Exploring Source-to-Source Compiler Transformation of OpenMP SIMD Constructs for Intel AVX and Arm SVE Vector Architectures

Patrick Flynn
pflynn5@uncc.edu
University of North Carolina at Charlotte
Charlotte, North Carolina, USA

Xinyao Yi
xyi2@uncc.edu
University of North Carolina at Charlotte
Charlotte, North Carolina, USA

Yonghong Yan
yyan7@uncc.edu
University of North Carolina at Charlotte
Charlotte, North Carolina, USA

Abstract

Over the past decade, SIMD (single instruction multiple data) or vector architectures have made significant advances, now existing across a wide range of devices from commodity CPUs to high performance computing (HPC) cores. Intel's AVX (Advanced Vector Extensions) architecture has been one of the most popular SIMD extensions to commodity and HPC CPUs from Intel. Over the past few years, Arm has made significant inroads with its new SVE (Scalable Vector Extension), used in the supercomputer of the top place in the Top500 list. As SIMD has become more advanced and more important, it has become equally important the compilers support these architecture extensions. In this paper, we present our approach of source-to-source compiler transformation of explicit vectorization constructs using the OpenMP SIMD directive. We present the design of a unified IR that is easily translated to AVX and SVE vector architectures. Finally, we conduct performance evaluations on Intel AVX and Arm SVE to demonstrate how this method of vectorization can bridge the gap between auto- and manual- vectorization.

Keywords: SIMD, vectorization, SVE, AVX2, AVX-512, compiler transformation, OpenMP

1 Introduction

SIMD (single-instruction multiple-data) or vector architecture has been an effective parallel processing solution to address the plateauing of Moore's law and the need for high performance of scientific data processing. Compared with multi-/many-core architecture and multi-threading parallelism, which can be considered as multiple-instruction multiple-data architecture (MIMD), SIMD uses long registers with special instructions that can perform several instances of a certain operation at once in the same amount of CPU time as a single operation. Often being referred to as vector processing, such an approach is most advantageous for parallelizing loops, which often consumes the most time in many computation-intensive applications.

Vector architectures have advanced and changed significantly in the last decade, particularly from dedicated vector architecture machines to general-purpose CPUs with vector extensions. In 2011, Intel shipped general-purpose CPUs with the second version of the Advanced Vector Extensions (AVX2), followed by the AVX-512 extensions in 2013 [2]. The AVX2 extensions contain 256-bit lane widths, and are very common in consumer CPUs. The AVX-512 extensions bring 512-bit lane widths into a few high-end consumer CPUs and many high-performance CPU product lines. Beyond increasing lane width to 512 bits, AVX-512 brings many advanced features, most notably masking, an ideal solution for strided loads and stores. The Arm architecture is another notable example of extending general-purpose CPUs to support SIMD. In 2016, Arm announced its Scalable Vector Extension (SVE) [1] vector architecture for its high-end Arm CPUs to extend the aging NEON SIMD extensions. The recent Fukagu supercomputer (ranked #1 in the November 2021 Top500 list and uses Fujitsu's Arm A64FX CPU with 512-bit vector length) represents the adoption of the Arm SVE architecture in production. Arm SVE takes a unique approach of supporting SIMD architecture of different vector lane widths. Rather than creating a new set of extensions for higher lane widths as Intel does, SVE provides a vector length agnostic model, which allows the same code to run on different hardware with varying vector lengths. In other words, vendors can create vectors as long or short as needed, and the same machine code will run on any hardware.

While it is well known that SIMD and vector architectures are well suited for processing data in parallel loops, programming SIMD or vector architectures in high-level programming languages has not been popular for application developers. Hardware vendors provide intrinsics, which are pseudo-functions that map directly to the vector instructions. Yet programming with such a method is difficult, time-consuming, and produces code tied to only one hardware architecture. Compiler auto-vectorization has been the most used approach for transforming a sequential loop to vector instructions, thus shielding programmers from the complexity of vector architectures and SIMD instructions. However,

PMAM'22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9339-3/22/04…$15.00
https://doi.org/10.1145/3528425.3529100

this automated approach is often constrained by the capability of parallelizing compilers, and has the disadvantage of preventing users from fine-tuning data parallel loops to SIMD processing. This limits the exploitation of the full capabilities of SIMD architectures enabled in the hardware.

For programmers, using the auto-vectorization approach limits the exploration and application provided by the manual, intrinsic-based approach. Studying what SIMD instructions are mapped to from high-level parallel loops would help programmers to further optimize applications written in a high-level language to better suit SIMD architectures, and consequently, performance. In this paper, we implement the OpenMP SIMD directive using source-to-source compiler transformation so users know exactly how high-level parallel loops are transformed to low-level SIMD intrinsics and instructions. The solution leverages the strength of the two aforementioned approaches as it allows users to specify SIMD processing semantics of data-parallel loops in a program, and the compilers to perform SIMD transformation onto Intel AVX or Arm SVE intrinsics.

The rest of the paper is organized into the following sections. Section 2 addresses the background and motivation of vectorization using a source-to-source compiler with OpenMP. Section 3 discusses the implementation we used. Section 4 gives an overview of the performance evaluation we did on a group of basic kernels. Finally, section 5 discusses related work and section 6 provides a conclusion.

2 Background and Motivation

In the introduction section, we briefly discussed the advantages of using a SIMD solution. We will now go into more detail, by comparing it with other two parallel architectures, i.e. GPUs and multi-core CPUs (via threading). Both threading and GPU offloading yield high performance once they begin work, but the task of either loading data or separating and managing threads often incurs significant overhead. While the performance gains in certain problems offset this, for other problems the gains are minimal enough to where the overhead is not justified. In addition, the efforts of migrating sequential programs to use threading or GPUs are often nontrivial in comparison to using SIMD or vector architectures.

