Java Vector API: Benchmarking and Performance Analysis[*]

Matteo Basso
matteo.basso@usi.ch
Università della Svizzera italiana (USI)
Lugano, Switzerland

Andrea Rosà
andrea.rosa@usi.ch
Università della Svizzera italiana (USI)
Lugano, Switzerland

Luca Omini
luca.omini@usi.ch
Università della Svizzera italiana (USI)
Lugano, Switzerland

Walter Binder
walter.binder@usi.ch
Università della Svizzera italiana (USI)
Lugano, Switzerland

Abstract

The Java Vector API is a new module introduced in Java 16, allowing developers to concisely express vector computations. The API promises both high performance, achieved via the runtime compilation of vector operations to hardware vector instructions, and portability. To the best of our knowledge, there is no study evaluating the performance of the new Java Vector API.

To bridge this gap, we propose JVBench, to the best of our knowledge, the first open-source benchmark suite for the Java Vector API. JVBench extensively exercises the features introduced by the Java Vector API, resulting in high API coverage. We use JVBench to evaluate the performance and portability of the Java Vector API on multiple architectures supporting different vector instruction sets. We compare the performance of the Java Vector API on our benchmarks w.r.t. other semantically equivalent implementations, including scalar (non-auto-vectorized) Java code as well as Java code auto-vectorized by the Just in Time (JIT) compiler. Finally, we report patterns and anti-patterns on the use of the Java Vector API significantly affecting application performance.

CCS Concepts: • **General and reference** → **Empirical studies**; **Evaluation**; **Performance**; • **Software and its engineering** → *Just-in-time compilers*; *Dynamic compilers*; *Runtime environments*; • **Computing methodologies** → **Parallel programming languages**; • **Computer systems organization** → **Single instruction, multiple data**.

Keywords: Benchmarks, SIMD, Java, Vector API, Parallelism, Just-in-time compilation, Code optimization.

ACM Reference Format:
Matteo Basso, Andrea Rosà, Luca Omini, and Walter Binder. 2023. Java Vector API: Benchmarking and Performance Analysis. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC '23), February 25–26, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3578360.3580265

[*]This work has been supported by Oracle (ERO project 1332) and by the Swiss National Science Foundation (project 200020_188688).

1 Introduction

Modern processors provide Single Instruction Multiple Data (SIMD) capabilities via *vector instructions* that exploit dedicated hardware. Vector instructions are particularly effective to access contiguous memory and perform the same operation on different elements. For this reason, transforming scalar loops to vectorized loops, i.e., loops that exploit vector instructions, is crucial to obtain significant speedups. We refer to the process of transforming scalar code to vectorized code as *vectorization.*

Vectorization is usually performed by the compiler, which recognizes suitable code using scalar instructions and automatically transforms it into code using vector instructions. We say that compilers *auto-vectorize* scalar code. Although auto-vectorization is an automatic technique, developers have to write scalar code following predetermined patterns (e.g., using counted loops [23]). Otherwise, auto-vectorization may fail in optimizing scalar code. To mitigate this issue, languages often offer high-level APIs or intrinsics that map to vector instructions and allow developers to exploit this type of parallelism explicitly. We call this strategy *explicit vectorization.*

In this paper, we focus on managed language runtime systems and, in particular, on the Java Virtual Machine (JVM). To express explicit vector operations using an object-oriented

API, Java offers the Java Vector API [12, 32], which was introduced in the Panama [3] project and is accessible starting from JDK 16. Using the Java Vector API, developers benefit from greater performance without renouncing the advantages of Java as a high-level programming language. Thanks to the runtime compilation of Java bytecode to optimized machine code, the same explicitly vectorized code implemented using the Java Vector API can be executed on several architectures (possibly supporting different hardware capabilities), increasing code maintainability and portability.

Contributions. Our work aims at analyzing the performance of the new Java Vector API; to the best of our knowledge, such an analysis has not been conducted before. To this end, evaluating the performance of the API with a benchmark suite that significantly exercises vectorization is crucial. Since we are not aware of any realistic application or benchmark that uses the Java Vector API, we design and develop JVBench, the first open-source benchmark suite extensively exercising the Java Vector API (Sec. 3). JVBench consists of several realistic and diversified benchmarks, specifically designed for evaluating vectorization and exercising most of the features of the Java Vector API, resulting in high API coverage. Our benchmarks are inspired from well established workloads to evaluate performance of vectorized code [8, 9, 42], which we recasted to the Java Vector API.

We use JVBench to evaluate the performance (in terms of execution time) of the Java Vector API w.r.t. other semantically equivalent implementations, including scalar (non-auto-vectorized) Java code and Java code auto-vectorized by the Just in Time (JIT) compiler (Sec. 4). Moreover, we evaluate the performance of the Java Vector API on different architectures, reporting cases where the lack of hardware capabilities leads to performance degradation w.r.t. the corresponding scalar implementation.

Thanks to our suite, we also identify several patterns and anti-patterns on the use of the Java Vector API significantly affecting application performance (Sec. 5). Our work offers concrete suggestions to implement high-performance vectorized application code, pinpointing performance issues and improvements that can be implemented in future JDKs.

We complement the paper by presenting background information (Sec. 2), a discussion on related work (Sec. 6), and our concluding remarks (Sec. 7).

2 Background

The Java Vector API [12, 32] allows expressing vector computations using classes and methods that abstract hardware vector registers and instructions. The abstract class `Vector<E>` represents the abstraction of the Java Vector API over vectors of element type `E`. In particular, the API provides abstract subclasses of `Vector<E>` that map to primitive element types, i.e., `ByteVector`, `ShortVector`, `IntVector`, `LongVector`, `FloatVector`, and `DoubleVector`. In addition to the element type, concrete subclasses of `Vector<E>` have associated a shape represented by class `VectorShape`. The shape determines the bit size of the vector and hence the hardware vector register that will map to that vector. The combination of element type and shape (represented by the interface `VectorSpecies`) determines the *vector length*, i.e., the number of elements the vector contains, computed as the bit size of the vector divided by the bit size of the element. For example, an instance of `IntVector` with a `VectorShape` of length 512 bits has a vector length of 16 elements (since integers in Java occupy 32 bits).

```
1 static void scalarAdd(int[] a, int[] b, int[] c) {
2   for (int i = 0; i < a.length; i++) {
3     c[i] = a[i] + b[i];
4   }
5 }
```

Figure 1. Method `scalarAdd`.

Operations on vectors are represented using instance methods defined in class `Vector<E>` and in its subclasses. Vector operations can be divided into *lane-wise* operations if they apply a scalar operator (e.g., addition) to each element of one or more vectors in parallel, or *cross-lane* operations if they process the whole vector (e.g., reductions that produce a scalar from a single vector). Operations on `Vector<E>` can be conditionally applied on a subset of elements using a mask parameter of type `VectorMask<E>`, which contains a boolean value for each element. Instances of class `VectorMask<E>` are mapped to predicate registers, i.e., hardware registers used to mask off certain lanes of the vector. instructions.

Internally, the JVM and the JIT compiler implement the Java Vector API leveraging intrinsic functions, i.e., the Java Vector API has a default Java implementation that is later replaced by more efficient machine code. At runtime, the JIT compiler emits machine code that uses vector registers, vector instructions, and predicate registers, as expressed by the application code. The default implementation is executed before JIT compilation occurs, as well as in the case where the underlying platform does not support some of the requested vector features. This design allows executing applications exercising the Java Vector API even on platforms that do not support some vector operations.

