Evaluating Auto-Vectorizing Compilers through Objective Withdrawal of Useful Information

SERGI SISO, Hartree Centre, Science and Technology Facilities Council, UK and University of Liverpool, UK
WES ARMOUR, Department of Engineering Sciences, University of Oxford, UK
JEYARAJAN THIYAGALINGAM, Science and Technology Facilities Council, Rutherford Appleton Laboratory, UK

Expectations from 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 extraction or via user annotations, is instrumental for auto-vectorization, and hence towards the overall performance. However, the exact information and its accuracy that is available to compilers at compile-time varies greatly, as does the resulting performance of vectorizing compilers. On this direction, frameworks like Test Suite for Vectorizing Compilers (TSVC) were able to set a precedent for evaluating the vectorization capability of compilers. The overarching approach of TSVC and similar frameworks is to evaluate the compilers under best possible scenarios: assuming that compilers have access to all possible useful information at compile time. Although this optimistic view is useful to see the capability of compilers for auto-vectorization, it does not reflect exactly the conditions found on real-world applications.

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

With our new method in place, 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 adapted TSVC suite and application-level proxy kernels. Our results show that the withdrawing of useful information has a noticeable impact on the performance of binaries generated by all evaluated compilers.

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. Arch. Code Optim.* 1, 1 (May 2019), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

[0]New Paper, Not an Extension of a Conference Paper

Authors' addresses: Sergi Siso, Hartree Centre, Science and Technology Facilities Council, Keckwick Lane, Daresbury, UK, University of Liverpool, Brownlow Hill, Liverpool, UK, sergi.siso@stfc.ac.uk; Wes Armour, Department of Engineering Sciences, University of Oxford, Keble Road, Oxford, UK, Wes.Armour@oerc.ox.ac.uk; Jeyarajan Thiyagalingam, Science and Technology Facilities Council, Rutherford Appleton Laboratory, Harwell Campus, Oxford, UK, t.jeyan@stfc.ac.uk.

1 INTRODUCTION

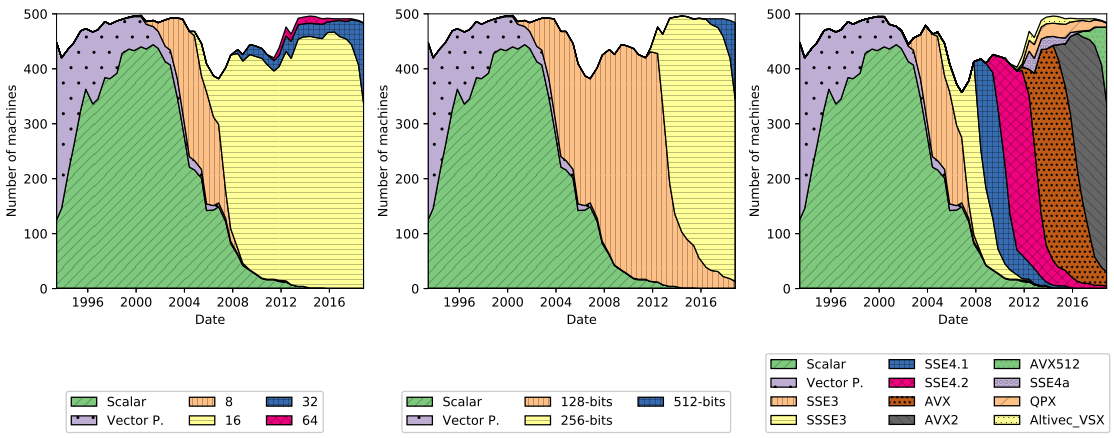
Vectorization is an essential optimization for maximizing the performance of applications on modern systems. The key principle behind vectorization is to identify opportunities to perform a similar set of operations over multiple data elements using a single instruction. Architectures that support such 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 generation of vectorized code. However, the exact resulting performance depends on a number of 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 [22]. Ideally, compilers should identify all possible candidates and vectorize them so that SIMD instructions are leveraged to perform as many operations in parallel as possible.

Although modern compilers provide automatic vectorization optimizations, the abilities of compilers to fully auto-vectorize a given piece of code are often limited. This is primarily due to the lack of contextual information that compilers can extract or perceive from the code. In fact, values of many variables become known only at run-time, which effectively stops vectorization at compile time. Even in cases where the information is available, compilers are conservative in their transformations. This lack of or insufficient information at compile-time simply forces compilers to generate sub-optimal vectorized code [13, 25, 28]. As such, it is important to evaluate whether compilers are able to produce vectorized code, and measure their efficiency in terms of thee capabilities of the targeted architecture.

The Test Suite for Vectorizing Compilers (TSVC) [5] and extensions of TSVC provided by Maleki *et al.* [16] were able to set a precedent for evaluating the vectorization capabilities, and are still widely used. TSVC and similar test suits consist of sets of kernels or loops with a specific implementation and a fixed, and usually generous, amount of contextual information provided to the compiler. Although the loops or kernels are designed to capture the vectorization capabilities, the context information yields the best possible scenario. With the context information feeding the compiler with all possible information needed for vectorization, compilers deliver the best possible performance. In other words, the approach provides a single picture on the performance of compilers under a specific scenario, which is almost or near the ideal case. Although this is useful to know the best possible vectorization capability that a compiler can deliver, the approach, in fact, fails to capture the full spectrum of compiler behavior when a limited amount of information is available. In reality, realistic scientific applications lack a substantial amount of compile-time information, which directly conflicts with the design approach adopted by TSVC.

Furthermore, since the inception of TSVC, both the compiler and processor landscapes have changed rather dramatically. For instance, a number of processors supporting vector capabilities have evolved substantially. A notable and well consolidated resource to witness this is the Top-500 supercomputers in the world [9].

By manually parsing and compiling the Top-500 list, associated system manuals, and relevant processor manuals of the Top-500 systems between the years of 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 the host processor and does not take into account accelerator hardwares available on the systems. Nevertheless, a number of observations can be drawn from the figure. Firstly, systems without SIMD vector capabilities are no longer present. Secondly, the length of vector registers of the newest machines quadrupled (from 128-bits to 512-bits) in recent years, and the adoption of each new instruction sets happens rather quickly. Finally, as Figure 1 (c) shows, systems with *AVX-2* instruction sets are dominating the Top-500 list, with the same 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 last 20 years.

