Utilizing Parallel Workers: LLVM's Vectorization Plan

Jonas Fritsch
Technical University of Munich
jonas.fritsch@tum.de

Abstract

Modern SIMD processors provide various vector registers and ISAs for programmers to utilize. However, manually vectorizing code can be time-consuming, so compilers look to auto-vectorize code for better performance. LLVM, as a widely used compiler toolchain, could only auto-vectorize basic scalar code. Intel proposed a comprehensive refactoring of the underlying system to enable LLVM's auto-vectorization to support more complex optimizations. The suggested Vectorization Plan (VPlan) architecture would be more modular and scalable. Since then, significant efforts have been made to implement this new system. Today, VPlan is already actively involved in vectorizing innermost loops. Other areas, such as outer loop vectorization, are also in development. In this paper we give an introduction into VPlan and summarize its latest development results.

1 Introduction

Modern CPUs are often equipped with multiple different vector registers. These registers are nowadays as wide as 512 bits, allowing for the processing of multiple data streams at once (SIMD). By batching multiple values together in one register, different ISAs like Intel AVX or ARM SVE allow the execution of the same instruction for all values in a vector register simultaneously. Utilizing this leads to significant performance improvements over the scalar equivalent most of the time.

However, manual code vectorization can quickly become time-consuming, especially when supporting different CPU architectures. Modern compilers aim to automatically transform scalar code to use vectorization when applicable.

As one of the most widely used compilation frameworks, LLVM [5] has implemented and refined its auto-vectorization over many years. It provides two different Vectorizers, one for innermost loops (LoopVectorize) and one for super-word parallelism (SLPVectorize) [16].

This system, however, had quite a few limitations, as the loop vectorizer could only handle innermost loops and neither outer loops, complex control flow, nor non-inlined functions. Additionally, while multiple different vectorizations for the same scalar code would be possible, the current vectorizers working directly on the LLVM IR could not model and compare the costs of such different vectorization approaches.

With these limitations in mind, Intel started an ongoing refactorization effort to migrate LLVM's auto-vectorization pipeline to utilize a more abstract Vectorization Plan (VPlan) [14, 17]. The final goal would be to unite LLVM's auto-vectorization in a single flexible system capable of optimizing SLPs, inner, and nested loops with complex control flow.

The auto-vectorization pass would create multiple different VPlans, based on initial legality checks. Each VPlan represents a different vectorization approaches, by modeling the underlying scalar control flow with its own abstracted vectorized version. Different VPlan-to-VPlan transformations can then optimize these abstracted control flows based on a cost model. After that, the best VPlan, and with it the best vectorization approach, is chosen by the cost model. This VPlan is then executed, by going over its own vectorized control flow representation and materializing it back into the underlying LLVM IR.

LLVM's VPlan is currently being integrated into the existing Loop Vectorizer and is already modeling most inner loop vectorizations and transformations [4]. Vectorization for outer loops is also in development and can be enabled by setting the `-enable-vplan-native-path` feature flag [1, 3]. In the future, the plan will be to merge both of these loop vectorization paths into one. Merging the existing SLP Vectorizer with VPlan is also planned.

```
1 for (int i = 0; i < N; i++) {
2     if (z[i]) {
3         x[i] += y[i];
4     }
5 }
```

Figure 1. A code snippet of a simple loop that would be vectorized over the variable `i`. The conditional branch would be flattened into a masked addition. The mask would be generated based on `z[i]`.

The rest of this paper is organized as follows. We first explain different vectorization strategies and vectorization constraints in general and describe the original auto-vectorization approach in LLVM. Section 3 then introduces the new VPlan approach, focusing on its most important individual components and gives an outlook on future developments. In Section 4 we highlight related work to LLVM's auto-vectorization. Concluding, we summarize VPlan, its current state and potential future.

