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 introduce 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 or more, 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 [7] 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) [24].

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) [21, 25]. The final goal would be to unite LLVM's auto-vectorization in a single flexible system capable of optimizing SLPs, and 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 approach, 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 intermediate representation and materializing it back into the underlying LLVM IR.

LLVM's VPlan system is currently being integrated into the existing Loop Vectorizer and is already modeling most inner loop vectorizations and transformations [5]. Vectorization for outer loops is also in development and can be enabled by setting the `-enable-vplan-native-path` feature flag [1, 4]. 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.

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

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. First, we explain different vectorization strategies and constraints in general and describe the original auto-vectorization approach in LLVM. Section 3 then introduces the new VPlan system, focusing on its core VPlan component. Next, we summarize the current state of VPlan as of LLVM version 19.1.4. Section 5 gives an outlook on future developments regarding VPlan. In Section 6, we highlight other approaches to auto-vectorization and comparisons of LLVM's auto-vectorization to other compilers such as GCC. Concluding, we summarize the intentions of VPlan, its current state, and its potential future.

2 Background

2.1 Vectorization Strategies

In general, vectorization strategies 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 one loop's induction variables.

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 [24].

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.

```
for (int i = 0; i < N; i++) {
    float sum = 0;

    for (int j = 0; j < 6; j++) {
        sum += y[j][i] * z[j];
    }

    x[i] = sum;
}
```

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 that is too low to vectorize (6). Additionally, the memory access pattern of `y` would be consecutive compared to scattered when vectorizing over `i` instead of `j`.

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, therefore, be more difficult.

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 by utilizing 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) 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 VF 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 previous iterations up to the $(i - \text{VF})$-th.

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 [26]. 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, meaning loops where the total iteration count cannot be determined before the loop is executed, can often not be vectorized. This is because their exit condition usually depends on the loop's body itself, making it impossible to predetermine how many iterations could be processed together before having to check the exit condition again.

Finally, some instructions do generally not have a direct vectorized instruction equivalent. For example Integer division, commonly implemented using the Newton-Raphson method, is already a complex and iterative operation on modern CPUs. Applying this algorithm to multiple data elements in a single vector register would result in even more implementation complexity and is thus considered not worthwhile in most cases. Workarounds for some specialized cases do exist [16].

2.3 Vectorization Costs

While vectorizing a given control flow may not be legal, the vectorized code can sometimes also perform 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 access the memory at `y[i]` only for the addition to be masked away. Different approaches to avoid such runtime penalties are branch-on-superword-condition code (BOSCC) [17, 22] or active-lane consolidation (ALC) [18, 19].

The actual performance of a vector instruction also heavily depends on the ISA and its hardware implementation. For example, gather, scatter, shuffle, and permutation instructions are typically considered slow. Control flow with many non-adjacent memory accesses or operations that combine different vector elements from the same register (horizontal reductions) can thus decrease overall 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/tail of the loop might be used to handle the remaining loop 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 must be executed.

2.4 Auto-Vectorization in LLVM

LLVM implements two different vectorizers for auto-vectorization, LoopVectorize (LV) and SLPVectorize [3, 24]. Auto-Vectorization is part of LLVM's middle-end and runs after module simplification. These vectorizers take 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 [20]. 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 artifacts when trying to transform the scalar loop into a vectorized 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 pipeline to use a more modular and scalable system called Vectorization Plan (VPlan) [21].

3 LLVM's Vectorization Plan

Intel's initial VPlan proposal aimed to unite any auto-vectorization strategy (SLP, Inner/Outer-Loop, Whole-Function) into a single flexible system [20, 21, 28]. This system would use VPlan objects to represent different vectorization approaches, abstracting away from the underlying scalar loop IR using its own hierarchical control flow graph (H-CFG). This allows iteratively exploring different optimization decisions by performing transformations on this graph without modifying the underlying LLVM IR. Each VPlan could calculate its cost and execute itself, materializing its vectorization approach back into the LLVM IR.

This planned system 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 [4]. The VPlan's cost model would supervise those transformations. Finally, the best VPlan would be chosen and executed, modifying the underlying IR.

