Received November 12, 2021, accepted December 8, 2021, date of publication December 10, 2021, date of current version December 21, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3134788

Fancier: A Unified Framework for Java, C, and OpenCL Integration

SERGIO AFONSO AND FRANCISCO ALMEIDA
Department of Computer Engineering and Systems, Universidad de La Laguna, 38200 San Cristóbal de La Laguna, Spain

Corresponding author: Sergio Afonso (safonsof@ull.es)

This work was supported in part by the Ministry of Science, Innovation and Universities under Project PID2019-107228RB-I00 and Grant FPU16/00942, in part by the European Commission (EC) [European Regional Development Fund (ERDF)] and the Spanish Ministerio de Asuntos Económicos y Transformación Digital (MINECO) under Project TIN2016-78919-R, in part by the Government of the Canary Islands under Project ProID2017010130, and in part by the Computación de Altas Prestaciones sobre Arquitecturas Paralelas Heterogéneas (CAPAP-H) Network.

ABSTRACT Graphics Processing Units (GPUs) have evolved from very specialized designs geared towards computer graphics to accommodate general-purpose highly-parallel workloads. Harnessing the performance that these accelerators provide requires the use of specialized native programming interfaces, such as CUDA or OpenCL, or higher-level programming models like OpenMP or OpenACC. However, on managed programming languages, offloading execution into GPUs is much harder and error-prone, mainly due to the need to call through a native API (Application Programming Interface), and because of mismatches between value and reference semantics. The Fancier framework provides a unified interface to Java, C/C++, and OpenCL C compute kernels, together with facilities to smooth the transitions between these programming languages. This combination of features makes GPU acceleration on Java much more approachable. In addition, Fancier Java code can be directly translated into equivalent C/C++ or OpenCL C code easily, which simplifies the implementation of higher-level abstractions targeting GPU or parallel execution on Java. Furthermore, it reduces the programming effort without adding significant overhead on top of the necessary OpenCL and Java Native Interface (JNI) API calls. We validate our approach on several image processing workloads running on different Android devices.

INDEX TERMS Application programming interfaces, hardware acceleration, heterogeneous systems, image processing, mobile computing, parallel programming, performance analysis.

I. INTRODUCTION

Heterogeneous computing devices based on parallel architectures, composed by multi-core processors and accelerators, have become ubiquitous on most domains. Whereas previously the performance advantages of these architectures benefitted only large-scale high-performance computing systems, now even low-power mobile and embedded devices are based on these architectures. This has enabled opportunities to use mobile architectures for increasingly compute-intensive applications, such as image processing, computer-generated imagery, etc. Taking advantage of these architectures requires software to be developed taking their specific features into account. Consequently, compute-intensive applications can see dramatic performance improvements if these architectures are efficiently exploited, but this can require the development of parallel code. Existing programming models, tools, and techniques for parallel code development can be still greatly improved, as there currently exists a significant programmability barrier compared to traditional programming. Parallel programming models are more complex, which adds development cost, resulting in a much lower adoption and eventually lower application performance. This is an issue present on the desktop and server space as well, but the mobile and embedded domains are in particularly early stages of adoption of these types of programming models.

Since public adoption of mobile platforms is so pervasive and their unused performance potential is so high, it is important to consider compatibility with these platforms in the design of new parallel programming models and tools. The vast majority of modern mobile devices are running the Android or iOS Operating Systems (OS), the former being much more widespread. The development of Android applications is mainly done in the Java and Kotlin programming languages, both seamlessly interoperating and running on top of the same runtime due to them being compiled into Java bytecode as part of the build process. Using this type

The associate editor coordinating the review of this manuscript and approving it for publication was Michele Nappi.

of managed high-level programming languages simplifies the development of interactive applications, at the cost of adding overhead that hinders execution performance. Consequently, providing parallel and accelerated execution capabilities to Java-based applications can provide important performance benefits for server, desktop, mobile, and embedded applications, due to the widespread adoption of this language among all these platforms which are also commonly based on heterogeneous parallel architectures. This potential has been explored for several years, evidenced by all the work around Java Grande [1], or using Java for large scale parallel applications. However, we believe that, even though the programmability problem of creating these types of Java applications has not been completely solved, the demands of the mobile sector make this issue more relevant than ever.

Typically, managed programming languages, such as Java, tend to run slower than native ones, like C/C++, due to the overhead of their runtime systems and their higher level of abstraction from the hardware and OS. However, some of these shortcomings have improved over time due to improvements on Virtual Machine optimizations and Just-in-Time (JIT) compilation techniques [2]. Nevertheless, Java execution on accelerators still requires the use of specialized Java Virtual Machines (JVMs) or libraries, or code translation tools integrating Java bytecode execution and native libraries with low-level access to such accelerators. There exist multiple approaches that tackle the issue of accelerating Java applications by reducing the programming complexity in different ways. Project Sumatra [3] and TornadoVM [4] are built on top of JVMCI (Java-Level JVM Compiler Interface) [5], a Java API allowing the integration of custom Java compilers into a running JVM. They can implement transparent code optimization techniques as if they were part of the JVM, but their main disadvantage is their requirement for the underlying JVM to support this interface. Due to security concerns arising from the given ability to completely replace code at runtime, it is unlikely this feature will be exposed to non-system code in Android. This limitation can be circumvented through the use of GraalVM Native Images [6] in place of the officially supported Android SDK and runtime, but that would result on a significant increase in development effort. On the other hand, working through JVMCI has the benefit of being able to decide at runtime, through a JIT compilation process, what processor to target for each task, migrating tasks to different processors according to observed behavior over time, and to seamlessly integrate with existing standard Java APIs. Tools like Paralldroid [7], Aparapi [8], Rootbeer [9], or ParallelME [10], on the other hand, do not impose that requirement, and they can translate Java code or compiled bytecode into native or accelerated implementations. Each of these alternatives define their own parallel programming model on top of Java, adding certain restrictions over what Java code is supported for parallel or accelerated execution. This makes it difficult to port parallel code written for one of these environments to another, and most of them have to solve a similar set of challenges, mainly relating to memory management, on top of making their particular parallel programming model work efficiently. There is a need for middleware that could handle their common issues, so that tools designers can focus on the parallel programming models and their runtime performance, while reducing the learning curve for application developers using them. Other alternatives such as language bindings like JCUDA [11] are targeted towards experts and do not reduce the programming effort significantly.

Despite these efforts, none of the existing approaches has been widely adopted by the developer or scientific community, so there does not exist a standard system implementing hardware acceleration of Java code yet. This means there is still room for new alternatives. In all mentioned tools, many features of the Java language are unsupported or highly inefficient to use on accelerators or even parallel or sequential native execution. In this work, where we present our Fancier framework,[1] we propose a common Java API that simplifies the automatic production of efficient native code for acceleration, which can be used as a platform to build independent automatic acceleration tools. This API, built on top of the OpenCL 1.1 standard libraries, includes fixed-size vector data types, a math library, and multiple containers for primitive data types and images, and it is designed to emulate value semantics used in native languages, allowing a simpler mapping of Java code to C/C++ or OpenCL for accelerators. OpenCL 1.1 has been chosen as the hardware acceleration backend due to its low overhead, support for general-purpose computations, fine-grained hardware control, and widespread adoption among all types of architectures, from low-power SoC to state-of-the-art high-performance computing accelerated distributed systems. The Java math library included in Fancier unifies the functions provided by the standard Java `Math` class and the OpenCL standard math library. Furthermore, the provided containers feature transparent zero-copy memory support on unified memory systems, while giving direct access to the memory from the Java, native, and OpenCL contexts. This is especially relevant on mobile SoC, where this type of memory architecture is the most common and where memory copy overhead can easily become a performance bottleneck. In addition to the Java API, Fancier provides a C/C++ API providing the same operations as the former, both being modeled after the OpenCL C language for accelerators. This means that Fancier unifies these three languages so that code can be easily ported from one to another, depending on the requirements for its execution. Lastly, the Fancier Native API provides functions and data structures to help with the integration of C/C++ execution within a Java application, greatly reducing the effort demanded by the use of the Java Native Interface (JNI) and improving readability without resulting on a measurable performance overhead. Our approach does not demand any modification to the Java compiler or the JVM, and it only depends on OpenCL.

[1] https://github.com/HPC-ULL/Fancier

There are several advantages to our approach. Programming models and tools for the automatic acceleration of Java code need to limit language features in order to produce performant and portable native code, and we propose a single independent and extensible Java subset easily translatable to native programming languages. Its adoption as an underlying platform would result in a significant reduction in the cost of creating such tools, as well as giving application developers using them a more homogeneous and stable set of data structures and functions, reducing their learning curve and simplifying the process of porting code from one parallel programming model for Java to another. Considering that the Fancier framework is designed to support many different approaches to defining parallel Java kernels for accelerated execution, it is clear that it needs to support particular requirements over memory management or runtime behavior demanded by those. Fancier features a native plugin system through which reusable components supporting particular behaviors and runtime requirements can be created easily. Plugins have access to the core Fancier Native API, greatly simplifying their implementation due to the seamless passing of data across Java, C/C++ and OpenCL. These plugins can support higher level parallel programming models built on top of Java while maintaining the core features of Fancier, and they can also implement specialized behaviors or code optimizations that multiple applications can benefit from. Given the ever-increasing amount of hardware architectures for multi-core processors and accelerators, programming models for parallel execution must be cross-platform in order to be widely adopted. For that reason, the Fancier framework is based exclusively on stable and well supported multi-platform standards such as Java, JNI, C, and OpenCL. In addition, it does not introduce any dependencies on specific Java compilers or JVMs and only runs on user-space, requiring no particular support from the underlying OS.

The main contributions of this work are the following:

1) The acceleration of Java applications is not prevalent in part due to the characteristics of the language and its runtime behavior, and in part due to development difficulty. We work towards a solution to these issues by introducing the Fancier framework, which facilitates the translation of Java to C/C++ and OpenCL, and their runtime integration by creating a unified interface and library. It defines a Java and C/C++ subset of features and functions where the mismatch between reference semantics of Java and the value semantics of C/C++ and OpenCL is addressed. By adopting this framework, the process of accelerating code within a Java application can be divided into stages, allowing a progressive and seamless acceleration of that code. Initially, an easy to develop and debug Fancier Java implementation would be created, porting that to Fancier Native after it has been debugged, and finally replacing the native kernel for an high-performance OpenCL one. The unification of the Java, Native, and OpenCL languages provided by our framework enables this methodology, which greatly reduces development cost and has no performance penalty.
2) The information flow between Java, C/C++ and OpenCL is complicated and requires manual management. There exist multiple ways of performing this, but most incur performance penalties that may go unnoticed. We analyze these alternatives and implement an optimal strategy for Fancier container data types that works transparently and significantly reduces development cost. Unified memory systems introduce the possibility of sharing memory buffers between a host processor and accelerators, which can provide great performance gains. However, the optimal management of this type of memory becomes difficult when buffers must be accessed from managed, native, and accelerated contexts. Our presented memory management strategy features efficient and transparent zero-copy read and write access from all contexts.
3) We evaluate the performance of Fancier Java, Native, and OpenCL implementations of a wide range of image processing kernels on various Android devices using a reliable methodology to present reproducible metrics. In these benchmarks, we show that Fancier Native code is highly optimized and can outperform regular C/C++ implementations in many cases, and Fancier Java code can be easier to develop, understand, and translate, or to give good performance depending on the use case.

