THEME ARTICLE: COMPILING FOR ACCELERATORS

Performance Left on the Table: An Evaluation of Compiler Autovectorization for RISC-V

Neil Adit and Adrian Sampson, *Cornell University, Ithaca, NY, 14850, USA*

Next-generation length-agnostic vector instruction set architecture (ISA) designs, the RISC-V vector extension, and ARM's scalable vector extension enable software portability across hardware implementations with different vector engines. While traditional, fixed-length single-instruction–multiple-data ISA instructions, such as Intel AVX and ARM Neon, enjoy mature compiler support for automatic vectorization, compiler support is still emerging for these length-agnostic ISAs. This work studies the compiler shortcomings that constitute the gap in autovectorization capabilities between length-agnostic and fixed-length architectures. We examine LLVM's support for both the RISC-V vector extension and traditional vector ISAs. We study a set of synthetic scalar loops to compare the breadth of support in the two settings, and we examine a real benchmark suite to compare autovectorized to hand-vectorized RISC-V code. We use both studies to distill a set of recommendations for engineering improvements and future research in compilers and programming models for length-agnostic vector programming.

Vector instructions let software conveniently exploit data parallelism within commodity central processing units (CPUs). A new wave of *length-agnostic* vector instruction set architectures (ISAs), led by ARM SVE[1] and RISC-V's vector extension,[2] address a main drawback in traditional single-instruction–multiple-data (SIMD) instruction sets: portability. Traditional ISAs use *fixed-length* vectors, tying them to a specific hardware configuration and requiring recompilation or re-engineering for each new vector engine width. Length-agnostic code remains portable across design points and generations of vector engines.

Compiler autovectorization[3–5] is a well-studied way to alleviate the effort of manual, intrinsic-based programming for a vector ISA. Most research and most compiler implementations, however, have focused on fixed-length vector ISAs. This article seeks to understand how autovectorizing compilers need to evolve to fully exploit length-agnostic ISAs. We perform two empirical evaluations to study compiler autovectorization in the context of the RISC-V vector (RVV) extension and the LLVM compiler infrastructure. First, we use a set of synthetic loops with broad coverage to identify autovectorization differences between fixed-length and length-agnostic ISAs. Next, using a set of data-parallel applications with hand-vectorized implementations, we measure the performance gap between intrinsic-based programming and compiler autovectorization configurations. To further understand this gap, we transform the applications' scalar code to model improvements in the compiler and programming model and measure their impact on closing in the gap.

THIS ARTICLE SEEKS TO UNDERSTAND HOW AUTOVECTORIZING COMPILERS NEED TO EVOLVE TO FULLY EXPLOIT LENGTH-AGNOSTIC ISAs.

We compile a list of proposals in Table 1, based on the issues we find from both the evaluations and estimate the difficulty of each proposal. We see these potential improvements as an outline for future work

0272-1732 © 2022 IEEE
Digital Object Identifier 10.1109/MM.2022.3184867
Date of publication 21 June 2022; date of current version 14 September 2022.

TABLE 1. We propose solutions for compiler autovectorization issues and rate the difficulty from a compiler's standpoint, ranging from well-defined engineering fixes (e) to compiler (c) and programming model (p) research problems.

	Proposals	Difficulty
A	Standardize IR representation (C)	***
	Runtime vector-length-based analysis (E)	*
	Multilength SLP vectorization (E)	**
	Vector reduction in dynamic loop (E)	*
B	Math library vectorization for RISC-V (E)	*
	Infer scalar width from vector code (C)	**
	Dynamic vector length scalability (C)	***
	Shuffle pattern detection (C)	***
	Algorithmic loop fusion (P)	*****
	Vectorizing specific loops (C,P)	****
	Tune algorithm to μarch (P)	*****

Notes: The proposals are grouped (A,B) based on the two evaluation benchmarks.

on rethinking autovectorization in the context of RISC-V and other scalable vector ISAs.

RELATED WORK

Maleki *et al.*[6] evaluated compiler autovectorization in GCC, Intel C Compiler, and IBM's XLC compilers for 128-bit fixed-length vector ISAs and proposed an extended version to the original test suite for vectorizing compilers (TSVC) benchmark.[7] Subsequently, additional compiler evaluations[8–10] have focused on advanced fixed-vector extensions: AVX2 and AVX-512. We focus on a more recent compiler toolchain for RVV length agnostic (VLA) and vector length specific (VLS) designs.

