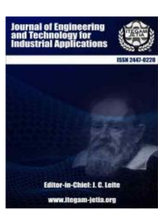
Journal of Engineering and Technology for Industrial Applications

ISSN ONLINE: 2447-0228

ITEGAM-JETIA

Manaus, v.11 n.51, p. 20-27. January/February., 2025.

DOI: https://doi.org/10.5935/jetia.v11i51.1310



RESEARCH ARTICLE OPEN ACCESS

COMPARATIVE EVALUATION BETWEEN JAVA APPLICATION USING JNI AND NATIVE C/C++ APPLICATION RUNNING ON AN ANDROID PLATFORM

Álison de Oliveria Venâncio[1], Thales Ruano Barros de Souza[2] and Bruno Raphael Cardoso Dias[3]

[1 2 3] Instituto de Pesquisas Eldorado. Manaus-Amazonas, Brazil.

[1]http://orcid.org/0009-0000-2850-185X, [2]https://orcid.org/0000-0001-6333-8840, [3]http://orcid.org/0000-0003-0517-7895

Email: alison.venancio@gmail.com, thalesrmb@gmail.com, brunodias89@gmail.com

ARTICLE INFO

Article History
Received: September 27, 2024
Revised: October 20, 2024
Accepted: November 01, 2024
Published: January 30, 2025

Keywords:
Android,
Embedded,
Linux,
Java,
JNI.

ABSTRACT

Android is a popular operating system based on the Linux kernel and has a Java-based framework. As it is built on Linux, it supports the development of applications written in C/C++, known as native applications. The Native Development Kit (NDK), along with the Java Native Interface (JNI), provides a solution for communication between Java applications and native C/C++ applications, resulting in a significant performance boost. This article evaluated the performance difference between Java applications using JNI with the NDK and native C/C++ applications, focusing on algorithms widely used in various areas such as automation, networking, telecom, cybersecurity, etc. We conducted sequence of executions initiated either through a graphical interface or via the Android Debug Bridge (ADB) command line, with timing performed by external hardware with its own firmware for this evaluation. Based on the results, we observed that in all test cases, the native application performs faster, except when there are variations related to process scheduling, which may rarely lead to a reversal of this pattern.

Copyright ©2025 by authors and Galileo Institute of Technology and Education of the Amazon (ITEGAM). This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).

I. INTRODUCTION

Android [1] is an operating system initially developed with a focus on mobile devices, but nowadays it is widely used in various applications such as cars, televisions, refrigerators, POS systems, and more. It is currently the most popular operating system [2] and is officially developed by the Open Handset Alliance. The source code is released as open-source software [3], and its structure is based on the Linux kernel, with a Java-based framework in its user space.As Android is a system based on Linux, it also supports the development of applications written in C/C++. These are known as native applications because they use libraries compiled specifically for the target system. Some examples of native libraries include libC, OpenGL, WebKit, etc. Since this type of application is closer to the kernel layer, its execution time is usually shorter, but there is a higher complexity in understanding and mastering the syntax of the language.Java is a programming language that utilizes a virtual machine to interact with the system, which results in higher processing costs due to the additional layers of software and this can reduce performance when accessing hardware devices. For this purpose, there is a tool called the Native Development Kit (NDK) [4] that allows communication between a Java application and a native C/C++ application, bringing a considerable performance gain to them.One of the main tools of the NDK is the Java Native Interface [5] (JNI). Applications that use JNI can incorporate native code written in C/C++ while still gaining the advantages of using a higher-level language. Additionally, JNI enables the utilization of native Linux libraries, in conjunction with the benefits of the Android framework simultaneously.

When developing an Android application, it is essential to consider how the system works, and one of the first aspects to evaluate is the execution time. The performance difference between a Java application and a C/C++ application has been a well-established study, but there are few analyses related to this topic applied on an Android platform using the NDK.

In this paper, we evaluate the performance difference between a Java application using JNI and a native C/C++ application. The focus is on testing consolidated algorithms widely used in various areas such as automation, networks, telecommunications, cybersecurity, etc. The main objective of this

work is to establish a comparison between these two development approaches in Android. This comparison provides valuable insights for developers who need to choose between the two approaches to achieve greater efficiency in their applications, either due to hardware limitations or battery consumption.

The article is organized as follows:

- Section II: Review of the algorithms used for performance comparison.
- Section III: Brief description of related works.
- Section IV: Explanation of the development methodology for the test pipeline, metrics, and analysis.
- Section V: Presentation of the test case results.
- Section VI: Analysis of obtained results.
- Section VII: Conclusion of the work and proposals for future research.

II. ALGORITHM DESCRIPTIONS

This section provides a comprehensive overview of the algorithms selected for our performance comparison. These algorithms are widely used across various domains, including data processing, network routing, signal processing, and cryptography. We have carefully chosen algorithms that exhibit different computational complexities and characteristics to ensure a thorough evaluation of the performance differences between Java and C/C++ implementations on Android.

