Hindawi
Scientific Programming
Volume 2021, Article ID 3264624, 15 pages
https://doi.org/10.1155/2021/3264624

Review Article

Evaluation of Compilers' Capability of Automatic Vectorization Based on Source Code Analysis

Jing Ge Feng,[1,2] Ye Ping He,[1,2] and Qiu Ming Tao[1,2]

[1]*Institute of Software Chinese Academy of Sciences, Beijing 100190, China*
[2]*University of Chinese Academy of Sciences, Beijing 100049, China*

Correspondence should be addressed to Jing Ge Feng; jingge@iscas.ac.cn

Received 31 October 2021; Revised 14 November 2021; Accepted 16 November 2021; Published 30 November 2021

Academic Editor: Fazli Wahid

Automatic vectorization is an important technique for compilers to improve the parallelism of programs. With the widespread usage of SIMD (Single Instruction Multiple Data) extensions in modern processors, automatic vectorization has become a hot topic in the research of compiler techniques. Accurately evaluating the effectiveness of automatic vectorization in typical compilers is quite valuable for compiler optimization and design. This paper evaluates the effectiveness of automatic vectorization, analyzes the limitation of automatic vectorization and the main causes, and improves the automatic vectorization technology. This paper firstly classifies the programs by two main factors: program characteristics and transformation methods. Then, it evaluates the effectiveness of automatic vectorization in three well-known compilers (GCC, LLVM, and ICC, including their multiple versions in recent 5 years) through TSVC (Test Suite for Vectorizing Compilers) benchmark. Furthermore, this paper analyzes the limitation of automatic vectorization based on source code analysis, and introduces the differences between academic research and engineering practice in automatic vectorization and the main causes, Finally, it gives some suggestions as to how to improve automatic vectorization capability.

1. Introduction

Automatic vectorization [1] is a method to realize data-level parallelization by analyzing program characteristics through a compiler and making full use of SIMD extension instructions. Compared with programmers manually writing SIMD vector programs using in-line assembly, intrinsic functions [2], and function libraries, it does not require programmers to deeply understand the functions and features of SIMD extension components, thereby reducing the burden of vector programming. Automatic vectorization has always been a hot research topic in the field of compiler performance optimization. Endless demand for computer performance and the development of SIMD extended instruction set promote the advancement of automatic vectorization technology. Vectorization technology brings new challenges. Firstly, SIMD extended instruction set only has relatively simple vector instructions such as basic operations and continuous memory access. Compilers can only vectorize programs with the same types of operations and regular memory access. Their main applications are limited to the multimedia field. With the rapid development of hardware processor technology, most processors currently support SIMD extension components, such as Intel's SSE/AVX instruction set、NEON [3]/SVE [4] instruction set introduced by ARM, and RISC-V [5] instruction set introduced by the University of California. With the advent of more powerful vector instructions such as discontinuous and reorganization instructions, the compiler can vectorize more complex programs. The application of automatic vectorization technology is extended to complex application scenarios such as scientific computing, signal processing, high-performance computing, and artificial intelligence.

The work of evaluation of automatic vectorization capability is necessary. Firstly, the main goal of automatic vectorization evaluation is to find the limitations of automatic vectorization, which is very beneficial to the further research and development of compilers. Then, in recent years, automatic vectorization has been very active in both academic research and industrial compiler circle. It is necessary to evaluate the ability of automatic vectorization for the latest version of compilers. In recent years, automatic vectorization research has developed rapidly and there have been more groundbreaking research works. For example, in 2017, Gao Wei [6] first put forward the concept of automatic vectorization parallelism. According to the number of parallel programs contained in the program, automatic vectorization parallel method was dynamically selected. This provided a new solution for the research of automatic vectorization. In 2018, Charith Mendis [7] proposed a mathematical representation model of the cost of automatic vectorization describing automatic vectorization as an integer linear programming problem. This provided a new idea for solving the problem of automatic vectorization grouping. In 2019, they abstracted automatic vectorization as a Markov decision process [8] and used the method of imitation learning to solve the group selection problem of automatic vectorization, which effectively improved the performance of automatic vectorization. Automatic vectorization supported by industrial compilers technology is also advanced by leaps and bounds. For example, in 2017, GCC supported an improved method of loop distribution combined with automatic vectorization. In the same year, LLVM developers proposed a Vectorization Plan (Vplan), which used an evaluation cost model to guide automatic vectorization and loop-related optimizations. Finally, there were certain differences in the application fields of different compilers. The automatic vectorization evaluation of specific compilers has practical application. For example, GCC is a standard compiler for Unix-like operating systems. LLVM is in academic research and there are many applications in domain and program-related analysis. LLVM has a wide range of applications in both academia and engineering. In academia, a lot of research work is based on the LLVM compiler source code to achieve more functions, such as PSLP, LSLP, SN-SLP, and other optimization methods. In the engineering world, many processor manufacturers are based on the LLVM compiler for secondary development, such as AMD and Huawei.

The researchers evaluated and analyzed the impact of different compilers, different SIMD extension instruction sets, and different data types on automatic vectorization by manually modifying the test program comparing the performance of automatic vectorization to infer the limitations of automatic vectorization. In 2012, S. Malek et al. [9] used TSVC, PACT, and Media Bench 2 to compare ICC (version 12.0), XLC (version 11.01), and GCC (version 4.7.0) automatic vectorization capabilities that have been evaluated. Test the speedup ratio of automatic vectorization to program performance, manually rewrite the test program, and compare it with automatic vectorization to optimize performance. Inferring the limitations of automatic vectorization, it is found that there is a big gap between automatic vectorization and manual vectorization in improving program performance. In 2015, Zhao Bo et al. [10] used NPB and SPEC CPU 2006 test programs to evaluate different SIMD extensions. The impact of instruction set on automatic vectorization. It is found that the longer the parallel length of SIMD extended instruction set, the greater the speedup of automatic vectorization. In 2017, Moldavanova et al. [11] used TSVC to evaluate the impact of different data-type programs on automatic vectorization and found that automatic vectorization has different effects. The performance impact of data type programs was different. The most compilers can vectorize fp type of codes much better than int type of code codes. With the bit width of codes longer, the performance improvement is more significant.

Researchers use different types of programs to evaluate the ability of automatic vectorization. In 2013, Li Chunjiang et al. [12] used SPEC CPU 2006 and SPEC OMPM 2001 combined with the method of manually writing test programs to automatically vector ICC (version 11.1), PGI (version 11.8), and GCC (version 4.6.1). In 2015, Mahesh Rajan et al. [13] used TSVC, LCALS, and SNL SIERRA test suites for CRAY (version 5.2.40), ICC (version 15.0.2), and GCC (version 4.9.2). The automatic vectorization ability of PARSEC was evaluated. The speedup ratio of automatic vectorization to the program performance improvement was tested, and the automatic vectorization ability was evaluated. In 2017, Yazdanpanah et al. [14] used the PARSEC [15] test program to evaluate and analyze automatic vectorization ability and found that ICC (version 16.0) and GCC (version 5.3) did not automatically vectorize most of the hot programs in PARSEC test program. In 2018, Amiri et al. [16] used the test program composed of the core function of multiplication matrix with manual and automatic vectorization capabilities, and they are evaluated. They found that the performance of using internal functions in different compilers to achieve vectorization is the same, while the performance of automatic vectorization using different compilers is quite different. The abovementioned automatic vectorization evaluation work is not combined with the compiler source code and in-depth analysis of the limitations of automatic vectorization, for example, lack of analysis of the theory and implementation of the automatic vectorization method. Previous evaluation work is "black box test." These methods are limited by the selected test program, cannot fully accurately evaluate the limitations of automatic vectorization, and cannot determine the essential reasons for limitations.

This paper focuses on automatic vectorization evaluation and researches the following questions: What is the optimization capability of automatic vectorization technology proposed and implemented by academia and industry? What are the limitations? What are the essential reasons for these limitations? The technical implementation of industrial compilers and the difference between academic research and optimization direction of subsequent automatic vectorization. These works contribute to the further development of automatic vectorization research and applications, and lay the foundation for improving the next generation of automatic vectorization compilation technology.

Compared with the existing automatic vectorization evaluation work, our paper is different in the following two aspects:

(1) This paper adopts the automatic vectorization evaluation method based on source code analysis of compilers and proposes a new idea for the establishment of an automatic vectorization standard evaluation system. The source code analysis method based on compiler can relatively comprehensively explore the limitations of automatic vectorization and determine the essential reason for the limitation. Based on this, this paper also analyzes and thinks about the differences in automatic vectorization between industry and academia and the reasons for the differences.

(2) Previous work mainly focused on evaluating the impact of different compilers, different hardware platforms, different SIMD, extended instruction sets, and different data types on automatic vectorization. This paper uses TSVC test suite to evaluate the impact of the same compiler in the past 5 years. Automatic vectorization evaluation and analysis work for each historical version. Unexpectedly for some TSVC programs, the automatic vectorization capability of the latest version of compilers is not as good as the historical version.

