SLP Vectorization in the Presence of Control Flow

Thomas Talbot (ttalbot), ttalbot@andrew.cmu.edu

Josh Clune (jclune), jclune@andrew.cmu.edu

https://ttalbot-cmu.github.io/

1 Introduction

1.1 The Opportunity

Loop vectorization is a compiler optimization that packs scalar operations from multiple loop iterations into SIMD instructions. Super-word level parallelism (SLP) vectorizes independent and isomorphic instructions within a single basic block. SLP is a more versatile approach than loop vectorization as it provides the ability to vectorize statements outside of loop structures.

Neither Loop vectorization nor SLP is designed to directly reason about control flow. Hence, parallelizable statements in different basic blocks are not automatically vectorized. In this project, we explored how loop unrolling combined with SLP can provide a framework to vectorize statements across basic control flow within the body of a loop.

1.2 Approach

Our approach can be viewed as a sequence of four passes: loop unrolling, path finding, CFG transformation, and SLP vectorization. We created configurable passes that run LLVM's loop unroller and SLP vectorizer in isolation. Loop unrolling + SLP is sufficient for handling loops with no internal control flow. To uncover parallelism across control flow, we designed an analysis pass and a code transformation pass. The path finding pass records all possible sequences of basic blocks executed within the body of a loop. We require that all internal control predicates must be loop invariant, and we do not perform any transformations if this requirement is not met. Hence, one path will be executed on each iteration of the loop. After path analysis, we transform the CFG by duplicating every possible path through the loop body into its own basic block. The transformation

pass, named "control flow vectorization", modifies the control flow graph of the original loop to allow SLP to vectorize statements that were originally in separate basic blocks

1.3 Related Work

1.3.1 Loop Vectorization

There have been multiple efforts to extend the scope of traditional loop vectorizers. In the seminal paper on loop vectorization, if conversion was introduced to vectorize control flow [1]. More recently, whole function vectorization was designed to handle more complicated control flow constructs and vectorize across different execution traces [2].

1.3.2 SLP Vectroization

Our project drew inspiration from the work in All You Need Is Superword-Level Parallelism: Systematic Control-Flow Vectorization With SLP [3]. In [3], a framework for generalizing SLP to uncover parallelism across different basic blocks and loop nests was introduced. Instead of using a CFG, they implemented predicated SSA; an intermediate representation that simplifies code motion and tracking control dependencies.

1.4 Contributions

We designed a sequence of four passes that could be used to vectorize statements across basic blocks within the LLVM infrastructure. While we were not able to generate correct code out of our CFG transformation pass and benchmark against LLVM's loop vectorizer, the design and implementation of our project is significant.

- Designing and implementing a path finding pass that tracks the sequences of possible basic blocks executed in a loop body.
- Designing and implementing a CFG transformation pass that merges sequences of basic blocks into a single basic block to expand the reach of SLP.

- Integrating the above two passes with LLVM's loop unroller and SLP vectorizer to create a new vectroization pipeline.

2 Design Details

Our approach consists of four passes: A loop unrolling pass, an analysis pass to determine possible paths through the loop body, a CFG transformation, and an SLP vectorization pass.

2.1 Loop unrolling

The first stage of our pipeline is to unroll the loop so that code from multiple iterations can potentially be vectorized. LLVM has an existing loop unroll pass[1], and initially, we hypothesized that it would be possible to simply call this pass for the first stage of our pipeline. However, testing demonstrated that this approach was infeasible. The primary issue was that, rather than blindly unroll loops by a specified amount, LLVM's loop unroll pass estimates the relative cost of performing the unrolling as opposed to leaving the loop unaltered and only performs the unrolling if it is estimated to be beneficial. For the LLVM loop unrolling pass' usual use case, this is a good idea that prevents the optimization from making the code less efficient. But for our use case, this was problematic because the cost estimator would not take into account any specific plans of performing additional passes on the unrolled output. In particular, the cost estimator could not take into account the potential benefits of parallelizing instructions from different iterations.

In the early stages of the project, we considered modifying the cost estimator to be more capable of taking into account the potential benefits afforded by future stages in our pipeline. After exploring the cost estimator's code and the loop unroll pass' code, we found that it was easier to simply build our own pass that manually calls the internal LLVM functions that would forcibly unroll the loop, bypassing the cost estimator entirely. The benefit of this approach is that our pipeline is not subject to the analysis of a cost estimator that was not built for our use case, but the cost of this approach is that the first stage of our pipeline requires passing in the unroll count manually. Making this parameter subject to manual tuning is unproblematic for testing purposes, but would be inconvenient in a wider setting. In order to make the pipeline more widely accessible, it would

[1] `https://llvm.org/doxygen/LoopUnrollPass_8cpp_source.html`

probably be necessary to write our own cost estimator to determine how much loops should be unrolled, or at minimum, determine a reasonable default value to attempt if the parameter is not manually selected.

