Optimizing Java Performance with the Vector API for Data-Parallel Computations

V. Pavlović*, K. Širić*, B. Mihaljević*, A. Radovan †
* RIT Croatia, Zagreb, Croatia
† Algebra University College, Zagreb, Croatia
viktor.pavlovic@mail.rit.edu, karlo.siric@mail.rit.edu,
branko.mihaljevic@croatia.rit.edu, aleksander.radovan@algebra.hr

Abstract— **The Java Vector API efficiently optimizes data-parallel operations using SIMD (Single Instruction, Multiple Data), making it particularly useful for applications that rely on vector-based numerical computations, such as simulations, machine learning, and data analytics. This paper analyzes the Vector API's design, supported features, and potential use in performance-critical domains such as game engines, data analytics, and scientific simulations.**

We analyzed the Vector API with practical test cases to measure its performance, focusing on features such as supported data types, operational flexibility, and the way older CPUs without appropriate SIMD support affect its efficiency. Our preliminary results demonstrate notable improvements in the performance of parallelizable tasks while also identifying limitations, including reliance on hardware SIMD support and reduced efficiency when fallback mechanisms are employed.

The Vector API demonstrates how Java can leverage hardware-accelerated computations, allowing software engineers to achieve significant performance gains without requiring the use of low-level programming languages. The findings suggest that the Vector API effectively bridges the gap between hardware-accelerated computations and Java's high-level programming model, providing a viable path for optimizing performance in specific use cases without requiring low-level programming expertise.

Keywords—Vector API, SIMD, Java, Data parallel operations

I. Introduction

The growing demand for high-performance computing in Java applications, such as scientific computing, multimedia processing, and machine learning, has prompted the development of features that enable explicit hardware acceleration. Traditionally, Java application development relied on JNI calls to leverage CPU-specific optimizations—a somewhat cumbersome process opposing Java's "*write once, run anywhere*" philosophy and introducing potential maintenance overhead for multi-platform deployments. In 2018, the introduction of the Vector API presented a modern approach to bridging that gap, allowing developers to write data-parallel operations directly in Java while still accessing SIMD (Single Instruction, Multiple Data) instructions on various hardware architectures.

In this paper, we aimed to examine the design of the recent implementation of Vector API, discuss its performance characteristics, and evaluate its suitability for different practical scenarios. The "Vector API (Incubator)" was first proposed [1] as incubating API in JDK Enhancement Proposal (JEP) 338 in Java Development Kit (JDK) 16, and later in JEP 414 (in JDK 17), JEP 417 (in JDK 18), JEP 426 (in JDK 19), JEP 438 (in JDK 20), JEP 448 in (JDK 21), JEP 460 (in JDK 22), and JEP 469 (in JDK 23). Most recently, JEP 489, named "Vector API (Ninth Incubator)," is to be delivered in JDK 24 (Q1 2025) with several new additions and functional changes. Because the Vector API has undergone multiple incubator phases, we focused in our research on the relatively stable version of the Vector API (Sixth Incubator) delivered ad JEP 448 in the JDK 21 (LTS version) to ensure the stability of our tests.

For performance analysis on practical use cases, we selected three benchmark programs—matrix multiplication, Fast Fourier Transform (FFT), and image processing—to illustrate both the potential performance gains and the practical challenges of using the API.

We also address limitations and fallback mechanisms when SIMD hardware support is absent or only partial, highlighting the trade-offs involved in writing explicitly vectorized Java code. We present how fallback logic behaves when the CPU lacks a newer Advanced Vector Extensions (AVX) instruction set, such as AVX-512 or AVX2, and relies only on older SSE (Streaming SIMD Extensions), such as SSE4.2, revealing the API's ability to degrade gracefully, although not always optimally.

II. Related Work

The introduction of the Java Vector API has enabled developers to leverage hardware-accelerated SIMD operations, bringing explicit vectorization capabilities into a high-level programming environment. As the API evolves, research has been conducted to evaluate its performance, analyze its effectiveness in comparison to existing vectorization strategies, and explore its applications in distributed computing and automatic vectorization. In this section, we present a review of several relevant studies that examine different aspects of the Java Vector API.