Based on the evaluation work of automatic vectorization, this paper found that the main problem of automatic vectorization is that the compiler lacks the ability to acquire and analyze important information related to automatic vectorization, such as dependency information, discontinuous memory access information, and so on. The inspiration for this is that fully acquiring and effectively analyzing information related to automatic vectorization is the key to automatic vectorization.

The first section of this paper introduces the current research status of automatic vectorization. The second section introduces the evaluation and analysis of automatic vectorization capabilities based on source code analysis. Firstly, programs are classified according to the main factors that affect automatic vectorization (program features and transformation methods). Then, they evaluate the automatic vectorization ability of each type of program and use TSVC to evaluate the automatic vectorization ability of multiple different historical versions of GCC, LLVM, and ICC compilers. Finally, they analyze the source code of compilers in depth, clarify the limitations of automatic vectorization, discuss the differences between industrial compiler implementation technology and academic research, and give optimization suggestions for automatic vectorization. Section 3 summarizes the full paper.

2. Academic Research of Automatic Vectorization

Automatic vectorization refers to the process of converting a scalar program into vector program by compiler under the constraints of the original program semantics and the support features of SIMD extension components. Automatic vectorization is mainly affected by the following four factors:

(1) Semantic analysis and transformation

Semantic analysis and transformation refers to how to analyze the code for automatic vectorization to ensure that the conversion does not change the semantics of the original program. Automatic vectorization is related to the compiler's semantic analysis and transformation capabilities. When the compiler is not sure whether the program transformation will change the semantics of the original program, it adopts a conservative strategy not to perform the transformation. The stronger the compiler's semantic analysis and transformation capabilities, the greater the possibility of automatic vectorization.

(2) Group analysis and transformation

Vectorization grouping is the process of converting multiple scalar data into one vector data, which affects whether it can be automatically vectorized and the benefits. Semantic analysis is the foundation and precondition of grouping analysis.

(3) Analysis and transformation of processor-related characteristics

Automatic vectorization is closely related to processor characteristics. In order to ensure the normal operation of program, the compiler cannot generate SIMD extension instructions that the processor does not support. In order to effectively improve performance, the compiler generates efficient vector instructions as much as possible [17].

(4) Evaluation analysis of performance

Automatic vectorization is related to compiler performance evaluation and analysis, and performance evaluation ultimately determines whether the program implements automatic vectorization.

2.1. Semantic Analysis and Transformation. The core of semantic analysis for automatic vectorization is dependency analysis. Operations that include dependencies cannot be stored in the same vector register, as the semantics of the original program will be changed. Dependencies are widespread in programs, when two different the operation accesses the same data. And, one of the operations is writing data; then, these two operations have dependencies. Sometimes, it is difficult to determine whether there is a dependency between statements in a program, such as a program that includes nonlinear array access and pointer indexing. Usually, the compiler uses static dependency analysis method. However, the dependencies of some programs can only be determined at runtime, and the dependencies of these programs cannot be completely determined only through static compilation analysis.

2.2. Group Analysis and Transformation. Grouping analysis and transformation refers to saving multiple scalar operations to the same vector register for parallel execution. Some logically complex programs contain multiple levels of nested loops. The compiler sometimes has a variety of vectorization grouping methods for these programs. The automatic vectorization grouping strategy affects the vectorization and its benefits. Gao Wei et al. [1] from the basic block level, loop level, and function-level granularity considered the grouping problem of programs in automatic vectorization.

Basic block-level automatic vectorization is a grouping method to find the vector parallelism of sentences in basic blocks. In 2000, Larsen [18] proposed SLP (Superword Level Parallelism) basic block-level automatic vectorization method, which was based on continuous memory access operations and combined the definition/use or use/definition chain to find multiple isomorphic sentences that can be executed in parallel. The researchers subsequently improved SLP method based on the expansion of seed [19], the selection of optimization chain [20], or based on global search method to optimize the grouping strategy and improve the performance of automatic vectorization. Researchers also expand the scope of use of SLP based on auxiliary program transformation [21–25], enabling SLP to realize automatic vectors across basic block sentence changes.

Loop-based automatic vectorization grouping is a method to find the vectorization parallel of sentences between loop iterations. Allen R and Kennedy et al. [26] proposed the loop-based automatic vectorization method. This method was based on dependency analysis and converts multiple scalar sentences that do not form a dependency loop between different iterations into vector sentences. Follow-up researchers have extended loop-based automatic vectorization to support outer loop vectorization [27] and multi-level nested loop optimal group selection [28].

Function-level automatic vectorization refers to the method of identifying data-level parallelism in the program from function-level granularity [29]. From the perspective of analysis object, the program structure of a function is sometimes complicated, for example, the function can contain sequential execution statements, branch decision statements and loop statements. This brings uncertainty to compiler program analysis.

Loops contain basic blocks. Basic block-level automatic vectorization methods can solve the automatic vectorization problem of loops. Functions can contain loops and basic blocks. Basic block-level automatic vectorization and loop-level automatic vectorization can be applied to solve the problems of automatic vector of functions.

2.3. Analysis and Transformation of Processor Related Characteristics. The essence of automatic vectorization is to use parallel features of SIMD extension components to improve performance. Automatic vectorization is closely related to the parallel features of SIMD extension components and memory access. Due to the limitations of SIMD extension components, the compiler cannot directly use SIMD extension instructions to achieve automatic vectorization for some programs.

In terms of computing parallel feature support, most processors only support the same type of computing parallel vector instructions. The compiler cannot directly use SIMD extension instructions to automatically vectorize multiple statements with different operation types. However in reality there are a large number of programs with different operation types, such as complex number operations.

In terms of memory access parallel feature support most processors only support aligned/contiguous vector memory access instructions or even support non-aligned/non-contiguous vector memory access instructions. However, the delay of non-aligned/non-contiguous vector memory access instructions is relatively large. Non-aligned memory access is one of the problems encountered in program vectorization process. Due to the large delay of non-aligned vector memory access instructions, if you use non-aligned memory direct vector access instructions, the vectorization of programs that include non-aligned memory accesses, will reduce the benefits of automatic vectorization [30] At present the problem of non-aligned automatic vectorization is more fully studied. In terms of analyzing alignment information, the compiler mainly uses static analysis methods. However, sometimes it is not possible to determine whether the program is aligned or not through static analysis [31]alone. Researchers used multiple versions [32] to solve this problem. The compiler is not sure whether it is aligned access or has been determined to be non-aligned when aligned access, data arrangement methods such as shifting and reorganization can be used to reduce non-aligned memory access [33–35]. Non-contiguous memory access is another problem encountered in program vectorization. Due to non-contiguous vector memory access instructions have a large delay.

2.4. $X \in R^{I_{l_1}^{author} \times I_{l_2}^{paper} \times I_{l_3}^{venue} \times I_{l_4}^{term}}$ Evaluation Analysis of Performance. The compiler uses the performance cost model to determine whether to vectorize. The performance cost model needs to be as accurate as possible to ensure that the entire program gains benefits from vector execution. It also needs to be relatively simple to reduce the compiler optimization time. Especially with the rapid growth of amount of program code, fast performance evaluation models are becoming more and more important. The accuracy of performance cost model and the speed of algorithm are often contradictory. The more accurate the performance cost model, the multiple factors of the processor need to be considered. This leads to an increase in the algorithm complexity of performance evaluation model. Most automatic vectorization studies use methods based on instruction delay statistics. There are also researchers who comprehensively consider memory access alignment, memory access continuity, and other factors to improve the performance evaluation cost model [36].

In addition, there are many researches on the use of SIMD to accelerate dynamic binary translation; for example, Yu Ping applies SLP to dynamic binary translation to improve performance [23].

3. Automatic Vectorization Evaluation Based on Source Code Analysis

This paper uses TSVC test suite to evaluate the ability of automatic vectorization, and combines source code analysis to explore the limitations of automatic vectorization and the essential reasons for the limitations.

3.1. Program Classification for Automatic Vectorization Capability Evaluation. Compilers have different automatic vectorization processing capabilities for different types of programs. The more complex the dependencies of program, the more difficult it is for compilers to automatically vectorize. Dependencies are the key factor affecting automatic vectorization. Programs with simple dependencies are easier to achieve automatic vectorization. In order to specifically evaluate the compiler's automatic vectorization processing capabilities for different types of programs, this paper classifies TSVC programs based on dependencies and refers to the classification information of the original TSVC program. Table 1 is the classification of automatic vectorization programs. The table includes typical feature programs (Feature) and program transformation methods (Method). Feature is classified according to program dependency characteristics and there are programs that hinder automatic vectorization of dependencies. We found that these partial programs can change their dependencies through program transformation and then can realize automatic vectorization. In Method this paper focuses on these programs and categorizes them according to transformation method that can change the dependency.

