Evaluating Auto-Vectorizing Compilers through Objective Withdrawal of Useful Information

SERGI SISO, Hartree Centre and University of Liverpool
WES ARMOUR, University of Oxford
JEYARAJAN THIYAGALINGAM, Rutherford Appleton Laboratory

The need for compilers to generate highly vectorized code is at an all-time high with the increasing vectorization capabilities of modern processors. To this end, the information that compilers have at their disposal, either through code analysis or via user annotations, is instrumental for auto-vectorization, and hence for the overall performance. However, the information that is available to compilers at compile time and its accuracy varies greatly, as does the resulting performance of vectorizing compilers. Benchmarks like the Test Suite for Vectorizing Compilers (TSVC) have been developed to evaluate the vectorization capability of such compilers. The overarching approach of TSVC and similar benchmarks is to evaluate the compilers under the best possible scenario (i.e., assuming that compilers have access to all useful contextual information at compile time). Although this idealistic view is useful to observe the capability of compilers for auto-vectorization, it is not a true reflection of the conditions found in real-world applications.

In this article, we propose a novel method for evaluating the auto-vectorization capability of compilers. Instead of assuming that compilers have access to a wealth of information at compile time, we formulate a method to objectively supply or withdraw information that would otherwise aid the compiler in the auto-vectorization process. This method is orthogonal to the approach adopted by TSVC, and as such, it provides the means of assessing the capabilities of modern vectorizing compilers in a more detailed way.

Using this new method, we exhaustively evaluated five industry-grade compilers (GNU, Intel, Clang, PGI, and IBM) on four representative vector platforms (AVX-2, AVX-512 (Skylake), AVX-512 (KNL), and AltiVec) using the modified version of TSVC and application-level proxy kernels. The results show the impact that withdrawing information has on the vectorization capabilities of each compiler and also prove the validity of the presented technique.

CCS Concepts: • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: Compiler evaluation, vectorization capability, auto-vectorization, vectorization test suite

ACM Reference format:
Sergi Siso, Wes Armour, and Jeyarajan Thiyagalingam. 2019. Evaluating Auto-Vectorizing Compilers through Objective Withdrawal of Useful Information. *ACM Trans. Archit. Code Optim.* 16, 4, Article 40 (October 2019), 23 pages.
https://doi.org/10.1145/3356842

This is a new article, not an extension of a conference paper.
Authors' addresses: S. Siso, Hartree Centre, Science and Technology Facilities Council, Keckwick Lane, Daresbury, UK, University of Liverpool, Brownlow Hill, Liverpool, UK; email: sergi.siso@stfc.ac.uk; W. Armour, Department of Engineering Sciences, University of Oxford, Keble Road, Oxford, UK; email: Wes.Armour@oerc.ox.ac.uk; J. Thiyagalingam, Science and Technology Facilities Council, Rutherford Appleton Laboratory, Harwell Campus, Oxford, UK; email: t.jeyan@stfc.ac.uk.

1544-3566/2019/10-ART40
https://doi.org/10.1145/3356842

1 INTRODUCTION

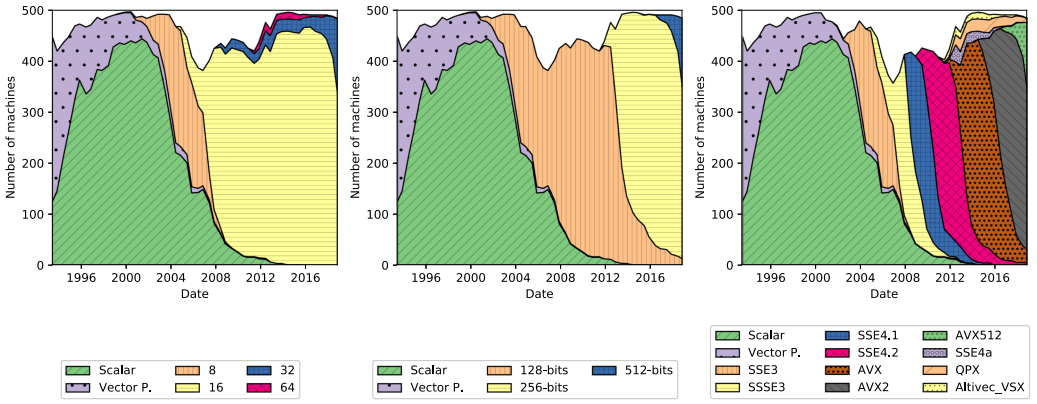
Vectorization is an essential optimization for maximizing the performance of applications on modern systems. The key principle behind vectorization is to identify opportunities to perform a similar set of operations over multiple data elements using a single instruction. Architectures that support such a form of parallelism are often referred to as Single Instruction Multiple Data (SIMD) systems. Modern compilers are able to automate, to a certain extent, the process of generating vectorized code. However, the exact vector performance of the resulting code depends on several factors, including type of application, quality of code, length of the vector registers, types of operations, and the number of available vector units. Depending on the programming language of choice, potential candidates for vectorization include loop nests, abstract statements operating over arrays, and memory accesses [25]. Ideally, compilers should identify all possible candidates for vectorization and then vectorize them using the available SIMD instructions to perform as many operations in parallel as possible.

Although modern compilers provide automatic vectorization optimizations, the capabilities of compilers to fully auto-vectorize a given piece of code are often limited. This limitation primarily stems from the limited capabilities of compilers to extract or perceive relevant contextual information from the code. For instance, values of many variables become known only at runtime, which effectively prevents a number of vectorization opportunities at compile time. Even in cases where the contextual information is available, compilers are conservative in their transformations. The lack of or insufficient information at compile time forces compilers to generate suboptimal vectorized code [16, 28, 31].

As such, it is important to evaluate the capabilities of different compilers to auto-vectorize software under multiple conditions when targeting vector achitectures. The Test Suite for Vectorizing Compilers (TSVC) [5] and extensions of TSVC provided by Maleki et al. [19] were able to set a precedent for evaluating the vectorization capabilities and are still widely used. TSVC and other similar benchmarks usually consist of a set of loops with specific implementation patterns, designed to demonstrate the vectorization capabilities of compilers. These implementations come with a generous amount of contextual information provided to the compiler. In other words, this approach provides a single picture of the performance of compilers under a set of preset scenarios, which are close to the ideal case. Although it is useful to know the best possible vectorization capability that a compiler can deliver, the approach fails to capture the full spectrum of compiler behavior when a limited amount of information is available. In practice, due to the realities of complex software engineering, many scientific applications lack a substantial amount of information that would be beneficial for vectorization at compile time.

Furthermore, since TSVC was first developed, both the compiler and processor landscapes have changed rather dramatically. Specifically, the number of processors supporting vectorization capabilities have evolved substantially. A notable and well-consolidated resource that demonstrates this change is the list of the Top-500 supercomputers in the world [10]. By manually parsing and compiling the Top-500 list, associated system manuals, and relevant processor manuals of the Top-500 systems between the years 1993 and 2018, we show how the capability of vector processing has changed over the course of 25 years in Figure 1.

This analysis focuses exclusively on host processors and does not take the accelerator hardware on the systems into account. Nevertheless, several observations can be drawn from the figure. First, systems without SIMD vector capabilities are now almost nonexistent. Second, the length of vector registers of the newest machines quadrupled (from 128 to 512 bits) in recent years, and the adoption of new instruction sets happens rather quickly. Finally, as Figure 1(c) shows, systems with AVX-2 instruction sets currently dominate the Top-500 list, but AVX-512 is rapidly gaining popularity.



(a) Number of Vector Registers. (b) Length of Vector Registers. (c) Vector ISAs.

Fig. 1. Variation of different vector capabilities in Top-500 systems over the past 20 years.

In fact, in addition to what is observed in the Top-500 list, there is also a thriving research community around modern vector Instruction Set Architectures (ISAs) such as the ARM Scalable Vector Extension (SVE) [30] and the RISC-V "V" Vector Extension [32]. Although at this time there are no production-level systems that make use these modern ISAs, they demonstrate the increasing interest in vector processing.

Instead of assessing the vectorization performance of compilers by supplying all of the useful information at compile time, an alternative approach is to objectively withdraw information that would otherwise aid in auto-vectorization and monitor the resulting performance.

In this article, we propose an extension to TSVC that controls the amount of information provided at compile time and assesses the benefit that releasing this information has toward auto-vectorization. Such an approach complements the TSVC benchmark suite and provides a more extensive picture of modern compiler auto-vectorization capabilities in a broader range of scenarios that are often encountered in production software. In doing this, we make the following contributions:

- We present an approach to extend TSVC to develop a more powerful framework for evaluating the efficacy of auto-vectorizing compilers.
- We formulate a set of well-defined classes of information that are influential toward vectorization and can be withdrawn or supplied to compilers.
- We demonstrate how objectively withdrawing and/or supplying information at compile time affects auto-vectorization.
- We demonstrate the applicability and utility of our method by evaluating and comparing the auto-vectorization capabilities of five different compilers (GNU, Intel, Clang, PGI, and IBM) across four representative contemporary vector platforms (AVX-2, AVX-512 (Skylake), AVX-512 (KNL), and AltiVec).
- We outline a new visualization methodology, which we call the *vectorization efficiency spectrum*, for easily representing and understanding the auto-vectorization efficiency of compilers.