In fact, in addition to what is observed in the Top-500 list, there is also a thriving research around modern vector ISAs such as the ARM Scalable Vector Extension (SVE) [27], the RISC-V "V" Vector Extension [29]. Although, to-date, there are no production-level systems that carry these modern ISAs, they witness the increasing interest on vector processing.

Instead of supplying the compiler with all possible information at compile-time, one could modified the benchmark suite to objectively supply (or withdraw) as much information as possible from compilers that would otherwise aid in the auto-vectorization. Such an approach will not only complement the approach adopted by TSVC, it will also extend that to know the real limitations of compilers.

With this in mind, in this paper, we propose an extension to the TSVC test suite that controls the amount of information provided at compile time and records the benefit in the compiler auto-vectorization when releasing this information. Such approach for testing the auto-vectorization capability with limited information not only emulates more realistic real-world scenarios, but also becomes a more meaningful method to test the auto-vectorization capabilities of modern compilers. In doing this, we make the following contributions:

- we present our approach to extend the TSVC suite 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 can be withdrawn;
- we demonstrate how objectively withdrawing and supplying information at compile-time affects auto-vectorization; and
- we demonstrate the applicability and utility of our method by evaluating 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, called vectorization efficiency spectrum, for easily representing and understanding the auto-vectorization efficiency of compilers.

The rest of this paper is organized as follows: In Section 2, we outline the TSVC suite and other related work, highlighting the limitations of using these to evaluate the auto-vectorization of modern compilers. This is then followed by Section 3 where we describe our proposed approach including a metric for quantifying the auto-vectorization performance, and a method for visualizing the efficiency of vectorization under different settings. In Section 4, we discuss the techniques we used to evaluate the proposed approach. In Section 5, we present the results on modern architectures and compilers. Finally, we conclude the paper in Section 6 with a brief description of the main observations.

2 RELATED WORK

Given that many application developers chose to develop major part of their applications in a scalar style, and to rely on the compiler to optimize the application, there is a compelling reason to determine the efficacy of compilers in their auto-vectorize capabilities. As such, Callahan, Dongarra and Levine developed the Test Suite for Vectorizing Compilers (TSVC) in 1988 [5], containing 100 FORTRAN loops with scalar semantics organized in 4 main categories: dependence analysis, vectorization, idiom recognition, and language completeness. These loops were used to evaluate different compiler-architecture combinations, and the TSVC suite became one of the robust mechanisms to assess the auto-vectorizing capabilities of compilers. However, with the development of new architectures and application structures (TSVC had several loops with GOTO statements) the suite became progressively outdated. Maleki *et al.* [16], provided a revision of the TSVC suite by converting the original loops to the C programming language, and extended the loops to cover vectorization issues, which were not part 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 vectorizes 90 loops, while XLC does 68 and GCC with 59 loops. They 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, and finally iii) inability of the compilers to re-order computations to avoid data dependencies or their inability to perform algorithm substitution.

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

"*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 are provided towards all loops, such as values of loop-parameters or conditionals. In terms of assessing the capabilities of compilers, what is considered as a trivial piece of information in the case of TSVC suite, may be, in fact, very effective piece of information for triggering and enabling a number of follow-on vectorization opportunities. These are well evidenced by several authors. For instance, Eichenberger *et al.* [10] propose vectorization techniques when the alignment is suboptimal. Another example, as demonstrated by Henderson *et al.* [13], and Siso *et al.* [25], number of applications achieving better vectorization performance when the loop-bound parameters are provided to the compiler. There is also a large body of work that aims to bridge the performance gap due to auto-vectorization. A number of examples can be found in the Partial Control-Flow Linearization by Moll *et al.* [18], which improves the if-conversion vectorization technique. Saito *et al.* [23] explore syntax extensions to support explicit vector programming.

Nevertheless, the synthetic TSVC suite is still widely used for evaluating the capabilities of modern auto-vectorizing compilers, and is, for instance, included in the LLVM benchmarking suite [15]. Although the view of supplementing the compiler/vectorizer with as much as possible information is beneficial for evaluating the best possible performance, in

reality, the information available to the compilers are very limited. This is not by choice, but primarily owing to the design complexity of applications which deal with multiple conflicting trade-offs such as code flexibility, modularization and readability or code quality. Under such circumstances, a more sensible approach would be to evaluate the compilers under realistic conditions — i.e. by withholding a certain degree of information.

There has been more recent analysis using TSVC suite. For instance, Moldavanova *et al.* [17] use TSVC for quantifying the performance of the first generation Intel Xeon Phi co-processor. Hossein *et al.* [2] create similar benchmarks to TSVC that resorts to common kernels ranging from simple matrix-matrix multiplications to more complex 2D discrete wavelet transform, opposed to synthetic loops as in TSVC. Although the PARSEC suite [3] was created for system level performance evaluation, to a certain extent it can also be used for evaluating the auto-vectorizing compilers. Fahimeh *et al.* [30] analyze the auto-vectorization limitations within the PARSEC benchmark suite, particularly to find out why only a few of the loops were vectorized. Cebrian *et al.* provide the ParVec benchmark suite [7], directly deriving it from the PARSEC suite. Similarly, Zhao *et al.* [31] analyses vectorization performance of the well known *NPB* and *SPEC CPU2006* benchmarks in various SIMD extensions. However, most of these approaches are pure performance analysis than trying to address the best vectorization performance versus realistic vectorization performance.

In order to get a better approximation of real-world vectorization performance, some computing domains have developed their own proxy applications. For instance, a number of signal processing and multimedia-specific application benchmarks have been created to assess the auto-vectorization performance of compilers. Ren *et al.* [20], Fritts *et al.* [11], and Alvanos *et al.* [1] all compare the performance of auto-vectorizers against hand-vectorized counterparts using a set of multimedia applications. All these studies identify and report the missed opportunities for vectorization. Additionally, there are available some extensive collections of proxy applications, such as the collection from the ECP project [21].

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] shares 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 macro-benchmarking is discussed in Satish *et al.* [24]. Their approach starts off from a naive C implementation by applying multiple SIMD optimizations.

An approach for automatically generating benchmarks is discussed in Deshpande *et al.* [8]. However, their overall objective was not to evaluate the SIMD performance or vectorization capabilities of compilers. Skadron *et al.* [26] highlight a number of practical issues in the current generation of benchmarking infrastructures. Breughe *et al.* [4] proposed a method for understanding the impact of different input parameters on the overall performance. This approach is different to conventional approaches where assessing the performance via workload characterization is rather common. Gong *et al.* [12] study the variations of compiler performance to different loop mutations. They demonstrate the instability of compilers to consistently provide optimal results when the implementation is slightly different. However, but they do not consider multiple levels of information provided at compile-time.

