LCSSA optimization for vectorization recognition rate improvement

Mengyao Chen*, Yunda Chai and Jiandong Shang

1Software Engineering, Zhengzhou University, Zhengzhou, Henan, 450000, China
2Electronics and Communication, Zhengzhou University, Zhengzhou, Henan, 450000, China
*Corresponding author’s e-mail: zzucmy@gs.zzu.edu.cn

Abstract: LLVM converts the loop to LCSSA during the loop transformation process. The loop of LCSSA has better locality and can facilitate other optimizations of the loop, but at this stage LLVM does not support some special LCSSA loops in the automatic vectorization process, in the case of external use of non-array and non-inductive variable instructions in the loop, automatic vectorization cannot be performed, which makes the compiler lose some optimization opportunities for automatic vectorization. In response to this problem in LLVM, this paper proposes an algorithm to reconstruct PHI nodes. By adding new basic blocks, reconstructing the value of PHI nodes to eliminate the influence of external use in the loop on automatic vectorization, so that the loop can be automatically vectorized. Improve LLVM's automatic vectorization capabilities. Through the test on the TSVC test set, the vectorized recognition rate has increased by 23%.

1. Introduction
In the process of modern compiler optimization, autovectorization is one of the main optimization methods. In the process of autovectorization, cyclic vectorization at the cyclic Level and SLP (Superword Level Parallelism) at the basic block Level are commonly used autovectorization methods. In the cycle automatic vectorization process, there are various optimization methods. For some cycles that cannot be directly vectorized, the method of changing the order of instructions can be adopted to improve the vectorization capability [1]. For the cycle that can be optimized, optimization performance can be improved by adjusting optimization options [2]. In addition, the change and processing of loops in automatic vectorization of loops is also one of the main means to improve the automatic vectorization capability of compilers [3-6].

LLVM-related research is currently a research hotspot at home and abroad. At present, a parallel calculation method based on the intermediate layer IR of the LLVM compiler has been proposed [7]; the vectorization method of the tail loop generated by the LLVM automatic vectorization [7] 8]. In addition to some optimization methods for loops, there are also optimization methods by controlling the size of code generation [9]; in terms of LLVM memory access, some documents also propose optimization by automatic reorganization of data in registers [11-13].

Most of the optimization of LLVM in the mid-range is based on the generated intermediate layer IR. In the process of converting high-level language to IR, LLVM will convert the loop into the LCSSA (loop-closed SSA) form to improve the optimization opportunity of the loop. The LCSSA cycle refers
to the closed loop of the SSA form. On the basis of the existing SSA form of the cycle, the outside of the cycle does not use the SSA definition inside the cycle. This creates a boundary between the loop and the outside of the loop to a certain extent, and makes the loop itself more independent. Now many compilers, including LLVM and GCC, generate loops in the form of LCSSA. The benefits of this can provide convenience for various optimizations including value range analysis, final value replacement, inductive variable analysis, and loop expansion. However, this transformation will affect whether the compiler can automatically vectorize the recognition loop to a certain extent.

Aiming at the problem of LLVM’s automatic vectorization recognition of LCSSA form loop, this paper designs an automatic vectorization recognition method based on LCSSA form loop on the basis of the existing automatic vectorization function of LLVM. It can solve the problem of support for automatic vectorization of LCSSA loops when the instructions with non-inductive variables in the loop are used externally, and improve the automatic vectorization ability of LLVM.

2. LCSSA loop in LLVM

2.1 LCSSA cyclic transformation

In the IR transformation stage of LLVM, LLVM reads the information of the circular basic block in the IR and transforms the circular basic block into the LCSSA form. Separate the original loop from other basic blocks by inserting PHI nodes. The loop shown in Figure 1-A will be transformed into the form shown in Figure 1-B after LCSSA recognition, and the original loop will be transformed into the LCSSA form at this time.

