Research Article

An SLP Vectorization Method Based on Equivalent Extended Transformation

Jingge Feng,1,2 Yeping He,1,2 Qiuming Tao,1,2 and Hengtai Ma1,2

1National Engineering Research Center for Fundamental Software, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
2University of Chinese Academy of Sciences, Beijing 100049, China

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

Received 9 January 2022; Revised 1 February 2022; Accepted 12 February 2022; Published 9 March 2022

Academic Editor: Mohammad Farukh Hashmi

Copyright © 2022 Jingge Feng et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

SIMD extensions provide an efficient energy consumption platform to support mobile systems. How to use SIMD instructions to improve program performance is a challenge. SLP (superword level parallelism) is an efficient solution to exploit the parallelism, oriented to SIMD, between statements in the basic blocks, and it has been widely used in almost all the mainstream compilers. SLP relies on finding isomorphic statements to pack together into vectors. However, the capability of autovectorization for nonisomorphic statements is insufficient. In this paper, we introduce SLP-E, a novel autovectorization method that can automatically vectorize the codes which contain nonisomorphic statements, translate the nonisomorphic statements into the isomorphic statements by equivalent extended transformation of expressions, and vectorize the isomorphic statements. SLP-E improves the application scope and benefits of SLP. We implement the SLP-E in LLVM and compare it with prior approaches. A set of applications that benefit from autovectorization are taken from the SPEC CPU 2017 benchmark to compare our approach and prior techniques. Experimental results show that SLP-E achieves more than 43.9% speedup, on average, over other similar methods.

1. Introduction

The number of mobile phones in the world is increasing more and more. The mobile phones require high energy efficiency and high performance systems. SIMD extensions have been widely used in desktop for multimedia applications [1]. SIMD extensions offer high performance, high power consumption, and portability and are also suitable for mobile systems [2–4].

Automatic vectorization [5] is a compilation and optimization method using SIMD extension unit [6] of processor to realize data-level parallelism of a program, which can effectively improve the performance of the program. In the past few decades, in the field of automatic vectorization, many fundamental problems have been solved and many optimization methods have been proposed [7–11], which are adopted by mainstream compilers. In recent years, with the emergence of new SIMD instruction sets (AVX512 [12], SVE [13], and RISC-V vector instruction set [14]), SIMD technology is developing continuously and is becoming more and more powerful. From basic vector instructions to mask and variable-length vector instructions [15], the parallel length of SIMD instructions is also getting longer and longer. For example, SVE instructions currently support 2048-bit operands. The development of these technologies also brings new problems and challenges to the field of autovectorization [16, 17].

There are mainly two automatic vectorization methods: one is automatic vectorization method based on loop [8–10], which is aimed at realizing the parallelism between iterations of the loop. The other is the superword level...
parallelism (SLP) automatic vectorization method [11], which is aimed at realizing the parallelism of codes within basic blocks. Essentially, these two approaches are complementary. This paper mainly studies the SLP method.

The traditional SLP method is generally implemented in the isomorphic statements, and the operation data of the isomorphic statements are merged together and executed in parallel. The isomorphic statements [11] refer to two or more statements that have an isomorphic relationship with each other (these statements have the same number, type, and order of operations). The traditional SLP cannot vectorize the nonisomorphic statements effectively. In recent years, research on the SLP vectorization method for nonisomorphic statements has attracted increasing attentions [18–20].

The problem of vectorization of nonisomorphic statements was first proposed by Porpodas [18] at the 2015 CGO conference. In recent years, many researchers have proposed solutions to this problem from different perspectives, including hardware-specific instructions and expression-based solutions.

(1) The vectorization method of nonisomorphic statements based on hardware special instruction mainly uses SIMD special instruction to realize the vectorization of nonisomorphic statements. For example, the PSLP method [18] obtains the differential parts of nonisomorphic statements based on the dependency graph of statements and refers to each other and expands them by adding select instructions (select), respectively, so as to convert nonisomorphic statements into isomorphic statements. VeGen [21] implemented a compilation framework that uses non-SIMD instructions to realize automatic vectorization of nonisomorphic statements. Methods based on hardware special instructions are generally limited by the processor platform and introduce additional operating costs.

(2) The nonisomorphic statement vectorization method based on expression equivalence transformation mainly uses expression equivalence transformation to convert nonisomorphic statements that satisfy certain conditions into isomorphic statements, thereby creating conditions for the implementation of SLP. For example, the LSLP method [19] analyzes and processes multiple nonisomorphic statements with differences in the order of operations and rearranges the commutative operations and operands based on the commutative law when the conditions are suitable to obtain isomorphic statements. The SN-SLP method [20] is similar to the LSLP method. When the conditions are suitable, the operations and operands in the nonisomorphic statements are rearranged based on the equivalence relation of subtraction or division (for example: \(a - b - c = a - c - b\)), so as to obtain isomorphic statements. The methods based on the equivalent transformation of expressions generally have no special requirements on the processor hardware and do not introduce additional executing costs. This paper mainly investigates and studies this kind of methods.

The existing methods based on expression equivalent transformation (LSLP and SN-SLP) essentially deal with the vectorization problem of nonisomorphic statements with differences in the order of operations. In this paper, we study another class of nonisomorphic statement vectorization problem with different numbers of operations. Such nonisomorphic statements are common in scientific computing and multimedia applications. In addition, some statements with the same number of operations will be converted into statements with different numbers of operations during compilation due to some scalar optimizations (such as constant folding and strength reduction [22]), thus losing the opportunity of the opportunities of SLP vectorization. For example, \(a[0] = i - 0\) and \(a[1] = i - 1\). The two statements originally have the same number of operations, which can be used for SLP vectorization. However, the existing optimization compilers almost convert the first statement to \(a[0] = i\), making these statements have different numbers of operations, preventing SLP vectorization. We found that the similar situations are more common in the multimedia applications, especially in mobile system. However, simply turning off constant folding and strength weakening, while avoiding the above problem partially, removes the performance gain of the transformation itself for scalar optimizations.