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 performance variation that a piece of information has on the vectorization performance. The amount of information exposed to the compiler is parametrized and can be objectively withdrawn in a controlled manner.

To achieve this in a controlled manner, we have built a C-preprocessor infrastructure that expands multiple macros representing different classes of static information, as discussed in the next sub-section. The overall evaluation relies on two aspects: i) specific meta-information defining the class of information that is withdrawn or withheld from the compiler, and ii) a set of benchmark loops derived from the TSVC suite produced by Maleki *et al.*. The evaluation process has an ultimate control over which class of information is presented or withdrawn or withheld. Based on this control information, the preprocessor expands the macros using TSVC suite as a template to generate a set of benchmarks. The generated benchmarks, unlike the TSVC suite, 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.

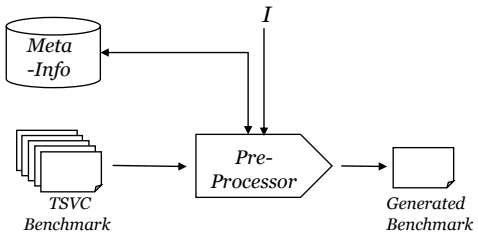


Fig. 2. Architecture for Withdrawing Information from Benchmarks, denoted by I.

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

3.2 Classification of Effective Information

We provide a classification based on the types of effective contextual information we would like to withdraw from the tests at compile-time. These are:

i) Loop Bounds;
ii) Parameters in Array Indices and Offsets;
iii) Parameters in Conditional evaluations; and
iv) Aliasing and Alignment of Arrays.

We discuss each of these classes with an appropriate example below:

(i) **Loop Bounds**: Although simple loop-nests can often be conditionally vectorized without full-fledged information on loop-bounds, complex loop-nests, however, often demand at least partial information for deciding the candidate loop for vectorization. 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.

Listing 1. Example with Loop Bound Parameters

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

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, N, and 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 sub-optimal. On the other hand, if M is very large, the `gather` operations on the accesses towards array A will be expensive and other additional costs related to the memory subsystem could be incurred. But 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.

(ii) **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.

Listing 2. Example with Parameters in the Array Indices

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

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 will refuse to auto-vectorize the loop or it will 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 is contiguous in memory and it can use cheaper load operations. However, if `Offset(j)` is a non-constant value (or if the value is unknown), the vectorization has to be implemented using gather operations.

(iii) **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 prior to fully vectorizing a given part of a code. More specifically, compilers' ability to vectorize a series of operations become impeded by path divergence resulting from conditional evaluations. Consider the example in Listing 3, consisting of a single loop with a conditional additional update on array A.

Listing 3. Example with parameters in the condition evaluation

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

Here, the left side operand of the conditional evaluation changes every iteration of the loop. If the `Constant` has a run-time 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 which contain multiple exit conditions, as compilers just vectorize certain patterns of loops with multiple exits.

(iv) **Aliasing and Alignment of Arrays**: Modern vector units are sensitive to the alignment of data [10]. For instance, Intel Xeon Phi relies on a 64-byte alignment for a number of follow-up optimizations. At the minimum, instructions facilitating load/store operations require the alignment information to be known at compile time. Default memory allocations, which default to the alignment of 8-byte is insufficient to meet any of these demands.
Consider the example in Listing 4, consisting of a single loop accessing 3 different arrays. Depending on the compiler visibility to the array declarations 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 the built-ins are compiler-specific.

Listing 4. Example with array attributes

```
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 ( i=1; i<= N; i++){
    A[i] = B[i] + C[i];
}
```

It is worth noting that, although alignment and aliasing were discussed by Maleki in [16], it was not fully explored in their paper. One possible hypothesis here is that, given that their approach was to provide the compiler with as much information as possible, there was an expectation that relevant attributes to be automatically inserted by the compiler.

To generate our adapted test suite, we use the preprocessor infrastructure described in Section 3.1 against templates of the TSVC benchmark with the desired contextual information classes supplied (or withdrawn). However, not every information class is expected to be present on every TSVC test. In such cases, the code generated by the preprocessor infrastructure is same as the original TSVC code. It is worth noting that among all the 135 TSVC tests, there are:

- one or multiple instances of information classes (i) and (iv);
- only 67 of those tests use information class (ii); and
- only 29 of those tests use information class (iii).

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 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 non-vectorized 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 as statistic information 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 sub-group categorization provided in previous TSVC implementations. These categories are: linear dependence, indirect addressing, equivalencing, loop re-rolling, packing, searching, recurrences, reductions, crossing thresholds, expansion, node splitting, loop restructuring, statement re-ordering, symbolics, control flow, global data flow and induction variable propagation;
(2) i is a class of information that is exposed to the compiler. I is the set that includes the four classes presented in section 3.2 plus an instance with no information provided at compile time and an instance with all four information classes provided. (Note that no class is a direct representation of the original TSVC benchmark as the information classes were not provided consistently through the test suite);
(3) a is an architecture on which the evaluation is being carried out and A the set containing all evaluated architectures; and
(4) c is a compiler on which the evaluation is being carried out and C 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:

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

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 another geometric mean across all categories.

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

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

3.4 Visualizing Vectorization Efficiency

As shown above, it is possible to reduce the dimensionality of results by data aggregation or by resorting to statistical analysis. However, the number of information categories and the number of compiler-architecture combinations cannot

be reduced any further. For instance, in our case, there are 17 loops categories and 16 compiler-architecture pairs. As such, any attempt to compare the vectorization efficiency of different compilers across all the loop categories it becomes visually challenging, particularly when one wanting to reduce the visual clutter.

For this reason we developed a new chart, which we refer to as *Vector Efficiency Spectrum*, which provides an easier approach, not only for comparing the vectorization efficiency of a given compiler in the absence of information, but also offers an approach for assessing their relative efficiencies. In this chart, 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 their efficiencies when information for category i is hidden at compile time. The lines can be drawn in color, but due to the large number of lines we connect the lines in each spectrum that represent the same compiler-architecture and we also connect them to the corresponding label in the legend. Figure 3 on the results section is an example of such chart.

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 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.