Figure 1 shows an example `scalarAdd` Java method that uses a scalar loop (lines 2–4) to add up the numbers of arrays `a` and `b` (provided as parameters at line 1) and store the sums in array `c` (line 3), also provided as a parameter. We assume that arrays `a`, `b`, and `c` have the same length.

Using the Java Vector API, the scalar loop in method `scalarAdd` can be re-implemented as shown in Figure 2. First, we declare a `static final` field that stores the optimal `Vector Species` (line 2) that will be later used in the vectorized loop. The `IntVector.SPECIES_MAX` field stores a species of element type `Integer` and shape of maximum length supported on

Table 1. Classes and methods of the Java Vector API exercised by each benchmark.

Benchmark		**axpy**	**blackscholes**	**canneal**	**jacobi2d**	**lavaMD**	**particlefilter**	**pathfinder**	**somier**	**streamcluster**	**swaptions**
Vector Type	DoubleVector	✓			✓		✓		✓		✓
	FloatVector		✓			✓				✓	
	IntVector		✓	✓				✓			
VectorMask			✓	✓			✓				✓
API Methods	Vector Creation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Vector Manipulation		✓								
	Unary		✓	✓		✓	✓				✓
	Binary	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Comparisons		✓				✓				✓
	Transcendental and Trigonometric		✓			✓	✓		✓		✓
	Reductions			✓		✓				✓	✓

```
1  static final VectorSpecies<Integer> SPP =
2          IntVector.SPECIES_MAX;
3
4  static void vectorAdd(int[] a, int[] b, int[] c) {
5    int i = 0;
6    int limit = SPP.loopBound(a.length);
7
8    for (; i < limit; i += SPP.length()) {
9      IntVector vA = IntVector.fromArray(SPP, a, i);
10     IntVector vB = IntVector.fromArray(SPP, b, i);
11     vA.add(vB).intoArray(c, i);
12   }
13
14   for (; i < a.length; i++) {
15     c[i] = a[i] + b[i];
16   }
17 }
```

Figure 2. Explicitly vectorized version of method `scalarAdd` (Figure 1), using the Java Vector API.

the executing platform. Then, we define a `vectorAdd` method that has the same arguments and return type of method `scalarAdd` (line 4). Method `vectorAdd` first declares an `i` and a `limit` variable of type `int` used to iterate over the arrays (lines 5 and 6, respectively). To initialize variable `limit`, we use the `VectorSpecies.loopBound(int length)` instance method that, given a `length` value, returns the largest multiple of the vector length that is less than or equal to that `length` value. This `limit` value is used to implement the vectorized loop at lines 8–12, which loops from `0` to `limit` with step `SPP.length()`, i.e., the vector length specified by `SPP`, henceforth referred to as `VLENGTH` for simplicity.

In the body of the vectorized loop, we declare variables `vA` and `vB` to store two vectors containing `VLENGTH` elements starting from index `i` of arrays `a` and `b`, respectively. These elements are loaded using the `IntVector.fromArray(..)` method calls at lines 9 and 10. Vectors `vA` and `vB` are then used to add up the elements and store the result in array `c` using the lane-wise `IntVector.add(..)` and `IntVector.intoArray(..)` instance methods (line 11), respectively. We note that the upper bound `limit` of the vectorized loop ensures that no out-of-bound exception is thrown when accessing arrays. However, to process potential tail elements, i.e., elements whose index is greater than or equal to `limit` but less than the array length, we implement the scalar loop at lines 14–16.

3 Benchmark Suite

In this section, we present JVBench, our benchmark suite that extensively exercises the Java Vector API. JVBench is open-source and is available at https://github.com/usi-dag/JVBench.

JVBench consists of ten benchmarks specifically designed for evaluating vectorization. In particular, we port ten C/C++ benchmarks to Java; these workloads [35] were proposed by Ramírez et al. [36] to benchmark vector microarchitectures. They are well-established in the literature and are in turn taken from existing benchmark suites, such as Rodinia [9], PolyBench [42], and ParVec [8] (a vectorized version of PARSEC [5, 43]). JVBench allows us to evaluate the performance of the Java Vector API on diversified benchmarks, i.e., on applications from different domains with different data-level parallelism patterns that exercise most of the vector instructions. For a detailed description of each benchmark and more information regarding the diversification of the benchmarks, we refer to the article by Ramírez et al. [36].

For each benchmark, we implement two different Java versions. The first version uses only Java scalar code, while the second version employs the Java Vector API to exploit vector instructions. This allows us to evaluate the speedups available to auto-vectorization and obtained by the Java Vector API, respectively, w.r.t. an equivalent scalar implementation.

Table 1 shows the features of the Java Vector API (reported on the y-axis) exercised by the vector implementation of each benchmark of our suite (reported on the x-axis). We divide features into three groups: *Vector Type*, *VectorMask*, and *API Methods*. *Vector Type* contains the subclasses of `Vector<E>`, *VectorMask* represents any masked operation, and *API Methods* contains a classification of the vector operations as reported by related work [13]. The table does not report subclasses of `Vector<E>` as well as categories of vector operations that are exercised by no benchmark. We do not exercise any *shape-changing* operations because as stated by the official JEP [32], shape-changing operations can negatively impact portability and performance. Finally, we do not report the vector length since our implementation always uses vectors of the maximum vector length, i.e., we use the maximum vector length based on the underlying architecture to maximize the potential runtime speedup.

The table shows that five benchmarks exercise `DoubleVector`, three benchmarks exercise `FloatVector`, and three benchmarks exercise `IntVector`. Each benchmark makes use of a single vector type with the exception of *blackscholes* that uses both `FloatVector` and `IntVector`. Four benchmarks (out of ten) exercise the class `VectorMask<E>` (and hence predicate registers) to perform masked operations, such as masked memory accesses, masked lane-wise operations, or masked cross-lane operations. As expected, all benchmarks use operations to create vectors. However, only *blackscholes* uses operations to manipulate vectors (in particular, `Vector.blend(..)` and `VectorMask.cast(..)`). Five benchmarks execute unary operations (such as `Vector.abs()` and `VectorMask.not()`), while all benchmarks perform binary operations (such as `Vector.add(..)`, `Vector.mul(..)`, `VectorMask.and(..)`, etc.). Three and five benchmarks exercise comparisons (such as `Vector.eq(..)`) and transcendental and trigonometric operations (such as `Vector.sqrt()`), respectively. Finally, four benchmarks perform reductions (e.g., by summing up all the elements of a vector). Considering the benchmarks altogether, JVBench covers most of the features of the Java Vector API.

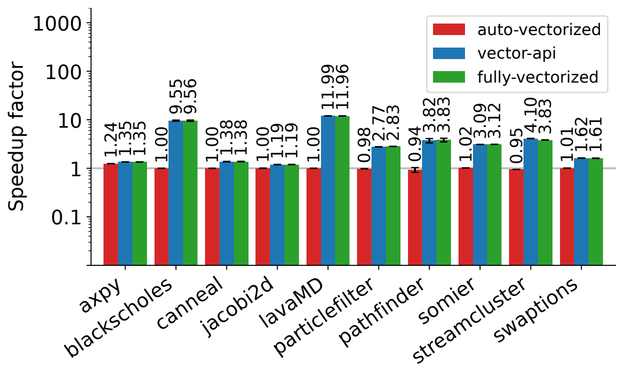
4 Evaluation

Here, we present our evaluation settings (Sec. 4.1) and evaluate the performance of the Java Vector API (Sec. 4.2).