![Figure 1 LCSSA cycle conversion](image)

After adding the PHI node, if the data in the original loop is used externally, the use outside the loop must revalue the value in the loop through the PHI node. In the identification process of the original loop basic block, LLVM will first check whether the instructions in the loop are used externally. LLVM adopts a fast search method. First, it will skip some instructions that are not used, next, we will search again in the instructions that have been filtered, identify the parts of the use chain outside the loop and collect their exit blocks, and add PHI nodes to the exit blocks according to the domination of the export blocks, the new PHI node will change the original definition-use chain, and the original use of the value in the loop will become the use of the PHI node. LLVM will update the SSA instruction at this stage to adjust the dominance between statements. To complete the LCSSA cycle transformation.

2.2 Insufficiency of LCSSA cyclic transformation

Converting the original loop to the LCSSA form can improve the locality of the loop, but this loop conversion also has some problems. In the LoopSimplify stage, if the LCSSA loop conversion is used, some loops may not be simplified. If the exit block of the loop L1 is the head block of the loop L2, when the LCSSA transformation is performed at the same time as L1 and L2, the LCSSA transformation of L1 will insert a new PHI node in L2. This change may destroy the LCSSA format of L2, and the use of variables in the L1 loop may exist outside the L2 and L2 loops at the same time.

In addition, the LCSSA loop conversion improves the locality of the loop, which means that the loop will be relatively independent, and the instructions inside the loop do not need to consider their
state outside the loop, which makes some instructions may be optimized to eliminate or expand. The
vectorization operation on the array in the loop will not affect the use of the array outside the loop, but
the vectorization extension for some scalar instructions may affect the entire use chain. The loop
shown in Figure 2 performs automatic vectorization transformation while performing LCSSA
transformation. The scalar instruction \( x \) in the original loop becomes a vector instruction. If the PHI
node is still used to read the value of the vector \( x \) inside the loop for the use of the scalar \( x \) outside the
loop, this will obviously cause an error.

\[
\begin{align*}
\text{for } (\ldots) \{ & \\
& X = \ldots \rightarrow X[n:n+4]=\ldots \\
& X2 = \phi (X) \rightarrow X2=\phi (X) \\
& \ldots = X2 + 4
\end{align*}
\]

Figure 2 Vectorization influence

In view of some situations that may arise, the optimization strategy of LLVM is not perfect. When
analyzing the legality of automatic vectorization, only the external use of inductive variables can be
processed, and the case where the instructions of non-inductive variables in the loop are used
externally is excluded, and the loops with non-inductive variable instructions for external use are not
vectorized. Doing so ensures the correctness of external values, but also loses the ability to vectorize
such loops. In response to this problem in LLVM, this paper proposes an automatic vectorization
method for LCSSA loops, which can vectorize externally used loops that contain non-inductive
variables in the loop, and effectively improve the vectorization recognition rate of LLVM.

3. LCSSA loop optimization

3.1 LCSSA loop legality
In the LLVM vectorization legality stage, LLVM will perform vectorization legality analysis on loops
that have undergone some simple loop transformations, and analyze the use chain of instructions
inside the loops. At present, LLVM only supports the use of inductive variables outside the loop. When
the use chain of other instructions reaches the outside of the loop, as shown in Figure 3, LLVM will
not vectorize the loop.

LV: Found an outside user for %sub14.lcssa = phi float [ %sub14, %for.body9 ]
LV: Can't vectorize the instructions or CFG

Figure 3 LLVM vectorization situation

In response to this problem, in the vectorization legality check stage, based on the recognition
algorithm used outside the instruction shown in the original algorithm 1, we add a new vectorization
legality judgment algorithm to determine the loop vectorization type in advance before considering
whether the loop has external use, and adjust the use relationship of the instructions inside and outside
the loop according to the vectorization type.

Algorithm 1: LLVM external use recognition algorithm

\begin{verbatim}
hasOutsideLoopUser()
//Instructions are not allowed for external use types
if(!allowed(Inst)){
  for(U:Inst->users){
    UI=cast<Inst>(U)
    //Instructions exist for external use
    if(!loop->contains(UI))
  }
}
\end{verbatim}
The loop generally has two vectorized states:

1) When the number of loop iterations cannot be divisible by the vectorization width, there is a tail loop in the vectorized loop, and the original loop is transformed into a vector loop and a scalar loop. At this time, the exit block of the vectorized loop is a scalar, and the instructions used inside and outside the loop depend on the instructions generated by the tail loop. Because the tail loop is still a scalar, the instruction has no effect on its external use relationship.