Prior work on evaluating next-generation vector compilers[11,12] has focused on comparing ARM SVE with ARM Neon and Intel AVX fixed-length ISAs. The prior work's focus on comparing between compilers leads them to focus only on the loops that are feasible to vectorize with current compilers. We evaluate RISC-V in this article and our goal is not only to bridge the gap between fixed- and scalable-vector designs but also to understand the remaining gap with hand-vectorized code.

EXPERIMENTAL SETUP

Our experiments use a recent source version of LLVM Clang 15.0.0.[13] We evaluate autovectorization for RVV and AVX-512 extensions. All configurations—scalar, hand-vector, and autovector—are compiled with `-Ofast` flag, which enables math library approximations in addition to `-O3` optimizations. The scalar and hand-vector configurations are compiled with `-fno-vectorize`, `-fno-slp-vectorize` to disable any compiler autovectorization. The autovector configuration for AVX-512 is also compiled with `-fveclib=libmvec` which allows LLVM to vectorize math lib calls using GLIBC vector math library.

Autovectorized versions can have three configurations:RVV-VLS, RVV vector length agnostic (RVV-VLA), and Intel AVX-512. The LLVM compilation flag for VLA is `-scalable-vectorization=on`, VLS is `-riscv-v-vector-bits-min=N`, where N determines the fixed vector width and AVX-512 is `-mavx512f -mavx512 cd`, which enables fixed vector length of 512 bits.

We extend the gem5 simulator[14] to support RISC-V vector instructions to evaluate performance. We use the Atomic CPU model to measure dynamic instruction-based statistics. We profile dynamic instruction count for AVX-512 natively on Intel Xeon Gold 6230 using `perf` and compare the autovectorized instruction speedup to RISC-V counterparts.

SYNTHETIC LOOP STUDY

We first study the breadth of LLVM's support for autovectorization for RISC-V using the TSVC benchmark. We compile all 151 loops from the TSVC benchmark and measure autovectorization differences between RVV VLS and VLA configurations. We use the instruction count speedup as a metric for compiler vectorization performance and define it for a configuration, c as

$$\text{speedup}_c = \frac{\text{Dynamic instruction count of scalar config}}{\text{Dynamic instruction count of config, } c}$$

RVV-VLS autovectorizes 13 loops in addition to 82 loops vectorized in both configurations. Among the 82 commonly vectorized loops, RVV-VLS and RVV-VLA have a geometric average of 7$\times$ and 6.3$\times$ instruction count speedup, respectively, over the scalar version for a vector length of 8 but have a few loops with differences in instruction selection. In addition, 13 loops are only autovectorized in RVV-VLS configuration because they need compile-time fixed vector length for vectorization passes. We discuss these cases in the following and propose relevant solutions, also summarized in Table 1(A).

› *Instruction selection differences:* VLS configuration can select strided loads (`vlse`), whereas VLA relies on the more general indexed loads (`vluxei`) for memory access pattern like as follows:

```
for(int i = 0; i < N; i+=2){
  a[i] = a[i - 1] + b[i];
}
```

TABLE 2. RiVec benchmark transformations to aid compiler autovectorization for an objective performance measurement.

Name	Suite	Transformations
Blackscholes	PARSEC	Skip math function
Canneal	PARSEC	Loop fusion
Jacobi-2-D	PolyBench	Restrict to nonaliasing memory; Simplify 2-D access
Pathfinder	Rodinia	Restrict to nonaliasing memory; Simplify memory access pattern
Particle filter	Rodinia	–
Streamcluster	PARSEC	–
Swaptions	PARSEC	Skip math function; inline function calls; loop interchanging

This is due to an underlying compiler representation issue for length agnostic ISAs. In general, the offsets of a gather instruction and shuffle masks cannot be represented as a numerical array since vector length is unknown for VLA, which hampers the backend instruction selection procedure.

Standardize IR representation and backend passes for gathering offsets and shuffle masks to be length agnostic.

› *Loop carried dependence analysis:* To vectorize a loop, dependence width should be greater than vector length. However, vector length is unknown for VLA at compile-time but could be speculated.[15]

Dynamically check hardware supported vector length to conditionally execute vector code.