2.2 Path generation

The second stage of our pipeline is to perform a loop analysis pass. The purpose of this loop analysis pass is to generate three pieces of information:

1. Whether the loop is a viable target for our pipeline
2. If the loop is a viable target for our pipeline, where each "path" should start
3. If the loop is a viable target for our pipeline, the set of possible paths through the loop body

2.2.1 Loop viability

In order for a loop to be a viable target for our pipeline, we require that there is exactly one basic block whose terminator is conditional with a loop variant condition[2]. In our codebase, we call this basic block the loop decider. The intuition for this requirement is that we want all of the control-flow internal to the loop to be loop invariant, and we want the only loop variant control flow to be the portion of the control flow that determines whether the loop iterates or exits.

If a loop has no loop decider, then the loop will necessarily either exit immediately, never entering the loop body, or run in an infinite loop. This is because the absence of a loop decider guarantees that all of the loop's control flow will be resolved identically in each iteration, meaning if the loop iterates even once, it will iterate indefinitely. In either case, if the loop body is never entered or if the loop is run indefinitely, our pipeline would provide no benefit, so requiring that there is a loop decider is not restrictive.

Requiring that there is exactly one loop decider is more restrictive in that many loops written in actual code do not have only loop invariant control flow. Unfortunately, this restriction is mostly inevitable given the purpose of our pipeline. If a loop has loop variant internal control flow, then in

[2]Note: For the purposes of this discussion, we consider an instruction loop variant if it cannot be hoisted out of the loop and loop invariant otherwise. These definitions would not be entirely accurate in all contexts, but because LLVM uses SSA, their internal functions can (and do) safely conflate the notions.

order to represent that code faithfully, it would be necessary for certain paths to contain internal loop variant conditional branch instructions. Since these conditional branch instructions would be loop variant, they could not be hoisted and checked prior to the path, so it would be necessary to insert loop variant control flow inside of a path. But this loop variant control flow would make it impossible for the path to be modeled in a single basic block, defeating the purpose of our pipeline. Technically, it could be possible that some paths through the loop body only contain loop invariant control flow, while others contain loop variant control flow, and the purely loop invariant control flow might be amenable to cross-iteration optimization, but this is an edge case we intentionally excluded for the sake of simplicity.

As a final comment on loop viability, it should be noted that although the path generation pass is primarily an analysis pass, it is possible for the pass to make some minor modifications to the code when determining if the loop is viable. The reason for this is that we found it necessary to use LLVM's "makeLoopInvariant" function rather than "isLoopInvariant" when assessing whether any given basic block is the loop decider. The difference between these two functions is that although the latter seems closer in theory to what we would want, LLVM's definition of loop invariance made it possible to overlook cases where a condition is safely hoistable (and therefore invariant by a general definition) but presently inside of the loop (and therefore not invariant by LLVM's definition). In instances where we find a condition that can safely be hoisted, we hoist that condition so the corresponding conditional branch instruction can be viewed as loop invariant (and therefore not disqualify the loop as a target for our pipeline).

2.2.2 Path start

At a high level, we wanted to structure the final CFG so that all control flow tests could be done at the beginning of the loop, and the results of the control flow tests would be used to determine which "path" basic block would be visited. All conditional tests except for the loop decider's test are loop invariant, so these tests (and the instructions they depend on) can safely be hoisted to the start of the loop regardless where we choose to start each path. However, the loop decider's test is loop variant and therefore needs to be in the correct location relative to the rest of the code. To ensure this, our loop analysis pass passes the unique loop decider to the next pass and ensures that each path starts just after the loop decider's basic block (technically, in our code, every our

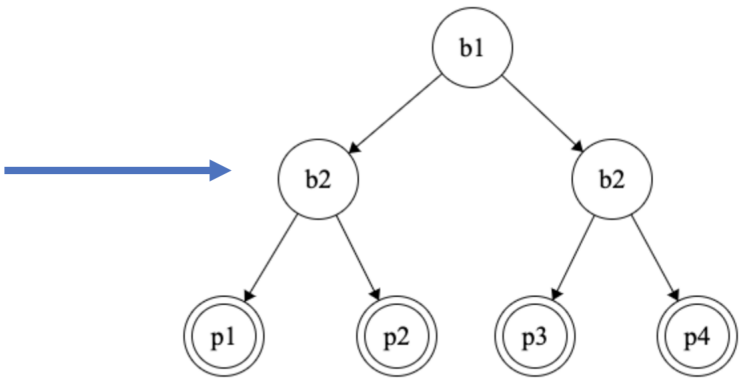
analysis pass has each path include the loop decider as the first basic block, but the "pass" basic blocks themselves do not duplicate the loop decider code at the start of the path).