This paper is structured as follows: Section II introduces the process by which Android applications are compiled and executed, Sect. III presents the Fancier framework in detail, Sect. IV includes the performance experiments used to validate our approach, and in Sect. V we give our conclusions and future related lines of work. We include extended performance data produced by our experiments in Appendix A.

II. APPLICATION EXECUTION ON ANDROID

A. MANAGED EXECUTION

Android applications are mainly developed using the Java and Kotlin managed programming languages. These languages simplify the development of interactive applications while providing the performance they require. As they both are compiled into Java Bytecode, different parts of an Android application can be implemented using any of these managed languages and their interoperation is close to transparent. However, their higher level of abstraction from the hardware comes with a performance cost over native programming languages. In order to mitigate that, since Android 5.0, Java Bytecode is transformed and optimized in various stages until it gets ahead-of-time (AOT) compiled into binaries that run natively with help from the Android Runtime (ART) [12]. Figure 1 shows the compilation and execution stages of an accelerated Android application.

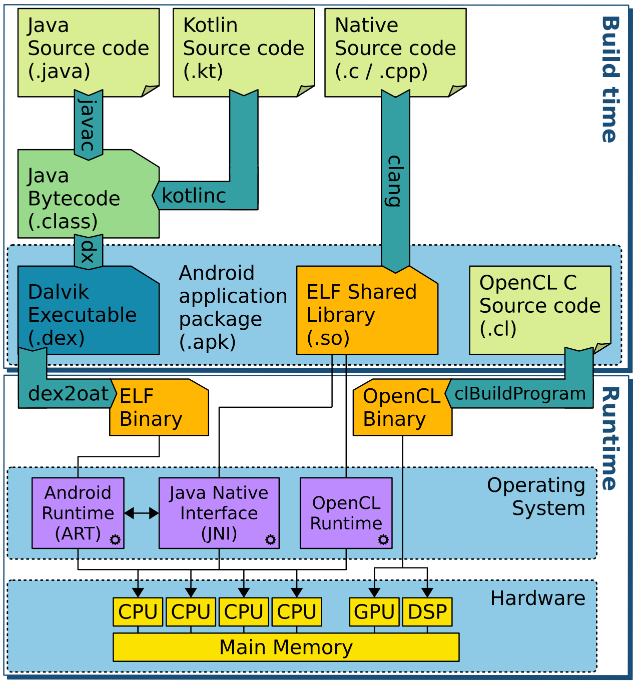


FIGURE 1. Compilation and execution of an Android application.

More recent Android releases, instead of AOT-compiling at install-time all Dalvik executables produced from Java bytecode, use a hybrid just-in-time (JIT) and AOT profile-guided optimization process [13]. This helps reduce install-time overhead without giving up noticeable application performance over time, but this behavior needs to be taken into account to properly benchmark managed code affected by it. Previous Android releases relied on a pure JIT process, more similar to standard Java Virtual Machine (JVM) implementations like HotSpot [14]. Even though this multi-stage optimization process allows for very highly efficient execution of Java and Kotlin code, there are cases where their performance becomes insufficient. This is where native programming languages are better suited to the problem.

B. NATIVE EXECUTION

Java applications can interface with native code through the use of the Java Native Interface (JNI) [15]. In Android, this is possible as well since Android's Native Development Kit (NDK) supports JNI [16]. It is a standard consisting of a series of C data types and functions that allow the interaction of C code with a running JVM–calling Java methods, creating and managing objects, and reading and writing to managed memory. For a Java application to offload some computation into native code, it needs to declare and call a method or methods marked as `native`, which in turn need to be implemented natively and compiled into a shared library to be loaded by the Java application. The native implementation of the method would use some JNI API calls as a bridge between Java and C in addition to the actual work to be done by it, adding some development complexity and execution overhead. These shortcomings, however, come with several advantages as well:

- It allows applications to link to and use Android-supported native system libraries, such as OpenGL and Vulkan. Additionally, other native libraries may be made independently available by device vendors.
- If non-trivial amounts of an application depend on existing native code, it makes sense using JNI to integrate it instead of re-implementing it in Java or Kotlin.
- Some highly compute or memory-intensive codes may still see important performance improvements if implemented natively, in spite of the multiple stages of optimization that are applied to managed languages.

C. ACCELERATED EXECUTION

Support for hardware acceleration of code execution is important for applications that contain very compute-intensive parallel code. By offloading execution of these types of codes to specialized processors like GPUs, dramatic performance improvements can be achieved. Officially, the acceleration of regular Android applications has been possible through the use of Renderscript, an Android-specific SIMT-based (Single Instruction, Multiple Threads) parallel programming language [17]. After a Renderscript kernel is written, an associated Java class is generated by the Android Build Tools, which lets the Java or Kotlin code control the integration between said kernel and the rest of the application. Even though this has been the only supported method for acceleration of general-purpose code on Android, it has been announced that it would be deprecated starting in version 12.

Conversely, OpenCL is a multi-platform standard and framework for general-purpose accelerated execution that has existed for longer than Renderscript, and it has enjoyed continued support by most of the main SoC vendors at the core of modern smartphones, such as Arm, Qualcomm, MediaTek, and HiSilicon. Even though this alternative has not seen official support from Android, its widespread adoption among platforms ranging from energy-efficient SoC to high-performance computing systems, together with its higher performance potential and finer-grained control, makes it an interesting alternative. Additionally, the officially supported framework for Android acceleration from version 12 is Vulkan, a multi-platform standard focused on computer graphics for which work is being done to allow running OpenCL kernels [18]. OpenCL is our best option as this work is focused on general-purpose cross-platform acceleration.

The main disadvantage of OpenCL or Vulkan compared to Renderscript is that, by being native libraries, they demand a higher development effort to integrate into a managed application written in Java or Kotlin. Furthermore, their high level of control and granularity over execution comes with a steeper learning curve. This is why approaches like those presented in this paper are still relevant and necessary.

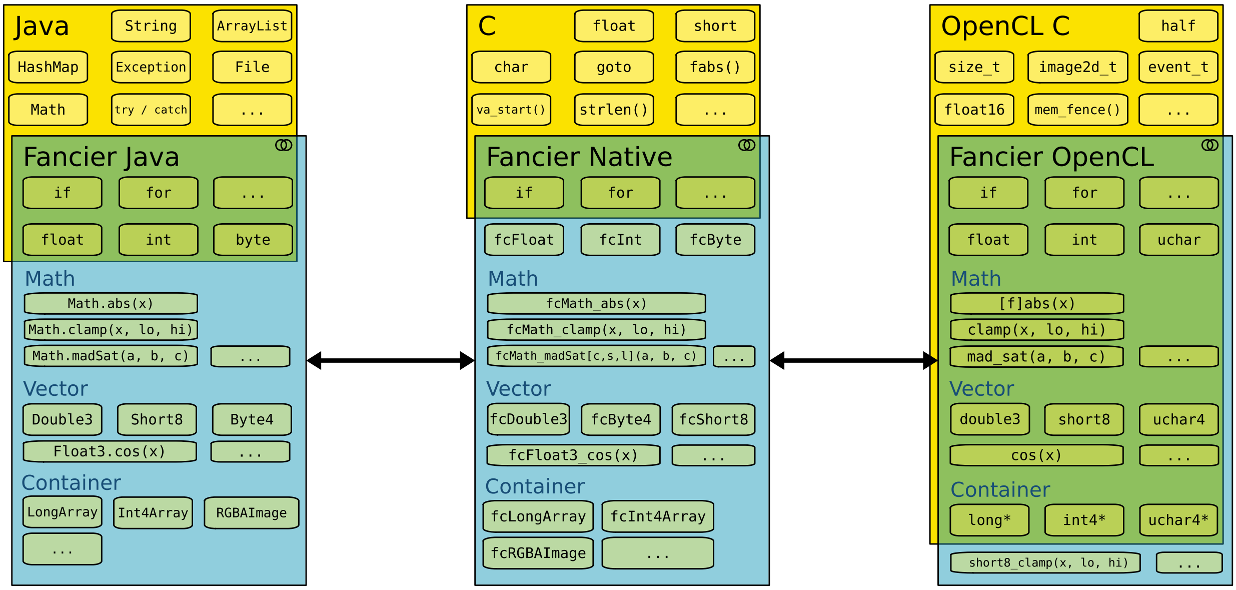


FIGURE 2. **Overview of the different layers of the Fancier framework.**

III. THE FANCIER FRAMEWORK

A. OVERVIEW

The Fancier framework is a multi-platform, modular, and extensible programming interface to facilitate the acceleration of Java applications without requiring any special features from the JVM they are running in. It achieves this by defining a Java and Native API modeled after the OpenCL C programming language for accelerators, and by providing several utilities that simplify the integration of C/C++ and OpenCL execution to a Java application efficiently. Fancier Java code is designed to be used as an expressive subset of Java that can easily be mapped into high-performance Fancier Native or OpenCL C code by an automated code transformation tool, and Fancier Native code is designed to be either a result of such a transformation or developed by hand. An overview of the mapping between the programming models defined on the various Fancier API layers is represented in Fig. 2. In simpler terms, Fancier defines a standardized Java subset for accelerated execution and provides a runtime to efficiently integrate C/C++ and accelerated execution into a Java application as described in Sect. III-B. This facilitates both the design and implementation of parallel accelerated programming models on top of Java and the manual development of accelerated Java applications.

The actual structure of this framework, which is template-based and targets Linux and Android platforms through various Java and native libraries, bundled with certain OpenCL C helper code, is discussed in Sect. III-C. Section III-D details the process by which Fancier Java code can optimally share memory with C/C++ and OpenCL on unified memory systems, avoiding unnecessary memory copies while transferring control between programming languages or processors.

B. DESIGN

Fancier uses OpenCL as the high-performance multi-platform backend for accelerated execution. OpenCL is a dual-source programming framework where code running on the host CPU and code for accelerators (kernels) are defined separately and have different purposes. Whereas host code manages memory buffers and sets up the execution parameters, kernel code defines the actual parallel computation. With regards to this, it is important to note that OpenCL traditionally features a partitioned memory space, where host and accelerator memory are separate, and updating data in both spaces must be done manually by OpenCL host code. In the Java to OpenCL execution path, the transition from managed Java memory to OpenCL host memory must be manually orchestrated, as well, making it a non-trivial interaction that Fancier resolves as described in Sect. III-D. Fancier augments Java and C/C++ code in a manner such as to make them the same as the equivalent OpenCL kernel code, so that they are as equal as possible – unifying them. Fancier significantly reduces the programming overhead of OpenCL host code by automatically handling memory buffers and runtime kernel compilation in an efficient manner. The multi-platform characteristic of OpenCL is the main reason why it has been chosen as the platform on top of which to build Fancier, as that allows it to accelerate applications in systems ranging from high-performance servers with accelerators to low-power heterogeneous devices.