2 Background

2.1 Vectorization Strategies

In general, vectorization techniques can be separated into different categories:

Inner Loop Vectorization Innermost loops are loops whose body's control flow does not contain any more loops. Vectorization of these loops is more straightforward as the only option to vectorize is over the loop's induction variable.

There are various ways to transform the control flow of a loop to accommodate operating on widened vector registers. Figure 1 shows an example of an if-conversion. Some more examples can also be found in LLVM's Auto-Vectorization Documentation [16].

Super-Word Parallelism (SLP) Vectorization SLP Vectorization vectorizes similar independent sequential instructions. This can often be seen after a loop has been partially or fully unrolled. The resulting similar scalar instructions might then be vectorized later by an SLP vectorizer.

Outer Loop Vectorizaton Outer loop vectorization focuses on control flow with nested loops. Such control flow often presents more challenges for auto-vectorization, as one could vectorize over the outer loop, inner loop, or a mix of both, representing a hybrid/multi-dimensional vectorization approach. Finding the best legal vectorization can therefor be more difficult.

```
1 for (int i = 0; i < N; i++) {
2     float sum = 0;
3
4     for (int j = 0; j < 6; j++) {
5         sum += y[j][i] * z[j];
6     }
7
8     x[i] = sum;
9 }
```

Figure 2. A code snippet of a nested loop where it would be beneficial to vectorize over the outer loop. The inner loop has a trip count (6) that is too low to vectorize. Additionally, the memory access pattern of `y` would be consecutive compared to scattered when vectorizing over `i` instead of `j`.

Figure 2 shows a nested loop code snippet where vectorization over the outer loop would be beneficial.

Function Vectorization Function vectorization refers to the vectorization of entire functions. A loop might contain a non-inlined function call, passing data related to the iteration variable. During auto-vectorization, this call could be replaced with a call to a newly constructed function with the same semantic control flow but operating on multiple values at once through vector registers.

2.2 Vectorization Legality

Not all loops can be vectorized. There are various factors that can ultimately hinder any vectorization attempts.

Some of the most common vectorization barriers are data dependencies. As the vectorized version of a code executes for Vectorization Factor (VF) N data streams simultaneously, it inherently changes the order of operations. To guarantee that the semantics of the code do not change, all operations on these N data streams must, generally, be independent. The simplest case where this is not provided is when the i-th loop iteration depends on any of the $(i-N)$-th previous iterations.

While explicit data dependencies are easier to see, a simple loop, as in Figure 1, could carry an

implicit dependency in line 3 if the `x` and `y` point to overlapping memory regions (pointer aliasing).

Not all data dependencies directly hinder any vectorization [18]. An exception can, for example, sometimes be made for reduction idioms, as seen in the inner loop in Figure 2, where `sum` would be a reduction variable.

Uncountable loops can typically also not be vectorized, due to their unknown number of iterations.

Finally some instructions like integer division do inherently not have a vectorized instruction equivalent.

2.3 Vectorization Costs

While vectorizing a given control flow may not be legal, the vectorized code sometimes performs worse than its scalar equivalent.

Some control flow transformations that seem beneficial at compile time might reveal a performance bottleneck at runtime. Considering the code snippet from Figure 1, it might be that almost all `z[i]` evaluate to false at runtime. The branch predictor of a CPU would then be able to guess the correct branch in the scalar code. The vectorized version would always load the `y[i]` into memory only for the addition to be masked away. Different approaches to avoid such runtime penalties are branch-on-superword-condition code (BOSCC) [10, 15] or active-lane consolidation (ALC) [11, 12].

Some vector instructions can be costly on most CPUs. Examples are gather, scatter, shuffle, and permutation instructions. Control flow with many non-adjacent memory loads/stores or operations that combine different vector elements, such as reductions, can decrease vectorization performance.

