Java Vector API: Benchmarking and Performance Analysis

Matteo Basso, **Andrea Rosà, Luca Omini, Walter Binder**

Università della Svizzera italiana, Switzerland

CC 2023
February 25, 2023

Introduction

- Java Vector API
  - Included in the Java Class Library since Java 16
  - Explicit vector (SIMD) operations using an object-oriented Java API
- High performance
  - Runtime compilation of vector operations to hardware vector instructions
- Portability
  - Explicit vectorization without renouncing the advantages of Java as a high-level programming language

Introduction

- ➢ Novel incubating API
- ➢ There is no study evaluating the performance of the Java Vector API
- ➢ There is no realistic benchmark that uses the Java Vector API
- ➢ Existing work
  - Explores the possibility of using the Java Vector API
  - Describes the Java Vector API without performing a detailed evaluation [1]

[1] M. Anton Ertl, "Software Vector Chaining." ManLang 2018.

Contributions

- We design and develop JVBench [1], the first open-source benchmark suite extensively exercising the Java Vector API
  - Realistic workloads resulting in high API coverage
- We use JVBench to evaluate the performance of the Java Vector API w.r.t. other semantically equivalent implementations
  - Scalar implementation
  - Auto-vectorized implementation
- We identify four patterns and anti-patterns on the use of the Java Vector API significantly affecting application performance

[1] https://github.com/usi-dag/JVBench

Background - Java Vector API

- Functional but not optimal Java implementation
  - Executed before that Just-In-Time (JIT) compilation occurs
  - Executed when the underlying platform does not support some of the requested vector features
- At runtime, the JIT compiler emits machine code that uses the supported vector registers and vector instructions
  - Removing the abstraction of the object-oriented API
- Execution of applications exercising the Java Vector API even on platforms that do not support some vector operations

JVBench - Benchmarks

Benchmark Name	Application Domain	Algorithmic Model
axpy	High Performance Computing	BLAS
blackscholes	Financial Analysis	Dense Linear Algebra
canneal	Engineering	Unstructured Grids
jacobi2d	Engineering	Dense Linear Algebra
lavaMD	Molecular Dynamics	N-Body
particlefilter	Medical Imaging	Structured Grids
pathfinder	Grid Traversal	Dynamic Programming
somier	Physics Simulation	Dense Linear Algebra
streamcluster	Data Mining	Dense Linear Algebra
swaptions	Financial Analysis	MapReduce Regular

- ➢ Evaluate the performance of the Java Vector API on diversified benchmarks
- ➢ Benchmarks well-established in the literature [1]

[1] C. Ramírez et al., "A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures." TACO 2020.

JVBench - API Coverage

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			✓		✓				✓	✓

- ➢ Classification of the vector operations as reported by related work [1]
- ➢ High API Coverage