II.1 QUICKSORT

Quicksort is renowned for its efficiency as a sorting algorithm and utilizes a divide-and-conquer strategy to organize data. The core mechanism involves partitioning an array into smaller subarrays around a pivot element. This partitioning step is recursively applied to the resulting subarrays until the entire array is sorted. The selection of the pivot element significantly affects the performance of Quicksort. In the best-case scenario, where the pivot divides the array into nearly equal halves, Quicksort achieves a time complexity of $O(n \log n)$. However, in the worst case, where the pivot selection results in highly imbalanced partitions, the time complexity can degrade to $O(n^2)$. To address these performance issues, techniques such as introsort, which combines Quicksort with Heapsort, and three-way partitioning are employed. Introsort ensures that the algorithm's performance remains $O(n \log n)$ in the worst case, while three-way partitioning helps improve performance by handling arrays with many duplicate elements more effectively [6].

```
QuickSort(arr, low, high):
   if low < high:
      pivot ← Partition(arr, low, high)
      QuickSort(arr, low, pivot - 1)
      QuickSort(arr, pivot + 1, high)
Partition(arr, low, high):
   pivot ← arr[high]
   i ← low – 1

   for j ← low to high - 1:
      if arr[j] ≤ pivot:
         i ← i + 1
          Swap(arr[i], arr[j])
   Swap(arr[i + 1], arr[high])
   return i + 1
```

Figure 1: Flow of the QuickSort algorithm showing the partitioning process and recursion to sort a list of numbers.
Source: Authors, (2025).

II.2 DIJKSTRA'S ALGORITHM

Dijkstra's algorithm is a fundamental graph search algorithm designed to determine the shortest path between nodes in a weighted graph. The algorithm operates by iteratively exploring neighboring nodes and updating the estimated distance to the destination node. A priority queue, often implemented as a minimum heap, is used to select the node with the smallest known distance for expansion. This approach ensures that the shortest path is identified efficiently.

The time complexity of Dijkstra's algorithm depends on the data structure used for the priority queue. When using a binary heap, the algorithm runs in $O((|V| + |E|) \log V)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. If a Fibonacci heap is used, the complexity can be reduced to $O(|E| + |V| \log |V|)$ [7]. Dijkstra's algorithm is widely applicable, including in network routing protocols, geographic information systems, and robotics for pathfinding.

It is important to note that Dijkstra's algorithm can only be used on graphs that have non-negative edge weights. For graphs containing negative edge weights, the Bellman-Ford algorithm or other techniques may be used [8], [9].

```
Dijkstra(graph, source):
   for each v of V:
      dist[v] ← ∞
   dist[source] ← 0
   priority_queue ← [source]

   while priority_queue is not empty:
      u ← node with smallest dist in priority_queue
      Remove u from priority_queue

      for each neighbor v of u:
         if dist[u] + weight(u, v) < dist[v]:
            dist[v] ← dist[u] + weight(u, v)
            Add v to priority_queue
   return dist
```

Figure 2: Illustration of Dijkstra's algorithm determining the shortest path in a weighted graph, focusing on updating distances and selecting nodes using a priority queue.
Source: Authors, (2025).

II.3 FAST FOURIER TRANSFORM

The Fast Fourier Transform (FFT) is a powerful algorithm for computing the Discrete Fourier Transform (DFT), which decomposes a signal into its constituent frequency components. The FFT is invaluable in various signal processing applications, including filtering, data compression, and spectral analysis. In image processing, FFT is employed for tasks such as convolution, filtering, and edge detection.

The efficiency of FFT arises from its recursive structure, which reduces the computational complexity from $O(n^2)$ for the naive DFT algorithm to $O(n \log n)$. This reduction is achieved by recursively dividing the DFT computation into smaller, more manageable subproblems. The FFT's ability to handle large datasets with reduced computational requirements makes it a crucial tool in both theoretical and applied signal processing [10].

```
FFT(A):
  n ← length(A)
  if n = 1:
    return A

  w_n ← e^(2πi/n)  // nth root of unity
  A_even ← FFT(A[0], A[2], ..., A[n-2])
  A_odd  ← FFT(A[1], A[3], ..., A[n-1])

  for k = 0 to n/2 - 1:
    w ← w_n^k
    A[k] ← A_even[k] + w * A_odd[k]
    A[k + n/2] ← A_even[k] - w * A_odd[k]

  return A
```

Figure 3: Diagram of the FFT decomposition process, showing how the input sequence is divided into even and odd components and processed recursively.
Source: Authors, (2025).

II.4 RIVEST-SHAMIR-ADLEMAN ALGORITHM

The Rivest-Shamir-Adleman (RSA), algorithm is a widely adopted public-key cryptosystem that provides a secure method for encrypting and decrypting information over public channels. The security of RSA is based on the mathematical difficulty of factoring large composite numbers. The algorithm involves generating a pair of keys: a public key used for encryption and a private key used for decryption. Encryption is performed by raising the message to the power of the public key exponent, modulo the product of two large prime numbers.

Decryption, on the other hand, is carried out using the corresponding private key. The computational complexity of RSA encryption and decryption is dominated by the modular exponentiation step, which has a time complexity of $O((\log n)^3)$, where n is the size of the modulus (product of the two primes).

The strength of RSA lies in the size of the keys and the computational challenge associated with factoring the product of large primes. RSA is extensively used in various security protocols, including SSL/TLS for secure web communications and digital signatures for authentication and data integrity [11].

```
RSA Key Generation:
  Choose two large primes p and q
  n ← p * q
  φ(n) ← (p - 1) * (q - 1)

  Choose e such that 1 < e < φ(n) and gcd(e, φ(n)) = 1
  Compute d such that (d * e) % φ(n) = 1
  Public key = (e, n)
  Private key = (d, n)

RSA Encryption(m, e, n):
  c ← (m^e) % n
  return c

RSA Decryption(c, d, n):
  m ← (c^d) % n
  return m
```

Figure 4: Representation of the RSA algorithm, detailing key generation, encryption, and decryption of a message using modular arithmetic.
Source: Authors, (2025).