The rest of this article is organized as follows. Section 2 outlines the TSVC benchmark suite and other related work, highlighting the limitations of using these benchmarks to evaluate the auto-vectorization of modern compilers. Section 3 describes the new proposed approach, including the metrics used to quantify the auto-vectorization efficiency and a method for visualizing the

efficiency under different settings. Section 4 describes the evaluation platforms. Section 5 presents the evaluation results on the chosen architectures and compilers. Finally, we conclude in Section 6 with a brief description of the main observations and directions for further work.

2 RELATED WORK

Given that the majority of the applications, or significant parts therein, are developed in a scalar style and expect the compiler to assume the responsibility of vectorizing them, there is a compelling reason to determine the efficacy of compilers in their auto-vectorization capabilities. One such approach, developed by Callahan et al. [5] is TSVC. This suite contains 100 FORTRAN loops with scalar semantics organized in four main categories: dependence analysis, vectorization, idiom recognition, and language completeness. These loops were used to evaluate different compiler-architecture combinations. Over time, TSVC became one of the robust mechanisms to assess the auto-vectorizing capabilities of compilers. However, with the development of new architectures and advances in software engineering practices (TSVC had several loops with GOTO statements), the suite became progressively outdated.

Maleki et al. [19] provided a revision of TSVC by converting the original loops to the C programming language, and extended the loops to cover additional vectorization issues that were not considered of the original suite. This resulted in 151 loops in the extended suite. Their work, published in 2011, evaluated the GNU GCC, the Intel ICC, and the IBM XLC compilers, and showed that ICC auto-vectorized 90 loops, whereas XLC and GCC auto-vectorized 68 and 59 loops, respectively. The authors also highlighted three main reasons for compilers failing to vectorize loops: (i) hardware limitations of the current vector extensions; (ii) compilers not being designed to support some programming patterns, such as loops with wrap-around variables (e.g., C unsigned integers where overflows/underflows wrap from 0 to the highest number represented); and finally (iii) the inability of the compilers to reorder computations to avoid data dependencies or their inability to perform algorithm substitution.

One of the design aspects of TSVC is that the benchmark consists of synthetic static loops with well-defined parameters. The authors of the extended work specifically emphasized the following:

> All arrays are [...] aligned and contain the restrict attribute and alignment assertions. One intention was to provide the compiler with as much information as possible. We believe that in many cases the restrict attribute and the alignment assertion could be automatically inserted by the compiler but we have not studied this issue.

A closer inspection of the loops confirms this statement such that a generous amount of information is provided to all loops, such as values of the loop bounds or values in the conditional expressions. In terms of assessing the capabilities of compilers, what is considered as a trivial piece of information in the case of TSVC may be, in fact, a very effective piece of information for triggering and enabling a number of follow-on vectorization opportunities. These are well evidenced by several techniques to improve the achieved vectorization when the static information is suboptimal. For instance, Eichenberger et al. [11] propose vectorization techniques when the alignment is not optimal, and Moll and Hack [21] propose a partial control-flow linearization method that improves the if-conversion vectorization technique. Other approaches, such as presented by Saito et al. [26], explore syntax extensions to support explicit vector programming.

Despite its limitations, the synthetic TSVC is still widely used for evaluating the capabilities of modern auto-vectorizing compilers and is, for instance, included in the LLVM benchmarking suite [18]. There has been more recent analysis using TSVC. For instance, Moldavanova and Kurnosov [20] use TSVC for quantifying the performance of the first-generation Intel Xeon Phi co-processor.

Similar to TSVC, Amiri et al. [2] created a set of benchmarks that use common kernels ranging from simple matrix-matrix multiplications to more complex 2D discrete wavelet transform, opposed to completely synthetic loops. Although the PARSEC suite [3] was created for system-level performance evaluation, to a certain extent it can also be used for evaluating auto-vectorizing compilers. Yazdanpanah [33] analyzes the auto-vectorization limitations within the PARSEC benchmark suite, particularly to determine why only a few of the loops were vectorized. Cebrian et al. [7] provide the ParVec benchmark suite, directly deriving it from the PARSEC suite. Similarly, Zhao et al. [34] analyze the vectorization performance of the well-known NPB and SPEC CPU2006 benchmarks with various SIMD extensions. However, most of these approaches are based on fixed implementations with an associated performance analysis and do not measure the performance variation when altering the implementation provided.

To get a better approximation of production scenarios, some computing domains have developed their own proxy applications. For instance, several signal processing and multimedia-specific application benchmarks have been created to assess the auto-vectorization performance of compilers. Ren et al. [23], Fritts et al. [14], and Alvanos and Trancoso [1] all compare the performance of auto-vectorizers against hand-vectorized counterparts using a set of multimedia applications. All of these studies identify and report a number of missed opportunities for vectorization. Additionally, there are several proxy applications to aid the development of future systems and programming models such as the application collection from the ECP project [24].

There is a body of work that indirectly assesses the vectorizing limitations of compilers by manually applying SIMD transformations. For instance, Cebrian et al. [6] share a concern that is similar to ours when evaluating real-world applications. However, they chose to manually implement any necessary SIMD operations instead of benchmarking or relying on auto-vectorization capabilities. Another approach that is similar to our macrogenerated benchmarking is discussed in Satish et al. [27]. Their approach starts off from a basic C implementation and applies multiple SIMD optimizations.

Another approach for automatically generating benchmarks is discussed in Deshpande et al. [8], however, their focus is not in evaluating the vectorization capabilities of compilers. Skadron et al. [29] highlight a number of practical issues in the current generation of benchmarking infrastructures, and Breughe and Eeckhout [4] propose a method for understanding the impact that different input parameters have on the overall performance. Gong et al. [15] study the variations of compiler performance to different loop mutations, and in doing so they demonstrate the instability of compilers to consistently provide optimal results when the implementation is slightly different. However, they do not consider multiple levels of information provided or withdrawn at compile time.

More recently, Doerfert et al. [9] presented an automated tool to measure the effect that missing static information has on a program. However, their approach is based on incrementally including certain program annotations into the code, and these are often compiler specific. Their implementation is around Clang and the LLVM compiler infrastructure, and they conclude that the static information can significantly improve the performance but that the compiler often fails to leverage the information.

3 OUR APPROACH

3.1 Method for Supplying or Withdrawing Information

Our approach centers around the idea of supplying or withdrawing static information at compile time to understand the effect that a piece of information has on the vectorization performance. To realize this, we need to parameterize the amount of information that is exposed to the compiler and be able to objectively supply or withdraw the information in a controlled manner.

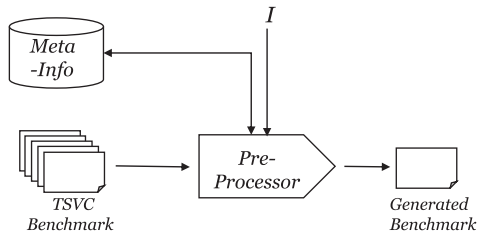


Fig. 2. Architecture for withdrawing information, I, from benchmarks.

To achieve this, we have built a C-preprocessor infrastructure that expands multiple macros representing different classes of static information, as discussed in the next section. The overall evaluation relies on two aspects: (i) specific meta-information defining the class of information that is withdrawn from the compiler and (ii) a set of benchmark loops derived from TSVC produced by Maleki et al.The evaluation process has control over which class of information is presented or withdrawn. Based on this control information, the preprocessor expands the macros using TSVC as a template to generate a set of benchmarks. The generated benchmarks, unlike TSVC, will carry only a defined piece of information to the compiler being evaluated. In other words, we control the information that flows into the compiler that aids the vectorization process. Figure 2 shows the overall architecture of the system we designed for this purpose.

In Figure 2, I is the information we would like to provide, and meta-info is the database of benchmarks, links TSVC, and templates. The generated benchmarks are then compiled using the compiler to be evaluated with appropriate compiler flags.

3.2 Classification of Effective Information

Here we provide a classification of information purely based on the types of contextual information that could be effective in vectorization. These are the classes of information that are withdrawn or supplied at compile time, including the following:

(1) Loop Bounds
(2) Parameters in Array Indices and Offsets
(3) Parameters in Conditional evaluations
(4) Array attributes.

We discuss each of these classes with a supporting code example. The code listings are provided for explanatory purposes and do not come from any of the benchmark suites or applications discussed in this article:

(1) ***Loop Bounds*:** Although loop nests can often be vectorized without full-fledged information on loop bounds, compilers can take advantage of additional information to decide which loops to vectorize and what vectorization strategy to use. Consider the example shown in Listing 1, consisting of two nested loops with access to the arrays A and B with different access patterns to each of the arrays.

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
        sum += A[j][i] + B[i][j];
    }
}
```

Listing 1. Example with Loop Bounds Parameters.

Vectorizing the loop over the *i*-index space as opposed to the *j*-index space could lead to significant performance implications. To decide the optimal candidate loop for vectorization, the exact values of the symbolic variables are needed. These include the values of M and N, as well as the length of vector registers of the target system. If M is smaller than the vector length, vectorizing the loop over the *j*-index could be suboptimal. However, if M is very large, the `gather` operations on the accesses to array A will be expensive and other additional costs related to the memory subsystem could be incurred. However, the contrary is true if the size of M and N is swapped. Making decisions at compile time requires not only a cost model but also information on the values or range of M and N.

(2) ***Parameters in Array Indices and Offsets*:** Understanding the access patterns of the arrays or other data structures inside loop nests is the basis for performing robust dependency analysis, which is often a precursor to vectorizing transformations. Consider the example in Listing 2, consisting of two nested loops with an update operation on the array A.

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
        A[i * DIM + j] = A[(i - 1) * DIM + j] + B[j + Offset(j)]
    }
}
```

Listing 2. Example with Parameters in the Array Indices.