Additionally, the vectorized code size will always be larger than the scalar equivalent. This is due to many factors: (1) The transformed control flow often includes more instructions. (2) As the loop trip count might not be a perfect multiple of the Vectorization Factor, a scalar epilogue/remainder of the loop might be used to handle the tail iterations. (3) Various runtime checks might be inserted throughout the vectorized control flow to handle, for example, pointers to overlapping memory regions, in which case the scalar version of the loop is executed.

2.4 Auto-Vectorization in LLVM

LLVM implements two different vectorizers for auto-vectorization, LoopVectorize (LV) and SLPVectorize [2, 16]. Auto-Vectorization is part of LLVM's middle-end and runs after module simplification has taken place. In general, these vectorizers take in scalar LLVM IR and transform it into optimized vectorized LLVM IR.

SLPVectorize is a SLP vectorizer working on straight-line code (non-loops). It works by analyzing the code bottom-up, keeping track of dependencies and grouping similar independent scalar instructions together if possible. Based on a cost model SLPVectorize then decides to vectorize these groups.

LoopVectorize was designed to focus only on vectorization of innermost loops [13]. When starting with an initial scalable loop in IR, it would first run a legality phase to check whether and how it is legal to vectorize the loop. This phase would produce different artifacts, such as if and what runtime aliasing checks would be required. Then, the cost model ran next, creating more cost-based artifacts based on the underlying scalar IR and the legality artifacts. Lastly, the transformation phase would vectorize the IR in one go based on all the previously produced artifacts.

However, this rigid flow of LoopVectorize presented a challenge. Extending LV to include outer loop and nested loop vectorizations would have been difficult, as LV was designed to flatten a loop's control flow into a single basic block. Producing and considering more and more artifacts when trying to transform the scalar loop into a vectorized one all at once also did not scale well.

As a potential solution, Intel proposed a long-term effort to refactor LLVM's whole auto-vectorization using a more modular and flexible system called Vectorization Plan (VPlan) [14].

3 LLVM's Vectorization Plan

The goal of Intels initial VPlan proposal was to unite any auto-vectorization systems (SLP, Inner-/Outer-Loop, Whole-Function) into a single flexible system [13, 14, 20]. This system would use VPlans to represent different vectorization approaches, abstracting away from the underlying scalar loop IR by using its own hierarchical control-flow-graph (H-CFG). This allows for iteratively exploring different

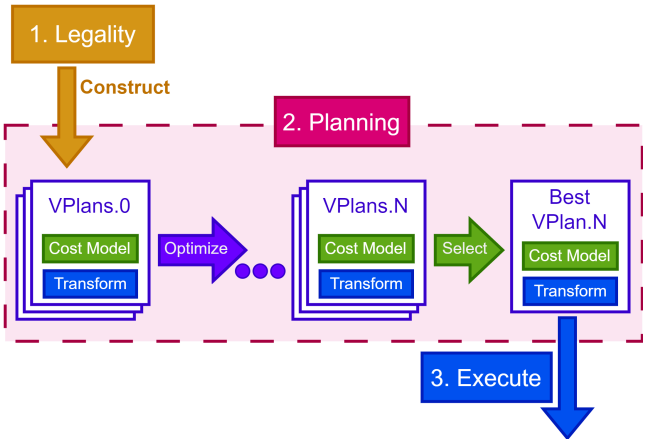

Figure 3. The planned future architecture for auto-vectorization in LLVM. The optimization step in phase 2 would iteratively apply VPlan-to-VPlan transformations.

```
for.body:
  %indvars.iv = phi i64 [ 0, %for.body.preheader ],
                        [ %indvars.iv.next, %for.inc ]
  %arrayidx = getelementptr inbounds i8, ptr %z, i64 %indvars.iv
  %0 = load i8, ptr %arrayidx, align 1
  %tobool.not = icmp eq i8 %0, 0
  br i1 %tobool.not, label %for.inc, label %if.then

if.then:
  %arrayidx2 = getelementptr inbounds float, ptr %y, i64 %indvars.iv
  %1 = load float, ptr %arrayidx2, align 4
  %arrayidx4 = getelementptr inbounds float, ptr %x, i64 %indvars.iv
  %2 = load float, ptr %arrayidx4, align 4
  %add = fadd float %1, %2
  store float %add, ptr %arrayidx4, align 4
  br label %for.inc

for.inc:
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond.not = icmp eq i64 %indvars.iv.next, %wide.trip.count
  br i1 %exitcond.not, label %for.cond.cleanup, label %for.body
```