› *SLP vectorization:* Merging a fixed number of instructions based on vector length.

Emit SLP vectorized code for cost-effective vector widths and dynamically execute one of them based on hardware vector length.

› *Product reductions:* Final reduction across vector register needs to be unrolled by the factor of vector length.

Perform vector register reduction in a loop.

› *Reverse loop traversal:* Vector memory requests need register reversal but the shuffling cost is undefined for VLA RISC-V backend.

Define the reversal cost for RISC-V backend.

APPLICATION BENCHMARK STUDY

To complement the synthetic loop study in the previous section, we also measure real benchmarks. For this study, we need a benchmark suite with existing hand-vectorized implementations for RISC-V. As far as we are aware, only one such suite exists: RiVec.[16] We extend the benchmark suite to work with the upstream LLVM repository, which now supports RVV v1.0.

We begin by comparing the performance of the hand-vectorized and autovectorized versions of the benchmarks, unmodified. This initial measurement reflects the performance gap on a current LLVM with no programmer cooperation whatsoever. To understand the makeup of this gap, we then conduct a series of experiments to measure the influence of compiler optimizations, programmer effort, or changes in the programming model. Each experiment carefully modifies the autovectorized source code in a specific way to approximate the impact of a potential change. Table 2 lists the modifications for each application. We use these experiments to quantify the potential impact of an improvement in compilation or programming for vector ISAs.

Unmodified Code

Figure 1(a) compares the dynamic instruction count speedup over scalar code on corresponding ISAs (RVV or AVX-512), of the hand-vectorized and compiler-generated configurations at a hardware vector length of 8.

Streamcluster: The compiler can effectively autovectorize the critical function: `dist` which has a streaming regular access pattern and a reduction operation. Figure 1(a) shows that the compiler autovectorized configurations have an even lower instruction overhead (better speedup!) than the hand-vector counterpart. The hand-vectorized configuration uses vector control instructions within the loop for dynamic vector length scalability (discussed in detail later), which increases the overall instruction overhead.

Blackscholes: It is embarrassingly parallel but the RISC-V autovectorized versions (RVV-VLA and RVV-VLS) have no speedup over the scalar version. The compiler is unable to vectorize math function calls rendering a scalar code for the RVV-VLA configuration. In the RVV-VLS configuration, the compiler serially unrolls math function calls to process them on the scalar machine before switching back to vector computation resulting in expensive register spilling and high instruction overhead. However, the compiler can use the GLIBC vector math library for AVX-512, which results in a 9.3$\times$ speedup over the scalar version.

Jacobi-2-D, Pathfinder: All the autovectorized configurations vectorize the applications to get comparable speedup to the hand-vectorized version. However, the autovector configurations fail to

