LLVM Vectorization Improvement: Interdependent Loop Inversion

Paul-Andrei Aldea Jason Liang Tairong Lou Rishi Patel Qishen Zhou

https://github.com/paulxro/cse583-vectorization

Abstract

Automatically vectorizing compilers are an integral component of modern high-performance computing systems. By translating scalar operations into vector instructions, compilers can fully exploit the capabilities of SIMD (Single Instruction, Multiple Data) hardware, delivering significant improvements in computational throughput. However, effective vectorization remains challenging, particularly in the case of for loops with complex data dependencies that traditional vectorization techniques struggle to resolve. Consequently, many real-world programs exhibit unrealized performance potential, even in the presence of aggressive compiler optimizations. Our evaluation of the TSVC (Test Suite for Vectorizing Compilers) reveals that several sub-benchmarks continue to exhibit poor execution times in LLVM, with GCC offering only modest improvements. To bridge this gap, we introduce interdependent loop inversion, a general-purpose transformation that restructures loop-carried dependencies to better expose opportunities for existing vectorization passes. Our technique achieves up to a 15.1× speedup in LLVM and an 26.8× speedup in GCC, without requiring any modifications to the vectorization logic of either compiler back-end.

1 Introduction

Auto-vectorizing compilers transform scalar operations into vector operations, enabling the full exploitation of SIMD (Single Instruction, Multiple Data) hardware. This can significantly boost computational throughput, making vectorization a critical optimization target in modern compilers such as LLVM and GCC.

Auto-vectorization is typically applied to loops due to their repetitive nature and the fact that they often dominate program runtime. Through analyses such as dependency checking and loop normalization, compilers identify loops that are both safe and profitable to vectorize.

Despite advances in compiler infrastructure, auto-vectorization remains a challenging problem. Many loops contain complex or loop-carried data dependencies that prevent vectorization, leaving substantial performance potential unrealized even under aggressive optimization. This limitation is evident in the TSVC (Test Suite for Vectorizing Compilers), where LLVM performs poorly on several benchmarks.

```
for (int i = 0; ...; i++) {
  a[i] = b[i] + c[i];
  for (int j = 1; ...; j++) {
    aa[j][i] = aa[j-1][i] * a[i];
  }
}
```

Listing 1. Example loop that LLVM fails to vectorize

Nested loops such as the one shown in Listing 1 fail to be auto-vectorized by LLVM due to inter-loop dependencies. However, if the loop structure is transformed, additional opportunities for vectorization become visible. To address this, we introduce **Interdependent Loop Inversion**, a general-purpose transformation that restructures such loops to better expose vectorization opportunities. Our evaluation shows that this technique yields substantial speedups on multiple TSVC benchmarks—up to 15.1× in LLVM and 26.8× in GCC—demonstrating its ability to address persistent performance gaps in automatic vectorization.

In this paper, we make the following contributions:

- We introduce **Interdependent Loop Inversion**, a new transformation that enables vectorization of loops with complex interdependencies.
- We evaluate the transformation on the TSVC suite and report significant performance improvements in LLVM and GCC.
- We discuss the limitations of the current approach and suggest directions for future work.

2 Related Work

To evaluate the vectorization capabilities of different compilers, several widely-used benchmarks are available. Our work is partially based on the Test Suites for Vectorizing

Compilers (TSVC-2) written by University of Bristol High Performance Computing group, which is written in C and translated from the original TSVC benchmark(Callahan et al., 1988) developed in Fortran over 30 years ago, with several bug fixes and modifications.

TSVC-2 contains 151 loops and 4 tests with various configurations to thoroughly test compiler vectorization capabilities. In TSVC-2, each function contains exactly one loop nest, and each loop performs 32-bit floating-point operations. However, all array operations in TSVC-2 are fixed-sized at compile time, with constant loop trip counts.

Several evaluations based on TSVC (Maleki et al., 2011) and TSVC-2 (Sakib et al., 2024) have already been conducted. Earlier work tested TSVC on two different platforms using GCC, ICC, and XLC, showing that GCC failed to vectorize significantly more code snippets than ICC on the same platform. More recent work compared the vectorization capabilities of GCC, Clang, ICX, and ARM Compilers for Linux (ACFL), and performed detailed assembly-level analysis to manually verify the vectorization results on both x86 and ARM platforms. The results also showed that GCC can generate better vectorized code under specific conditions, such as reverse array access patterns and non-unit but constant stride memory accesses.

