Analyzing the Vectorization Capability of the Java Vector API

Andrea Rosà, Matteo Basso, Walter Binder
Università della Svizzera italiana (USI), Switzerland
Email: andrea.rosa@usi.ch, matteo.basso@usi.ch, walter.binder@usi.ch

***Abstract*—The *Java Vector API* (JVA) is a novel feature of the Java Virtual Machine (JVM), allowing developers to express vector computations that are automatically translated to vector hardware instructions at runtime. This paper focuses on the vectorization capability of the API, which has not been studied by the literature yet. We investigate how effective the JVA is in automatically vectorizing scalar instructions, comparing it with the auto-vectorization capability of the HotSpot C2 compiler. Our results show that using the JVA results in much fewer (i.e., 79.62% on average on processors supporting *AVX-512*) instructions executed than using C2 auto-vectorization for carrying out the same work.**

***Index Terms*—Java Vector API, vectorization, SIMD, dynamic analysis, Java Virtual Machine.**

I. Introduction

Nowadays, processors support *vector instructions*, i.e., dedicated instructions making use of special hardware (vector and predicate registers) to implement Single Instruction Multiple Data (SIMD) capabilities. For a long time, developers of applications running on the Java Virtual Machine (JVM) lacked official libraries to express vector computations in their code. The use of vector instructions in the compiled code relied on an optimization done by the dynamic compiler called *auto-vectorization*, which recognizes scalar code that can be converted to code using suitable vector instructions as supported by the underlying architecture. Auto-vectorization is useful, but it relies on the ability of the compiler to identify code that can be vectorized.

Since JDK 16, the JVM offers the Java Vector API (henceforth abbreviated in JVA) [1], a feature that enables developers to write explicit vector operations in a Java application, using classes and methods (provided by the JVA) that abstract hardware vector registers and instructions. When a method using the JVA is dynamically compiled, the compiler checks which kind of vector instructions are supported by the underlying processor, emitting machine code that uses vector registers, vector instructions, and predicate registers according to the hardware support of the processor. If the underlying architecture does not support some of the vector operations expressed through the JVA, the JVM falls back to a default scalar implementation to ensure graceful degradation.

While the JVA is increasingly being used, several aspects of the JVA have not been studied yet. In this paper, we focus on the capability of the JVA to vectorize code, studying the number of vector instructions executed when using the JVA and how much the JVA can reduce the total (scalar and vector) amount of instructions executed for carrying out the same work (as multiple scalar instructions can be replaced with a single vector instruction). These metrics are very useful in indicating the effectiveness of the JVA in automatically converting scalar code to vector operations. We perform this study on JVBench [2], which is the first publicly released benchmark suite targeting the JVA and is composed of 10 applications exercising different features of the JVA. We analyze the JVA on three different architectures, each supporting different vector capabilities—particularly, the Intel Advanced Vector Extensions (AVX) [3], versions *AVX*, *AVX2* and *AVX-512*—and we compare the results with the auto-vectorization performed by the HotSpot C2 compiler [4] (i.e., the dynamic compiler shipped with OpenJDK). Our results show that using the JVA results in significantly more vector instructions executed than auto-vectorization, while notably reducing the total amount of instructions executed, i.e., by 20.85%, 38.26%, and 79.62% on average w.r.t. auto-vectorization when the processor supports *AVX*, *AVX2* and *AVX-512*, respectively.

II. Experimental Methodology

Profiler for Detecting Executed Vector Instructions. To study the vectorization capability of the JVA, we develop a new profiler resorting to binary instrumentation via *pin* [5]. The tool detects and counts the execution of every vector instruction (without using sampling) defined in the Intel Instruction Set Extensions [3], including those defined by Streaming SIMD Extensions (SSE, version 1 to 4), *AVX*, *AVX2* and *AVX-512*, which are used by the JVA when performing the vectorization.

Many benchmark suites for the JVM (including JVBench) rely on the Java Microbenchmarking Harness (JMH) [6] to structure workload execution into multiple independent *iterations*. Our profiler separates the collected data per iteration, allowing us to exclude phases that are typically ignored for measurements (e.g., JVM initialization, benchmark setup and tear-down), as we did for collecting the data presented in Sec. III.

