Accelerate HPC and AI applications with RVV auto vectorization

Ming Yan[1], Hualin Wu[1]

[1]Terapines Technology (Wuhan) Co., Ltd

Abstract

RISC-V is an open and free ISA designed to be modular. It enables new business models. Increasingly, AI chips and other DSAs are adopting RISC-V. With the finalization of the RISC-V Vector extension, the industry is looking for innovative solutions to accelerate AI and HPC applications. Maintaining a wide range of hand-optimized intrinsic kernels for various AI chips is costly and difficult to scale. We believe that RVV auto-vectorization is the key technique to alleviate this burden on human resources. We have evaluated the auto-vectorization performance of our LLVM-based compiler ZCC on some popular kernels, and it shows that ZCC has an 18% better dynamic instruction count performance compared to hand-optimized kernels written in RVV built-in functions. To the best of our knowledge, ZCC is the only one that can successfully auto-vectorize both inner and outer loops by fully utilizing RVV features. ZCC has also achieved a 30% better performance (dynamic instruction count) compared to LLVM on SPECInt 2006.

Introduction

Vector instruction extensions are widely used to accelerate High Performance Computing (HPC) applications such as climate modeling, drug design, structural analysis, machine learning, and more. There are two types of vector processor implementations: one is the widely used SIMD engines, such as SSE and AVX, implemented in x86 processors; the other is scalable vector, such as ARM SVE and RISC-V scalable vector extensions.

Both SIMD and scalable vector process data in parallel. SIMD operations are performed on fixed-size vector registers, fixed-size vector registers can limit the scalability and efficiency of the architecture. Unlike SIMD, scalable vector supports variable vector length, enabling the architecture to scale across a wide range of hardware and workloads.

The RISC-V vector extension is designed to be a scalable, future-proof ISA. A single compiled RVV (RISC-V Vector) program can be deployed onto wide range Vector Processor Units to fully utilizing the hardware resources, regardless of how many vector processing lanes or units are designed into the processor. The actual vector length (VL) processed by each RVV instruction is determined at runtime by the hardware, based on the requested VL from the vsetvl instruction and the maximum VL supported by the hardware.

Motivation

Converting HPC applications in high-level languages such as C, C++ and Fortran into vector instructions can be challenging. Generally, there are two approaches to vectorizing application source code: 1. Users can manually write vector built-in functions within their application code. 2. A more efficient method involves allowing the compiler to automatically transform application code into vector instructions. While the first approach may initially achieve higher performance, hand-optimized code with built-in functions can be difficult to maintain. Furthermore, they may be challenging to adapt to varying hardware configurations, hinder the compiler's ability to perform additional optimizations, and ultimately limit hardware design space exploration and evolution.

Auto-vectorization is a popular research topic in the HPC field and is implemented in both open-source and commercial compilers, such as LLVM and GCC. We have studied the auto-vectorization implementations in LLVM and EPI LLVM[1]. The SIMD auto-vectorization for both innermost loops and outer loops is relatively mature, thanks to compile time known vector register width and the introduction of scalar tail loops, scalable vector auto-vectorization is bugged and very limited in both compilers.

Auto-vectorization for innermost loop and outer loops employs distinct techniques. For innermost loop, we flatten them into a single basic block by transforming control flows into masked execution. In contrast, for outer loop auto-vectorization, the control flows remain unchanged after vectorization.

Our work

ZCC is an optimized compiler based on LLVM by Terapines Technology. We have improved it to produce much better performance and higher code density than open source ones. Here are examples of some optimizations we have implemented in ZCC to improve RVV performance.

1. We have improved the cost model to calculate register grouping by estimating a best LMUL from 8 to 1/8 while avoiding register spills. The LMUL is halved and search process continues if register spills may happen. The algorithm stops upon identifying a LMUL that doesn't lead to register spills. This

optimization is implemented in LLVM IR level, allowing for an approximate calculation of register pressure.
2. The second enhancement we have introduced in ZCC involves implementing reduction operations within the inner loop vectorization. This optimization eliminates the need to duplicate scalar operations as vector operations, thereby freeing up vector registers and promoting increased parallelism.
3. We ported Intel's SIMD outer loop optimization for loops in series #1[2] to support RVV features, such as introducing VP execution in VPlan to eliminate tail loop. The importance of outer loop optimization is to optimize memory access pattern, such as turning stride memory access to unit-stride memory access.