We also found existing comparisons (Modin, 2024) between GCC and Clang using TSVC-2 to evaluate auto-vectorization performance on RISC-V Vector Extensions (RVV). The study performed a detailed analysis of the test cases where either Clang or GCC failed to vectorize the code. Notably, it showed that on RVV, the test case `s235` could be vectorized by GCC but not by Clang.

3 Solution Overview

Loops with inter-dependent data flows often resist vectorization due to apparent loop-carried dependencies. Our key insight is that many of these dependencies are not fundamental, but instead arise from the program's structure—and can be eliminated through a systematic reordering of its data and control flows without altering program semantics. We introduce **Interdependent Loop Inversion**, a transformation designed to eliminate such dependencies and unlock vectorization opportunities. Our solution is implemented as an LLVM compiler back-end pass and enables more aggressive application of existing automatic vectorization techniques.

To make this concrete, we describe the conceptual phases of our approach, illustrating how loop restructuring exposes opportunities for vectorization. Consider the unvectorized code in Listing 1. We model the inter-iteration dependencies across the `i` and `j` dimensions using the following diagram, where each arrow denotes a data dependency:

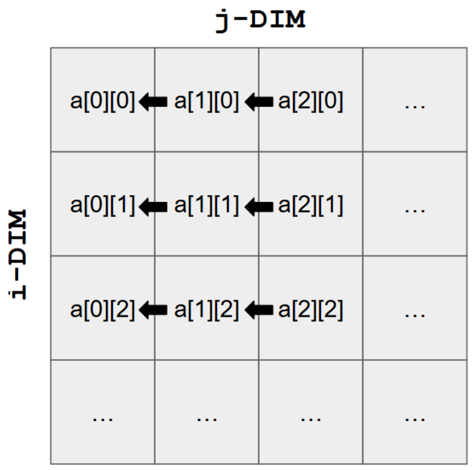

Figure 1. Sample dependency graph for Listing 1.

In this example, the inner loop (across the j dimension) is inherently sequential due to these dependencies, preventing vectorization. However, by inverting the loop nest to iterate over the i dimension in the inner loop, the computation becomes trivially vectorizable. This transformation does not alter the semantics of the program—it simply reorders the evaluation of loop iterations. In the above figure, this change would compute each column in parallel, instead of each row sequentially.

Yet, such inversion is not directly legal due to a dependency in line 2, which introduces a data flow tied to the original iteration order over i. Our transformation addresses this constraint by leveraging a key observation: the results computed in the outer loop are not consumed out of order in the inner loop. That is, later values of `a[i]` are not used by earlier iterations of the inner loop. As a result, the entire outer loop can be executed prior to the inner loop without altering program semantics. More concretely, we observe that the computations in the inner loop are sequentially disjoint from the outer-loop computations. This separation permits reordering the execution of loops while preserving correctness. To illustrate this, consider the following augmented version of the original code, in which the outer-loop body has been isolated into a separate loop:

```c
for (int i = 0; ...; i++) {
  a[i] = b[i] + c[i];
}
for (int i = 0; ...; i++) {
  for (int j = 1; ...; j++) {
    aa[j][i] = aa[j-1][i] * a[i];
  }
}
```

Listing 2. Augmented version of unvectorized code in Listing 1

With the interfering computation now hoisted out of the loop nest, we are free to invert the loops—prioritizing iteration across the i dimension in the inner loop. This reordering not only enables vectorization but also improves cache locality, as memory accesses become sequential rather than strided. When this transformed version is passed to LLVM's vectorization pipeline, the loop is fully vectorized, resulting in a substantial reduction in execution time compared to the original code.

Our transformation generalizes this approach through the following high-level steps:

- Identify unvectorized doubly nested loops within the IR of a function.
- Determine whether loop reordering could expose vectorization opportunities.
- Analyze outer-loop statements to identify code that can be safely hoisted without violating data dependencies.
- Separate hoistable code into a preceding loop and invert the loop nest.