In order to realize the analysis and vectorization of nonisomorphic statements with different numbers of operations, this paper proposes a new SLP vectorization method—the SLP-E method. The SLP-E method introduces and utilizes a new type of expression equivalent transformation, namely, equivalent extension transformation, which mainly uses the expression equivalent extension transformation relationship (such as \(x = x + 0\)) to expand the number of operations to bridge the difference in the number of operations of nonisomorphic statements, so as to convert them into isomorphic statements for vectorization.

Existing methods based on expression equivalent transformation (such as LSLP and SN-SLP) cannot solve the vectorization problem of statements with different numbers of operations, because the existing expression equivalent transformation can only solve the vectorization problem of nonisomorphic statements with different operation orders by adjusting the order of the original statement dependence graph from the topological structure level. The equivalent extended transformation modifies the number of nodes (for example, the equivalent extended transformation \(x = x + 0\)).

To perform isomorphic transformation on nonisomorphic statements with different numbers of operations, we need to focus on solving two problems:

**Problem 1:** identification of the isomorphic transformation objects, that is, how to identify the nonisomorphic statements with different numbers of operations that can implement isomorphic transformation in the set of program statements being processed. Compared with nonisomorphic
statements with different operation orders, the similarity of program structure between two or more statements in non-isomorphic statements with different numbers of operations is lower, and the isomorphic transformation involves the change of the number of operations, its optimized object recognition is more complex than the former. The traversal search method based on local similarity adopted by the existing LSLP and SN-SLP methods cannot identify the optimized object effectively, so new solutions need to be explored.

Problem 2: the realization of the isomorphic transformation, that is, how to bridge the difference in the number of nonisomorphic statement operations based on equivalent transformation, needs to achieve three objectives: (1) the number and type of each operation of the statement are the same after conversion; (2) program semantics remain unchanged after transformation; and (3) the overall performance evaluation of the program is improved after conversion. Because the change of the number of operations of the equivalent extension transformation will affect the vectorization transformation of the operation it depends on, LSLP and SN-SLP cannot determine the effect of program transformation on overall performance gains with the information of local operation type, so new solutions need to be found.

In order to solve the above problem 1, SLP-E first uses the similarity information of nodes in the data dependency graph corresponding to the processed statements to obtain the maximum common subgraph and then obtains the global differential graph between statements. For the global differential graph, it identifies the statements that can be optimized with the transformation features of expression equivalent extended relations.

In order to solve the above problem 2, SLP-E selects the equivalent extended transformation that can bridge the difference between the number and type of operations according to the operation type and arrangement location information in the differential graph of the optimized object statements and then decides whether to perform vectorization with the performance evaluation.

The equivalent extended transformation adopted by SLP-E supports a variety of transformation types. Most binary operations have their corresponding equivalent extended transformation expressions. For example, the addition operation can be extended by \( x = x + 0 \), the subtraction operation can be extended by \( x = x - 0 \), the multiplication operation can be extended by \( x = x \times 1 \), and so on. At the same time, these basic extensions can be combined to handle more types of programs, for example, two statements \( a[0] = b[0] \) and \( a[1] = b[1] \times 4 + 3 \). We can extend \( b[0] \) in statement 1 to \( b[0] \times 1 + 0 \), thus making the two statements isomorphic.

The main differences between the SLP-E method and existing SLP vectorization method for nonisomorphic statements are as follows: (1) the SLP-E method is based on the equivalent transformation of expressions. Compared with the method based on special hardware instructions, it has no special requirements for SIMD extended hardware and does not emit instructions with additional operating costs. (2) Compared with the existing methods based on expression equivalence transformation (LSLP and SN-SLP), the SLP-E method solves the problem of the vectorization of a new class of nonisomorphic statements.

In terms of specific analysis and transformation methods, this paper introduces more global information analysis to identify optimization opportunities and evaluate optimization benefits; the transformation method can not only adjust the original dependency graph from the topological level but also change the number of nodes.

This paper makes the following contributions:

1. The SLP vectorization problem of nonisomorphic statements with different numbers of operations is analyzed and proposed, and a SLP-E vectorization method is proposed to solve this problem. This method uses a new transformation method (equivalent extended transformation) and combines the structural characteristics of the data dependency graph of the statement to convert the nonisomorphic statements with different numbers of operations in the program into the isomorphic form, which extends the application scope of the SLP method and improves the vectorization capability of SLP.

2. In order to solve the identification problem of isomorphic transformation objects (problem 1), an analysis method based on the maximum common subgraph is proposed, using the similar information of the nodes of the data dependency graph corresponding to the statements; obtain the isomorphic part of the data dependency graph corresponding to multiple statements; and then obtain the differential part. It lays the foundation for further determining the isomorphic transformation object and equivalent extended transformation position. In order to solve the realization problem of isomorphic transformation (problem 2), a classification processing method was proposed, which carried out the operation and operand extension according to the operation type and layout information in the data dependency graph corresponding to the statements.

3. The SLP-E method was implemented on LLVM V11.01 and tested and evaluated based on benchmark sets such as SPEC CPU 2017. The results show that, compared with the SLP method implemented by LLVM (including LSLP and SN-SLP methods), the average performance of the SLP-E method in the kernel test and the overall test is improved by 43.9% and 3.8%, respectively.

2. Motivation

In 2000, Larsen and Amarasinghe [11] proposed the SLP vectorization method, regarding the consecutive memory access as the starting point of extending the isomorphic chain, combined with dependency analysis, through the "def-use" chain and "use-def" chain extension, to find the vectorizable isomorphic statements and execute them in parallel.
The SLP method can implement vectorization for isomorphic statements in basic blocks and effectively improve the performance of programs, which has attracted the attention and further research of many researchers.