III. RELATED WORKS

Some work has already been done to measure the performance of Android applications, such as the one by [12], which made comparisons of applications running on an Android emulator under a Linux x86 system. The study concluded that native applications can be up to 30 times faster than a Java application executing the same algorithm, and this time can be improved up to 10 times if the Java application uses JNI. However, since the tests were executed on an emulator, the results may not fully reflect the reality of an embedded system. Additionally, the experiments were limited to calculations with mathematical integers, which may not be sufficient to capture the performance difference.

According to [13] executed 11 algorithms for the comparison between a pure Java application running a shared library via the virtual machine and one running the same library via JNI on specific hardware. In their results, they found that, overall, an application using JNI performs 34.20% better than one using the virtual machine. However, in 3 out of the 11 tests, the Java application performed better. Notably, the author developed their own task execution timer within their Java application, allowing a biased result when the system decides that this is not a priority task.

This paper is based on the work of [14], in which they used 6 algorithms and compared the results between a native application and a Java application with JNI. In contrast to what might be expected, their results indicated that the algorithms called via JNI were faster than those in the native application, except for Dijkstra's algorithm. The authors concluded that native applications were slower due to the native Android library, GNU C, and the compilation done with the GNU Compiler, in comparison with the bionic C/C++ library used in the development of the native layer of their JNI application.

A limitation in the authors' methodology is that each algorithm was executed 15 times, but it is not clear if there was variation in the inputs, as the result graphs are almost constant, with some slight variations that may be related to other processes that the operating system was running at the time. Another limitation is how the test timing was implemented, which was done via software within their target (system). Since Android is not a real-time operating system, this method of timing may lead to biased results when the system determines that certain tasks are not a priority.

IV. MATERIALS AND METHODS

In this work, we developed a pipeline for evaluating the execution time of native and Java applications using a timer external to the device running the application. We chose to use an external timer instead of one programmed within the applications to avoid potential bias caused by the system's task scheduling during the timing of execution. By doing so, we isolated the timer as an external device solely responsible for measuring the time of each execution.

Figure 5 provides a detailed view of the developed pipeline.

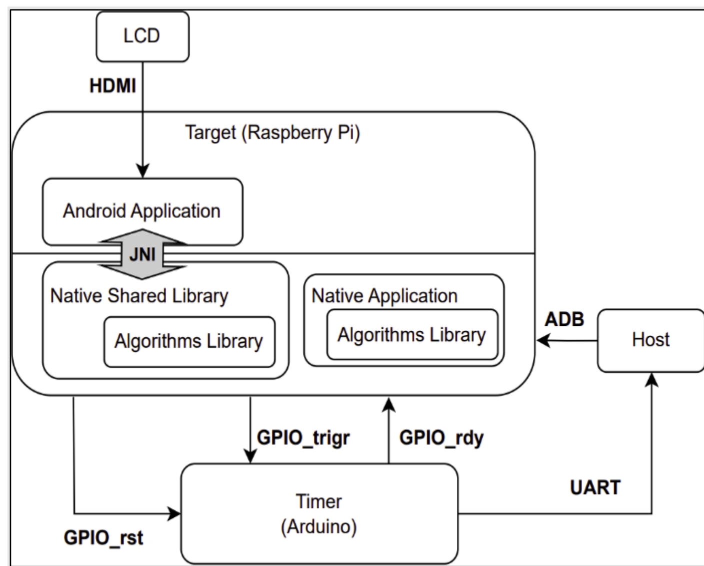


Figure 5: Developed pipeline for algorithm evaluation: the JNI application tests are initiated via LCD, while the native application tests are started through the Android Debug Bridge (ADB); When the target begins executing the algorithm, it triggers the external timer, and at the end of the test, it stops the timer.
Source: Authors, (2025).

The standard way to access a native application on an Android device is through the command line, as it is executed by an external machine referred to as the Host, via ADB. The method used to access the Java application is through a graphical interface, where the Target utilizes an LCD screen to display the options of algorithms to be executed.

When the Host machine or the LCD requests the execution of an algorithm, a sequence of executions is called according to the chosen algorithm. For instance, if the Quicksort algorithm tests are triggered via the LCD, at that moment, tests with 8 different input sizes are programmed, and each of these sizes will be executed 100 times. In other words, the counting is done 800 times in this example of sequence of executions.

When the test initialization is requested, the Target verifies if the timer is available to start the countdown. If it is not available, Target waits for its release; if it is available, it triggers the timer, initiating the countdown and starting the execution of an algorithm. When an algorithm completes its execution, the application triggers the timer again, ending the countdown, and at this moment, the timer sends the test result to the Host and notifies the Target that is available to count again.

For the test execution, the hardware chosen was Raspberry Pi 4B, and the operating system used was Android 13. To measure the time, an external device (Arduino Micro) controlled via General Purpose Input/Output (GPIO) by Target was utilized.