To ensure that our transformation preserves program semantics, we perform a conservative dependency analysis before applying loop inversion. Specifically, we verify that all statements hoisted out of the inner loop exhibit no data or control dependencies on values computed within the inner loop. This guarantees that reordering does not introduce out-of-order data accesses or violate execution constraints. Additionally, we require that the outer-loop code does not induce side effects or rely on inter-iteration communication that would be affected by the changed execution order. In practice, we rely on LLVM's alias analysis, guided by Clang's strict aliasing assumptions, to determine whether memory accesses are disjoint and to verify the safety of hoisting and reordering. This analysis can be extended through programmer-provided alias metadata (e.g., `noalias` or `restrict`) to enable broader applicability in less constrained code regions.

4 Implementation

Our transformation is implemented as a custom `FunctionPass` within the LLVM framework. We leverage LLVM's existing `LoopInfo` analysis to identify candidate perfectly nested loops that may benefit from inversion. Once such candidates are located, we attempt to hoist inter-loop code into a separate pre-loop, then reorder the original nested structure to prioritize iteration over the formerly outer loop.

At present, our pass does not perform automatic legality checks to ensure semantic equivalence after transformation. Instead, we evaluate it on carefully constructed examples where the transformation is known to be correct. While developing more comprehensive static analyses to validate hoistability remains an open challenge, we hypothesize that a pragmatic approach may involve programmer annotations (e.g., via `#pragma`) to indicate safe transformation opportunities. In addition, it is also unclear whether a particular section of code is vectorizable solely by the IR representation. Consequently, our pass would need to rely on additional information from separate vectorization passes for information on whether the transformation would be beneficial at all.

We also acknowledge that our current implementation is not yet fully functional and requires further engineering to support a broader range of inputs. The primary challenges stem from limitations in existing LLVM infrastructure, particularly in safely restructuring control flow graphs (CFGs) involving tightly coupled loop nests. Our transformation introduces nontrivial rewrites to the IR, including loop peeling, hoisting across basic blocks, and reordering of induction variables, which are not directly supported by LLVM's standard loop transformation utilities. These gaps necessitate additional low-level IR manipulation, increasing both the implementation complexity and the risk of introducing subtle correctness bugs. As such, our codebase provides a partial implementation of the pass, outlining the intended transformation and our progress to date. To evaluate its effectiveness, we manually apply the loop inversion transformation on benchmark programs as a proof-of-concept, as discussed in Section 5.

5 Results

To evaluate the effectiveness of Interdependent Loop Inversion, we tested three loop-intensive benchmarks: the `s235` loop from the TSVC benchmark suite, a variant with strided memory access (`StridedAccess`), and a variant with dual memory modifications (`DoubleModify`). These were executed across three compiler configurations: unmodified Clang v18.18 (`Clang`), GCC v11.4.0 (`GCC`), and our modified Clang v18.18 (`Clang-ILI`). All tests were compiled with aggressive vectorization and floating-point optimization flags:

- `GCC`: `-O3 -fstrict-aliasing -fivopts -ftree-vectorize -ffast-math`
- `Clang`: `-O3 -fstrict-aliasing -fvectorize -fslp-vectorize-aggressive -ffast-math`

5.1 Execution Time

Table 1 shows the total execution time (in seconds) for 1000 iterations of each benchmark. `Clang-ILI` significantly

Table 1. Execution Time (s) for 1000 Iterations

Test Case	Clang	GCC	Clang-ILI
TSVC-s235	0.319	0.357	**0.021**
StridedAccess	0.074	0.071	**0.009**
DoubleModify	0.331	0.281	**0.016**

Table 2. Static Instruction Count

Test Case	Clang	GCC	Clang-ILI
TSVC-s235	761	385	**864**
StridedAccess	765	385	**889**
DoubleModify	763	386	**839**

outperformed both baseline `Clang` and `GCC` achieving a speedup of 7.93x-20.7x across these three test cases.

Importantly however, over all other test cases in the TSVC benchmark we were unable to manually apply Interdependent Loop Inversion and as a result, `Clang-ILI` performed no better than the `Clang` in the 149 other test cases.