Marneni [2] explores the integration of the Java Vector API into distributed computing frameworks, particularly in the context of optimizing matrix multiplication in the ScalaTion framework. These early results demonstrate that SIMD acceleration via the Vector API can yield significant performance improvements over traditional distributed approaches, reducing computational overhead without requiring JNI-based optimizations. While their study primarily focuses on distributed systems, it provides valuable insights into the potential of the Java Vector API for large-scale computational workloads.

Another research from Nie et al. [3] introduced two complementary approaches to vectorization in Java: a Java Vectorization Interface (JVI) and an automatic vectorization strategy. Their work predates the Java Vector API but addresses similar concerns regarding Java's ability to utilize SIMD instructions efficiently. Their JVI provides an abstraction layer for vector operations, while their automatic vectorization approach attempts to generate vectorized code without developer intervention. The findings from their study emphasize the challenges of automatic vectorization, particularly in memory bandwidth constraints and loop optimization strategies, which remain relevant in our research of Java's explicit vectorization model.

Basso et al. [4] systematically compared vectorized Java implementations against scalar and auto-vectorized Java code, providing insights into performance trends and best practices. They presented JVBench, the open-source benchmark suite designed to evaluate the Java Vector API's performance across different hardware architectures. The authors identify key factors affecting performance, including vector instruction set support and JIT compilation optimizations. Their findings align well with our research, reinforcing that while the Java Vector API delivers significant advancement over scalar implementations, its effectiveness depends on hardware compatibility. However, our research explored additional practical applications, such as scientific computing and image processing.

Moreover, in Rosà et al. [5], the same authors compared the Java Vector API efficiency against the HotSpot C2 compiler's auto-vectorization. Their analysis reveals that using explicit vectorization via the API leads to a substantial reduction in the number of executed instructions, particularly on AVX-512-enabled processors, where the API achieves up to a 79.62% reduction compared to auto-vectorized code. This highlights the explicit control the API provides over vectorization, a factor we also observe in our performance evaluation. However, while they focus on low-level vector instruction execution, our research considers both micro-benchmark performance and broader application-level challenges.

Finally, Löff et al. [6] proposed replacing traditional vectorized intrinsics with pure Java code leveraging the Java Vector API, aiming to maintain performance while improving portability and maintainability. They evaluated this approach on core Java Class Library functions such as hashCode and equals, demonstrating that explicit vectorization via the API can achieve performance on par with manually optimized intrinsics. This research is closely related to our study, as it reinforces the potential of the Vector API to serve as a viable alternative to native SIMD intrinsics, reducing the dependency on platform-specific optimizations.

Based on these findings, our research aimed to analyze the performance gains of the more recent version of Vector API and how it performs in numerically intensive benchmarks such as matrix multiplication, FFT computation, and image processing, which cover common practical application use cases.

III. BENCHMARKING

In this section, we describe the implementation of our benchmark programs using the Java Vector API. All benchmarks assume JDK 21 (or above) for Vector API support. While these examples are simplified for clarity, they demonstrate how vector operations can be used for substantial performance gains in arithmetic-heavy tasks.

A. *Matrix Multiplication*

Matrix multiplication is a common, computation-intensive operation in scientific computing, game development (for transformations), and data analytics, which could be implemented as presented in Fig. 1, and we used it as our first benchmark.

```
final VectorSpecies<Float> SP =
  FloatVector.SPECIES_256;
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++) {
      float sum = 0f;
      int k = 0;

      for (; k + SP.length() <= N;
         k += SP.length()) {
         FloatVector aVec =
           FloatVector.fromArray(SP, A, i*N + k);
         FloatVector bVec = FloatVector.zero(SP);

         for (int idx = 0; idx < SP.length();
           idx++) {
            bVec = bVec.add(B[(k + idx)*N + j]);
         }
         FloatVector mul = aVec.mul(bVec);
         sum +=
           mul.reduceLanes(VectorOperators.ADD);
      }

      for (; k < N; k++) {
         sum += A[i*N + k] * B[k*N + j];
      }

      C[i*N + j] = sum;
   }
}
```

Figure 1. Source code for the Matrix Multiplication benchmark

This approach highlights the need for partial vector loads (or masked operations) if the matrix dimension is not divisible by the vector length. In practice, developers may adopt block tiling for better cache efficiency, but this snippet suffices to illustrate vector usage.

