Introduction

When a new processor is released, optimizing workloads to minimize latency, maximize throughput, reduce memory footprint, and lower power consumption is paramount. Often, a few frequently executed code blocks—kernels—dominate execution time, so optimizing these (typically via hand-tuned intrinsics or assembly) is crucial for fully exploiting the processor's ISA. Many domains rely on libraries (e.g., BLIS [1], OpenBLAS [2], Eigen [3]) that bundle such kernels, but hand-optimizing them for a new processor can take many human-months, while compiling high-level C/C++ code with an optimizing compiler often produces suboptimal performance—as evidenced by Alireza et al. (Sep 2023) [4].

Our approach instead leverages existing hand-optimized kernels from other source ISAs by auto re-vectorizing them to the new processor's ISA (target ISA) using compiler infrastructure. The intuition is that such code inherently possesses efficient, vector-friendly structures compared to generic high-level code. Prior work by Charith et al. (Feb 2019) [5] demonstrated the effectiveness of using LLVM IR passes to re-vectorize code for newer vector ISA versions (e.g., from AVX to AVX2/AVX512). We have developed a tool that uses LLVM IR transformations to convert vector intrinsic code from x86 AVX and ARM Neon to RISC-V Vector code. We also compare our method with alternative approaches—such as header-based translations [6, 7] and a non-open-source tool from a RISC-V processor maker [8]. Our contribution is the full disclosure of our LLVM pass-based approach, with the tool and source code made available to the RISC-V community.

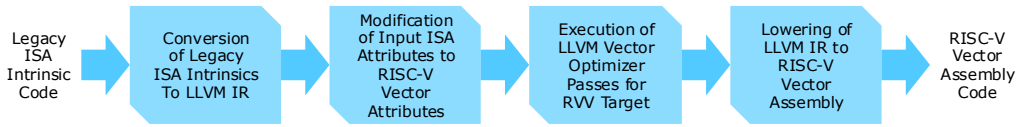
Methodologies

Auto re-vectorization Method / Algorithm

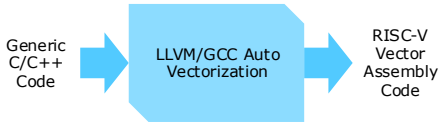
The methodology was designed to systematically transform other vector/SIMD ISA (x86 AVX, ARM SVE/Neon, etc) intrinsic code into RISC-V Vector assembly and analyze its performance metrics. The procedure consists of first converting the input intrinsic code (other vector ISA) to LLVM Vector IR, then modifying the input ISA attributes to RISC-V Vector attributes, applying LLVM vector optimization passes (for a specific RISC-V Vector Processor) and finally lowering the optimized LLVM Vector IR to the target RISC-V Vector CPU assembly code.

The details are described in the following steps using the example of converting x86 AVX code to RISC-V Vector code:

- Auto Re-Vectorization Flow



- General Compilation Flow



Conversion of AVX Intrinsic Code to LLVM Vector IR

The C/C++ AVX intrinsic code was compiled to the LLVM intermediate representation (IR) using the clang compiler. Example command:

clang -target x86_64-unknown-linux-gnu -S -emit-llvm -mfma <source_file>.c -o <llvm_ir_file>.ll

Modification of x86 Attributes to RISC-V Vector Attributes

The generated LLVM IR file is modified to replace x86-specific attributes with RISC-V Vector-specific attributes. Specifically, attributes such as “target triple” “target-cpu” “target-features” “tune-cpu” were changed to RISC-V Specific attributes, that align with Risc-V Vector architecture.

Application of LLVM Optimizer Passes

The next step is to apply LLVM Optimizer (opt) passes on the modified IR File. Before applying the optimizer passes the “optnone” Attribute was removed from the modified IR File as the optnone attribute suppresses essentially all optimizations on a function or method.