Relevant achievements mainly focus on the research on building isomorphic chains based on isomorphic statements, specifically including two aspects: (1) how to build an isomorphic chain, including local greedy strategies and global strategies [23, 24]. The method of building an isomorphic chain with a local greedy strategy is to build an isomorphic chain from bottom to top based on the local greedy strategy, including isomorphic chain construction based on dynamic programming [25], automatic vectorization factor adjustment optimization [26], and clipping optimization of isomorphic chain based on real-time cost evaluation [27]. The method of building an isomorphic chain with a global strategy is to build an isomorphic chain based on a global search method, including seed selection optimization based on the principle of maximum reuse [28], hierarchical selection optimization of isomorphic chains from local to global [29], and isomorphic chain selection optimization based on integer linear programming [30]. (2) How to vectorize the statements in different basic blocks, such as vectorization of branch statements and loops [31]. The automatic vectorization method for branch statements is mainly based on the IF-conversion SLP method, which reduces the numbers of redundant instructions and mask instructions [32]. Vectorization methods for loops include SLP-oriented loop unrolling optimization [33–35], selection optimization of vector methods based on program parallelism features [36], and vector recognition optimization based on directed graph reachability [37]. In addition, SLP methods have also been applied in the fields of dynamic code conversion [38] and optimization of vector code in inline assembly form [39], etc.

The SLP method merges the operation data of isomorphic statements together and executes them in parallel. However, it terminates vectorization when encountering nonisomorphic statements, which causes certain limitations to the practical application of SLP. On the one hand, the ratio of nonisomorphic statements is more in actual programs: most of the intuitive form of program code written by programmers is nonisomorphic, and some compiler optimization algorithms sometimes destroy the original isomorphic statements in the program. On the other hand, there are some nonisomorphic statements that can be equivalently converted to isomorphic statements. Therefore, how to extend the SLP method to nonisomorphic statements has become one of the research directions of vectorization methods.

In recent years, there have been a variety of SLP extension methods for nonisomorphic statements (LSLP, SN- SLP, PSLP, etc.) However, as mentioned in the introduction of this paper, all the above methods have limitations. Methods based on hardwared special instructions are limited by the SIMD extended processor platform, or they will introduce additional operating costs, resulting in little or no benefits. The method based on expression transformation can only solve the vectorization problem of nonisomorphic statements with differences in the order of operations.

Figure 1 is a process diagram of a piece of program code processed by several different automatic vectorization methods. Among them, Figures 1(a) and 1(b) are the original code fragment (the number of operations of the statement is different) and its corresponding data dependency graph. Figures 1(d), 1(f), and 1(g), respectively, show the program conversion process diagrams based on various methods. Figures 1(c), 1(e), and 1(h) are vector grouping diagrams for program transformations; among them, the dotted rectangle represents the additional cost instructions that need to be generated for automatic vectorization. Benefit in the vector grouping diagram represents the benefit of automatic vectorization, that is, the difference between the performance improvement is little. However, the automatic vectorization is profitable, and the compiler implements the automatic vectorization, otherwise the automatic vectorization has no benefits, and the compiler does not perform program conversion.

The SLP method starts with C[i] and C[i + 1] which are consecutive memory accesses as seeds and extends the isomorphic statement through the def-use chain and the use-def chain; different types of operations (multiplication and addition) are found during the extension process, and further extension of the isomorphic chain is stopped. The compiler evaluates that vectorization yields no gains, so the original program is not transformed, and the vector grouping for the SLP method is shown in Figure 1(c).

As for the hardware-based SLP vectorization method, the VeGen method cannot implement SLP vectorization for the original program, because most processors do not directly support SIMD extension instructions for the original program vectorization. Figures 1(d) and 1(e) show the processing flow of the original program by the PSLP method. PSLP can vectorize the original program, but it needs to emit select instruction with extra cost, and the performance improvement is limited.

For the method based on expression transformation, although the LSLP method can reorder the operands of multiplication or addition, the reordered statements are still different in structure and cannot be vectorized. The grouping diagram is shown in Figure 1(c), the processing flow of the SN-SLP method is similar.

In fact, the original program in Figure 1(a) can be converted into an isomorphic form through a specific equivalence relation, as shown in Figures 1(f)–1(h); $A[i] + B[i]$ can be converted into $A[i] + B[i] + 0$; vectorization can bring more benefits (the value of Benefit is 4).

As can be seen from the above example, for nonisomorphic statements with different numbers of operations, existing vectorization methods cannot vectorize them or the performance improvement is little. However, the adaptive range of vectorization can be further improved by equivalent extension transformation. The SLP-E method is inspired by this discovery. The equivalent extended transformation is used to solve the SLP...
vectorization problem for nonisomorphic statements with different numbers of operations.

3. The Framework of SLP-E

The overall compilation framework of this paper includes preprocessing, the processing of SLP-E, and postprocessing, as shown in Figure 2. Preprocessing is to unroll the loop with small loop body before the processing of SLP-E, in order to increase the chances of vectorization; the processing of SLP-E is to vectorize the code that meets certain conditions; postprocessing is to convert the code into binary programs through register allocation and instruction generation, etc.

The processing of SLP-E consists of three parts: isomorphism transformation, code scheduling, and vector code generation. Isomorphic transformation is the equivalent transformation of nonisomorphic statements that satisfy the conditions into isomorphic statements. Code scheduling and vector code generation are to perform instruction scheduling on a code to generate a code in vector form.

Isomorphic transformation in the processing of SLP-E is the focus of this paper with extension transformation, which uses equivalent extension expressions to convert nonisomorphic statements in programs into isomorphic statements.

4. The SLP-E Method

In order to solve the problem for vectorization of nonisomorphic statements which contains different numbers of operations, The SLP-E method adopts a new transformation method, equivalent extension transformation, which makes the number of operations of these statements the same. Equivalent extension expression means if a binary operation ⊗, constant c, and y (y can be a variable, constant, or expression) satisfy y == y⊗c or y == c⊗y, then y⊗c or c⊗y are called the equivalent extension expressions of y. Most binary operations satisfy this expression.
SLP-ETransform.
Input: Graph1 is a data dependency graph corresponding to program1, Graph2 is a data dependency graph corresponding to program2.
Output: CSG which is the transforming representation graph for extended transformation.
1. \((\text{diff1}, \text{diff2}, \text{mcs}) = \text{MCSBasedDifferentialAna}(\text{graph1}, \text{graph2});\) // The common subgraph is used to find the difference between statements
2. \(\text{diffpairgraph} = \text{DiffPair}(\text{diff1}, \text{diff2}, \text{mcs});\) // Analyze where to insert an equivalent extension expression
3. \(\text{WHILE}(\text{diffpairgraph} \neq \text{NULL})\)
4. \(\text{diffpair} = \text{Pop}(\text{diffpairgraph});\)
5. \((\text{csg}, \text{hasdiffgraph}) = \text{EquivalentExtensionOfExpression}(\text{diff1}, \text{diff2}, \text{diffpair}, \text{mcs}, \text{csg});\) // Implement extended transformation
6. \(\text{IF}(\text{hasdiffgraph})\) // For non-isomorphic statements that still exist after extension, add selection instruction processing method [18] is used for conversion
7. \(\text{csg} = \text{AddSelect}(	ext{csg});\)

Algorithm 1: The transform function of the SLP-E method.