2.2.3 The Path Tree

In our original version of the loop analysis pass, we only returned the set of possible paths as a vector of vectors of basic block pointers. However, returning the set of paths in such an unstructured manner made the third stage of our pipeline prohibitively complicated, so we modified the path generation pass to provide the path information in a more structured format. Specifically, the loop analysis pass generates all possible paths and puts them in a tree which represents and indicates the basic blocks that are visited along the path.

For example, if a program contains two sequential if-else statements, there are four possible paths which might be traversed (p1: in which both if-cases are followed, p2: in which the first if-case and the second else-case is followed, p3: in which the first else-case and the second if-case is followed, and p4: in which both else cases are followed). Each of these four paths would be represented as leaves in the tree (each containing a vector of basic blocks describing the whole path) and the nodes of the tree would contain the terminator instructions of the basic blocks that are visited along the path. A (very simplified) example of the path tree generated by such an example is provided below.

```
 1 if (b1) {
 2   inst1;
 3 }
 4 else {
 5   inst2;
 6 }
 7 if (b2) {
 8   inst3;
 9 }
10 else {
11   inst4;
12 }
```



By providing the paths in this format, the CFG transformation pass can easily build the control flow needed to determine which path to enter by walking down the tree from the root (corresponding to the terminator instruction of the loop decider) to each leaf (corresponding to each path).

2.3 CFG transformation

The third stage of our pipeline is to use the information provided by the loop analysis pass to transform the loop into a form amenable to SLP vectorization. Specifically, the CFG transformation ensures that if the loop is a viable target for our pipeline, then after the transformation, the following facts are true:

- All basic blocks in the loop up to the loop decider remain unaltered.
- The loop decider is unaltered except its terminator instruction is replaced by a terminator that conditionally branches to a block outside of the loop or to the first block of the new loop body which we call the first control block.
- The first control block contains all instructions that the loop's internal control flow depends on then conditionally branches to other control blocks.
- The set of control blocks created by our pass follow the tree structure defined by the path tree provided by the second stage of the pipeline.
- Leaves in the path tree provided by the second stage of the pipeline are replaced with basic blocks containing all non-terminator instructions in the path defined by the leaf.
- Each basic block that replaced a leaf has its final terminator instruction replaced with either:
  - A loop-variant conditional branch to either itself or a basic block outside of the loop. The loop variant condition used is the loop variant condition from the loop decider. This is done if and only if the final basic block in the path is the loop decider itself.
  - An unconditional branch instruction exiting the loop. This is done if and only if the final basic block in the path is outside of the loop (which can occur if the original code contains a break statement).

By pushing all of the loop's internal control flow up into the control blocks which are processed at the beginning of each loop iteration, our pass makes it possible to place instructions from different original basic blocks, and even different original loop iterations, into the same basic block, creating more potential opportunities for SLP vectorization.

2.4 SLP vectorization

The final stage of our pipeline, after modifying the CFG to make the loop more amenable to SLP vectorization, is to actually perform SLP vectorization. As with loop unrolling, LLVM already has an existing pass to perform this function, but unlike loop unrolling, there were not any complications in basically just calling it. We still wrote a custom pass to call the SLP vectorizer directly, but the final stage of our pipeline basically does just that.

3 Experimental Setup

Our experiments evaluated the effectiveness of loop unrolling + SLP versus traditional loop vectorization. We did not benchmark the entire control flow vectorization pipeline (including the path finding pass and CFG transformation). The CFG transformation pass is only effective on carefully crafted examples and is not adequate to handle the more complicated control flow and loop structures that exist in the selected benchmarks. The experiments were performed on a set of PolyBench linear algebra benchmarks that exhibit opportunities for inner and outer-loop vectorization [4].