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	2.6 GHz	1.3 GHz	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	1024K	1024K	512K
L3 Cache	22528K	-	8192K
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

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 generation. We manually analyzed the generated code to verify or probe the results. In all of the cases, for each test, we repeated all scalar and vector executions 5 times and taken the smallest reported value, in order to minimize the perturbations from the operating system.

Table 2. Compiler Flags Used in the Evaluation

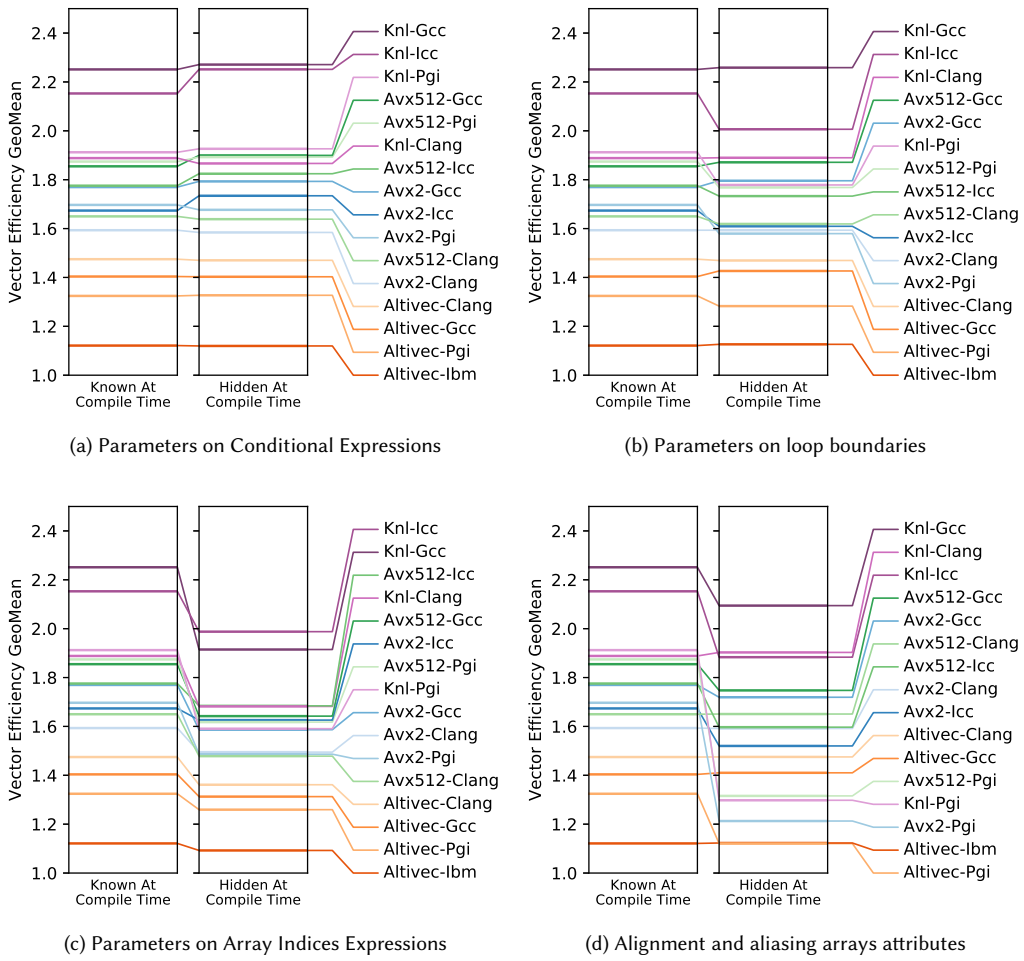
Compiler	Vectorization Disabled	Vectorization enabled
GNU	-march=$ISA -O3 -ffast-math -fno-tree-vectorize	-march=$ISA -O3 -ffast-math (-mprefer-vector-width=512 on Skylake-avx512)
Intel	-x$ISA -O3 -fp-model fast=2 -no-vec	-x$ISA -O3 -fp-model fast=2 (-qopt-zmm-usage=high on skylake-avx2)
Clang	-march=$ISA -O3 -ffast-math -fno-vectorize	-march=$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 suite and a selection of application-level proxy kernels. In Section 5.1, we present the results of the TSVC suite, when different classes of information (as defined in Section 3.2) are withdrawn and withheld at compile-time. We show the resulting impact using vectorization efficiency. For easier the 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 in order to provide a closer look into the strengths and weaknesses of modern compiler auto-vectorization capabilities, when all the information are provided to the compiler. Finally, to ensure that the approach has a realistic utility in 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 classified the types of information that are available to compilers into four classes: loop bounds, parameters in array indices and offsets, parameters in conditional evaluations, and aliasing and alignment of arrays. We consider each of these information classes in turn, and we consider two cases: information belonging to the class under consideration being fully supplied, and 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 we use here has two parts: one when all information is known at compile time (left), and another for the case when all information is withdrawn at compile-time (right).



(a) Parameters on Conditional Expressions

(b) Parameters on loop boundaries

(c) Parameters on Array Indices Expressions

(d) Alignment and aliasing arrays attributes

Fig. 3. Vectorization Spectrum for the TSVC cases

In addition to these individual cases, we show the case that classes of information fully withdrawn or supplied in Figure 4. In addition to the vectorization spectrum charts, we summarize the overall vector efficiency geometric mean for each compiler and platforms in Table 3, grouped together for each information class.

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 overall observation, the following case-specific observations can be made:

(1) In almost all of the cases, hiding information from the compilers reduces their effective vectorization efficiency. The most relevant exception to this is the case of Conditional Parameters. Withdrawing the conditional parameters yields better performance and improves the average auto-vectorization efficiency among all compilers except the *Clang* compiler.

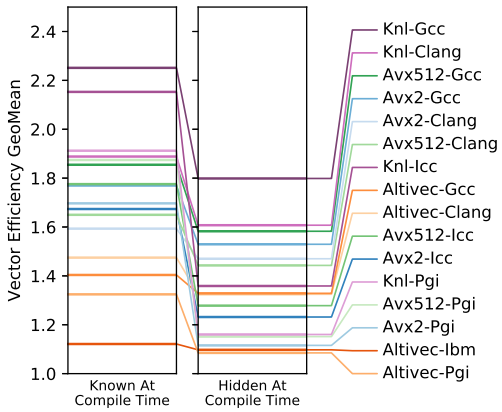


Fig. 4. Vectorization Spectrum for the TSVC cases

Table 3. Percentage variation of Vectorization Efficiency Geometric Mean when providing Compile-Time Information on different compilers.