One of the main obstacles for transforming Java code in order to run on accelerators is its reference semantics. Any object in Java, except primitive data types, is handled via references. This is in contrast to the type of native programming languages used for accelerated execution, such as OpenCL or CUDA, where value semantics are used instead. This provides them with code optimization advantages. Java code only using primitive Java data types and Fancier objects, designed with this feature, can easily be transformed into a native implementation, since they are almost equal to Fancier Native and OpenCL C kernel code. Modularity is another one of Fancier's advantages, as it has been designed to be extended depending on particular needs. Native runtime extensions can be defined using its plugin system, ensuring that the core Fancier functions can support their execution.

There are three main layers to Fancier: Java, Native, and OpenCL. The Fancier Java API is designed for conciseness, ease of use and simplicity to transform into a native implementation. Performance is not a consideration in this case because it is not intended to be executed directly. However, we provide an alternative higher-performing set of Java methods on the same Java classes with the purpose of removing the main performance bottleneck of that API, which is the creation of small objects. The way Fancier forces value semantics is by always returning a new result object when operating with Fancier objects, so by pre-allocating objects for all intermediate results we reduce pressure on the Java memory management system. We see the main use case for this higher-performing API would be debugging, since it still closely matches the simplest possible Fancier implementation, but it runs significantly faster, as we show in Sect. IV-D. We believe that higher-performing Fancier Java implementations should be automatically generated from concise Fancier Java code, so that application developers could write a single implementation and target different Fancier APIs depending on the task. The Fancier Java API defines fixed-size vector data types, primitive and vector arrays, 2D images, and a math library, giving it the main functions exposed by the OpenCL C programming model. Vectors of sizes 2, 3, 4, and 8 are supported, for the types of `Byte`, `Short`, `Int`, `Long`, `Float`, and `Double`. The Fancier math library contains the main functions from the standard Java `Math` class, and the math functions included in the OpenCL C standard library. These are implemented natively and called through JNI from Java. Fancier vector data types include methods for applying these math functions to each element of the vector. Fancier Java arrays (e.g., `IntArray`, `Byte4Array`, ...) and `RGBAImages` are allocated natively to be able to take advantage of the unified memory between main CPU and accelerators, so they are not guaranteed to be automatically garbage collected if there are no more references pointing to them. However, they can be created in try-with-resources statements to prevent memory leaks, as well as manually calling their `release()` method. The way in which these containers are allocated and managed is detailed in Sect. III-D. In Fig. 3 we show a possible Fancier Java implementation of a conversion to gray scale. It is a simple sequential implementation that uses vector data types and operations to reduce the development workload. Classes and methods of the public Fancier Java API are highlighted.

The Fancier Native API is a C library containing the same public functions and data structures as its Fancier Java counterpart, but it also provides several other facilities to simplify Java to C/C++ and C/C++ to OpenCL interactions. Each math function, initialization and release, and getter and setter method of container classes, is implemented natively by the way of two functions: a JNI wrapper, and the C implementation that gets called by said wrapper and other native code. Vector data types mirror the design of the Java counterparts, as well, so they are closely matched. Figure 4, showing a Fancier Native implementation of the Gray Scale kernel, is a clear example of how closely this API follows the Fancier Java counterpart, and what the development effort for using this API is. The required Java code is trivial, as it only needs to declare the method signature and mark it as `native`. Then, up to line 27, the Fancier Native sequential implementation of the kernel contains the equivalent function calls the Fancier Java implementation shown in Fig. 3 used. Additionally, the JNI code bridging Java and native execution is shown, starting on line 29. It is responsible for obtaining the C/C++ representation of the Java objects passed as parameters, ensuring that they can be read and written by native code, and passing them to the previously described Fancier Native implementation. Thanks to the use of Fancier Native utilities for JNI and Fancier Java classes, this code is much simplified from the alternative of using JNI on its own, while adding no execution overhead. Additionally, it provides a simple logging API and a method of reporting errors as Java exceptions that the main application can capture and handle. In contrast to Fancier Java, the Fancier Native API is high-performance, although it has a higher development cost if used manually. Its strength is that its added development cost could be completely eliminated by the creation of a code transformation tool, due to simpler Fancier Java implementations containing all the information necessary to produce Fancier Native counterparts.

```java
public class GrayScale {
  public static void run(RGBAImage input,
                         RGBAImage output) {
    final int width = input.getWidth();
    final int height = input.getHeight();
    final Float3 weights =
      new Float3(0.299f, 0.587f, 0.114f);

    for (int y = 0; y < height; ++y) {
      for (int x = 0; x < width; ++x) {
        Byte4 pixelIn = input.get(x, y);
        byte gray = (byte) Float3.dot(
          pixelIn.asByte3().convertFloat3(),
          weights);
        output.set(x, y, new Byte4(gray,
          gray, gray, pixelIn.w));
      }
    }
  }
}
```

FIGURE 3. Example Fancier Java sequential implementation of gray scale.

During initialization, Fancier automatically configures the OpenCL environment, by creating the global and application-accessible `cl_platform_id`, `cl_device_id`, `cl_context`, and `cl_command_queue` objects, as well as an information structure holding various features about the

```
public class GrayScale {
  public static native void run(
    RGBAImage in, RGBAImage out);
}
```

```
inline int index(int width, int x, int y) {
  return y * width + x;
}

void gray_scale(fcRGBAImage* input,
                fcRGBAImage* output) {
  const int width = input->dims.x;
  const int height = input->dims.y;
  const fcFloat3 weights = fcFloat3_create111(
    0.299f, 0.587f, 0.114f);

  const fcByte4* in = input->pixels->c;
  fcByte4* out = output->pixels->c;

  for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
      int i = index(width, x, y);
      fcByte4 pixel_in = in[i];
      fcByte gray =
        fcFloat3_dot(fcByte3_convertFloat3(
          fcByte4_asByte3(pixel_in)),
          weights);
      out[i] = fcByte4_create1111(gray,
        gray, gray, pixel_in.w);
    }
  }
}

JNIEXPORT void JNICALL
Java_com_package_GrayScale_run(JNIEnv* env,
    jclass cls, jobject jni_in,
    jobject jni_out) {
  fcRGBAImage* in =
    fcRGBAImage_getJava(env, jni_in);
  fcRGBAImage* out =
    fcRGBAImage_getJava(env, jni_out);

  if (!fcRGBAImage_valid(in) ||
      !fcRGBAImage_valid(out)) {
    fcException_throwNative(env, __FILE__,
      __LINE__, "GrayScale_run",
      FC_EXCEPTION_BAD_PARAMETER);
    return;
  }

  fcError err = fcRGBAImage_syncToNative(in);
  FC_EXCEPTION_HANDLE_ERROR(env, err,
    "syncToNative:in", FC_VOID_EXPR);

  err = fcRGBAImage_syncToNative(out);
  FC_EXCEPTION_HANDLE_ERROR(env, err,
    "syncToNative:out", FC_VOID_EXPR);

  gray_scale(in, out);
}
```

FIGURE 4. Example Fancier native sequential implementation of gray scale.

selected accelerator, as reported by `clGetDeviceInfo()` function calls. The current implementation chooses a single OpenCL platform and device, which it uses to create the context and command queue objects that would be used by the application. This simplifies targeting single-accelerator mobile and embedded systems like those on which we are currently evaluating our approach, but it would be of great interest to also cover the multi-accelerator case that is more common on desktop and server environments, as well as some next-generation high-performance SoC. This addition is entirely feasible within the bounds of the presented approach, as it is designed for such extensions and its impact on the public Fancier APIs would be localized. We enable the community to investigate interesting related features by making our framework open source.

Accelerated Java applications using Fancier for OpenCL execution are able to take full advantage of the complete set of features of the OpenCL 1.1 standard, with the advantage that the passing of data between the Java, C/C++ and OpenCL layers is much simpler, and the control flow between Java and C/C++ is more seamless. In this sense, Fancier OpenCL applications can do what any OpenCL application can, but with a reduced development workload. Application developers would use standard OpenCL functions and global OpenCL objects created by Fancier directly from C/C++ to set up the execution of parallel kernels, and take advantage of Fancier containers to take care of memory buffer creation and movements. The transparent process by which these buffers are managed, detailed in Sect. III-D, is currently tailored to unified memory architectures, but the implementation can be easily extended to improve performance on non-unified memory systems, which is the case for most desktops and servers. Parallel kernels can be executed synchronously or asynchronously, depending on application needs, but considering that the command queue created by Fancier is in-order and the memory synchronization functions of Fancier containers are blocking, safe and efficient asynchronous execution of Fancier OpenCL code is easily attainable. Just as with any other OpenCL-accelerated application, the simultaneous execution of multiple Fancier applications would result in a reduction of performance due to competition for computing resources. In that case, the OS would be responsible for scheduling the access to the accelerator, as Fancier applications do not communicate among themselves.

All Fancier containers are allocated through OpenCL API calls. This means that, no matter if a Fancier array or image is created in Java or C/C++ code, it has associated with it an OpenCL memory buffer that can be used to support parallel execution on accelerators. The provided functions that developers would need to use in order to manage these memory buffers, as detailed in Sect. III-D, are much simpler than the alternative of manually writing JNI and OpenCL API calls to accelerate the execution of a Java application. The Fancier Native API provides some utilities for obtaining and compiling OpenCL C kernels at runtime. These utilities automatically add certain math functions supported by Fancier Java and Fancier Native which are not included in the OpenCL C standard library. This way we ensure that every function supported by the Fancier framework is available in Java, C/C++, and OpenCL C. In Fig. 5, an implementation of the Gray Scale kernel in OpenCL C, matching closely the previously described Fancier Java and Fancier Native implementations, is shown. It is important to note that this

```
inline int index(int width, int x, int y) {
  return y * width + x;
}

kernel void gray_scale_ocl(
    global const uchar4* in,
    global uchar4* out) {
  const float3 weights =
    (float3)(0.299f, 0.587f, 0.114f);
  const int i = index(get_global_size(0),
    get_global_id(0), get_global_id(1));

  const uchar4 pixel_in = in[i];
  const uchar gray = convert_uchar(
    dot(convert_float3(pixel_in.xyz),
      weights));

  out[i] = (uchar4)(gray, gray,
    gray, pixel_in.w);
}
```

FIGURE 5. **Example OpenCL C parallel implementation of gray scale.**

is a parallel implementation, and that its contents match the code inside the sequential loops presented in Figs. 3 and 4. Fancier Java is designed to allow a wide range of parallel programming models to be built on top, each with their own approach to letting application developers define, implicitly or explicitly, the kernels that can be executed in parallel and what their iteration range would be. This last step of the process would enable the automatic generation of parallel Fancier OpenCL code from sequential Fancier Java.

C. IMPLEMENTATION

Vector structures and operations on all layers of Fancier (Java, Native, and OpenCL) are implemented using language-independent templates. The use of templates allows writing a single implementation for each layer to provide support for all vector types and sizes. OpenCL C already supports these data types, and its standard libraries can operate on them, so only a very reduced set of operations are provided and implemented in this way by Fancier. Templates are written using the Mako template library for Python [19], which allows embedding Python definitions and control statements with other text, so that the resulting text can be dynamically produced. This is a common approach used on web applications, which in this case adds a step to the build process of Fancier. Java has support for templates, but it is limited, as it cannot have primitive data types as template parameters without encapsulation in `Object` subclasses, difficulting low-level access to the data, and it is not possible to parametrically generate methods. In C, macros can be used for this purpose, but they become increasingly hard to develop, debug, and read as they become more complex. Therefore, we decided to instead create a set of language-independent templates that we can use to produce the final Java and C implementations of the vector structures and functions of the framework.