B. *FFT Computation*

The Fast Fourier Transform (FFT) operation is central to many digital signal-processing tasks. Although the full implementation of an FFT can be intricate, Fig. 2 presents a simplified snippet illustrating how vector operations can speed up part of the computation (e.g., elementwise "butterfly" operations in a Cooley–Tukey FFT) which we used in our second benchmark.

```
public static void butterflyOp(float[] real,
  float[] imag, int offset, int step) {

   final VectorSpecies<Float> SPEC =
     FloatVector.SPECIES_256;
   FloatVector r1 =
     FloatVector.fromArray(SPEC, real, offset);
   FloatVector i1 =
     FloatVector.fromArray(SPEC, imag, offset);
   FloatVector r2 =
     FloatVector.fromArray(SPEC, real,
     offset + step);
   FloatVector i2 =
     FloatVector.fromArray(SPEC, imag,
     offset + step);
   FloatVector tmpR = r1.sub(r2);
   FloatVector tmpI = i1.sub(i2);
   tmpR.intoArray(real, offset + step);
   tmpI.intoArray(imag, offset + step);
   (r1.add(r2)).intoArray(real, offset);
   (i1.add(i2)).intoArray(imag, offset);
}
```

Figure 2. Source code for the Fast Fourier Transform benchmark

Typically, butterflyOp() would be called multiple times at different offsets/steps to complete the FFT pass. Even though this snippet is simple, it demonstrates how partial steps in the FFT can be accelerated via vector loads and stores.

C. *Image Processing*

Image-processing workloads (e.g., applying filters, adjusting brightness/contrast, and similar operations) often perform the arithmetic on millions of pixels. Fig. 3 presents a typical brightness adjustment benchmark used in our third benchmark.

```
final ByteVector.Species SPEC =
  ByteVector.SPECIES_128;

for (int i=0; i<image.length; i+=SPEC.length()) {
   ByteVector pixelVector =
     ByteVector.fromArray(SPEC, image, i);
   FloatVector floatPixels =
     pixelVector.reinterpretAsFloats();
   FloatVector brightened =
     floatPixels.mul(1.2f).max(0.0f).min(255.0f);
   ByteVector result =
     brightened.reinterpretAsBytes();
   result.intoArray(image, i);
}
```

Figure 3. Source code for the image brightness adjustment benchmark

The conversion to float for multiplication, followed by clamping, demonstrates typical vector usage for image tasks, and it is assumed that each pixel is an unsigned byte. This approach is simpler than rewriting large loops in native code, yet it significantly boosts performance on SSE/AVX-capable CPUs.

IV. PERFORMANCE ANALYSIS

In this section, we present our results of each benchmark using the Java Vector API, comparing the performance of vectorized approaches against both conventional scalar loops and, where applicable, JVM auto-vectorization. All experiments were conducted on an AMD Ryzen 9 7940HS system running Arch Linux with JDK 21. Each test was executed ten times, with warm-up runs discarded and outliers removed before averaging the results. To assess the impact of hardware limitations, we also simulated an SSE4.2 fallback environment to evaluate performance on processors with older SIMD instruction sets.

A. *Execution Times Analysis*

Table 1 provides a summary of execution times (in milliseconds) for three computational workloads: matrix multiplication (1024×1024), Fast Fourier Transform (FFT) with one million samples, and grayscale image brightness adjustment for a 4K-resolution image. The table presents the average execution times. Additionally, performance under the SSE4.2 fallback scenario is reported to illustrate the impact of limited SIMD instruction support.

TABLE I. AVERAGE RESULTS FOR MATRIX MULTIPLICATION, FFT, AND GRAYSCALE IMAGE BRIGHTNESS ADJUSTMENT

Benchmark	**Average Benchmark Results** (in ms)		
	Scalar (AVX2)	***Vector API (AVX2)***	***Vector API (SSE4.2)***
Matrix Multiplication (1024x1024)	14488	8292	11640
Fast Fourier Transform (1M Samples)	55.2	28.0	40.0
Image Processing (4K grayscale)	120.1	67.2	95.3

1) *Matrix Multiplication (1024×1024)*

In the matrix manipulation benchmark, the Java Vector API achieves an average execution time of 8292 milliseconds, compared to 14488 milliseconds for a naive scalar loop, resulting in approximately 1.7× performance improvement on a system equipped with AVX2 support. Under the SSE4.2 fallback configuration, the average execution time increases to 11640 milliseconds, reducing the performance gain to approximately 1.1×.