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

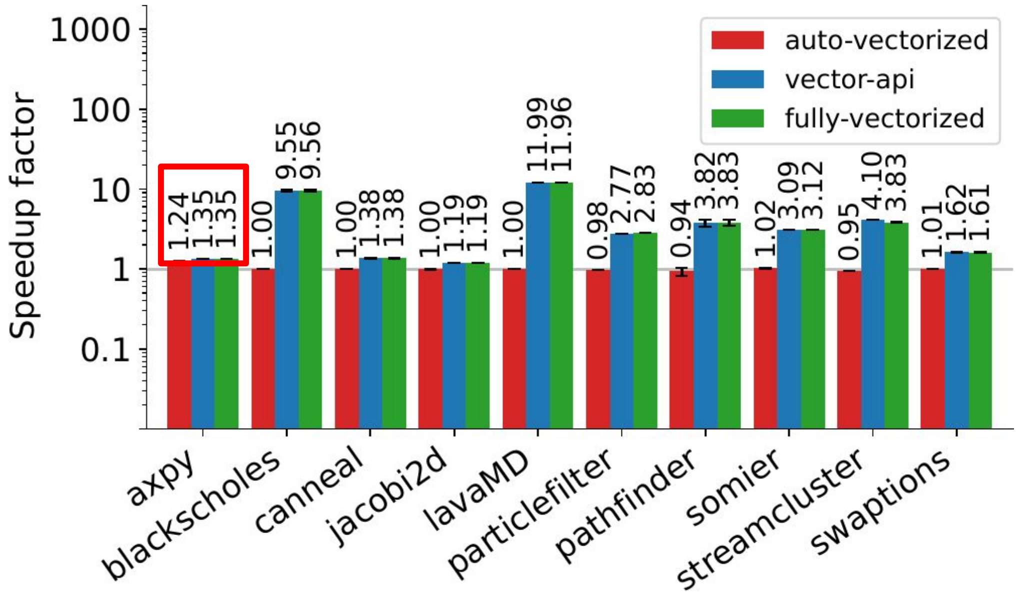
Java Vector API Evaluation

- Evaluation of the performance of the Java Vector API w.r.t. other semantically equivalent implementations
- We conduct our experiments using OpenJDK 19 and the HotSpot C2 JIT compiler
- We run our experiments on three different machines:
  - M_{AVX}: *sse** and *avx* Intel-defined CPU flags (`VectorShape` of length 128 bits)
  - M_{AVX2} : *sse**, *avx*, *fma*, and *avx2* Intel-defined CPU flags (`VectorShape` of length 256 bits)
  - M_{AVX512} : *sse**, *avx*, *fma*, *avx2*, and *avx512* Intel-defined CPU flags (`VectorShape` of length 512 bits)

Java Vector API Evaluation

- We evaluate four different versions of each JVBench benchmark
  - **scalar (baseline)**: no vectorization, no auto-vectorization
  - **auto-vectorized**: auto-vectorization
  - **vector-api**: Java Vector API, no auto-vectorization
  - **fully-vectorized**: Java Vector API, auto-vectorization
- We collect 10 steady-state measurements for each benchmark

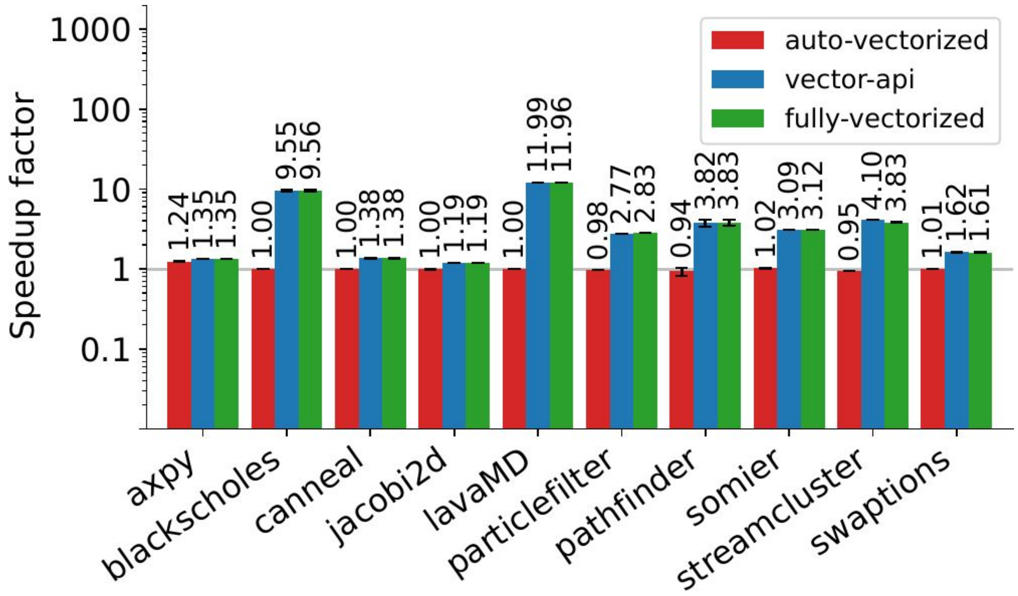
Java Vector API Evaluation



M_{AVX512}

- Auto-vectorization offers only poor performance improvements
- *axpy* is the only effectively auto-vectorized benchmark

