Computing, Telecommunication and Control, 2024, Vol. 17, No. 4, Pp. 7–15.
Информатика, телекоммуникации и управление. 2024. Том 17, № 4. С. 7–15.

Information Technologies
Информационные технологии

Research article
DOI: https://doi.org/10.18721/JCSTCS.17401
UDC 004.422

EVALUATING THE PERFORMANCE OF JAVA VECTOR API IN VECTOR EMBEDDING OPERATIONS

N.A. Tomilov, V.P. Turov ✉
ITMO University, St. Petersburg, Russian Federation
✉ firemoon@icloud.com

Abstract. Hardware vector instructions are widely used to improve the performance of computations. The Java Vector API introduced in Java 16 allows using them portably on any platform supported by the Java Virtual Machine (JVM). In this paper, we evaluate performance benefits from rewriting typical vector search operations, such as computing distance between two vector embeddings, using the Java Vector API. We compare the performance of these vectorized implementations with semantically equivalent scalar code. Furthermore, we compare the Java Vector API with native C++ implementations, called from Java code via different Java-to-native interfaces, namely Java JNI, Project Panama (Foreign Function and Memory API), and manipulating Java JIT compiler via JVM CI and Nalim library. Benchmarking results suggest that in certain situations using Vector API can produce a measurable increase in performance of low-level operations, which can be translated into speedup of high-level algorithms such as Product Quantization. However, under certain scenarios, using Vector API is slower than relying on automatic vectorization provided by JVM, and most benchmarks suggest that invoking calculations implemented in C++ is faster even with all performance penalties incurred by native code invocations. Using techniques to lower these penalties, for example, by avoiding memory copy operations, can decrease the execution time by five times compared to Vector API and by ten times compared to plain Java code. However, in cases where using native code is prohibited, Vector API can still demonstrate a noticeable performance uplift, which can be beneficial for vector-related calculations in Java applications.

Keywords: vector embeddings, Java JVM, Java JNI, Java JVM CI, Project Panama, native code invocation

Citation: Tomilov N.A., Turov V.P. Evaluating the performance of Java Vector API in vector embedding operations. Computing, Telecommunications and Control, 2024, Vol. 17, No. 4, Pp. 7–15. DOI: 10.18721/JCSTCS.17401

Научная статья
DOI: https://doi.org/10.18721/JCSTCS.17401
УДК 004.422

ИЗМЕРЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ JAVA VECTOR API ПРИ ЕГО ИСПОЛЬЗОВАНИИ В ОПЕРАЦИЯХ НАД ВЕКТОРНЫМИ ПРЕДСТАВЛЕНИЯМИ

Н.А. Томилов, В.П. Туров ✉

Национальный исследовательский университет ИТМО,
Санкт-Петербург, Российская Федерация

✉ firemoon@icloud.com

Аннотация. Аппаратные векторные инструкции широко используются для повышения производительности вычислений. Java Vector API, представленный в Java 16, позволяет переносимо использовать их на любой платформе, поддерживаемой виртуальной машиной Java. В данной работе выполняется оценка производительности при реализации типичных операций поиска по векторным представлениям, таких как вычисление расстояния между двумя векторными представлениями, с использованием Java Vector API. Производительность векторизованных реализаций этих операций сравнивается с семантически эквивалентным скалярным кодом. Кроме того, производится сравнение Java Vector API с нативными реализациями на C++, вызываемыми из Java-кода через различные интерфейсы взаимодействия Java с нативным кодом, а именно Java JNI, Project Panama (Foreign Function and Memory API) и управление JIT-компилятором через JVM CI и библиотеку Nalim. Результаты тестирования показывают, что в определенных ситуациях использование Vector API может привести к заметному увеличению производительности низкоуровневых операций, что может выражаться в ускорении высокоуровневых алгоритмов, таких как Product Quantization. Однако в некоторых сценариях использование Vector API оказывается медленнее по сравнению с автоматической векторизацией, предоставляемой JVM, и большинство тестов показывают, что вызов вычислений, реализованных на C++, занимает меньше времени по сравнению с Vector API, даже с учетом всех накладных расходов, возникающих при вызовах нативного кода. Используя методы для снижения этих накладных расходов, например, избегая операций копирования памяти, можно уменьшить время выполнения в пять раз по сравнению с Vector API и в десять раз по сравнению с обычным Java-кодом. Тем не менее, в случаях, когда использование нативного кода запрещено, Vector API все еще может демонстрировать заметное повышение производительности, что может быть полезно для вычислений, связанных с векторными представлениями, в Java-приложениях.