Perhaps the best example of this is with loops. Loops that perform a single operation or a small number of operations on a large set of data do not always lend themselves well to threading. At the same time though, even with loop unrolling, simply sequentially doing each operation one at a time is just as slow. SIMD architectures provide the best of both worlds in these scenarios. At first glance, vector instructions are not all that different from conventional single-operation instructions. A SIMD instruction typically will load a block of data from memory into extra wide registers, and then perform the same operation at once on all the data. When finished, it can be stored back to the specified location. Because it is done in parallel, it takes the same amount of time (in clock cycles) as conventional instructions. This leads to a factor of speedup based on the number of elements in the vector, which can become very substantial over large arrays.

2.1 Approaches to Using Vector Architectures and the Pros and Cons of Each

The majority of programming languages, including the most common- C, C++, and Fortran- do not have any constructs that map directly to vector programming. As a result, either specialized functions are needed, or the compilers have to find a way to do vectorization themselves. There are multiple approaches, with pros and cons to each.

2.1.1 Auto-Vectorization by Compilers. Auto-vectorization is the ideal method, and probably the most researched one. Despite being "auto", it is not always that. Optimizing compilers such as GCC, LLVM, and other vendor compilers do a decent job of analyzing and vectorizing simple data parallel code. They however have challenge of vectorizing loops that requires information that are hard to be analyzed by compiler. For example, a data parallel loop that has reduction operation has loop-carried dependencies. It is often challenging to vectorize, but can be easily vectorized with if the reduction variables are explicitly specified, e.g. using OpenMP simd directive and reduction clause. While the mainstream compilers may do a degree of vectorization on their own, they typically require some sort of user intervention, such as architecture flags to auto-vectorize to the greatest extent. From our own experience, we have found that the compilers often do not optimize fully to the hardware available without several very specific flags- for example, Clang often generated only AVX-2 or even SSE4 instructions despite AVX-512 hardware being available.

2.1.2 Programming with Vector Intrinsics. At the opposite end of the spectrum are vector intrinsics. Intrinsics are pseudo-functions provided by the compiler that have either a 1:1 mapping to the underlying assembly, or are very close to that. Intrinsics offer a big advantage in that the transformation is guaranteed, and depending on the programmer's knowledge of the hardware, a very high level of optimization can be obtained- sometimes greater than that of an optimizing compiler. However, by those statements you can likely see the disadvantages- some of which we discussed earlier. Vectorization by intrinsics is not cross-platform. In other words, if we write a loop that is vectorized using AVX-512 intrinsics, it will not work for Arm SVE. Secondly, vectorizing well requires time and skill, which is not something the programmer always has or should need to have.

2.1.3 Explicit Vectorization, such as OpenMP SIMD. Explicit vectorization is a third option, which one could say

is the middle ground between auto-vectorization and manual vectorization. We argue that this is the ideal solution over implicit auto-vectorization, as it gives the end-user programmer control and transparency over the code, while not forcing the compiler to do anything it cannot do.

While some compilers such as Clang/LLVM and Intel compiler provide directives that can perform this explicit vectorization, solutions such as OpenMP's SIMD directive provide a standard, cross-platform, cross-compiler solution. The OpenMP SIMD directive is used with a for-loop to specify code that should be vectorized. Using various clauses, the user can fine-tune the vectorization specifying things like vector lane width, safe width, alignments, and others.

2.2 The Benefits of Source-to-Source Compiler Transformation

Auto-vectorization is a well-established science. All the mainstream compilers (GCC, LLVM, ICC, etc) support it to varying degrees, and given the right combination of well-written code, compiler flags, and so forth, very good, highly-optimized code can be generated. Although this is great, there are disadvantages to auto-vectorization, some of which we discussed above. One of these disadvantages is that with auto-vectorization, you have only limited control over the vectorization process, and more often than not, the final result is hidden from the user. As a result, you have to assume the compiler knows best and assume the most optimized code will be generated, which is often not true when a compiler has to make conservative decision of vectorization.

Source-to-source compiler transformation offers a good solution to these issues, especially when coupled with an explicit vectorization method such as using the OpenMP SIMD directive. As its name implies, the end result of source-to-source is also source code, which is intrinsics for SIMD that is human-readable and compilable. Let us consider the benefits of this approach as follows:

- Guaranteed vectorization: Because intrinsics are generated, you are guaranteed to have those instructions in the final compiled binary form. You are also guaranteed to have the same style of vectorization you have in your source code.
- Transparency: Because intrinsics are used, you can easily follow what the compiler does to vectorize.
- Control of the code: While you should generally trust the compiler, if you know of a better method to do something, you can make whatever micro-optimizations you need to the generated source.
- Control of the vectorization: The OpenMP explicit SIMD directive allows certain degree of control over the SIMD directive, by utilizing the available clauses to potentially change the outcome based on your needs.

3 Implementation

With this in mind, you can understand why vectorization is so desirable and why source-to-source compiler transformation is a viable option. Our implementation is based on ROSE source-to-sourcec compiler. The compiler generates intrinsics for the Intel AVX and Arm SVE vector architectures. In this section, we will discuss the transformation process of the OpenMP SIMD directive. All SIMD directives follow a general process of lowering first to 3-address code, secondly to an internal SIMD IR, and finally, to the platform-specific intrinsics. The design of our IR makes it easy to add new architectures as needed.