Here, if DIM is smaller than M and smaller than the vector length, there is no guarantee that a single vector operation will respect the same read/write ordering of array A that the scalar version has. This issue is known as loop-carried dependency. In this case, depending on the compiler cost model, it may decide against auto-vectorization of the loop or it may provide the necessary runtime guarantees or constrained implementation. Additionally, there are performance improvements that compilers can apply with known indexing patterns. In this specific case, if `Offset(j)` is constant, the access to array B becomes strided in memory, and if the constant assumes a unit value, it may become contiguous in memory. In both cases, the compiler may be able to generate cheaper load operations. However, if `Offset(j)` is a nonconstant value (or if the value is unknown), the vectorization has to be implemented using gather operations.

(3) ***Parameters in Conditional Evaluations*:** The ability to resolve or predict the branching behavior of a given code snippet is another characteristic that compilers often rely on to vectorize a given part of a code. Masking and if-conversion transformations [12] are optimizations often applied to help the compiler vectorization capabilities. However, the ability to vectorize a code may still be impeded by path divergences resulting from conditional evaluations where not enough information can be inferred. Consider the example in Listing 3, consisting of a single loop with a conditional update on array A.

```
for (int i = 1; i <= N; i++) {
    if (B[i] != Constant) {
        A[i] += B[i] * C[i];
    }
}
```

Listing 3. Example with Parameters in the Condition Evaluation.

Here, the left-side operand of the conditional evaluation changes every iteration of the loop. If the `Constant` has a runtime value of zero, the compiler can semantically remove the `if` condition (the expression $B[i] * C[i]$ will also evaluate to zero, and zero is the identity operator for the aggregation), resulting in a more efficient vectorized code, as no masking operations are needed. Idiomatic `if` removal optimizations are also beneficial on loops that contain multiple exit conditions, as compilers just vectorize certain patterns of loops with multiple exits.

(4) ***Array Attributes (e.g., Aliasing and Alignment)***: Modern vector units are sensitive to the alignment of data [11]. Maleki et al. [19] discussed this issue when implementing TSVC but ultimately decided to always provide the alignment and aliasing information with the tests. Consider the example in Listing 4, consisting of a single loop that accesses three different arrays.

```
float* restrict A __attribute__((aligned(64)));
float* restrict B __attribute__((aligned(64)));
float* restrict C __attribute__((aligned(64)));
...
A = __builtin_assume_aligned(A, 64);
B = __builtin_assume_aligned(B, 64);
C = __builtin_assume_aligned(C, 64);
...
for (int i = 1; i <= N; i++) {
    A[i] = B[i] + C[i];
}
```

Listing 4. Example with Array Attributes.

Depending on the compiler visibility of the array declarations in the context of the loop, it may or may not be able to infer the aliasing and alignment attributes. The built-in `__assume_aligned` statement can help the compiler in assigning the correct attribute values. However, this is rarely used, and such built-ins are compiler specific.

To generate our modified test suite, we use the preprocessor infrastructure described in Section 3.1 to supply (or withdraw) the desired contextual information into templates of the TSVC benchmark suite. Not every information class is expected to be present on every TSVC test. In such cases, the code generated by the preprocessor infrastructure will remain unmodified. The test-information coverage across 138 TSVC tests is as follows:

- Information classes (1) and (4) are present across all 138 tests.
- Information class (2) is present across 67 of these tests.
- Information class (3) is present across 29 of these tests.

3.3 Quantifying Vectorization Efficiency

The metric used to assess the efficacy of auto-vectorizing compilers is the vector efficiency η, defined as the ratio between the elapsed time E_t to execute a specific test t with and without auto-vectorization. Thus,

$$\eta = \frac{E_t^0}{E_t^{\hat{v}}}, \tag{1}$$

where $E_t^{\hat{v}}$ and E_t^0 are the vectorized and nonvectorized execution times for the test t, respectively. Although runtimes with and without the vectorization of benchmarks provide a measure of

success, comparing runtime performance across a combinations of tests (with specific categorizations), platforms, and compilers leads to a high-dimensional space, resulting in a large amount of data, which is further increased by the notion of withdrawing or supplying different classes of information at compile time. For this reason, the results are presented with statistical measurements on multiple data aggregations from the benchmark values. We define the following set of parameters for the data aggregation:

(1) T is a category of similarly structured tests following the categorization provided in previous TSVC implementations. Our analysis uses the LLVM benchmark suite [18] categories, which are linear dependence, indirect addressing, equivalencing, loop rerolling, packing, searching, recurrences, reductions, crossing thresholds, expansion, node splitting, loop restructuring, statement reordering, symbolics, control flow, global data flow, and induction variable propagation.
(2) i is a class of information that is withdrawn from the compiler (by default, all other information classes are exposed to the compiler). I is the set that includes the four classes presented in Section 3.2 plus an instance with all four information classes withdrawn at the same time. None of the given classes is a direct representation of the original TSVC benchmark. This is because the information classes are not consistently present across the test suite.
(3) a is an architecture on which the evaluation is being carried out, and A is the set containing all evaluated architectures.
(4) c is a compiler on which the evaluation is being carried out, and C is the set of all evaluated compilers.

With these parameters being defined, let $E_{t,c,a,i}$ be the elapsed time of a specific test t when the compiler c and the architecture a is used, with information of class i being exposed at compile time. The vectorization efficiency under these circumstances is expressed as follows:

$$\eta(t,c,a,i) = \frac{E^{0}_{t,c,a,i}}{E^{\hat{v}}_{t,c,a,i}}, \tag{2}$$

For a given category T, an average vector efficiency, η_T, can be given using the geometric mean from all $t \in T$. Furthermore, the overall vectorization efficiency η_{Total} is given by a geometric mean across all categories. Using the mean of each category keeps a balanced representation of all categories in the final score regardless of the number of tests in each category.

$$\eta_T(c,a,i) = \left(\prod_{t\in T} \eta(t,c,a,i)\right)^{\frac{1}{\text{size(T)}}} \tag{3}$$

$$\eta_{\text{Total}}(c,a,i) = \left(\prod_{T\in\text{Categories}} \eta_T(c,a,i)\right)^{\frac{1}{\text{size(Categories)}}}. \tag{4}$$

3.4 Visualizing Vectorization Efficiency

As described in the previous section, it is possible to reduce the dimensionality of the results by performing statistical analysis and data aggregation. However, the number of information classes and the number of compiler-architecture combinations cannot be reduced any further. In the our case, there are 17 loop categories and 16 compiler-architecture pairs. As such, any attempt to compare the vectorization efficiency of the compilers across all of the loop categories becomes visually challenging. For this reason, we developed a new plot type, which we refer to as the vector efficiency spectrum. This chart provides an intuitive representation, not only for comparing the

Table 1. Systems Used for Evaluation

	ScafellPike SKL	ScafellPike KNL	Panther
Processor	Intel Xeon Skylake	Intel Xeon Phi Knight's Landing	IBM Power8
Model	Gold 6142	7210	8335-GTA
Architecture	x86 (64 bit)	x86 (64 bit)	pcc64le
Number of Cores	2×16	64	2×8
Core Frequency	2.6GHz	1.3GHz	3.86GHz
Vector ISA	AVX-2 and AVX-512	AVX-512 (KNL)	VMX/AltiVec
Vector Length	256 and 512 bits	512 bits	128 bits
Memory	128GB	109GB	512GB
L1 D Cache	32K	32K	64K
L2 Cache	1,024K	1,024K	512K
L3 Cache	22,528K	—	8,192K
OS	Linux 3.10	Linux 3.10	Linux 3.10
Compilers Used	GCC 8.1.0 Clang 6.0.0 Intel 2018u4 PGI 2018.4	GCC 8.1.0 Clang 6.0.0 Intel 2018u4 PGI 2018.4	GCC 8.1.0 Clang 6.0.0 IBM XLC 13.5 PGI 2018.4

vectorization efficiency of a given compiler in the absence of information but also for offering an approach for assessing their relative efficiencies.

In this plot, we spatially map the efficiency values $\eta_{\text{Total}}(c, a, i)$ for different architecture-compiler pairs as horizontal lines on two graphs placed next to each other. On the left spectrum, we show the vectorization efficiency of different architecture-compiler pair when information for category i is known at compile time. On the right spectrum, we show the vectorization efficiencies for the same architecture-compiler pair when information for category i is withdrawn at compile time. Although the lines have distinct color, due to the large number of lines, comparing them becomes an arduous task. For this reason, we connect the lines representing the same compiler-architecture pair across the spectra and to their entry in the legend.

4 EVALUATION

We tested five different compilers across three representative architectures (covering four different vector ISAs, namely AVX-2, AVX-512 (Skylake), AVX-512 (KNL), and AltiVec) containing different vector processing capabilities. Although there are references to SVE and RISC-V Vector Extensions in the literature, to-date there are no production-ready implementations of these ISAs. As such, they are not included in the evaluation process. We provide the details of the platforms, as well as the compilers used on each platform, in Table 1. We have deliberately included vendor- and platform-specific compilers in our evaluation.