Ключевые слова: векторные представления, Java JVM, Java JNI, Java JVM CI, Project Panama, вызов нативного кода

Для цитирования: Tomilov N.A., Turov V.P. Evaluating the performance of Java Vector API in vector embedding operations // Computing, Telecommunications and Control. 2024. Т. 17, № 4. С. 7–15. DOI: 10.18721/JCSTCS.17401

Introduction

Modern processors have instruction sets that enable the simultaneous execution of certain operations on arrays of data. These instructions are referred to as SIMD instructions, where SIMD stands for Single Instruction Multiple Data. The use of such instructions significantly accelerates computations over numerical arrays and matrices [1, 2]. The process of transforming a loop that performs calculations on single data elements into a loop that operates on data blocks using SIMD instructions is called vectorization.

Vectorization can significantly accelerate any data array processing. One example of such data processing is the organization of search operations in large document databases, where machine-learning

techniques are used to transform the data into representations that reflect the semantic structure of textual [3] and multimodal documents in the form of vector embeddings (representations), which are arrays of floating-point numbers. In this approach, an efficient search can be organized by constructing an index of these embeddings, converting the search query into its own embedding, and then performing the search for the nearest vector embeddings using the constructed index [4]. Vectorizing the computation of the distance between two vectors significantly reduces the time required for vector search [5]. In general, vector embeddings can have any dimensionality, but in vector search operations the dimensionality of each vector is typically small and each vector is often represented as an array of floating-point numbers with up to 1000 elements. Another key point of vector search operations is having a large number of relatively small operations (e.g., distance calculation or computing an average vector) [6].

High-level programming languages often have their own mechanisms for automatic vectorization, either during runtime, as in Java JVM, or at the compilation stage, as in C++. In case of Java, the JVM can automatically vectorize only a small set of operations[1]; in other cases, vectorization must be implemented manually, either through certain intrinsic functions of the language, such as FMA (Fused-Multiply-Add), or by implementing computational operations in a low-level language with access to assembly-level SIMD instructions, and then invoking the implemented functions from Java through various mechanisms for executing platform-dependent code. Java 16 introduced an additional mechanism for working with vector instructions – Java Vector API – which enables convenient use of vector instructions without the need for platform-dependent code. This mechanism offers a significant performance boost compared to Java's automatic vectorization [7], and despite its experimental status, it is already being adopted in some software products, such as Apache Lucene[2].

In this paper, we test the performance of the Java Vector API in the scope of operations on vector embeddings that are used in vector search. We compare the usage of this API to the implementation of the same operations in Java without vector instructions, as well as to the implementations in C++, where we call the corresponding functions from Java using mechanisms, such as Java JNI, Java JVM CI, and the Project Panama (Foreign Function and Memory API).

Methods of using vector instructions in Java

As mentioned above, there are three methods of working with vector instructions in Java: using intrinsics, invoking a shared library that was built in some other language with support for vector instructions, and using Vector API.

The first method involves using intrinsics, such as the FMA operation, which are optimized functions provided directly by the JDK developers. Unfortunately, the existing set of intrinsics is mostly limited to cryptographic operations, and it is not practical to use them when implementing operations applied in vector search.

The second method involves implementing vector computational operations in a low-level language, such as C or C++, and subsequently invoking these operations from Java JVM [8]. This approach allows us to optimize computational functions more precisely and take full advantage of any vector or other assembly instructions, automatic code vectorization provided by the low-level compiler [9], or even already implemented and highly optimized shared libraries, resulting in higher performance compared to plain Java [10]. However, this approach comes with several significant drawbacks. The most obvious are the need to write platform-dependent code and the need to modify it for each new architecture or even processor generation to achieve maximum performance. Additionally, this method comes with the complexity of working directly with the required vector and assembly instructions, direct memory handling, and potential security risks [11].