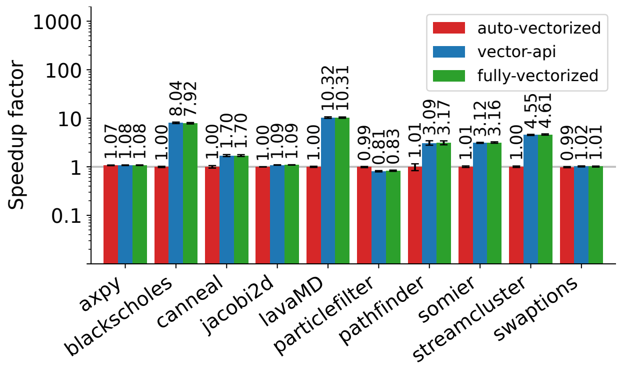
4.1 Evaluation Settings

We run all experiments on three machines $M_{AVX_{512}}$, M_{AVX_2}, and M_{AVX}. $M_{AVX_{512}}$ is equipped with an 18-core Intel i9-10980XE (3.00 GHz) with 256 GB of RAM; M_{AVX_2} is equipped with an 16-core Intel i9-10885H (2.40 GHz) with 32 GB of RAM; M_{AVX} is equipped with two NUMA nodes, each with an 8-core Intel Xeon E5-2680 (2.7 GHz) and 64 GB of RAM. Frequency scaling, turbo boost, and hyperthreading are disabled, CPU governor is set to "performance". All the three machines run Linux Ubuntu (kernel v. 5.4.0-58-generic).

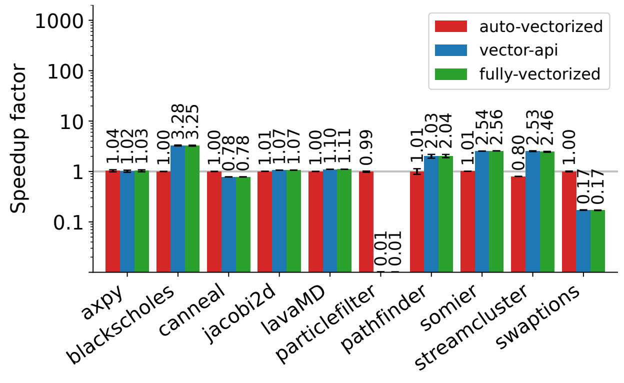
$M_{AVX_{512}}$ supports most of the recent vector instructions defined in the *Streaming SIMD Extensions* (SSE) and *Advanced Vector Extensions* (AVX), while M_{AVX_2} and M_{AVX} do not. M_{AVX} supports only the Intel-defined CPU flags `sse`, `sse2`, `ssse3`, `sse4_1`, `sse4_2`, and `avx`, while M_{AVX_2} supports also the flags `avx2`, and `fma`. In addition to the flags of M_{AVX_2}, $M_{AVX_{512}}$ supports also the flags `avx512f`, `avx512dq`, `avx512cd`, `avx512bw`, `avx512v`, and `avx512_vnni`. This allows evaluating the performance of the Java Vector API in the best case scenario, i.e., in the case where the JIT compiler can compile all vector operations to the corresponding vector instructions (machine $M_{AVX_{512}}$), and in two suboptimal scenarios (machines M_{AVX_2} and M_{AVX}), i.e., in a case where some vector operations are executed using the default Java implementation of the Java Vector API. The maximum vector size supported by $M_{AVX_{512}}$, M_{AVX_2}, and M_{AVX} is 512, 256, and 128 bits, respectively. We note that the Java Vector API aims at offering reliable runtime compilation and performance only on the x64 and AArch64 architectures [32]. Hence, our evaluation settings ensure reasonable hardware-platform coverage w.r.t. the hardware platforms supported by the API.



(a) $M_{AVX_{512}}$



(b) M_{AVX_2}



(c) M_{AVX}

Figure 3. Evaluation of the Java Vector API.

We perform our experiments on OpenJDK 19, i.e., the latest release at the time of writing. The JDK uses the HotSpot C2 compiler, which implements Superword Level Parallelism [24]. In C2, the superword optimization is applied only to unrolled loops, and unrolling is performed only for counted loops [23]. We plan to conduct our experiments on other JVM implementations (such as OpenJ9 [10] and GraalVM [2]) as part of our future work since, at the time of writing, the development of the API-related compiler optimizations on these implementations is ongoing [1, 11].

4.2 Evaluation Results

We execute four different versions of each benchmark of our suite: *scalar*, *auto-vectorized*, *vector-api*, and *fully-vectorized*. The *scalar* version uses the Java scalar implementation of the benchmark, disabling the compiler auto-vectorization; this version serves as a baseline to evaluate the speedups achieved by the other three versions. The *auto-vectorized* version uses the Java scalar implementation of the benchmarks without disabling the compiler auto-vectorization; it is useful to assess the speedups automatically enabled by the compiler. The *vector-api* version uses the Java Vector API implementation of the benchmarks, disabling the compiler auto-vectorization; it allows assessing the speedups enabled by the Java Vector API. Finally, the *fully-vectorized* version uses the Java Vector API implementation and the compiler auto-vectorization simultaneously. This version enables the analysis of possible interferences of the Java Vector API and the compiler auto-vectorization. We note that it is not our goal to compare the performance speedups enabled by the Java Vector API with the performance speedups presented in the original article by Ramírez et al. [36].

Figure 3 reports the speedup factor computed as $T_{scalar}/T_{version}$, where $T_{version}$ refers to the execution time of either *auto-vectorized*, *vector-api*, or *fully-vectorized*, and T_{scalar} refers to the execution time of *scalar*. Each plot refers to a different machine. In particular, the plots 3a, 3b, and 3c report the speedup factors on $M_{AVX_{512}}$, M_{AVX_2}, and M_{AVX}, respectively. The benchmarks are reported on the x-axis of the plot, while the speedup factor is reported on the logarithmic y-axis. Above each bar, we report the exact value of the speedup as the mean of 10 measurements. The black error bars represent the 95% confidence intervals (CI) of the measurements.

We analyze now the three versions of the benchmarks. By analyzing the *auto-vectorized* version, we notice that *axpy* is the only benchmark the compiler is able to auto-vectorize effectively, and hence it is the only benchmark that yields a speedup similar to the other versions. The speedups of the *auto-vectorized* version are 1.24×, 1.07×, and 1.04× on $M_{AVX_{512}}$, M_{AVX_2}, and M_{AVX}, respectively. The reason is that *axpy* is much simpler than the other benchmarks of the suite—auto-vectorization is not inhibited by potentially aliasing references or dependencies between loop iterations. These results confirm the need for an API to explicitly express vector computations, such as the Java Vector API.