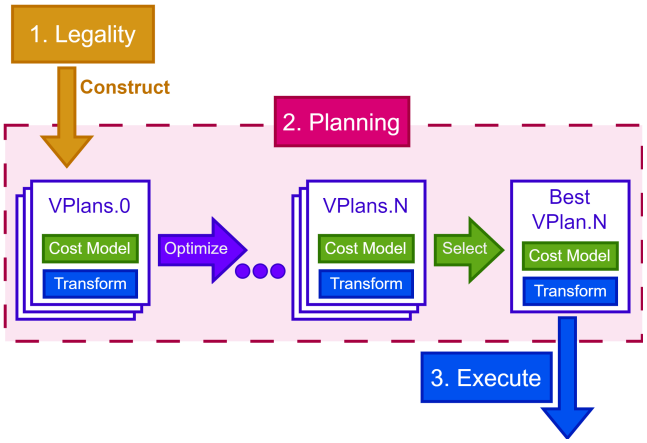

Figure 3. The planned future architecture for auto-vectorization in LLVM. The optimization step in phase 2 would iteratively apply VPlan-to-VPlan transformations.

3.1 VPlan Structure

Before explaining how far this system has been implemented in LLVM, we will review 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. For example, one VPlan might model a general vectorization approach for various Vectorization Factors (VFs) and Unroll Factors (UFs).

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 its vectorization candidate. This control-flow graph and its components are the main elements of a single VPlan. It is hierarchical in that a single node of this graph can be a complete subgraph, representing more complex nested behavior. Specifically, any node of this H-CFG is 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 entry and 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 help model 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 equivalent vector loop part of a VPlan's H-CFG that models the vectorization approach described in Figure 1. Note that the complete H-CFG of this VPlan consists of many more nodes to handle legality checks (coming before the vectorized loop) or the entire scalar epilogue loop (coming after the vectorized loop).