2) *FFT Computation (1M samples)*

The vectorized benchmark implementation of the FFT "butterfly" pass completes on average in 28.0 milliseconds on an AVX2-capable processor, effectively halving the average execution time relative to the 55.2 milliseconds required by the scalar implementation. When executing under the SSE4.2 fallback environment, performance degrades to 40.0 milliseconds on average, yielding a more modest gain of approximately 1.4× over the scalar baseline.

3) *Image Processing (4K grayscale)*

The brightness adjustment benchmark completes in 67 milliseconds using AVX2 on average, compared to 120 milliseconds with a scalar approach, representing approx. 1.8× improvement. Under the SSE4.2 fallback mode, the execution time increases to 95.3 milliseconds on average, providing a more limited yet measurable performance increase of approximately 1.3×.

In addition to the average results in Table 1, results could present execution times across runs for each Matrix Multiply scenario (Scalar, Vector API on AVX2, and Vector API fallback on SSE4.2). Fig. 4 presents individual benchmark execution times (representing runs #1 through #10) and the corresponding execution time (in milliseconds), illustrating per-run variability and consistency, and a clear trend emerges regarding the stability and efficiency of different computational approaches.

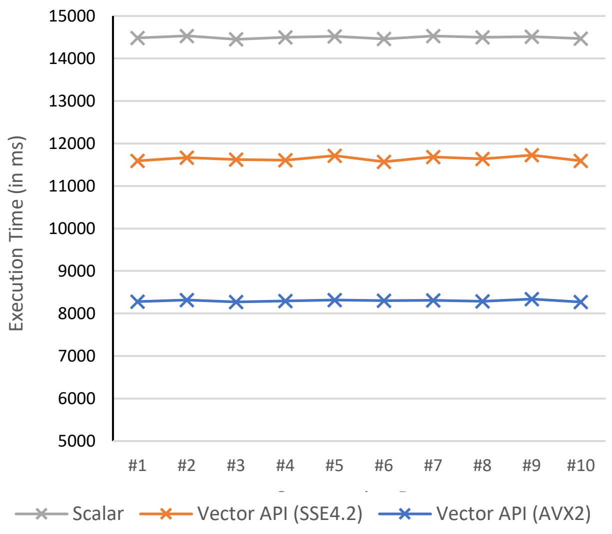

Figure 4. Individual execution times for the matrix multiplication runs, comparing Scalar, Vector API (AVX2), and Vector API (SSE4.2)

The results also demonstrate that, despite being faster than purely scalar execution, the SSE4.2 fallback mode is unable to match the performance of fully vectorized execution. With respect to execution variability, standard deviations remain relatively low at 2–3% for AVX2, whereas SSE4.2 exhibits some minor fluctuations, with deviations rising to approximately 5–8%. This increased variability in the fallback environment is likely attributed to the reliance on partial vector instructions and masked operations, which introduce additional processing overhead.

4) Summary of Results – Performance Gains

Our findings indicate that the Java Vector API delivers substantial performance improvements over naive scalar loops. The presented data further supports the following observations:

- Scalar loops exhibit the slowest performance, averaging approximately 14.5 seconds.
- The Vector API with AVX2 support maintains a clustered execution time of around 8.3 seconds, reflecting a significant improvement.
- The Vector API operating under SSE4.2 falls between the two, averaging approximately 11.6 seconds.

The benchmarks on modern hardware with AVX2 support exhibit performance gains of approximately 1.7× for matrix multiplication, approximately 2× for FFT computation, and approximately 1.8× for image processing. In contrast, execution under older SIMD instruction sets (SSE4.2) results in more modest yet notable performance gains. These findings highlight the importance of advanced CPU vectorization capabilities in achieving optimal execution speeds while also demonstrating the Java Vector API's ability to maintain functionality and provide measurable improvements even in constrained hardware environments.

V. Discussion

In this section, we analyze our results and findings, explore limitations and challenges, and discuss the broader implications of fallback mechanisms across different hardware architectures.

A. Limitations and Challenges

While the Java Vector API offers substantial advantages—particularly when deployed on modern hardware with advanced SIMD instruction sets—several constraints may affect its widespread adoption in practical applications. This section outlines the most significant challenges, including hardware dependencies, development complexity, memory considerations, and the impact of fallback mechanisms.