Next, we consider the *vector-api* version on $M_{AVX_{512}}$ and M_{AVX_2}. Our results show that most of the benchmarks highly benefit from explicit vectorization, with overall slightly better results on $M_{AVX_{512}}$ than on M_{AVX_2}. This is expected, since the maximum vector size on $M_{AVX_{512}}$ is 512 bits while the maximum vector size on M_{AVX_2} is 256 bits. Explicit vectorization on *blackscholes* and *lavaMD* yields impressive speedups of 9.55× and 11.99× on $M_{AVX_{512}}$ and 8.04×, and 10.32× on M_{AVX_2}, respectively. The reason is that *blackscholes* and *lavaMD* perform many vector computations.

Among the evaluated benchmarks, *axpy*, *canneal*, *jacobi2d*, and *swaptions* benefit the least from explicit vectorization, with a maximum speedup of 1.62× (*swaptions* on $M_{AVX_{512}}$). Despite the moderate 2.77× speedup on $M_{AVX_{512}}$, *particlefilter* introduces a slowdown of 0.81× on M_{AVX_2}. In particular, *particlefilter* is the only benchmark that encounters a slowdown w.r.t. its corresponding *scalar* version when using the Java Vector API on M_{AVX_2} and $M_{AVX_{512}}$. We are investigating the causes of this slowdown. Speedups obtained by using the *vector-api* version range from 1.19× (*jacobi2d*) to 11.99× (*lavaMD*) and from 0.81× (*particlefilter*) to 10.32× (*lavaMD*) on $M_{AVX_{512}}$ and M_{AVX_2}, respectively. The average speedup factor[1] is 2.98× on $M_{AVX_{512}}$ and 2.39× on M_{AVX_2}.

The *vector-api* version on M_{AVX} yields significantly smaller speedups than the ones on $M_{AVX_{512}}$ and M_{AVX_2}. This is because M_{AVX} only supports a maximum vector size of 128 bits, and because M_{AVX} does not support many hardware vector instructions. For instance, M_{AVX} does not support predicate registers and hence cannot efficiently execute masked operations. This is reflected by the poor performance of benchmarks *canneal* (0.78×), *particlefilter* (0.01×), and *swaptions* (0.17×). In these benchmarks, the Java Vector API falls back to a Java implementation that allocates the vector instances and executes Java methods encoding vector operations as scalar loops. Since the execution of *blackscholes* is not dominated by masked operations, *blackscholes* still yields a speedup of 3.28×. Speedups obtained by using the *vector-api* version on M_{AVX} range from 0.01× (*particlefilter*) to 3.28× (*blackscholes*), 0.77× on average.

In terms of speedups, we notice that the *fully-vectorized* and *vector-api* versions are comparable (except *particlefilter* on M_{AVX_2} due to the reasons explained in the previous paragraph). This indicates that the compiler auto-vectorization does not interfere with the Java Vector API.

To summarize, on modern architectures, the Java Vector API provides significant performance benefits w.r.t. auto-vectorization, which is ineffective in optimizing almost every benchmark in our suite. On architectures that do not support

[1]Average speedup factors across multiple benchmarks are computed using the geometric mean.

```
 1  static final VectorSpecies<Integer> SPP =
 2          IntVector.SPECIES_MAX;
 3
 4  static void vectorAdd(int[] a, int[] b, int[] c) {
 5    for (int i = 0; i < a.length; i += SPP.length()) {
 6      VectorMask<Integer> mask =
 7          SPP.indexInRange(i, a.length);
 8      IntVector vA =
 9          IntVector.fromArray(SPP, a, i, mask);
10      IntVector vB =
11          IntVector.fromArray(SPP, b, i, mask);
12      vA.add(vB).intoArray(c, i, m);
13    }
14  }
```

Figure 4. Method `vectorAdd` of Figure 2 using the `indexInRange` API.

modern vector instructions and predicate registers, the performance benefits are less evident, confirming the graceful performance-degradation goal of the API for most of the evaluated benchmarks. The graceful performance-degradation goal is not achieved for three of the evaluated benchmarks, indicating the need for benchmarking when implementing performance-critical code on old hardware.

5 Patterns and Anti-Patterns

In this section, we present patterns and anti-patterns exploiting the Java Vector API, i.e., performant API usages and semantically equivalent less performant API usages, respectively. In particular, we present patterns and anti-patterns that we found in our benchmarks. Since the benchmarks are representative of real-world applications using vectorization, the reported patterns and antipatterns have practical relevance for all users. First, we present our evaluation settings (Sec. 5.1). Then, we present the performance implications of using the `indexInRange` API (Sec. 5.2) and transcendental and trigonometric lane-wise operations (Sec. 5.3). Finally, we discuss the `xor` and `fma` operations (Sec. 5.4 and 5.5).

5.1 Evaluation Settings

We run our experiments to evaluate patterns and anti-patterns using the same evaluation settings described in Sec. 4.1. However, we evaluate each pattern/anti-pattern separately only on the benchmarks of our suite that can exercise that pattern/anti-pattern. For example, we evaluate the `pow` operation (Sec. 5.3) only on four benchmarks of our suite, i.e., all the benchmarks of our suite that need to compute the power of a number. In this section, each figure reports the speedup factor of each pattern and anti-pattern using the same formula presented in Sec. 4.2, i.e., $T_{scalar}/T_{pattern}$, where $T_{pattern}$ refers to the execution time of the implementation that uses a certain *pattern* and T_{scalar} refers to the execution time of the *scalar* version. We note that the Java Vector API implementation of our benchmarks (employed in Sec. 4.2) uses the patterns that we have found, leading to the best performance.

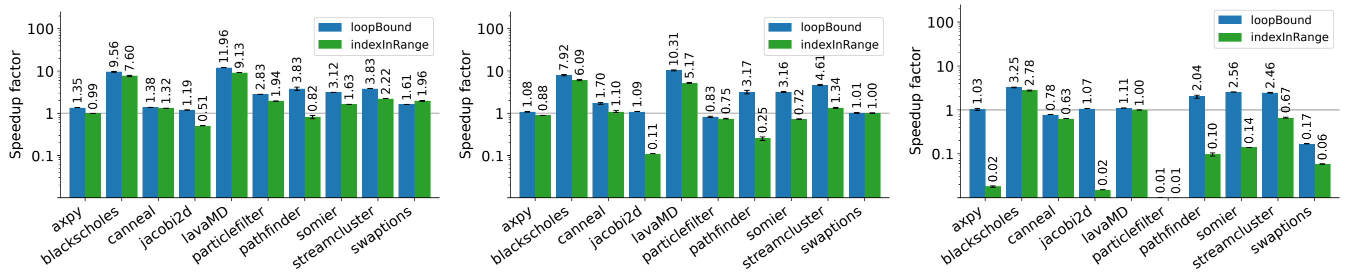
5.2 indexInRange API

Figure 2 shows an example `vectorAdd` method that uses the upper bound provided by the `VectorSpecies.loopBound(..)` method (line 6) to loop over the array and vectorize the addition (lines 8–12). The main drawback of this approach is that the tail elements exceeding the upper bound must be processed by a scalar loop (lines 14–16). The programmer is burdened to implement both a vectorized loop and a scalar loop that are semantically equivalent, duplicating code and hence, increasing maintenance costs and complexity.