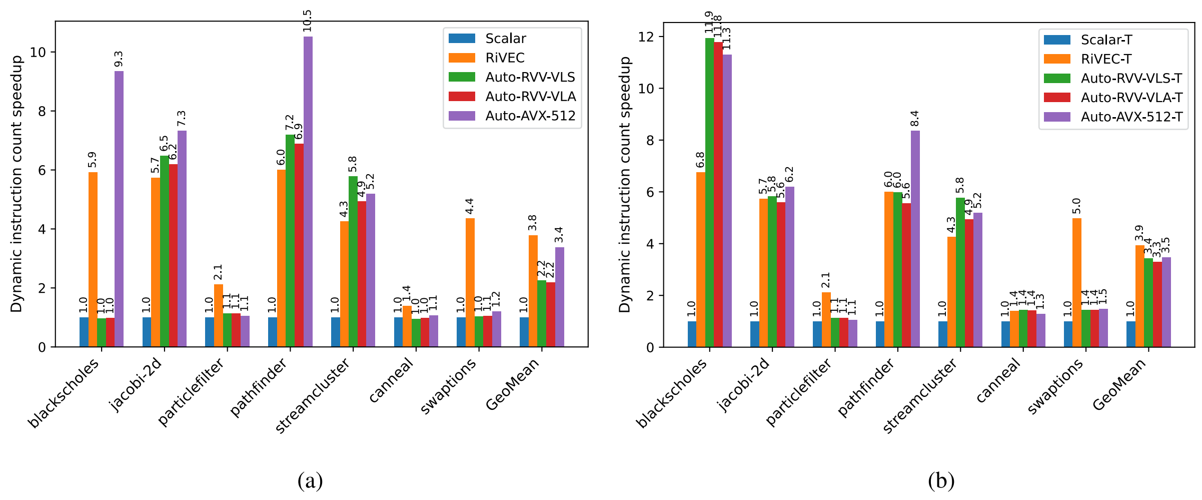

FIGURE 1. Dynamic instruction count speedup over the scalar version using a vector length of 8. Transformations for the auto-vector configurations are based on Table 2, whereas the serial and RiVec versions are transformed to skip math functions only. (a) Unmodified benchmarks. (b) Transformed benchmarks (labeled as $-T$).

identify data reuse patterns leading to redundant memory accesses.

Particlefilter, Swaptions: For these benchmarks, the compiler is unable to autovectorize critical sections resulting in minimal speedup over scalar code.

Table 1(B) summarizes the areas of improvement needed to improve the performance of the compiler autovectorized code when compared to the hand-vector version. We discuss these gaps in detail using the context of the evaluation results.

Vector Math Libraries

For some benchmarks, we find that a significant impediment to compiler autovectorization in RISC-V is the use of math function (`libm`) calls in otherwise vectorizable code. An inner loop may be easily parallelizable but contain a call to a scalar `log10`, for example, that prevents LLVM from vectorizing the entire loop. Both Blackscholes and Swaptions have such function calls in the critical sections of the code.

To measure the performance impact of this limitation, we construct special versions of the two affected benchmarks that "factor out" the influence of these math functions. In both the hand-vectorized and autovectorized versions, we replace the problematic math functions with no-ops. The resulting comparison approximates the remaining performance gap if the compiler could perfectly autovectorize code with math functions.

Figure 1(a) shows the results all transformed benchmarks: in Blackscholes, factoring out math functions closes the gap entirely, but autovectorization for Swaptions is still limited by other factors (discussed ahead). The autovectorized configurations after transforming Blackscholes, have over 11× speedup compared to the 6.8× speedup for the hand-vector counterpart. This margin is due to better fused-instruction selection and loop invariant optimization by the compiler. However, since math lib calls take a major fraction of the code execution, the hand-vectorized version might have glossed over these optimizations.

These advantages show that the compiler does a good job at instruction selection and optimizations, for simple compute patterns. Moreover, autovectorization can take advantage of the boring, fiddly optimizations and let programmers focus on the bigger picture.

LLVM should support autovectorizing code with `libm` calls by replacing them with calls to a vectorized math library for RISC-V.

Vector-Scalar Width Mismatch

The RISC-V vector extension (RVV) and AVX ISA allows a flexible element width in vector registers, in contrast to the fixed-width scalar registers defined by the base RISC-V ISA. This flexibility can cause problems when code has interactions between scalar and vector values. If an application uses 32-bit values everywhere but is compiled for RV64, then the scalar values will be promoted to 64-bit registers (using the `i64` type in LLVM). The values in vector registers, however, remain 32-bit values (e.g., using the `<8 x i32>` vector type in LLVM).

The result is that the compiler generates unnecessary instructions to convert between different element widths and might use extra vector registers to accommodate widened elements. To avoid this pitfall, we fix the primary data type for all benchmarks to use 64-bit values and compile for the RV64 base ISA. However, to make autovectorization more accessible, LLVM and other compilers should evolve to elegantly handle element size mismatches.

LLVM should infer scalar width from vectorized data types.

Dynamic Vector Length Scalability

When programming with RVV intrinsics, programmers can stripmine loops and dynamically adjust the number of elements handled per iteration:

```
//dynamic vector length
int hwl = vsetvl(N);
for (int i = 0; i < N; i += hwl){
  hwl = vsetvl(N-i);
  ...
}
```

This adjustment is especially useful in cases where the loop trip count is not a multiple of the maximum hardware vector length. However, the LLVM autovectorization only executes vector code in the maximum hardware vector-width, shown using pseudocode:

```
//maximum hardware vector length
int max_hwl = read_csr_vlen();
for (int i = 0; i < N; i += max_hwl){
    if ((N-i)<max_hwl) break; //execute left-
        over as scalar code
...
}
```

This can lead to poor scalability, usually with larger vector units. Figure 2 shows poor scalability in Jacobi-2-D for the compiler generated autovectorized versions since the loop trip count is not perfectly divisible by vector length (due to a convolution-style computation). We notice no apparent instruction reduction for both VLA and VLS autovectorized versions on going from vector length 16–32 since the scalar overhead of loop "tails" offsets the instruction savings from the increased vector length.