[1] Vladimir Ivanov. Vectorization in HotSpot JVM. Available: https://cr.openjdk.java.net/~vlivanov/talks/2017_Vectorization_in_HotSpot_JVM.pdf (Accessed: 29.09.2024)

[2] Available: https://github.com/apache/lucene/pull/12311 (Accessed: 06.12.2024)

A less obvious drawback of this approach is the overhead associated with data copying. When transferring control to platform-dependent code, the JVM typically does not allow direct access to memory managed by the JVM. If the data for computations resides in JVM memory, we must copy it to memory outside the JVM, and then we need to free that memory ourselves. This copying is necessary for each data array and, in addition to the time spent on the copying itself, incurs context switching from managed code to the JVM and other overheads [12]. When we need to pass matrices, being arrays of arrays, to managed code, the context-switching overhead can become so significant that it is more efficient to copy the data into a single large-dimensional array and transfer it, rather than copying each row of the matrix as a separate array. Alternatively, we can store all data in unmanaged memory beforehand, which complicates access to that data from the JVM.

Although the problems associated with executing unmanaged code in a managed environment are generally insurmountable, modern Java provides several mechanisms to reduce overhead from memory copying and context switching between managed and unmanaged code. One of the most effective methods for reducing overhead is through altering the just-in-time (JIT) compilation within the JVM. This allows us to completely replace the function body generated by the JIT compiler, specifically substituting the proper call to platform-dependent code, which involves memory copying and context switching, with a direct call to the platform-dependent function [13, 14]. However, this method has significant drawbacks, including the lack of exception handling, complete blocking of the thread invoking unmanaged code, and having access to the JVM from unmanaged code. Consequently, we cannot copy objects or complex data structures, such as arrays of arrays, into unmanaged code; we can only work with primitive types or one-dimensional arrays. When we need to pass two-dimensional arrays, such as arrays of vectors, we must manually copy the data into a large one-dimensional array. However, despite these limitations, this method remains popular and is used in frameworks for heterogeneous computing, such as Tornado VM [15], and in the Nalim library[3], which utilizes Java JVM CI capabilities. Another method for reducing overhead, aside from managing the JIT compiler, is the Foreign Function and Memory API, or Project Panama. This API, which recently achieved Released status in Java 22, offers a modern alternative to JNI and allows, under certain conditions, direct access to JVM memory from unmanaged code, including complex data structures like objects or arrays of arrays. This means that while the overhead from switching thread states between managed and unmanaged code still exists, we can entirely eliminate memory copying for complex data structures, potentially accelerating computations for arrays of vectors even more than through JIT compilation management.

The third method involves using the recently introduced Java Vector API. This framework provides a high-level abstraction over various vector instructions available on different architectures supported by the Java JVM. When using this framework, developers do not need to consider the target architecture or which specific vector instructions to utilize; they only need to prepare the data correctly and invoke a certain function, for example, addition. This function will be translated into the appropriate vector instruction, often the most suitable one, depending on the processor architecture and its supported instruction set. If the processor lacks support for vector instructions, the framework transparently falls back to using standard scalar instructions. Apart from ease of use, the Java Vector API offers several advantages, including the absence of manual memory copying to and from unmanaged code, and full support from the JVM, including proper exception handling and effective organization of computations in a multithreaded environment. Moreover, not having to deal with unmanaged code allows developers to use vector instructions for speeding up computations in contexts where using unmanaged code is prohibited or requires extensive precautions, such as in the financial sector.

However, a significant drawback is the need for manual organization of computations using vector instructions, which, albeit not directly related to assembly instructions, still closely resembles the use of vector instructions on the processor. Furthermore, there are no optimizations available if the instructions are

[3] Available: https://github.com/apangin/nalim (Accessed: 06.12.2024)

used incorrectly or inefficiently. For instance, when computing the sum of two vectors with a dimension of 390, the developer must perform a summation of 48 parts of vectors of length 8 in a loop, and then separately sum the remaining 6 elements manually, assuming the processor supports a maximum vector width of 256, meaning it can handle operations on up to 8 32-bit floating-point numbers at once. Any mishandling of this vector alignment will cause performance degradation. In addition, it is important to note that as of the time of writing, the Java Vector API remains in Incubator Preview status, and there is no information on when it will become fully supported and available for use in production environments.