Operations have their corresponding equivalent extension expressions, such as \(y = y + 0, y = y - 0, y = y * 1, y = y/1,\) and \(y = y \ll 0.\)

Under certain conditions, converting \(y\) in a program statement into its equivalent extended expression can increase the number of isomorphic statements without introducing additional costly instructions, because these added instructions can be converted into vector form together with scalar code in the original program, without introducing additional costs in vector programs.

We use the differential information between statements and combine the method of pattern matching to perform equivalent extension and transformation of the program, which is divided into two steps: (1) identify the statements in the program code that can be transformed into isomorphic statements by extended transformation: find the differences between statements and then further identify and locate them through pattern matching; (2) implementation of extended transformation: according to the operations and the order of operation arrangement, the processing object is converted into its appropriate equivalent extended expression. Algorithm 1 shows the flow of the extended transformation.

4.1. Differential Analysis Based on Maximum Common Subgraph. In order to identify a statement that can be transformed by the isomorphism of the extension transformation, the SLP-E method needs to analyze the difference between the statements.

4.1.1. Basic Conception. Firstly, introduce the basic concepts of graphs. A graph is a data structure \((V, E)\), where \(V\) is a set of nodes and \(E\) is a set of edges. \(E\) is a set describing binary relations between nodes. For two graphs \(G1\) (\(V1, E1\)) and \(G2\) (\(V2, E2\)), if the two graph node sets \(V1\) and \(V2\) have a one-to-one mapping relationship and their edge sets \(E1\) and \(E2\) have a one-to-one mapping relationship, then \(G1\) and \(G2\) are said to be isomorphic. For two graphs \(G1'\) and \(G2'\), if they are subgraphs of graphs \(G1\) and \(G2\), respectively, and \(G1'\) and \(G2'\) are isomorphic, then \(G1'\) and \(G2'\) are called common subgraphs of \(G1\) and \(G2\). If \(G1'\) and \(G2'\) are the common subgraph with the largest number of nodes in \(G1\) and \(G2\), then \(G1'\) and \(G2'\) are called the maximum common subgraph of \(G1\) and \(G2\) [40].

A data dependency graph is a representation graph of the program, which is used to describe the data dependence relationship between statements, instructions, or operations/operands in the program [41]. For two data dependency graphs \(G1(V1, E1)\) and \(G2(V2, E2)\), if their maximum common subgraphs are \(G1'(V1', E1')\) and \(G2'(V2', E2')\), \(G1\) and \(G2\) minus \(G1'\) and \(G2'\), respectively, the remaining part is called the differential graph of \(G1\) and \(G2\) (relative to each other), which will be expressed as \(G1''(V1'', E1'')\) and \(G2''(V2'', E2'')\). \(V1'' = V1 - V1', E1'' = E1 - E1', V2'' = V2 - V2',\) and \(E2'' = E2 - E2'\). An example of the differential graph is shown in Figure 3(c). Sometimes, the nodes of the edge in the differential graph are not in it; we call this kind of edges as “outer edge.”

Sometimes, the differential graph is not connected, and each maximum weakly connected graph in the differential graph is called a differential subgraph. For the two data dependency graphs \(G1\) and \(G2\), \(G1'\) and \(G2'\) are the maximum common subgraphs of \(G1\) and \(G2\). If there are two differential subgraphs \(G1''\) and \(G2''\) (they are subgraphs of \(G1\) and \(G2\), respectively), the set of connecting nodes between \(G1'\) and \(G1''\) is \(V1\) (nodes in \(V1\) are all nodes of \(G1'\)), and the set of connection nodes between \(G2'\) and \(G2''\) is \(V2\); if there is one-to-one mapping relationship between elements of \(V1\) and \(V2\), each pair in node set \(V1\) and node set \(V2\) matches the nodes in the mapped node pair (denoted as node \(V1j\) and node \(V2j\)), which are the nodes in graph \(G1\) and graph \(G2\), respectively. The direction of the edge connected by node \(V1j\) and differential subgraph \(G1''\) is the same as that connected by node \(V2j\) and differential subgraph \(G2''\); then \(G1''\) and \(G2''\) are called corresponding differential subgraphs, such as diff1 and diff2 in Figure 3(c).

For two nodes \(S_i\) and \(S_j\) from different data dependence graphs, if the following four conditions are met, \(S_i\) and \(S_j\) are called matching node pairs, denoted as \(<S_i, S_j>:\)
(1) $S_i$ and $S_j$ are operations within the same basic block.

(2) If $S_i$ and $S_j$ are both operations, $S_i$ and $S_j$ should be the same type, and the type of their input operands and output operands should be the same.

(3) If $S_i$ and $S_j$ are both memory access operations, $S_i$ and $S_j$ should be consecutive, and the type of their input operands and output operands should be the same.

(4) $S_i$ and $S_j$ are independent of each other, there is no direct or indirect dependency, and they can be scheduled in parallel [30].

For a set consisting of one or more matching node pairs, if there are two matching node pairs in the set, they are denoted as $P_i$ and $P_j$ which satisfy the following: (1) $P_i$ and $P_j$ do not have a dependency cycle; (2) $P_i$ and $P_j$ do not contain the same node; then, these are called a matching node pair group.