In Feature, this paper categorizes programs according to typical characteristics that depend on and influence its analysis factors. Among them it is easy to realize automatic vectorization for programs that do not contain inter-iteration dependencies (No dependency). Programs containing branch decision statements (Control flow) and programs containing unconditional jump statements (Goto) are easy to introduce control dependencies to the program. Dependency analysis brings uncertainty. Programs that include cross-function calls (Function) and programs that include aliases (Alias) involve inter-procedural and alias analysis and transformation methods, which increase the difficulty of determining dependencies. Programs that include reduction operations (Reduction) and programs that contain induction variables (Induction) often introduce dependencies that hinder automatic vectorization. For simulating operations, programs that include indirect array access (Indirect addressing) bring uncertainty to the dependency analysis of fetched data. Including reverse order Array access programs (Loop reversal) need to perform dependency analysis based on the arrangement of reverse order accesses. Programs that contain continuous memory access within the same loop (Rerolling) involve automatic vectorization parallel and related dependency analysis within and between loop iterations. This paper separately evaluates other programs that contain dependencies but do not contain dependent loop statements (Regular Dependency).

In Method this paper classifies programs that contain dependencies that hinder automatic vectorization according to the program transformation method. Node splitting changes the dependencies of the program by splitting the variables in statement. Loop distribution based on the statements in the loop body, splitting a loop into multiple loops is helpful to realize automatic vectorization of loops that do not contain dependency loops. Loop interchanging can change the dependence of inner and outer loops. Loop peeling splits a loop into multiple loops based on different iteration orders, which helps to realize automatic vectorization of loops that do not contain dependent loop relationships. Scalar expansion is by expanding the scalar. It is an array representation that changes the dependencies contained in programs. Statement reordering changes the dependency between statements by changing the execution order of different statements. Different types can record the same program containing multiple types of features at the same time.

3.2. Automatic Vectorization Evaluation. This paper uses TSVC to evaluate the ability of automatic vectorization. TSVC was developed by Callahan, Dongarra and Levine, and was later extended by Maleki et al. [9] to evaluate the test suite of automatic vectorization capabilities. This paper uses TSVC version evaluation extended by Maleki et al. automatic vectorization capability. TSVC's program is simple and easy to analyze including 151 test programs. It contains nested loops in the form of different typical operations and memory access, such as branch determination, unconditional jump, indirect array access, its core data type is single-precision floating point. TSVC is often used in academia to evaluate the ability of automatic vectorization. In recent years, the open source community of industrial compilers has also paid special attention to it. TSVC has become the benchmark test set of LLVM. At GCC Cauldron Summit in 2015, Sebastian Pop [37] proposed to evaluate the automatic vectorization capability of GCC based on TSVC and improve it.

This paper chooses GCC, LLVM, and ICC compilers for evaluation, mainly for the following 3 reasons:

(1) GCC, LLVM, and ICC have all supported the automatic vectorization function [38–41], which supports the automatic vectorization of programs including reduction statements, branch decision statements, induction variables and non-contiguous memory access programs. GCC, LLVM and ICC is in academic research and there are many applications in domain and program-related analysis, such as performance and security domain.

(2) In recent years, the automatic vectorization capabilities of GCC, LLVM, and ICC have developed rapidly. In terms of GCC's support for automatic vectorization, in 2015 GCC supported automatic vectorization of inductive variable assignment procedures in conditional branches [42]. In 2016 GCC supported automatic vectorization of loop epilogues

Table 1: Automatic vectorization classification description.

Group	Type	Description
Feature	Control flow	Contain if statements
	Function	Contain function statements
	Goto	Contain goto statements
	Indirect addressing	Contain indirect addressing statements
	Induction	Contain induction statements
	No dependency	There is no dependency which prevent automatic vectorization
	Loop reversal	Contain reversal memory access in a loop
	Reduction	Contain reduction statements
	Regular dependency	Contain regular dependency in a loop
	Rerolling	Rerolling
	Alias	Contain aliasing statements
	Symbolic resolution	Contain variables which prevent automatic vectorization
Method	Node splitting	Node splitting
	Loop distribution	Loop distribution
	Loop interchanging	Loop interchanging
	Loop peeling	Loop peeling
	Scalar expansion	Scalar expansion
	Statement reordering	Statement reordering

[43]. In 2017, GCC supported an improved method of automatic vectorization based on alias information to break the loop of program dependence [44]. In 2018, GCC supported a vectorization factor selection method based on an estimated cost model [45]. In 2019, GCC supported SLP method based on mask loading and memory access [46]. LLVM's automatic vectorization technology is also developing rapidly. In 2015 LLVM supported the automatic vectorization method based on loop distribution [47]. LLVM developed in 2017 the author proposed the Vplan, which used the cost model to comprehensively guide cycle-related transformations and automatic vectorization [48]. ICC has mainly improved the support for AVX512 instruction set and OpenMP [49] in recent years.

(3) GCC, LLVM and ICC are widely used in reality. GCC and LLVM are open source compilers, and their corresponding open source communities are the most active compiler communities. ICC is a commercial compiler developed by Intel, which is recognized as automatic compiler with strong vectorization ability. We searched international paper databases such as ACM, IEEE, Elsevier and Springer, and domestic databases such as Wanfang and HowNet, finally obtained more than 100 papers directly related to automatic vectorization. Statistics on the use of specific compiler development or experimental comparison in these papers are shown in Table 2. The first line indicates the compiler, the second line indicates the creator of corresponding compiler, and the third line indicates the number of papers that use corresponding compiler. Among them, ICC, GCC, and LLVM are used in the field of automatic vectorization research.

This paper evaluates and tests the historical versions of GCC, LLVM, and ICC in the past 5 years. The version and compilation option information of specific compiler is shown in Table 3. This paper compares the performance of test program under the conditions of turning on and turning off automatic vectorization optimization option. The compilers all use the default O3 optimization option c99 compilation standard, and use Sse4.2 instruction set. Baseline options are the benchmark compilation optimization options after the compiler turns off automatic vectorization. The vector optimization compilation options (Vectorization options) enable the automatic option of the vectorization function. Since GCC's automatic vectorization is closely related to the optimization of induction variables and array element conversion. GCC's compilation options add induction variable optimization (Fivopts) and vector array conversion optimization (Flax-vector-conversions). The core of TSVC main data type of program is single-precision floating-point. GCC is conservative in the optimization of floating-point operations and needs to turn on the unsafe-math-optimizations compilation option.

Table 2: Application of compilers in Automatic vectorization.

Compiler	ICC	GCC	LLVM	OPEN64	SW-VEC	XL	SUIF
Creator	Intel	GNU	Illinois	SGI	SW	IBM	Stanford
Numbers	33	27	20	16	13	6	6

The processor model of the computer used in the experiment is Intel i7-4790, its main frequency is 3.2 GHz, it supports AVX2, AVX1 and SSE vector instruction set, the vector register of AVX2 is 256 bits long and can handle 4 doubles or 8 floats at the same time. AVX1 and SSE have a 128 bit vector register that can handle two doubles or four floats simultaneously. L1D is 32 KB (8way, 64 B/line), L2 is 256 KB (8 way, 64 B/line), L3 is 8 MB (shared memory), memory is 20 GB, and the operating system is Ubuntu16.04, glibc2.8. The specific test process is as follows: Based on the compilation and optimization options of baseline options

Table 3: Compiler's versions and flags used in the evaluation.

Compiler	GCC	LLVM	ICC
Version	9.2.0,8.3.0,7.4.0,6.5.0,5.5.0	9.0.0,8.0.0,7.0.1,6.0.1,5.0.1	19.0.1,18.0.5,17.0.4,16.0.4,15.0.6
Baseline options	-std = c99 -O3 -fivopts -flax-vector-conversions -funsafe-math-optimizations -msse4.2 -fno-tree-vectorize	-std = c99 -O3 -msse4.2 -fno-vectorize	-std = c99 -O3 -msse4.2 -no-vec
Vectorization options	-std = c99 -O3 -fivopts -flax-vector-conversions -funsafe-math-optimizations -msse4.2	-std = c99 -O3 -msse4.2	-std = c99 -O3 -msse4.2

and Vectorization options respectively, TSVC program is compiled using GCC, LLVM, and ICC. Test the compilation and execution time respectively, run 10 times, and take the arithmetic average. Then use GNU Objdump tool to print the program and compile the optimized assembler. By analyzing the assembly program, it is determined whether the program realizes automatic vectorization.