In the following subsections, we will explain each of the modules presented in Figure 5.

IV.1 HOST

For the host machine, represented in Figure 5 as 'Host', was used a computer with Ubuntu 20.04.6 LTS 64-bit system. The Host has two tasks in the developed pipeline, executing tests of the Android Native Layer via ADB and receiving the times of each test, both native and JNI, through Universal Asynchronous Receiver/Transmitter (UART) communication.

IV.2 TIMER

The execution time is calculated using an external hardware, represented in Figure 5 as 'Timer (Arduino)'. The Timer receives a pulse on a GPIO to indicate the start of an execution. During the execution, it signals that it is busy counting through another GPIO. When the execution is completed, another pulse is sent to the same GPIO, indicating that the counting can stop. Upon receiving the second pulse on the first GPIO, the Arduino writes the execution time in microseconds to the serial port, which will be received by the Host. Due to the external communication, this method takes some time that should be disregarded in the algorithm results, referred to as the 'communication offset' between the Target and the Timer.

IV.3 TARGET

The test target, represented in Figure 5 as 'Target (Raspberry Pi)', used Android version 13 ported to the Raspberry Pi 4B [15]. This platform featured the Broadcom BCM2711 SoC, with 4 ARM Cortex-A72 64-bit cores running at 1.8GHz and 8GB of RAM [16]. For cross-compilation, the Low Level Virtual Machine (LLVM) compiler infrastructure, which is the standard in current versions of the Android Open Source Project, (AOSP) was used in conjunction with Clang, the C/C++ compiler present in LLVM, both of which were included in the Android NDK (Native Development Kit). The version of the Android NDK used was r17c [17]. To access the interface of the JNI application, a 7-inch LCD screen with a resolution of 1024x600 pixels was utilized, and to execute the program of the native layer, the Host was used.

As the timer is an external hardware that communicates with Android via GPIO, it was necessary to develop a native library to access it. This library defines a class, called Timer, that encapsulates the management of GPIO in 3 methods that:

1) Indicate whether the timer is available.
2) Trigger and stop the timer.
3) Reset the test counting.

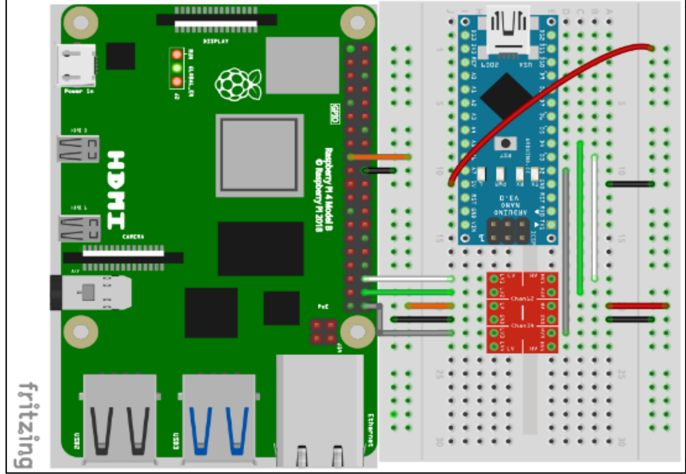


Figure 6: Circuit connections between the target and timer.
Source: Authors, (2025).

Figure 6 represents the circuit connection between the Target and the timer. To make external hardware accessible to the Java application layer on Android, a Hardware Abstraction Layer (HAL) must be created. To access the timer, the implemented HAL utilizes the Timer library mentioned in the previous paragraph and creates a service in the Android framework, enabling access through the JNI application.

IV.4 TARGET ALGORITHM LIBRARY

For the test execution on the target, we selected some well-established algorithms, such as Quicksort, Dijkstra, Fast Fourier Transform (FFT), and the Rivest-Shamir-Adleman (RSA) algorithm, which were implemented in a single library and statically compiled along with the executed binaries. Here is a brief description of the algorithms and how they were executed in this work:

- Quicksort, from the automation category, is an algorithm for sorting arrays. We executed arrays of lengths 1000, 2000, 4000, 6000, 8000, 10000, 12000, and 14000. For each array length, we performed 100 tests with the worst-case scenario for the algorithm, including arrays that are already sorted or have all elements equal.
- Dijkstra's algorithm, from the networks category, calculates the minimum cost between the vertices of a graph. Weighted graphs were run with vertex numbers of 200, 400, 600, 800, 1000, 1200 and 1400, and for each number of vertices 100 different randomly generated graphs with connections and weights were tested.
- Fast Fourier Transform (FFT), from the telecommunication category, decomposes a polynomial signal into the frequency domain. We executed inputs with power of 2, which indicates the degree of the polynomial signal to be transformed. The exponents used in the tests were 14, 15, 16, 17, 18, 19, and 20.
- The Rivest-Shamir-Adleman (RSA) algorithm, from the cybersecurity category, encrypts messages using a private and public key generated from prime numbers. With a fixed key, we encrypted and decrypted texts of lengths 2000, 4000, 6000, 8000, 10000, 12000, and 14000, with any ASCII character. For each length, we tested 100 different strings generated randomly.

IV.5 NATIVE APPLICATION

A Native Application is a type of application that uses native libraries, i.e. libraries that can communicate directly with the system. The Native Application is a type of application that makes use of native libraries, meaning libraries that can communicate directly with the system. As Android is a Linux-based system, this type of application is developed in C/C++, and after compilation, a single binary file is generated that can be executed using the command line provided by adb. Figure 7 represents the binary of the application with algorithm library compiled together with it.