The H-CFG in Figure 5 consists of 3 nodes on the highest layer. `vector.preheader` and

```
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`.

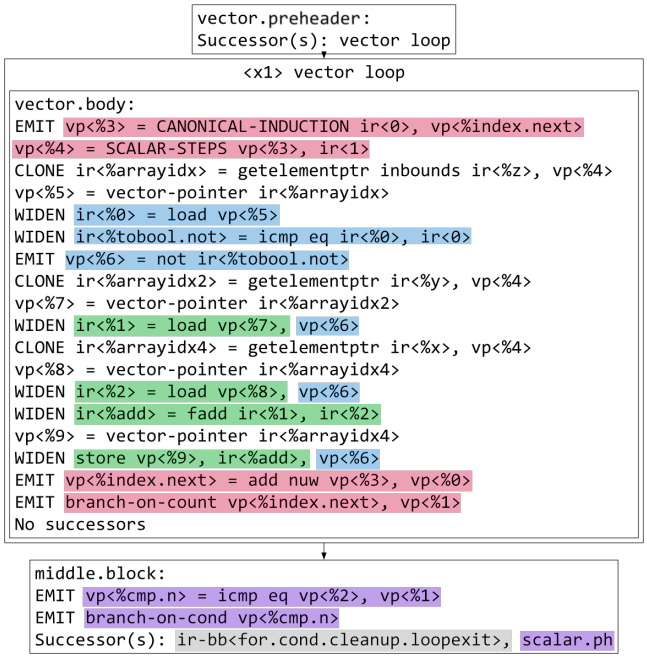

Figure 5. The vectorized loop part of a VPlan's hierarchical CFG modeling the conditional loop depicted in Figure 1. Surrounding nodes were omitted for clarity. Some predefined values are `vp<%0>` = `VF * UF`; `vp<%1>` = vector trip count; `vp<%2>` = original trip count.

`middle.block` are simple `VPBasicBlocks` surrounding the main `vector loop` node. The `vector loop` node is a `VPRegionBlock` that does not replicate its contents (indicated by `<x1>`; otherwise, it would be `<xVFxUF>`).

Inside the `vector loop` region is only a single `VPBasicBlock` node `vector.body`. This block models the entire vectorized loop via `VPRecipes`.

Recipes `VPRecipes` can be seen as instructions of an intermediate representation specialized for auto-vectorization. A single `VPRecipe` inside a VPlan is based on zero or more LLVM IR instructions of the underlying scalar IR. These underlying IR instructions are called *ingredients*. Each `VPRecipe` will materialize back into one or more LLVM IR instructions when it is executed.

There are currently 40 different final types of `VPRecipes`. The types used within Figure 5 and the most important ones are:

- `VPReplicateRecipe` will replicate one IR instruction either once (`CLONE`) or `VF * UF` 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.
- `VPHeaderPHIRecipe` is a sub-type that models any induction, such as first-order recurrence, pointer induction, or reduction as a phi node.
- `VPCanonicalIVPHIRecipe` derives from `VPHeaderPHIRecipe` and represents a `CANONICAL-INDUCTION` variable via a *scalar* value.
- `VPScalarIVStepsRecipe` models the individual offsets (`SCALAR-STEPS`) per vector-lane from a common Integer or Floating-Point induction value.

Most of the `VPRecipes` in Figure 5 have capitalized `HINTS` indicating their type. Additionally IR instructions/`VPRecipes` of the same color in Figure 4 and Figure 5 are related in their semantical effect.

The only `VPRecipes` in our example VPlan H-CFG that do not semantically model any instructions found in the scalar IR are from the `middle.block` (purple). This block decides if the scalar epilogue loop is still executed after the vectorized loop by comparing the vector trip count to the original loop trip count. This additional

vector trip count is needed as the original loop trip count might not be a perfect multiple of `VF * UF`, the number of elements processed in a single vector loop iteration.

The red recipes generally handle the induction variable `i` and the termination of the vector loop. `vp<%3>` is this loop's scalar induction variable, which goes from 0 to `vp<%1>`, the vector trip count. The per vector-lane offset's step size is 1 (`ir<1>`), meaning that each vector-lane offset from the base induction variable (`vp<%3>`) is one greater than the previous lane. Thus the memory is accessed consecutively.

The blue recipes model the condition guarding the addition. `vp<%6>` holds the conditional mask and is used for the guarded memory operations.

Finally, the green recipes represent the addition. First, the data from both `y` and `x` gets loaded, then added together, and then stored back into `x`.

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`.

Such transformations include for example:

- Simplifying `VPRecipes` by removing blend operations with only one unique input, removing redundant type casts, and simplifying logical patterns.
- Removing dead `VPRecipes` whose produced values have no users consuming them.
- Moving loop-invariant `VPRecipes` out of the vector loop region. A `VPRecipe` is considered loop-invariant if it does not have any side effects, does not read from memory, is not a phi node, and all its operands are defined outside of the vector loop region.

4 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 [5, 20, 25]. While outer loop vectorization via VPlan has already seen some development, any outer loop vectorization features are still experimental and must be explicitly enabled by passing the `-enable-vplan-native-path` feature flag [1, 4].

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

4.1 Phase 1: Legality

The main part of this first phase is checking whether it is even legal to vectorize a given loop. In addition to the general vectorization legality concepts mentioned in Section 2.2, LoopVectorize has a several 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 back-edges. An indirect branch is a branch that jumps to a value evaluated at runtime. A back-edge in a loop is an edge from the loop's body back to its start.
- 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 predicated 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.
- The number of runtime checks to guarantee no pointers used by recurrences overlap must not be too high.