Experiment setup

To test the performance of the Vector API, we created an interface that describes various operations for vector embeddings. We then implemented the following versions of this interface:

1. Standard Java;
2. Java with Vector API;
3. Java with Calls to Platform-Dependent Code Implemented in C++:
    - Using Java JNI, which includes all potential overheads;
    - Using the Nalim library, which leverages Java JVM CI to manage JIT compilation, eliminating context-switching overhead, but requiring memory copying for two-dimensional arrays;
    - Using the Project Panama (Foreign Function and Memory API), which incurs overhead from switching to native code, but does not require memory copying for one-dimensional and two-dimensional arrays.

The implemented operations can be divided into several sets.

The first set includes operations on a single vector or a pair of vectors of dimensionality D and consists of the following operations:

- Calculation of the Average value of the vector elements;
- Calculation of the Variance, also known as Dispersion, of the vector element values;
- Calculation of the Angular, also known as Cosine, and Euclidean Distances between two vectors [16].

The second set includes operations performed on multiple vectors simultaneously and uses an array of vectors (a two-dimensional array) as one of the input parameters:

- Calculation of the Average Vector from N source vectors;
- Calculation of N Angular and Euclidean Distances between one vector and N other vectors.

The third set includes complex computations that utilize operations from the first two sets:

- Clustering using the k-Nearest Neighbors (kNN) method from N source vectors into K clusters, which involves calculating N distances and calculation of the Average Vector [17];
- Product Quantization of N source vectors [18] to M subvectors with a quantization depth of nBits, which uses kNN-clustering.

The purpose of this set is to measure the extent of the overhead incurred due to the need to convert sets of vectors, which are arrays of arrays, into a one-dimensional array for passing to unmanaged code.

The sets of vectors were randomly generated and consisted of N arrays of 32-bit floating-point numbers (FP32) of size D. The following parameters were used:

- For the first two sets of operations:
    - D = 256, 384, 768, N = 100, 250, 500;
- For kNN-clustering:
    - D = 256, N = 5000, 10000;
    - K = 20, 50;
- For PQ:
    - D = 256, N = 10000;
    - M = 4, 8;
    - nBits = 8, 12, 16.

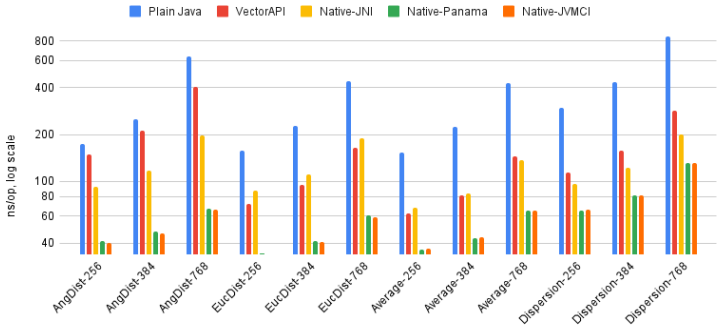

Fig. 1. Results for Operations on a single vector set

The test setup has the following specifications: AMD Ryzen 7 7700X (8C16T); 32GB RAM; Operating System: Ubuntu 22.04; a framework of comparing vector search algorithms, implemented in previous research [19], that uses the Java Microbenchmark Harness (JMH) [20]. The JMH was configured with a warm-up mode to allow the JIT compiler to perform necessary optimizations, including automatic vectorization. The C++ code was compiled using GCC 11.4.0. For all computations except JVMCI, JDK 22 version 22.0.2+9 was used. For JVMCI, JDK 17 version 17.0.10+7 was selected, because it was the most recent long-term support version of the JVM that the Nalim library works correctly with. In newer JVM versions, JVM CI was modified in a way that caused the library to malfunction, and modifying it is beyond the scope of this article. The source code for the benchmark and all implemented vector operations is available on GitHub[4].

Experiment results

The results for the first set of operations are presented in Fig. 1. The measured execution time for a single operation is in nanoseconds, where lower values indicate better performance.