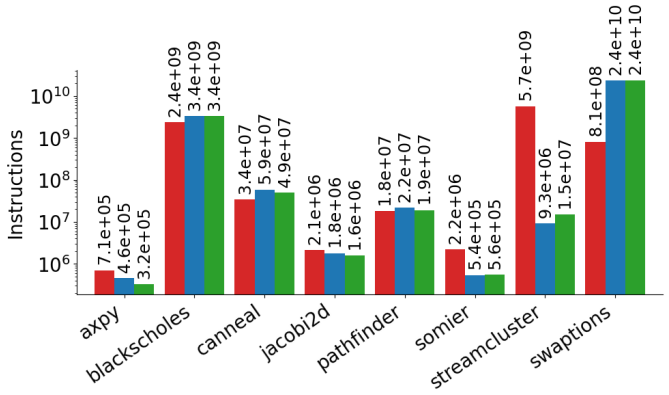
Evaluation Settings. We perform our analysis on three different machines (M_{AVX}, M_{AVX2}, $M_{AVX\text{-}512}$). M_{AVX} is equipped with two NUMA nodes, each with an 8-core Intel Xeon E5-2680 (2.7 GHz) and 64 GB of RAM. This processor only supports the oldest version of *AVX*. M_{AVX2} has a 16-core Intel i9-10885H (2.40 GHz) with 32 GB of RAM, and supports *AVX2*. $M_{AVX\text{-}512}$ is equipped with an 18-core Intel i9-10980XE (3.00 GHz) with 256 GB of RAM and supports the latest

TABLE I: Summary (average) of results on M_{AVX2}.

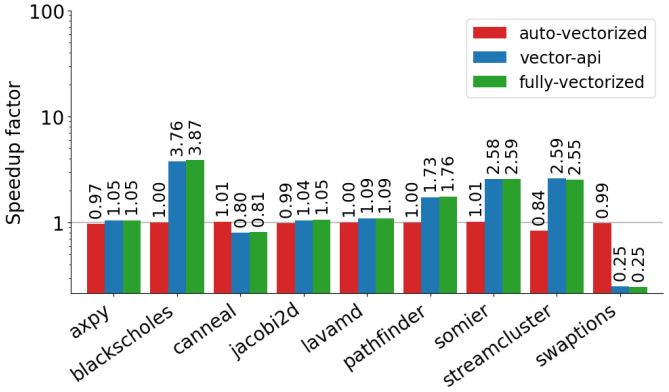
	# vector instructions executed	*exec. time speedup*	*% total instr reduced w.r.t. auto-vectorization*
auto-vectorized	$1.73 \cdot 10^9$	1.02	-
vector-api	$3.54 \cdot 10^9$	3.41	38.26
fully-vectorized	$3.55 \cdot 10^9$	3.42	-

version of *AVX* (*AVX-512*). The three machines reflect different technologies, vector sizes (128, 256, and 512 for *AVX*, *AVX2* and *AVX-512*, respectively) and vectorization potential, ensuring that our study has a reasonable hardware and platform coverage. All processors support the latest version of SSE (v4.2). We perform our experiments on OpenJDK 21, (the latest long-term support release at the time of writing), which uses HotSpot C2 [4] as a last-tier dynamic compiler. Our target workloads are all benchmarks from the latest release of JVBench (v1.0.1). We note that on *AVX*, some of the analyses timed out on two benchmarks, as also experienced by the creators of JVBench [7]. We omit the corresponding results in Fig. 1 and Table II.

In addition to studying the vectorization capability of the JVA, we aim to compare it with the auto-vectorization performed by C2. For this reason, we run our analyses on three different benchmark versions: *auto-vectorized* (i.e., we run an equivalent *scalar* version of the workload, written in Java and provided in the JVBench release, enabling C2 auto-vectorization); *vector-api* (i.e., we run the benchmark, implemented using the JVA, with C2 auto-vectorization disabled); *fully-vectorized* (i.e., same as *vector-api*, but with C2 auto-vectorization also enabled). When reporting the execution-time speedups (Fig. 1b and 2b), we also run a fourth benchmark version, *scalar* (i.e., the scalar version of the workload provided by JVBench, but disabling C2 auto-vectorization), that we use as baseline.



(a) Number of executed vector instructions



(b) Execution-time speedup factor

Fig. 1: Evaluation results on M_{AVX}.