Figure 4. The scalar IR for the main loop section of the conditional loop depicted in Figure 1. Note that `getelementptr inbounds` was abbreviated to `getelmptr inbs`.

optimization decisions by performing transformations on this CFG without modifying any LLVM IR. Each VPlan would be able to calculate its cost and execute itself, materializing the vectorization approach back into LLVM IR.

This planned systems is shown in Figure 3. At first, the Legality Phase would check if the scalar code would be legal to vectorize at all, and if so, initial VPlans would be constructed. In the following Planning Phase, these initial VPlans would be optimized using various VPlan-to-VPlan transformations [3]. The VPlan's cost model would supervise those transformations. Finally, the best VPlan would be chosen and executed, modifying the underlying IR.

3.1 VPlan Structure

Before explaining how far this system is already implemented in LLVM we will go over the different elements of a `VPlan` in more detail.

The purpose of a single VPlan object is to model a candidate for vectorization. Such a single candidate can potentially represent multiple different vectorized loops in the end. For example, one VPlan might model a general vectorization approach for multiple different Vectorization Factors (VFs) and multiple different Unroll Factors (UFs) if applicable.

H-CFG This is achieved by introducing a new layer of abstraction from the underlying loop's scalar IR. Each VPlan stores its own Hierarchical-CFG (H-CFG) modeling the vectorization candidate. This control-flow graph and its components are the main element of a single VPlan. The nodes of this H-CFG are one of three types:

- `VPBasicBlock` stores a list of `VPRecipes`, similar to the LLVM IR's `BasicBlock` holding a list of IR Instructions.
- `VPRegionBlock` represents a subgraph with a single VP entry node and a single VP exit node. A `VPRegionBlock` can be marked to replicate its subgraph `VF * UF` times when executing. This is useful for modeling scalarized or predicated instructions within the final vectorized loop body.
- `VPIRBasicBlock` is a special `VPBasicBlock` that wraps an underlying IR `BasicBlock`. These nodes are useful for modeling the scalar parts around the final vectorized loops such as the loop preheader and the epilogue loop.

Figure 4 shows the loop part of the scalar IR for the code example of Figure 1. Comparatively, Figure 5 shows the loop part of the H-CFG of a VPlan that models the vectorization approach described in Figure 1.

Recipes `VPRecipes` can be seen as instructions of an IR specialized for auto-vectorization. A single `VPRecipe` inside a VPlan is based on zero or more LLVM IR instructions of the underlying scalar IR and will materialize into one or more LLVM IR instructions when executed.