For the two nodes in the matching node pair, the ratio of the number of nodes of the same type in the two graphs and the total number of nodes in the two graphs is taken as the matching score of the matching node pair, in order to describe the similarity between the two nodes. The specific method is that find the subgraph with these two nodes as the root in the data dependency graph and then take the ratio of the number of nodes of the same type to the total number of nodes in the two subgraphs as the matching score, as shown in formula (1); $M$ and $N$ are the number of nodes in the two subgraphs, $I$ is the total number of types of nodes in the two graphs, and $X_i$ and $Y_i$ are the number of nodes of the $i$-th type in the two subgraphs.

$$\text{Score} = \frac{M + N - \sum_{i=1}^{I} |X_i - Y_i|}{M + N}.$$  (1)

4.1.2. Common Subgraph Method. The maximum common subgraph problem is a NP-hard problem [42]. In this paper, a heuristic method called NM-MCS method is proposed to obtain the maximum common subgraph. The NM-MCS method uses the matching score of the matching node pair to guide the construction of the relatively optimal matching node pair group and then further finds the matching edge set and the maximum common subgraph. Specific steps are as follows:

(1) Find the matching node pairs and matching scores of the data dependency graph corresponding to the statements; refer to lines 1-6 of Algorithm 2. Firstly, traverse the nodes of the data dependency graph corresponding to the statements. If the two nodes are from different data dependency graphs and can form matching node pairs, they are stored in the matching node pair set $P$. Secondly, the matching nodes in $P$ are scored according to formula (1) and sorted from high to low. Thirdly, select the first $K$ matching node pairs in $P$, each matching node pair is regarded as a matching node pair group $m_i$, and insert all $m_i$ into the set $C_i$ of the matching node pair group. We use $C_n$ to represent the set of matching node pairs; there are $|C_n|$ numbers of $m_i$, $m_{i(C_n)}$ is the lowest matching node pair in $C_n$.

(2) Find the relatively optimal pair of matching nodes, as shown in lines 7-18 of Algorithm 2. The matching score of matched node pairs is used to guide the iterative construction of the relatively optimal matched node pair group.

(3) Insert the elements in $P$ to each matching node pair of $C_i$ in turn to get $m_i$. When $m_i$ meets the constraint condition of the matching node pair group, $m_i$ is inserted into $C_{i+1}$, and each matched node in $C_{i+1}$ was scored separately. The group score is the sum of the scores of all matching node pairs in the group, denoted by $d_{m_i}$. The elements of $C_{i+1}$ are sorted from high to low according to the score, and only the first $K$ matching node pairs are reserved in $C_{i+1}$. The above steps are iterated until $C_{n+1}$ is empty, and the uppermost element in $C_n$ is the relatively optimal matching node pair.

(4) Find the maximum common subgraph; see lines 19-20 in Algorithm 2. Traverse the edge of the node connection in the matching node pair group. If the start and end nodes of the two edges match, respectively, then the two edges match. After the matching edge is obtained, the maximum common subgraph is further obtained according to the matching node pair and the matching edge set.

The following describes the processing flow of the NM-MCS method with an example of research motivation, as
NM-MCS

Input: Graph1 is a data dependency graph corresponding to program1, Graph2 is a data dependency graph corresponding to program2. K is the maximum value of the candidate queue.

Output: The maximum common subgraph of Graph1 and Graph2.

1. $P = \text{CollectMatchingNodePair}()$; // Collect all possible matching node pairs to form a set $P$
2. $d = \text{CalScore}(P)$; // Compute the score of each element in $P$
3. $C_i = P[1:k]$; // $C_i$ contains the first $k$ values of $P$
4. $d_{\text{worst}} = d(m_{C_i})$;
5. $n = 1$;
6. WHILE ($n < \text{max}[V_a, V_b]$); // $V$ represents the total number of nodes in the data dependency graph
7. $C_{n+1} = \varnothing$;
8. FOR ($m_i = m_{n+1}(x,y)$); // Whether the $m_i$ satisfies the constraint condition of matching node pair group
9. $C_{n+1} = \text{AddElement}(m_i)$; // Insert $m_i$ into $C_{n+1}$
10. $C_{n+1} = \text{DeleteElement}(k+1, |C_{n+1}|)$;
11. $d_{\text{worst}} = d(m |C_{n+1}|)$;
12. IF ($C_{n+1} = \varnothing$)
13. $\text{MatchingNode} = C_n(0)$;
14. $\text{MatchingEdge} = \text{GetEdge}($MatchingNode$)$;
15. $\text{MCS} = \text{GetGraph}($MatchingNode$, $MatchingEdge$)$;
16. ELSE 
17. $n = n+1$;

Algorithm 2: NM-MCS method.

(a) Input program (b) Data dependency graph

double A[SIZE];
double B[SIZE];
double C[SIZE];

C[i] = A[i] * B[i];
C[i + 1] = A[i + 1] * B[i + 1] + 5;

(c) Score of Candidate matching node pairs

{x1, y1}, {x2, y2}, {x3, y3}, {x4, y6}

(e) Group of matching node pair

Figure 4: An example of process of the NM-MCS method.
shown in Figure 4. Figures 4(a) and 4(b) are the original program and the corresponding data dependency graph, respectively. Figure 4(c) shows the matching scores of elements in set $P$. Figure 4(d) is a diagram of the conversion process of NM-MCS for the original program. Firstly, the NM-MCS method inserts the first $k$ matching node pairs in $P$ into the set $C_1$. Secondly, insert the elements in $P$ into the matching node pairs in the set $C_1$ in turn. If the element in $C_1$ met the conditions for matching node pairs, insert it into the set $C_2$. Figure 4(e) shows the relatively optimal matching node pairs obtained by the NM-MCS method. Figure 4(f) is the maximum common subgraph obtained by the NM-MCS method. Figure 4(g) is the differential graph, the remaining part of the data dependency graph after subtracting the maximum common subgraph.

4.2. Optimization Object Recognition for Equivalent Extended Transform. In order to carry out isomorphic transformation of nonsisomorphic statements with different numbers of operations based on extended transformation, it is necessary to identify the statements that can be an isomorphic transformation through extended transformation in the program code. To determine the specific position of the equivalent expansion transformation in the program, this paper introduces the concept of the extensible differential subgraph pair.