The Fancier Java API is divided between its Android variant and its JRE variant. Applications would use one of the two variants depending on the system to be deployed on. For the most part, both variants contain the same core API, and their main difference is the support for creating `RGBAImage` objects from an Android `Bitmap`, and the ability to update a `Bitmap` with the contents of an `RGBAImage`. Further specializations could be added in the future. The Fancier Native API is implemented as a C library, which provides maximum support and interoperation with other libraries and languages. Apart from its core features, such as the JNI and OpenCL integration functions, vectors, images, and containers, it supports the addition of plugins built on top of this core. By creating Fancier Native plugins, specific native functionality can be added to multiple applications, while making use of the Fancier Native system to reduce the development workload. A Java application can easily load and unload native plugins as required. In Fig. 6, we show the build process for the Fancier framework. Any modifications to vector structures or functions, math, or containers have to be made to the corresponding templates. Then, Mako is used to produce the final Java, C, and OpenCL C source codes. Other source files not generated through templates can be directly modified. Fancier OpenCL C source code has to be bundled directly with the final binaries and only compiled at runtime together with the particular kernels for the application, a task that is automatically managed by the `fcOpenCL_compileKernelFile()` and `fcOpenCL_compileKernelAsset()` Fancier Native functions. The Fancier Native core and plugins are built using Make [20] for standard Linux builds, and ndk-build [21] for Android builds, producing the `libfancier.so` core shared library and one shared library for each plugin. Fancier Java code is built using Gradle [22], producing the `fancier-jre.jar` and `fancier-android.jar` libraries both containing the core Fancier Java classes.

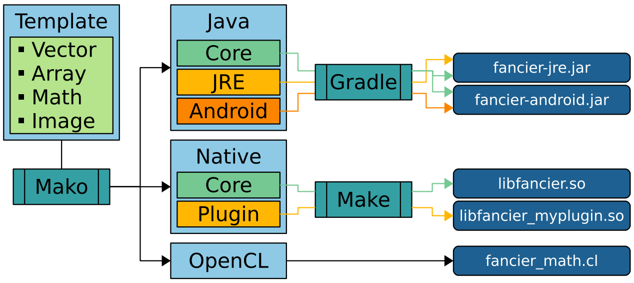


FIGURE 6. **Build process of Fancier libraries.**

Most Fancier Java functions are implemented natively and integrated via JNI. This makes it so that there are no duplicate implementations for any given function, and the Java API is kept as simple as possible while providing the same features as the Fancier Native API. As an example of the execution flow of a call to a Fancier Java method, we show in Fig. 7 the call diagram for the creation of an `RGBAImage` from an array of integers representing pixel data. When the application invokes the Java constructor, it calls the native `initNative()` method that allocates a C structure linked to that Java object by storing a pointer to it on a `long` private attribute of the class. This is performed by the `fcRGBAImage_allocJava()` native function, where

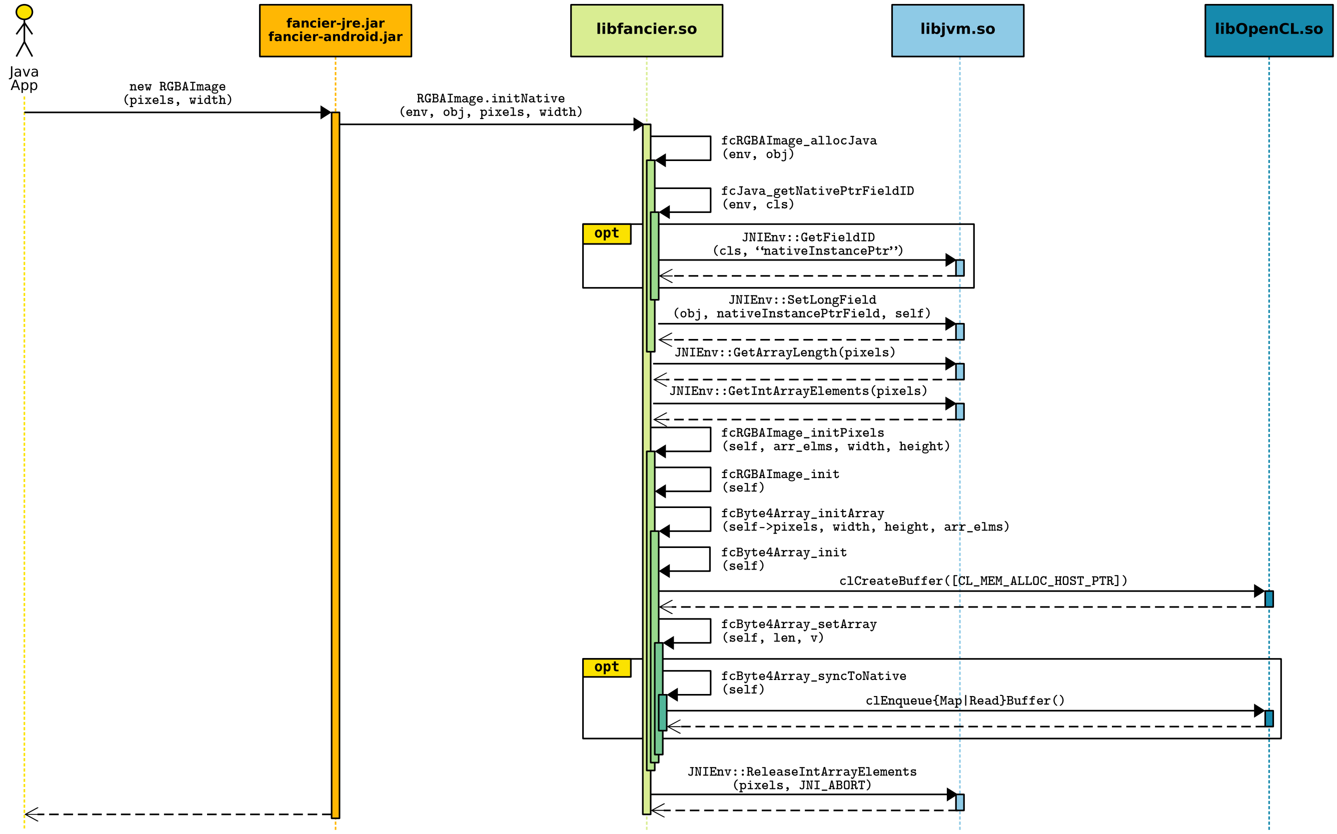

Java App
fancier-jre.jar
fancier-android.jar
libfancier.so
libjvm.so
libOpenCL.so
new RGBAImage (pixels, width)
RGBAImage.initNative (env, obj, pixels, width)
fcRGBAImage_allocJava (env, obj)
fcJava_getNativePtrFieldID (env, cls)
opt
JNIEnv::GetFieldID (cls, "nativeInstancePtr")
JNIEnv::SetLongField (obj, nativeInstancePtrField, self)
JNIEnv::GetArrayLength(pixels)
JNIEnv::GetIntArrayElements(pixels)
fcRGBAImage_initPixels (self, arr_elms, width, height)
fcRGBAImage_init (self)
fcByte4Array_initArray (self->pixels, width, height, arr_elms)
fcByte4Array_init (self)
clCreateBuffer([CL_MEM_ALLOC_HOST_PTR])
fcByte4Array_setArray (self, len, v)
opt
fcByte4Array_syncToNative (self)
clEnqueue{Map|Read}Buffer()
JNIEnv::ReleaseIntArrayElements (pixels, JNI_ABORT)

FIGURE 7. Fancier Java image creation call diagram.

first the fcJava_getNativePtrFieldID() function that gets the field ID of the mentioned attribute of the RGBAImage Java class is called, unless the ID was already cached from a previous access to it. Then the fcRGBAImage native structure is allocated in the heap, and the SetLongField() JNI function is used to store the pointer of the newly allocated structure in that class field as a long integer. After that, the native structure is initialized in fcRGBAImage_initPixels() after extracting the native data from the Java parameters, which is done through the GetArrayLength() and GetIntArrayElements() JNI functions. The fcRGBAImage_initPixels() function involves initializing a reference counter to the native structure in fcRGBAImage_init(), and passing the array data coming from Java to the fcByte4Array_initArray() function, which is the one that initializes a new fcByte4Array object that will hold the actual data of the image and store it in an OpenCL memory buffer. For that, the fcByte4Array_initArray() function calls the clCreateBuffer() function to create the buffer to where pixel data is then copied through the fcByte4Array_setArray() function. The copy is made while the memory buffer is mapped to host memory, and ensuring proper memory alignment. At the end, after releasing the native reference to the Java parameters through the ReleaseIntArrayElements() JNI function, control is returned back to the Java application. The Fancier Java math class is similarly implemented, by forwarding Java calls to C, but without the handling of complex structures and OpenCL interactions.

D. MEMORY OPTIMIZATIONS

On compute-intensive workloads, such as those that would benefit the most from offloading Java to native or even accelerated execution using Fancier, it is of paramount importance to consider memory management. Data movements and copies can have great performance implications and should be avoided whenever possible. However, on an OpenCL-accelerated Java application there exist three independent memory spaces. These spaces are the Java-managed heap, the OS-managed native heap, and the OpenCL device memory. There are various ways in which memory buffers can be moved and accessed across memory space boundaries, but our goal is finding a way to guarantee that only unavoidable memory movements are performed. Each of the different methods discussed in this section is shown and numbered in Fig. 8, and these numbers are used in the text where the corresponding method is described.

