Utilizing Parallel Workers: LLVM's Vectorization Plan

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

Abstract

Lorem ipsum

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 [3] 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) [9].

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) [7, 10]. 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 flows. The auto-vectorization pass would create and transform multiple different VPlans, each modeling a different vectorization approach, abstracted away from the underlying LLVM IR. Those VPlans are compared against each other based on a cost model, and finally, the best one, which might also not vectorize at all, is materialized into the LLVM IR.

LLVM's VPlan is currently being integrated into the existing Loop Vectorizer and is already modeling most inner loop vectorizations and transformations [2]. Vectorization for outer loops is also in development and can be enabled by setting the `-enable-vplan-native-path` flag [1]. In the future, the plan will be to merge both of these loop vectorization paths into one.

2 Background

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

2.1 Inner Loop Vectorization

Innermost loops are loops whose body's control flow does not contain any more loops, bound or unbound. 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 [9].

2.2 Super-Word Parallelism (SLP) Vectorization

SLP Vectorization vectorizes similar independent instructions. A compiler might have, for example, fully unrolled a smaller loop in a previous code

```
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]`.

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

simplification pass. The resulting similar scalar instructions might then be vectorized later by an SLP vectorizer.

2.3 Outer Loop Vectorizaton

Outer loop vectorization focuses on control flow with nested loops. Such loops often present 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. Carrying out and comparing costs of such different vectorization approaches can be complex and require a flexible system like, for example, VPlans.

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

2.4 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 instead.

2.5 Vectorization Similarities

All of the above-mentioned vectorization techniques can be brought into relation with each other. For example, function vectorization can be transformed into loop vectorization by creating a loop around the function body and vectorizing it and vice versa. The same applies for SLP Vectorization and loop vectorization when unrolling a loop.

2.6 Vectorization Legality

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

2.7 Vectorization Costs

While it may just not be legal to vectorize a given control flow, sometimes the vectorized code might also just 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 at runtime almost all of `z[i]` evaluate to false. 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. There are different approaches to avoid such runtime penalties such as branch-on-superword-condition code (BOSCC) [4, 8] or active-lane consolidation (ALC) [5, 6].

Some vector instructions can be quite costly on most CPUs. An example would be gather/scatter or shuffeling and permutation instructions. Control flow with a lot of non-adjacent memory load/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 oftentimes simply 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

3 LLVM's Vectorization Plan

Lorem ipsum

4 Related Work

Lorem ipsum

5 Summary and Future Work

Lorem ipsum

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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

[7] 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.

[8] 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.

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

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