Assuming that graph $a$ and graph $b$ are subgraphs of the data dependency graph corresponding to two statements in the program, if $a$ and $b$ satisfy the following three conditions, the graph pair composed of $a$ and $b$ is called an extensible differential subgraph pair, denoted by $<a, b>$.

1. The graph $a$ only has one edge

2. The operation types of nodes in $b$ only include addition, subtraction, multiplication, division, or shift operations. If a noncommutative operation is included, the first operand node has no outer edge

3. Graph $a$ and graph $b$ are the corresponding differential subgraphs

The extensible differential subgraph is used to describe the statements that can effectively perform the isomorphic transformation by the extensible transformation and to locate the optimized position. According to the definition, the two elements $<a, b>$ in the extensible differential subgraph pair have the same number, direction, and position as the connection nodes of the common subgraphs, and graph $a$ has only one edge; all nodes in $b$ have equivalent extended expressions corresponding to the same operation. Under this condition, we can convert graph $a$ into a graph which is isomorphic to graph $b$ by performing an equivalent extension transformation, thus reducing the overall differential between different statements.

The SLP-E method searches the extensible differential subgraph pairs according to the above three constraints. Firstly, according to the differential graph of the statements obtained in the previous section, the differential subgraph is obtained according to the definition of the differential subgraph. Secondly, search in group 1 and group 2 according to condition 3; when the elements in the two graphs come from the subgraphs of the data dependency graphs corresponding to different statements and are the corresponding difference subgraphs, then the two graphs can form a pair of extensible differential subgraphs.

The positions of multiple equivalent extended transformations in program are based on the extensible differential
subgraph; a specific example is shown in Figure 5. Figure 5(a) shows statements $S_1$ and $S_2$, where $\odot$ represents binary operations; $M_i$ represents operands; $X_1$, $X_2$, and $X_3$ represent variables, constants, or expressions; and the red part represents the difference between $S_1$ and $S_2$. Figure 5(d) is a pair of extensible differential subgraphs depicting the differential part of $S_1$ and $S_2$, representing the positions of multiple equivalent extension transformations.

4.3. Implementation of Equivalent Extended Transformation. In the previous section, by looking for the pair of extensible differential subgraphs, we identified the statements in the program code that can carry out isomorphic transformation through extended transformation and located the position of implementing this transformation. This section describes how to perform an extension transformation to convert nonisomorphic statements into isomorphic statements. From the perspective of graph transformation, this problem can be further transformed into the following: how to deal with the extendable difference subgraph pair $<a, b>$ and extend and transform the graph $a$ into graph $a'$, so that graph $a'$ and graph $b$ are isomorphic, while ensuring the functional and semantic equivalence of graph $a'$ and graph $a$. In this paper, according to the operation types and arrangement location information in the differential graph of optimization object statements introduced in the previous section, the equivalent extension that can bridge the difference of operation number and type is selected. The extended transformation in SLP-E consists of two transformation operations:

1. Transform graph $a$ into graph $a'$, so that graph $a'$ and graph $b$ contain the same operation and arrangement of nodes in order to make the graph $a$ and graph $b$ both isomorphic to each other

2. Add leaf nodes of specific operands to graph $a$ according to the type and arrangement order of the operations, in order to make the semantics of the converted graph $a'$ and graph $a$ equivalent

<table>
<thead>
<tr>
<th>Type</th>
<th>Operation</th>
<th>Insert the leaf node to direct nodes</th>
<th>Insert the leaf node to indirect nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>The value of operand</td>
<td>The value of the first operand</td>
</tr>
<tr>
<td>Single 0 node</td>
<td>Add</td>
<td>0</td>
<td>Output of the node</td>
</tr>
<tr>
<td></td>
<td>Sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Left shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Right shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single 1 node</td>
<td>Mul</td>
<td>1</td>
<td>Output of the node</td>
</tr>
<tr>
<td></td>
<td>Div</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Operands of leaf node list.

Figure 6: Speedup ratios of various methods on kernel functions.
We denote the input and output nodes of a connected to the maximum common subgraph as $V_{in}$ and $V_{out}$, respectively, as shown in Figure 6(d). The data dependency graph dealt with in this paper should be a tree structure, and programs corresponding to this structure of data dependency graph are common in reality. In the tree structure, there is only one and only path from $V_{in}$ to $V_{out}$, which is recorded as $V_{path}$. If the data of the maximum common subgraph corresponding to the statements remains unchanged along each operation node on the $V_{path}$, then graph $a'$ and graph $a$ are functionally semantically equivalent. If the data dependency graph is not a tree structure, for example, there are multiple used nodes, this paper copies the node to the multiplicity of the node, so that each node contains only one use and then converts the nontree structure into a tree-structured graph.

This paper found that the way to add a leaf node of a specific operand to the tree corresponding to the statements is a tree-structured graph. We regard this node type on the $V_{path}$ as direct nodes and regard the other nodes as indirect nodes.

The way a leaf node of a specific operand is added to a direct node is only relevant to the type of operation on that node. The SLP-E method adds another input operand node under the condition that the operation type of the direct node and an input operand node on the $V_{path}$ are known, so that the input and output operands of the node on the $V_{path}$ are the same. In this paper, the nodes are divided into single 0 and single 1 nodes according to the operation type, and the operand nodes of constant 0 and 1 are, respectively, inserted into the direct nodes of these two types, as shown in the direct node insertion leaf node column in Table 1.

The way a leaf node of a specific operand is added to an indirect node depends on both the operation type and the output value of the node. The SLP-E method inserts the leaf node of the operand with the same value as the output at the position of the first operand. For the position of the second operand, insert the leaf node of the specific operand according to the operation type of the indirect node, as shown in the indirect node insert leaf node column in Table 1.

The specific steps of the equivalent extension transformation in the SLP-E method are described below:

1. Determine whether all the elements in the extensible differential subgraph pair $<a, b>$ to be processed are in the form of trees. If not, perform tree transformation for these with copying nodes.