3.1 The Input Code

Although you can convert other types of loops, the OpenMP specification stipulates that the SIMD directive only applies to for-loops. For-loops make for an easier transformation since you can eliminate the extra step of determining the loop increment, as you would have to in other loops. The first step of the transformation is conversion of the for-loop in its abstract syntax tree (AST) form to a 3-address format in the compiler frontend. Consider the AXPY example:

```
1 #pragma omp simd
2 for (int i = 0; i<N; i++) {
3     Y[i] += a * X[i];
4 }
```

Figure 1. AXPY Example in C Language

SIMD architectures follow a RISC-style load/store format. This includes the AVX extensions, even though the rest of x86 uses CISC instructions. Therefore, the first step is conversion to a 3-address format, where the left-hand operand is either a memory location or a scalar variable. Using our compiler, with the "−simd-target=3addr" command line option, we get the following code:

```
1  for (i = 0; i <= 1199; i += 1) {
2      float __vec0;
3      __vec0 = Y[i];
4      float __vec1;
5      __vec1 = a;
6      float __vec2;
7      __vec2 = X[i];
8      float __vec3;
9      __vec3 = __vec2 * __vec1;
10     float __vec4;
11     __vec4 = __vec3 + __vec0;
12     Y[i] = __vec4;
13  }
```

Figure 2. AXPY Example in 3-address Format

The first two lines represent a load to a vector register, as do the lines following it. You will then notice two lines representing math instructions. Finally, you will notice a store instruction. This code is all valid C. But, because of the format, we can look directly at these lines in the next pass, and make a 1:1 conversion to platform-specific intrinsics.

Multi-dimensional arrays posed an interesting challenge. Let's say you have an array reference like this: A[i][j]. Ideally, you would want to convert it to something like A[i*N+j]. The solution was to create an intermediate pointer variable, which would effectively convert it to a single dimension. Take this for the A[i][j] example:

```
1 float *__ptr0 = A[i][0];
2 float __vec1 = __ptr0[j];
```

Figure 3. Intermediate Pointer Variable for A[i][j]

3.2 The SIMD IR

After lowering the program AST to its 3-address format, we then convert everything to an internal IR which we call SIMD IR. The SIMD IR is a simple IR based off the ROSE AST. We did think of originally doing our own internal IR, but we decided that creating new specialized classes from the existing AST made more sense, and would lead to a simpler design. The SIMD IR nodes are based off the nodes for binary operations, which allow for a left and right operand. Because the SIMD IR is not meant to represent any concrete code on its own, it is not used to replace the original program AST.

The SIMD IR has proved to be a very efficient method of implementation. The IR has an almost 1:1 mapping to whatever the underlying architecture is. We originally started with the Intel AVX architecture. After we had that working, adding Arm SVE support only took about a week because of the nature of the architecture-neutral IR for SIMD.

3.3 The Final Transformation

The final step is lowering from SIMD IR to platform-specific intrinsics. As talked about in the previous section, this is mostly a 1:1 translation.

Depending on the instruction, some optimizations can be made. For example, when one of the mathematical operands is a scalar variable or value, it is most efficient to broadcast that value outside the loop, and reuse the register on each iteration. Another example is when you are storing to a scalar variable (as in the case of reduction). Maintaining a register with partial values, and then reducing that register to a final result outside the for loop when it finishes executing is the most efficient implementation, and leads to very significant speed increases.

3.4 Supported Vector Operations

While we currently do not support every vector operation, our compiler is currently capable of transforming the most common ones, which covers a wide area. Vector architectures have expanded significantly with new generations of architectures, with instructions for masks, conditional statements, comparisons, type conversions, and complicated mathematical operations. We hope to add more support in the future. In this section, we will break down our current areas of support.

3.4.1 Math Operations and Data Types. The conventional add, subtract, multiply, and divide operations are all supported. New operations can easily be added with new IR types. Floats, doubles, and 32-bit signed integer operations are all supported. One of our goals is to provide casting support to allow for types to be interchanged as needed.

3.4.2 Load, Store, and Broadcast. Conventional load and stores for all types are fully supported. As mentioned in the first subsection, loads and stores of multi-dimensional arrays require the address to be broken down. Broadcasts are also fully supported. To enhance speed and performance, a broadcast is only performed once, and placed before the vectorized loop.

3.4.3 Masking (Gather and Scatter). Masked loads, or gathers, are a new feature we added to support indirect loads. Originally, we used a gather intrinsic that only used an integer array to determine what should be loaded from the source array. However, this was highly inefficient and led to only negligible performance gains. After examining Clang's method, we decided to use specialized mask registers, a feature introduced with AVX-512. Using this method resulted in substantial performance gains, eventually outperforming the Clang version we originally used as reference.

Scatter, or masked stores, is also supported, but we did not have a kernel applicable to testing this. However, scatters are very similar to gathers (the only difference being the obvious-one loads and one stores), so we expect that in a scenario where a scatter is required, a similar ratio of performance gains would be observed.

3.5 The OpenMP SIMD Directive

The OpenMP SIMD directive is used to annotate a for-loop for compiler to performance vectorization of the loop. The OpenMP specification for SIMD directive also outlines several clauses that can be used with the directive to control vectorization.

3.5.1 The simdlen & safelen Clauses. The "simdlen" and "safelen" clauses can be used to control the vector length. "Simdlen" denotes the length we wish to use, while safelen denotes the maximum (or safe) length we should use.

Currently, these clauses only affect vectorization on the Intel AVX architecture. The vector lengths of AVX must be known at compile time for vectorization. Because of the multiple extensions that are used, there are three commonly used vector lengths, i.e. 4 for SSE that uses XMM* vector registers, 8 for AVX2 that uses YMM* vector registers, and 16 for AVX-512 that uses ZMM* vector registers. We currently use a straightforward approach of rounding to the nearest applicable vector length. For example, if the user specified an odd number such as 12, we would round up to 16. Anything less than or equal to 8 would use the AVX2 extensions.

Currently, "simdlen" and "safelen" are ignored for the Arm architecture, because Arm SVE is designed to be vector length agnostic.