It can be observed that using the Vector API significantly reduces the execution time for a single operation compared to standard scalar Java. However, the execution time for calculations using the Vector API is nearly indistinguishable (sometimes longer, sometimes shorter) from that of calculations implemented in C++ with an optimizing compiler and function calls via JNI. The execution time for calculations implemented in C++ and invoked through JVM CI or Project Panama is substantially lower than that using JNI, and is two to four times less than the execution time for calculations using the Vector API. This indicates significant overhead from both using JNI and the Vector API. Nevertheless, the speedup from the Vector API compared to regular Java computations is notable and directly depends on the number of vector operations: the more operations are necessary to perform the computation, the less pronounced the speedup becomes, as seen when comparing computation time of angular distance (which needs three operations in total) and Euclidean distance (which needs only one operation) for the same number of dimensions. Such difference depending on the number of operations also demonstrates the overhead involved when invoking the Vector API.

The results for the second set of operations are presented in Fig. 2. Fig. 2a shows the computation time for the average of N vectors of dimensionality D, Fig. 2b displays the computation time for N Euclidean distances between a given vector and N specified vectors, and Fig. 2c illustrates the similar time for angular distances.

When using the Vector API, the computation time for the average vector is approximately equal to that of standard Java and significantly less than the time required for calling C++ code via JNI or JVM CI. The performance degradation of the C++ code is attributed to the overhead of converting N arrays

[4] Available: https://github.com/nikita-tomilov/vector-vapi-jni-jvmci-panama (Accessed: 06.12.2024)

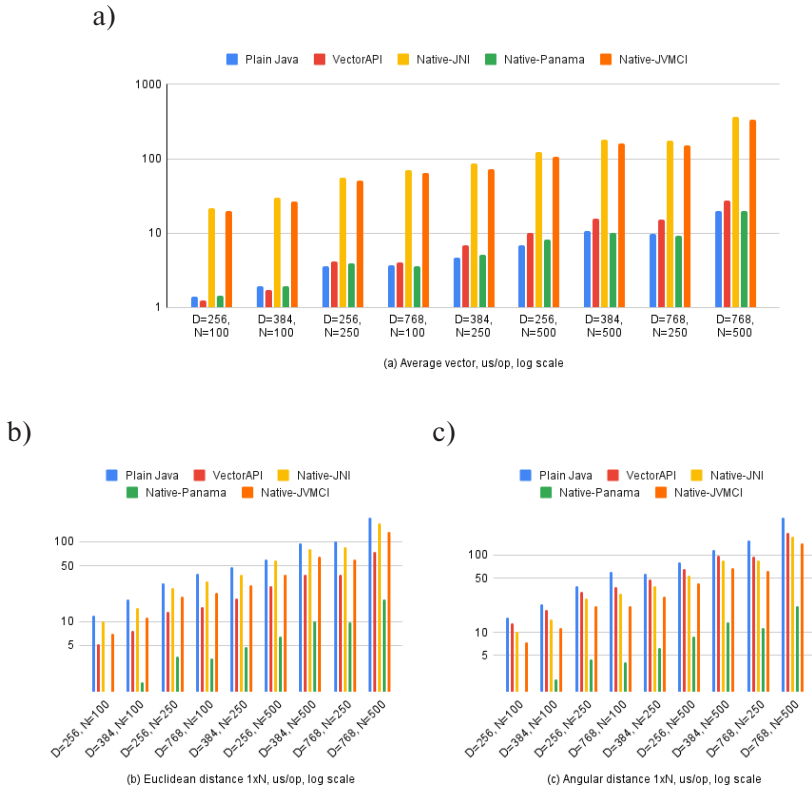

Fig. 2. Results for the multi vector operations

of dimensionality D into a one-dimensional array of size N×D for passing to unmanaged code. A similar conversion is necessary for calculating N distances; however, in this case, the slow data transfer to unmanaged code is offset by the more optimized code generated by the optimizing compiler, resulting in the use of C++ code in conjunction with JVM CI yielding the greatest speedup for distance calculations. Nevertheless, although in both cases the computation of N distances using the Vector API is significantly faster than using standard Java, and in the case of Euclidean distance, it is somewhat faster than using C++ and JNI, the combination of C++ and Project Panama remains an order of magnitude faster, thanks to the advantages provided by the optimizing compiler and the absence of memory copying overhead.

The results for the third set of operations are presented in Fig. 3. Fig. 3a shows the time for clustering using the kNN method, while Fig. 3b demonstrates the execution time for Product Quantization.