2. Insert a nonleaf node into graph $a$. Insert the operation node and the corresponding edge of the same type and arrangement order as graph $b$ into graph $a$ (the leaf node of $b$ is not included at this time).

3. Insert the leaf node of the direct node into graph $a$. Traverse the nodes along the $V_{path}$ from the node $V_{in}$ to the node $V_{out}$, while marking the traversed nodes and edges. If the traversal node is connected to the outer edge, insert the operand leaf node of the specified constant value into it (the inserted node does not contain $V_{in}$ and $V_{out}$); refer to Table 1 for the value of the leaf node operand inserted by the direct node, otherwise, assign the constant value specified by the direct node in Table 1 to the edge connecting the input operand to the node.

4. Insert the leaf node of the indirect node to the edge $a$.

Taking the $V_{out}$ as the root node, use the method of bottom-up breadth-first traversal, based on the value of the marked edge in step 3, as shown in Table 1, for inserting the values of the leaf node operands into the indirect node.

The following illustrates the processing flow of equivalent extended transformation in the SLP-E method with the example of research motivation, as shown in Figure 7. Figures 7(a) and 7(b) show the statements S1, S2, and the corresponding data dependency graphs, respectively. Figure 7(c) shows the extensible difference subgraph pair.
Table 2: Description of the kernel functions.

<table>
<thead>
<tr>
<th>Serial number</th>
<th>Kernel</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>jdct-ifast</td>
<td>Different numbers of operations</td>
<td>Discrete cosine transform (MediaBench)</td>
</tr>
<tr>
<td>2</td>
<td>calc-pair-energy</td>
<td>Different numbers of operations</td>
<td>Biological system simulation (SPEC CPU 2006 444)</td>
</tr>
<tr>
<td>3</td>
<td>gl_render_vb</td>
<td>Different numbers of operations</td>
<td>OpenGL function library (SPEC CPU 2000 177)</td>
</tr>
<tr>
<td>4</td>
<td>ssim-end1</td>
<td>Different operation types</td>
<td>Video compression (SPEC CPU 2017 625)</td>
</tr>
<tr>
<td>5</td>
<td>Box-UVCoord</td>
<td>Different operation types</td>
<td>Image ray tracing (SPEC CPU 2006 453)</td>
</tr>
<tr>
<td>6</td>
<td>Start-pass-fdctmgr</td>
<td>Different operation types</td>
<td>Image compression (MediaBench cjpeg)</td>
</tr>
<tr>
<td>7</td>
<td>Boy-surface</td>
<td>Different orders of operation</td>
<td>BoySurface generation algorithm (SPEC CPU 2006 453)</td>
</tr>
<tr>
<td>8</td>
<td>vsumsqr</td>
<td>Different orders of operation</td>
<td>Sum of powers (SPEC CPU 2006 453)</td>
</tr>
<tr>
<td>9</td>
<td>mult-su3-nn</td>
<td>Different orders of operation</td>
<td>Quantum mechanics (SPEC CPU 2006 433)</td>
</tr>
</tbody>
</table>

(a) Code of jdct-ifast

```
```

(b) Code of calc-pair-energy

```
pli[0] = j; vlist[0] = i - 3; 
pli[1] = j + 1; vlist[1] = i - 2; 
```

(c) Code of gl\_render\_vb

```
fmb[0] = qtbl->quantval[0] * aanscalefactor[4] * 1;   
```

5. Evaluation

To verify the effectiveness of our method, we implemented the SLP-E method on the LLVM v11.01 compiler. SPEC CPU 2017/2006/2000 and MediaBench2 [43] benchmarks which have some media and artificial intelligent tests widely used in mobile system were used to test the proposed method from two aspects of the kernel test and overall test, respectively, and compared with the SLP method implemented by LLVM itself (abbreviated as LLVM-SLP). The SLP methods implemented in the current LLVM include LSLP and SN-SLP methods, and these methods will not be compared with our method in this paper separately.

The CPU model used in this experiment is Intel i7-4790; the main frequency is 3.2 GHz; it contains 4 physical cores; each physical core can support 2 logical cores; 32 KB of L1D cache (8-way, 64 B/line), 256 KB of L2 cache (8-way, 64 B/line), and 8 MB of L3 cache (shared memory) and supports AVX2, AVX1, and SSE vector instruction set. The tested memory is 20 GB, and the operating system is Ubuntu 20.04.

5.1. Kernel Test. Kernel function tests are used to test the optimization capabilities of different auto-vectorization methods for specific types of programs. Representative code fragments of the kernel were extracted from the SPEC CPU and MediaBench benchmark sets, which are mainly computationally intensive codes in the multimedia field and scientific computing field. These kernels contain more non-isomorphic statements, including typical differences between statements in a program, including different types of operations, different numbers of operations, and different orders of operations, as shown in Table 2. In this paper, the same kernel is executed multiple times in the loop, so that the overall execution time exceeds 10 minutes, which is used to reduce the impact of environmental factors on performance data.

In this paper, the execution time of LLVM-SLP and SLP-E methods on the kernel test program was tested, and the speedup of their closing SLP optimization and opening O3 optimization was calculated relative to LLVM, which was represented by O3.

Both LLVM-SLP and SLP-E turn on specific optimizations based on the above O3 optimization conditions. The performance results are shown in Figure 6, where the horizontal axis represents the test program, and the vertical axis represents the performance speedup of different methods relative to O3 on the test program. The average speedup of the LLVM-SLP method and the SLP-E method is 1.23 and 1.77, respectively.

The following is an analysis of the test results of LLVM-SLP and SLP-E on the compilation and optimization of actual application kernel functions for three types of
programs with different numbers of operations, different types of operations, and different sequences of operations.

For programs with different numbers of class 1, LLVM-SLP cannot effectively improve the performance of the program, while SLP-E can effectively improve the performance of the program. As for the gl_render_VB function, its core code fragment is shown in Figure 8(a); LLVM-SLP fails to implement vectorization, because the compiler converts the fourth statement \( \text{vlist}[3] = i - 0 \) into \( \text{vlist}[3] = i \) through constant folding optimization before vectorization, converting the original program into a non-isomorphic statement, which is not vectorized. SLP-E converts the \( i \) of the fourth statement into \( i - 0 \) through extension transformation and then converts the original program into isomorphic statements and implements vectorization for the program, which effectively improves the performance of the program. The calc_pair_energy and jdct-ifast codes are both similar.