3.2.1. Compile Time Evaluation. This paper evaluates the time taken by compilers to compile TSVC. Figure 1 is a statistical graph of the time for GCC, LLVM, and ICC to compile TSVC with the automatic vectorization option turned on and off. It reflects the compilation and optimization speed of TSVC programs for a specific version of compilers, where the horizontal axis represents compilers, and the vertical axis represents the compilation time in seconds. In general, automatic vectorization increases the compilation time. As shown in Figure 1(a), in 5 versions of GCC as GCC version is updated, the compilation and optimization time is getting longer and longer, and the impact of automatic vectorization on the overall compilation time of GCC is gradually reduced. Especially in the latest version 9, the compilation time introduced by automatic vectorization only accounts for 4.5% of total compilation time. Among the 5 versions of LLVM, as shown in Figure 1(b), with the update of LLVM version, the compilation of LLVM time has increased significantly. Especially, the latest version 9 of LLVM, its compilation optimization time is 1.45 times that of LLVM5 version. Among the 5 versions of ICC, as shown in Figure 1(c), with the update of ICC version, the compilation time of ICC fluctuates. Among them, the compilation time of ICC16 is the longest, and the compilation time of ICC17 is the shortest. As shown in Figure 1(d), we compare the latest versions of GCC, LLVM, and ICC. The total compilation time of LLVM and the compilation time introduced by automatic vectorization are the longest. The total compilation time of GCC and the compilation time introduced by automatic vectorization are the shortest.

3.2.2. Running Time Evaluation. This paper evaluates the running time of TSVC. Figure 2 is a statistical chart of the program running time, which reflects the impact of automatic vectorization on overall performance of the TSVC program. The horizontal axis in Figure 2 represents the compiler version, and the vertical axis represents TSVC after automatic vectorization. The total running time is in seconds. Figure 2(a) shows the automatic vectorization optimization performance of five historical versions of GCC. After GCC turns on automatic vectorization, it can significantly improve the performance of TSVC. With version update, the running time of TSVC shows a slight decrease. Figure 2(b) shows the automatic vectorization performance of 5 historical versions of LLVM. After LLVM turns on the automatic vectorization, the performance of TSVC can be effectively improved as the version is updated. The running time of TSVC program fluctuates, and its latest version 9 is not the best for TSVC optimization ability of its historical version. Figure 2(c) shows the automatic vectorization performance of 5 historical versions of ICC. ICC can significantly improve the performance of TSVC with the update of version after turning on automatic vectorization. The running time of TSVC fluctuates, and its latest version 19's optimization ability for TSVC is not the best in historical version. Figure 2(d) is the performance comparison chart of the latest version of GCC, LLVM, and ICC. Automatic vectorization is turned on and off under the condition of compilation options, the performance of the program after ICC compilation and optimization is the best, and the performance after LLVM compilation and optimization is the worst. Automatic vectorization makes the performance of GCC and ICC compilers significantly improved and makes LLVM compilation. The performance of the program is only slightly improved. It can be seen that the automatic vectorization ability of GCC and ICC is relatively strong, while the automatic vectorization ability of LLVM is relatively weak.

We are based on the same type of compiler (GCC, LLVM,or ICC) using different historical versions to compile TSVC with the automatic vectorization option turned on and off.Figure 3is a statistical diagram of the optimal performance of TSVC programs. The horizontal axis represents different versions of compilers with the automatic vectorization option turned on and off, and the vertical axis represents the number of programs that compile TSVC with the corresponding version to obtain the optimal performance. If the optimal performance of the program exists in multiple versions at the same time, then record the program in these versions at the same time. We found that:Usethe latest version of GCC, LLVM,and ICC to compile TSVC with automatic vectorization turned on,andthe performance of most programs in TSVC is not the best in historical versions. At present, the latest GCC9, LLVM9,and ICC19 in the historical version correspondsto the same type of compiler,andthe number of compiled TSVC optimal performance programs accounted for 33.8%, 13.2%,and 16.6% respectively. GCC9 compiled TSVC S115, S277, and S315 compared with the performance of the best compiler of its

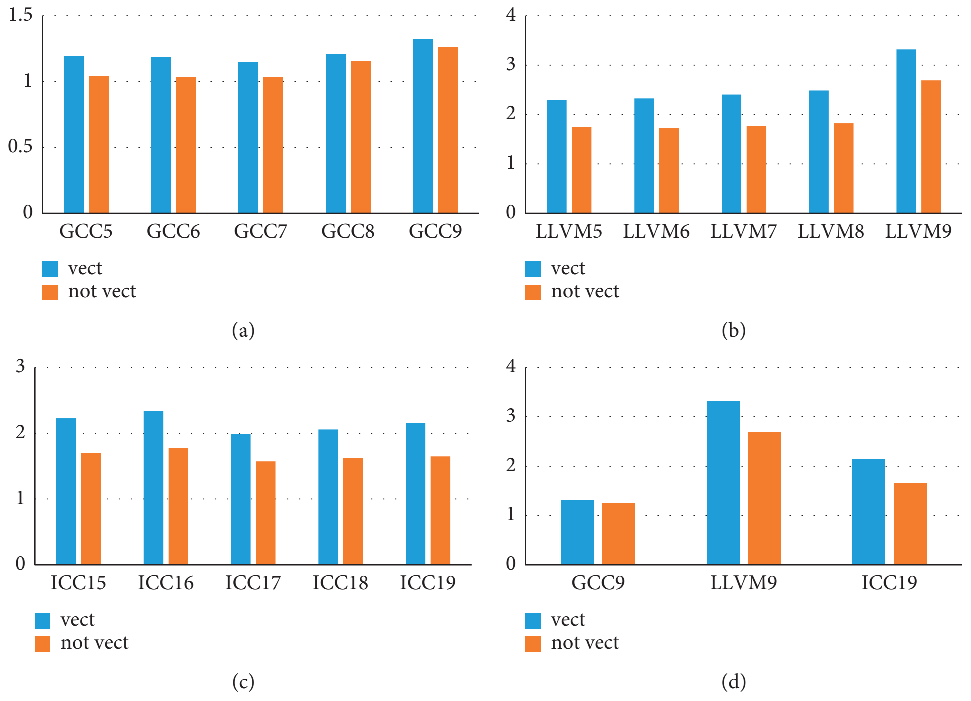


Figure 1: Compare the compilation time in TSVC. (a) Compilation time in TSVC by using GCC. (b) Compilation time in TSVC by using LLVM. (c) Compilation time in TSVC by using ICC. (d) Compilation time in TSVC by using new compilers.

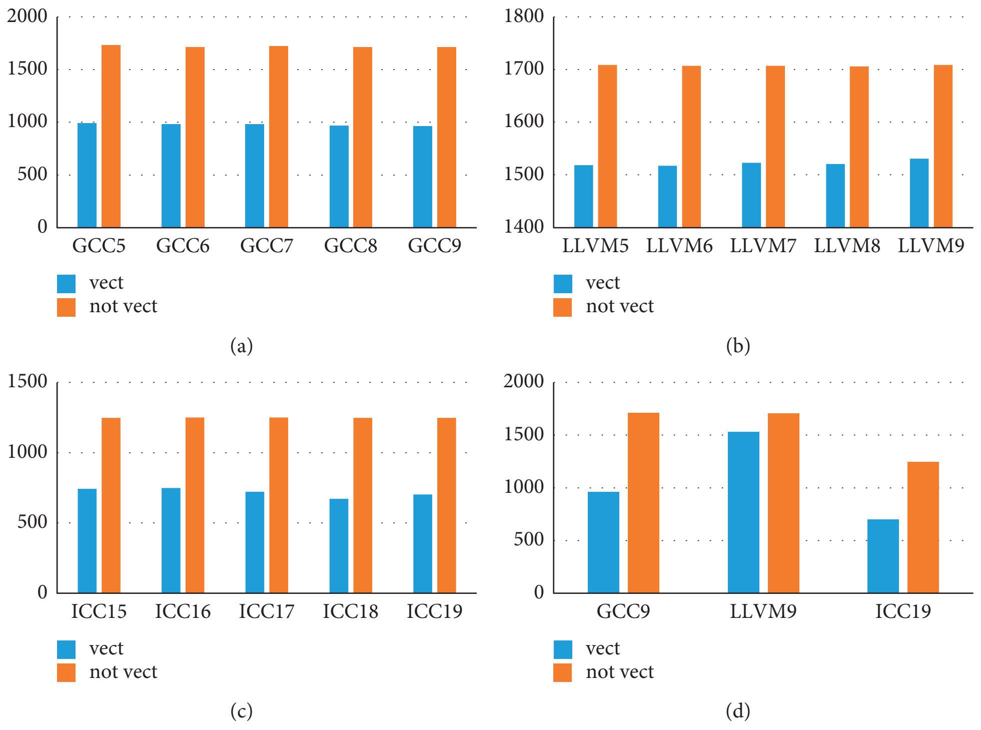
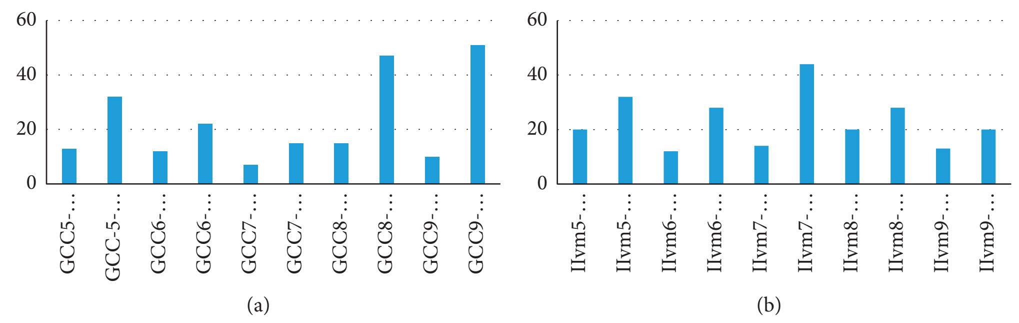