5.2 Static Instruction Count

To assess code complexity, we measured static instruction count using compiler-generated assembly. The results, as shown in Table 2, show that `Clang-ILI` increased static instruction count by approximately 1.16x in the worst case. These results are consistent with our expectations; interdependent Loop Inversion doubles the for loops used and extra move instructions may be needed to ensure data is packed correctly for vectorized instructions.

Indeed when inspecting the generated assemblies for `Clang` and `Clang-ILI` on `TSVC-s235`, we found that 118 instructions were added to `Clang-ILI`'s assembly for the additional loop. However we also noted that 15 instructions were removed in `Clang-ILI`. These instructions were scalar/single single multiply/add/subtract instructions that were replaced by their respective vectorized instructions.

Additionally, `Clang-ILI`'s assembly contained 1.06x more move instructions (27 instructions). However 21 of these instructions came from the added loop and not from vectorization.

Table 3. Dynamic Instruction Count

Test Case	Clang	Clang-ILI
TSVC-s235	744,653	**437,810**
StridedAccess	400,713	**362,223**
DoubleModify	743,116	**439,517**

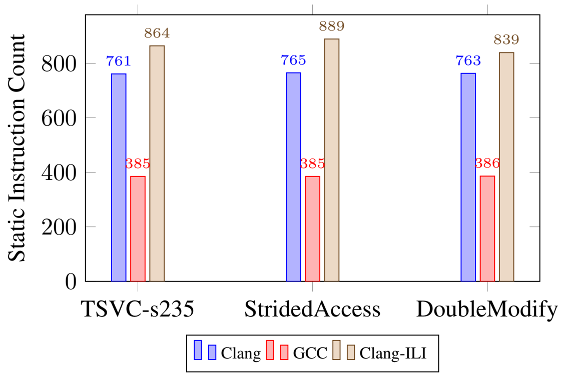

Figure 2. Static assembly instruction count

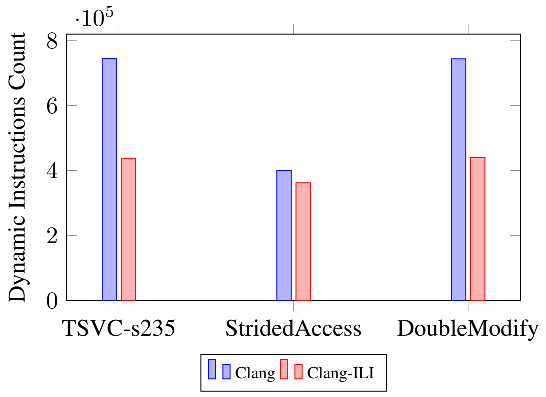

Figure 3. Dynamic instruction count (Clang only)

5.3 Dynamic Instruction Count

Table 3 summarizes dynamic instruction counts observed during 1000 iterations. These were collected for `Clang` and `Clang-ILI`.

`Clang-ILI` consistently reduced dynamic instruction counts, with a 41.2% reduction for `TSVC-s235`, indicating improved runtime efficiency through vectorized execution and reduced loop overhead.

6 Future Research

Future work will focus on completing a robust implementation of the transformation pass within LLVM, including comprehensive legality checks for correctness. Additionally, integrating analysis feedback from existing vectorization passes to make profitability decisions dynamically could yield more fine-tuned decision making. We also plan to generalize the transformation to support imperfectly nested loops and more complex control-flow structures, expanding its utility in real-world codebases. Finally, we intend to conduct a broader evaluation across less carefully crafted benchmarks to better quantify the performance impact and practical applicability of our approach.

REFERENCES

Callahan, D., Dongarra, J., and Levine, D. Vectorizing compilers: a test suite and results. In *Supercomputing '88:Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Vol. I*, pp. 98–105, 1988. doi: 10.1109/SUPERC.1988.44642.

Maleki, S., Gao, Y., Garzarán, M. J., Wong, T., and Padua, D. A. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 372–382, 2011. doi: 10.1109/PACT.2011.68.

Modin, K. A comparison of auto-vectorization performance between gcc and llvm for the risc-v vector extension, 2024.

Sakib, N., Prabhu, T., Santhi, N., Shalf, J., and Badawy, A.-H. A. Comparison of vectorization capabilities of different compilers for x86 and arm cpus. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2024. doi: 10.1109/HPEC62836.2024.10938481.