3.1 Benchmark Programs

Benchmark	Description
3mm	Three matrix multiplications
2mm	Two matrix multiplications
mvt	Matrix vector product and transpose
durbin	Toeplitz system solver
syrk	Symmetric rank-k operations

3.2 Environment Setup

Our development environment uses an Ubuntu 22.04.01 virtual machine running on a Windows 11 host. The benchmark programs were compiled and optimized using Clang version 15.0.7 and LLVM version 10.0.0. The programs are built for the x86-64 instruction set. The underlying hardware is a $12th$ generation Intel i7-1260P CPU with a 2.10 GHz clock rate.

3.3 Benchmark Framework

We measured the cycle count of each benchmark under three different optimization strategies. First, we compile the test source code with Clang at -O1 optimization with the loop vectorizer and SLP vectorizer disabled. Further options are passed to clang to enable the Polybench utilities for measuring execution time; POLYBENCH TIME and POLYBENCH CYCLE ACCURATE TIMER. The Polybench utilities source code must be compiled and linked separately which was done using the GCC compiler. The final executable outputs the total cycle count of the benchmark and is recorded in the "O1" column of Table 1. Next, we measured the performance of the benchmarks at O1 with loop vectorization. We added a modular optimizer call to a custom pass that calls LLVM's loop vectorizer in isolation. The remaining compiling and linking process is otherwise the same. To measure the performance of unrolling + SLP we added two modular optimizer calls to passes that we wrote that run LLVM's loop unroller and SLP vectorizer in isolation. The test scripts can be found in our project repository under the benchmarks folder.

3.4 Testing Control Flow Vectorization

We did not benchmark the entire vectorization pipeline (including the path finding pass and CFG transformation). The path finding pass and CFG transformation are only effective on handcrafted examples with simplified control flow. We were unable to combine the CFG transformation after the unroll pass. Hence, the testing for control flow vectorization runs three passes from our overall pipeline: path finding, CFG transformation (control flow vectorization), and SLP vectorization. Three tests show our optimization generating SIMD instructions by combining independent scalar operations from different basic blocks. It should be noted that even on the three tests for control flow vectorization, the IR generated by the CFG transformation pass is not guaranteed to be

correct.

The tests are in the project repository under the control flow vectorization test folder. Test 3 generates vector store instructions by combining the store to A[i] on line 6 and the store to A[i+1] on line 11. Additionally, the store to B[i] and B[i+1] is also vectorized with a SIMD factor of two. The LLVM IR in test3-slp.ll shows the transformed CFG with merged control blocks and updated control predicates. Test 3 generates an executable that produces the expected result. Test 4 shows control flow vectorization for a longer path. The addition and store operations into A[i], A[i+1], A[i+2], and A[i+3] are combined into a vector add and vector store instruction with a SIMD factor of 4 (see test4-slp.ll). Similarly, the multiplication operations to A[i], A[i+1], A[i+2], and A[i+3] are also vectorized with a SIMD factor of 4. Test 4 generates an executable that produces the expected result. Test 5 shows how our vectorization pipeline would behave with a larger unroll factor e.g. 8. In this test, the SLP vectorizer chooses to issue two VADD/VMUL and two VSTORE instructions with a SIMD factor of 4 rather than a single vectorized operation with a SIMD factor of 8. Test 5 generates an executable that produces the expected result.

We did not perform extensive testing of our CFG transformation pass. It is not known to handle the vast majority of control flow and loop structures. In practice, a fully generalized CFG transformation would be difficult to implement with the existing LLVM IR. Although our path finding pass and CFG transformation pass produced interesting results on a few handcrafted examples, we do not expect that the current implementation would generalize correctly to most programs.

4 Experimental Evaluation

We measured the number of cycles for each of the three optimization strategies: O1, O1 + loop vectorization, O1 + unrolling and SLP. The results of the experiments are recorded in Table 1 and Figure 1. The mixed results confirmed our expectation that neither vectorization strategy is superior. On some benchmarks, loop unrolling + SLP outperformed the loop vectorizer by around a factor of 2x (3mm and 2mm). This speedup can be attributed to the flexible nature of the SLP vectorizer; it does not directly reason about loop structures and can vectorize statements at any level of a loop nest. LLVM's loop vectorizer exclusively vectorizes statements within the innermost loop. The matrix multiplication benchmarks (3mm and 2mm) benefited significantly from outer

Test	O1	O1 + Loop Vectorization	O1 + Unrolling and SLP
3mm	16698023662	13683892071	7026340763
2mm	8685934232	9020784340	4428082317
mvt	53579586	58507323	40096547
durbin	4674957	3745879	4566010
syrk	1396098008	1088966129	1429905500

Table 1: Cycle counts of different vectorization techniques.