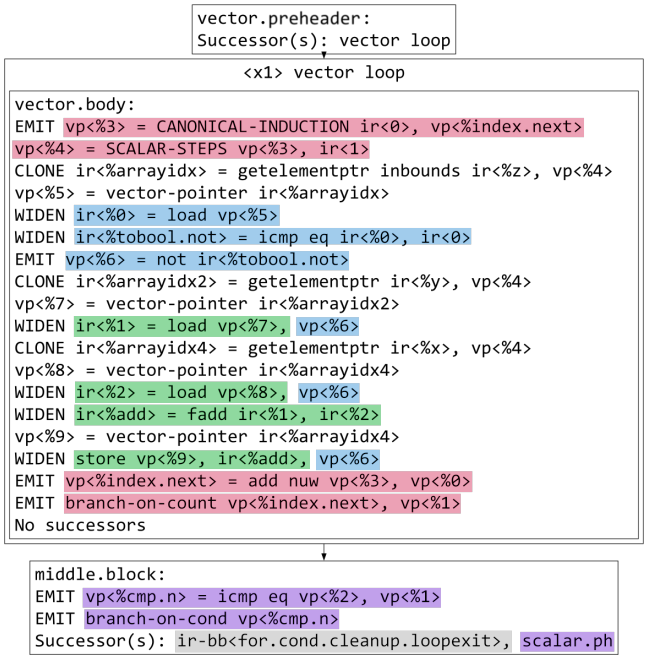

Figure 5. The vectorized loop section of the hierarchical CFG of a VPlan modeling the conditional loop depicted in Figure 1. Surrounding `VPIRBasicBlock`-nodes are omitted. Some predefined values are `vp<%0>` = VF * UF; `vp<%1>` = vector-trip-count; `vp<%2>` = original trip-count.

There are many different types of `VPRecipes`, most notably:

- `VPReplicateRecipe` will replicate one IR instruction either once (`CLONE`) or multiple times (`REPLICATE`) into the vectorized IR when executed.
- `VPWidenRecipe` will `WIDEN` one IR instruction into a vector-equivalent when executed.
- `VPInstruction` has no underlying IR instruction it is based on and is used to model newly needed instructions for the vectorized IR. When executed it will `EMIT` one or more new IR instructions.

The `VPRecipes` in Figure 5 have a `PREFIX` infront of them indicating their type. Additionally instructions/recipes of the same color between Figure 4 and Figure 5 are related. The only recipes in our example VPlan CFG that do not semantically model any instructions found in the scalar IR are from the `middle.block` (purple). This block is used to decide if the scalar epilogue loop is to be executed following the vectorized loop.

Transformations Another main benefit of the modular VPlan system is that most vector-optimization cases can be directly implemented as standalone VPlan-to-VPlan transformations. A VPlan-to-VPlan transformation is a static method that either transforms a given VPlan or creates a new VPlan based on a given VPlan. The analysis for these transformations can also be done directly on the VPlan via its `VPRecipes`.

3.2 State of VPlan (LLVM 19.1.4)

In its current state, VPlan is already deeply integrated into LoopVectorize (LV) and is used to model, optimize and execute vectorization of innermost loops [4, 13, 17]. While outer loop vectorization via VPlan has already seen quite some development, any outer loop vectorization features are still experimental and need to be explicitly enabled by passing the `-enable-vplan-native-path` feature flag [1, 3].

To better understand how VPlan is already being used in LLVM, we will now follow along the LoopVectorize pass and summarize its key components.

3.2.1 Phase 1: Legality

The main part of this phase is checking whether it is even legal to optimize a given loop. In addition to the general vectorization legality concepts mentioned in Section 2.2, LV has a number of extra conditions to check.

Some of the most important legality checks done are:

- The scalar loop's control flow must not contain indirect branches or multiple backedges. An indirect branch is a branch that jumps to a value evaluated at runtime. A backedge in a loop is an edge from the loop back to it.
- Loops with non-linear control flow must be if-convertible. This means any branch or switch statement must not exit the loop, all instructions of such a block can be executed speculatively (e.g. via masking)
- Loop-carried dependencies must be vectorizable (see Section 2.2). In LV this means explicitly that such a value must *at least* either be an induction variable, a reduction variable, or a fixed-order recurrence.

- The loop must not contain function calls outside of standard math library calls.
- Types must be vectorizable. Vectorizable types are integers, floating point types, and pointers.
- The number of runtime checks to guarantee no pointers used by recurrences overlap must not be too high.

One can find a lot more details about these legality checks and their implementations by looking at the LLVM codebase under `LoopVectorizationLegality::canVectorize`.

