vector extensions and extends the Intel C++ product compiler to transform these vector extensions into optimized SIMD instruction sequences for vectorization of functions and loops. Although the method is effective for vectorizing functions and loops, the extension syntax is cumbersome and not programmer-friendly, and it cannot vectorize loops containing system calls. Masten et al [19] proposed a pass called VecCLONE to implement function-level vectorization before the loop vectorization pass. The approach leverages the functionality of current loop vectorizers and paves the way for future enhancement features. It relies on the OpenMP standard [20], which requires developers to add specific directive declarations to function definitions and call locations to achieve optimization, but the method cannot handle system call functions.

## 3 Methodology and Implementation

Modern compilers (e.g., LLVM [21] and GCC [22]) usually do not vectorize when dealing with loops containing system calls to prevent side effects introduced by system calls, such as modifying the global state or changing memory data. While this ensures program correctness, it may also cause some loops that could have been optimized to lose their chance, thus reducing the execution efficiency of the program. In fact, some system calls (such as printf) do not affect loop parallelism in some cases. If the compiler can identify these safe system calls and allow them to be vectorized, it can improve execution efficiency while maintaining program correctness.

The aim of this study is to propose a new optimization strategy whereby by introducing a new OpenMP guidance statement clause (systemcall), the programmer can explicitly inform the compiler

```
#pragma omp simd systemcall
for (int k = 0; k < N; ++k)
{
   // code before …
   c[k] = a[k] + b[k];
   printf("%d, ", c[k]);
   // code after …

}
```

**Figure 1: Inclusion of the "systemcall" clause.**

that certain system calls do not have side-effects, thus enabling the compiler to make better vectorized optimization decisions during optimization. This not only ensures the correctness of the program, but also provides room for performance improvement, especially when a large number of cyclic computations are involved, avoiding unnecessary performance loss. In the intermediate optimization phase of the compiler, we retain the traditional loop vectorization functionality and introduce a vectorization mechanism specifically for system calls on top of it. This enables the compiler to maximize the efficiency of program execution without compromising correctness when dealing with system calls. Particularly in the optimization of system calls in nested loops, this approach solves complex scenarios that cannot be handled efficiently by traditional methods.

### 3.1 System Call Vectorization Workflow

In the process of system call vectorization, the compiler's job is not only to perform regular vectorization of loops, but more importantly, it needs to identify the system calls in them and perform reasonable optimization. Since system calls (e.g., printf) may have different side-effects and output order requirements in different contexts, the compiler must can properly vectorize system calls while ensuring the semantic correctness of the program. To this end, we propose a scheme for implementing loop vectorization containing system calls via guidance statements. The programmer can explicitly inform the compiler through a guidance statement (e.g., #pragma omp simd systemcall) that a loop contains system calls, but that these calls have no side-effects and can thus be optimized for vectorization (as shown in Figure 1). Through this control mechanism, the compiler can accurately understand the programmer's intention and thus avoid making wrong optimization decisions. However, traditional compilers usually do not consider the possibility of vectorization of system call functions (e.g., printf). We enhance the ability to handle the vectorization of system call functions (e.g., printf) based on the existing loop vectorization optimization, and propose two processing mechanisms for the vectorization of single-layer loops and nested loops respectively.

The important innovation of this thesis is to add a specialized vectorization mechanism for system calls (especially those without side effects) while retaining the loop vectorization function of traditional compilers. Through this mechanism, the compiler can maximize program performance without affecting program correctness.

Figure 2 illustrates a specialized vectorization framework for system calls. The implementation of this framework is done in the

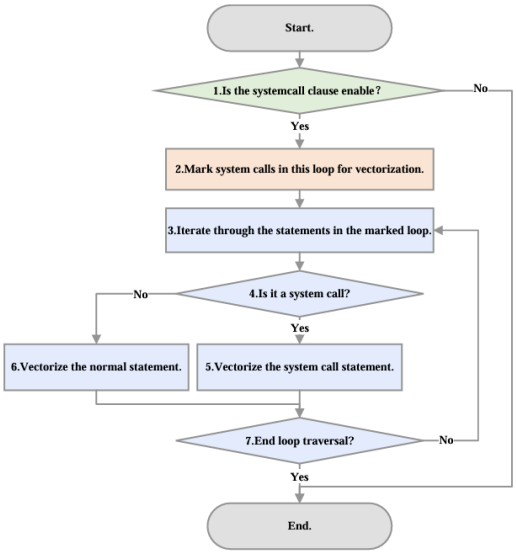


**Figure 2: Vectorization framework for loops containing system calls.**

intermediate representation phase at compile time. For system call vectorization, the programmer needs to add the guidance statement to the loop and add the systemcall clause. The compiler recognizes the systemcall clause and marks the system calls in this loop as requiring vectorization. The compiler then iterates through the statements in the loop body and performs a specialized vectorization mechanism on the system call. Through this mechanism, the compiler can maximize program performance without affecting program correctness.