The exact flags used for different compilers are provided in Table 2. The choice of flags are compiler- and platform specific, but they have been chosen with aggressive optimization enabled for each compiler. Additionally, for the AVX-512 (Skylake) platform, we included additional flags that would prioritize the AVX-512 instruction set, wherever permitted. However, the compilers are free to use the AVX-2 instructions as appropriate. In addition to these flags, we included a request for detailed vectorization reports to be generated, whenever they are available, along with the assembly code. This files assissted us in the analysis of the results. In all of the cases, we repeated all scalar and vector tests five times and took the smallest reported value to minimize any unnecessary perturbation, such as the ones from the operating system.

Table 2. Compiler Flags Used in the Evaluation

Compiler	Vectorization Disabled	Vectorization Enabled
GNU	-march=$ISA -O3 -ffast-math -fno-tree-vectorize	-march=$ISA -O3 -ffast-math (-mprefer-vector-width=512 on Skylake-avx512)
Intel	-x$ISA -O3 -fp-model fast=2 -no-vec	-x$ISA -O3 -fp-model fast=2 (-qopt-zmm-usage=high on skylake-avx2)
Clang	-march=$ISA -O3 -ffast-math -fno-vectorize	-march=$ISA -O3 -ffast-math
IBM	-O3 -qhot=novector:fastmath -qnoaltivec -qsimd=noauto	-O3 -qhot=fastmath
PGI	-tp=$ISA -O3 -fast -fastsse -Mnovect	-tp=$ISA -O3 -fast -fastsse

5 RESULTS

In this section, we present the results of our exhaustive evaluation covering the modified TSVC and a selection of application-level proxy kernels. In Section 5.1, we present the results of TSVC when different classes of information are withdrawn at compile time. We show the resulting impact using vectorization efficiency. For easier interpretation of the results, we utilize the vector visualization spectrum charts outlined in Section 3.4. Then, in Section 5.2, we introduce the test categorization of TSVC to provide a closer look into the strengths and weaknesses of modern compiler auto-vectorization capabilities when all of the information classes are provided to the compiler. Finally, to ensure that the approach is relevant for real-world applications, we evaluate the vectorization efficiency of compilers on six application-level proxy kernels. The relevant results are presented in Section 5.3.

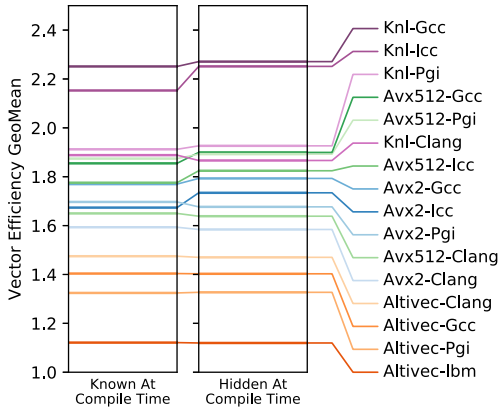
5.1 TSVC with Information Classes on Current Architectures and Compilers

As discussed in Section 3.2, we separated the type of information that is available to compilers into the four classes defined in Section 3.2. We consider each of these information classes in turn, and we measure two scenarios, when the information under consideration is fully supplied to the compiler and when the information is withdrawn at compile time. Information belonging to all other classes remain fully supplied. We show the vectorization efficiency of different compilers for each of the cases in Figure 3. Each vectorization spectrum chart has two parts, the one when with the information supplied at compile time on the left and the second with the information withdrawn on the right.

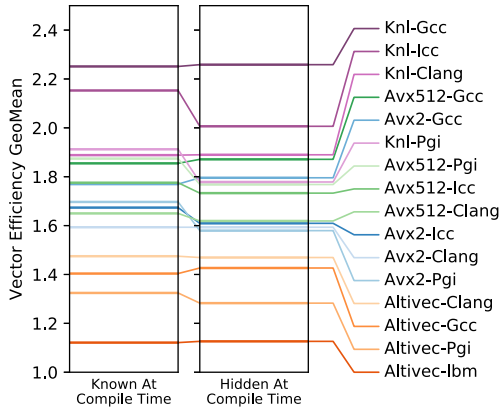
In addition to withdrawing each of the classes independent of each other, we also provide the case where all classes of information are supplied or withdrawn at the same time in Figure 4.

The geometric mean variation between supplying and withdrawing information at compile time is shown in Table 3. The vector efficiency improvements seen in the table as positive values are the same as the gain observed by following a spectrum line from right to left in the associated vector spectrum chart. The negative values are efficiency reductions of having the information supplied at compile time.

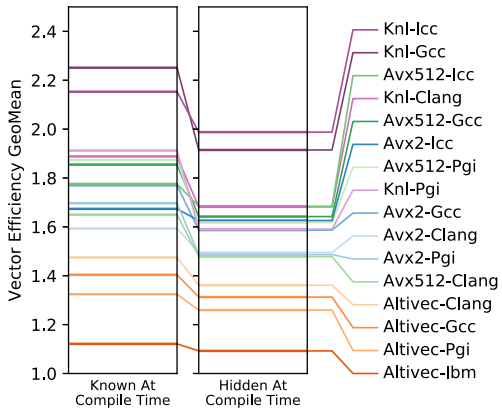
The overall observation here is that exposing information improves the mean vectorization performance of compilers when compared against the case where all classes of information are fully withdrawn. In addition to this general observation, the following case-specific observations can be made:



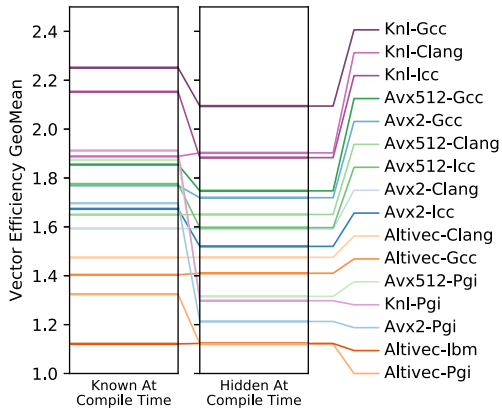
(a) Parameters on Conditional Expressions



(b) Parameters on loop boundaries



(c) Parameters on Array Indices Expressions



(d) Alignment and aliasing arrays attributes

Fig. 3. TSVC vector efficiency spectrums for each information class.

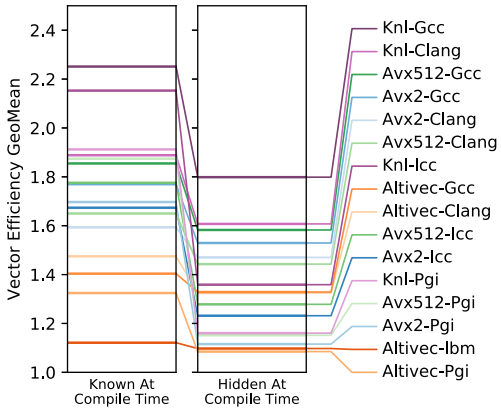


Fig. 4. TSVC vector efficiency spectrum when all classes are exposed or withdrawn simultaneously.

Table 3. Percentage Variation of Vectorization Efficiency (Expressed as Geometric Mean) When Information Is Supplied at Compile Time

Information Class	Platform	Compilers				
		GNU	Intel	Clang	IBM	PGI
Conditional Parameters	AltiVec	0.1	—	0.3	0.1	−0.2
	AVX-2	−1.3	−3.5	0.6	—	1.2
	AVX-512 (Skylake)	−2.4	−2.4	0.7	—	−1.0
	AVX-512 (KNL)	−0.9	−4.4	1.2	—	−0.7
Loop Bounds Parameters	AltiVec	−1.6	—	0.4	−0.5	3.3
	AVX-2	−1.4	4.0	0.0	—	7.4
	AVX-512 (Skylake)	−0.9	2.5	1.9	—	6.0
	AVX-512 (KNL)	−0.3	7.3	−0.1	—	7.5
Index Parameters	AltiVec	6.9	—	8.3	2.6	5.2
	AVX-2	11.5	2.9	6.6	—	14.2
	AVX-512 (Skylake)	12.9	5.4	11.6	—	15.9
	AVX-512 (KNL)	17.6	8.3	12.3	—	20.2
Variable Attributes	AltiVec	−0.5	—	0.0	−0.2	18.2
	AVX-2	2.9	10.1	0.1	—	39.9
	AVX-512 (Skylake)	6.1	11.2	0.0	—	42.4
	AVX-512 (KNL)	7.5	14.3	−0.8	—	47.3
All Parameters	AltiVec	5.7	—	11.3	2.2	22.1
	AVX-2	15.7	35.9	8.4	—	52.1
	AVX-512 (Skylake)	17.2	38.9	14.4	—	62.7
	AVX-512 (KNL)	25.2	58.5	17.5	—	64.8

(1) In almost all of the cases, hiding information from the compilers reduces their effective vectorization efficiency. The most significant exception to this is the case of Conditional Parameters. Withdrawing the conditional parameters yields better performance and improves the average auto-vectorization efficiency across all compilers except the Clang compiler. This counterintuitive auto-vectorization improvement when compilers have less contextual information comes mainly from two distinct issues. First, the Intel compiler does not vectorize the tests *S*261, *S*253, *S*3251, *S*254, and *S*1251 (all from the *Expansion* category) when the conditional parameter values are supplied to the compiler; however, these same tests are vectorized when the information is hidden. Looking at the compiler optimization reports with the supplied information, we found that each of these loops raises a "vector dependence prevents vectorization" remark. A manual inspection of the loops shows that they have no actual dependencies, or, in the cases where they have dependencies, those could have been been easily resolved by reordering the relevant statements in the body of the loop. This issue only appears on the Intel compiler and for this specific loop structure.