3.5.2 Reduction. The reduction clause is used to indicate accumulation of data elements that are computed by the loop. To vectorize a loop that has reduction, a vector register is initialized before the loop with the values corresponding to the reduction operations, e.g. 0 for addition, and 1 for multiplication. Let us name it as reduction register. Then in each loop iteration, the reduction register is used for accumulating the partial results for the reduction operation. However there is no need to accumulate the elements of the reduction register inside the loop. Finally, at the end of the loop, the elements in the reduction register are accumulated to produce a single reduction result.

Thus in general, vectorization is not parallel in the sense of threading, this translates to maintaining a register with partial results. At the end of the loop, the elements in this register are accumulated to produce the final result, which is then stored to the variable indicated in the reduction clause. Both Intel AVX and Arm SVE have horizontal math instructions for accumulating the elements of a vector register, which eliminates the need for an additional loop for accumulating register elements for implementing reductions. For Intel AVX, horizontal pair-wise accumulation such as VHADDPS with intrinsics such as _mm256_hadd_ps can be used for addition reduction. Multiple of those instructions are needed to accumulating all the elements of a register into one result since the instruction performs tree-style reduction. For Arm SVE, only one horizontal reduction instruction, e.g. faddv, is needed to accumulate all the elements of a vector register to produce a single reduction results. We believe the Arm SVE's horizontal reduction is implemented in the same way of pair-wise accumulation.

When testing auto-vectorization, we found that mainstream optimizing compilers were unable to automatically create a reduction operation on their own, even in very simple cases such as a simple array reduction. Even with various compiler flags and after consulting the documentation, in no case was Clang or GCC unable to vectorize a reduction without some sort of explicit instruction in the code. Clang has a pragma that can force the reduction, which still yields excellent performance and correct computational results, but of course this leads to the obvious implication that it is neither portable across compilers nor is this auto-vectorization to begin with. In some cases, the compiler can make up for this with methods such as loop unrolling, but only in naive examples such as AXPY can the user reasonably count on their compiler to perform auto-vectorization.

3.5.3 `adda` and `addv` Instructions in ARM SVE for reduction. It is important to note that Arm SVE also provides another instruction to implement reduction operation, namely `adda`. `adda` is a left-to-right reduction operation, whereas `addv` uses reduction trees. In the left-to-right approach of `adda`, all elements of the vector are added sequentially starting from the lowest position and moving to the highest. This approach is very similar to a sequential loop. `addv`, on the other hand, uses a tree-shape algorithm to recursively perform pairwise reduction of all elements of the vector.

Consider an example. Let us assume vector A contains 16 float elements, E0-E15. We need to accumulate all elements of vector A and store them in scalar register R. Since we are operating on floating-point elements, we will use the `fadda` and `faddv` instructions. The `fadda` instruction will add all the elements in order, starting from E0 and working up to E15. When using `faddv`, the elements in A will first be divided into two parts. Each part is then recursively divided into smaller parts until only two elements are left to do the addition. At this point, all the elements are accumulated from bottom-to-top to get the final result.

In our experiment, we used Arm C/C++/Fortran Compiler 21.0 based on LLVM. By default, the compiler chooses `fadda` to implement the reduction. In most cases, using this instruction resulted in worse performance. If we want to the compiler to use `faddv`, we need to use the `-ffp-mode=fast` compiler flag when compiling the source OpenMP code. The experimental results in Section 4 include the efficiency comparison of the two methods. Our compiler only supports the generation of faster `faddv` instructions.

4 Evaluation

We evaluate our compiler on two machines: our Carina server for Intel AVX, and the Ookami cluster from Stonybrook University, which uses the ARM AArch64 CPUs that support ARM SVE. To provide as meaningful a comparison as possible, the hardware we chose has a similar core count, SIMD lane width, and CPU frequency. Table 1 shows the configurations we used in more detail.

To evaluate performance, we used five kernels, all of which encompass the fundamental vector operations and form the base of more complicated problems. We ran three versions of each kernel. The first version was compiled without any optimization or explicit vectorization. The results from this test formed the baseline. The second version used

Table 1. Summary of the Configuration of Test Platforms

Server Name	Carina	Ookami
Model name	Intel(R) Xeon(R) Gold 6230N CPU @ 2.30GHz	Fujitsu A64FX @ 2.2GHz
Architecture	x86_64	aarch64
Number of Cores	40	48
Vector Length	512-bit for AVX-512, and 256-bit for AVX2	512-bit ARM SVE
Memory	512GB DDR4-2933 DRAM	32GB HBM
OS	Ubuntu 18.04 LTS	CentOS Linux release 8.1.1911 (Core)
Compilers	Clang/LLVM 12.0.1	Arm C/C++/Fortran Compiler 21.0 based on LLVM 12.0.1

the OpenMP SIMD directive to explicitly vectorize the loop and compiled using Clang/LLVM 12.0.1 on Intel CPU and Arm C/C++/Fortran compiler on Arm CPU. Finally, the third-version is the same as the second version, but compiled using our source-to-source compiler. On both machines, we used Clang/LLVM 12.0.1, although on Arm, we used their fork of LLVM 12.0.1 called "armcompiler". The serial version was compiled using level-0 optimization, while the two SIMD versions were compiled using level-2 optimizations with all native extensions enabled. Although int-32 and double datatypes had similar acceleration ratios, we used the float datatype in all our tests. Except for our last kernel, we ran each test 20 times from within the program. All performance times are measured in seconds.

We also measured code size, but we found that the size was generally comparable between Clang and our compiler. In general, the deviation was not more than roughly 0.02%, depending on the kernel. This is largely expected since the assemblies between Clang and our compiler were similar.

