# 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 can nowadays be as wide as 512 bit, 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 at once. Utilizing this can often lead to significant performance improvements over the scalar equivalent.

However, as manual code vectorization can quickly become very 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 [2] had 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) [4].

This system however had quite a few limitations, as the loop vectorizer could only handle innermost loops and neither outer loops, complex control flow, or non-inlined functions. Additionally, while multiple different vectorizatons for the same scalar code would be possible, the current vectorizers working directly on the LLVM IR had no capability of modelling and comparing 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 use a more abstract Vectorization Plan (VPlan) [3, 5]. 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, compare, and transform multiple different VPlans each modelling a different vectorization approach and finally materialize the best one into the LLVM IR.

In its current state LLVM's VPlan system is being integrated into the existing Loop Vectorizer and already models most of the inner loop vectorization. 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

Lorem ipsum

## 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] 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.

[3] 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.
[4] LLVM Team. 2021. Auto-Vectorization in LLVM. https://llvm.org/docs/Vectorizers.html Accessed: 2024-11-18.
[5] LLVM Team. 2024. Vectorization Plan. https://llvm.org/docs/VectorizationPlan.html Accessed: 2024-11-18.