At the transition between Java and C, Java-managed primitive arrays may be accessed by native code using the Get<Type>ArrayElements() family of JNI functions with a chance that direct access to the underlying memory

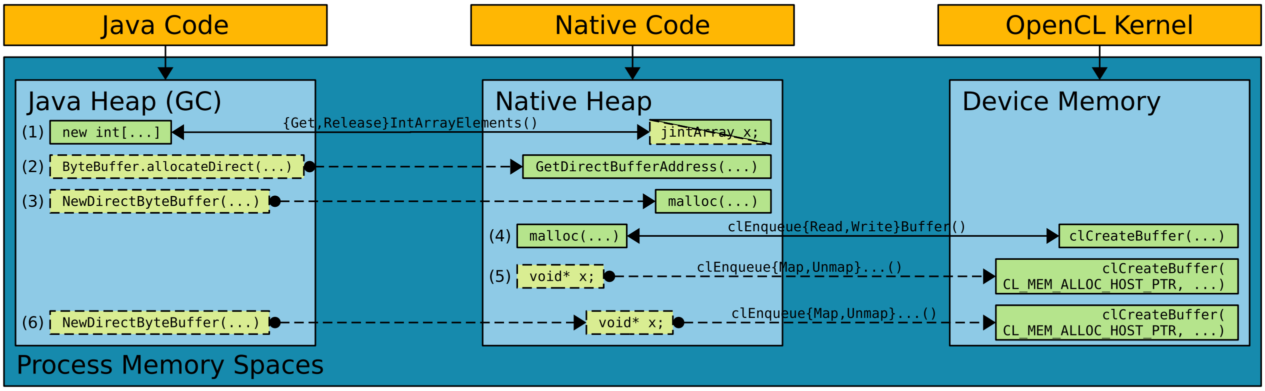

Java Code
Native Code
OpenCL Kernel
Java Heap (GC)
Native Heap
Device Memory
(1) new int[...]
{Get,Release}IntArrayElements()
jintArray x;
(2) ByteBuffer.allocateDirect(...)
GetDirectBufferAddress(...)
(3) NewDirectByteBuffer(...)
malloc(...)
(4) malloc(...)
clEnqueue{Read,Write}Buffer()
clCreateBuffer(...)
(5) void* x;
clEnqueue{Map,Unmap}...()
clCreateBuffer(CL_MEM_ALLOC_HOST_PTR, ...)
(6) NewDirectByteBuffer(...)
void* x;
clEnqueue{Map,Unmap}...()
clCreateBuffer(CL_MEM_ALLOC_HOST_PTR, ...)
Process Memory Spaces

FIGURE 8. Memory spaces on an accelerated Java application.

is provided (1). However, this is not a guarantee and may result in memory copies. Alternatively, Java provides the ByteBuffer class designed for native I/O operations that, created via its allocateDirect() method, allows native and Java code to read and write its contents directly in exchange for a higher allocation cost (2). Additionally, through JNI's NewDirectByteBuffer, they can be created and point to already-allocated native memory (3). Their main issue is that their use is more cumbersome than plain Java arrays and it can be more inefficient if accessed very frequently. Additionally, the ByteBuffer class loses type information about what it is carrying, which can be solved by creating a view buffer of a given type pointing to the same native data (e.g., FloatBuffer, IntBuffer, ...).

Many accelerator architectures feature physically separated memory for the host processor and accelerators. OpenCL exposes this separation by not allowing native memory to be directly passed and used by accelerated kernels, but instead forcing the use of memory buffers that must be allocated on the corresponding accelerator through calls to the clCreateBuffer() function (4). However, modern mobile SoC feature architectures similar to that shown in Fig. 9, where CPU and GPU share the same main memory while having separate caches. This means that sharing memory buffers between CPU and GPU should be possible without copies if caches are handled correctly. OpenCL allows the creation of these zero-copy shared memory buffers by adding the CL_MEM_ALLOC_HOST_PTR flag to a clCreateBuffer() call (5). It is important to note that OpenCL on these devices only supports the allocation of shared memory buffers through that method, and already-allocated native memory via, e.g., malloc() cannot be used in this way. Shared OpenCL memory buffers are only available for write access either on the host or one accelerator at a time, which is handled by using the clEnqueueMapBuffer() and clEnqueueUnmapMemObject() functions. This avoids issues related to race conditions and keeps cache coherence between CPU and accelerators.

Fancier takes these factors into account and implements a method to optimally manage memory allocations on all image and array containers provided (6), which we describe next. As mentioned in Sect. III-B, associated to each of the native structures for these container objects there is an OpenCL memory buffer and a pointer to a native memory allocation, together with a flag that tells from where memory can currently be accessed. These buffers are initially created using the clCreateBuffer() function, using the previously described CL_MEM_ALLOC_HOST_PTR flag to make them available to the native memory space without copies. Initialization from external Java arrays is done through simple calls to memcpy(), whereas, depending on where memory is currently accessible from, other Fancier containers may be copied by calling clEnqueueCopyBuffer(), clEnqueueWriteBuffer(), clEnqueueReadBuffer() or memcpy(). These copies are only necessary on copy constructors, and it is possible to share a single native buffer between multiple Fancier Java containers. They can be at any time accessible either from an accelerator or the host processor, and the public Java and C functions syncToOCL() and syncToNative() are provided in order to manually manage this mapping. Fancier keeps track of where each buffer is mapped and automatically calls these functions, when necessary, as well. These sync functions, in turn, make blocking calls to clEnqueueMapBuffer() and clEnqueueUnmapMemObject() as needed. In addition, containers in the Fancier Java API have a getBuffer() method that returns a ByteBuffer pointing to the same native buffer that is being shared by the host CPU and OpenCL accelerator, usable any time said buffer is updated in the host. Fancier also provides utilities to simplify the access to these structures from Java, via indexed versions of the getBuffer() and setBuffer() methods, which handle memory alignment and byte-order issues. Furthermore, Fancier provides methods to create primitive arrays with a copy of the contents of the buffers, for these cases when copying memory has less overhead than accessing it through a ByteBuffer, or when a copy of the data only available

TABLE 1. Hardware platforms.

	Sony Xperia Z	Huawei P8 Lite
OS	Android 5.1.1	Android 8.0
SoC	Qualcomm Snapdragon APQ8064	HiSilicon Kirin 655
CPU	Qualcomm Krait (4-core @ 1.5 GHz)	Arm Cortex-A53 (4-core @ 2.1 GHz) + Arm Cortex-A53 (4-core @ 1.7 GHz)
GPU	Qualcomm Adreno 320 (4 CU @ 400 MHz)	Arm Mali-T830 MP2 (2 CU @ 900 MHz)
RAM	2GB LPDDR2 @ 533 MHz	3GB LPDDR3 @ 933 MHz
Year	2013	2017

from Java is needed. This presented API provides a very close to transparent way of taking advantage of the unified memory architectures of modern mobile SoC by avoiding any unnecessary memory movements, while enabling read and write access from all three layers: Java, Native, and OpenCL. The current implementation assumes a unified memory architecture, so it is not tailored for systems where accelerator and main system memory are independent. Nevertheless, the presented design and approach can still support those other types of memory systems, and modifications for these cases are simple. Multi-accelerator systems, on the other hand, may require API changes to allow specifying which accelerator should have access to updated memory at every point.

IV. EVALUATION

A. HARDWARE PLATFORMS

We used two Android mobile devices in order to evaluate the overhead or performance improvements of the different Fancier APIs over reference implementations. Their results are representative of a wide range of older and newer smartphones. On one hand we use the Sony Xperia Z (**SXZ**), representing older hardware, and on the other hand we use the Huawei P8 Lite (**P8L**), a more modern smartphone. Table 1 summarizes the relevant features of each of these devices.

The main architectural difference between these devices is the presence of an Arm big.LITTLE CPU cluster in the P8L, which is a feature prevalent in modern devices. Figure 9 contains a simplified block diagram of the SoC of this device, showing its memory organization and main compute elements. Apart from the absence of two CPU clusters, the basic architecture of the SXZ is very similar, as mobile SoC are unified memory architectures where all processors and accelerators share access to a single main memory.

B. KERNEL IMPLEMENTATIONS

We developed multiple equivalent implementations of several image filters in order to evaluate the performance of the different Fancier Java and Native APIs, compared to reference Java and C/C++ implementations using Android `Bitmap` objects [23], which are the standard method of representing and processing images in this OS. We chose image processing kernels for evaluation because these are the predominant type of high-performance operations that many Android applications contain. Furthermore, we chose embarrassingly parallel kernels that would easily map into the SIMT programming model used by OpenCL, in order to show the potential advantages of accelerated execution in these devices, even without developing any hardware-specific manual optimizations. The image processing kernels we implemented, using a 32-bit RGBA pixel format, are:

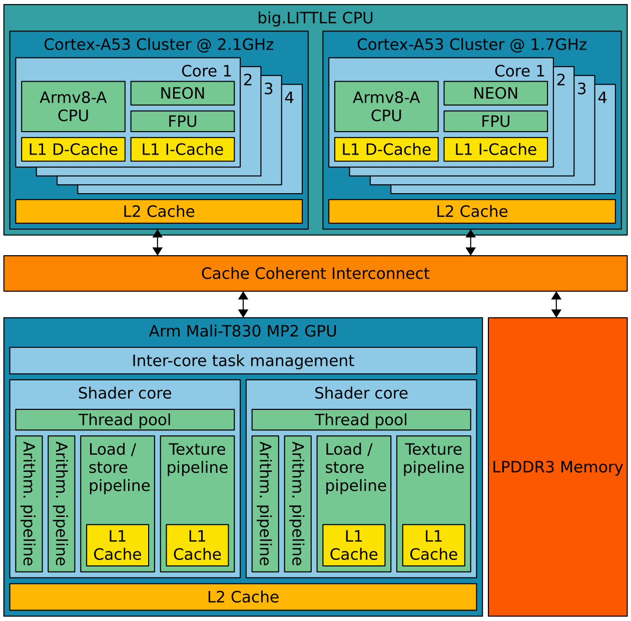


FIGURE 9. HiSilicon Kirin 655 System-on-Chip diagram.

- Bilateral (**BL**): An edge-preserving smoothing filter. It is a stencil code where each neighbor is weighted according to its color and distance to the center pixel.
- Gaussian Blur (**GB**): A smoothing filter based on the Gaussian function. It is implemented as two successive passes where one applies the filter considering the horizontal neighbors and the other considers only the vertical neighbors of each pixel. This reduces computation and provides the same result as a single-kernel variant due to it being separable [24]. Execution time includes creation and release of an intermediate buffer.
- Contrast (**CO**): A pixel-wise kernel that applies a parameterized contrast enhancement of an image.
- Convolve 3 × 3 (**C3**): A convolution kernel using a 3 × 3 mask, implemented without loops.
- Convolve 5 × 5 (**C5**): A convolution kernel using a 5 × 5 mask, implemented without loops.
- Fisheye (**FE**): A distortion kernel which applies a fish-eye lens effect to an image. The coordinates of each pixel are transformed using several math functions, and the resulting output pixel is calculated through a bilinear interpolation of these transformed coordinates.
- Gray Scale (**GS**): A pixel-wise kernel that converts a color image to gray scale.
- Levels (**LV**): A pixel-wise kernel that applies a saturation and contrast levels change to an image.

- Median (**ME**): A median filter, which is a stencil code that evaluates the neighbors of each pixel within a given radius and applies the median intensity of these input pixels to the output. Our implementation only uses the red pixel channel, producing a gray scale output.
- Posterize (**PO**): A filter that applies pre-selected colors to given ranges of input pixel intensity. It is a pixel-wise kernel that is called multiple times, one per range. We use five ranges in our testing.