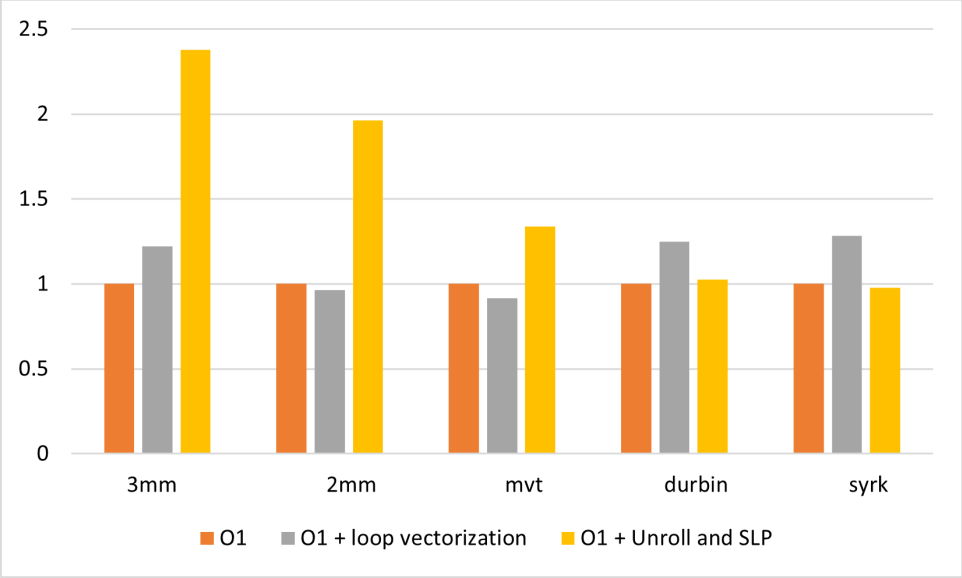


Figure 1: Speedup of execution time of different vectorization techniques, O1 is 1x.

loop vectorization. On other benchmarks, LLVM's loop vectorizer slightly outperformed unrolling + SLP by a factor of 1.25x (durbin and syrk). This is likely due to the static nature of our unroll + SLP configuration. We always unroll by a factor of 4, which imposes a constraint on the SLP vectorizer. LLVM's loop vectorizer can evaluate different unroll factors and select the appropriate amount of unrolling based on its internal cost model. Our experiments confirm that unrolling + SLP can outperform loop vectorization in certain scenarios (outer loop vectorization). However, an optimizing compiler would benefit from using both techniques since neither one outperforms the other.

5 Surprises and Lessons Learned

The biggest surprise and pitfall we encountered was the difficulty we experienced with the CFG transformation pass. Specifically, we were ultimately unable to get our pass to correctly reason about phi nodes.

There are two main issues pertaining to phi nodes. First, when the basic blocks containing all of the instructions from a path are created, phi nodes need to be removed since they no longer occur at appropriate locations. Second, at the beginning of each path's basic block, phi nodes need to be added to appropriately handle variables declared outside of the loop but potentially modified within the loop.

For the first issue, we planned to step through each path's basic block, and for each phi node, leverage the fact that for each basic block in each path, we know the previous basic block in the path. For example, if a path consisted of B1 followed by B2 and B2 had a phi node that takes value x from B1 and value y from B8, then in the path where B2 was preceded by B1, we could replace that phi node with x. In the alternate path where B2 was preceded by B8, we could instead replace the phi node with y.

For the second issue, we planned to step through each path's basic block and gather the loop variant instructions and add appropriate phi nodes to the beginning of each path's basic block. The phi nodes would have one case for if the path's basic block was preceded by a control block and another case for if the path's basic block was preceded by itself.

The primary complication that arose when attempting to implement these strategies, aside from the inherent difficulty and complexity of working with phi nodes, was the fact that we were attempting to manage multiple clones of basic blocks from the original loop. The first issue of eliminating phi nodes that would otherwise appear inside of path basic blocks is not significantly impacted by this complication since the clone function preserves phi nodes' incoming block references, but the second issue is severely complicated.

LLVM's CloneBasicBlock[3] function allows the user to pass in a value map that will store the mapping from values in the original basic block to values in the cloned basic block. Using this value map, it's possible to iterate through each path basic block's instructions and replace operands with

[3] `https://llvm.org/doxygen/Cloning_8h_source.html`

the values appropriate to the cloned block rather than the original block. But when we attempted to introduce new phi nodes to address the fact that certain variables might need to have their original reference (if they are being used after being assigned to in a portion of the code prior to the path's basic block) or their cloned reference (if they are being used after being modified in a previous iteration of the path basic block) we encountered more difficulty than we budgeted for and were unable to make it work.