To simplify the Java source code, removing the scalar loop, the Java Vector API offers a `VectorSpecies.indexInRange(offset, limit)` instance method returning a mask of the given species where only the lanes at index N are set, such that $N + \mathit{offset} \in [0..\mathit{limit} - 1]$. Figure 4 shows the `vectorAdd` method of Figure 2 implemented using `indexInRange`. While the scalar loop (Figure 2, lines 14–16) has been completely removed, the remaining `for` statement loops until `i` reaches the length of the array `a.length` (line 5). For each iteration, the `indexInRange` method is invoked providing `i` as offset and `a.length` as limit (line 7) and returns the mask to be used to load from (lines 8–11) and write into arrays (line 12) without throwing out-of-bounds exceptions—the mask prevents access to the arrays beyond their length.

On platforms supporting predicate registers (such as $M_{AVX_{512}}$ and M_{AVX_2}), the developers of the Java Vector API (according to the JEP [32]) would prefer users to implement code using `indexInRange`, which should lead to performance comparable to `loopBound`. On platforms that do not support predicate registers (such as M_{AVX}), the `indexInRange` API would achieve only suboptimal performance. Following these statements, we are interested in 1) testing whether `loopBound` and `indexInRange` performance is indeed comparable or one is slightly faster than the other, and 2) whether the `indexInRange` API leads to acceptable performance on platforms that do not support predicate registers.

Figures 5 shows the performance of the two APIs on all the benchmarks of our suite. In particular, experimental results show that `loopBound` is faster than `indexInRange` both on platforms that support and on platforms that do not support predicate registers (except *swaptions* on $M_{AVX_{512}}$). On platforms that support predicate registers, `loopBound` yields a speedup that ranges from 0.83× (*particlefilter* on M_{AVX_2}) to 11.96× (*lavaMD* on $M_{AVX_{512}}$), 2.67× on average, while `indexInRange` yields a speedup that ranges from 0.11× (*jacobi2d* on M_{AVX_2}) to 9.13× (*lavaMD* on $M_{AVX_{512}}$), 1.33× on average. On these platforms, `indexInRange` achieves speedups on most of the evaluated benchmarks. For benchmarks *axpy*, *jacobi2d*, *pathfinder*, and *somier*, `indexInRange` introduces a slowdown w.r.t. the corresponding *scalar* version. The reason is that a masked vector operation (e.g., a vector memory access and a vector computation) is typically slower than the corresponding unmasked operation.



(a) $M_{AVX_{512}}$ **(b)** M_{AVX_2} **(c)** M_{AVX}

Figure 5. Evaluation of the `loopBound` and `indexInRange` APIs.

On platforms that do not support predicate registers, `loopBound` is significantly faster than `indexInRange`, with speedups ranging from 0.01× (*particlefilter*) to 3.25× (*blackscholes*) and from 0.01× (*particlefilter*) to 2.78× (*blackscholes*) for `loopBound` and `indexInRange`, respectively. The average speedup is 0.76× for `loopBound` and 0.14× for `indexInRange`. On these platforms, `indexInRange` introduces a slowdown w.r.t. the corresponding *scalar* version on most of the benchmarks.

In contrast to the suggestion of the developers of the Java Vector API [32], experimental results suggest the usage of the `loopBound` method over the `indexInRange` method to achieve better performance. Users may consider using the `loopBound` method to implement portable code that does not lead to performance degradation on architectures that do not support predicate registers, despite increasing code size and complexity. For instance, we recommend the usage of the `loopBound` method for the development of third-party Java libraries where the executing architecture is not defined a priori, old, or subject to change. Finally, experimental results highlights the need for compiler optimizations to improve the performance of the `indexInRange` API.

5.3 Transcendental and Trigonometric Lane-Wise Operations

On x64, to support transcendental and trigonometric lane-wise operations on floating point vectors (e.g., *sin*, *tan*, *log*, *pow*, and more), the Java Vector API leverages the Intel Short Vector Math Library (SVML) [14, 15]. In particular, based on the `VectorSpecies`, the JIT compiler replaces transcendental and trigonometric lane-wise operations of the Java Vector API with calls to SVML functions.

We analyze whether the compiler properly optimizes code that leverages these operations. In particular, we evaluate two versions of four benchmarks of our suite that compute the square of floating point numbers stored in a vector `vec`. Version *pow* uses the expression `vec.pow(2)` while version *mul* uses the semantically equivalent expression `vec.mul(vec)`. We expect that, thanks to compiler optimizations, versions *pow* and *mul* result in similar performance.

Figure 6 shows that *mul* is faster than *pow* on all the evaluated benchmarks and machines. Moreover, *pow* yields a slowdown on all the evaluated benchmarks (except *blackscholes* on $M_{AVX_{512}}$ and M_{AVX_2}) w.r.t. the corresponding *scalar* version. The reason is that *mul* does not need to perform any function call to the SVML library (in contrast to *pow*) and the `mul` operation is compiled directly to a single vector instruction. In contrast to our expectation, the measurements indicate that the compiler does not optimize (even if it could) invocations to the SVML library—the expression `pow(2)` can be strength-reduced at compile time to `vec.mul(vec)`.

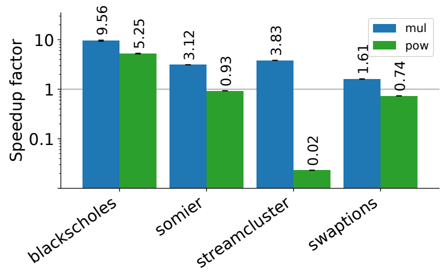
Among the evaluated benchmarks, *streamcluster* suffers the most from performance degradation when using the `pow` operation. We note also that M_{AVX_2} and M_{AVX} do not support hardware vector instructions for *pow*, which is supported only by $M_{AVX_{512}}$. This explains the moderate speedup of *pow* on M_{AVX_2} w.r.t. the significant speedup of *pow* on $M_{AVX_{512}}$ for the *blackscholes* benchmark (2.10× and 5.25×, respectively). The average *mul* speedup is 3.68× on $M_{AVX_{512}}$, 3.29× on M_{AVX_2}, and 1.37× on M_{AVX}, while the average *pow* speedup is 0.54× on $M_{AVX_{512}}$, 0.79× on M_{AVX_2}, and 0.27× on M_{AVX}.

Our experimental results suggest to use transcendental and trigonometric lane-wise operations with care. Moreover, our findings suggest a potential lack of compile-time optimizations of vector operations. We plan to open an issue on this matter and analyze more in-depth potential missed optimizations as part of our future work.

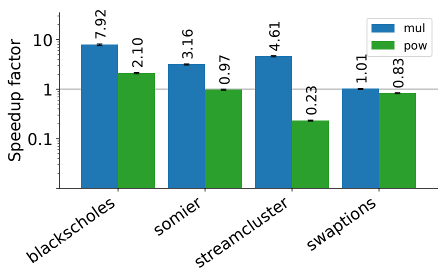
5.4 Xor Operation

The `VectorMask` class, i.e., the abstraction of the Java Vector API representing hardware vector masks, offers several instance methods to manipulate masks, including `not()`, `and(..)`, `andNot(..)`, `or(..)`, and `eq(..)`. However, `VectorMask` does not provide any public `xor(..)` method that is compiled directly to the corresponding hardware vector instruction, even though such a hardware vector instruction is supported by some architectures.