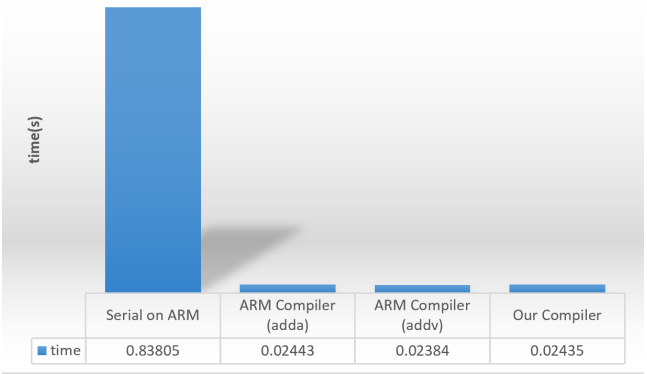
4.1 AXPY

AXPY is a classic, fundamental test for vectorization. It encompasses the most common vector operations: addition, multiplication, and broadcasting. In order to get measurable results, for this test we used a problem size of 102,400,000 elements. The results can be seen in Fig. 4.

AXPY is a very straightforward program, so Clang can automatically vectorize it. The test results on Intel are generally in line with our expectations. Since AXPY is not a computationally intensive program and its execution efficiency is limited by data movement, similar results are obtained using auto-vectorization and OpenMP SIMD directive.

The results on Arm are very close, and upon examining the assembly, the outputs for the AXPY function were very similar. The same is true for the Intel AVX. However, we also noticed that Clang uses AVX2 instructions by default, even with AVX-512 present, and the performance is similar to our version, which uses AVX-512. Using compiler flag "-march=knl", Clang would compile to the AVX-512 instructions. However, the performance with AVX-512 is roughly equal to that of AVX2 in most situations. A full explanation of this can be found near the end of this section.

(a) Runtime (seconds) for AXPY: on Intel 6230N for AVX2 (256-bit vector length) and AVX-512 (512-bit vector length)



(b) Runtime (seconds) for AXPY: on Fujitsu A64FX (512-bit vector length)

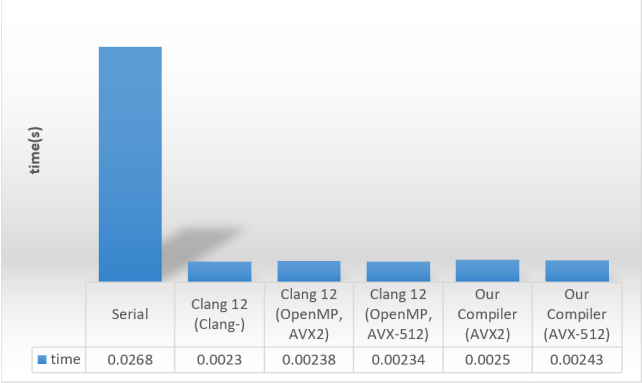
Figure 4. Runtime (seconds) for AXPY: on Intel 6230N for AVX2 (256-bit vector length) and AVX-512 (512-bit vector length) and on Fujitsu A64FX (512-bit vector length), problem size: 102400000

4.2 Sum

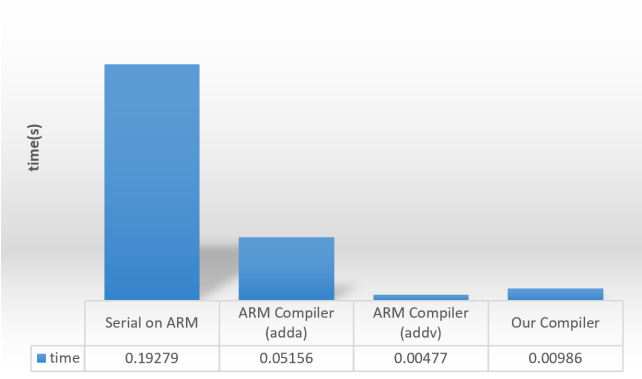
The sum test is classic, fundamental test for reduction, a commonly used operation in parallel and vectorized programs. In this test, we used a problem size of 10,240,000 elements. The results can be see in Fig. 5.

The performance on Intel AVX was predictable and yielded the same results. The results on Arm SVE, however, were interesting and varied based on whether the "fadda" or the "faddv" instruction was used. By default, the Arm compiler uses the "fadda" instruction, and while using this is certainly better than the serial, it is slower than that generated by our compiler. However, when the "-ffp-model=fast" flag is passed to the Arm compiler, it generates significantly faster results than the "fadda" version.

Clang is not able to auto-vectorize the Sum example because of the existence of reduction in the loop body. We then added Clang's pragma of "loop vectorize(enable) as reference to enable vectorization by Clang/LLVM. Both the performance and the correctness were almost identical to the OpenMP versions. Upon examining the source code, we found that the assembly for the vectorized loop was almost identical, leading us to think that the compiler used the same routines to perform the vectorization.



(a) Runtime (seconds) for Sum: on Intel 6230N for AVX2 (256-bit vector length) and AVX-512 (512-bit vector length). Clang/LLVM is not able to auto-vectorize the code. We then added "pragma clang loop vectorize(enable)", the code is vectorized, and the performance shows this versions

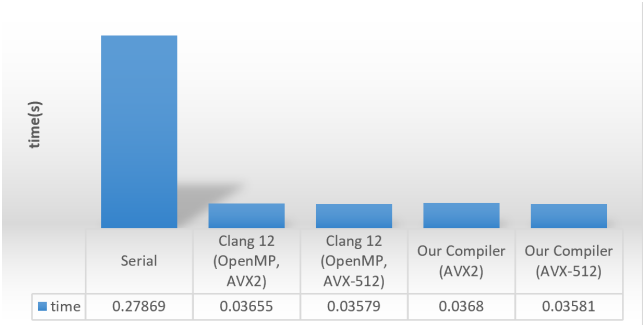


(b) Runtime (seconds) for Sum: on Fujitsu A64FX(512-bit vector length)