LLVM allows predication-based vectorizing of the loop tail using the flag: -prefer-predicate-over-epilogue=-predicate-else-scalar-epilogue but this is orthogonal to dynamic vector length control in RVV and can cause unnecessary register spilling in larger loops with conditional branches.

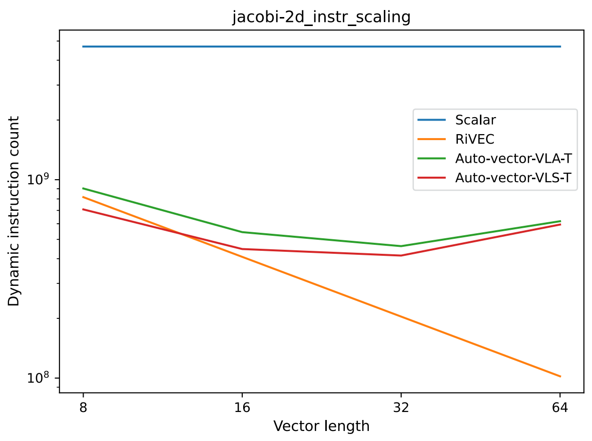

FIGURE 2. Dynamic instruction scaling across different hardware vector length in Jacobi-2-D plotted on logscale. The overhead of running scalar instructions (due to nonscalability) increases at higher vector length for compiler generated code.

LLVM should generate loops that embrace the scalable vector style: instead of assuming a fixed vector length and using scalar instructions for loop "tails," it should generate code that uses `vsetvl` to dynamically adjust the length on every iteration.

Shuffle Pattern Detection

Both Pathfinder and Jacobi-2-D have overlapping memory access patterns, which is illustrated using a simplified example as follows:

```
for (int i = 1; i < N; i++) {
  b[i] = a[i-1] + a[i] + a[i+1];
}
```

The hand-vectorized code uses RVV shuffle instructions: `vslide1up` for `a[i-1]` and `vslide1down` for `a[i+1]` to shift the values in the vector register of `a[i]`, avoiding redundant memory accesses. Such an optimization entails two components for the compiler:

- analyzing overlapping memory access patterns to remove redundant loads;
- representing shuffle patterns in the IR and selecting optimal instructions in the backend.

In general, selecting special-purpose vector shuffle instructions is hard for compilers.[17] LLVM can analyze simple recurrence patterns in the absence of aliasing but fails in more complicated cases like conditional branches (in Pathfinder) and 2-D array accesses (in Jacobi-2-D). We

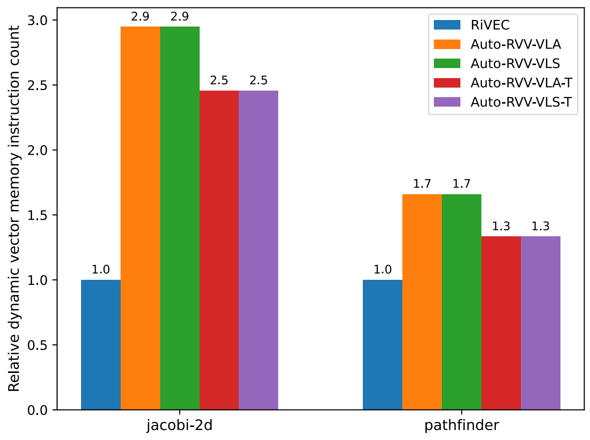

FIGURE 3. Vector memory requests relative to hand-vectorized baseline. The transformation allows compiler to reduce redundant loads in both the autovectorized configurations.

apply transformations from Table 2 to simplify these applications to the example discussed previously and assess LLVM performance. The manual transformations allows the compiler to recognize one (out of the two) first-order recurrence pattern between `a[i]` and `a[i-1]`.