Additionally, during this phase, all recurrences of the loop, such as reductions or inductions, are analyzed, and their phi nodes are stored for the later initialization of VPlans.

4.2 Phase 2: Planning

After the Legality Phase has analyzed the scalar loop and it was deemed legal to vectorize, the `LoopVectorizationCostModel` and `LoopVectorizationPlanner` are initialized. The latter one then constructs the initial VPlans for

different vectorization factors based on the properties gathered during the legal phase and the cost model initialization. These properties are, for example, recurrence variables or the maximum viable vectorization factor.

A VPlan is initialized by 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 epilogue loop itself. Then the scalar LLVM IR is iterated in topological order, and each scalar IR instruction is transformed into a `VPRecipe`. After that the original scalar loop is represented by a general vectorization approach. Lastly, several VPlan transformations are executed, transforming and optimizing the VPlan's H-CFG and its `VPRecipes`.

After all 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 an optimal VF of 1, any vectorization is not profitable compared to the scalar version.

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 direct vectorized equivalent. This is suboptimal for VPlan as VPlans work on `VPRecipes` abstracted away from the underlying IR. These costs per instruction are then generally lookup tables influenced by the chosen target architecture.

A remainder of the old `LoopVectorize` implementation before VPlan is the before-mentioned `LoopVectorizationCostModel`. It is a single object that computes costs based on the scalar loop IR without any relation to different VPlans. For example, it stores the instructions that will remain scalar after vectorization or the LLVM values for which the cost can be ignored. The latter includes, for instance, loop invariant operations that would be hoisted out of the loop or trivially dead instructions.

Additionally, each `VPRecipe` has a function to calculate its own cost. For most `VPRecipes` that reference one or more LLVM IR ingredients, these are then used to calculate the cost of the recipe. On the other hand, `VPRecipes` that do not directly model an underlying LLVM scalar IR instruction, such as `VPInstruction`, can generally not compute their own cost yet, thus returning a cost of zero for now.

For simple innermost loops with few vectorization approaches, relying on the old cost model might not have too much of a performance impact. However, once more VPlan transformations and complex outer/hybrid loop vectorization are implemented, this new cost model would be an important detail of the VPlan system to reliably choose the most optimal vectorization candidate.

4.3 Phase 3: Execution

After the best VPlan has been chosen, it is executed. For this VPlan, some final transformations are run depending on a specifically chosen VF and UF. Then a vectorized code skeleton is created in the IR, and the VPlan executed. The H-CFG of the VPlan is walked, and each node executed, which in turn executes all of its `VPRecipes` in order. Each `VPRecipe` produces its respective output IR.

A `VPTransformState` object is passed along to store information needed for generating the output IR, such as the chosen vectorization factor or generated values that were used by later `VPRecipes`.

While Figure 6 shows the general CFG of a vectorized loop in LLVM, some of its nodes can also be omitted in special cases.

For example, if all the pointers used inside the function are marked as `noalias`, the compiler is guaranteed that they will not overlap and the `vector.memcheck` block will be omitted. While the C language and most modern C++ compilers have an explicit `restrict` keyword for this, Rust automatically applies the `noalias` attribute to safe references when compiling via LLVM.

Additionally, before the initial VPlans are constructed at the beginning of Phase 2, it is also checked if the scalar epilogue loop could be folded into the vector loop itself by increasing the vector trip count over the actual loop trip count and masking all operations. This tail-loop folding approach in LLVM has not yet been enabled by default.

Otherwise, if the newly created scalar epilogue loop could be vectorized, it would be in a second loop-vectorize pass by, again, creating VPlans, finding the best one, and executing it.

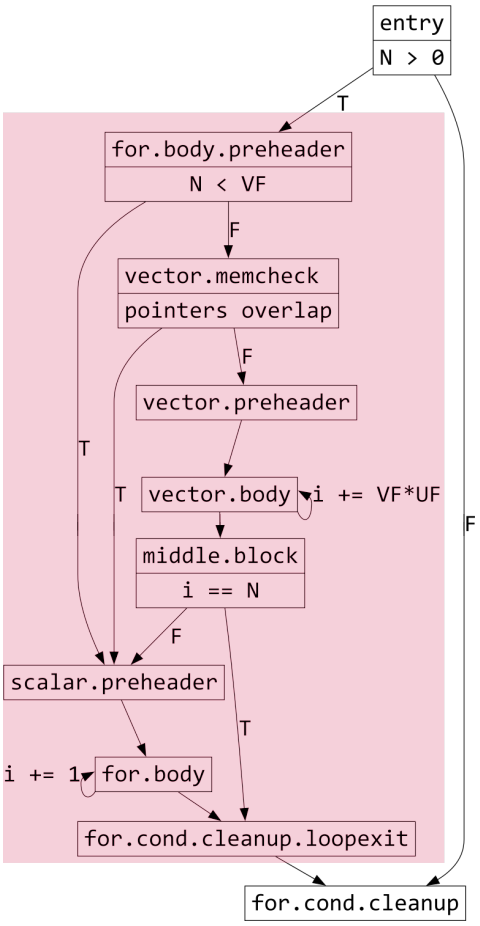

Figure 6. The generalized CFG for a vectorized loop with trip count `N`, iteration variable `i`, Vectorization Factor `VF`, and Unroll Factor `UF` in LLVM. A VPlan would so far explicitly model the red-highlighted region in its H-CFG. If a node branches, the condition is stated in the body, and the two possible path edges are marked with T (True) and F (False).

5 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].

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