III. Experimental Results

We present our analysis in Fig. 1 (M_{AVX}), Fig. 2 ($M_{AVX\text{-}512}$) and Table II. Due to lack of space, we do not show the data collected on M_{AVX2} (but we report the average results obtained across all benchmarks in Table I). We prioritize discussing results on M_{AVX} over M_{AVX2} as the JVA performs significantly differently (and worse) on M_{AVX}, as also noted by related work [7]. All shown data represents the average of the execution of 10 steady-state benchmark iterations. In the figures, the exact mean value is reported above each bar, while the black error bars represent the 95% confidence intervals of the measurements (which are very small in all benchmarks, a sign of the stability of the collected metrics). Fig. 1a and 2a show the number of vector instructions executed in each benchmark version, collected with our profiler. Fig. 1b and 2b show the *speedup factor* (in terms of execution time) of the different benchmark versions w.r.t. *scalar*. We include this metric to correlate our results with an intuitive performance metric. We do not apply our profiler when measuring the speedup.

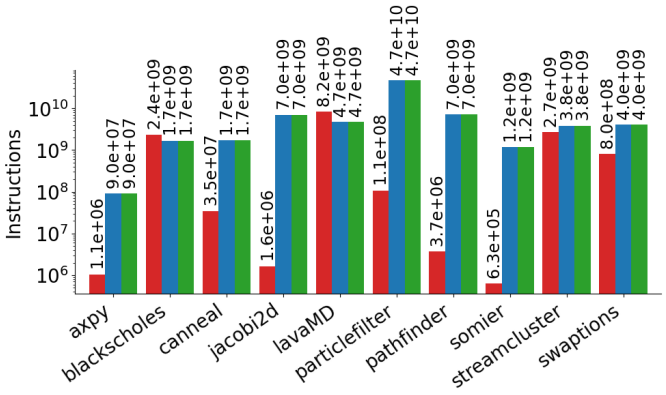
Regarding M_{AVX}, the number of vector instructions executed via auto-vectorization and via the JVA is of the same order of magnitude in several benchmarks (axpy, blackscholes, canneal, jacobi2d, pathfinder, Fig. 1a). In swaptions the JVA leads to the execution of many more vector instructions ($2.4 \cdot 10^{10}$) than via auto-vectorization ($8.1 \cdot 10^8$), while the situation is opposite in somier and particularly in streamcluster (where auto-vectorization and the JVA differ by three orders of magnitude), The average across all benchmarks is $1.12 \cdot 10^9$ for auto-vectorization and $3.44 \cdot 10^9$ for the JVA—using the JVA results in $3.07\times$ more executed vector instructions.

As Fig. 1b shows, the JVA leads to better performance than auto-vectorization (average speedup factor of $1.65\times$, while auto-vectorization leads to no noticeable difference in terms of performance, i.e., $0.98\times$), with a peak in blackscholes ($3.76\times$). Interestingly, the JVA performs poorly (causing slowdowns) in canneal ($0.80\times$) and swaptions ($0.25\times$). These slowdowns are caused by the fact that the vector operations dominating such benchmarks (e.g., masked and vector-manipulation operations) are not supported by *AVX* (manually verified), and therefore the execution falls back to the default scalar implementation. Using the JVA together with auto-vectorization (version *fully-vectorized*) leads to a further minor performance increase ($1.67\times$, compared to $1.65\times$ when using only the JVA). A possible explanation is that code in the Java Class Library has not yet been recast to make use of the JVA, but auto-vectorization can vectorize such code.

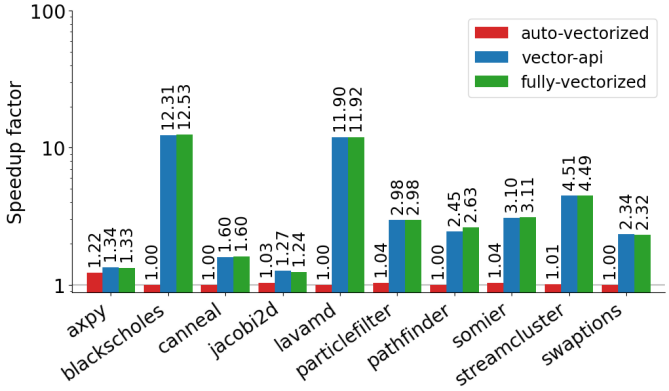
Table II reports the difference between the total (scalar and

TABLE II: Difference between the total instructions executed via auto-vectorization and the JVA.