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
	AVX2	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

A closer examination reveals that this counter-intuitive auto-vectorization improvement comes mainly from two distinct issues. The *Intel* compiler stops vectorizing the tests $S261$, $S253$, $S3251$, $S254$ and $S1161$ (all from the *Expansion* category) across all platforms. Looking at the compiler optimization reports, we found that each of these loops raises a "*vector dependence prevents vectorization*" remark. However, manual checking of the loops

ensured that there are no actual dependencies. Even for cases where there were dependencies, those dependencies could have been been easily resolved by reordering the relevant statements in the body of the loop. As this issue only appears in the *Intel* compiler for this specific loop structure, we can only believe this to be a bug in the compiler.

Additionally, the *GNU* and *PGI* compilers have a negative outlier on test *S*276 (on all x86 platforms). Listing 5 shows the source for the *S*276, where the *cp_n1* parameter is exposed to or hidden 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 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 it is just an 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 the scenarios, *PGI* produces a similar vectorized implementation using the *blendvps* (SIMD conditional copy) operation. However, when the parameters are known at compile-time, it chose to distribute the value of the conditional with an *vextractf128* and multiple *vpshufd* operations in every loop, while it generated a pure vector version when the parameter is unknown. These additional operations add a significant overhead that reduces the vectorization performance. It is also noteworthy 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.

Listing 5. Source code of TSVC test S276

```
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];
    }
}
```

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

When analyzing the regressions on the *GNU* compiler, we find that tests *S*276 and *S*115 make significant (negative) contributions towards the overall performance (thus on the geometric mean). Performance issues surrounding the test case *S*276 has also been discussed in (1). In this case, the variable *LEN* (Listing 5) represents the loop bound whose value information is withdrawn. This indirectly impacts the conditional expression, leading to the performance issues explained above.

As for the loop *S*115, it is worth noting that this benchmark represents a triangular iteration space through a two-dimensional array with and offset (See Listing 5). 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 towards the accesses to the array *a*. However, with *LEN*2 having a value of 256, compiler considers the triangular

loop is too small for a vectorized implementation, and resorts to a scalar implementation. When the loop bound parameter values are given at runtime, the compiler uses a multi-versioning strategy to resolve the 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.

Listing 6. Source code of TSVC test S115

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

(3) Exposing Index Parameters provide 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 provide 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 suppliying Variable attributes. A further exploration of this issue in comparison to other compilers shows that *Clang* does not enable additional vectorization opportunities when the information is supplied that other compilers did manage to exploit. This represents one of the main weakneses of this compiler, and in fact, as shown in Figure 3d when array attribute is supplied it has the worst vectorization efficiency geometric mean on each x86 platform, while 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 relatively least sensitive to the information differences, with variation of 2.2%, (5.6%–25.2%), and (8.4%–17.5%) respectively. While, 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 to the underlying vector lengths. This is readily observable in the vectorization spectrum charts. For instance, spectral lines for the *AVX512* (KNL and Skylake) platforms achieve better efficiency compared to the *AVX2* 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, *GNU* compiler gets 5.7% on Altivec when all parameters are given at compile-time while it improves the overall performance 25.2% under the same conditions in the *AVX512-KNL* platform.

5.2 TSVC with All Information Exposed at Compile-time

In the previous section, we analyzed the impact of information withdrawal on the overall vectorization efficiency, for the different TSVC tests. However, TSVC can in turn be subdivided to 17 categories of loops testing different compiler capabilities. The full analysis on the impact that each information class has on the overall performance (for each category) can be rather detailed, and prohibiting here to discuss within the page limits of this manuscript. For the reasons of brevity, we limit the discussion to some of the notable results here, particularly when the maximum amount of information is provided. This particularly relevant here for two reasons: first is that it closely matches the original scope of the TSVC suite (although it is not an exact match), and secondly it should, ideally, demonstrate the

best auto-vectorization performance of compilers. The results are presented in Figures 5 and 5, covering each of the vector platforms. A number of observations can be made from these results:

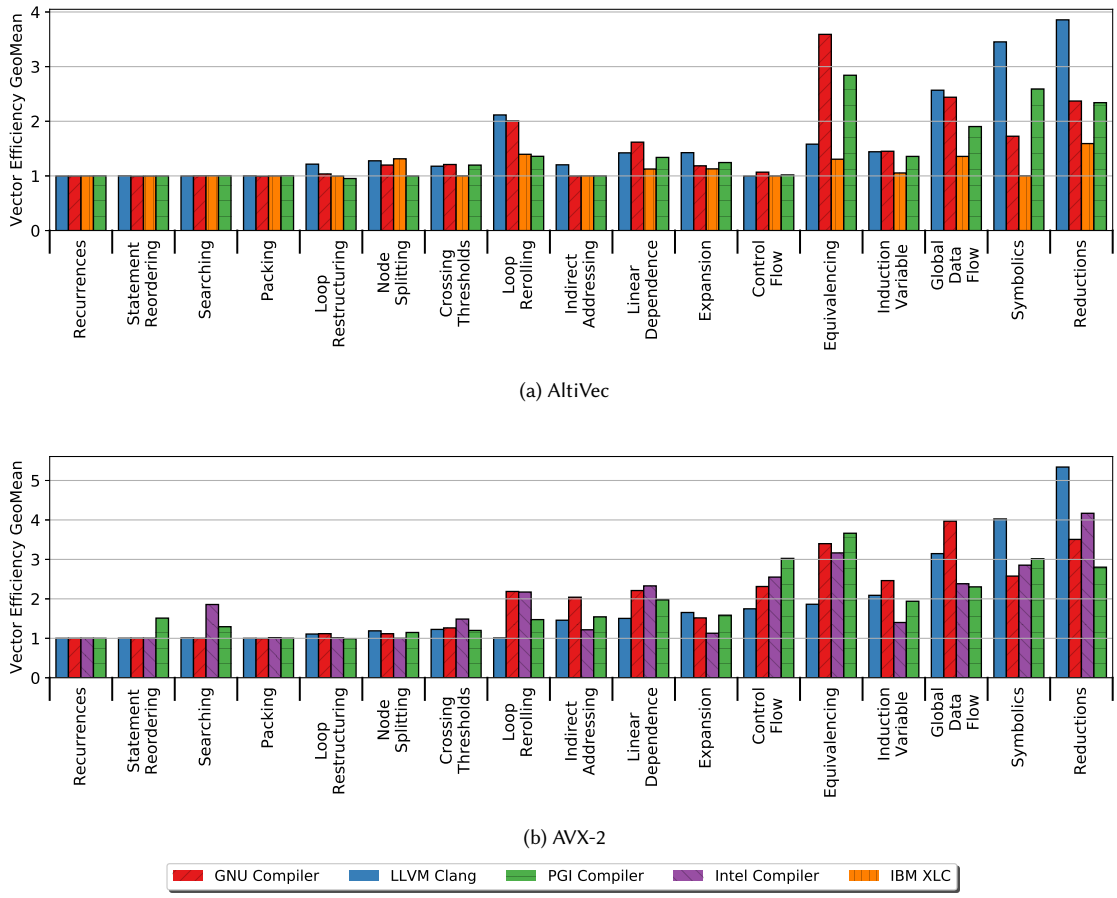


Fig. 5. Overall performance vectorization when all relevant information are provided to the compiler (Altivec and AVX2)

(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 gap between the vectorized and non-vectorized versions being large. When directly comparing the two *AVX-512* platforms, the *KNL* platform has slightly better auto-vectorization performance than *Skylake*. This is true in most of the cases. We believe that this is due to significant improvement on the *Intel* compiler's capability to perform *Reductions*

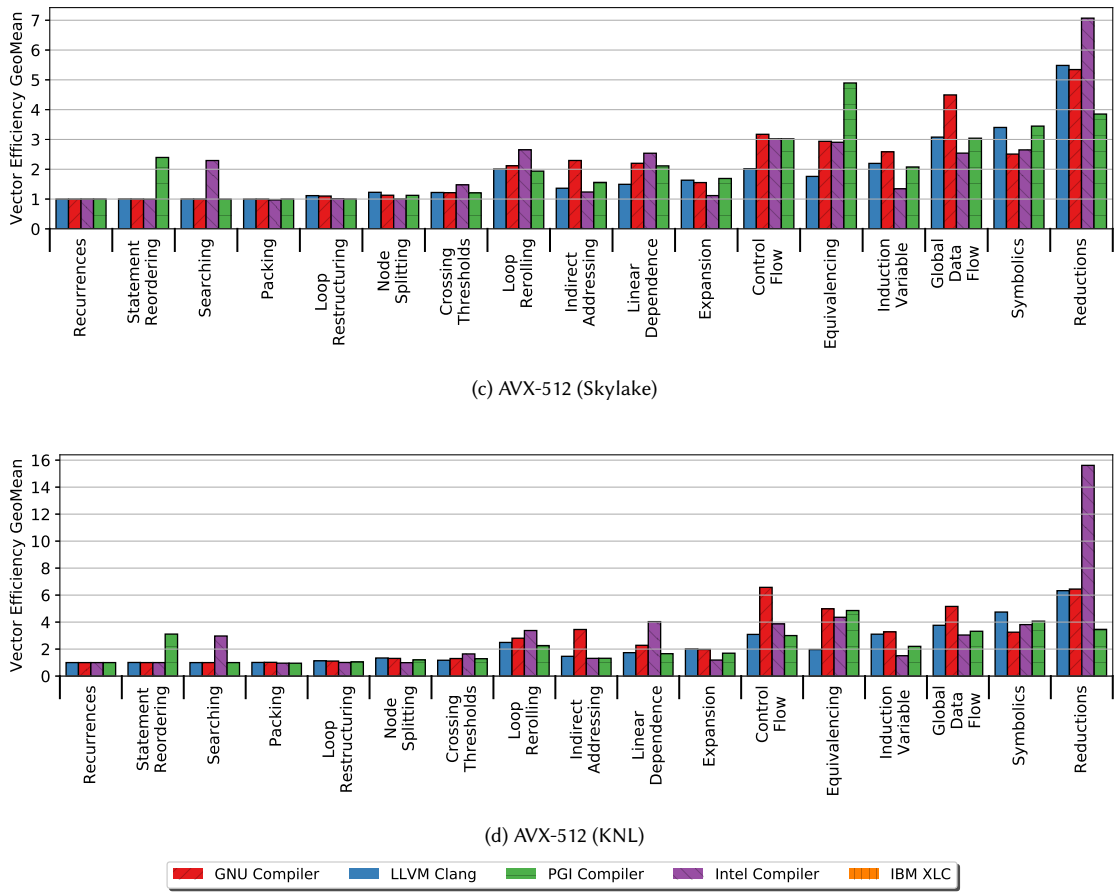


Fig. 5. Overall performance vectorization when all relevant information are provided to the compiler (AVX512-KNL and AVX512-Skylake)

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.

(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 hard to be vectorized by most compilers. *Statement Reordering* was only vectorized by the *PGI* compiler on the three x86 platforms. The *Searching* was only vectorized by the *Intel* compiler on x86 platforms and by the *PGI* compiler exclusively on *AVX2*.
For instance, Listing 7 shows the source for the test S331 from the *Searching* category. In this example (and in all other tests) *Intel* has been the only compiler which generated the AVX-512 *vpcmpud* operation (integer

SIMD comparison that creates a mask as result). This has lead the Intel compiler to be the only one to vectorize this particular test in the two AVX-512 platforms.

Listing 7. Source code of TSVC test S331

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

(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 yielded 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 the TSVC test S311 (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 non-vectorized 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-bits `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 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 micro-architectural aspects. Further analysis is essential to confirm this hypothesis.

Listing 8. Source code of TSVC test S311

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

Listing 9. Assembly code from scalar S311 test with ICC on KNL platform

```
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 10. Assembly code from auto-vectorized S311 test with ICC on 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
```

5.3 Applications Kernels

In order to verify that the proposed approach has a utility beyond synthetic benchmarks, we evaluated the macro-based infrastructure proposed in this paper on six proxy kernels that represent the common building blocks found in numeric applications. 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). These configurations are:

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

The choice of proxy kernels are partially motivated by the benchmarks used to evaluate OpenMP SIMD [14] and ISPC [19]. Both OpenMP SIMD and ISPC have similar vectorization scope to ours. The implementations we have closely resembled the ISPC. Although they are not full-fledged applications, they share similar hot-spots with many computer simulation softwares. It is also worth noticing that the capacity of each compiler to enable auto-vectorization optimizations is heavily dependent on the specific implementation. For this reason we provide a brief description for each of the proxy application, and the aspects surrounding their implementation, including information that was withdrawn at compile-time:

- **Binomial Options**: It is an iterative pricing model often used in finance workloads. The implementation can expose or withdraw the information pertaining to the number of time steps in the iterative procedure (loop bounds), and the array attributes ensuring the non-aliasing of the arrays, from the compiler at compile-time.