Figure 2: : Execution time of TSVC. (a) Execution time of TSVC. (b) Execution time of TSVC. (c) Execution time of TSVC. (d) Execution time of TSVC.

historical version, and the performance of program has decreased from 37.3% to 72.2%. LLVM9 compiles the S1111 and S442 programs of TSVC, and the performance of the best compiler of LLVM is reduced by 40.5% and 60.7%, respectively. ICC19 compiles the S252 and S4114 programs of TSVC, and the performance of the optimal compiler in



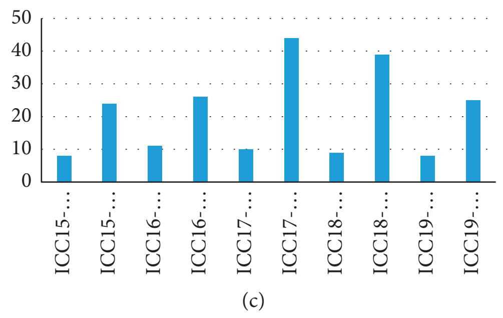


Figure 3: Number of best performance programs in TSVC, which is compiled by compilers in multiple versions. (a) Number of best performance programs in TSVC. (b) Number of best performance programs in TSVC, which is compiled by GCC, which is compiled by LLVM. (c) Number of best performance programs in TSVC, which is compiled by ICC.

historical version of ICC drops by 60.4% and 31.5%,respectively.

3.2.3. Statistical Evaluation for the Number of Automatic Vectorization Programs. This paper counts the number of programs inwhichdifferent compilers implement automatic vectorization of TSVC test suites.Figure 4is a statistical diagram for the number of TSVC programs that can be automatically vectorized by compilers.The horizontal axis represents the version of compiler and the vertical axis represents the number of programs that the corresponding compiler can vectorize TSVC successfully. ICC can vectorize the most programs in TSVC successfully, and it is much better than GCC and LLVM. This paper counts the number of automatic vectorization programs of the same type of compiler and finds that in different versions of GCC as the version is updated, GCC can vectorize more and more TSVCs. LLVM and ICC are in their respective versions,andthe number of vectorized programs fluctuates. The number of vectorized TSVC programs of the latest version of LLVM and ICC is not the most in its historical version. We found historical compilation by analyzing compiler source code and viewing the log information of compiling TSVC. The main reasons why the latest version of compilers can be vectorized are: (1) The improvement of the automatic vectorization evaluation cost model;the automatic vectorization evaluation model of the previous historical version of compilers is notveryaccurate. Sometimes,although it can be vectorized for a specific program, its performance drops instead. For example, such as S352 program, ICC15 and ICC16 can vectorize S352 successfully, but performance of

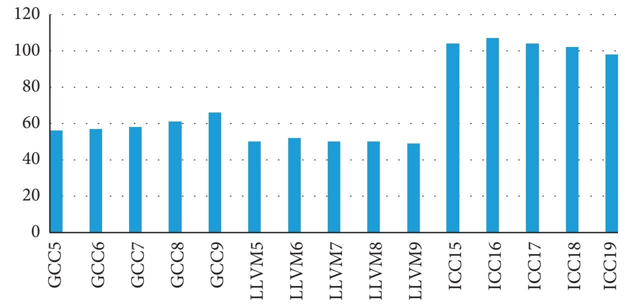


Figure 4: Number of automatic vectorization programs in TSVC.

S352 degrades. The automatic vectorization evaluation model of ICC17 is further improved, so that the compiler does not automatically vectorize S352. (2) The update of compilers optimization method interferes with the processing capability of automatic vectorization for specific programs, such as the S252 program. ICC17 can realize automatic vectorization of the program and can significantly improve the performance of programs. However, ICC18 and ICC19 do not implement automatic vectorization for this program. We use the compilation option Qopt-report to view the compilation log information and find that ICC19 believes that S252 program has a dependency that prevents automatic vectorization, and thus does not implement automatic vectorization.

This paper conducts automatic vectorization evaluation for different types of programs in TSVC. Table 4 is a statistical table of the automatic vectorization speedup ratio of TSVC compiled with GCC9, LLVM9, and ICC19. It reflects the ability of automatic vectorization to process specific types of programs. The numbers column in the table indicates the number of specific

Table 4: Automatic vectorization speedup ratio.

Group	Type	Numbers	GCC	LLVM	ICC
Feature	Control flow	16	2|1.36	1|0.90	13|1.94
	Function	6	4|1.93	5|1.82	6|2.24
	Goto	12	1|1.99	0	7|1.76
	Indirect addressing	10	0	1|0.99	0
	Induction	10	8|2.00	4|1.5	7|1.79
	No dependency	16	12|2.15	9|2.12	11|2.96
	Loop reversal	2	2|1.89	1|1.91	1|2.03
	Reduction	16	9|8.81	0	13|3.66
	Regular dependency	9	3|1.5	2|1.91	2|1.41
	Rerolling	3	1|2.33	3|1.00	2|2.68
	Alias	5	5|2.29	5|2.44	4|1.25
	Symbolic resolution	10	10|2.45	9|2.43	7|2.29
Method	Node splitting	6	1|1.01	2|1.93	6|2.28
	Loop distribution	3	1|4.02	0	3|1.51
	Loop interchanging	6	3|2.20	1|1.07	3|1.81
	Loop peeling	4	0	0	2|2.51
	Scalar expansion	14	6|1.82	6|1.82	8|1.81
	Statement reordering	3	0	0	3|1.77

types of programs in TSVC. The data on the left of the vertical bararethe number of automatic vectorization programs implemented by compilers, and the data on the right of the vertical bar is the average speedup ratio of automatic vectorization. In general, once the programs in TSVC realize automatic vectorization, the performance improvement is significant. ICC has the strongest automatic vectorization ability, and the number of test programs that can be vectorized accounts for the entire TSVC 64.9% of the program, the next is GCC. GCC can vectorize 43.7% of the test programs in TSVC. The automatic vectorization ability of LLVM isrelativelyweak under this test condition and can only vectorize 32.4% of the test programs in TSVC. The compiler has a strong ability to automatically vectorize programs that contain simple dependencies. However, the automatic vectorization processing capability ofrelatively-complex programs including conditional/unconditional branch jump statements and indirect array access is insufficient.

3.3. Automatic Vectorization Analysis. The previous section introduced the experimental results of using TSVC to evaluate automatic vectorization. This section focuses on programs with poor automatic vectorization performance, combined with the compiler source code to analyze the limitations of automatic vectorization and the essential reasons for limitations. According to the program classification method introduced in Section 2.1, they are introduced from two aspects: typical programs and program transformations.

3.3.1. Typical Program. Since the compiler has a serious lack of automatic vectorization capabilities for branch decision statements and indirect array access programs, this paper focuses on the analysis of these two types of programs.

(1) Automatic vectorization of branch decision sentences: Branch decision statements often appear in programs. TSVC has 16 test programs that contain branch decision statements. GCC9 and LLVM9 can only vectorize 2 and 1 test programs of this type of program,respectively, and ICC19 can vectorize 13 of them. As shown inFigure 5, taking the S271 program of TSVC as an example, it shows that the compiler is for this type of program. The processing capability of program automatic vectorization. ICC19 can realize the automatic vectorization of S271, but GCC9 and LLVM9 do not realize automatic vectorization. Through the source code analysis of GCC and LLVM, it is found that GCC and LLVM are based on control dependence to data dependence (If-conversion) method, and realizes the automatic vectorization of branch decision program. For S271 program, GCC and LLVM did not implement automatic vectorization because they did not perform If-conversion.

The automatic vectorization method based on If-conversion was proposed by Allen R et al. to convert control dependence into data dependence [50], and then performed automatic vectorization. Figure 6 is an example diagram of automatic vectorization based on If-conversion; on the left it is the original program, and the right side shows that the compiler uses If-conversion and uses a select instruction to automatically vectorize the statement containing branch decision.