LLVM uses a *dedicated* IR intrisic (`llvm.experimental.vector.splice`) for representing this pattern for VLA configuration, unlike the more standard `shufflevector` instruction used for VLS. These instructions are lowered to RISC-V shuffle instructions: `vslidedown` and `vslideup`, in the backend.

Figure 3 shows the decrease in memory requests in the transformed autovector configurations for both benchmarks. Since the compiler is partially successful in reducing redundant memory loads, the transformed autovector configurations still produces 2.5× and 1.3× higher memory requests compared to the hand-vectorized version for Jacobi-2-D and Pathfinder, respectively.

In some cases, the shuffling patterns across vector elements form the core of critical loops. Particlefilter is one such case, where the computed pattern is expressed using a sophisticated instruction `vfirst`, designed to select the first nonzero vector element. Hence, the compiler fails to vectorize the critical sections of the benchmark leading to poor performance.

LLVM needs to improve shuffle pattern analysis for generic and backend-specific patterns and use generalizable mask representation for VLA configuration.

Algorithm-Driven Loop Fusion

A lot of intrinsic-based vector programming comes down to customizing algorithms to achieve better performance for a given configuration.

The hand-vector version of Canneal uses loop fusion, among other techniques, to improve vectorization. A simplified code block from Canneal is shown as follows:

```
for (int i = 0; i < fanin; ++i){
  a = a + fanin_val[i];
}
for (int i = 0; i < fanout; ++i){
  a = a + fanout_val[i];
}
```

Since the loops are restricted by graph `fan-in` and `fan-out` degrees, even at large data simulations, the loop bounds can be smaller than the hardware vector length. In such cases, fusing the loops can provide efficient vectorization opportunities to scale to larger hardware vector lengths. However, loop fusion is not trivial and requires setting up combined arrays to facilitate it. We perform this transformation, inspired by the hand-vectorized version, for the compiler autovectorization configurations:

```
for (int i = 0; i< fanin+fanout; ++i){
  a = a + all_val[i]; // has both fanin, fanout nodes
}
```

This algorithmic transformation allows autovectorized code to run vector instructions at the maximum supported hardware vector length and close the gap with hand-vectorized version, as shown in Figure 1(b).

The future programming model for vectorization should be able to guide programmers toward transformations, such as loop fusion.

Vectorizing Specific Loops

In general, LLVM's autovectorization focuses on vectorizing the innermost loop in each loop nest. In situations where interchanging loops are not trivial, the compiler might fail to see vectorization opportunities or vectorize irrelevant loops. This effect arises at various places in Swaptions.

One such instance, after interchanging loops and simplifying 2-D accesses, looks like this

```
for (int i = 0; i < N; ++i){
         int sum = 0;
         for(int j = 0; j < M; ++j){
           sum += a[j][i];
         }
         b[i] = c[i] + sum;
       }
```

In the benchmark: $M = 3$, so just vectorizing the inner loop is not very useful. In addition, even if the inner loop were not vectorizable, the compiler would give up and not look at broader vectorization opportunities that might be visible to the programmer. The hand-vectorized version can vectorize the outer loop, which is much more scalable due to the larger loop trip count (known to the programmer) and results in simpler unit-strided vector memory accesses. This strategy allows the hand-vector version to scale well with increasing hardware vector length. The autovectorized versions fall short due to focusing on innermost loops by default.

LLVM autovectorizer should support outer-loop vectorization. However, identifying scalable loops for vectorizing requires simultaneous loop interchanging optimizations and cost analysis, which can be hard for compilers and should be offloaded to programmers.

Adapt Algorithms to the Microarchitecture

In the previous code example, the variable N is used as a blocking parameter for better caching in the serial version of the code. The hand-vector code changes the algorithm to set the variable to the hardware vector length using vector intrinsics. This unique feature of vector-length agnostic ISAs like RVV allows the algorithm to automatically adapt to different microarchitectures.

While high-level algorithmic changes are out of scope for a traditional C compiler, they represent an opportunity for higher-level languages and DSLs that compile to vector ISAs.[18]

Future work should explore programming models that make this kind of algorithmic parameterization available to programmers without requiring manual tuning of hardware intrinsics.

NEXT-GENERATION VECTOR ISAs PORTEND A NEW ERA FOR MAINSTREAM PARALLEL PROGRAMMING MODELS. THEIR POPULAR UPTAKE, HOWEVER, REQUIRES MOVING BEYOND MANUAL INTRINSIC-BASED PROGRAMMING.