*3.1.1 Stage 1: Adding the systemcall Clause to Source Code.* The clause systemcall in stage 1 is added in the source code to explicitly inform the compiler, through guidance statements (e.g., #pragma omp simd systemcall), that a certain loop, although it contains system calls, has no side-effects from these calls, and thus can be vectorized for optimization. Through this control mechanism, the compiler can accurately understand the programmer's intent and thus avoid making wrong optimization decisions. We modify the compiler front-end to recognize systemcall clauses and add them to the chain of OpenMP guidance statements for subsequent intermediate representation stages to fetch.

*3.1.2 Stage 2: Marking Loops Containing the systemcall Clause.* Stage 2 flags loops that contain the clause systemcall, indicating that the loop contains a system call function and that the programmer has instructed that it has no side effects to vectorize. In the compiler intermediate representation phase for implementation, we add a private variable systemcall to the loop class. The private variable systemcall is assigned by checking whether there is a systemcall clause in the OpenMP directive statement chain generated by the compiler front-end, and setting it to 0 if there is no systemcall clause, or setting it to 1 if there is.

*3.1.3 Stage 3: Specialized Vectorization Mechanism for System Calls.* Phase 3 begins to iterate over the statements of the loop body marked by systemcall clause, performing the normal vectorization procedure for non-system call statements and our specialized vectorization mechanism for systemcall functions. When vectorizing the system call functions of nested loops and single-layer loops, multiple scalar function calls are generated, and the loop structure is modified to adapt to the vectorization operation. The specific vectorization mechanism is presented in detail in Sections 3.2 and 3.3.

## 3.2 Vectorization Mechanism for System Calls in Single-Layer Loops

When the compiler attempts loop vectorization, it considers whether multiple iterations can be executed in parallel. The system call function printf caches the output data in an internal buffer until the buffer is full or a function such as fflush is called, or the program exits before the data is actually written to the console. Although this behavior has no effect on the computational logic of the program, in a parallel environment, multiple threads calling printf at the same time may cause problems with the order of the output. During vectorization, if more than one vectorization thread tries to execute printf at the same time, it may cause the output to be garbled or block the threads waiting for the output to complete. Therefore, the compiler may choose not to vectorize a system function call such as printf if it is in a loop.

Given that system call functions such as printf are not suitable for vectorization, we propose that the compiler performs a short sequential execution while vectorizing it. Specifically, multiple elements of each loop are processed in the same processing cycle, such that multiple scalar printf calls are made sequentially each time the loop is executed in parallel, as shown in Figure 3. The compiler performs the normal vectorization optimization of the computational part and performs a short sequential execution of the system calls such as printf to ensure the correctness of the output. Through this specific processing mechanism, the compiler can optimize the performance of the program to the maximum extent possible while ensuring that system calls do not destabilize the stability and correctness of the parallel computation, and guaranteeing the vectorization of the loop as a whole.

The specific implementation process, is shown in Algorithm 1. The first step is to extract the vectorization factor by obtaining the vectorization factor (VF), which is the number of elements to be processed in each vector operation, through vinfo. For example, if the value of VF is 4, it means that 4 data elements are processed in parallel each time. Next, the vectorized version of each argument is obtained to generate the list of arguments needed for the new function call. For each parameter of the function call, the corresponding vectorized parameter is obtained based on the parameter position. If an argument is an address expression pointing to a string constant, it is used directly as a scalar argument. Otherwise, for each vectorization factor (i.e., each element processed in parallel) the elements at the corresponding positions of the vector elements are extracted and new scalar operations are created for these elements. Then, a new scalar function call is created, and constructed using the processed arguments, and a return value is assigned to the call.