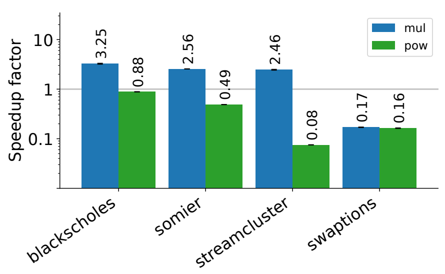
To implement an `xor` operation between two masks, the programmer can combine other existing operations to perform a semantically equivalent computation yielding the



(a) $M_{AVX_{512}}$



(b) M_{AVX_2}



(c) M_{AVX}

Figure 6. Evaluation of the `vec.mul(vec)` and `vec.pow(2)` patterns.

```
public Float512Mask not() {
  return xor(maskAll(true));
}

public Float512Mask eq(VectorMask<Float> mask) {
  Objects.requireNonNull(mask);
  Double512Mask m = (Double512Mask)mask;
  return xor(m.not());
}
```

Figure 7. Java implementation of the not and eq Java Vector API operations on masks in OpenJDK.

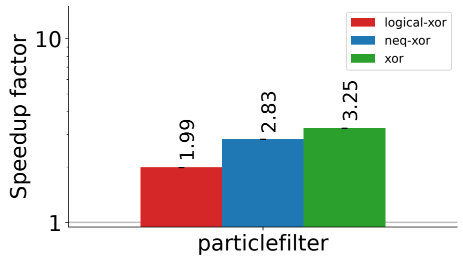


Figure 8. Evaluation of the xor pattern on $M_{AVX_{512}}$.

same result. For example, given `mask1` and `mask2`, two instances of `VectorMask` of the same generic type, `mask1 xor mask2` is equivalent to the logical expression

```
mask1.or(mask2).andNot(mask1.and(mask2));
```

and also to the expression

```
mask1.not().eq(mask2);
```

However, by inspecting the source code of OpenJDK, we note that `eq(..)` and `not()` (and hence also `andNot(..)`, internally implemented as `mask1.and(mask2.not())`) rely on a package-visible method `xor(..)`, as shown in Figure 7. This `xor(..)` method, which cannot be referenced by application code, is efficiently compiled to the corresponding hardware vector instruction if the underlying architecture supports it.

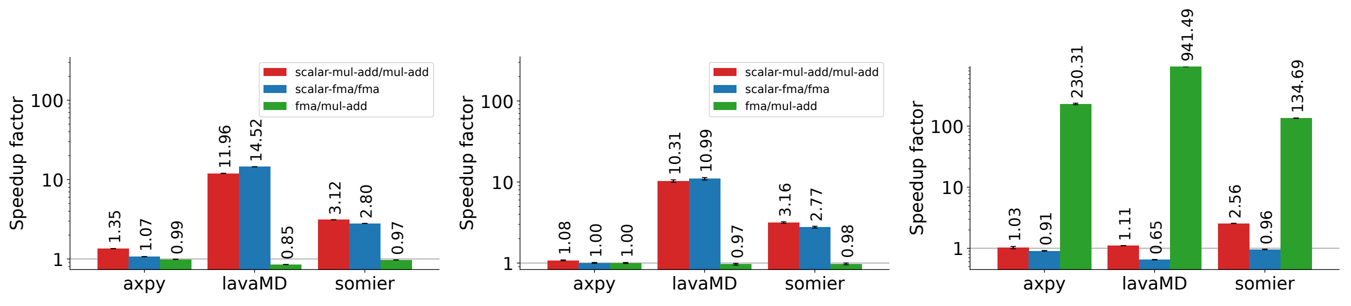
Among the benchmarks of our suite, *particlefilter* requires a binary `xor(..)` operation on two masks. For this reason, we evaluate three implementations of the `xor(..)` operation. The first and the second implementations named *logical-xor* and *neq-xor* rely on the first and second patterns presented in the previous paragraph, respectively. The third implementation simply named *xor* uses the aforementioned package-private `xor(..)` method. To do so, we compile and execute the *xor* implementation using a custom JDK build that defines the `xor(..)` method as `public`.

Figure 8 shows the experimental results of the xor patterns on $M_{AVX_{512}}$. We do not report a plot for M_{AVX} and M_{AVX_2}, since *particlefilter* (i.e., the only evaluated benchmark for these patterns) yields poor performance on M_{AVX} and M_{AVX_2} even using the most efficient pattern (as shown in Sec. 4.2). As expected, our results show that the public `xor` method yields better performance than *neq-xor*, which in turns yields better performance than *logical-xor*. The speedup of *xor*, *neq-xor*, and *logical-xor* is 3.25×, 2.83×, and 1.99×, respectively.

To summarize, defining the `xor(..)` method as `public` would greatly benefit users, who currently have to use alternatives. We suggest the developers of the Java Vector API to consider the inclusion of the `xor(..)` operation in the specification of the Java Vector API. We will open an issue on this matter. Among the alternatives, users can implement the `xor` operation between two masks using the `mask1.not().eq(mask2)` pattern.

5.5 Fused Multiply-Add (FMA) Operation

Nowadays, many processors support fused multiply-add (FMA) instructions for both floating-point scalar and SIMD operations. An `fma(a, b, c)` operation, where `a`, `b`, and `c` can be either floating-point numbers or vectors, is semantically equivalent to the expression `a * b + c`. However, while the expression `a * b + c` involves the execution of two instructions and hence two rounding errors (one for the multiplication and one for the addition), a single FMA instruction evaluates the result rounding only once after the addition. For this reason, an FMA instruction yields a result that is typically closer to the true mathematical result w.r.t. its corresponding expression. The Java Vector API allows exploiting FMA instructions operating on both `float` and `double` numbers via instance methods defined in the classes `FloatVector` and `DoubleVector`, respectively. In this section, to simplify



(a) $M_{AVX_{512}}$ **(b)** M_{AVX_2} **(c)** M_{AVX}

Figure 9. Evaluation of the `fma` pattern.

code and explanation, we focus on `FloatVector`. However, we note that the evaluated benchmarks use both `FloatVector` and `DoubleVector`. The two FMA instance methods operating on `FloatVectors` are reported below:

```
fma(float b, float c);
fma(Vector<Float> b, Vector<Float> c);
```

Values `b` and `c` are provided as parameters, while `a` is the instance of `FloatVector` on which the method is called.

To test the performance of the `fma` operations, we evaluate four different implementations of the benchmarks *axpy*, *lavaMD*, and *somier*. In contrast to the previous sections, we evaluate two vector implementations, namely *fma* and *mul-add*, and two scalar implementations, namely *scalar-fma* and *scalar-mul-add*. *fma* uses the aforementioned methods, i.e., benchmarks perform `a.fma(b, c)`, while *mul-add* uses the corresponding `a.mul(b).add(c)` expression. The scalar implementations *scalar-fma* and *scalar-mul-add* use `Math.fma(a, b, c)` and the expression `a * b + c`, respectively. Experimental results are reported in Figure 9. In particular, we compare *fma* with *scalar-fma* (*scalar-fma/fma* column) and *mul-add* with *scalar-mul-add* (*scalar-mul-add/mul-add* column) to analyze the performance of the Java Vector API w.r.t. a corresponding scalar implementation, i.e., a scalar implementation that maintains the same rounding error (and hence accuracy). In case the different accuracy in the comparison between the two implementations is not a concern, comparing *fma* with *mul-add* (*fma/mul-add* column) allows determining the most efficient vector implementation.