The results for the third set of operations partially replicate those of the second set: the parity between scalar Java and the Vector API remains for the measurement of Product Quantization time, with both benchmarks showing significant time overhead for JNI and JVM CI due to memory copying. In both benchmarks, Project Panama is the fastest option, benefiting from the absence of memory copying overhead. However, for kNNs, the Vector API is faster than scalar Java in all cases, achieving up to a twofold increase in performance at best.

Conclusion

The use of the Java Vector API significantly accelerates vector operations in applications written in Java or running in the JVM. However, computations implemented using the Vector API and manual vectorization generally lag behind calls to functions implemented in C++ that utilize automatic vectorization and mechanisms to reduce overhead associated with calling unmanaged code, such as the Project Panama

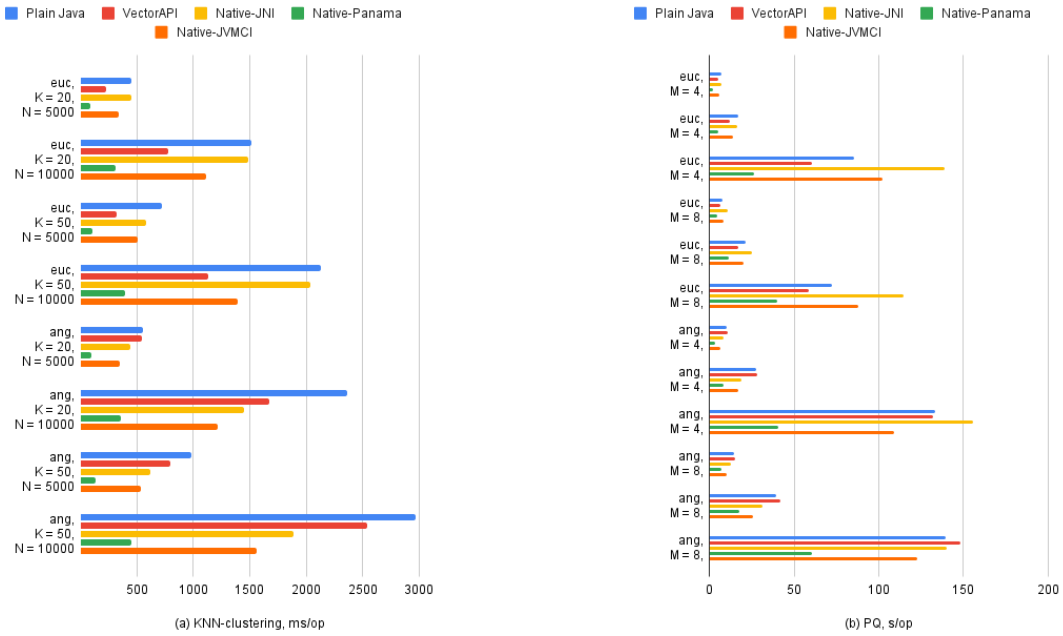

Fig. 3. Results for the complex vector operations

(Foreign Function and Memory API). Nevertheless, in scenarios where the use of unmanaged code is not feasible, as well as in cases where the conversion and transfer of data to managed code incur significant overhead despite the use of optimized methods for calling unmanaged code, the Vector API serves as a good alternative to traditional Java computations.

REFERENCES

1. **Mula W., Kurz N., Lemire D.** Faster population counts using AVX2 instructions. The Computer Journal, 2017, Vol. 61, No. 1, Pp. 111–120. DOI: 10.1093/comjnl/bxx046

2. **Cree M.J.** An exploration of using the Intel AVX2 gather load instructions for vectorised image processing. 2018 International Conference on Image and Vision Computing New Zealand (IVCNZ), 2018, Pp. 1–9. DOI: 10.1109/IVCNZ.2018.8634707

3. **Grbovic M., Cheng H.** Real-time personalization using embeddings for search ranking at Airbnb. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2018, Pp. 311–320. DOI: 10.1145/3219819.3219885

4. **Berry M.W., Drmac Z., Jessup E.R.** Matrices, vector spaces, and information retrieval. SIAM Review, 1999, Vol. 41, No. 2, Pp. 335–362. DOI: 10.1137/S0036144598347035

5. **Douze M., Guzhva A., Deng C., Johnson J., Szilvasy G., Mazaré P., Lomeli M., Hosseini L., Jégou H.** The Faiss library, 2024. DOI: 10.48550/arXiv.2401.08281