Each of these kernels has been implemented in several variants, described below:

- **Bitmap Java**: A reference Java implementation that uses Android classes `Bitmap` and `Color` to extract, process, and write pixel data.
- **Fancier Java**: A Java implementation using the Fancier Java API. It is expected to be low performance due to its creation of high amounts of small objects. Its advantages are its relative simplicity to translate into Fancier Native or OpenCL C code and its low programming effort.
- **Fancier Java Perf.**: A higher-performance Fancier Java implementation, which reuses small objects and avoids creating them within a loop. It is harder to develop and read, but it could be automatically produced from regular *Fancier Java* code.
- **Bitmap Native**: A reference C/C++ implementation using `Bitmap` objects and the NDK's `jnigraphics` native library to access their data from native code. It uses JNI for the interaction between Java and C, without the use of the utilities provided by Fancier. This represents the performance potential of executing compute-intensive kernels natively in Android, as well as its associated development effort.
- **Fancier Native**: A C/C++ implementation called from Java using Fancier images, JNI utilities for Java and C interaction and exception handling, and Fancier Native vector types and operations. This can be compared to *Bitmap Native* in order to measure the overhead of the *Fancier Native* libraries and functions, and its improvements in programmability.
- **Fancier OpenCL**: A parallel OpenCL implementation using Fancier JNI utilities for Java and C interaction, and Fancier images for transparent OpenCL host and device buffer management. OpenCL C kernel implementations are equivalent to the corresponding *Fancier Java* implementations, in that they use the same data types, operations, and control structures, in a manner such that one can be directly deduced from the other. However, OpenCL host code needs to bridge the gap between Java and OpenCL execution as well, adding some development overhead diminished by the *Fancier Native* utilities. This variant represents the potential that GPU-accelerated execution on these devices can have, and how using Fancier lowers its complexity.

Our study does not include a *Bitmap OpenCL* implementation, as it would not add any new relevant information over the *Bitmap Native*, *Fancier Native*, and *Fancier OpenCL* variants. The OpenCL C kernel code for these examples would be the same as the one for the *Fancier OpenCL* implementation. On the other hand, the integration between Java and C/C++ execution required in order to run OpenCL implementations is evaluated through the comparison between *Bitmap Native* and *Fancier Native*. Lastly, OpenCL host code is not significantly changed on the *Fancier OpenCL* variant over any regular OpenCL application, only being able to reduce memory management to a single pair of function calls.

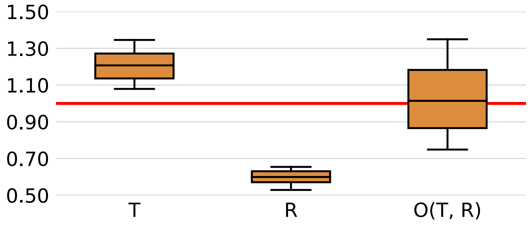


FIGURE 10. Overhead calculation example.

C. EVALUATION METHODOLOGY

Performance experimentation on mobile devices is a sensitive task that needs to be carefully designed and managed to obtain representative, precise, and reproducible results. This is mainly because of the highly dynamic performance characteristics of the SoC they are based on, due to the power and thermal constraints they have. We implemented the reliable benchmarking methodology presented in [25], within the framework introduced in that publication, which we used to make all performance evaluations. The benchmarking application was compiled in release mode, producing native binaries in Arm mode (32-bit), and targeting SDK version 29.

Each implementation of every image-processing kernel described in Sect. IV-B was evaluated by applying it over a set of inputs of different sizes repeatedly, measuring execution time on each instance. The used inputs, together with their respective resolutions in pixels are the following: **VGA** (640 × 480); **XGA** (1024 × 768); **HD1** (1280 × 720); **HD2** (1366 × 768); **HD+** (1600 × 900); **FHD** (1920 × 1080); **QHD** (2560 × 1440); **UHD** (3840 × 2160). The benchmarking application was launched 10 times for each combination of image-processing kernel and implementation, with 30-second intervals separating each launch. Within each one of these launches, the corresponding kernel implementation was run over each input. Each of these runs consisted of at least 4 repetitions, up to 10 repetitions or 5 minutes, whichever happened first. Executions had to be kept relatively short, hurting the precision of results, because the application could get killed or the device could shut down or reboot on its own. That is the reason why we offset this issue by executing each implementation on a separate application launch, running for a bounded amount of time, and repeating this several independent times. The performance of each kernel

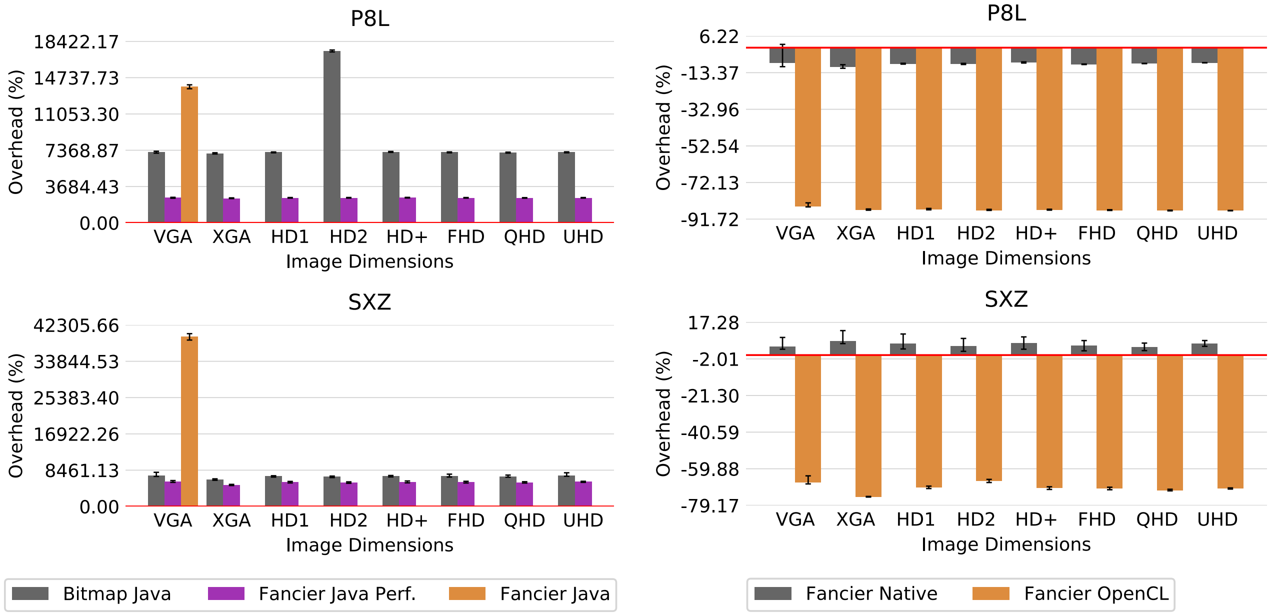


FIGURE 11. Overhead of multiple implementations of Convolve 5 × 5 relative to the reference C/C++ implementation. Red line is 0% overhead, lower is better.

implementation and input size combination on each device was measured between 40 and 100 times in total. Benchmarks were set up to run in a thread separate from the main one responsible for the user interface (UI thread), to help alleviate this problem and prevent "Application not responding" errors.

This is an Android-specific problem, which may be related to the OS detecting these high-load prolonged time intervals as an indication that the application is malfunctioning, and to a stringent GPU watchdog timer affecting OpenCL implementations. In some cases, we were unable to reliably run enough repetitions to obtain meaningful data, so for these particular combinations we did not collect any data. *Fancier Java* implementations of all kernels were only evaluated for the smallest input sizes due to its much higher execution time, demonstrating its unfitness for direct execution without conversion into one of the other Fancier alternatives.

D. PERFORMANCE RESULTS

We use the overhead metric to represent our performance results, which we define as the relative difference between the measured execution time and a reference. We use *Bitmap Native* as the reference to compare each other implementation to. That one being the fastest reference implementation, we expected overheads to be high on Java implementations, small on *Fancier Native* versions and negative on *Fancier OpenCL*, where a GPU was used to execute in parallel.

As described in Sect. IV-C, for each combination of image-processing kernel, implementation, input, and device, we ran a series of benchmarks providing several execution time measurements. This provides us not with a single representative value, but a distribution of values. The larger the number of measurements, the more confidence we can have on the results. In order to calculate an overhead metric between two of these distributions, we need to consider all values from both distributions, obtaining an overhead distribution as a result. If we consider each measurement of a distribution a possible value that said distribution could have, the overhead distribution between two execution time distributions must contain all possible overhead measurements obtained by operating on each pair of measurements from the cartesian product of the values from the two distributions.[2] Assuming an execution time distribution called T and a reference execution time distribution called R, the overhead distribution $O(T, R)$ would be calculated as shown in (1).

$$O(T, R) = \left\{ \frac{t - r}{r} \mid t \in T, r \in R \right\} \tag{1}$$

Overhead figures in this paper have been produced by plotting the median value of the corresponding overhead distributions, together with the 10th and 90th percentiles in the form of error bars. This way we are able to compare distributions of performance measurements while considering their associated imprecision. The horizontal red line in these graphs represents the 0% overhead baseline. Higher positive bars represent higher overheads, or less performance, whereas negative bars below the baseline indicate speed-ups. Figure 10 displays in a box plot format an example calculation of the overhead between two synthetic normal distributions T and R, designed with mean values of 1.2 and 0.6, respectively,

[2] Distributions in this case are unordered lists, where equal values may appear multiple times, as opposed to sets.

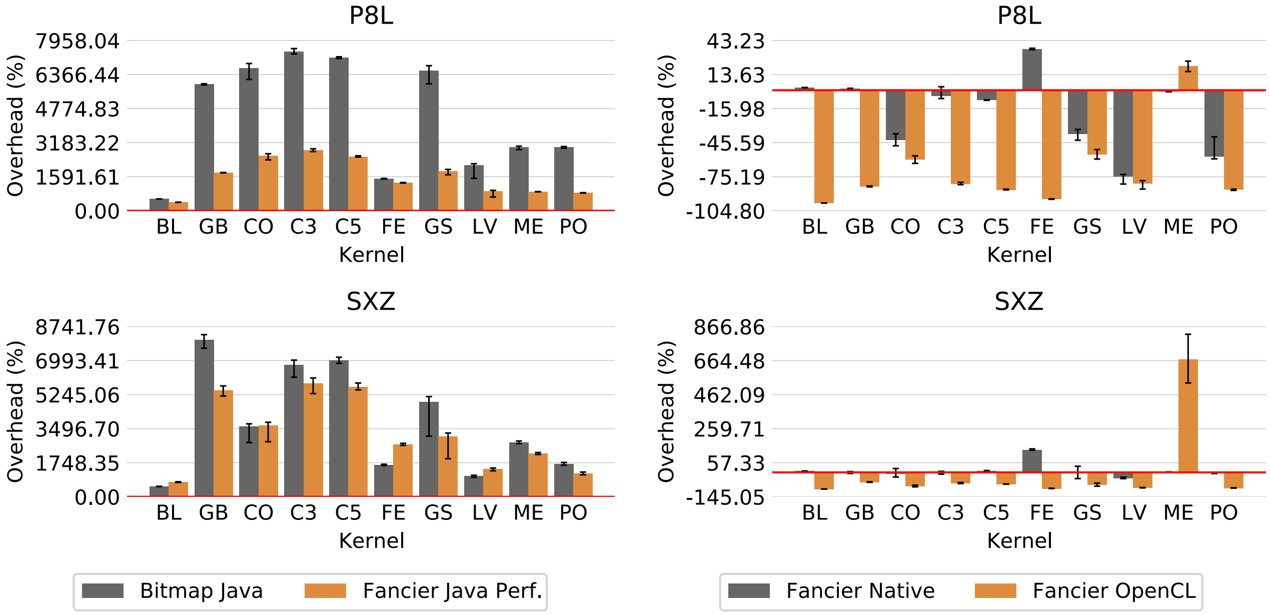


FIGURE 12. Overhead of multiple kernels and implementations relative to the reference C/C++ implementation. Red line is 0% overhead, lower is better.

and standard deviations of 0.1 and 0.05, respectively, in order to obtain an overhead distribution comparable within the same scale. We can see that, as expected, $O(T, R)$ is centered around 1, or 100% (1.2 is double 0.6, so a 100% overhead), and the imprecision of T and R both contribute towards the imprecision of the overhead distribution. In that figure, the information that we can extract is that T has an 80% probability of being between 1.10 and 1.35, R has the same probability of being within 0.5 and 0.65 and, consequently, $O(T, R)$ has an 80% probability of being in the range from 75% to 135%. This approach, which we have intuitively defined, does not require T or R to be normal and it can be applied in order to produce combined metrics between distributions defined by a list of samples.