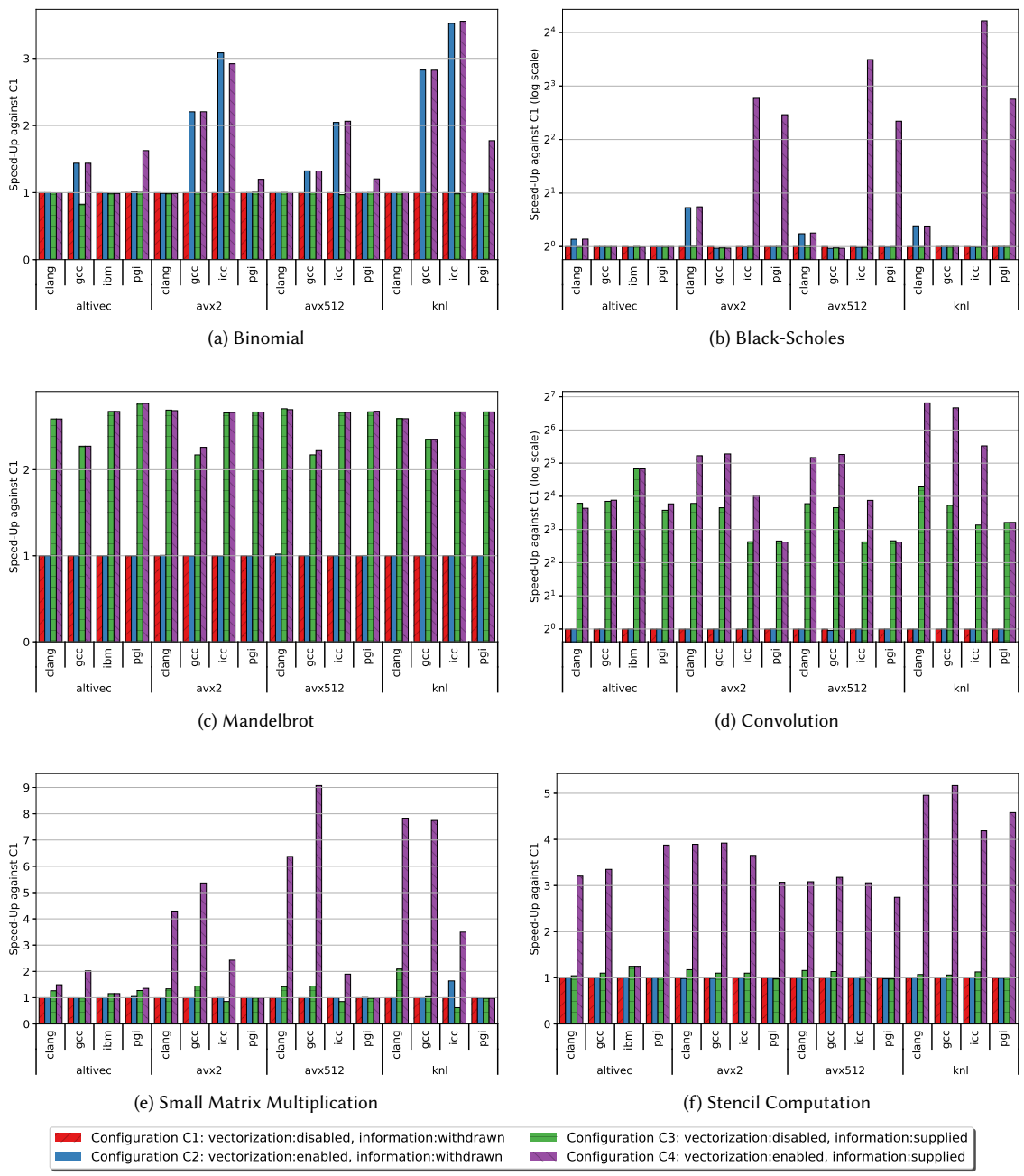
- **Black-Scholes**: This is another pricing model used in financial workloads. The implementation computes the Black-Scholes formula for each option. In this case, the non-aliasing attribute in multiple arrays is the only additional information that can be supplied to or withdrawn from the compiler.
- **Mandelbrot**: It computes the Mandelbrot set for a complex function $f(z) = z^2 + c$, for non-diverging complex number z, $z \in \mathbf{Z}$ and $z > 0$. The algorithm is performed over a two-dimensional image, with the function $f(z)$ computed for each pixel. The information withdrawn includes the loop bounds and some arithmetic parameters that are part of the computation.
- **Convolution**: It applies a convolution mask to a two-dimensional image by sliding the mask in a two-dimensional space. The information provided or withdrawn from the compiler includes the non-aliasing parameters, the size of the image, the size of the mask and the values for the mask. All these parameters specify the bounds of different nested loops in the algorithm.
- **Small Matrix Multiplication**: It 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 the kernel.
- **Stencil Computation**: It is a three-dimensional 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.

We present the results in Figure 6. For each proxy application, we compare the speedup of configuration C_1 against other configurations. In other words, the baseline is C_1 (Vectorization: Disabled, Information: Withdrawn). This speed-up normalization abstracts away the scalar performance difference between compilers and architectures, and instead, focuses on the benefits of the auto-vectorization when more information is provided at compile-time. Each of the application kernels has a distinct behaviors when additional information is provided.
Again, a number of observations can be made:

- *Binomial Options*: 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. Finally, the *Clang* and *IBM* compilers were not able to auto-vectorize this kernel even when additional information is presented.
- *Block-Scholes*: The *Clang* compiler was able to auto-vectorize the algorithm without the non-aliasing attributes. The *ICC* and *PGI* compilers were able to vectorize the code when additional information is provided, and obtained much better vectorization performance in the *AVX2* and *AVX512* (both Skylake and KNL) platforms.
- *Small Matrix Multiplication*: 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 is enabled, these three compilers achieved considerable vectorization performance using the additional information.