We conducted benchmark tests of ZCC against widely-used computer vision kernels, which were hand-optimized using RVV built-ins and compiled with EPI LLVM (version 9cfcff6873), LLVM 16, and GCC 12.2. As illustrated in Figure 1, ZCC exhibits an average 18% reduction in dynamic instruction count compared to hand-written intrinsics and achieves a performance up to 100 times faster than that of EPI LLVM, LLVM, and GCC.

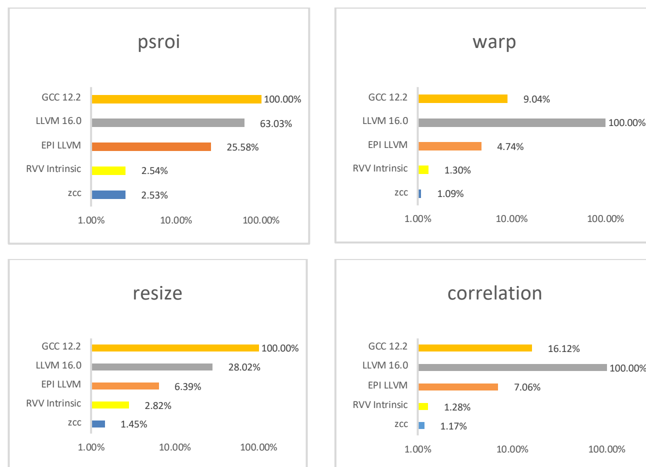

Figure 1. Relative dynamic instruction counts of 4 AI kernels running on RV32IMCV ISA simulator, vlenb=512. Lower is better. Source code is available at https://github.com/dodohack/rv_lib

Our compiler can successfully auto-vectorize outer loops in correlation, psroi and wrap with the assistance of "pragma" which indicates which level of loops should be vectorized and ensures there is no memory overlap between input and output buffers. Moreover, outer loop vectorization may enhance memory access patterns in some cases. For example, while profiling on FPGA, we observed a 50% cycle count improvement in the correlation kernel compiled by ZCC compared to the RVV intrinsic version, even though the dynamic instruction count improvement was only 8.5%. The resize kernel compiled by ZCC exhibits almost a 100% dynamic instruction count improvement over the RVV intrinsic version, as our compiler can generate operations with twice the size of register groups.

We also utilized SPECInt2006 to benchmark ZCC against LLVM and GCC. As demonstrated in Figure 2, ZCC exhibits a 30% improvement in dynamic instruction count performance compared to LLVM on RV64GCBV. ZCC still shows a 13% performance improvement compared to LLVM on RV64GCB. The auto-vectorization feature in ZCC contributes to an average performance gain of 17% in SPECInt2006.

We unable to evaluate the auto-vectorization performance of LLVM and GCC on SPECInt2006, as both compilers encountered crashes during the compilation of some test cases when auto-vectorization was enabled.

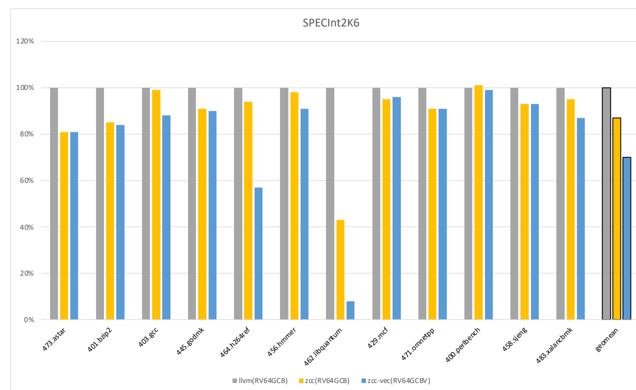

Figure 2. Dynamic instruction counts of SPECInt2006, running on RV64GCBV ISA simulator, vlenb=512. Lower is better.

Future work

The outer loop auto-vectorization implemented in ZCC is still in its early stages. Outer loop vectorization can be grouped into four series [2], but only series #1 has been implemented in ZCC thus far, optimizations for series #2, #3, #4 will be gradually implemented in ZCC.

Conclusion

ZCC has already shown great dynamic instruction count improvements over upstream/commercial compilers. With future work on target dependent optimization, ZCC can substantially reduce the cost of maintaining AI kernels and math libraries, ultimately accelerating the development of DSA and HPC hardware and software.

References

[1] https://repo.hca.bsc.es/gitlab/rferrer/llvm-epi
[2] https://lists.llvm.org/pipermail/llvm-dev/2017-December/119523.html