	M_{AVX}		$M_{AVX\text{-}512}$	
Benchmark	*Difference*	*% w.r.t. auto-vect.*	*Difference*	*% w.r.t. auto-vect.*
axpy	$6.67 \cdot 10^{7}$	2.82	$5.92 \cdot 10^{8}$	21.52
blackscholes	$7.28 \cdot 10^{10}$	67.31	$1.03 \cdot 10^{11}$	92.59
canneal	$-4.98 \cdot 10^{10}$	−102.45	$5.20 \cdot 10^{9}$	11.89
jacobi2D	$9.75 \cdot 10^{10}$	46.43	$2.01 \cdot 10^{11}$	87.81
lavaMD	-	-	$3.13 \cdot 10^{11}$	93.57
particlefilter	-	-	$3.17 \cdot 10^{11}$	57.72
pathfinder	$3.76 \cdot 10^{11}$	62.59	$5.96 \cdot 10^{11}$	96.65
somier	$1.88 \cdot 10^{11}$	85.56	$1.77 \cdot 10^{11}$	84.26
streamcluster	$1.14 \cdot 10^{11}$	50.06	$1.52 \cdot 10^{11}$	64.58
swaptions	$-4.65 \cdot 10^{11}$	−258.42	$1.44 \cdot 10^{11}$	75.64



(a) Number of executed vector instructions



(b) Execution-time speedup factor

Fig. 2: Evaluation results on $M_{AVX\text{-}512}$.

vectorial) instructions executed using auto-vectorization and the JVA. Positive values indicate that the JVA leads to the execution of fewer instructions, which is an improvement (less instructions are needed to carry out the same work). With the JVA, the total number of instructions executed is significantly lower than with auto-vectorization, with the exception (for M_{AVX}) of canneal and swaptions (which is expected due to the reasons explained in the previous paragraph). The maximum reduction of executed instructions w.r.t. auto-vectorization is 85.56% (somier) while the average is 20.85% on M_{AVX}.

The vectorization capabilities of the JVA shown on M_{AVX} are magnified on $M_{AVX\text{-}512}$, where the JVA leads to the execution of many more vector instructions than auto-vectorization ($7.82 \cdot 10^{9}$ on average, Fig. 2a), while auto-vectorization stops at $1.43 \cdot 10^{9}$ (using the latter results in more vector instructions executed than the JVA only in blackscholes and lavaMD). Auto-vectorization enables an average speedup (Fig. 2b) of $1.03\times$, particularly in axpy ($1.22\times$, due to the fact that axpy does not introduce many aliasing references or dependencies between loop iterations which would complicate auto-vectorization), indicating that auto-vectorization offers a slightly better support for *AVX-512* than for *AVX*. However, the JVA leads to an average performance gain of $4.38\times$, much superior to the one obtainable on M_{AVX} ($1.65\times$). Also on $M_{AVX\text{-}512}$, combining auto-vectorization with the JVA leads to slightly better performance ($4.42\times$ on average). Finally, Table II shows that the JVA is very effective in reducing the amount of total executed instructions if *AVX-512* is supported, resulting in a maximum reduction of executed instructions (w.r.t. auto-vectorization) of 96.65% (pathfinder) and an average of 79.62%.

In summary, our results show that the JVA has a significant vectorization capability, superior to the one of C2 auto-vectorization, particularly on processors offering modern dedicated hardware for vector computations. We note that on average, the JVA reduces the amount of instructions executed (w.r.t. auto-vectorization) by 20.85% on M_{AVX}, by 38.26% on M_{AVX2} and by 79.62% on $M_{AVX\text{-}512}$; as expected, these reductions are proportional to the different vector sizes of the three processors (128, 256 and 512, respectively).

IV. Conclusions

This paper highlights that using the JVA results in significantly more vector instructions executed than C2 auto-vectorization as well as a much lower amount of total (scalar and vector) instructions executed for performing the same work (79.62% on average w.r.t. C2 auto-vectorization on processors supporting *AVX-512*).

Acknowledgments

We thank Thomas Bertini for his work on the profiler and its evaluation. The research presented in this paper was supported by Oracle (ERO project 1332) and by the Swiss National Science Foundation (project 200020_188688).

References

[1] "JEP 469," https://openjdk.org/jeps/469, 2024.
[2] "JVBench Benchmark Suite," https://github.com/usi-dag/JVBench, 2024.
[3] "Intel® Instruction Set Extensions Technology," https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html, 2024.
[4] M. Paleczny, C. Vick, and C. Click, "The Java HotSpot Server Compiler," in *Java Virtual Machine Research and Technology Symposium*, 2001.
[5] Intel, "Pin - A Dynamic Binary Instrumentation Tool," https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html, 2024.
[6] OpenJDK, "Java Microbenchmark Harness (JMH)," https://github.com/openjdk/jmh, 2024.
[7] M. Basso, A. Rosà, L. Omini, and W. Binder, "Java Vector API: Benchmarking and Performance Analysis," in *CC*, 2023, p. 1–12.