There are three problems with automatic vectorization method based on If-conversion: The first problem is that If-conversion is performed on all programs without the guidance of the cost model, and more redundant select instructions are generated. The second problem is that the vector parallelism contained in the control flow unit is not considered, and it is easy to produce more redundant branch-processing-related instructions. The third problem is that the automatic vectorization method based on If-conversion is restricted by If-conversion. Once the program cannot perform If-conversion, the program cannot be automatically vectorized.

Researchers of automatic vectorization improved automatic vectorization method based on If-conversion. Aiming at the first problem based on If-conversion automatic vectorization method in 2007,Shinet al. [51]used predicates to determine whether a specific statement of the program will be executed, bypassing some statements that are not actually executed. This method reduced the number of redundant instructions generated. In 2018,Simon Moll et al.[52]adopted a partial branch linearization method based on data flow and control flow analysis to reduce the number of unnecessary conditional branch conversions for If-based.In 2017, Gao Wei et al. [53]vectoried the codes within the basic block specified by branch statement and proposed a direct control flow vectorization method based on data reuse between basic blocks.Vectorization contained loops of more isomorphic sentences and the cost model was used to guide the generation of vectorized instructions. This method did not depend on If-conversion and is suitable for conditional branch corresponding basic blocks containing more vectorized parallel programs. For the third problem of If-conversion, we did not find a corresponding solution. A solution to this type of problem is introduced in Section 5.

```
float a[1024],b[1024],c[1024];/*Global
varibal*/
for (int i= 0; i< LEN; i++)
{
    if (b[i] > (float)0.)
        a[i] += b[i] * c[i];
}
```

(a)

```
mulss 0x80b8c0 (%rax),%xmm0
addss 0x82acc0 (%rax),%xmm0
movss %xmm0,0x82acc0 (%rax)
add $0x4,%rax
cmp $0x1f400,%rax
```

(b)

```
cmpltps %xmm1,%xmm2
cmpltps %xmm4,%xmm5
mulps %xmm0,%xmm1
mulps %xmm3,%xmm4
andps %xmm1,%xmm2
andps %xmm4,%xmm5
addps 0x853400 (,%rdx,4),%xmm2
addps 0x853410 (,%rdx,4),%xmm5
movups %xmm2,0x853400 (,%rdx,4)
movups %xmm5,0x853410 (,%rdx,4)
```

(c)

Figure 5: TSVC S271.

```
for (i=0;i<256;i++)              for (i=0;i<256;i+=4)
{                                {
    if (m[i]!=4)                     v4= (4,4,4,4);
        n[i]=m[i];      ⇨            vp=m[i:i+3]!=v4;
}                                    n[i:i+3]=select (n[i:i+3], m[i:i+3],vp);
                                 }
```

Figure 6: An example of automatic vectorization method based on If-conversion.

(2) Automatic vectorization of indirect array access: Indirect array[54]is a programwhereinthe index of the exponent group is obtained indirectly through array access valuesor array access operations. For example,Figure 7is the core program of TSVC S4112, where a[], b[] are single-precision floating-point-type arrays. ip is a 32 bit integer pointer, *s* is a single-precision floating-point variable, b[ip [i]] is an indirect array access, and access to the content of array*b* requiresidxretrieval. For some programs that include array access[55], the compiler does not vectorize without knowing whether it accesses continuous, aligned,or dependent information[56].

TSVC has 10 indirect array-access-type programs. ICC19 and GCC9 do not implement automatic vectorization of this type of program. LLVM9 only vectorizes one of them. This paper found through source code analysis that GCC uses similar pattern matching methods to handle indirect array access types. The program does not support the mode of indirect array access programs in TSVC. LLVM supports automatic vectorization of indirect array access programs, but the efficiency of generating vector instructions is low and its cost model determines that automatic vectorization has no benefit. It is worth mentioning that ICCsupports vectorization of indirect array access types. For example, ICC17 can automatically vectorize 6 programs in TSVC,whichcan significantly improve the performance oftheprogram. We use the compilation optionQopt-report to view the compilation loginformation of ICC19 and find that ICC19 determines that there is a block that prevents automatic vectorization. There is no benefit to the dependency or evaluation cost and the automatic vectorization of indirect array access program in TSVC is not implemented.

The academic community mainly uses special hardware vector instructions or local memory access reorganization methods to solve the automatic vectorization problem including indirect array access procedures. For example, in 2018,Jianget al.[57]based on the equivalent transformation of the associative law using AVX512 conflict detection and horizontal reduction instructions can effectively handle the vectorization of indirect array access programs based on commutative reduction types. In the same year,Yao Jinyang et al. [58]assigned indirect arrays to temporary arrays and loaded the data in the temporary arrays to SIMD vector register to achieve vectorization. If you need to store it in an indirect array after the operation is over, most of the indirect array access programs of TSVC can be processed by the above method. However, the above method has not been applied to industrial compilers. The automatic vectorization technology of industrial compilers for indirect array access programs needs to be further improved by learning from the methods of academia.

3.3.2. Program Transformation Related to Automatic Vectorization. According to the evaluation work of automatic vectorization, this paper finds that automatic vectorization is closely related to multiple program transformation methods. Although some programs have dependencies that hinder automatic vectorization, the dependencies can be changed through program transformation to achieve automatic vectorization. For TSVC and automatic vectorization-related transformation programs (Method class in Table 1), ICC's automatic vectorization processing capability is better than GCC and LLVM. GCC and LLVM cannot realize automatic vectorization for most of these programs. Through the source code analysis of GCC and LLVM, it is found that the automatic vectorization of GCC and LLVM is not perfect in combination with other related program transformation methods. Because there is no necessarypre-transformation of program, the dependency that hinders automatic vectorization in the program has not been changed, and the program has not been vectorized. The following takes TSVC S221as an example to illustrate the limitations of compilers'automatic vectorization of methodprograms. Asshown inFigure 8, there is a true dependencebetweendifferent iterations of the program,-whichhinders automatic vectorization. Ifthecompiler transformsthe program into two loops, loop 1 can be vectorized and loop 2 cannot be vectorized. The vectorization of loop 1 by compilers improves the performance and ICC

```
for (int i= 0; i< LEN; i++)
{
    a[i] += b[ip[i]] * s;
}
```

(a)

```
……
movslq 0x0 (%rbp,%rdx,8),%rcx
movss 0x6f87c0 (,%rcx,4),%xmm0
mulss %xmm2,%xmm0
addss 0x853400 (,%rdx,8),%xmm0
movss %xmm0,0x853400 (,%rdx,8)
movslq 0x4 (%rbp,%rdx,8),%rsi
movss 0x6f87c0 (,%rsi,4),%xmm1
mulss %xmm2,%xmm1
addss 0x853404 (,%rdx,8),%xmm1
……
```

(b)

Figure 7: TSVC S4112.

handles it like this. However, GCC and LLVM did not implement automatic vectorization of TSVC S221 programs.

There have been related studies on the combination of automatic vectorization and other program transformations in the academic field. In 2013, Suo Weiyi [59] obtained definition/use and use/definition relations and reuse between different basic blocks based on control flow and data flow analysis, using the information of loop distribution to increase the number of parallel programs of the program and improve the performance of automatic vectorization. In the same year, Kong *M* et al. [60] based on the polyhedron model, combined vectorization, parallelization, and locality factors for program transformation. This method selected the relatively optimal vectorization scheme according to the program characteristics, which effectively reduced the generation of memory access and rearrangement instructions. In 2017, Zhao Jie et al. [61] used dependency analysis to guide the transformation, and divided the dependencies according to the program features. For true dependence, anti-dependence, and output dependence, techniques such as program rearrangement and node splitting were used to change the dependence between sentences. This method can realize the vectorization of programs containing anti-dependency statements. The above method can solve the automatic vectorization problem of most method programs in TSVC. The fusion technology of automatic vectorization and related transformations supported by industrial compilers needs to be further improved.

This paper sorts out the analysis and transformation methods related to automatic vectorization. Table 5 is a statistical table of program analysis and transformation related to automatic vectorization. Program transformation affects automatic vectorization's preservation judgment, operation arrangement, memory access arrangement, and the number of parallel statements.

Preservation analysis and transformation directly affect automatic vectorization and its benefits. In terms of correlation analysis and transformation, dependency analysis, alias analysis, and inter-procedure analysis are all related to automatic vectorization. Dependency analysis and alias analysis affect the compiler's judgment of dependency between statements. Inter-procedural analysis affects the compiler's cross-function judgment of dependencies between call statements. Some program transformations affect the dependencies between statements, which in turn affect automatic vectorization, such as loop distribution, If-conversion, and scalar expansion. Loop distribution can divide the statements in a loop into vectorization part and non-vectorizable parts. After loop distribution, the automatic vectorization of vectorizable parts in statement in the loop is realized. If-conversion converts control dependence into data dependence and increases the possibility of compilers to auto-vectorize programs that contain control dependence. Scalar expansion and array expansion are based on, at the expense of memory overhead, reducing dependencies within loops increasing the possibility of automatic vectorization. Similarly, scalar renaming and array renaming can reduce anti-dependence. Loop interchange changes the execution order of statements in the loop, thereby changing the statements in the loop instruction scheduling optimization and reducing anti-dependence by changing the execution order of instructions, thereby increasing the possibility of automatic vectorization. Software pipeline can change the execution order of statements and affect the compiler's judgment on dependency of statements.