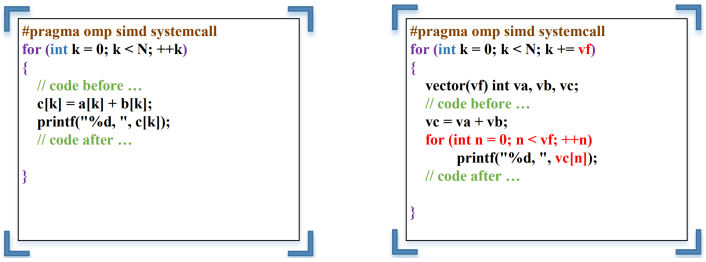


**Figure 3: Vectorization mechanism for system calls in single-layer loop structures.**

**Algorithm 1** vectorize_systemcall (vec_info *vinfo, stmt_vec_info stmt_info, gimple_stmt_iterator *gsi)

```
loop_vinfo     dyn_cast<loop_vec_info>(vinfo)
VF             LOOP_VINFO_VECT_FACTOR (loop_vinfo)
for each j in VF, do
    for each i in num_args, do
        op         get_function_argument(stmt_info, i)
        if is_address_expression(op), do
            scalar_args.append(op)
            continue
        end
        vec_def    get_vector_definition(op, i, stmt_info)
        scalar_arg extract_vector_element(vec_def, j)
        scalar_args.append(scalar_arg)
    end
    call       create_function_call(stmt_info, scalar_args)
    insert_function_call(gsi, call)
end
return true
```

If the function has a return value, the return value is stored in a temporary variable. Finally, the new call statements are inserted, replacing the original function call statement with VF number of scalar calls.

## 3.3 Vectorization Mechanism for System Calls in Nested Loops

The problem of vectorization of system function calls in nested loop architectures is significantly challenging in the field of program optimization. Taking the typical scenario of vectorization factor VF=4 as an example, while the normal output of system functions such as printf can be maintained by four scalar expansions in a single-level loop environment, the direct application of this strategy in nested loop architectures leads to a lack of semantic integrity. This limitation is particularly prominent in the nested loop architecture shown in Figure 4 (a): the outer loop for (int k = 0; k < N; ++k) realizes alternating read and write operations on the array c through two inner loops. The first inner loop for (int n = 0; n < 8; ++n) (hereafter collectively referred to as the n loop) performs the update operation on the first 8 elements of the array (c[n] = k * n), and the second inner loop for (int m = 0; m < N; ++m) (hereafter collectively referred to as the m loop) outputs all the elements of the array via the printf function.

The inability to vectorize this structure arises from its inherent data dependence pattern. Since the value of k in each outer loop iteration modifies the first 8 elements of the array c, and the inner m loop accesses the entire array c during each iteration, this operation introduces inter-iteration data dependence. Specifically, different k values modify the same array elements, causing each outer loop iteration to depend on the computation results of the previous iteration. Vectorization typically requires that data between different iterations be independent, allowing parallel processing of multiple iterations. However, due to inter-iteration data dependence, the compiler cannot parallelize these operations. Additionally, if a strategy similar to single-loop iteration is applied to system function calls, as shown in Figure 4 (b), the program's output order will differ completely from the scalar code's semantics, leading to errors.

To address the technical barriers of nested loop vectorization, this study proposes a solution based on systematic restructuring of the loop structure. Through key techniques such as iteration space reorganization and data access pattern optimization, a feasible vectorization implementation path is constructed while ensuring the correctness of the execution sequence.

*3.3.1 Iteration Space Reorganization.* As shown in Figure 5 (a), the nested loop structure exhibits a typical "partial update - full output" characteristic at the computational pattern level. Specifically, the first inner loop performs a batch update on the first 8 elements of the array through multiplication (c[n] = k * n), followed by the second inner loop that performs a full output operation (printf("%d,", c[m])). This execution sequence creates a strict timing in the original code: after each outer loop iteration updates 8 elements, the full state of the array is immediately output.

Traditional vectorization methods, when dealing with system calls, typically adopt the N×VF execution mode as shown in Figure 5 (b). When the vectorization factor (VF) is 4, this mode processes four outer loop iterations (k=0 to k=3) in parallel, causing the elements of the array c[0] to c[7] to undergo four batch modifications before triggering four system function calls in the second inner loop. This operational paradigm fundamentally conflicts with the original code semantics, where the core issue lies in the fact that the original specification requires the full array output to occur immediately after each outer loop iteration update (as shown in Figure 5 (c)). The vectorization process should parallelize the VF iterations of the inner loop, rather than redundantly applying the VF operations to the system call statements within the inner loop body, as shown in Figure 5 (b).