1) Hardware Dependencies

The performance benefits of the Java Vector API are inherently tied to the SIMD capabilities of the underlying processor (e.g., AVX2, AVX-512, NEON). In environments where SIMD support is limited or absent, performance gains are significantly reduced as the JVM reverts to scalar execution to maintain functional correctness [7]. Although this fallback mechanism ensures cross-platform compatibility, it can lead to performance degradation. For example, our experimental results indicate that matrix multiplication achieves an improvement of approximately 1.7× on AVX2-enabled hardware, but this advantage drops to approximately 1.1× when executed under the SSE4.2 fallback scenario.

A clear distinction emerges between different computing environments. In high-performance systems such as enterprise servers or gaming PCs, where modern SIMD instruction sets are widely available, the Vector API can be effectively leveraged to maximize computational efficiency. Conversely, in embedded systems or legacy desktop processors with limited vectorization support, the fallback mechanism often provides only marginal improvements over traditional scalar execution. Consequently, developers must carefully navigate the trade-off between ensuring broader portability and achieving optimal performance across diverse hardware platforms.

2) Development Complexity

Although the Java Vector API offers a more accessible alternative to low-level intrinsics [8], implementing efficient vectorized code remains a relatively complex task. Developers must carefully manage several factors to utilize SIMD capabilities, including:

- Memory Alignment: Ensuring data is correctly aligned with vector boundaries is crucial to prevent inefficient partial loads and stores, which can negatively affect performance.
- Handling Remainder Loops: When array sizes are not exact multiples of the vector length, additional

logic is required to process the remaining elements, often necessitating specialized remainder loops.

- Data Structure Design: Arrays and data layouts must be optimized to maintain predictable memory access patterns, minimizing cache inefficiencies and maximizing SIMD throughput.

Addressing these challenges can increase code complexity [9]. In relatively straightforward high-performance computing (HPC) scenarios, manually rewriting computational loops to leverage vectorization is often feasible. However, in large-scale applications, systematically identifying performance-critical code segments and converting them into efficient vectorized implementations represents a substantial effort. Despite these challenges, the complexity associated with using the Vector API remains considerably lower than managing JNI-based SIMD intrinsics in languages such as C or C++, making it a more practical option for Java-based high-performance applications.

3) Memory Access Patterns

Data should be stored in a contiguous and well-aligned manner to maximize performance, as irregular memory access patterns can negatively affect performance gains or even negate them entirely. Vectorized loads and stores are most efficient when sequential data aligns neatly with cache lines, reducing the need for costly memory-fetch operations [10].

In our matrix multiplication benchmark, elements from matrix B are accessed column-wise, requiring gathering operations that introduce additional overhead. A potential optimization would involve transposing matrix B beforehand, allowing for more efficient contiguous access within the main computational loop. In contrast, many image-processing workloads already utilize a row-major format, which aligns well with vector boundaries and simplifies memory access patterns. Ensuring that data structures are designed with these considerations is crucial for achieving optimal vectorization efficiency.

4) Fallback Mechanisms

In cases where the underlying hardware lacks support for specific SIMD features—such as when AVX instructions are unavailable, and only SSE2 is supported—the Java Vector API reverts to either partial vectorization or scalar execution. While this fallback mechanism guarantees functional correctness, it can introduce significant performance overhead, particularly due to the use of masked operations or the need to work with narrower registers in loop iterations. Our experimental results indicate that while SSE4.2-based execution yields moderate performance gains of approximately 1.1× to 1.4× over scalar code, these are notably lower than the 1.7× to 2× improvements achieved with AVX2.

Although the Vector API is designed to automatically select the most appropriate vector species based on available hardware features, additional manual tuning may be necessary for HPC applications. In large-scale systems, fallback-related overhead can become substantial enough to impact overall throughput negatively. Developers working on performance-critical applications may benefit from explicitly detecting CPU capabilities and implementing alternative execution paths to optimize performance across different hardware configurations.

5) Potential Performance Gains with Concurrency

The results based on our benchmarks focus on the single-threaded execution of the Java Vector API. Further performance improvements on multi-core systems could potentially be achieved by introducing parallelism, for example, by utilizing Java parallel streams or newly added virtual threads to distribute vectorized computations across multiple CPU cores. In theory, such an approach could significantly enhance throughput on multi-core architectures.