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

5.1 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 [5], 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-related decisions. However, the transition to a VPlan-based cost model has been started. This new model would derive costs based on `VPRecipes` directly instead of the abstracted scalar IR instructions. Besides that, the community also focuses on enabling tail-loop folding by default, which would lead to a significant performance increase in some instances.

5.2 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.

5.3 Merging with SLP Vectorize

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

5.4 Function Vectorization

There seems to be currently no active development on whole-function vectorization in the llvm-project's upstream.

6 Related Work

6.1 Region Vectorizers

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 [27]. 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 to SLP Vectorization and loop vectorization by unrolling a loop.

RV aims to have such a generalized auto-vectorization by vectorizing over *regions*. A region is defined as any control flow with a single entry and one or more non-divergent exits. Non-divergent exits mean that control flow must not diverge to different exits starting from the same entry. Such a region can then be, for example, an inner/outer loop, function, or SLP code [10, 12].

To achieve this, RV implements partial control flow linearization [11, 13] 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 [12, 14].

In a later paper, Moll et al. further generalized this region vectorizer to be able to vectorize nested loops in a hybrid/multi-dimensional way, pushing optimization capabilities even further [15].

6.2 Comparison: GCC and LLVM

As another popular compiler, GCC has implemented and consistently improved its auto-vectorization over the years [6, 23]. Compared to LLVM, GCC also supports vectorization of outer loops and tail-loop folding by default.

In recent years, different research has been done about how compilers, especially GCC and LLVM, compare in terms of auto-vectorization.

Feng et al. compared multiple versions of GCC, LLVM, and Intel's ICC compiler in auto-vectorizing the TSVC benchmark set targeting x86-64 [2]. They found that in terms of compilation time, GCC 9 has much less overhead ($\approx 4\%$) when auto-vectorization is enabled compared to LLVM 9 ($\approx 24\%$). In both cases, LLVM 9 takes twice as long to compile than GCC 9. ICC 19 sits right between GCC 9 and LLVM 9 regarding compilation time.

While in terms of runtime, GCC 9 and LLVM 9 take around the same time with auto-vectorization disabled, enabling it improves GCC 9's performance by around 44%, while for LLVM 9 it is only 12%. ICC 19 has a better runtime than GCC 9 and LLVM 9 in both the non-vectorized and the auto-vectorized case.

The authors also investigated the number of successfully auto-vectorized code routines inside the TSVC benchmark divided into different vectorization categories. Out of a total of 151 different routines, GCC 9 vectorized 68, LLVM 9 vectorized 49, and ICC 19 did 98.

Feng et al. concluded that auto-vectorizers were mostly limited by analyzing capabilities needed to reason about specific optimization approaches and inaccurate cost models.

While LLVM 9 performs poorly in this paper, it is important to notice that at the time of LLVM 9, VPlan was relatively early into development (≈ 3 years) and was still missing support for a lot of commonly used vectorization idioms such as reductions, for example.

In a more recent auto-vectorization evaluation comparing GCC 14 with LLVM 17 on the similar TSVC 2 benchmark targeting RISC-V, the author investigated the produced vectorized code and drew comparisons to similar papers [9]. The SVE and RISC-V vector extensions are vector-length agnostic, meaning that, unlike SSE or AVX, vector instructions are not tied to specific vector register sizes. This makes the code more compatible and scalable across different underlying hardware.

In the paper, the author concludes that the achieved performance uplifts by auto-vectorization are very similar between GCC 14 and LLVM 17, with GCC only being marginally better. LLVM 17 tends to favor generating vector code targeting specific vector sizes. However, it is unclear whether this has an actual drawback.

Finally, as mentioned before, the LLVM auto-vectorization community is currently working on enabling tail-loop folding by default after noticing a $\approx 24\%$ performance difference between the latest versions of GCC and LLVM in a SPEC CPU 2017 benchmark targeting RISC-V [8].

7 Summary and Future Work

Compilers have been using auto-vectorization to help developers optimize their code across platforms by making the best possible use of a target's SIMD capabilities. Over the years, LLVM's original auto-vectorization system evolved into an inflexible pipeline, making it hard to adapt to new features like outer loop vectorization.