3.2.2 Phase 2: Planning

After the Legality Phase has analyzed the scalar loop, initial VPlans are constructed. A VPlan is initialized by first creating the H-CFG skeleton building blocks like preheaders, a region block for the vector loop, a middle block with the potential runtime check for the scalar epilogue loop, and a block for the scalar loop. After the H-CFG has been created, the scalar LLVM IR is iterated in topological order and each scalar IR instruction is transformed into a `VPRecipe`. After finishing modeling each scalar IR instruction into the VPlan, a number of VPlan transformations are executed, optimizing the initialized VPlan.

After the initial VPlans have been optimized the best Vectorization Factor is computed. This is done by comparing all generated VPlans for all their possible VFs and choosing the most profitable one. If this step computes a optimal VF of 1, it means that any form of vectorization is not profitable.

Cost Model Currently LV and VPlan still mostly rely on the old cost model. This legacy cost model derives costs based on the scalar LLVM IR instructions or, if applicable, their vectorized equivalent. This is suboptimal for VPlan as VPlans work on `VPRecipes` abstracted away from the underlying IR. These costs per instructions are then generally just lookup tables influenced by the underlying compile target.

3.2.3 Phase 3: Execution

After the best VPlan has been chosen it is executed (`LoopVectorizationPlanner::executePlan`). For this VPlan some final transformations are made that depend on a specifically chosen VF and UF. First a vectorized code skeleton is created in the IR, then the VPlan is executed. Upon execution the H-CFG of the VPlan is walked and each node is executed, which in turn executes all of its `VPRecipes` in order.

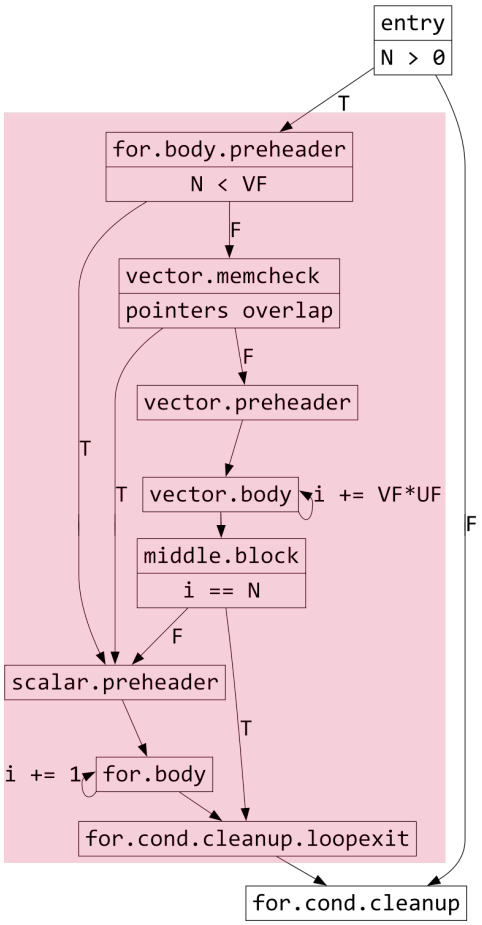

Figure 6. The generalized CFG for a vectorized loop with trip count `N`, iteration variable `i`, vectorization factor `VF`, and unroll factor `UF` in LLVM. A VPlan would explicitly model the red-highlighted region in its H-CFG.

Figure 6 shows the general CFG of a vectorized loop in LLVM and for branches the general branch condition per block.

In case that the epilogue loop could also be further vectorized, it will be in a second pass by creating VPlans for it, finding the best one and executing it.

3.3 Future of VPlan

Many of the VPlan internals and surrounding systems are still in active development. To stay up-to-date with the latest developments regarding auto-vectorization in LLVM, one can follow the Community Forum[1] or attend the monthly online meetings[2].