2) In the absence of a tail loop, the original loop is completely transformed into a vector state. At this time, for the non-array and non-deterministic induction variable instructions in the loop, if they are used externally, the instructions in the loop will become vector instructions. And the use outside the loop is still a scalar, it is obviously wrong to assign a value to a scalar with a vector. In this case, the parameters of the loop need to be adjusted, and the vectorized instruction is processed according to the vectorized width to make it a scalar.

3.2 Repair of PHI nodes outside the loop

After determining the type of loop vectorization, it is necessary to identify the instructions in the loop. There are many situations in the instructions in the loop:

1) The instructions in the loop are not used externally. In this case, the loop can be vectorized according to the original vectorization process;

2) There is external use in the loop, and the instruction that exists outside is an inductive instruction. At this time, LLVM allows the existence of this type of external use loop for vectorization.

3) The instructions in the loop are used externally, and the instructions that are used externally are non-inductive instructions in the loop. After vectorization, take the four-bit vectorization width as an example. The instruction generated by the last iteration in the original loop becomes the fourth bit of the instruction generated by the last iteration in the vectorized loop. At this time, the PHI node outside the original loop needs to be repaired, and the vector instruction is selected according to the vectorization width;

4) In the loop, there are variables that point to the loop itself and variables that point to the outside of the loop. At this time, first-order recursive processing is performed on the loop, and then the operation (3)

In the process of PHI node repair, it is necessary to add a value operation for the PHI node, because the PHI node is always at the head of the basic block, and the PHI node cannot take a value for a certain bit of the vector instruction. Therefore, it is necessary to add a new basic block outside the original basic block and the loop basic block for the vector value. In general, the instruction that needs to be fetched will have the last bit of the vector generated by the last iteration of the loop. Add a new PHI node according to the value in the added basic block. At this time, there will be two PHI nodes outside the loop, one is the newly added PHI node, and the other is the original PHI node that takes the value from the original loop. The original PHI node is invalid because of loop vectorization, so it will be optimized by the compiler during the dead code removal phase. The algorithm is shown in Algorithm 2.

Algorithm 2: PHI node value algorithm
fixLCSSAPHIs()
for(LCSSAPhi : LoopExitBlock->phis){
    if (LCSSAPhi.getNumIncomingValues == 1){
        ScalarExt = VectorLoopValueMap.getVectorValue
Builder.SetInsertPoint
if (VectorizeLoopKind != LP_mask) {
    Ext = Builder.CreateExtractElement(ScalarExt, Builder.getInt32(VF - 1), "vector.recur.extract")
} else {
    num = ExpectedCount % VF;
    Ext = Builder.CreateExtractElement(ScalarExt, Builder.getInt32(num - 1), "vector.recur.extract")
}
}
}

The loop shown in the figure cannot be vectorized in the original LLVM, and the IR generated in the form of the LCSSA loop is shown in the figure.

![Figure 4: The original LCSSA loop](image1)

After modification, it can be vectorized as shown in the figure and the PHI node can be correctly valued.

![Figure 5: LCSSA loop after vectorization](image2)

4. Experimental results and analysis

4.1 Experimental environment
The experimental environment is shown in Table 1

| Test environment | LLVM7.1.0 |
|------------------|-----------|
| Intel(R) Xeon(R) CPU E5-2682 v4 @ 2.50GHz x86_64. |

| test suite | TSVVC(Test Suit for Vectorizing Compilers) |
|------------|------------------------------------------|
| SPEC2006(Standard Performance Evaluation Corporation) |

4.2 Test results and analysis
In the robustness test, the SPEC2006 test set is used for testing. The modification during the test does not affect the correctness of the program compilation and the generated program, but the vectorization recognition rate is limited. The main reason is in the test cases in SPEC2006. the data definition-use
chain inside and outside the program loop is more complicated. The current revision is not enough to deal with some complicated situations under the premise of ensuring correctness. Further improvements are needed in the future.