After it became clear that we were going to be unable to correctly handle phi nodes in our CFG pass, we attempted to pivot to a different approach. LLVM has a pass reg2mem[4] that demotes all registers to memory references. A consequence of this demotion is that the only values live basic blocks are allocs and loads. If, rather than using phi nodes, the original loop code used allocs and loads, then we thought it might be possible to use our CFG pass basically as is. We would no longer need to introduce an additional phi node for instances where a value can be modified inside or prior to a path block. It would suffice to have both assignments be to the same memory location, which would happen automatically if instruction in the path basic block was a clone of the instruction from the original loop.

Although this strategy would probably be simpler to implement than a correct handling of phi nodes, we unfortunately did not became aware of this possibility until rather late in the project. We made an effort to implement the strategy regardless, but ran into issues when the reg2mem pass interfered with the loop analysis pass.

In order for the loop analysis pass to work, it needs to be run prior to the reg2mem pass. This is because instructions that can read from memory are not considered to be loop invariant. It would be possible to modify the code that determines whether an instruction is loop invariant to bypass the mayReadFromMemory() check, and we actually experimented with this, but this approach turned out to be too permissive. Since even the canonical induction variable would be converted to a memory location, no instruction would be considered loop variant without this check, meaning the pass would be unable to find the loop decider.

Although the loop analysis pass needs to be run prior to the reg2mem pass, it wasn't possible to enforce that ordering because the reg2mem pass would alter the source code and the pass manager would determine that the loop analysis pass' information was invalid because the source

[4] `https://llvm.org/doxygen/Reg2Mem_8cpp_source.html`

code changed. The pass manager, seeing that our CFG transformation pass depended on our loop analysis pass, would run our loop analysis pass a second time, after the reg2mem pass, yielding bad output from our loop analysis pass. To resolve this issue, we attempted to create a copy of the reg2mem pass and include some lines to tell the pass manager that the reg2mem preserved the information from our loop analysis pass. Unfortunately, the reg2mem pass is structured a bit differently from most of the passes we have written and interacted with. Specifically, it is intertwined with other scalar optimizations in such a way that attempting to copy the source code from the Reg2Mem.cpp file resulted in difficult to understand error messages.

In retrospect, we think that figuring out how to make a modified reg2mem pass that does not invalidate the results from our loop analysis pass would have been a better initial strategy than attempting to correctly handle phi nodes first. Unfortunately, because we did not realize this until late into the project, we were unable successfully implement this approach. If we were to do it over, the two main things changes we would make would be to expect to spend more time to issues pertaining to phi nodes and to start with the reg2mem strategy that we now believe would be simpler to implement.

6 Conclusions and Future Work

We explored an alternative approach for vectorizing statements across basic control flow within the body of a loop. Although we were unable to get our full pipeline to work beyond a couple of small handcrafted examples, we were able to evaluate the performance of loop unrolling followed by SLP vectorization on a set of PolyBench linear algebra benchmarks. The most obvious immediate direction for future work would be to create a modified reg2mem pass that does not invalidate results from our loop analysis pass so that we could test the performance of our full pipeline. Beyond this, it would also be possible to extend this approach to the case where a loop contains some loop variant control flow but also valid paths consisting only of loop invariant control flow so that the pipeline could be potentially applicable on a wider variety of loops. In practice, a fully functional control flow vectorizer would be difficult to implement with a CFG. A more control-dependent aware intermediate representation like predicated SSA may lead to better results.

7 Source Code

Our source code has been turned in via Canvas.

8 Distribution of total credit

We believe that the total credit should be distributed equally.

References

[1] R. Allen and K. Kennedy, "Automatic translation of fortran programs to vector form," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 4, p. 491–542, oct 1987. [Online]. Available: https://doi.org/10.1145/29873.29875

[2] R. Karrenberg and S. Hack, "Whole-function vectorization," in *International Symposium on Code Generation and Optimization (CGO 2011)*, 2011, pp. 141–150.

[3] Y. Chen, C. Mendis, and S. Amarasinghe, "All you need is superword-level parallelism: Systematic control-flow vectorization with slp," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 301–315. [Online]. Available: https://doi.org/10.1145/3519939.3523701

[4] L.-N. Pouchet, "Polybench/c the polyhedral benchmark suite," 2021. [Online]. Available: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/