Additionally, the GNU and PGI compilers have a negative outlier on test *S*276 on all x86 platforms. Listing 5 shows the source for *S*276, where the *cp_n1* parameter is exposed or withdrawn from the compiler. Looking at the assembly code produced by both compilers, we can see that the GNU compiler generates a scalar branching implementation when all conditional parameters are known at compile time. However, when the conditional expression contains parameters unknown at compile time, the generated code is

significantly different: the compiler generates vector code using the *vcmpltps*, *vmaskmovps*, and *vmovups* instructions. The *vcmpltps* instruction is a SIMD version of the "less than" operation. The *vmaskmovps* and *vmovups* instructions are the masking operation in AVX-2 and AVX-512, respectively. Given that the compiler is capable of generating both AVX-2 and AVX-512 instructions, we believe that it is just a heuristic issue where the compiler chose to go with the worst-performing implementation for the compile-time-known case.

The PGI case is, however, different from these. In both scenarios, PGI produces a similar vectorized implementation using the *blendvps* (SIMD conditional copy) operation. However, when the parameters are known at compile time, it chooses to distribute the value of the conditional with a *vextractf128* and multiple *vpshufd* operations in every loop, whereas it generates a pure vector version when the parameter is unknown. These additional operations add a significant overhead that reduces the vectorization performance. It is also worth noting that PGI does not generate AVX-512 instructions inside this loop, and, in fact, it defaults to the AVX-2 implementation for all x86 platforms.

```
mid = LEN / 2;
...
for (int i = 0; i < LEN; i++) {
    if (i + cp_n1 < mid) {
        a[i] += b[i] * c[i];
    } else {
        a[i] += b[i] * d[i];
    }
}
```

Listing 5. Source Code of TSVC Test *S*276.

(2) Exposing Loop Bounds Parameters provides some moderate improvements with the Intel (2.5% to 7.3%) and PGI (3.3% to 7.5%) compilers. But this has also a marginal overall negative impact (between −0.3% and −1.6%) on the GNU compiler.

When analyzing the regressions on the GNU compiler, we find that tests *S*276 and *S*115 make significant negative contributions to the score with supplied information. For instance, in the AVX-2 platform, the tests are not vectorized when the information is supplied but have a vector efficiency of ×3.9 and ×4.7, respectively, when the loop bounds values are withdrawn.

Performance issues surrounding the test case *S*276 have also been discussed in (1). In this case, the variable *LEN* (Listing 5) represents the loop bound whose value information has been withdrawn. As for loop *S*115, it is worth noting that this benchmark represents a triangular iteration space through a 2D array with an offset (Listing 6). When the parameters for the loop bounds *bp_n*1 and *LEN*2 are known, the compiler is able to ensure the condition $j < i$ and discards the possible data dependencies in the accesses to array *a*. However, with *LEN*2 having a value of 256, the compiler considers that the triangular loop is too small for a vectorized implementation and resorts to a scalar implementation. When the loop bounds parameters values are withdrawn, the compiler uses a multiversioning strategy to resolve possible data dependency issues. With no information about the iteration length, the compiler generates a vectorized version for the no-data-dependency case, which turns out to be more efficient.

```
for (int j = 0; j < LEN2; j++) {
    for (int i = j + bp_n1; i < LEN2; i++) {
        a[i] -= aa[j * LEN2 + i] * a[j];
    }
}
```

Listing 6. Source Code of TSVC Test *S*115.

(3) Exposing Index Parameters provides significant auto-vectorization improvements, especially on the PGI (5.2% to 20.2%), GNU (6.9% to 17.6%), and Intel (6.6% to 12.3%) compilers.
(4) Variable Attributes information also provides significant auto-vectorization improvements for the overall TSVC test, especially, on the Intel (10.1% to 14.3%) and PGI (18.2% to 47.3%) compilers. Only the Clang compiler shows a negligible impact when supplying variable attributes. In fact, this compiler only exploits the *restrict* keyword when it is associated to function arguments [13]. This represents one of the main weaknesses in Clang vectorization, as shown in Figure 3(d); when array attributes are supplied, it has the worst vectorization efficiency, expressed as geometric mean, on each x86 platform, whereas when the information is withdrawn, it has the second best geometric mean on the same platforms.
(5) When all parameters are hidden from the compiler, the auto-vectorization performance decreases substantially in comparison to the vectorization achieved when all contextual information is given. This is true across all compiler-platform pairs. We observe that the IBM, GNU, and Clang compilers are the least sensitive to the information differences, with variation of 2.2% (5.6% to 25.2%) and (8.4% to 17.5%) respectively. Meanwhile, the Intel compiler shows a variation between 15.9% and 58.5% and the PGI compiler being the most sensitive with performance variation from 22.1% to 64.8%.
(6) In general, the geometric means of the auto-vectorization efficiencies are directly correlated with the underlying vector lengths. This is readily observable in the vectorization spectrum charts. For instance, spectral lines for the AVX-512 (KNL and Skylake) platforms achieve better efficiency compared to the AVX-2- and AltiVec-based platforms. Furthermore, when compilers are supplied with additional information, platforms with longer vector lengths show the largest difference in auto-vectorization improvement. For example, the GNU compiler gets 5.7% on AltiVec when all parameters are given at compile time, whereas it improves the overall performance by 25.2% under the same conditions in the AVX-512 (KNL) platform.

5.2 TSVC Loop Categories with Optimistic Information Supplied

In the previous section, we analyzed the impact of information withdrawal on the modified TSVC using the aggregated vectorization efficiency metric. However, TSVC can also be subdivided into 17 categories of different loops structures. Although our macro-based testing infrastructure is able to produce the same analysis separately for each of the categories, the space limitations do not allow us to discuss all of them here in detail. We therefore limit our analysis of the TSVC categories to the case where the maximum amount of information is supplied consistently. This case is useful to determine the best auto-vectorization performance achievable of modern compilers and architectures, but as we argue in this article, we believe that there are relevant additional insights that can be obtained with the information withdrawal analysis.

Figures 5 and 6 present the TSVC results with contextual information always supplied to each of the compilers and vector platforms. Again, the metric used in these figures is the geometric mean

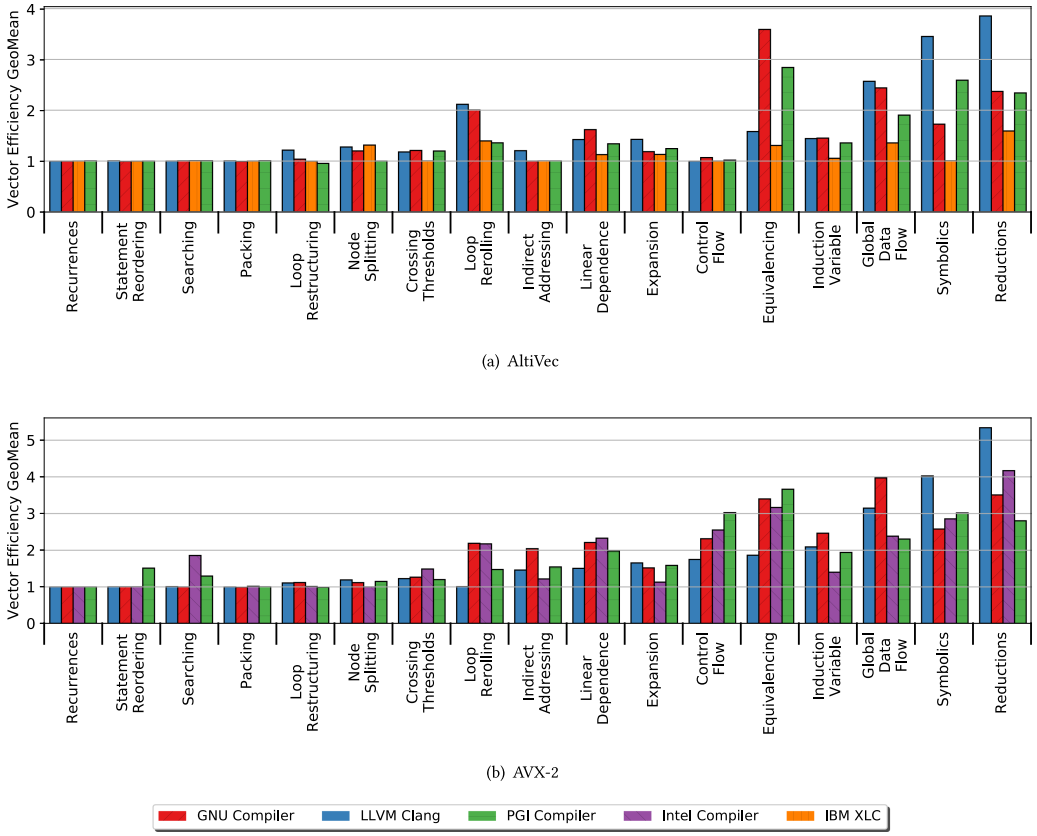


Fig. 5. Vectorization efficiency of each TSVC category when all information is supplied to the compilers (AltiVec and AVX-2).

of the vector efficiencies. This metric provides insight about the auto-vectorization capabilities of the compilers, but it should not be used as a metric to asses which compiler gets the absolute best performance, as the scalar effects are being purposely normalized. Several observations can be made from these results:

(1) All compilers, regardless of the target vectorization ISA, show similar auto-vectorization behavior: the loop categories that cannot be vectorized by the Clang compiler on the AVX-2 and AVX-512 (Skylake and KNL) ISAs are the same. However, the differences in performance across architectures (for a given compiler) and compilers (for a given architectural platform) are clearly noticeable.