For programs with different types of operations in class 2, LLVM-SLP and SLP-E fail to significantly improve performance for this class of programs. Neither LLVM-SLP nor SLP-E implement vectorization for the ssim_end1 function, and there was little difference in the performance of their optimized programs. The compiler converts the first division into a multiplication in the box_UVCoord function before autovecotorization, so that the two statements are converted into nonisomorphic forms. Both LLVM-SLP and SLP-E implement vectorization for this program, but their performance did not improve. By analyzing the assembly code, we found that they both implemented vectorization by generating vector division, vector multiplication, and recombination instructions, and the actual performance of these instructions was not much different from that of scalar programs. For the start_pass_fdctmgr function, the LLVM-SLP and SLP-E methods implement vectorization, which effectively improves the performance of the program.

The vectorization method tested in this paper can effectively improve the performance of class 3 programs with different operation statements, and the difference between with each other is very small. The boy-surface and vsumsqr functions contain many commutative operations. The vectorization methods tested in this paper can be vectorized successfully. LLVM-SLP and SLP-E (SLP-E is an extension optimization based on LLVM-SLP) can be rearranging their order which allows the compiler to find more isomorphic statements, which further improves program performance. The multi-su3-nn function contains many additions and subtractions, and the vectorization methods tested in this article can be automatically vectorized.

According to the kernel function test results, we found that for programs with different types of operations, SLP-E has a significant improvement effect compared with the existing vectorization methods of nonisomorphic statements. For the other two types (different operation types and different operation statements), the optimization effect is basically the same as the existing method, because SLP-E performs extended transformation on the program, which complements the difference in the number of operations.
operations in the program and is suitable for the vectorization of some programs with different types of operations. For the other two types of programs, SLP-E cannot use extended transformation. To further improve the performance of the program, the same processing method as LLVM-SLP is also used.

5.2. Overall Test. In this paper, the SPEC CPU and Media-Bench2 test sets were used to test the overall speedup of LLVM-SLP and SLP-E methods. The benchmark test sets were executed for 12 times, and the arithmetic mean value was taken for the performance test data. The overall speedup ratio is shown in Figure 9, where the horizontal axis represents the test program, and the vertical axis represents the speedup ratio of the optimization method relative to O3 optimization on the overall benchmark test set. The average speedups of the LLVM-SLP method and the SLP-E method are 1.07 and 1.11, respectively (the previous LSLP [19] and SN-SLP [20] compared with LLVM-SLP; the overall performance improvement is less than 4%). The overall performance improvement of the SLP-E method is better than that of the LLVM-SLP method. SPEC CPU and Media-Bench2 are important test sets in the field of compiler optimization. Many of the test programs derived from scientific computing and multimedia applications have been studied and optimized for many years [30], and it is still valuable to be able to improve their performance by optimizing the compiler.

We found that in the overall performance test, there are many programs that can effectively trigger (can trigger optimization and get performance gains) the extended transformation optimization of SLP-E, a total of 358, covering a variety of extended transformation forms, for example, $x = x + 0$, $x = x - 0$, and $x = x < 0$. In these triggered programs, some statements have a different number of operations, and some statements originally have the same number of operations, but the compiler’s constant folding optimization converts them into statements with different numbers of operations before vectorization. Constant folding optimizations and vectorization optimizations are analyzed and transformed independently in LLVM. Constant folding optimization does not take into account vectorization optimization and sometimes breaks the original isomorphic statement, thereby hindering the vectorization optimization.

The main reason why SLP-E can effectively improve the performance of some test programs is that by extending the transformation, the scope of application of the isomorphism transformation is extended, and the ability of SLP vectorization is improved, such as 433.milc, 453.povray, and 623.xalanchmk_s. These programs contain more expressions of types such as "$x + 0$", "$x \cdot 1$", and "$x < 0$.” Before vectorization, the compiler removes the expression and converts the isomorphic statements into the nonisomorphic statements. The SLP-E method converts nonisomorphic statements into isomorphic statements through equivalent extension transformation and implements automatic vectorization of the program.

We found that some programs after vectorization performed worse than scalar programs in the overall test, such as the 410.bwaves and 416.gamess programs. By analyzing the LLVM evaluation model and compiling the assembly code generated by the program with different optimization methods, we find that the performance degradation of the program is caused by the deviation between the LLVM static evaluation performance acceleration ratio and the actual performance of the program, rather than the SLP-E optimization. LLVM fails to not fully consider memory access factors and scalar instruction level parallelism in superscalar processor and determines that the program transformation which originally has no benefit is beneficial and then carries out inappropriate automatic vectorization, which leads to the degradation of program performance. For the 410.bwaves and 416.gamess test programs, the program performance optimized by the SLP-E method is not lower than that of the LLVM-SLP method.

6. Conclusions and Future Works

Mobile computing platforms need high performance and high energy consumption characteristics. SIMD extensions are suitable for mobile computing platforms. This paper proposes a SLP autovectorization method based on equivalent extension transformation (SLP-E), which solves the problem of identification and isomorphism of isomorphic transformation objects. The conversion implementation problem extends the scope of application of SLP and improves the vectorization processing capability of SLP. In this paper, the SLP-E method is implemented on the LLVM compiler and verified based on the test set such as SPEC CPU 2017. The experimental results show that, compared with the SLP method implemented by LLVM itself (including methods such as LSLP and SN-SLP), the SLP-E method has an average performance improvement of 43.9% and 3.8% in the kernel function test and the overall test, respectively.

In the follow-up work of this paper, in addition to improving and optimizing SLP-E method, more program equivalent transformation and corresponding vectorization transformation methods are sought and mined to realize SLP vectorization of nonisomorphic statements, so as to further improve the vectorization capability of SLP. In addition, how to use multiple equivalent transformations in SLP will also be explored and studied.

Data Availability

The data included in this paper are available without any restriction.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

References