The effectiveness of combining vectorization with concurrency depends on careful workload distribution. Concurrency introduces additional overhead, including thread scheduling and synchronization, which may offset the benefits of parallel execution. Furthermore, suboptimal data partitioning or cache contention between threads can reduce expected improvements, limiting overall efficiency.

For HPC applications, an optimal approach may involve integrating coarse-grained parallelism (with multiple threads) with fine-grained SIMD vectorization within each thread. While this hybrid strategy has the potential to maximize computational throughput, it also introduces additional complexity in implementation and tuning. Achieving the best balance between parallel execution and vectorization efficiency requires careful profiling and optimization based on the characteristics of the specific workload and hardware architecture.

VI. Conclusion

The Java Vector API presents a significant advancement in facilitating high-performance, data-parallel computations within the Java ecosystem. Our analysis and comparison, encompassing matrix multiplication, Fast Fourier Transform (FFT), and image processing workloads, demonstrates that explicit SIMD utilization can significantly reduce execution times. By integrating hardware-accelerated vectorization while maintaining Java's inherent memory safety and portability, the Vector API effectively narrows the performance gap that has historically led developers to rely on native SIMD intrinsics in lower-level languages.

However, the practical adoption of the Vector API is not without challenges. Developers must carefully consider hardware dependencies, as performance gains are highly contingent on the availability of SIMD instruction sets such as AVX2 or AVX-512. In cases where the underlying architecture lacks sufficient vectorization support, the fallback mechanisms ensure functional correctness but at the cost of significantly diminished performance. Additionally, the complexity of writing and optimizing vectorized code requires careful attention to memory alignment, loop structures, and efficient data access patterns.

Future research directions should focus on automating the identification of code regions suitable for vectorization, refining fallback mechanisms to mitigate performance degradation, and expanding the API's capabilities to support a broader range of data types and computational patterns.

Furthermore, exploring the integration of the Vector API with parallelization frameworks and GPU offloading strategies could unlock additional performance improvements for compute-intensive Java applications.

Our results confirm that the Java Vector API is a robust and viable solution for accelerating computationally demanding workloads in pure Java. While it may not yet match the flexibility of native SIMD implementations, it offers a compelling balance between performance and portability. As the OpenJDK Vector API project continues to evolve, anticipated enhancements—including additional vector species, improved memory alignment heuristics, and advanced fallback detection—are expected to further strengthen the Vector API as a key component in optimizing data-parallel computations within the Java ecosystem.

REFERENCES

[1] Oracle Corporation, "JEP 338: Vector API (Incubator)," [Online]. Available: https://openjdk.java.net/jeps/338

[2] C. Marneni, "Distributed Computing and JAVA VectorAPI for performance optimization for ScalaTion Framework," 2009. [Online]. Available: http://cobweb.cs.uga.edu/~jam/home/theses/chandana_project/Final_Report.pdf

[3] J. Nie, B. Cheng, S., Li, L., Wang, and X-F. Li, ,"Vectorization for Java,", IFIP International Conference on Network and Parallel Computing (NPC 2010), LNCS 6289, pp. 3-17, 2010.

[4] M. Basso, A. Rosa, L. Omini and W. Binder, " Java Vector API: Benchmarking and Performance Analysis," Proc. of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC 2023), ACM NY, pp. 1-12, 2023.

[5] A. Rosa, M. Basso, and W. Binder, "Analyzing the Vectorization Capability of the Java Vector API, " 12th International Symposium on Computing and Networking Workshops (CANDARW), IEEE, pp. 403-405, 2024.

[6] J. Löff, F. Schiavio, A. Rosa, M. Basso, and W. Binder, " Vectorized Intrinsics Can Be Replaced with Pure Java Code without Impairing Steady-State Performance," ICPE '24: Proc. of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE' 24), ACM, pp. 14-24, 2024.

[7] J. Kavanaugh-Brown, "Vector math made easy: John Rose and Paul Sandoz on Java's Vector API," Java Magazine, 2021. [Online]. Available: https://blogs.oracle.com/javamagazine/post/java-vector-api-simd

[8] Intel Corporation, "Intel® Intrinsics Guide," [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[9] P. Sandoz, "The Vector API in JDK 17," presented at the JVM Language Summit (JVMLS), 2021.

[10] D. Delabassee, J. Rose, P. Sandoz, "The Vector API", Inside Java, 2020. [Online]. Available: https://inside.java/2020/11/17/podcast-007/