| Lab  | 400 | 401 | 403 | 429 | 456 | 458 | 462 | 464 | 471 | 473 | 483 | 433 | 444 | 450 | 453 | 470 |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Res  |     |     |     |     | pass | pass | pass | pass | pass | pass | pass | pass | pass | pass | pass | pass |

TSVC test is mainly used to test the vectorization ability of the compiler, including vectorization recognition ability and vectorization correctness and acceleration effect. In the TSVC test, the original LLVM can automatically vectorize 52 test cases. After modifying the LCSSA loop automatic vectorization recognition function of LLVM, LLVM added 12 test cases automatic vectorization recognition to the TSVC test set. The vectorized recognition rate has increased by 23%.

![Figure 6 Comparison of vectorization before and after modification](image)

5. Summary
Based on the existing problems of automatic vectorization recognition in LLVM, this paper proposes an improvement method for the current lack of LLVM's ability to recognize LCSSA loop automatic vectorization. When the use chain of the non-inductive variable instruction in the loop extends outside the loop, analyze the loop vectorization state, convert the vector instruction in the loop into a scalar instruction before the PHI node takes the value, and make the PHI node outside the loop retake the value. It has been tested and verified that this method is an effective method to improve the recognition rate of vectorization. For loops that cannot be vectorized with external use chains, the method can effectively eliminate the influence of external use chains on loop vectorization. It can improve the vectorization recognition rate of LLVM and provide support for other automatic vectorization-related optimizations of LLVM.

References
[1] S Valli, V Ganapathy. A Heuristic Approach for Inter Statement Parallelism using Reordering[J]. Taylor & Francis, 2015, 42(3).
[2] Gao guojun, Ren zhilei, Zhang jingxuan, Li xiaochen, Jiang he. Research Progress of Compiler Optimization Sequence Selection[J]. Science in China: Information Science, 2019, 49(10): 1267-1282.
[3] Gao yuchen, Zhao rongcai, Han lin, Lin yanbing. Research on Automatic Parallelization Technology of Loop [J]. Journal of Information Engineering University, 2019, 20(01): 82-89.

[4] Gao wei, Han lin, Zhao rongcai, Xu jinlong, Chen chaoran. Cyclic SIMD vectorization method guided by vector parallelism [J]. Journal of Software, 2017, 28(04): 925-939.

[5] Gao wei, Zhao rongcai, Han lin, Pang jianmin, Ding rui. SIMD Overview of Automatic Vectorization Compiler Optimization [J]. Journal of Software, 2015, 26(06): 1265-1284

[6] Zhao jie, Zhao rongcai, Han lin, Xu jinchen. Research on Automatic Parallelization of MPI Carrying Anti-dependence in Loop [J]. Computer Science, 2012, 39(06): 297-300

[7] Zhu yan, Zhong lujie. Data-dependent parallel computing method based on LLVM intermediate representation [J]. Computer Application Research, 2020, 37(02): 437-442

[8] Huang yabin, Li chunjian, Feng luxia. Realization of tail loop vectorization based on LLVM [C]. China Computer Society. Proceedings of the 20th Annual Conference of Computer Engineering and Technology and the 6th Microprocessor Technology Forum. China Computer Society: Computer Engineering and Technology Professional Committee of China Computer Society, 2016: 133-139.

[9] Shalini Jain, Utpal Bora, Prateek Kumar, Vaibhav B. Sinha, Suresh Purini, Ramakrishna Upadrasta. An analysis of executable size reduction by LLVM passes [J]. CSI Transactions on ICT, 2019, 7(2).

[10] Petr Ročkai, Vladimír Štill, Ivana Černá, Jiří Barnat. DiVM: Model checking with LLVM and graph memory [J]. Elsevier Inc., 2018, 143.

[11] Vítor Bujés Ubatuba De Araújo, Álvaro Freitas Moreira, Rodrigo Machado. Týr: A Dependent Type System for Spatial Memory Safety in LLVM [J]. Elsevier B.V., 2016, 324.