(2) Platforms with longer vector lengths deliver better vectorization performance. This is as expected, with longer registers being able to deliver (theoretically) better performance. However, this makes the performance gap between the vectorized and nonvectorized versions more significant. When directly comparing the two AVX-512 platforms, it appears that the KNL platform offers slightly better auto-vectorization performance than the Skylake platform. This is true in most of the cases. We believe that this is due to significant improvement in the Intel compiler's capability to perform *Reduction* operations. However, this does not take into account the scalar performance of each platform and thus is not a measure of the overall best performance.

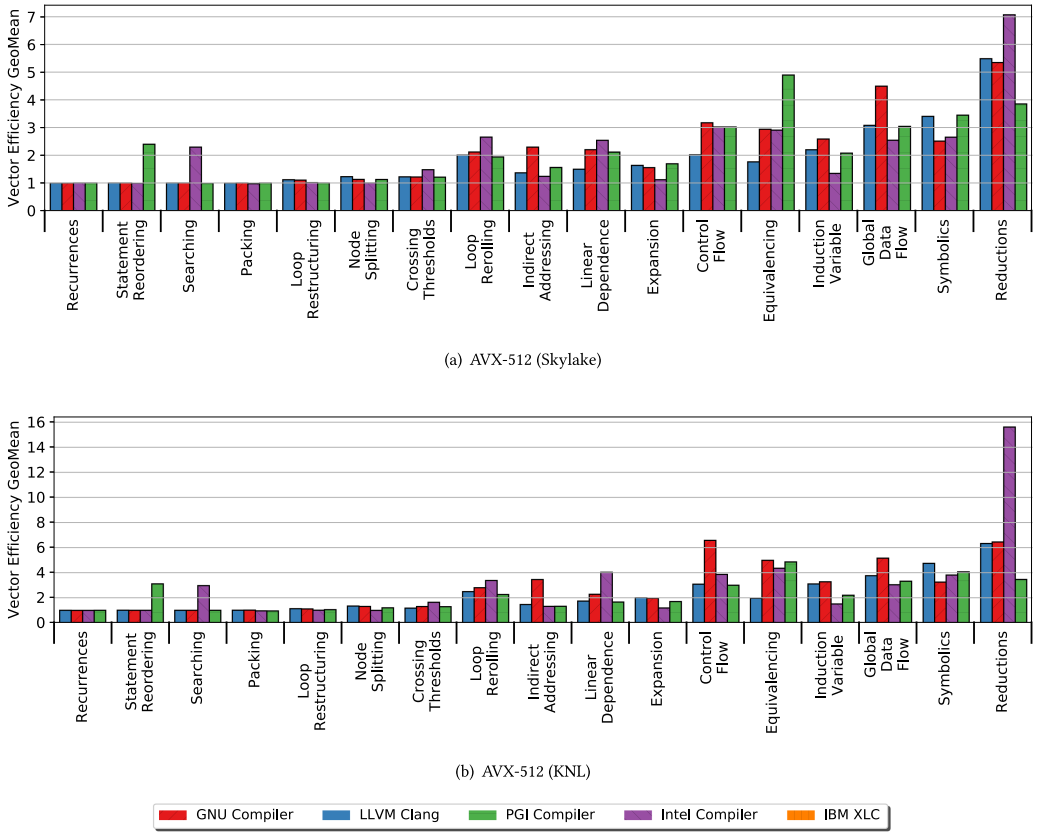


Fig. 6. Vectorization efficiency of each TSVC category when all information is supplied to the compilers (AVX-512 (KNL) and AVX-512 (Skylake)).

(3) Regarding the multiple TSVC categories:
   (a) The *Recurrences* and *Packing* categories were not vectorized by any of the compilers regardless of the underlying platform.
   (b) The *Statement Reordering* and *Searching* categories also proved challenging for most of the compilers. *Statement Reordering* was only vectorized by the PGI compiler on the three x86 platforms. The *Searching* category was only vectorized by the Intel compiler on x86 platforms and by the PGI compiler exclusively on AVX-2.

   For instance, Listing 7 shows the source for test *S*331 from the *Searching* category. In this example (and in all other tests), Intel was the only compiler that generated the AVX-512 *vpcmpud* operation (integer SIMD comparison that creates a mask as result). This has led the Intel compiler to be the only one to vectorize this particular test in the two AVX-512 platforms.

```
j = -1;
for (int i = 0; i < LEN; i++)
    if (a[i] < cp_n0)
            j = i;
```

Listing 7. Source Code of TSVC Test *S*331.

(c) The *Loop Restructuring*, *Node Splitting*, and *Crossing Thresholds* categories just get marginal vectorization efficiency (< ×1.3) scores.
(d) The *Loop Rerolling*, *Indirect Addressing*, *Linear Dependence*, *Expansion*, and *Induction Variables* categories have moderate auto-vectorization efficiency (around 25% of the theoretical peak performance of the platform). The Intel compiler struggled to vectorize *Indirect Addressing* and *Expansion*. However, it offered the best results for *Loop Rerolling* and *Linear Dependence*.
(e) The *Control Flow* category is not vectorized on the AltiVec platform, but compilers manage to get up to 40% of the theoretical peak vector performance on x86 platforms.
(f) *Global Data Flow* and *Symbolics* categories show good auto-vectorization results across all platforms and compilers. The GNU compiler generally provided the best vectorization for the former, and the Clang compiler offered the best vectorization for the latter.
(g) Nearly all compilers excelled on the *Reductions* category regardless of the platform and offered the best auto-vectorization efficiency. A closer examination of individual tests showed that for some tests, even the theoretical maximum vector efficiency was exceeded. An examination of TSVC test *S*311 (Listing 8), which represents a canonical addition reduction example, shows that the vectorized version uses the appropriate set of instructions, vector registers, and unrolling factors. For instance, for the ICC-KNL compiler-platform case, the nonvectorized assembly (Listing 9) heavily relied on the `vaddss` instructions, `xmm` registers, and unrolling factor of 2, whereas for the vectorized assembly (Listing 10), `vaddps` instructions, `zmm` registers, and the unrolling factor of 8 were used.

When executed on the AVX-512 (KNL) platform, the vectorized version is 30× faster than the scalar version. The theoretical performance difference from the 512-bit `vaddps` vectorization compared to the scalar `vaddss` instruction is 16×. We believe that the remaining speedup could have come from achieving better instruction-level parallelism on each addition operation, as pipelined arithmetic instructions have a throughput that is bigger than one instruction per clock cycle. However, the scalar version throughput could be limited by an insufficient unrolling factor or by limitations stemming from the underlying microarchitectural aspects. Further analysis is required to confirm this hypothesis.

5.3 Application Kernels

To verify that the proposed approach has a utility beyond synthetic benchmarks, we evaluated the impact that supplying and withdrawing the same static information has on six proxy kernels. Although they are not full-fledged applications, they represent typical computation units found in diverse numerical applications. Additional evidence that supplying the additional information at compile time has impact on more complex application can be demonstrated by multiple applications that recorded vector performance improvements by exploiting such information. For instance, Henderson et al. [16] and Siso et al. [28] present two scientific applications that improved the vector performance when length of the loop bounds was provided statically.

The choice of these six specific proxy kernels was motivated by the benchmarks used to evaluate the OpenMP SIMD pragmas [17] and the ISPC programming language [22]. The implementations are closely based on the ISPC C baseline implementations of these same kernels. However, not all of these kernels have instances of each of the information classes described in Section 3.2, and therefore this analysis is limited to four configurations (C_1, C_2, C_3, and C_4), representing the following:

```
sum = (float)0.;
for (int i = 0; i < LEN; i++) {
    sum += a[i];
}
```

Listing 8. Source Code of TSVC Test *S*311.

```
B1.24:
    lea     (%rax,%rax), %edx
    addl    $1, %eax
    vaddss  (%rdi,%rdx,4), %xmm1, %xmm1
    vaddss  4(%rdi,%rdx,4), %xmm0, %xmm0
    cmpl    $16000, %eax
    jb      ..B1.24

    vaddss  %xmm0, %xmm1, %xmm0
```

Listing 9. Assembly Code from the Scalar *S*311 Test with ICC on the KNL Platform.

```
..B1.30:
    vaddps  (%rdi,%rcx,4), %zmm0, %zmm0
    vaddps  64(%rdi,%rcx,4), %zmm22, %zmm22
    vaddps  128(%rdi,%rcx,4), %zmm21, %zmm21
    vaddps  192(%rdi,%rcx,4), %zmm20, %zmm20
    vaddps  256(%rdi,%rcx,4), %zmm19, %zmm19
    vaddps  320(%rdi,%rcx,4), %zmm18, %zmm18
    vaddps  384(%rdi,%rcx,4), %zmm17, %zmm17
    vaddps  448(%rdi,%rcx,4), %zmm16, %zmm16
    addq    $128, %rcx
    cmpq    %rax, %rcx
    jb      ..B1.30

    vaddps  %zmm22, %zmm0, %zmm22
    vaddps  %zmm20, %zmm21, %zmm20
    vaddps  %zmm18, %zmm19, %zmm18
    vaddps  %zmm16, %zmm17, %zmm16
    vaddps  %zmm20, %zmm22, %zmm17
    vaddps  %zmm16, %zmm18, %zmm19
    vaddps  %zmm19, %zmm17, %zmm0
```

Listing 10. Assembly Code from the Auto-Vectorized *S*311 Test with ICC on the KNL Platform.