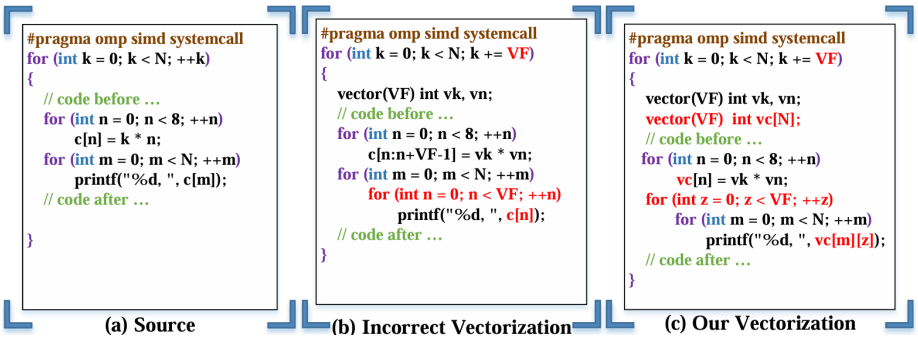


**Figure 4: Vectorization mechanism for system calls in nested loops.**

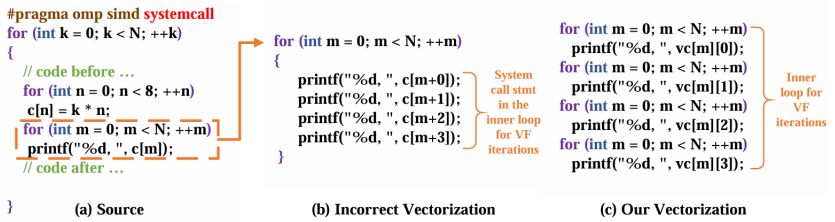


**Figure 5: Example of iteration space reorganization.**

To ensure semantic consistency, this study proposes a new iteration space reorganization method. This method transforms the traditional N×VF execution mode into the VF×N mode, as shown in Figure 5 (c). Specifically, the reorganization method ensures that each vectorization channel outputs the entire array c's elements during execution, thereby effectively maintaining the semantic consistency of the program before and after vectorization. This reorganization not only helps preserve the vectorization characteristics of the outer loop but also ensures that the complete state of the array is correctly output after each outer loop iteration update.

*3.3.2 Optimization of Data Access Patterns.* After completing the iteration space reorganization, the focus shifts to addressing data race issues in a parallel execution environment. When the outer loop is SIMD-vectorized, iterations corresponding to different k-values share the memory space of the array c[0] to c[7], which results in a potential write-overlap risk during the inner loop's write operation c[n] = k * n. Specifically, when iterations k_i and k_j (where $i \neq j$) are concurrently executed in vector channels, writing to the same array element will disrupt the semantic determinism of the program, leading to unpredictable results.

To resolve this issue, we introduce a vectorized array *vc*, which contains *N* vector elements, each of which in turn contains *VF* integer elements, as shown in Figure 4 (c). The introduction of *vc* ensures that each vectorization channel operates independently on the array *c*, thereby avoiding data overwriting (i.e., each vectorization channel has a complete version of array *c*). This guarantees the correctness of the print operations. Specifically, by printing the vectorized array *vc* through a double-loop structure, we ensure that the array *c* for each vectorization channel (with each array containing *N* elements) is correctly output. This method not only ensures the correct print order in the nested loops but also prevents conflicts caused by multiple channels operating on the same array during the vectorization process. Consequently, this approach improves computational performance while preserving the correctness of the program.

## 4 Experiment and Results Analysis

### 4.1 Evaluation Methodology

We implemented the proposed approach in the gcc-12.3.0 compiler and designed some test cases containing system calls on our own. In order to test the performance impact of system call instructions versus non-system call instructions (i.e., computation-related instructions), we constructed the test cases with the ratio of system call instructions to non-system call instructions and arranged them in ascending order of the percentage of system call instructions. The test cases cover a variety of typical arithmetic scenarios, including addition, multiplication, division, floating-point operations, convolution, matrix multiplication, sorting, and math function calls.