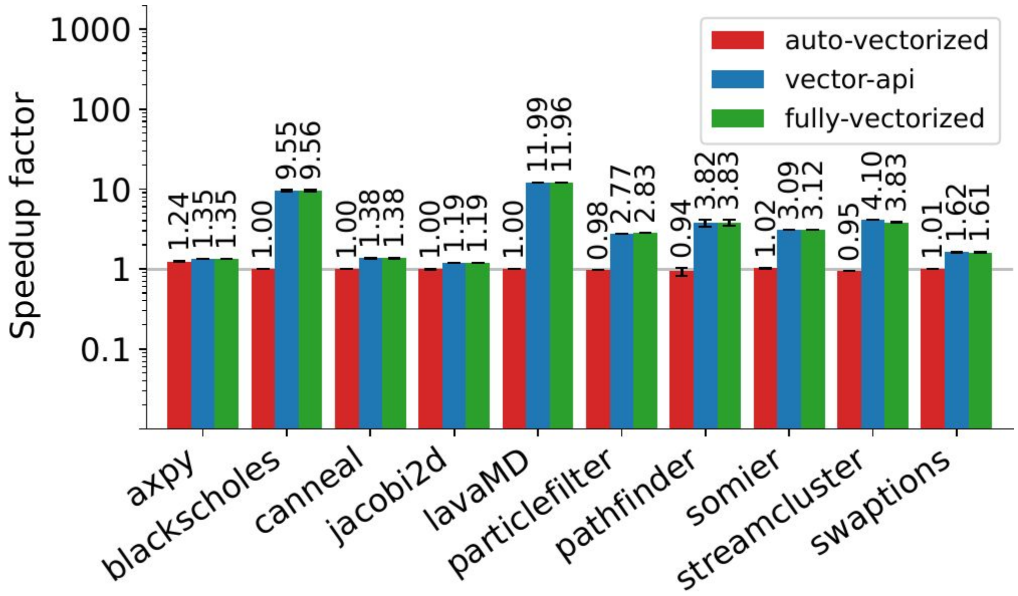
Java Vector API Evaluation



M_{AVX512}

- The Java Vector API is instead effective
  - Speedup factors up to 11.99×
  - On M_{AVX512}, 2.98× on average (geomean)

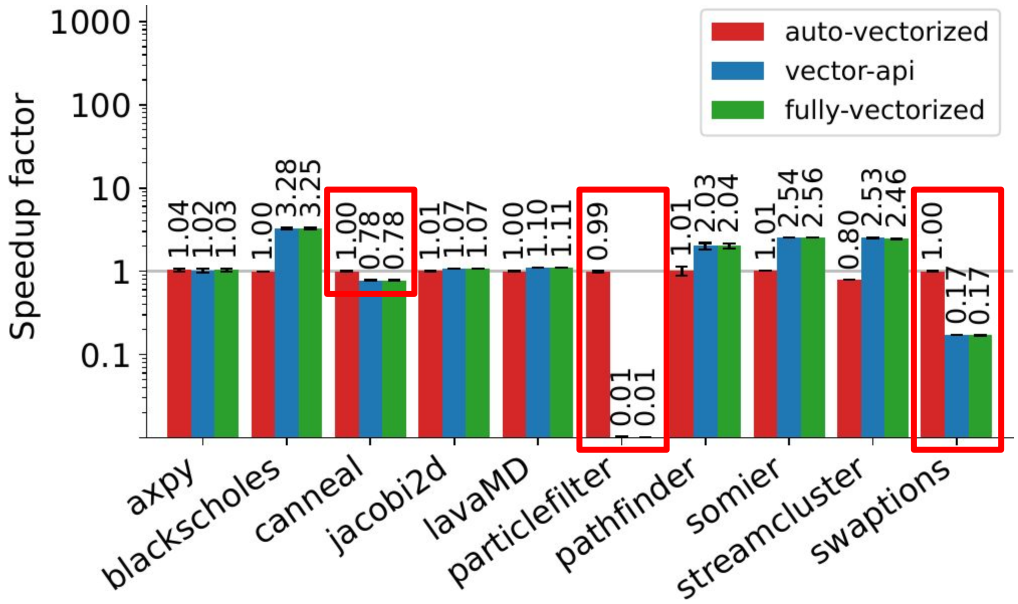
Java Vector API Evaluation



M_{AVX512}

- ➢ No significant difference between the vector-api and the fully-vectorized versions
- ➢ The compiler auto-vectorization does not interfere with the Java Vector API