On $M_{AVX_{512}}$ and M_{AVX_2}, versions *fma* and *mul-add* introduce a significant speedup w.r.t. their corresponding scalar versions (except for benchmark *axpy* whose speedups are close to 1). In particular, *fma* introduces a maximum speedup of 14.52× (*lavaMD* on $M_{AVX_{512}}$) while *mul-add* introduces a maximum speedup of 11.96× (*lavaMD* on $M_{AVX_{512}}$). Moreover, *fma* is slightly faster than *mul-add* on all the evaluated benchmarks (except *axpy* on M_{AVX_2}).

However, experimental results on M_{AVX} significantly differ from the ones obtained on both $M_{AVX_{512}}$ and M_{AVX_2}. Due to the limited maximum vector size, version *mul-add* introduces a moderate speedup w.r.t. its corresponding scalar version (1.03×, 1.11×, and 2.56× on *axpy*, *lavaMD*, and *somier*, respectively). Surprisingly, version *fma* introduces a slowdown for all the evaluated benchmarks, indicating that scalar code using `Math.fma` is faster than the corresponding vector version. The slowdown ranges from 0.65× (*lavaMD*) to 0.96× (*somier*) and is motivated by the fact that M_{AVX} does not support the `fma` hardware instructions (*fma* CPU flag). The vector *fma* version executes a Java implementation that loops over the elements of the Java vectors and invokes that scalar `Math.fma` for each element, resulting in additional runtime computation w.r.t. the corresponding scalar version. The *mul-add* version is 230.31×, 941.49×, and 134.69× faster than *fma* on *axpy*, *lavaMD*, and *somier*, respectively. Results suggest to avoid both scalar and vector `fma` APIs on platforms that do not properly support them.

To summarize, when floating-point calculation accuracy is a concern, users may exploit the Java Vector API only if the executing platform supports the *fma* CPU flag. If the executing platform does not support the *fma* CPU flag, depending on the input size and execution frequency of the `fma` operation, a scalar implementation may be preferable. When floating-point calculation accuracy is not a concern, the programmer may consider implementing vector computations using the `a.mul(b).add(c)` pattern. In this way, application code leads to performance speedups regardless of the executing architecture. We suggest to the developers of the API to include performance notes in the documentation.

6 Related Work

To the best of our knowledge, no benchmark suite for the JVM focuses on data parallelization and uses the Java Vector API. SPECjvm2008 [39] exercises computationally intensive workloads. SPECjbb2015 [40] simulates an IT infrastructure of an online supermarket. DaCapo [6] offers complex Java applications with non-trivial memory loads and ScalaBench [38] focuses on Scala programs. Renaissance [34] focuses on data parallelization, contains diversified modern concurrency and parallelism workloads, and is mainly designed for testing JIT compilers. None of these benchmark suites exercises the Java Vector API.

Other languages offer benchmark suites specifically designed to evaluate vectorization. As already detailed in Sec. 3, our benchmark suite is based on the RISC-V Vectorized Benchmark Suite proposed by Ramírez et al. [36], which in turn contains workloads taken from Rodinia [9], PolyBench [42], and ParVec [8] (a vectorized version of the PARSEC [5, 43] suite). VectorBench [27, 37] is a C++ benchmark suite that uses intrinsics for vector instructions. The suite consists of both a scalar and a vectorized version of the benchmarks. Finally, the Test Suite for Vectorizing Compilers (TVSC) [7] is collection of 100 Fortran loops used to test the effectiveness of an automatic vectorizing compiler.

To the best of our knowledge, no related work presents a detailed performance analysis of the Java Vector API. Marneni [25] explores the possibility to add distributed functionalities to ScalaTion [28] and the possibility of using the Java Vector API to write computational intensive applications in Java. Unfortunately, the work does not present an evaluation of the Java Vector API on several architectures, using benchmarks that cover most of the API features, and presenting patterns and anti-patterns. Similarly, Ertl [17] describes the Java Vector API without performing any evaluation.

In managed languages, one prominent approach to develop highly optimized code that exploits explicit vectorization consists in the definition and implementation of low-level native functions. In the case of Java, these native functions are invoked via the Java Native Interface (JNI). Halli et al. [21] report a performance comparison between Java code and optimized native functions, showing how Java can benefit from JNI calls. Stojanov et al. [41] propose an automated and systematic approach to let developers access vector instructions using an embedded domain-specific language (EDSL). Vector instructions are mapped to optimized C kernels invoked via JNI. Our evaluation shows that the Java Vector API enables implementing vectorized code directly in Java without sacrificing the benefits of managed languages. We leave a performance comparison of the Java Vector API with the above approaches as future work.

The most relevant approach to explicit vectorization in Java is the implementation of the Java Vector Interface (JVI) [30] in the JIT baseline compiler Jitrino [20]. Similar to the Java Vector API, JVI enables using vector operations through Java methods that are compiled by Jitrino to hardware vector instructions. Unfortunately, JVI can be used only with the Jitrino compiler and the Apache Harmony VM [19], an Apache project that has reached its end of life.

Other work offers APIs and strategies that allow using vector instructions in both managed and unmanaged languages. In particular, Eidt and Gooding [16] introduces SIMD support for .NET via a high-level library that exploits intrinsics, similar to the Java Vector API. Mono.SIMD [33] is a work-in-progress explicit SIMD API for the Mono C# compiler. McCutchan et al. [26] introduce explicit vector instructions in Dart and Javascript VMs. Boost.SIMD [18] and Intel's Array Building Blocks [29] simplify the usage of SIMD hardware within a standard C++ programming model. Finally, Nuzman et al. [31] propose a split vectorization framework to facilitate portable auto-vectorization across diverse SIMD targets.

7 Concluding Remarks

Limitations. Our analysis focuses on an incubating API of the JDK. This implies that the API may substantially evolve between JDK versions. As detailed in Sec. 4, all experiments were executed on JDK19 (the latest release at time of writing) to take into account the most up-to-date version of the API. We believe that our benchmark suite, analysis and pattern identification may help the developers of the Java Vector API identify performance issues and improve the implementation before the final release.

JVBench includes benchmarks using a wide spectrum of vector types, masks, and API methods, exercising most of the features offered by the API on workloads that are recognized to be representative of vector operations. However, JVBench does not exercise all the features defined in the specification of the Java Vector API. For instance, no benchmark exercises less frequently used vector types, `Shifts/Rotates` operations, and *shape-changing* operations. We plan to expand JVBench to reach full API coverage as part of our future work.

Conclusions and Future Work. In this paper, we analyze performance and portability of the Java Vector API, a new JDK module that allows expressing explicit vector operations through an object-oriented interface. We present JVBench, to the best of our knowledge, the first open-source benchmark suite for the Java Vector API, which features a high API coverage. We use JVBench to evaluate the performance of the Java Vector API, showing that the explicit vectorization enabled by the API greatly improves performance w.r.t. auto-vectorization and scalar code. Moreover, we reported several patterns and anti-patterns that significantly influence runtime performance w.r.t. the executing machine architecture, suggesting API usages and improvements.

As part of our future work, in addition to expanding our suite as mentioned above, we plan to use JVBench to evaluate the performance of the Java Vector API on ARM AArch64 architectures. Moreover, we plan to report our findings to the developer of the Java Vector API.

Data Availability Statement

We provide an artifact that consists of a Docker [22] image embedding JVBench together with a set of tools that can be used to collect, process, and plot performance measurements [4]. The artifact also contains the performance measurements used to generate the figures of the paper.