In Fig. 11 we show the overheads calculated for all implementations of the *C5* kernel and all inputs. As we found performance behavior across inputs was similar across all kernels, we extrapolate our analysis of this figure to the other cases. For more detail, Appendix A contains a summary of all experiments. What we find is that the overhead stays relatively consistent across inputs, except from *Bitmap Java* implementations on the P8L, which perform much worse on the *HD2* input. The fact that this only happens on that device and input, but on all kernels, seems to indicate a possible Bitmap implementation anomaly that has a poor interaction with images of that particular size. We can also see that, even if an implementation runs faster on one device, the best alternative on another device might be different. As an example of this, we can see *Fancier Native* is faster than *Bitmap Native* on the P8L but slower on the SXZ. Another trend we can observe on these graphs is that the relative performance order among implementations does not depend on the input. For instance, if a given kernel implementation on a device is slightly faster than the reference on the *HD1* input, it will be nearly the same on the *UHD* input.

TABLE 2. Programmability evaluation based on source lines of code. Reference Bitmap implementations and relative difference to Fancier implementations shown.

Kernel	Bitmap Java	Fancier Java	Fancier Java Perf.	Bitmap Native	Fancier Native	Fancier OpenCL
BL	57	-36.84%	-14.04%	121	-42.15%	+33.88%
GB	110	-28.18%	-14.55%	154	-20.78%	+58.44%
CO	32	-50.00%	+6.25%	97	-49.48%	+42.27%
C3	70	-45.71%	±0.00%	148	-43.24%	+19.59%
C5	147	-47.62%	-13.61%	235	-40.85%	-6.38%
FE	112	-45.54%	-3.57%	177	-48.59%	+3.95%
GS	14	-14.29%	+57.14%	79	-40.51%	+64.56%
LV	76	-56.58%	-25.00%	141	-53.90%	+25.53%
ME	66	-18.18%	+4.55%	131	-32.82%	+31.30%
PO	23	-13.04%	+34.78%	103	-30.10%	+52.43%

Figure 12 summarizes the calculated overheads on both devices, and for all kernels and implementations, excluding *Fancier Java*, using the *HD1* input. We can see that *Fancier Java Perf.* implementations on the P8L are all, to varying amounts, an improvement over the *Bitmap Java* counterparts. On the SXZ, the highest-performing Java implementation depends on the kernel. On 30% of cases, *Bitmap Java* is faster, while on 60% of them *Fancier Java Perf.* performs better. We believe the different Java execution methods on Android versions 8.0 and 5.1.1 could be the main reason why the difference between implementations is not the same in both devices. It is important to note the large overhead that, to varying degrees, all Java implementations have on all devices and kernels. Despite all the effort put towards optimizing Java

TABLE 3. Summarized results of all performance evaluations on the P8L device. The median execution times in seconds (s) are shown, and the magnitude of the inter-quartile range relative to the median is given to represent the relative spread of measurements.

Implem.	Input	Bilateral	Gaussian Blur	Contrast	Convolve 3x3	Convolve 5x5	Fisheye	Gray Scale	Levels	Median	Posterize
Bitmap Java	VGA	117.027 s 0.011%	5.125 s 0.424%	0.449 s 0.765%	2.126 s 0.368%	5.624 s 0.391%	1.235 s 0.834%	0.387 s 0.508%	0.414 s 2.397%	51.111 s 4.009%	1.277 s 0.472%
	XGA	299.955 s 0.015%	13.133 s 0.426%	1.146 s 0.685%	5.432 s 0.428%	14.402 s 0.292%	3.153 s 0.873%	0.988 s 0.385%	1.057 s 2.543%	129.999 s 4.015%	3.260 s 0.442%
	HD1	353.557 s 0.028%	15.368 s 0.422%	1.342 s 0.685%	6.365 s 0.393%	16.865 s 0.303%	3.697 s 0.865%	1.159 s 0.721%	1.238 s 2.510%	153.221 s 3.993%	3.821 s 0.496%
	HD2	878.377 s 0.745%	29.391 s 0.176%	2.716 s 0.958%	17.069 s 0.270%	46.211 s 0.597%	8.623 s 0.439%	2.471 s 0.306%	2.575 s 1.003%	418.527 s 1.441%	9.821 s 1.700%
	HD+		24.005 s 0.417%	2.094 s 0.683%	9.942 s 0.410%	26.345 s 0.666%	5.778 s 1.376%	1.804 s 0.421%	1.931 s 2.477%	238.061 s 4.039%	5.966 s 0.445%
	FHD		34.569 s 0.425%	3.012 s 0.684%	14.311 s 0.511%	37.941 s 0.407%	8.310 s 0.858%	2.601 s 0.894%	2.778 s 2.482%	342.958 s 4.043%	8.582 s 0.440%
	QHD		61.385 s 0.420%	5.345 s 0.689%	25.436 s 0.427%	67.476 s 0.319%	14.767 s 0.850%	4.607 s 0.388%	4.934 s 2.384%		15.238 s 0.428%
	UHD		137.772 s 0.414%	12.023 s 0.660%	57.234 s 0.418%	151.770 s 0.315%	33.217 s 0.868%	10.357 s 0.406%	11.097 s 2.519%		34.358 s 0.437%
Fancier Java	VGA	263.731 s 1.389%	9.333 s 0.714%	0.753 s 1.487%	4.053 s 1.307%	10.760 s 0.819%	2.773 s 0.852%	0.667 s 3.017%	0.850 s 2.188%	93.383 s 1.938%	2.241 s 2.510%
Fancier Java Perf.	VGA	89.968 s 0.492%	1.603 s 0.327%	0.178 s 1.889%	0.832 s 1.798%	2.028 s 1.980%	1.085 s 1.123%	0.113 s 2.693%	0.190 s 2.742%	16.608 s 0.174%	0.389 s 0.154%
	XGA	231.937 s 0.484%	4.137 s 0.231%	0.454 s 1.700%	2.123 s 1.795%	5.184 s 1.998%	2.775 s 1.158%	0.288 s 2.763%	0.483 s 2.135%	40.852 s 0.310%	0.994 s 0.134%
	HD1	271.212 s 0.483%	4.793 s 0.226%	0.532 s 1.389%	2.471 s 1.829%	6.075 s 1.994%	3.252 s 1.131%	0.340 s 2.706%	0.568 s 2.396%	49.625 s 0.156%	1.163 s 0.094%
	HD2	308.416 s 0.480%	5.461 s 0.323%	0.605 s 1.217%	2.834 s 1.733%	6.910 s 2.006%	3.700 s 1.180%	0.386 s 2.690%	0.651 s 2.129%	54.436 s 0.158%	1.325 s 0.167%
	HD+		7.503 s 0.284%	0.832 s 1.988%	3.891 s 1.810%	9.508 s 1.980%	5.077 s 1.183%	0.523 s 2.690%	0.891 s 1.939%	74.092 s 0.348%	1.816 s 0.115%
	FHD		10.770 s 0.342%	1.195 s 1.411%	5.594 s 1.757%	13.711 s 1.988%	7.310 s 1.194%	0.768 s 2.688%	1.283 s 1.873%	107.344 s 0.101%	2.611 s 0.152%
	QHD		19.280 s 0.198%	2.121 s 2.018%	9.875 s 1.836%	24.337 s 1.997%	12.989 s 1.130%	1.338 s 2.703%	2.281 s 1.820%		4.629 s 0.104%
	UHD		43.191 s 0.312%	4.774 s 2.032%	22.401 s 1.811%	54.712 s 1.995%	29.216 s 1.145%	3.006 s 2.697%	5.147 s 1.914%		10.449 s 0.082%
Bitmap Native	VGA	17.335 s 0.011%	0.077 s 1.457%	0.006 s 2.569%	0.028 s 2.170%	0.077 s 1.047%	0.077 s 1.099%	0.006 s 2.377%	0.018 s 2.745%	1.668 s 0.042%	0.042 s 9.706%
	XGA	46.113 s 0.018%	0.252 s 0.252%	0.017 s 3.599%	0.072 s 13.128%	0.202 s 0.682%	0.197 s 0.199%	0.015 s 6.105%	0.047 s 3.293%	4.131 s 0.135%	0.106 s 1.225%
	HD1	54.300 s 0.018%	0.255 s 0.252%	0.020 s 5.294%	0.085 s 2.067%	0.232 s 0.190%	0.231 s 0.193%	0.017 s 6.192%	0.056 s 20.959%	5.010 s 0.053%	0.125 s 0.564%
	HD2	60.596 s 0.030%	0.289 s 0.382%	0.021 s 11.416%	0.097 s 1.684%	0.264 s 0.181%	0.263 s 0.178%	0.020 s 51.494%	0.064 s 6.257%	4.781 s 0.211%	0.142 s 0.537%
	HD+		0.395 s 0.302%	0.030 s 5.087%	0.131 s 0.670%	0.361 s 0.232%	0.361 s 0.177%	0.028 s 38.312%	0.086 s 2.284%	6.490 s 0.013%	0.195 s 0.355%
	FHD		0.575 s 0.343%	0.045 s 33.487%	0.189 s 0.384%	0.522 s 0.065%	0.521 s 0.114%	0.039 s 7.561%	0.123 s 0.629%	9.526 s 0.009%	0.278 s 0.073%
	QHD		1.067 s 0.309%	0.082 s 4.224%	0.335 s 0.252%	0.932 s 0.033%	0.925 s 0.055%	0.070 s 11.076%	0.219 s 0.268%		0.488 s 0.173%
	UHD		2.304 s 0.109%	0.178 s 0.378%	0.760 s 0.061%	2.088 s 0.032%	2.096 s 0.072%	0.155 s 0.842%	0.489 s 0.163%		1.115 s 0.041%

Implem.	Input	Bilateral	Gaussian Blur	Contrast	Convolve 3x3	Convolve 5x5	Fisheye	Gray Scale	Levels	Median	Posterize
Fancier Native	VGA	17.770 s 0.003%	0.079 s 15.325%	0.004 s 3.496%	0.027 s 3.562%	0.071 s 2.881%	0.104 s 0.617%	0.004 s 5.563%	0.005 s 3.323%	1.655 s 0.025%	0.017 s 3.263%
	XGA	47.060 s 0.080%	0.251 s 0.316%	0.010 s 6.144%	0.068 s 3.402%	0.181 s 0.707%	0.266 s 0.369%	0.009 s 3.045%	0.012 s 3.573%	3.860 s 0.448%	0.044 s 4.165%
	HD1	55.537 s 0.044%	0.258 s 0.324%	0.011 s 6.899%	0.080 s 2.040%	0.212 s 0.167%	0.313 s 0.400%	0.011 s 4.377%	0.014 s 4.017%	4.948 s 0.026%	0.052 s 29.599%
	HD2	62.927 s 0.450%	0.293 s 0.275%	0.013 s 4.715%	0.091 s 1.709%	0.241 s 0.265%	0.356 s 0.200%	0.012 s 5.129%	0.016 s 5.646%	4.812 s 0.765%	0.059 s 23.516%
	HD+		0.403 s 0.795%	0.017 s 5.835%	0.124 s 0.953%	0.332 s 0.224%	0.489 s 0.305%	0.017 s 4.812%	0.022 s 5.416%	6.164 s 0.011%	0.081 s 7.378%

TABLE 3. *(Continued.)* Summarized results of all performance evaluations on the P8L device. The median execution times in seconds (s) are shown, and the magnitude of the inter-quartile range relative to the median is given to represent the relative spread of measurements.