Since most processors only support the parallel operation of the same operation type, the operation arrangement affects automatic vectorization. Intensity reduction and instruction fusion affect the operation arrangement of automatic vectorization-related sentences, and the intensity reduction transformation replaces operations with higher instruction delays. For operations with relatively low latency, in traditional compilers, intensity reduction optimization only considers the benefits of its own scalar statements, not the benefits of automatic vectorization. Intensity reduction optimization can increase the benefits of scalar operations, but sometimes it will destroy the arrangement of automatic vectorization. As shown in Figure 9, the compiler can automatically convert the last line of multiplication operations into left shift operations, which makes the types of operations of these statements different and hinders automatic vectorization. Similarly, instruction fusion optimization merges multiple instructions into one, which affects the layout of the calculation data of automatic vectorization.

The memory access arrangement, such as memory access alignment or continuous nature, affects automatic vectorization and its benefits. Loop tiling [61] and loop interchange affect the memory access arrangement of automatic

```
for (int i= 1; i< LEN; i++)
{
    a[i] += c[i] * d[i];
    b[i] = b[i-1] + a[i] + d[i];
}
```

```
for (int i= 1; i< LEN; i++)
{
    a[i] += c[i] * d[i];
}
for (int i= 1; i< LEN; i++)
{
    b[i] = b[i-1] + a[i] + d[i];
}
```

(a)

```
……
movss 0x6ebac0 (,%rcx,4),%xmm0
mulss %xmm2,%xmm0
addss 0x846700 (,%rdx,8),%xmm0
movss %xmm0,0x846700 (,%rdx,8)
movslq 0x4 (%rbp,%rdx,8),%rsi
……
```

(b)

```
……
mulps %xmm2,%xmm3
movups %xmm1,0x40 (%rsp,%r12,4)
addps 0x853400 (,%r9,4),%xmm3
movups %xmm3,0x853400 (,%r9,4)
……
```

(c)

Figure 8: An example of loop distribution.

Table 5: Analysis and transform methods that are related to Automatic vectorization.

Number	Influence factors	Transform methods
1	Semantics	Dependence analysis, alias analysis, interprocedural analysis Loop distribution, if-conversion, scalar expand, scalar rename Array rename, loop interchange, instruction schedule, software pipeline
2	Opcode	Strength reduction optimization, instruction fusion
3	Memory	Loop tiling, loop interchange
4	Number of parallel programs	Loop unrolling, loop fusion, redundancy elimination

```
a[0] = b[0] *5;        a[0] = b[0] *5;
a[1] = b[1] *6;        a[1] = b[1] *6;
a[2] = b[2] *7;        a[2] = b[2] *7;
a[3] = b[3] *8;        a[3] = b[3]<< 3;
```

Figure 9: An example of intensity reduction optimization.

vectorization-related sentences, which in turn affects the benefits of automatic vectorization. Loop tiling and loop interchange can change the order of statement execution, rearrange the fetched data, and affect cache performance.

Most processors only support vectorization parallelism with powers of 2 and a limited length. The number of parallelized statements affects vectorization and its benefits. Loop unrolling, loop fusion, and redundant instruction elimination affect the number of statements that affect automatic vectorization parallelism. Loop unrolling expands loop multiple iterations to increase the number of parallel statements inside the loop body. Loop fusion merges multiple loops into the same loop, expanding the number of statements inside the loop body. Redundancy elimination optimization sometimes reduces automatically the number of vectorization parallel statements.

4. Conclusion

Automatic vectorization has always been a hot topic in the field of compiler performance optimization. This paper uses TSVC to evaluate and analyze multiple historical versions of GCC, LLVM, and ICC for automatic vectorization capabilities by using source code analysis methods and combining industrial compiler support technologies. The limitations of automatic vectorization have been clarified and optimization suggestions have been given based on the current situation of academic research. In order to establish a standardized evaluation system, the evaluation method based on source code analysis is conducive to systematically clarifying the limitations of automatic vectorization and the essential reasons for these limitations. This paper is found through evaluation: (1) The limitation of automatic vectorization for industrial compilers is not only in automatic vectorization method itself but also in the analysis and auxiliary transformation methods related to automatic vectorization. (2) At present, the main problem with automatic vectorization is that compilers lack the ability to acquire and analyze important information related to automatic vectorization. (3) The latest versions of GCC, LLVM, and ICC compile and optimize some programs of TSVC, and compare them with the corresponding historical versions, and their performance is not optimal. The main reason is the inaccurate evaluation cost model of compilers and the interference from other compilation optimizations on automatic vectorization.

With the rapid development of software, there are more and more logically complex programs, and the evaluation of automatic vectorization becomes more and more difficult. For those programs with complex logic which cannot be automatically vectorized, it is increasingly difficult for researchers to manually determine whether the program really cannot be vectorized or is caused by insufficient automatic vectorization capabilities. The automation of evaluation is very beneficial for the development of automatic vectorization technology. The academia has carried out research work on loop automatic vectorization evaluation based on dependency analysis. We will start research work on vectorization related evaluation of basic blocks and functions based on coverage statistics in the future.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The author declares that there are no conflicts of interest regarding this study.

References

[1] W. Gao, R. C. Zhao, L. Han, J. M. Peng, and R. Ding, "Research on SIMD automatic vectorization compiling optimization," *Ruan Jian Xue Bao/Journal Of Software*, vol. 26, no. 6, pp. 1265–1284, 2015.

[2] "Streaming Simd Extensions[Eb/Ol]," 2016, https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions.

[3] R. G. Venu, *Neon Technology Introduction, ARM Corporation 4.1*, 2008.

[4] N. Stephens, S. Biles, M. Boettcher et al., "The ARM scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[5] A. Waterman, Y. Lee, D. A. Patterson et al., *The RISC-V instruction set manual*Vol. 1, California University Berkeley Dept Of Electrical Engineering And Computer Sciences, CA, USA, 2014.

[6] W. Gao, L. Han, R. C. Zhao, J. L. Xun, and C. R. Chen, "Vectorization method for loops guided by SIMD parallelism," *Journal of Software*, vol. 27, no. 5, 2016.

[7] C. Mendis and S. Amarasinghe, "goSLP: globally optimized superword level parallelism framework," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, 2018.

[8] C. Mendis, C. Yang, Y. Pu et al., "Compiler automatic vectorization with imitation learning," *Advances in Neural Information Processing Systems*, Article ID 14609, 2019.

[9] S. Maleki, Y. Gao, M. J. Garzaran et al., "An evaluation of vectorizing compilers," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 372–382, TX, USA, October 2011.

[10] B. Zhao, W. Gao, R. Zhao, L. Han, H. Sun, and Y. Li, "Performance evaluation of npb and spec cpu2006 on various simd extensions," in *Proceedings of the International Conference on Big Data Computing and Communications*, August 2015.

[11] O. V. Moldovanova and M. G. Kurnosov, "Automatic vectorization of loops on intel 64 and intel xeon phi: analysis and evaluation," in *Proceedings of the International Conference on Parallel Computing Technologies*, pp. 143–150, Springer, Nizhni Novgorod, Russia, September 2017.

[12] C. J. Li, J. J. Huang, Y. Xu, Y. F. Du, and J. Chen, "Evaluation and analysis of effects of automatic vectorization in typical compilers," *Computer Science*, vol. 40, no. 4, pp. 41–46, 2013.

[13] M. Rajan, D. W. Doerfler, M. Tupek et al., "An investigation of compiler vectorization on current and next-generation intel processors using benchmarks and sandia's sierra applications," 2015.

[14] F. Yazdanpanah, "An approach for analyzing auto-vectorization potential of emerging workloads," *Microprocessors and Microsystems*, vol. 49, pp. 139–149, 2017.

[15] X. Zhan, Y. Bao, C. Bienia, and K. Li, "Parsec3.0," *ACM SIGARCH - Computer Architecture News*, vol. 44, no. 5, pp. 1–16, 2017.

[16] H. Amiri, A. Shahbahrami, A. Pohl, and B. Juurlink, "Performance evaluation of implicit and explicit SIMDization," *Microprocessors and Microsystems*, vol. 63, pp. 158–168, 2018.

[17] W. H. Zhang, J. H. Zhu, H. J. Zhang, and B. Y. Zang, "Optimizing SIMD parallelism through bitwidth analysis," *Chinese Journal of Computers*, vol. 32, no. 11, pp. 2168–2177, 2009.