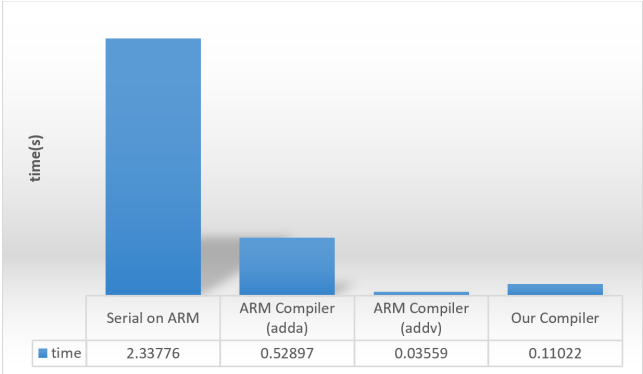
Figure 5. Runtime (seconds) for Sum: on Intel 6230N for AVX2 (256-bit vector length) and AVX-512 (512-bit vector length), and on Fujitsu A64FX(512-bit vector length), problem size: 10240000

4.3 Matrix-Vector

Both AXPY and Sum are memory intensive kernels in that elements are read only once and the computational intensity is low according to the roofline model. Matrix-vector multiplication can be considered a combination of AXPY and Sum; thus it is memory intensive as well. The matrix data is read once and the vector array is reused multiple times. In this test, we used a matrix of 10240 X 10240 size, and a vector of 10240 elements. The results can be seen in Fig. 6.



(a) Runtime (seconds) for Matrix-Vector Multiplication: on Intel 6230N for AVX2 (256-bit vector length) and AVX-512 (512-bit vector length)



(b) Runtime (seconds) for Matrix-Vector Multiplication: on Fujitsu A64FX (512-bit vector length)

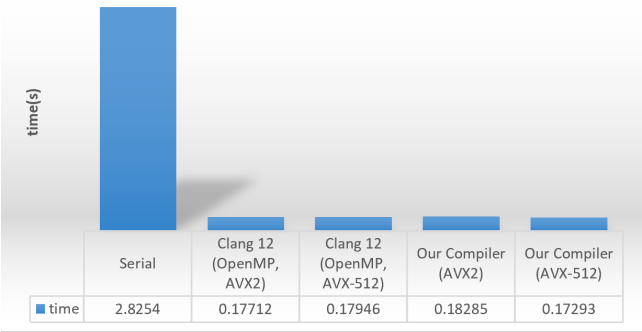
Figure 6. Runtime (seconds) for Matrix-Vector Multiplication: on Intel 6230N for AVX2 (256-bit vector length) and AVX-512 (512-bit vector length) and on Fujitsu A64FX (512-bit vector length), input matrix size: 10240 * 10240, input vector size: 10240

The results on Intel were all very similar between AVX2 and AVX-512 and between Clang and our compiler. The results on Arm were surprising at first as our version significantly outperformed the Clang version. Upon inspection, once again we found that the Arm compiler was using "fadda" for reductions by default. Adding the "-ffp-model=fast" flag

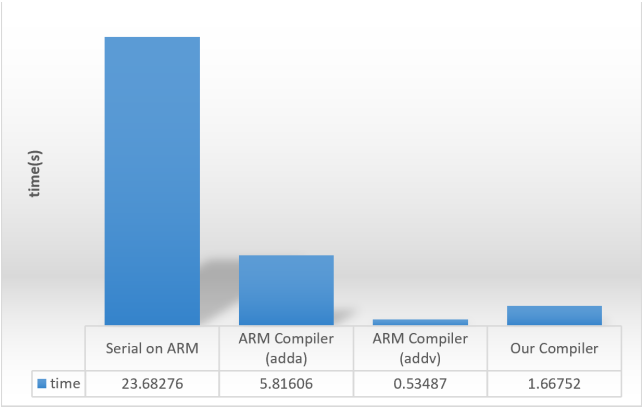
significantly improved performance. With this flag, the Arm compiler out performed our version somewhat. The assembly outputs were very similar, but the reason for the performance gain appears to be the "faddv" instruction (as expected) and extensive use of the "fmla" instruction (fused-multiply-add).

4.4 Matrix-Matrix

The matrix-matrix kernel was our most intensive test. It is also very similar both in design and results to the matrix-vector test. For this test, we used a problem size of 1024 (two matrices of 1024 X 1024). The reasons behind the results are the same as those behind matrix-vector. The results can be seen in Fig. 7.



(a) Runtime (seconds) for Matrix-Matrix Multiplication: on Intel 6230N for AVX2 (256-bit vector length) and AVX-512 (512-bit vector length)



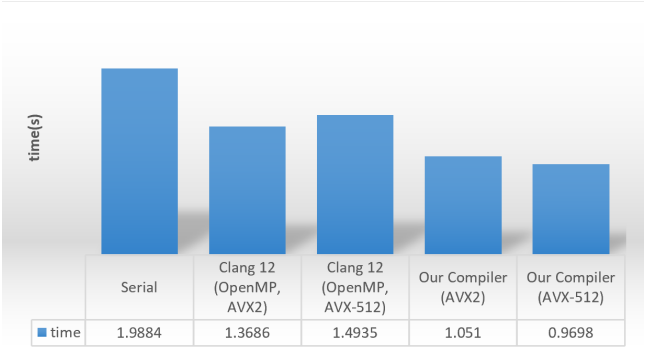
(b) Runtime (seconds) for Matrix-Matrix Multiplication: on Fujitsu A64FX (512-bit vector length)

Figure 7. Runtime (seconds) for Matrix-Matrix Multiplication: on Intel 6230N for AVX2 (256-bit vector length) and AVX-512 (512-bit vector length) and on Fujitsu A64FX (512-bit vector length), problem size: 1024 * 1024