As a solution, Intel proposed a new modular system called VPlan. By now, VPlan has been in active development for several years. This system would

use multiple VPlan objects to efficiently model and explore different vectorization approaches simultaneously. It adds another layer of abstraction by using `VPRecipes` to model vectorization-specific instructions over the underlying LLVM IR. This allows for efficiently exploring multiple optimization decisions simultaneously, keeping the underlying implementation flexible and modular. In the end, a cost model currently 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.

While inner loop vectorization is already implemented, support for outer loops is still experimental. In the near future, development on outer loop vectorization will continue, and merging it with the LoopVectorize VPlan path is planned. Besides that, development is currently focusing on enabling tail-loop folding by default.

Active development of other approaches, such as SLP and whole-function vectorization based on the VPlan architecture, has yet to be started. Lastly, the initially planned cost model which enabled a standalone VPlan to calculate its own cost has not been implemented yet but could become a vital system in the future of VPlan.

To help evaluate the idea and current state of VPlan, it might be interesting to benchmark and compare it against different auto-vectorization systems, such as LLVM before VPlan, GCC, ICC, and region vectorizers [11, 15, 27]. Interesting metrics to compare would be compile time, performance and size of the produced code, differences in the chosen vectorization approaches, differences in legality checks that lead to loops not being vectorized, correctness, and stability. These metrics can then differ based on the target architecture.

Various suitable benchmark sets exist that have also been used for similar comparisons, such as TSVC 2 or SPEC CPU 2017.

In conclusion, the new VPlan architecture has already transformed LLVM's previously rigid auto-vectorization pipeline into a much more modular and flexible system. While some important features are still missing, the LLVM community has continuously improved VPlan over the last few years and will continue to do so.

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] Jing Ge Feng, Ye Ping He, and Qiu Ming Tao. 2021. Evaluation of compilers' capability of automatic vectorization based on source code analysis. *Scientific Programming* 2021, 1 (2021), 3264624. https://doi.org/10.1155/2021/3264624

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

[4] 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.

[5] 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.

[6] Jakub Jelínek. 2023. Vectorization optimization in GCC. https://developers.redhat.com/articles/2023/12/08/vectorization-optimization-gcc#auto_vectorization Accessed: 2025-01-22.

[7] 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. https://doi.org/10.1109/CGO.2004.1281665

[8] Luke Lau. 2025. RISC-V EVL tail folding. https://github.com/llvm/llvm-project/issues/123069 Accessed: 2025-01-21.

[9] Klara Modin. 2024. A comparison of auto-vectorization performance between GCC and LLVM for the RISC-V vector extension. (10 2024). https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-354873

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

[11] Simon Moll. 2021. Vectorization system for unstructured codes with a Data-parallel Compiler IR. https://doi.org/10.22028/D291-35500

[12] 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.

[13] 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

[14] 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.

[15] Simon Moll, Shrey Sharma, Matthias Kurtenacker, and Sebastian Hack. 2019. Multi-dimensional Vectorization in LLVM. In *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing (WPMVP'19)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/3303117.3306172

[16] Wojciech Muła. 2024. Dividing unsigned 8-bit numbers. http://0x80.pl/notesen/2024-12-21-uint8-division.html Accessed: 2024-12-29.

[17] 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.

[18] 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. https://dl.acm.org/doi/10.5555/3615924.3615932

[19] 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

[20] 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.

[21] 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.

[22] 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. https://doi.org/10.1109/PACT.2007.4336219

[23] GCC Team. 2023. Auto-vectorization in GCC. https://gcc.gnu.org/projects/tree-ssa/vectorization.html Accessed: 2024-11-04.

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

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

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

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

[28] 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.