Java Vector API Evaluation



M_{AVX}

- Poor performance on M_{AVX} for benchmarks *canneal*, *swaptions*, and *particlefilter*
  - Usage of masked operations
  - Execution of the Java implementation of the Vector API

Java Vector API Evaluation - Summary

- Auto-vectorization offers only poor performance improvements
- The Java Vector API yields speedup factors up to 11.99×
- On old machines, the Java Vector API introduces a slowdown w.r.t. an equivalent scalar implementation

Patterns and Anti-Patterns

- Performant API usages and semantically equivalent less performant API usages, respectively
- We analyze four different patterns/anti-patterns:
  - `loopBound` and `indexInRange`
  - Transcendental and Trigonometric Lane-Wise Operations
  - Xor Operation
  - Fused Multiply-Add (FMA) Operation

Patterns and Anti-Patterns - indexInRange

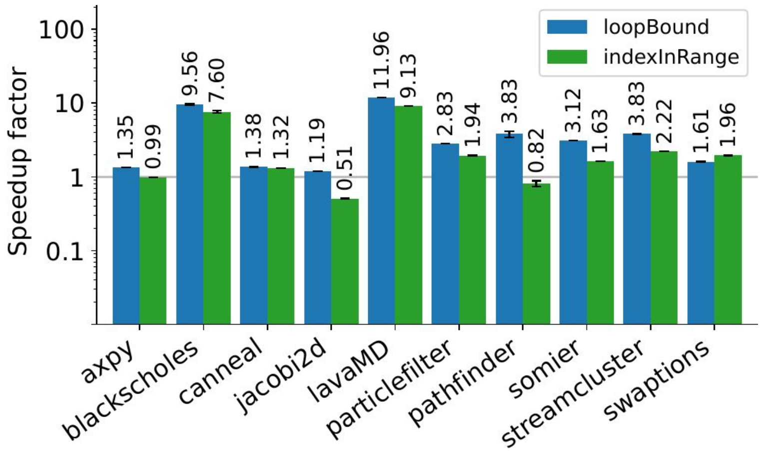
loopBound

```
static final VectorSpecies<Integer> SPECIES =
    IntVector.SPECIES_MAX;

void vectorAdd(int[] a, int[] b, int[] c) {
  int i = 0;
  int limit = SPECIES.loopBound(a.length);

  for (; i < limit; i += SPECIES.length()) {
    IntVector vA = IntVector.fromArray(SPECIES, a, i);
    IntVector vB = IntVector.fromArray(SPECIES, b, i);
    vA.add(vB).intoArray(c, i);
  }

  for (; i < a.length; i++) {
    c[i] = a[i] + b[i];
  }
}
```

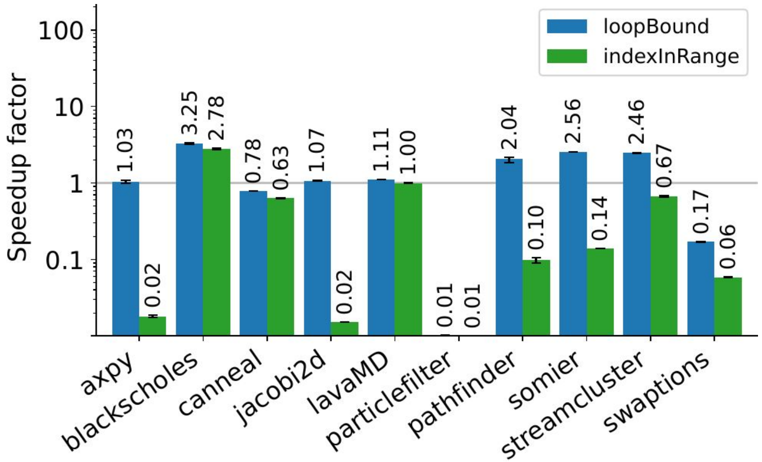
indexInRange

```
static final VectorSpecies<Integer> SPECIES =
    IntVector.SPECIES_MAX;

static void vectorAdd(int[] a, int[] b, int[] c) {
  for (
    int i = 0;
    i < a.length;
    i += SPECIES.length()
  ) {
    VectorMask<Integer> mask =
        SPECIES.indexInRange(i, a.length);
    IntVector vA =
        IntVector.fromArray(SPECIES, a, i, mask);
    IntVector vB =
        IntVector.fromArray(SPECIES, b, i, mask);
    vA.add(vB).intoArray(c, i, mask);
  }
}
```