Native Application

/vendor/bin/NativeTestApp

algorithm_lib

Figure 7: Representation of the native application compiled together with the algorithm library.
Source: Authors, (2025).

IV.6 JAVA APPLICATION USING JNI

Android applications developed in Java and Kotlin are the main ways for users to interact with a device, using a graphical interface. When compiled, the Java application generates a bytecode, which, differently from native applications that are executed directly by the system, requires a virtual machine to translate it. The virtual machine used by Android is called the Android Runtime (ART). Due to the distinct execution and compilation of Java and C/C++, the developed method for communication between these two types of languages is the Java Native Interface (JNI), which integrates a Java method to access functions from a shared native library. Figure 8 illustrates the flow of access by the Java application to the shared library, which, in this case, contains the functions from the algorithm library to be executed.

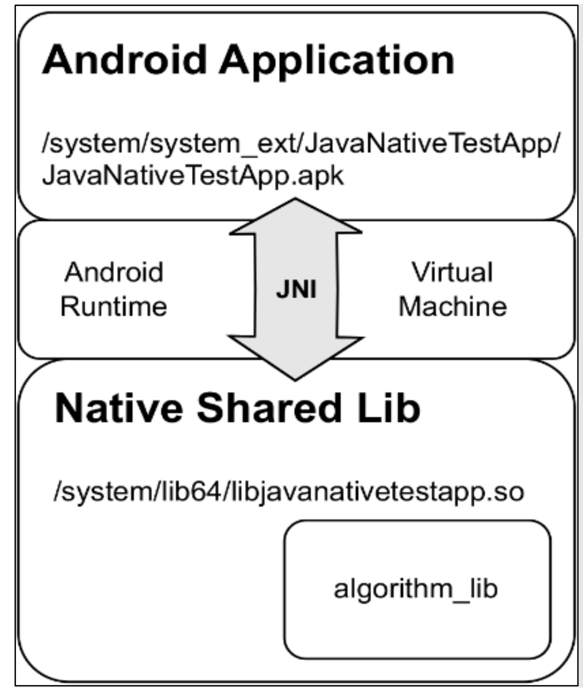


Figure 8: Representation of the communication flow between a Java application and a shared native library using JNI.
Source: Authors, (2025).

V. RESULTS AND DISCUSSIONS

The results were collected from the sequence of executions for each algorithm and plotted in graphs for better visualization. First, it was necessary to calculate the average of the communication offset between the applications and the Timer. This value is subtracted from the algorithm results to obtain a value that closely approximates the real execution time.

V.1 ESTIMANION OF THE COMMUNICATION OFFSET BETWEEN THE SYSTEM AND THE TIMER

Figure 9 shows the difference between the communication offsets for each application and the external timer. This means that the Timer is called without any algorithm running, resulting in only the time taken for the pulse to be sent twice – once to start the counting and another to stop it. The x-axis represents the number of executions, and the y-axis represents the time obtained in each repetition. The average of these times gives the value of the communication offset, which will be subtracted from the execution time of the algorithms.

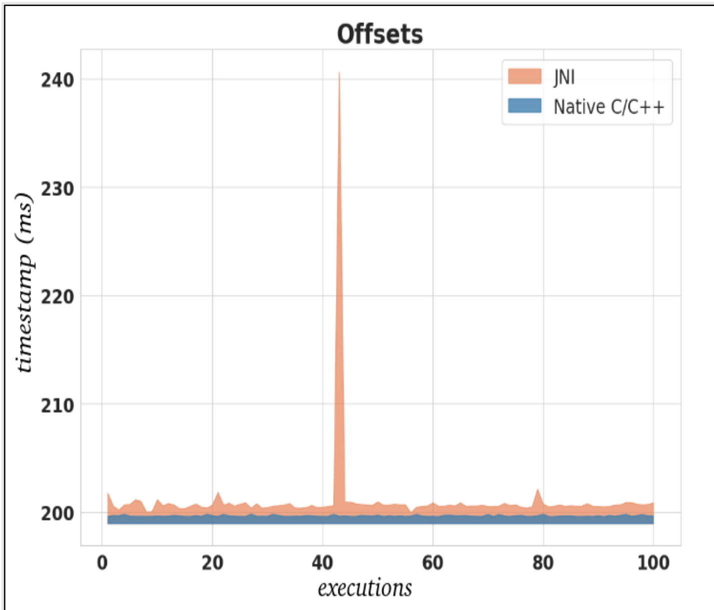


Figure 9: Overhead due to Communication in Java Applications (100 Runs).
Source: Authors, (2025).