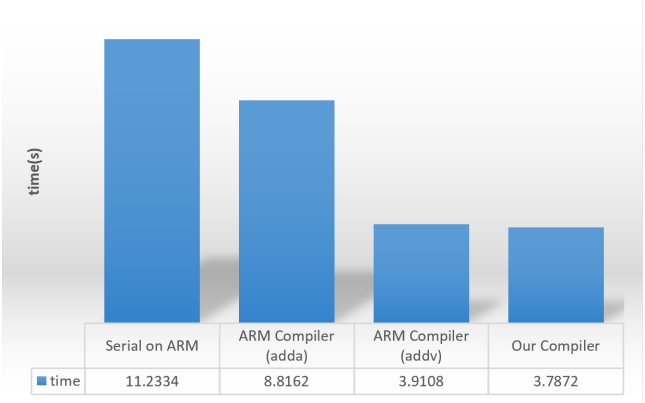
4.5 SparseMV

The sparse matrix-vector (MV) kernel provides an example of indirect memory access, an expensive and challenging operation to optimize. For this test, we used a problem size of 10,240 elements, but unlike the previous tests, the algorithm was only ran once due to the intensity of the calculation. The results are in Fig. 8.

The Arm SVE results were not surprising, but the Intel AVX results were interesting. Up until now, AVX-512 was getting roughly the same performance as AVX2, but here the AVX-512 versions from Clang and from our compiler outperformed the AVX2 versions. The reason is because of the gather instructions available on AVX-512. AVX-512 introduces masking registers and instructions, which is a more efficient way to do indirect memory accesses than directly using array indices as AVX2 requires.



(a) Runtime (seconds) for Sparse Matrix-Vector Multiplication: on Intel 6230N for AVX2 (256-bit vector length) and AVX-512 (512-bit vector length)



(b) Runtime (seconds) for Sparse Matrix-Vector Multiplication: on Fujitsu A64FX (512-bit vector length)

Figure 8. Runtime (seconds) for Sparse Matrix-Vector Multiplication: on Intel 6230N for AVX2 (256-bit vector length) and AVX-512 (512-bit vector length) and on Fujitsu A64FX (512-bit vector length), input matrix size: 10240 * 10240, input vector size: 10240

4.6 Explanation of Intel Results for AVX2 and AVX-512

Despite Intel AVX-512 having double the lane width of AVX2, the fact that there was little to no performance increase is likely surprising. We inspected the assembly for each case, and we found no differences that could explain such a discrepancy. We then considered the nature of the computation and did some research on Intel's architecture and on other cases [15], so we have some ideas of the reason.

The first thing to consider is the arithmetic intensity of the kernels we use. Except for matrix multiplication, the kernels we used are mostly memory intensive. Loading and storing from memory is the longest part of the operation, and the lack of data reuse makes the cache less helpful. Regardless of the vector length, the same amount of memory has to be accessed, and while using a vector does improve performance compared to doing one operation at a time, there will reach a point where performance will began to plateau, and this point seems to be at the 256-bit (AVX2) length.

The second thing to consider is the latency of the instructions. We checked the Intel specs, and found that AVX-512 has a higher latency than the equivalent load instruction on AVX2. While the latency difference is not huge- 7 on Icelake and Skylake for AVX2 as opposed to 8 on the same series for AVX-512- over a large operation size, the difference will begin to add up. While this does not explain the closeness of the measurements on its own, we suspect this is a factor.

The third and mostly likely reason is CPU frequency throttling [3]. When AVX-512 instructions are used, the CPU down-throttles the frequency to keep it within its power limits. While this occurs on AVX2, the much greater length of AVX-512 leads to significant down-throttling. While more work is technically done per loop iteration, the down-throttling means that more time is required for the work to be carried out, resulting in the performance loss. This is the reason GCC and Clang default to AVX2, even on high optimization levels. AVX2 seems to be the highest extension that can be used without significant down-throttling.

5 Related Work

5.1 SIMD Optimization Technology

Dorit Nuzman et al. demonstrated vectorization for outer loops [11]. They re-studied the method of outer loop vectorization and paid attention to the properties of modern short SIMD architecture. Compared with the innermost loop vectorization which can only provide an acceleration factor of 1.53, the outer loop vectorization can provide a significant performance improvement with a factor of 3.13x.

Ken Kennedy et al. proposed a context optimization method for SIMD execution [5]. Disabling the processor that is not involved in the current calculation by changing the machine context is a problem that the SIMD compiler must solve. They optimize context partitioning and context splitting to reducing the cost of context changes. Their method can reduce the execution time of the hand-optimized MPL version by 12%. Angela Pohl et al. addressed potential performance concerns for the vector length agnostic (VLA) model introduced by Arm and being added to RISC-V [12]. They showed that VLA code reaches 90% that of fixed length models (ie, Intel). Additionally, they showed that due to higher memory demands, performance does not increase proportionally with higher vector lengths.

5.2 Source to Source Vectorization

We found that the only source-to-source vectorization work was Scout proposed by Olaf Krzikalla et al. in 2011. Scout is designed as a source-to-source translator that generates code with SIMD intrinsics [7]. Based on the syntax tree generated by the Clang parser, they implement the vectorization optimization technique. The unroll-and-jam technique is applied to vectorize the loop body, supporting multiple SIMD instruction sets including SSE and AVX2. In the experiments, they observed that the overall speedup was nearly 1.5. Unlike Scout, our work has unique advantages. Firstly, Scout only supports vectorization with target languages AVX2 and SSE. We support the more modern and higher-performance AVX-512, and we support the new ARM SVE extensions, giving us cross-platform support. Secondly, we use more common algorithms as our benchmarks to test performance. In a variety of algorithms, our designs have very good performance optimizations.