The experiments were performed on an Intel(R) Xeon(R) w5-3433 processor running at 2.0 GHz with 256 GB of RAM and a 64-bit Ubuntu 24.04.1 operating system. During the tests, we used the GCC compiler's optimization options "-O3 -fno-tree -vectorize -fopenmp -march=skylake-avx512 -ffast-math". The "-O3" option is used to enable higher-order optimizations, "-fno-tree-vectorize" explicitly disables auto-vectorization, "-fopenmp" is used to enable OpenMP directive statements, "-march=skylake-avx512" specifies optimizations for the Skylake architecture and AVX-512 instruction set, and "-ffast-math" enables fast math optimizations. To validate the effectiveness of our approach, we add the newly proposed "systemcall" clause to our test cases and compare them with the

comparison experiments. In the comparison experiments, auto-vectorization (-ftree-vectorize) was enabled, while other GCC options were kept unchanged, but the newly added "systemcall" clause was not introduced.

## 4.2 Experimental Results Analysis

The experimental results show that for 24 test cases, our proposed method achieves an overall average acceleration ratio of 1.4677 times compared to the comparison experiments, as shown in Figure 6. Based on the experimental data, it can be observed that as the ratio of the number of computation instructions to the number of system call instructions increases, the acceleration ratio shows an overall upward trend although there are some fluctuations. It is worth noting that the introduction of system call functions in program execution may have some impact on vectorization performance. Specifically, processing system calls when vectorizing operations usually requires a temporary conversion to scalar execution mode. During this process, the required data must be extracted from the vector registers, and this extra operation tends to adversely affect the execution efficiency of the program. The smaller the ratio of system call functions, the more significant the overall performance improvement of the program. However, when the ratio reaches 15, a significant decrease in the speedup ratio occurs. After analyzing this phenomenon, we found that this phenomenon mainly stems from the fact that at a ratio of 15, mathematical function calls start to be introduced, resulting in a certain impact on the change of the acceleration ratio.

When we further analyzed the intermediate files, we found that most of the test cases have a vectorization factor of 8, which means that the vectorization operation processes 8 data elements per instruction execution. This vectorization strategy can theoretically significantly improve computational parallelism and reduce execution time. However, vectorization is not without cost. In some cases, vectorized instructions may introduce additional overhead, such as additional instructions that may be required during data loading, storage, and conditional judgment operations. These additional overheads may, under certain conditions, outweigh the parallelism improvement brought by vectorization, which in turn leads to performance degradation. Therefore, the fluctuation of the acceleration ratio of some test cases in the experiment is reasonable.

## 5 Conclusion and Discussion

In this paper, we propose a loop-vectorized optimization technique for containing system calls, which aims to overcome the conservative strategy adopted by traditional compilers when facing system calls by introducing guidance statements and appropriate processing mechanisms, so as to effectively improve the execution efficiency of programs. Experimental results show that the proposed method achieves an average speedup ratio of 1.4677 times compared to the compiler's highest optimization level (O3) in 24 test cases containing system calls. In particular, when the ratio of the number of computed instructions to the number of system call instructions gradually increases, the acceleration ratio shows an overall upward trend although there are some fluctuations.

By directing statements to precisely control which loops can be vectorized and which need to maintain the original execution order, this study not only ensures program correctness, but also maximizes the potential of vectorized optimization. Especially in loops containing fewer system calls, the optimization method shows more significant performance gains. However, the additional overhead introduced by the sequential execution of system calls still restricts the effectiveness of the vectorized optimization to a certain extent, especially in the case of a high percentage of system calls, and its impact on the vectorized performance cannot be ignored. Future research can further explore how to reduce the performance bottleneck induced by system calls, especially in high-performance applications involving massively parallel computing, and the scalability and adaptability of the optimization methods are still directions worth exploring in depth.

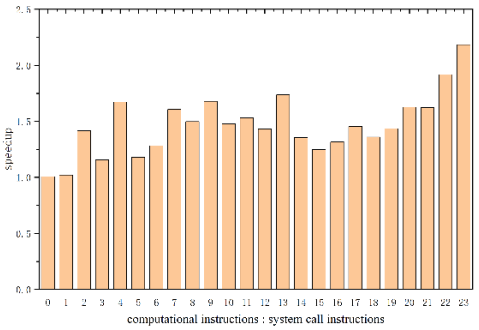


**Figure 6: Speedup of test cases.**

## Acknowledgments

This paper is one of the research results of the 2025 Laboratory for Advanced Computing and Intelligence Engineering (ACIE) project.