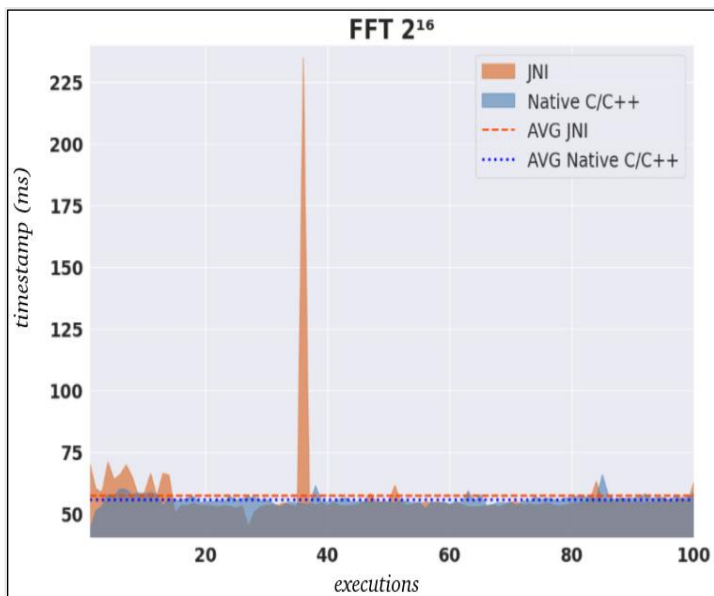


Figure 10: Variation in FFT Algorithm Execution Time for 100 Tests with 2^{16}. Inputs.
Source: Authors, (2025).

As shown, the native application had an average offset of 199.68ms, while the Java application had an average offset of 200.65ms. This small difference is possibly related to the layers that the system needs to pass to communicate with a JNI application.

V.2 CALCULATION OF ALGORITHM EXECUTION TIMES

Figure 11 presents the execution times of various algorithms for different input sizes. The x-axis represents the input size, while y-axis shows the execution time (excluding communication overhead). Each column displays the average execution time across 100 test runs for a specific input size, measured for both the native application and the JNI implementation.

Error bars indicate the standard deviation of these execution times. Figure 11a illustrates that the native application consistently outperforms JNI across all input sizes. Notably, the native application's execution time is between 3% (6,000 inputs) and 20% (2,000 inputs) faster than JNI.

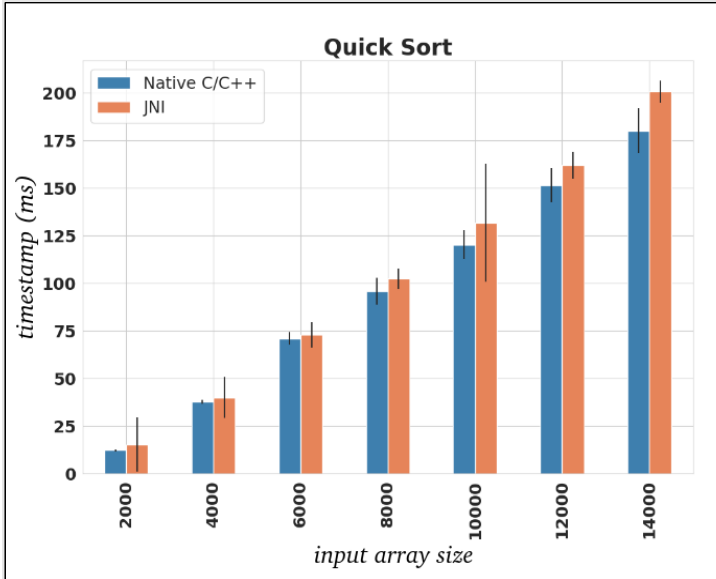


Figure 11a: Result obtained with the *Quick Sort* algorithm for input vectors with sizes ranging between 2000 and 14000.
Source: Authors, (2025).

Similar trends are observed in Figure 11b. The native application generally executes faster than JNI, with an average difference of 4% to 5.5%. In rare instances where JNI is faster, the maximum advantage is around 1.4% compared to the native application's average execution time.

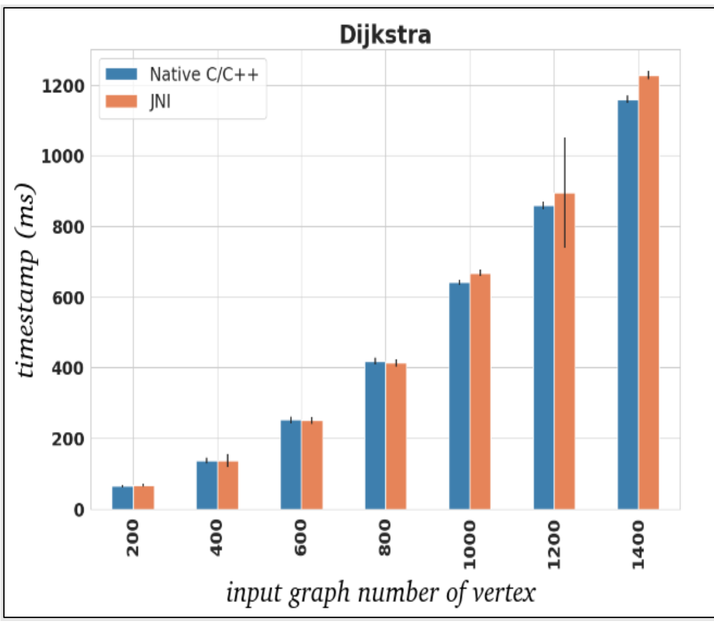


Figure 11b: Result obtained with the Dijkstra algorithm for graphs with number of vertices varying between 200 and 1400.
Source: Authors, (2025).