CONCLUSION

Next-generation vector ISAs portend a new era for mainstream parallel programming models. Their popular uptake, however, requires moving beyond manual intrinsic-based programming. The goal should be to let programmers express high-level parallelism strategies while letting the compiler focus on what compilers do well: selecting instructions, scheduling computations, and removing redundancy. To this end, we first show areas of improvement that allow emerging ISAs to leverage the autovectorization abilities of the more mature fixed-vector counterparts. Subsequently, we delve into the newer compiler and programming model areas, which can aid autovectorization techniques to match hand-vectorized code.

ACKNOWLEDGMENTS

Our results use a gem5 simulator implementing RISC-V vector extension that was originally written by Khalid Al-Hawaj and Tuan Ta. In addition, we thank Christopher Batten, Khalid Al-Hawaj, and Tuan Ta for the valuable discussions throughout the project. We also thank the entire CAPRA research group at Cornell for their feedback and support.

This work was supported in part by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA), in part by the NSF under Awards 1845952 and 1723715, and in part by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under Grant Agreement FA8650-18-2-7863. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

REFERENCES

1. N. Stephens *et al.*, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, Mar./Apr. 2017.
2. RISC-V2021, "Working draft of the proposed RISC-V V vector extension," 2021. [Online]. Available: https://github.com/riscv/riscv-v-spec
3. S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2000, pp. 145–156.
4. D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for SIMD," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2006, pp. 132–143.

5. D. Nuzman and A. Zaks, “Outer-loop vectorization: Revisited for short simd architectures,” in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 2–11.
6. S. Maleki *et al.*, “An evaluation of vectorizing compilers,” in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2011, pp. 372–382.
7. D. Callahan, J. J. Dongarra, and D. Levine, “Vectorizing compilers: A test suite and results,” in *Proc. 1988 ACM/IEEE Conf. Supercomputing*, vol. I, 1988, pp. 98–105.
8. M. Rajan, D. W. Doerfler, M. Tupek, and S. Hammond, “An investigation of compiler vectorization on current and next-generation Intel processors using benchmarks and Sandia’s Sierra applications,” in *Cray User Group*, 2015.
9. O. V. Moldovanova and M. G. Kurnosov, “Auto-vectorization of loops on Intel 64 and Intel Xeon Phi: Analysis and evaluation,” in *Proc. Int. Conf. Parallel Comput. Technol.*, 2017, pp. 143–150.
10. S. Siso, W. Armour, and J. Thiyagalingam, “Evaluating auto-vectorizing compilers through objective withdrawal of useful information,” *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, pp. 1–23, 2019.
11. A. Pohl, M. Greese, B. Cosenza, and B. Juurlink, “A performance analysis of vector length agnostic code,” in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2019, pp. 159–164.
12. A. Poenaru and S. McIntosh-Smith, “Evaluating the effectiveness of a vector-length-agnostic instruction set,” in *Proc. Eur. Conf. Parallel Process.*, 2020, pp. 98–114.
13. LLVM, “The LLVM compiler infrastructure,” Apr. 2022. [Online]. Available: https://github.com/llvm/llvm-project/commit/e70533ae6c57756111689abf7826a3c632255866
14. N. Binkert *et al.*, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
15. M. H. Sujon, R. C. Whaley, and Q. Yi, “Vectorization past dependent branches through speculation,” in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn.*, 2013, pp. 353–362.
16. C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal, “A RISC-V simulator and benchmark suite for designing and evaluating vector architectures,” *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, pp. 1–30, 2020.
17. A. VanHattum, R. Nigam, V. T. Lee, J. Bornholt, and A. Sampson, “Vectorization for digital signal processors via equality saturation,” in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2021, pp. 874–886.
18. J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” *ACM Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

NEIL ADIT is a Ph.D. student in the Department of Electrical and Computer Engineering, Cornell University, Ithaca, NY, 14850, USA. His research interests include compilers and vector architectures. Contact him at na469@cornell.edu.

ADRIAN SAMPSON is on the faculty of the Department of Computer Science at Cornell University, Ithaca, NY, 14850, USA, where he designs and programs computers. Contact him at asampson@cs.cornell.edu.