## References

[1] Rauber T, Rünger G. Parallel programming: For multicore and cluster systems. [sl]: Springer science & business media, 2013 [J]. Citado na, 2013: 30.
[2] Zhou C, Hassman Z, Xu R, *et al.* SIMD Dataflow Co-optimization for Efficient Neural Networks Inferences on CPUs[J]. arXiv preprint arXiv:2310.00574, 2023.
[3] ZHENG R H, PAI S. Efficient execution of graph algorithms on CPU with SIMD extensions[C]//Proceedings of 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Piscataway: IEEE Press, 2021: 262-276.
[4] BÖHM C, PLANT C. Massively parallel graph drawing and representation learning[C]//Proceedings of 2020 IEEE International Conference on Big Data (Big Data). Piscataway: IEEE Press, 2020: 609-616.
[5] YAMAZAKI S. Future possibilities and effectiveness of JIT from elixir code of image processing and machine learning into native code with SIMD instructions[R]. 2021.
[6] BIAN H D, HUANG J Q, LIU L B, *et al.* ALBUS: a method for efficiently processing SpMV using SIMD and load balancing[J]. Future Generation Computer Systems, 2021, 116: 371-392.
[7] Feng J, He Y, Tao Q. Automatic vectorization, the recent progress and future [J]. Journal of communication, 2022 (003) : 043. DOI: 10.11959 / j.i SSN. 1000-436 - x., 2022051.
[8] Watanabe H, Nakagawa K M. SIMD vectorization for the Lennard-Jones potential with AVX2 and AVX-512 instructions[J]. Computer Physics Communications, 2019, 237: 1-7.
[9] Lee J, Petrogalli F, Hunter G, *et al.* Extending OpenMP SIMD support for target specific code and application to ARM SVE[C]//Scaling OpenMP for Exascale Performance and Portability: 13th International Workshop on OpenMP, IWOMP 2017, Stony Brook, NY, USA, September 20–22, 2017, Proceedings 13. Springer International Publishing, 2017: 62-74.

[10] Tagliavini G, Mach S, Rossi D, *et al.* Design and evaluation of SmallFloat SIMD extensions to the RISC-V ISA[C]//2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2019: 654-657.
[11] Allen R, Kennedy K, Porterfield C, *et al.* Conversion of control dependence to data dependence[C]//Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. New York: ACM Press, 1983: 177-189.
[12] Bik A J C. Applying multimedia extensions for maximum performance[J]. 2004.
[13] Hampton M, Asanovic K. Compiling for vector-thread architectures. In: Proc. of the 6th Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO). 2008.
[14] Nuzman D. loop aware SLP in GCC[C]//GCC Developers Summit. 2007.
[15] Kandiah V, Lustig D, Villa O, *et al.* Parsimony: Enabling SIMD/Vector Programming in Standard Compiler Flows[C]//Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization. 2023: 186-198.
[16] Moll S, Hack S. Partial control-flow linearization[J]. ACM SIGPLAN Notices, 2018, 53(4): 543-556.
[17] Rapaport G, Zaks A, Ben-Asher Y. Streamlining Whole Function Vectorization in C Using Higher Order Vector Semantics[C]//Parallel & Distributed Processing Symposium Workshop. IEEE, 2015. DOI:10.1109/IPDPSW.2015.37.
[18] Tian X, Saito H, Girkar M, *et al.* Compiling C/C++ SIMD extensions for function and loop vectorization on multicore-SIMD processors[C]//2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum. IEEE, 2012: 2349-2358.
[19] Masten M, Tyurin E, Mitropoulou K, *et al.* Function/Kernel Vectorization via Loop Vectorizer[C]//Workshop on the LLVM Compiler Infrastructure in HPC. 2018.
[20] OpenMP Architecture Review Board, OpenMP Application Program Interface Version 5.0 Reference Guide, unpublished, 2018. Accessed 2025-01-14. [Online]. Available: https://www.openmp.org/wpcontent/uploads/OpenMPRef-5.0-0519-web.pdf
[21] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//International symposium on code generation and optimization, 2004. CGO 2004. IEEE, 2004: 75-86.
[22] GCC Community, GCC 12.3.0, unpublished, 2023. Accessed 2025-01-14. [Online]. Available: https://ftp.gnu.org/gnu/gcc/gcc-12.3.0.