[18] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation - PLDI*, pp. 145–156, Vancouver, Britith Columbia, Canada, June 2000.

[19] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir, "A compiler framework for extracting superword level parallelism," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 347–358, 2012.

[20] J. Huh and J. Tuck, "Improving the effectiveness of searching for isomorphic chains in superword level Parallelism," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 718–729, ACM, MA, USA, October 2017.

[21] V. Porpodas, R. C. O. Rocha, E. Brevnov et al., "Super-node slp: optimized vectorization for code sequences containing operators and their inverse elements," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 206–216, IEEE, Washington, DC, USA, February 2019.

[22] V. Porpodas, R. C. O. Rocha, and L. F. W. V. W.-S. L. P. Góes, "Autovectorization with adaptive vector width," in *Proceedings of the 2018 international conference on parallel architecture and compilation (PACT), Ser. PACT.*, Limassol, Cyprus, November 2018.

[23] Y. P. Liu, D. Y. Hong, J. J. Wu, S. Y. Fu, and W. C. Hsu, "Exploiting SIMD asymmetry in ARM-to-x86 dynamic binary translation," *ACM Transactions on Architecture and Code Optimization*, vol. 16, no. 1, pp. 1–24, 2019.

[24] Y. Chen, C. Mendis, M. Carbin, and A. Saman, "VeGen: a vectorizer generator for SIMD and beyond," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 902–914, USA, April 2021.

[25] A. A. Haj, N. K. Ahmed, T. Willke, Y. S. Shao, A. Krste, and S. Ion, "NeuroVectorizer: end-to-end vectorization with deep reinforcement learning," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pp. 242–255, San Diego CA USA, February 2020.

[26] J. R. Allen, K. Kennedy, C. Porterfield, and W. Joe, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 177–189, ACM, Austin, TX, USA, January 1983.

[27] D. Nuzman and A. Zaks, "Outer-loop vectorization-revisited for short simd architectures," in *Proceedings of the Parallel Architectures and Compilation Techniques (PACT), 2008 International Conference on. IEEE*, pp. 2–11, Toronto, ON, Canada, October 2008.

[28] S. Wei, R.-C. Zhao, and Y. Yao, "Loop-Nest auto-vectorization based on SLP," *Journal of Software*, vol. 23, no. 7, pp. 1717–1728, 2012.

[29] R. E. A. Moreira, C. Collange, and F. M. P. Quintão, "Function call Re-vectorization," *ACM SIGPLAN Notices*, vol. 52, no. 8, pp. 313–326, 2017.

[30] C. L. Yu and Y. W. Wang, "Design and implementation of SIMD unaligned memory access structure," *Computer Engineering*, vol. 42, no. 9, pp. 1–4, 2016.

[31] P. Wu, A. E. Eichenberger, and A. Wang, "Efficient SIMD code generation for runtime alignment and length conversion," in *Proceedings of the International Symposium on Code*

Generation and Optimization, 2005, pp. 153–164, IEEE, San Jose, CA, USA, March 2005.

[32] C. Fan, R. C. Zhao, Z. Shan, and P. Y. Li, "Storage optimization for struct vectorization," *Journal Of Chinese Computer Systems*, vol. 37, no. 9, pp. 1889–1897, 2016.

[33] A. E. Eichenberger, P. Wu, and K. O'brien, "Vectorization for SIMD architectures with alignment constraints," *ACM SIGPLAN Notices*, vol. 39, no. 6, pp. 82–93, 2004.

[34] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "Performance impact of misaligned accesses in simd extensions," in *Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2006)*, pp. 334–342, Netherlands, Veldhoven, November 2006.

[35] J. L. Xu, R. C. Zhao, and X. Y. Xu, "Memory access optimization for vector program of simd form," *Computer Science*, vol. 42, no. 12, pp. 18–22, 2015.

[36] K. Trifunovic, D. Nuzman, A. Cohen et al., "Polyhedral-model guided loop-nest automatic vectorization[C]//Parallel Architectures and Compilation Techniques," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pp. 327–337, Raleigh, NC, USA, September 2009.

[37] G. C. C. Cauldron, "GNU Wiki," 2015, https://GCC.gnu.org/wiki/cauldron.

[38] A. Anderson, A. Malik, and D. Gregg, "Automatic vectorization of interleaved data revisited," *ACM Transactions on Architecture and Code Optimization*, vol. 12, no. 4, 2015.

[39] A. J. C. Bik, M. Girkar, and P. M. Grey, "Automatic intra-register vectorization for the Intel architecture," *International Journal of Parallel Programming*, vol. 30, no. 2, pp. 65–98, 2002.

[40] I. Rosen, D. Nuzman, and A. Zaks, "Loop-aware SLP in GCC," in *Proceedings of the GCC Developers Summit*, pp. 131–141, Ontario, Canada, July 2007.

[41] "Automatic Vectorization in LLVM[EB/OL]," 2019, http://LLVM.org/docs/%20Vectorizers.html.

[42] "Vectorizing conditional expressions [EB/OL]," 2015, https://gcc.gnu.org/ml/gcc-patches/2015-09/msg00690.html.

[43] "Support vectorization of loop epilogues[EB/OL]," 2016, https://gcc.gnu.org/ml/gcc-patches/2016-05/msg01562.html.

[44] "Enable tree loop distribution[EB/OL]," 2017, https://gcc.gnu.org/ml/gcc-patches/2017-06/msg00124.html.

[45] "Consider Multiple Vector Sizes for Vectorization Based on Cost[EB/OL]," 2018, https://gcc.gnu.org/ml/gcc-patches/2018-06/%20msg01397.html.

[46] "Add SLP support for masked loads[EB/O L]," 2019, https://gcc.gnu.org/ml/gcc-patches/2019-01/msg00909.html.

[47] "Loop distribution/Partial vectorization[EB/OL]," 2015, http://lists.llvm.org/pipermail/llvm-dev/2015-January/080431.html.

[48] "Vectorization Plan[EB/OL]," 2017, https://llvm.org/docs/Proposals/VectorizationPlan.html.

[49] S. F. Liu, Y. Q. Zhang, and X. Z. Sun, "An improved guided loop scheduling algorithm for OpenMP," *Journal of Computer Research and Development*, vol. 47, no. 4, pp. 687–694, 2010.

[50] J. E. Smith, G. Faanes, and R. Sugumar, "Vector instruction set support for conditional operations," in *Proceedings of the 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, ACM, BC, Canada, June 2000.

[51] J. Shin, "Introducing control flow into vectorized code," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pp. 280–291, IEEE, Brasov, Romania, September 2007.

[52] S. Moll and S. Hack, "Partial control-flow linearization," in *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*, pp. 543–556, ACM, Philadelphia, PA, USA, June 2018.

[53] W. Gao, Y. Y. Li, H. H. Sun, Y. B. Li, and R. C. Zhao, "Improved simd vectorization method in the presence of control flow[J]," *Journal of Software*, vol. 28, no. 8, pp. 2046–2063, 2017.

[54] H. Sun, R. Zhao, W. Gao, Y. Gong, and L. Gang, "Exploiting Pure Superword Level Parallelism For Array Indirections," in *Proceedings of the 2015 Seventh International Symposium on Parallel Architectures, Algorithms and Programming (PAAP),*, pp. 13–19, Nanjing, China, December 2015.

[55] Q. Wang, L. Han, J. Y. Yao, and H. Liu, "Research on vectorization technology for irregular data access," in *Proceedings of the International Symposium on Parallel Architecture, Algorithm and Programming*, pp. 321–334, Springer, Haikou, China, June 2017.

[56] S. Kim and H. Han, "Efficient SIMD code generation for irregular kernels," *ACM Sigplan Notices*, vol. 47, no. 8, pp. 55–64, 2012.

[57] P. Jiang and G. Agrawal, "Conflict-free vectorization of associative irregular applications with recent SIMD architectural advances," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pp. 175–187, Vienna Austria, February 2018.

[58] J. Y. Yao, R. C. Zhao, Q. Wang, and Y. Y. Li, "Vectorization methods for indirect array index," *Computer Science*, vol. 45, no. 9, pp. 220–223+236, 2018.

[59] W. Y. Suo, R. C. Zhao, Y. Yao, and X. M. Zhang, "SLP optimization algorithm using across basic block transformation and loop distribution," *Computer Science*, vol. 40, no. 10, pp. 24–28, 2013.

[60] M. Kong, R. Veras, K. Stock, P. Sadayappan, F. Franz, and N. P. Louis, "When polyhedral transformations meet SIMD code generation," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 48, no. 6, pp. 127–138, Seattle, Washington, USA, June 2013.

[61] J. Zhao, H. Cui, Y. Zhang, X. Feng, and J. Xue, "Revisiting loop tiling for datacenters: live and let live," in *Proceedings of the 2018 international conference on supercomputing. ACM*, pp. 328–340, Beijing, China, June 2018.