3.3.1 VPlan and Integration with LV

Setting up and integrating the VPlan structure into LV has seen tremendous effort over the last few years. Florian Hahn presented future directions for VPlan [4], which mostly revolve around transforming more LV-based decisions into explicit VPlan-to-VPlan transformations. Additionally, the legacy LV cost model still influences many cost model-related decisions. However, the transition to a VPlan-based cost model has already started. This new model would derive costs based on `VPRecipes` directly instead of the abstracted scalar IR instructions.

3.3.2 Outer Loop Vectorization

Development around outer loop vectorization has slowed as the focus shifted to VPlan internals and integration into LV. The inner and outer loop paths are still separated, and the experimental `-enable-vplan-native-path` flag must be set to activate any outer loop vectorization. Further development of the outer loop path and eventually merging it with the innermost loop path is planned.

3.3.3 Merging with SLP Vectorize

Merging the existing SLP Vectorizer with the current VPlan infrastructure is in the early planning phase.

3.3.4 Function Vectorization

There is currently no active development for whole-function vectorization in the llvm-project's upstream.

4 Related Work

Next to the main llvm-projects upstream, many projects are using or extending the LLVM toolchain.

One such project that tries to enable auto-vectorization support for inner/outer loops and whole-functions is the region vectorizer (RV) from the University Saarland [19]. The idea is that all the different vectorization strategies shown in Section 2.1 can be generalized into a single vectorization approach.

For example, function vectorization can be transformed into loop vectorization by creating a loop around the function's body and vectorizing it and vice versa. The same applies for SLP Vectorization and loop vectorization by unrolling a loop.

RV's goal is to have such a generalized auto-vectorization by vectorizing over *regions*. A region is defined as any control flow that has a single entry and one or more non-divergent exits. Non-divergent exits mean that starting from the same entry, control flow must not diverge to different exits. Such a region can then be for example an inner/outer loop, function, or SLP code [6, 7].

To achieve this, RV implements partial control-flow linearization [8] and a divergence analysis that can handle unstructured CFGs. There has been an effort to partially implement this control-flow linearization and divergence analysis into LLVM's upstream [7, 9].

5 Summary and Future Work

In conclusion, VPlan has been in active development for several years now. When compiling with LLVM, VPlans are used to model different vectorization approaches. They do so by adding a new layer of abstraction over the underlying LLVM IR. This VPlan layer uses `VPRecipes` to model vectorization specific instructions. This abstraction allows for efficiently exploring many different optimization decisions at once, keeping the underlying implementation flexible and modular. In the end a cost model, currently still working based on LLVM IR instead of `VPRecipes` is used to compare different VPlans and choose the most optimal one. Finally this chosen VPlan is executed, materializing its `VPRecipes` back into LLVM IR.

Outer loop vectorization support via VPlan is still experimental and separated from the inner loop LV integration. In the near future, development on outer loop vectorization will continue, and merging it with LV's VPlan path is planned.

[1] https://discourse.llvm.org/c/ir-optimizations/loop-optimizations/62

[2] https://discourse.llvm.org/t/monthly-vectorizer-online-sync-up/78978/3

Active development of other approaches, such as SLP and whole-function vectorization based on the VPlan architecture, has yet to be started.

References

[1] Diego Caballero and Intel Vectorizer Team. 2018. Extending LoopVectorize to Support Outer Loop Vectorization Using VPlan. https://llvm.org/devmtg/2018-04/slides/Caballero-Extending%20LoopVectorize%20to%20Support%20Outer%20Loop%20Vectorization%20Using%20VPlan.pdf Accessed: 2024-11-18.

[2] Renato Golin. 2014. Auto-Vectorization in LLVM. https://llvm.org/devmtg/2014-02/#talk2 Accessed: 2024-12-04.

[3] Florian Hahn, Satish Guggilla, and Diego Caballero. 2018. Outer Loop Vectorization in LLVM: Current Status and Future Plans. https://llvm.org/devmtg/2018-10/talk-abstracts.html#talk21 Accessed: 2024-11-22.