6. **Sanca V.** Ailamaki A. Efficient data access paths for mixed vector-relational search. Proceedings of the 20th International Workshop on Data Management on New Hardware, 2024, Pp. 1–9. DOI: 10.1145/36620-10.3663448

7. **Basso M., Rosà A., Omini L., Binder W.** Java Vector API: Benchmarking and performance analysis. Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, 2023, Pp. 1–12. DOI: 10.1145/3578360.3580265

8. **Stojanov A., Toskov I., Rompf T., Püschel M.** SIMD intrinsics on managed language runtimes. Proceedings of the 2018 International Symposium on Code Generation and Optimization, 2018, Pp. 2–15. DOI: 10.1145/3168810

9. **Naishlos D.** Autovectorization in GCC. GCC Developers' Summit, 2004, Pp. 105–118.

10. **Vivanco R., Pizzi N.** Computational performance of Java and C++ in processing large biomedical datasets. IEEE CCECE2002. Canadian Conference on Electrical and Computer Engineering. Conference Proceedings (Cat. No. 02CH37373), 2002, Vol. 2, Pp. 691–696. DOI: 10.1109/CCECE.2002.1013025

11. **Grichi M., Abidi M., Jaafar F., Eghan E.E., Adams B.** On the impact of interlanguage dependencies in multilanguage systems empirical case study on Java Native Interface applications (JNI). IEEE Transactions on Reliability, 2021, Vol. 70, No. 1, Pp. 428–440. DOI: 10.1109/TR.2020.3024873

12. **Halli N.A., Charles H.-P., Mehaut J.-F.** Performance comparison between Java and JNI for optimal implementation of computational micro-kernels, 2015. DOI: 10.48550/arXiv.1412.6765

13. **Stepanian L., Demke Brown A., Kielstra A., Koblents G., Stoodley K.** Inlining Java native calls at runtime. Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, 2005, Pp. 121–131. DOI: 10.1145/1064979.1064997

14. **Grimmer M., Rigger M., Stadler L., Schatz R., Mössenböck H.** An efficient native function interface for Java. Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, 2013, Pp. 35–44. DOI: 10.1145/2500828.2500832

15. **Fumero J., Papadimitriou M., Zakkak F.S., Xekalaki M., Clarkson J., Kotselidis C.** Dynamic application reconfiguration on heterogeneous hardware. Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2019, Pp. 165–178. DOI: 10.1145/3313808.3313819

16. **Abu Alfeilat H.A., Hassanat A.B.A., Lasassmeh O., Tarawneh A.S., Alhasanat M.B., Eyal Salman H.S., Surya Prasath V.B.** Effects of distance measure choice on K-nearest neighbor classifier performance: A Review. Big Data, 2019, Vol. 7, No. 4, Pp. 221–248. DOI: 10.1089/big.2018.0175

17. **Seidl T., Kriegel H.-P.** Optimal multi-step K-nearest neighbor search. ACM SIGMOD Record, 1998, Vol. 27, No. 2, Pp. 154–165. DOI: 10.1145/276305.276319

18. **Jégou H., Douze M., Schmid C.** Product quantization for nearest neighbor search. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2011, Vol. 33, No. 1, Pp. 117–128. DOI: 10.1109/TPAMI.2010.57

19. **Turov V.P., Tomilov N.A., Babaiants A.A.** Razrabotka instrumenta sravneniia algoritmov vektornogo poiska [Development of a tool for comparing vector search algorithms]. XI Kongress Molodykh Uchenykh [XI Congress of Young Scientists], 2022, Vol. 1, Pp. 446–450.

20. **Laaber C., Leitner P.** An evaluation of open-source software microbenchmark suites for continuous performance assessment. Proceedings of the 15th International Conference on Mining Software Repositories, 2018, Pp. 119–130. DOI: 10.1145/3196398.3196407

INFORMATION ABOUT AUTHORS / СВЕДЕНИЯ ОБ АВТОРАХ

Tomilov Nikita A.
Томилов Никита Андреевич
E-mail: programmer174@icloud.com

Turov Vladimir P.
Туров Владимир Павлович
E-mail: firemoon@icloud.com

Submitted: 30.09.2024; Approved: 09.11.2024; Accepted: 18.11.2024.
Поступила: 30.09.2024; Одобрена: 09.11.2024; Принята: 18.11.2024.