Figure 11c depicts the performance of the RSA algorithm. The native application demonstrates consistent speedup compared to JNI for all input sizes. However, the performance gap narrows with increasing input size. The native application exhibits a maximum efficiency gain of 6.5% (2,000 inputs) and a maximum of 0.7% (1,200 inputs) over JNI.

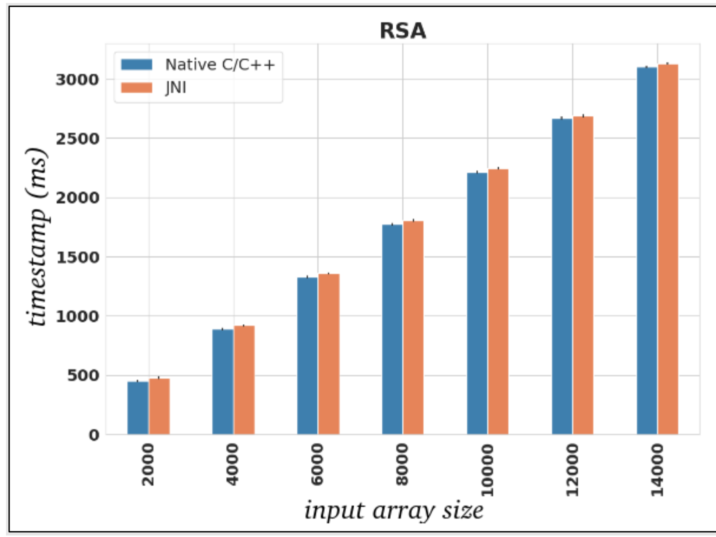


Figure 11c: Result obtained with the RSA algorithm for input vectors with sizes between 2000 and 14000.
Source: Authors, (2025).

In the analysis of the FFT algorithm, it was necessary to make a scale break as shown in Figure 11d, as the time variation according to the inputs was high, varying from 8.4 ms for 2^{14} inputs and 2194 ms for 2^{20} inputs. The native application is faster than JNI at most input sizes, with the average ranging between 3% and 12%. However, in cases where the JNI application is faster, there is a large variation, being 5% to 15% faster than the native one, on average executions.

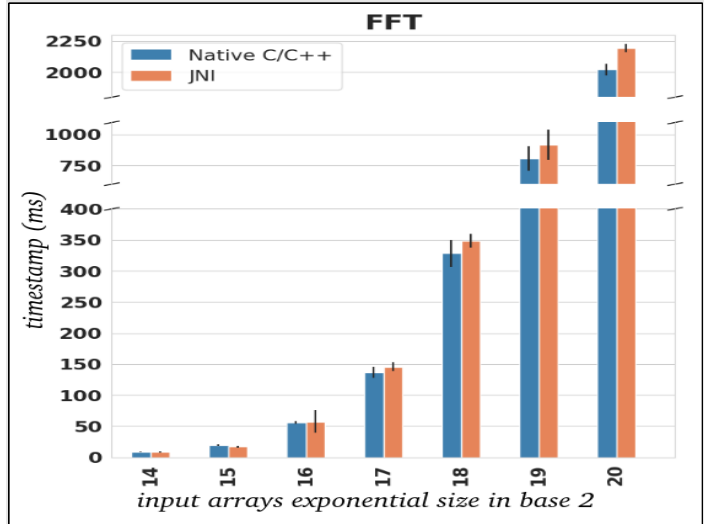


Figure 11d: Results of running the FFT algorithm on vectors size $2^{14}$to 2^{20}.
Source: Authors, (2025).

VI. ANALYSIS OF RESULTS

Our experiments show that native applications generally outperform Java applications using JNI, as shown in the graphs of Figure 11. This advantage stems from the way each application interacts with the system. Native applications have built-in algorithm libraries allowing their code to run directly with the system without needing translation. In contrast, Java applications using JNI require an extra layer of communication. This translation process through JNI adds some overhead, slowing down the execution.

These factors not only make native applications faster but also less prone to execution time variations. Figure 10 illustrates this point for the FFT algorithm with a 2^{16} input. The native application's execution time remains consistent, while the Java application's time fluctuates more. Although the Java application might be faster in rare moments, the native application is generally faster and more reliable.

These variations explain the rare instances where the Java application surpassed the native application. Specifically, this occurred with the Dijkstra algorithm for graphs with 600 vertices and de FFT algorithm for a 2^{15} input. The reason for these variations lies in Android. Since Android is a Linux-based system, it doesn't guarantee real-time performance. This means that in some situations, the system might prioritize other tasks, causing temporary slowdowns for the Java application. However, these slowdowns shouldn't be a common occurrence.

VII. CONCLUSIONS AND FUTURE WORKS

This article provides a comprehensive comparison between the performance of native C/C++ applications and Java applications utilizing a shared library accessed via the Java Native Interface (JNI) from the Android Native Development Kit (NDK). Both applications were compiled using the same toolkit to ensure a level playing field for comparison.

The experimental results demonstrate a consistent performance advantage for the native C/C++ application across all tested algorithms. Specifically, the performance improvements were significant, with the native application outperforming the Java application by up to 20% for the Quicksort algorithm. For Dijkstra's algorithm, the performance gain was 5.5%, while for the Fast Fourier Transform (FFT), the improvement was 12%. The RSA algorithm showed a performance enhancement of 6.5%. These results underscore the efficiency of native code execution, particularly in scenarios where computational intensity is high.