[4] Florian Hahn and Ayal Zaks. 2023. VPlan: Status Update and Future Roadmap. https://llvm.org/devmtg/2023-10/slides/techtalks/Hahn-VPlan-StatusUpdateAndRoadmap.pdf Accessed: 2024-11-19.

[5] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, USA, 75.

[6] Simon Moll. 2016. Extending LoopVectorize to Support Outer Loop Vectorization Using VPlan. https://www.youtube.com/watch?v=cyldhrzqxPA Accessed: 2024-12-03.

[7] Simon Moll and Sebastian Hack. 2017. VPlan + RV: A Proposal. https://www.llvm.org/devmtg/2017-10/slides/Moll-Vplan.pdf Accessed: 2024-11-19.

[8] Simon Moll and Sebastian Hack. 2018. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 543–556. https://doi.org/10.1145/3192366.3192413

[9] Simon Moll, Thorsten Klößner, and Sebastian Hack. 2018. RFC: A new divergence analysis for LLVM. https://llvm.org/devmtg/2018-04/slides/Moll-A%20new%20divergence%20analysis%20for%20LLVM.pdf Accessed: 2024-11-19.

[10] Ashutosh Nema and Anupama Rasale. 2023. Improving Vectorization for Loops with Control Flow. https://llvm.org/devmtg/2023-05/slides/QuickTalks-May10/05-EuroLLVM23%20-%20Improving%20Vectorization%20for%20Loops%20with%20Control%20Flow.pdf Accessed: 2024-11-20.

[11] Rouzbeh Paktinatkeleshteri, João P. L de Carvalho, Ehsan Amiri, and J. Nelson Amaral. 2023. Efficient Auto-Vectorization for Control-flow Dependent Loops through Data Permutation. In *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering (CASCON '23)*. IBM Corp., USA, 74–83.

[12] Wyatt Praharenka, David Pankratz, João P. L. De Carvalho, Ehsan Amiri, and José Nelson Amaral. 2022. Vectorizing divergent control flow with active-lane consolidation on long-vector architectures. *Journal of Supercomputing* 78, 10 (July 2022), 12553–12588. https://doi.org/10.1007/s11227-022-04359-w

[13] Gil Rapaport and Ayal Zaks. 2017. Introducing VPlan to the Loop Vectorizer. https://llvm.org/devmtg/2017-03//assets/slides/introducing_vplan_to_the_loop_vectorizer.pdf Accessed: 2024-11-16.

[14] Hideki Saito and Intel Vectorizer Team. 2016. Extending LoopVectorizer towards supporting OpenMP4.5 SIMD and outer loop auto-vectorization. https://llvm.org/devmtg/2016-11/Slides/Saito-NextLevelLLVMLoopVectorizer.pdf Accessed: 2024-11-18.

[15] Jaewook Shin. 2007. Introducing Control Flow into Vectorized Code. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*. IEEE Computer Society, USA, 280–291.

[16] LLVM Team. 2021. Auto-Vectorization in LLVM. https://llvm.org/docs/Vectorizers.html Accessed: 2024-11-18.

[17] LLVM Team. 2024. Vectorization Plan. https://llvm.org/docs/VectorizationPlan.html Accessed: 2024-11-18.

[18] Cornell University. 2023. Vector-Aware Coding - Data Dependencies. https://cvw.cac.cornell.edu/vector/coding/data-dependencies Accessed: 2024-12-04.

[19] Compiler Design Lab Saarland University. 2024. RV: A Unified Region Vectorizer for LLVM. https://github.com/cdl-saarland/rv Accessed: 2024-11-11.

[20] Ayal Zaks and Gil Rapaport. 2017. Vectorizing Loops with VPlan – Current State and Next Steps. https://llvm.org/devmtg/2017-10/slides/Zaks-Vectorizing%20Loops%20with%20VPlan.pdf Accessed: 2024-11-16.