The optimizer passes applied are:

- default<O2>

- loop-vectorize
- slp-vectorizer
- load-store-vectorizer

Example command:

opt -S -debug-pass-manager -passes="default<O2>,loop-vectorize,slp-vectorizer,load-store-vectorizer" <modified_llvm_ir_file>.ll -o <optimized_llvm_ir>.ll -mtriple=riscv64 -mcpu=<target CPU for riscv64 architecture>

Generation of RISC-V Vector Assembly

The optimized LLVM IR was lowered into RISC-V Vector Assembly using the LLVM backend. Example command:

llc -march=riscv64 -mattr=<target-features> -mcpu=<target CPU for riscv64 architecture> -o <output_rvv_asm_file>.s <optimized_llvm_ir>.ll

Discussion

LLVM provides the llvm-mca tool [9] to analyze machine code using LLVM's processor scheduling models. It retrieves instruction timing, latencies, and throughput for a given CPU. We used llvm-mca to analyze generated RISC-V Vector assembly code, measuring cycle counts and performance estimates for specific target processors. Example command:

llvm-mca -mtriple=riscv64 -mcpu=<target-cpu> -iterations=1 < rvv_asm.s > <output.txt>

Results

We compared the performance of a matrix multiplication kernel (GEMM, 4x4 double precision), written in reference C code as well as in C intrinsics of x86 AVX. The reference C code was auto-vectorized to RVV using LLVM (Generic Flow), and the intrinsic code was converted to RVV using our Auto re-vectorization tool (for targets platforms A and B, described below in table 1). We compared the performance of these two RVV assemblies using the llvm-mca tool. The results are given below:

Table 1: Matmul (GEMM 4x4 double precision) performance comparison.

Target Device	Generic Flow (cycles)	Auto Re-vectorization Flow (cycles)	Performance Gain
Device A (RV64GCV + RVV 1.0)	548	315	**1.74x**
Device B (RVA22 + RVV 1.0)	91	52	**1.75x**

A. Eight-stage, dual-issue, in-order scalar pipeline and decoupled vector ALU pipeline RV64GCV 64bit RISC-V CPU with 512-bit RVV 1.0 Vector Unit
B. Thirteen-stage, four-issue, out-of-order superscalar pipeline RVA22 64-bit RISC-V CPU with dual 128-bit RVV 1.0 Vector Units

We see that the performance of the code generated by our Auto re-vectorization tool is more than 1.7x better than the auto-vectorized code generated from reference C code. We intend to do similar comparison of more kernel types, which is expected to give even better results, since matrix multiplication is well optimized in compilers compared to other compute kernels.

Comparison with Alternative Approaches

There are couple of open-source tools available for converting x86 SSE code and ARM Neon code to RVV code [6, 7]. They use an alternative approach for this conversion. They replace the intrinsic headers with the corresponding implementation using RVV or C code wherever there is no one-to-one RVV replacement for the SSE/Neon instructions. The disadvantage of this approach is that a separate implementation of the header is required for each different version of the ISAs, while our approach leverages the LLVM infrastructure itself. The second disadvantage of the header replacement approach is that it generates one-to-one replacement of the input code, which might not be the optimal RVV code for a given Vector microarchitecture. Our approach leverages LLVM infrastructure to do optimizations suitable for the given target Vector microarchitecture.

Authors
Nisanth Mathilakath Padinharepatt - Senior Staff Software Engineer

Sanket Lonkar - Software Developer

References

[1] https://github.com/flame/blis

[2] http://www.openmathlib.org/OpenBLAS/

[3] https://eigen.tuxfamily.org/

[4] (Khadem et al., 2023) Alireza Khadem, Daichi Fujiki, Nishil Talati, Scott Mahlke,Reetuparna Das, "Vector-Processing for Mobile Devices: Benchmark and Analysis", In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), Oct. 2023

[5] (Mendis et al., 2019)C. Mendis, A. Jain, P. Jain, and S. Amarasinghe, "Revec: Program rejuvenation through revectorization," in Proceedings of the 28th International Conference on Compiler Construction, ser. CC 2019. New York, NY, USA: Association for Computing Machinery, 2019, p.29–41. doi: **10.1145/3302516.33073572**

[6] https://github.com/pattonkan/sse2rvv

[7] https://github.com/howjmay/neon2rvv

[8] SiFive Recode: https://www.sifive.com/software

[9] https://llvm.org/docs/CommandGuide/llvm-mca.html

Disclaimer