However, it is worth noting that for very small input sizes, the execution times for both types of applications were relatively faster and exhibited greater susceptibility to system variations. In these cases, there were instances where the Java application achieved faster execution times, influenced by system fluctuations and variations in processing load.

When comparing the results of this study with those reported by Kim, Cho, Kim, Hwang, Yoon, and Jeon [14], it is evident that the complete migration of AOSP to using the Bionic library and the LLVM compilation toolkit has significantly optimized the performance of native applications. This transition has resulted in native applications consistently outpacing Java applications that use JNI. The improvements in the Bionic library and LLVM toolchain have contributed to this enhanced performance by optimizing low-level operations and compilation processes.

Looking ahead, future research will explore additional performance comparisons by examining Java/native communication via JNI against communication using Binder Inter-Process Communication (IPC). Binder IPC, introduced in Android 10, represents a paradigm shift in the Hardware Abstraction Layer (HAL) development compared to the traditional JNI standard. This investigation will aim to assess how Binder IPC influences performance and efficiency in comparison to JNI, providing further insights into optimizing communication strategies within Android apqplications.

VIII. AUTHOR'S CONTRIBUTION

Conceptualization: Álison de Oliveira Venâncio.
Methodology: Álison de Oliveira Venâncio.
Investigation: Álison de Oliveira Venâncio.
Discussion of results: Álison de Oliveira Venâncio.
Writing – Original Draft: Álison de Oliveira Venâncio.
Writing – Review and Editing: Thales Ruano Barros de Souza, Bruno Raphael Cardoso Dias.
Supervision: Thales Ruano Barros de Souza, Bruno Raphael Cardoso Dias.
Approval of the final text: Álison de Oliveria Venâncio, Thales Ruano Barros de Souza, Bruno Raphael Cardoso Dias.

IX. ACKNOWLEDGMENTS

This work was supported by the training program of the Instituto de Pesquisas Eldorado. The research was conducted during the specialization course in Embedded Systems at the SENAI São Paulo Faculty of Technology – "Anchieta" Campus.

X. REFERENCES

[1] What is Android. Accessed: Sep. 26, 2024. [Online]. Available: https://www.android.com/what-is-android.

[2] Statcounter GlobalStats. Accessed: Sep. 26, 2024. [Online]. Available: https://gs.statcounter.com.

[3] Android Open Source Project. Accessed: Sep. 26, 2024. [Online]. Available: https://source.android.com.

[4] Android NDK. Accessed: Sep. 26, 2024. [Online]. Available: https://developer.android.com/ndk.

[5] S. Liang, "The Java Native Interface Programmer's Guide and Specification", 1st ed.: Addison-Wesley, 1999.

[6] A. Aftab, M. A. Ali, A. Ghaffar, A. U. R. Shah, H. M. Ishfaq, and M. Shujaat, "Review on performance of quick sort algorithm", International Journal of Computer Science and Information Security, vol. 19, no. 2, pp. 114-120, 2021.

[7] Y. Sun, M. Fang, M. and Y. Su, "AGV Path Planning based on Improved Dijkstra Algorithm", Journal of Physics: Conference Series, vol. 1746, no. 1, 2021.

[8] U. S. R. Murty, John. A. Bondy. "Graph Theory", 1st. ed.: Springer-Verlag, 2008.

[9] F. Mukhlif, and A. Saif, "Comparative study on Bellman-Ford and Dijkstra algorithms", in International Conference on Communication, Electrical and Computer Networks, 2020.

[10] H. A. Ghani, M. R. A. Malek, M. F. K. Azmi, M. J. Muril and A. Azizan, "A review on sparse Fast Fourier Transform applications in image processing", International Journal of Electrical & Computer Engineering, vol. 10, no. 2, pp. 1346-1351, 2020.

[11] A. B. Alhassan, A. H. Mahama and S. Alhassan, "Residue architecture enhanced audio data encryption scheme using the Rivest, Shamir, Adleman algorithm", International Journal of Advanced Engineering and Technology, vol. 6, no. 2, pp. 21-29, 2022.

[12] L. Batyuk, A.D. Schmidt, H.G. Schmidt, A. Camtepe and S Albayrak, "Developing and Benchmarking Native Linux Applications on Android", Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 7, pp. 381-392, 2009.

[13] C. M. Lin, J. H. Lin, C. R. Dow and C. M. Wen, "Benchmark Dalvik and Native Code for Android System", in Second International Conference on Innovations in Bio-inspired Computing and Applications, Shenzhen, China, 2011, pp. 320-323, doi: 10.1109/IBICA.2011.85.

[14] Y. J. Kim, S. J. Cho, K. J. Kim, E. H. Hwang, S. H. Yoon and J. W. Jeon, "Benchmarking Java application using JNI and native C application on Android," in 12th International Conference on Control, Automation and Systems, Jeju, Korea (South), 2012, pp. 284-288.

[15] Android for Raspberry Pi4. Accessed: Sep. 26, 2024. [Online]. Available: https://github.com/android-rpi/device_arpi_rpi4.

[16] Raspberry Pi4 Model B. Accessed: Sep. 26, 2024. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b.

[17] Google LLC. (2024). NDK Revision History. Accessed: Sep. 26, 2024. [Online]. Available: https://developer.android.com/ndk/downloads/revision_history.