References

[1] Oracle and/or its affiliates. 2022. Graal implementation of the Vector API. https://github.com/oracle/graal/blob/216a9dcade7e8b7399f0807b1bacb31567af353a/compiler/src/org.graalvm.compiler.hotspot/src/org/graalvm/compiler/hotspot/meta/UnimplementedGraalIntrinsics.java#L327

[2] Oracle and/or its affiliates. 2022. GraalVM. https://www.graalvm.org/

[3] Oracle Corporation and/or its affiliates. 2022. Project Panama: Interconnecting JVM and Native Code. https://openjdk.java.net/projects/panama

[4] Matteo Basso, Andrea Rosà, Luca Omini, and Walter Binder. 2023. Artifact associated to the paper "Java Vector API: Benchmarking and Performance Analysis" published in CC’23. https://doi.org/10.5281/zenodo.7499096 artifact.

[5] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph. D. Dissertation. Princeton University.

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis *(OOPSLA)*. 169–190. https://doi.org/10.1145/1167515.1167488

[7] D. Callahan, J. Dongarra, and D. Levine. 1988. Vectorizing Compilers: a Test Suite and Results *(Supercomputing, Vol. 1)*. 98–105. https://doi.org/10.1109/SUPERC.1988.44642

[8] Juan M. Cebrian, Magnus Jahre, and Lasse Natvig. 2015. ParVec: Vectorizing the PARSEC Benchmark Suite. *Computing* 97, 11 (2015), 1077–1100. https://doi.org/10.1007/s00607-015-0444-y

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[10] IBM Corp. 2022. Eclipse OpenJ9. https://www.eclipse.org/openj9/

[11] IBM Corp. 2022. OpenJ9 implementation of the Vector API. https://github.com/eclipse-openj9/openj9/blob/78038a17db84716d97cb57e1b76bec7975a00bf7/runtime/compiler/optimizer/VectorAPIExpansion.cpp#L2203

[12] Intel Corporation. 2017. Vector Api Writing Own Vector. https://www.intel.com/content/dam/develop/public/us/en/documents/vector-api-writing-own-vector-final-9-27-17.pdf

[13] Intel Corporation. 2018. Java Vector API. https://cr.openjdk.java.net/~vlivanov/talks/2018_JVMLS_VectorAPI.pdf

[14] Intel Corporation. 2022. Intrinsics for Short Vector Math Library Operations (SVML). https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-short-vector-math-library-ops.html

[15] Intel Corporation. 2022. Using Intel AVX without Writing AVX. https://www.intel.com/content/dam/develop/external/us/en/documents/usingavxwithoutwritingavx-183181.pdf

[16] Carol Eidt and Tanner Gooding. 2020. SIMD Support in .NET: Abstract and Concrete Vector Types and Operations *(CGO)*. 229–241. https://doi.org/10.1145/3368826.3377926

[17] M. Anton Ertl. 2018. Software Vector Chaining *(ManLang)*. 1–9. https://doi.org/10.1145/3237009.3237021

[18] Pierre Estérie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Lapresté. 2014. Boost.SIMD: Generic Programming for Portable SIMDization *(WPMVP ’14)*. 1–8. https://doi.org/10.1145/2370816.2370881

[19] The Apache Software Foundation. 2022. Apache Harmony. https://harmony.apache.org/

[20] The Apache Software Foundation. 2022. DRLVM Jitrino Just-in-time Compiler. https://harmony.apache.org/subcomponents/drlvm/JIT.html

[21] Nassim Halli, Henri-Pierre Charles, and Jean-François Méhaut. 2015. Performance Comparison between Java and JNI for Optimal Implementation of Computational Micro-kernels *(ADAPT)*. 7 pages.

[22] Docker Inc. 2023. Docker. https://www.docker.com/

[23] Vladimir Ivanov. 2017. Vectorization in HotSpot JVM. https://cr.openjdk.java.net/~vlivanov/talks/2017_Vectorization_in_HotSpot_JVM.pdf

[24] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets *(PLDI ’00)*. 145–156. https://doi.org/10.1145/358438.349320

[25] Chandana Marneni. 2019. Distributed Computing and JAVA VectorAPI for Performance Optimization for ScalaTion Framework. http://cobweb.cs.uga.edu/~jam/home/theses/chandana_project/Final_Report.pdf

[26] John McCutchan, Haitao Feng, Nicholas Matsakis, Zachary Anderson, and Peter Jensen. 2014. A SIMD Programming Model for Dart, Javascript,and Other Dynamically Typed Scripting Languages *(WPMVP)*. 71–78. https://doi.org/10.1145/2568058.2568066

[27] Charith Mendis, Ajay Jain, Paras Jain, and Saman Amarasinghe. 2019. Revec: Program Rejuvenation through Revectorization *(CC)*. 29–41. https://doi.org/10.1145/3302516.3307357

[28] John A. Miller and the University of Georgia. 2022. ScalaTion Project. http://cobweb.cs.uga.edu/~jam/scalation.html

[29] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. 2011. Intel’s Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language *(CGO ’11)*. 224–235. https://doi.org/10.5555/2190025.2190069

[30] Jiutao Nie, Buqi Cheng, Shisheng Li, Ligang Wang, and Xiao-Feng Li. 2010. Vectorization for Java *(NPC)*. 3–17. https://doi.org/10.1007/978-3-642-15672-4_3

[31] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. 2011. Vapor SIMD: Auto-Vectorize Once, Run Everywhere *(CGO ’11)*. 151–160. https://doi.org/10.5555/2190025.2190062

[32] Paul Sandoz. 2022. JEP 426: Vector API (Fourth Incubator). https://openjdk.org/jeps/426

[33] Mono Project. 2022. Mono Documentation: Mono.Simd Namespace. http://docs.go-mono.com/?link=N%3aMono.Simd

[34] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM *(PLDI)*. 31–47. https://doi.org/10.1145/3314221.3314637

[35] RALC88. 2022. riscv-vectorized-benchmark-suite. https://github.com/RALC88/riscv-vectorized-benchmark-suite

[36] Cristóbal Ramírez, César Alejandro Hernández, Oscar Palomar, Osman Unsal, Marco Antonio Ramírez, and Adrián Cristal. 2020. A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures. *ACM Trans. Archit. Code Optim.* 17, 4 (2020), 1–30. https://doi.org/10.1145/3422667

[37] revec. 2019. VectorBench: Benchmarks for vectorization. https://github.com/revec/VectorBench

[38] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine *(OOPSLA)*. 657–676. https://doi.org/10.1145/2076021.2048118

[39] Kumar Shiv, Kingsum Chow, Yanping Wang, and Dmitry Petrochenko. 2009. SPECjvm2008 Performance Characterization. In *Computer Performance Evaluation and Benchmarking*. 17–35. https://doi.org/10.1007/978-3-540-93799-9_2

[40] spec. 2015. SPECjbb2015. https://www.spec.org/jbb2015/

[41] Alen Stojanov, Ivaylo Toskov, Tiark Rompf, and Markus Püschel. 2018. SIMD Intrinsics on Managed Language Runtimes *(CGO)*. 2–15. https://doi.org/10.1145/3168810
[42] Ohio State University. 2022. PolyBench: The Polyhedral Benchmark Suite. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/
[43] Princeton University. 2022. PARSEC. https://parsec.cs.princeton.edu/index.htm

Received 2022-11-10; accepted 2022-12-19