Patterns and Anti-Patterns - indexInRange



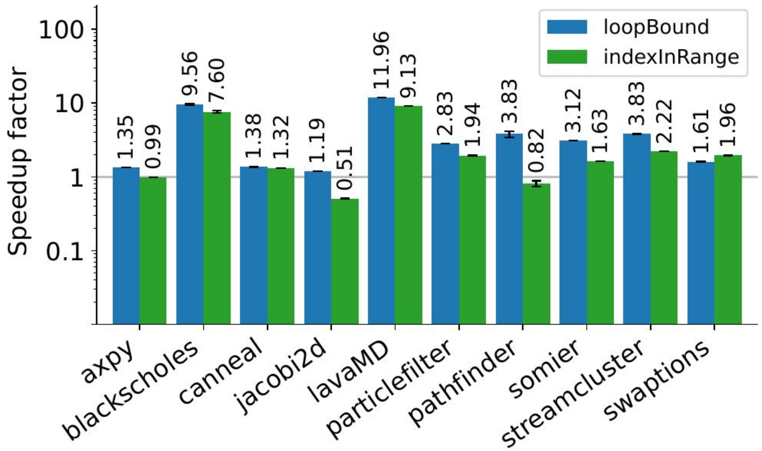
M_{AVX512}



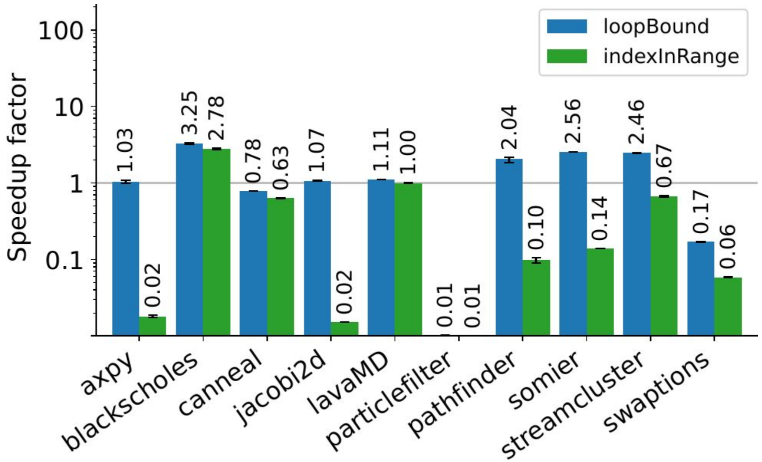
M_{AVX}

- The `loopBound` method achieves better performance
- Performance degradation when using masked operation on architectures that do not support them

Patterns and Anti-Patterns - indexInRange



M_{AVX512}



M_{AVX}

- ➢ Usage of `loopBound` to implement portable code that does not lead to performance degradation
  - Development of third-party Java libraries

Discussion

- Our analysis focuses on an incubating API of the JDK
  - JVBench may help the developers of the Java Vector API improve the implementation before the final release
  - JVBench may help compiler developers improving auto-vectorization
- JVBench includes benchmarks using a wide spectrum of vector types, masks, and API methods
  - JVBench does not exercise all the features defined in the specification of the Java Vector API
  - Expand the API Coverage as part of our future work

Conclusions

- We presented JVBench, the first open-source benchmark suite for the Java Vector API
- We used JVBench to evaluate the performance of the Java Vector API
  - The explicit vectorization enabled by the API greatly improves performance w.r.t. auto-vectorization and scalar code
- We reported four patterns and anti-patterns that significantly influence runtime performance

Thanks for your attention

- ➢ JVBench repository: https://github.com/usi-dag/JVBench
- ➢ JVBench artifact
  - Docker image: https://zenodo.org/record/7499096
  - Source code: https://github.com/usi-dag/JVBench-artifact
- ➢ Contacts:

Matteo Basso

matteo.basso@usi.ch