The other proxy kernels all show similar auto-vectorization behaviors across compilers:

- All compilers were able to auto-vectorize the *Mandelbrot* algorithm, when contextual information is provided at compile-time. However, all compilers managed to improve their scalar performance by a factor of at least two when additional information is provided.



(a) Binomial

(b) Black-Scholes

(c) Mandelbrot

(d) Convolution

(e) Small Matrix Multiplication

(f) Stencil Computation

Fig. 6. Compiler comparison of macro-benchmark kernels with multiple levels of information

- Again, all compilers faired well on the *Convolution*, even with the contextual information withdrawn at compile-time. When adding the information, all compilers gets significant scalar performance improvements and in addition, *Clang*, *GNU* and *Intel* compilers offered noticeable speed-ups in the x86 platforms.
- Finally, all compilers were able to auto-vectorize the *Stencil Computation* without the non-aliasing information. However, the vectorization performance improved when non-aliasing 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 CONCLUSIONS

In this paper we have proposed a new methodology for evaluating the auto-vectorizing capability of compilers. Our proposed methodology relies on objectively supplying and withdrawing useful information that are available at compile-time. This approach is orthogonal to the approach adopted in the original and extended versions of the TSVC benchmarks [5, 16], where they made every possible piece of information available to the compiler at compile-time. We see that the approach outlined here provides a complementary and more detailed mechanism to test modern vectorizing compilers. The original approach evaluates the best possible vectorization efficiency, whereas our approach evaluates the range of efficiency, starting with the worst.

We applied the method to the TSVC tests suite and multiple application-level proxy benchmarks, 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* compilers. We also devised a new visualization mechanism, namely *Vector Efficiency Spectrum*, to present, interpret and understand these benchmarking results.

The first observation is that in modern compilers, the resulting performance from auto-vectorized loops are still far from the architectural peak performance. The exact reasons for the sub-optimal performance can either be ascribed to dependencies that prevent vectorization and to towards 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 deciding the performance of resulting code from auto-vectorization. The exact significance of each of the information class on the final performance varies greatly across compiler-architecture pairs. In decreasing order of importance, these classes are: Loop Bound Parameters, Index Parameters and Variable Attributes. Moreover, we found that the parameters found inside Conditional Expressions presented an auto-vectorization regression when supplied to any of the compilers. Furthermore, the overall negative effect of withdrawing information is experienced on all compilers. When more than one class of information are withdrawn, the effects are compounded, his can be seen particularly to some application-level kernels.

Conclusively, there are two approaches to evaluating compiler, assuming that compilers have access to all information or being selective on the amount of available information. Each leads to complementary methods for evaluating the vectorization efficiency of compilers. The main advantage of the latter (and thus the proposed) approach is that it is able to match with real-world conditions, where the information available to the compilers at compile time are limited. Therefore this new approach provides a new set of evaluation tools which can help compiler developers identify weaknesses on current compiler implementations; and it can help programmers understand and plan which information they want to express statically in the code to maximize vectorization, or alternatively remove unnecessary contextual information for better readability.

REFERENCES

[1] M. Alvanos and P. Trancoso. 2016. Video SIMDBench: Benchmarking the Compiler Vectorization for Multimedia Applications. In *2016 Euromicro Conference on Digital System Design (DSD)*. 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 Trans. Archit. Code Optim.* 10, 4 (Dec. 2013), 37:1–37:24.

[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 Computer Society Press, 98–105.

[6] Juan M. Cebrian, Magnus Jahre, and Lasse Natvig. 2014. Optimized hardware for suboptimal software: The case for SIMD-aware benchmarks. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2014), 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 Perform. Eval. Rev.* 40, 2 (Oct. 2012), 99–105.

[9] 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.

[10] 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, USA, 82–93.

[11] Jason E. Fritts, Frederick W. Steiling, Joseph A. Tucek, and Wayne Wolf. 2009. MediaBench II Video: Expediting the Next Generation of Video Systems Research. *Microprocess. Microsyst.* 33, 4 (June 2009), 301–318.

[12] Zhangxiaowen Gong, Alexandru Nicolau, Josep Torrellas, Zhi Chen, Justin Szaday, David Wong, Zehra Sura, Neftali Watkinson, Saeed Maleki, David Padua, and Alexander Veidenbaum. 2018. An empirical study of the effect of source-level loop transformations on compiler stability. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29. DOI: http://dx.doi.org/10.1145/3276496

[13] 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* (1st ed.). MKF Publishers., USA, Chapter 2, 7–23.

[14] 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)*. Springer-Verlag, Berlin, Heidelberg, 59–72.

[15] LLVM. 2018. LLVM test-suite: TSVC. (2018). http://llvm.org/svn/llvm-project/test-suite/trunk/MultiSource/Benchmarks/TSVC/

[16] 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 Computer Society, Washington, 372–382.

[17] 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*, Victor Malyshkin (Ed.). Springer International Publishing, Cham, 143–150.

[18] 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 2018*. ACM Press, New York, New York, USA, 543–556. DOI: http://dx.doi.org/10.1145/3192366.3192413

[19] M. Pharr and W. R. Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*. San Jose, USA, 1–13.

[20] G. Ren, P. Wu, and D. Padua. 2005. An Empirical Study On the Vectorization of Multimedia Applications for Multimedia Extensions. In *19th IEEE International Parallel and Distributed Processing Symposium*. 89b–89b.

[21] 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. (10 2018). DOI: http://dx.doi.org/10.2172/1482870

[22] 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)*. Springer-Verlag, 224–237.

[23] 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*, Naoya Maruyama, Bronis R. de Supinski, and Mohamed Wahib (Eds.). Springer International Publishing, Cham, 173–186.

[24] 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 *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 440–451.

[25] Sergi Siso, Luke Mason, and Michael Seaton. Code modernization of DLMESO LBE to achieve good performance on the Intel Xeon Phi. In *Proceedings of the EMerging Technology (EMiT) Conference*. (2016), B.D. Rogers, D. Topping, F Mantovani, and M.K. Bane (Eds.). 15–18.

[26] 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.

[27] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (mar 2017), 26–39. DOI: http://dx.doi.org/10.1109/mm.2017.35

[28] 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. *Sci. Program.* 2015, Article 1 (Jan. 2016), 1:1–1:1 pages.

[29] Andrew Waterman. 2016. *Design of the RISC-V Instruction Set Architecture.* Technical Report. Electrical Engineering and Computer SciencesUniversity of California at Berkeley.

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

[31] 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*, Yu Wang et.al (Ed.). Springer, 257–272.