(1) Configuration C_1: vectorization: disabled, information: withdrawn
(2) Configuration C_3: vectorization: enabled, information: withdrawn
(3) Configuration C_2: vectorization: disabled, information: supplied
(4) Configuration C_4: vectorization: enabled, information: supplied.

Figure 7 shows the results obtained for each kernel, configuration, compiler, and architecture. It uses C_1 (Vectorization: Disabled, Information: Withdrawn) in each compiler platform as a baseline to compute the speedup of the four different configurations. Each of the application kernels shows distinct behaviors when additional information is provided. It is also worth noting that each kernel's particular implementation and the information that is supplied or withdrawn has an impact to the obtained results. For this reason, we provide a brief description for each of the proxy kernels and the aspects surrounding their implementation, together with the main performance observations from each compiler:

- ***Binomial Options*:** This is an iterative pricing model often used in finance workloads. The implementation can supply or withdraw the information pertaining to the number of timesteps in the iterative procedure (loop bounds) and the array attributes ensuring the non-aliasing of the arrays. In this kernel, the GCC and ICC compilers already vectorize the code without any additional information provided at compile time. By contrast, the PGI compiler auto-vectorizes the code only when the extra information is presented. The Clang and IBM compilers were not able to auto-vectorize this kernel even when additional information is presented.
- ***Black-Scholes*:** This is another pricing model used in financial workloads. The implementation computes the Black-Scholes formula for each option. In this case, the nonaliasing attribute in multiple arrays is the only additional information that can be supplied to or withdrawn from the compiler. In this kernel, the Clang compiler was able to auto-vectorize the algorithm without the nonaliasing attributes. The ICC and PGI compilers were able to vectorize the code when additional information was provided and obtained much higher vectorization performance in the AVX-2 and AVX-512 (both Skylake and KNL) platforms.

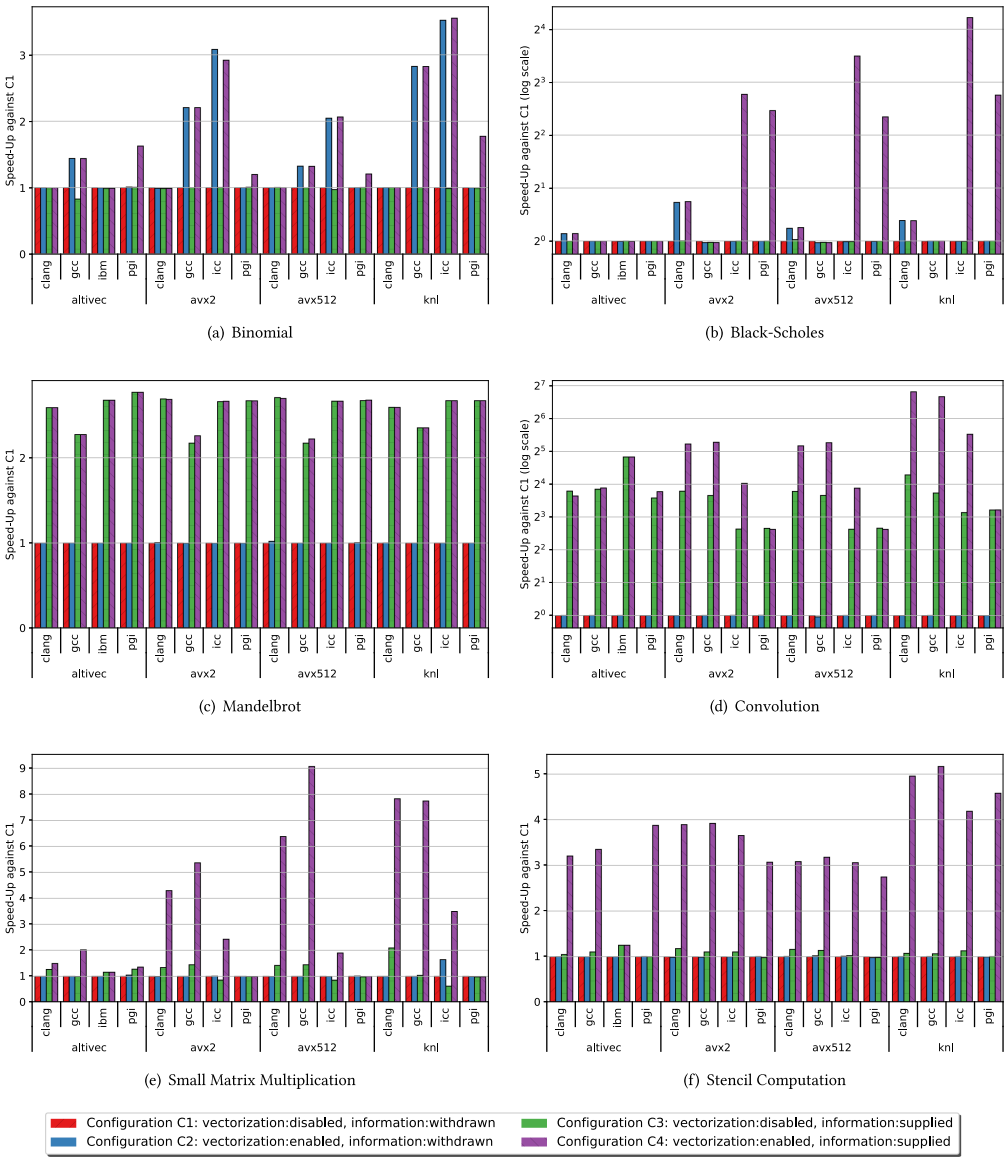


Fig. 7. Compiler comparison of macrobenchmark kernels with multiple levels of information.

- ***Mandelbrot*:** This kernel computes the Mandelbrot set for a complex function $f(z) = z^2 + c$ for nondiverging complex number z, $z \in \mathbf{Z}$ and $z > 0$. The algorithm is performed over a 2D image, with the function $f(z)$ computed for each pixel. The information withdrawn includes the loop bounds and a number of arithmetic parameters that are part of the computation. None of the compilers were able to auto-vectorize the Mandelbrot kernel, irrespective of the amount of information provided at compile time. However, all compilers managed to improve their scalar performance by a factor of at least 2 when additional information was provided.
- ***Convolution*:** This kernel applies a convolution mask to a 2D image by sliding the mask in a 2D space. The information supplied to or withdrawn from the compiler includes the

nonaliasing parameters, the size of the image, the size of the mask, and the values for the mask. All of these parameters specify the bounds of different nested loops in the algorithm. In the kernel, all compilers managed to obtain significant scalar speedups when the information was supplied. Additionally, the Clang, GNU, and Intel compilers offered additional auto-vectorization improvements on top of the performance gained by the scalar optimizations. This auto-vectorization is not observed on the same compilers when the information is withdrawn.

- ***Small Matrix Multiplication*:** This kernel implements a straightforward *ijk* loop matrix multiplication. Here we used the matrix dimensions of size 32, which is small enough to minimize the memory constrains but sufficient enough to warrant reasonable amount of vectorization. The extra information that could be given to the compiler are the matrix sizes, which define the multiple loop bounds. In this kernel, the GNU and Clang compilers were able to obtain some scalar performance advantage when supplied with compile-time information. The Intel compiler has a small scalar performance decline. However, when the auto-vectorization was enabled, these three compilers achieved considerable vectorization performance using the additional information.
- ***Stencil Computation*:** This is a 3D stencil kernel that iteratively updates a voxel value by computing the average of six neighboring voxels. Stencils are often the basis for mesh or grid-based computations, such as finite-element or finite-difference methods. The only useful information hidden or supplied at compile time for this particular implementation is the aliasing attributes of the arrays. All compilers were able to auto-vectorize the stencil computation without the nonaliasing information. However, the vectorization performance improved when nonaliasing information was made available, except for the IBM compiler. Interestingly, the AVX-2 vectorization performance was better than the performance for the AVX-512 case in the same platform.