5.3 Auto-Vectorization

Michael Klemm first proposed the OpenMP SIMD instruction in 2012 [6]. The SIMD instruction will allow programmers to guide the vectorization process, thereby achieving more efficient and portable use of the SIMD level. Florian Wende et al. explored coding strategies to improve SIMD performance on different compilers and platforms [14]. They proposed a portable SIMD coding scheme called "enhanced explicit vectorization" for which different compilers can understand and generate code equally. The microbenchmarks they developed show that on Haswell, SIMD execution gives speedups between 2x and 4x. Oliver Reiche et al. proposed automatic vectorization technologies for image processing DSL in the context of source-to-source compilation, and integrated these technologies into the open-source DSL framework [13]. Compared with the non-vectorized execution using the latest existing C/C++ compiler, the geometric average speed of benchmarks obtained from ISPC and image processing has increased by 3.14.

Ameer Haj-Ali et al. proposed an end-to-end method that can vectorize loops using deep reinforcement learning(RL) [4]. They integrated RL in LLVM compiler and used deep learning to capture different instructions, dependencies, and data

structures to determine the optimal vectorization factor dynamically. Their experiments show that the performance can be increased up to 4.73 times compared with the baseline.

Dorit Nuzman et al. developed a compiler containing a new generic vectorization technique for interleaved data, which can effectively vectorize non-contiguous access patterns with a constant stride of power of 2 [10]. The SIMD models available today do not directly support operations on disjoint vector elements. For interleaved data, once it is correctly reorganized, it will significantly benefit from SIMD. Their experimental results show that for interleaving levels (stride) as high as 8, the speedups in execution time can be up to 3.7. They also developed an automatic vectorization program in GCC [9]. Their design can be adapted to various SIMD architectures, and different alignment mechanisms are designed for different SIMD platforms.

Matthew Lambert et al. demonstrated how the auto-vectorization capabilities of Clang/LLVM and GCC could be used to enhance performance of the Four Russians Matrix Multiplication problem [8]. They used a method to store multiple small prime numbers into a single word called bit-packing. This in combination with vectorization yielded significant speedups.

For auto-vectorization, checking whether the code can be vectorized is necessary. But in our source-to-source vectorization, we can avoid the overhead of this step. The SIMD directive in OpenMP can directly indicate the part that needs to be vectorized. Therefore, the design becomes more straightforward and yields performance.

6 Conclusion

In this paper, we demonstrated the merits of source-to-source vectorization using our compiler combined with explicit vectorization using OpenMP. Vectorization using SIMD architectures has become an important part of the optimization process in many high-performance and scientific applications. While it is a well-researched field, and while many of the mainstream compilers do a rather good job at it, there are disadvantages to the auto-vectorization process, one of them being the most important: applicability. At the same time, there are also disadvantages to the manual vectorization process, another more difficult alternative. Explicit source-to-source vectorization bridges the gap, bringing the convenience of auto-vectorization and the control of manual, intrinsic-based vectorization. Performance evaluations lend weight to this conclusion. In no case was our source-to-source vectorization significantly worse. In all cases, it was almost as good, if not better.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No 2015254. The authors would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the innovative high-performance Ookami computing system, which was made possible by a $5M National Science Foundation grant (1927880).

References

[1] [n. d.]. *ARM C Language Extensions for SVE*. https://developer.arm.com/documentation/100987/0000/

[2] [n. d.]. *Intel® AVX-512 Instructions*. https://software.intel.com/content/www/cn/zh/develop/articles/intel-avx-512-instructions.html

[3] Mathias Gottschlag, Philipp Machauer, Yussuf Khalil, and Frank Bellosa. 2021. Fair Scheduling for AVX2 and AVX-512 Workloads. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 745–758. https://www.usenix.org/conference/atc21/presentation/gottschlag

[4] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 242–255.

[5] Ken Kennedy and Gerald Roth. 1994. Context optimization for SIMD execution. In *Proceedings of IEEE Scalable High Performance Computing Conference*. IEEE, 445–453.

[6] Michael Klemm, Alejandro Duran, Xinmin Tian, Hideki Saito, Diego Caballero, and Xavier Martorell. 2012. Extending OpenMP* with vector constructs for modern multicore SIMD architectures. In *International Workshop on OpenMP*. Springer, 59–72.

[7] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E Nagel. 2011. Scout: a source-to-source transformator for SIMD-optimizations. In *European Conference on Parallel Processing*. Springer, 137–145.

[8] Matthew A. Lambert and B. David Saunders. 2017. Compiler Auto-Vectorization of Matrix Multiplication modulo Small Primes. In *Proceedings of the International Workshop on Parallel Symbolic Computation* (Kaiserslautern, Germany) *(PASCO 2017)*. Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/3115936.3115943

[9] Dorit Nuzman and Richard Henderson. 2006. Multi-platform auto-vectorization. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 11–pp.

[10] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of interleaved data for SIMD. *ACM SIGPLAN Notices* 41, 6 (2006), 132–143.

[11] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop vectorization: revisited for short simd architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2–11.

[12] Angela Pohl, Mirko Greese, Biagio Cosenza, and Ben Juurlink. 2019. A Performance Analysis of Vector Length Agnostic Code. In *2019 International Conference on High Performance Computing Simulation (HPCS)*. 159–164. https://doi.org/10.1109/HPCS48598.2019.9188238

[13] Oliver Reiche, Christof Kobylko, Frank Hannig, and Jürgen Teich. 2017. Auto-vectorization for image processing DSLs. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 21–30.

[14] Florian Wende, Matthias Noack, Thomas Steinke, Michael Klemm, Chris J Newburn, and Georg Zitzlsberger. 2016. Portable simd performance with openmp* 4. x compiler directives. In *European Conference on Parallel Processing*. Springer, 264–277.

[15] Hong Zhang, Richard T. Mills, Karl Rupp, and Barry F. Smith. 2018. Vectorized Parallel Sparse Matrix-Vector Multiplication in PETSc Using AVX-512. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3225058.3225100