6 CONCLUSION

In this article, we have proposed a new methodology for evaluating the auto-vectorizing capability of compilers. Our proposed methodology relies on objectively supplying or withdrawing useful information at compile time. This approach is orthogonal to the one adopted by the original and extended versions of the TSVC benchmarks [5, 19], where they have a fixed, and optimistic, amount of contextual information available for the compiler to use at compile time. The new proposed approach provides a complementary and more detailed mechanism to test modern vectorizing compilers, as it is not only able to replicate the TSVC approach but also evaluates a range of scenarios with varying degrees of supplied information.

We applied the method to TSVC and multiple application-level proxy kernels, and performed an exhaustive evaluation on four vector architectures (AltiVec, AVX-2, AVX-512 (Skylake), AVX-512 (KNL)) using five compilers:(GNU, Clang, PGI, Intel, and IBM). We also devised a new visualization mechanism, namely the vector efficiency spectrum, to present, interpret, and understand these benchmarking results.

The first observation is that in modern compilers, the resulting performance from auto-vectorization optimization is still far from the architectural peak performance. The reasons for the suboptimal performance can either be ascribed to dependencies that prevent vectorization or to the inability of the compilers to apply the targeted optimization. A second observation is that the amount of useful information presented to the compiler at compile time is crucial in determining the performance of resulting vectorized code. The exact significance of each of the information classes on the final performance varies across the different compilers and architectures tested. In decreasing order of importance, these classes are the Index Parameters, followed by the Array

Variable Attributes and then the Loop Bounds Parameters. Moreover, we found that supplying the values in the Conditional Parameters class leads to an auto-vectorization reduction by all compilers tested. Furthermore, when more than one class of information is withdrawn, these effects are compounded, particularly with some application-level kernels.

There are two approaches to evaluate compiler auto-vectorization capabilities of compilers: one is to assume that compilers have access to all information, and the other is being selective in the amount of available information. The main advantage of the latter approach, which has been proposed in this article, is that it is able to consider a broader set of scenarios found in scientific applications. Therefore, the proposed approach provides a new set of evaluation tools that can help compiler developers identify weaknesses in current compiler implementations and can assist application developers to understand and plan which information to statically include or withdraw in their code to maximize the performance of vectorizing compilers.

REFERENCES

[1] M. Alvanos and P. Trancoso. 2016. Video SIMDBench: Benchmarking the compiler vectorization for multimedia applications. In *Proceedings of the 2016 Euromicro Conference on Digital System Design (DSD'16)*. 168–175.

[2] Hossein Amiri, Asadollah Shahbahrami, Angela Pohl, and Ben Juurlink. 2018. Performance evaluation of implicit and explicit SIMDization. *Microprocessors and Microsystems* 63 (2018), 158–168.

[3] Christian Bienia and Kai Li. 2009. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking, and Simulation.*

[4] Maximilien B. Breughe and Lieven Eeckhout. 2013. Selecting representative benchmark inputs for exploring microprocessor design spaces. *ACM Transactions on Architecture and Code Optimization* 10, 4 (Dec. 2013), Article 37, 24 pages.

[5] D. Callahan, J. Dongarra, and D. Levine. 1988. Vectorizing compilers: A test suite and results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing (Supercomputing'88)*. IEEE, Los Alamitos, CA, 98–105.

[6] Juan M. Cebrian, Magnus Jahre, and Lasse Natvig. 2014. Optimized hardware for suboptimal software: The case for SIMD-aware benchmarks. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. 66–75.

[7] Juan M. Cebrian, Magnus Jahre, and Lasse Natvig. 2015. ParVec: Vectorizing the PARSEC benchmark suite. *Computing* 97, 11 (2015), 1077–1100.

[8] Vivek Deshpande, Xing Wu, and Frank Mueller. 2012. Auto-generation of communication benchmark traces. *SIGMETRICS Performance Evaluation Review* 40, 2 (Oct. 2012), 99–105.

[9] Johannes Doerfert, Brian Homerding, and Hal Finkel. 2019. Performance exploration through optimistic static program annotations. In *High Performance Computing*. Springer, 247–268.

[10] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. 2013. The LINPACK benchmark: Past, present and future. *Concurrency and Computation: Practice and Experience* 15, 9 (2013), 803–820.

[11] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, New York, NY, 82–93.

[12] Jesse Z. Fang. 1997. Compiler algorithms on if-conversion, speculative predicates assignment and predicated code optimizations. In *Languages and Compilers for Parallel Computing*, D. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (Eds.). Springer, Berlin, Germany, 135–153.

[13] Hal Finkel. 2017. Restrict-qualified pointers in LLVM. Retrieved September 4, 2019 from https://llvm.org/devmtg/2017-02-04/Restrict-Qualified-Pointers-in-LLVM.pdf.

[14] Jason E. Fritts, Frederick W. Steiling, Joseph A. Tucek, and Wayne Wolf. 2009. MediaBench II video: Expediting the next generation of video systems research. *Microprocessors and Microsystems* 33, 4 (June 2009), 301–318.

[15] Zhangxiaowen Gong, Alexandru Nicolau, Josep Torrellas, Zhi Chen, Justin Szaday, David Wong, Zehra Sura, et al. 2018. An empirical study of the effect of source-level loop transformations on compiler stability. *Proceedings of the ACM on Programming Languages* 2 (OOPSLA), 1–29. DOI: https://doi.org/10.1145/3276496

[16] Tom Henderson, Jhon Michalakes, Indraneil Gokhale, and Ashish Jha. 2015. Numerical weather prediction optimization. In *High Performance Parallelism Pearls Volume Two: Multicore and Many-Core Programming Approaches*. MKF Publishers, 7–23.

[17] 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 *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World (IWOMP'12)*. 59–72.

[18] LLVM. 2018. LLVM Test-Suite: TSVC. Retrieved September 4, 2019 from http://llvm.org/svn/llvm-project/test-suite/trunk/MultiSource/Benchmarks/TSVC/.

[19] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An evaluation of vectorizing compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE, Los Alamitos, CA, 372–382.

[20] Olga V. Moldovanova and Mikhail G. Kurnosov. 2017. Auto-vectorization of loops on Intel 64 and Intel Xeon Phi: Analysis and evaluation. In *Parallel Computing Technologies*, V. Malyshkin (Ed.). Springer, Cham, Switzerland, 143–150.

[21] Simon Moll and Sebastian Hack. 2018. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, 543–556. DOI: https://doi.org/10.1145/3192366.3192413

[22] M. Pharr and W. R. Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *Proceedings of the 2012 Innovative Parallel Computing Conference (InPar'12)*. 1–13.

[23] G. Ren, P. Wu, and D. Padua. 2005. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*. 89b.

[24] David F. Richards, Omar Aaziz, Jeanine Cook, Hal Finkel, Brian Homerding, Peter McCorquodale, Tiffany Mintz, Shirley Moore, Abhinacv Bhatele, and Robert Pavel. 2018. FY18 Proxy App Suite Release. Milestone Report for the ECP Proxy App Project. Retrieved September 4, 2019 from https://osti.gov.

[25] Christopher D. Rickett, Sung-Eun Choi, and Bradford L. Chamberlain. 2005. Compiling high-level languages for vector architectures. In *Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing (LCPC'04)*. 224–237.

[26] Hideki Saito, Serge Preis, Nikolay Panchenko, and Xinmin Tian. 2016. Reducing the functionality gap between auto-vectorization and explicit vectorization. In *OpenMP: Memory, Devices, and Tasks*, N. Maruyama, B. R. de Supinski, and M. Wahib (Eds.). Springer, Cham, Switzerland, 173–186.

[27] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. 2012. Can traditional programming bridge the Ninja performance gap for parallel computing applications? In *Proceedings of the 2012 39th Annual International Symposium on Computer Architecture (ISCA'12)*. 440–451.

[28] Sergi Siso, Luke Mason, and Michael Seaton. [n.d.]. Code modernization of DLMESO LBE to achieve good performance on the Intel Xeon Phi. In *Proceedings of the EMerging Technology Conference*.15–18.

[29] Kevin Skadron, Margaret Martonosi, David I. August, Mark D. Hill, David J. Lilja, and Vijay S. Pai. 2003. Challenges in computer architecture evaluation. *IEEE Computer* 36 (2003), 30–36.

[30] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, et al. 2017. The ARM scalable vector extension. *IEEE Micro* 37, 2 (March 2017), 26–39. DOI: https://doi.org/10.1109/mm.2017.35

[31] Xinmin Tian, Hideki Saito, Serguei V. Preis, Eric N. Garcia, Sergey S. Kozhukhov, Matt Masten, Aleksei G. Cherkasov, and Nikolay Panchenko. 2016. Effective SIMD vectorization for Intel Xeon Phi coprocessors. *Scientific Programming* 2015 (Jan. 2016), Article 1, 1 page.

[32] Andrew Waterman. 2016. *Design of the RISC-V Instruction Set Architecture*. Technical Report. Electrical Engineering and Computer Sciences, University of California, Berkeley.

[33] Fahimeh Yazdanpanah. 2017. An approach for analyzing auto-vectorization potential of emerging workloads. *Microprocessors and Microsystems* 49 (2017), 139–149.

[34] Bo Zhao, Wei Gao, Rongcai Zhao, Lin Han, Huihui Sun, and Yingying Li. 2015. Performance evaluation of NPB and SPEC CPU2006 on various SIMD extensions. In *Big Data Computing and Communications*, Y. Wang et al. (Eds.). Springer, 257–272.

Received January 2019; revised August 2019; accepted August 2019