	FHD		0.584 s 0.319%	0.026 s 40.023%	0.180 s 0.454%	0.476 s 0.146%	0.705 s 0.113%	0.024 s 6.757%	0.032 s 29.004%	9.095 s 0.020%	0.113 s 1.547%
	QHD		1.100 s 0.512%	0.045 s 9.890%	0.320 s 0.240%	0.853 s 0.057%	1.251 s 0.551%	0.044 s 10.227%	0.056 s 3.852%		0.198 s 0.382%
	UHD		2.370 s 0.158%	0.100 s 10.101%	0.721 s 0.061%	1.919 s 0.073%	2.830 s 0.141%	0.097 s 9.337%	0.125 s 1.124%		0.455 s 0.207%
Fancier OpenCL	VGA	0.350 s 0.400%	0.015 s 2.516%	0.004 s 6.310%	0.007 s 3.783%	0.012 s 6.299%	0.005 s 5.053%	0.004 s 7.105%	0.005 s 5.949%	0.970 s 6.449%	0.007 s 6.723%
	XGA	0.903 s 0.354%	0.035 s 1.971%	0.007 s 5.324%	0.013 s 3.484%	0.027 s 1.848%	0.010 s 4.302%	0.007 s 6.163%	0.009 s 3.378%	2.360 s 10.975%	0.015 s 5.107%
	HD1	1.052 s 0.230%	0.041 s 1.947%	0.008 s 5.563%	0.016 s 3.903%	0.031 s 1.682%	0.012 s 2.491%	0.008 s 7.984%	0.011 s 4.537%	6.061 s 3.577%	0.017 s 4.226%
	HD2	1.173 s 0.116%	0.039 s 1.221%	0.012 s 8.335%	0.016 s 3.863%	0.035 s 1.747%	0.014 s 4.158%	0.013 s 8.227%	0.016 s 6.708%	2.743 s 5.484%	0.027 s 10.201%
	HD+		0.064 s 2.021%	0.011 s 3.787%	0.023 s 2.623%	0.048 s 1.091%	0.018 s 1.809%	0.011 s 4.904%	0.015 s 2.318%	7.845 s 12.842%	0.024 s 3.542%
	FHD		0.093 s 1.137%	0.015 s 3.966%	0.032 s 1.667%	0.068 s 1.749%	0.025 s 2.973%	0.014 s 4.774%	0.021 s 2.743%	6.156 s 6.789%	0.034 s 3.069%
	QHD		0.169 s 0.882%	0.025 s 3.297%	0.056 s 2.113%	0.120 s 1.273%	0.043 s 2.261%	0.024 s 3.747%	0.035 s 2.603%		0.055 s 2.408%
	UHD		0.393 s 1.659%	0.060 s 25.693%	0.127 s 9.504%	0.268 s 0.621%	0.100 s 10.444%	0.056 s 25.338%	0.081 s 18.568%		0.123 s 7.733%

execution on Android, for compute-intensive kernels, such as the ones we are evaluating, C/C++ implementations are still clearly the best choice. This performance disparity together with the increased programming effort required to develop native implementations, as shown in Sect. IV-E, is what creates the need for tools like Fancier.

With regards to native implementations, as expected, *Fancier OpenCL* code is significantly faster than the equivalent *Bitmap Native* counterparts. The only instance where this is not the case is on the *ME* kernel, which consists of a series of branches and loops that prevent GPUs from performing optimally. Performance results on different inputs for this kernel have a much higher variability than the rest as well, possibly hinting at an important dependence on the grouping of work items or memory organization. With respect to the comparison between *Fancier Native* and *Bitmap Native* implementations we find that, in 10%[3] of the evaluated kernels on the P8L and 60%[4] on the SXZ, their differences are within margin of error. This means that, in these instances, our benchmarks show that there are not enough differences to reject the hypothesis that both implementations are the same regarding performance. Or, more simply, that we did not measure any overhead or performance improvement there. Additionally, in 30%[5] and 20%[6] of the cases in the P8L and SXZ, respectively, performance differences are reduced enough to make programming effort the deciding factor.

Considering one of the main goals of Fancier is to simplify the integration of native code execution in Java applications, but not to improve performance over what it is typically achieved by hand, it is especially surprising finding that in 50%[7] and 10%[8] of the kernels on the P8L and SXZ, respectively, *Fancier Native* significantly improves performance over the reference *Bitmap Native* implementation. We believe the factors that could be making a difference in these cases are related to compiler optimizations and memory. For the first factor, many of the simpler element-wise vector operations were implemented in headers allowing inlining, and a wider range of pure functions were annotated to allow memorization of results. With respect to the second factor, by allocating memory as shared through the OpenCL API, the OpenCL driver forces a certain memory alignment, which could have resulted in a better cache performance in exchange for a slower allocation time. Conversely, the *Fancier Native* implementation of the *FE* kernel displays a significant reduction of performance compared to the reference on both devices. A preliminary inspection of the generated assembly code seemed to suggest that there were some compiler optimizations that could not be applied to the Fancier variant, showing that there are still some corner cases to be studied where *Fancier Native* adds significant overhead.

E. PROGRAMMABILITY RESULTS

In addition to performance differences, it is also important to evaluate the differences in programmability by using the various Fancier APIs compared to only Java and C/C++ implementations. We used source lines of code as an easily obtainable and comparable proxy for programmability, and we show our results in Table 2.

[3] *C3.*

[4] *GB, CO, C3, GS, ME, PO.*

[5] *BL, GB, ME*

[6] *BL, C5.*

[7] *CO, C5, GS, LV, PO.*

[8] *LV.*

TABLE 4. Summarized results of all performance evaluations of Java implementations on the SXZ device. The median execution times in seconds (s) are shown, and the magnitude of the inter-quartile range relative to the median is given to represent the relative spread of measurements.

Implem.	Input	Bilateral	Gaussian Blur	Contrast	Convolve 3x3	Convolve 5x5	Fisheye	Gray Scale	Levels	Median	Posterize
Bitmap Java	VGA	240.970 s	8.370 s	0.813 s	3.373 s	8.643 s	2.502 s	0.714 s	0.806 s	75.814 s	2.118 s
		1.331%	1.013%	1.825%	1.806%	7.339%	2.110%	2.342%	3.189%	2.442%	2.443%
	XGA	625.118 s	21.565 s	2.054 s	8.579 s	22.165 s	6.361 s	1.825 s	2.110 s	195.525 s	5.382 s
		1.258%	1.352%	1.669%	1.501%	2.114%	2.153%	2.072%	2.201%	1.233%	2.322%
	HD1	735.263 s	25.130 s	2.408 s	10.086 s	25.662 s	7.432 s	2.145 s	2.436 s	229.555 s	6.274 s
		0.671%	1.569%	1.352%	1.674%	1.101%	1.559%	3.055%	2.731%	2.322%	1.934%
	HD2	834.547 s	28.702 s	2.743 s	11.323 s	29.316 s	8.478 s	2.412 s	2.733 s	258.562 s	7.162 s
		1.681%	1.633%	1.774%	0.963%	1.371%	1.925%	1.797%	1.816%	1.038%	1.650%
	HD+		39.051 s	3.750 s	15.541 s	40.286 s	11.668 s	3.288 s	3.827 s		9.815 s
			1.275%	1.317%	1.077%	1.311%	1.380%	1.627%	2.332%		1.667%
	FHD		56.174 s	5.421 s	22.530 s	58.474 s	16.710 s	4.757 s	5.460 s		14.113 s
			0.811%	1.349%	1.732%	5.212%	1.233%	2.356%	2.107%		1.968%
	QHD		101.333 s	9.606 s	40.191 s	103.803 s	29.661 s	8.439 s	9.718 s		25.011 s
			1.739%	2.554%	0.945%	2.083%	1.933%	1.489%	2.065%		1.752%
	UHD		226.350 s	21.560 s	90.105 s	233.963 s	66.573 s	19.020 s	21.765 s		56.105 s
			0.916%	1.505%	1.324%	6.079%	2.132%	1.633%	2.004%		0.938%
Fancier Java	VGA	1,257.9 s	37.044 s	4.353 s	18.589 s	47.437 s	14.701 s	2.983 s	5.380 s	388.865 s	10.142 s
		2.054%	1.875%	1.800%	1.867%	1.918%	1.728%	2.331%	1.835%	1.747%	2.203%
Implem.	Input	Bilateral	Gaussian Blur	Contrast	Convolve 3x3	Convolve 5x5	Fisheye	Gray Scale	Levels	Median	Posterize
Fancier Java Perf.	VGA	323.354 s	5.758 s	0.826 s	2.918 s	7.001 s	4.014 s	0.463 s	1.084 s	60.401 s	1.506 s
		3.145%	2.601%	3.654%	3.895%	3.796%	2.887%	6.566%	3.584%	2.957%	2.688%
	XGA	836.230 s	14.719 s	2.076 s	7.436 s	17.813 s	10.198 s	1.164 s	2.716 s	150.252 s	3.836 s
		2.408%	2.188%	3.482%	2.564%	2.382%	1.887%	2.955%	4.146%	2.592%	2.959%
	HD1	988.570 s	17.156 s	2.444 s	8.700 s	20.790 s	11.950 s	1.375 s	3.192 s	183.520 s	4.506 s
		2.359%	2.515%	2.370%	2.894%	2.206%	2.088%	2.813%	2.475%	2.156%	4.290%
	HD2	1,115.4 s	19.640 s	2.780 s	9.991 s	23.625 s	13.580 s	1.555 s	3.613 s	203.063 s	5.125 s
		3.312%	2.793%	3.777%	3.845%	2.423%	2.241%	3.230%	3.630%	3.178%	2.845%
	HD+		26.730 s	3.796 s	13.631 s	32.558 s	18.647 s	2.149 s	4.964 s		6.982 s
			2.869%	3.319%	3.279%	2.786%	2.222%	3.598%	3.179%		2.726%
	FHD		38.547 s	5.487 s	19.597 s	46.756 s	26.783 s	3.063 s	7.107 s		10.082 s
			2.262%	2.906%	2.451%	2.705%	2.317%	3.539%	2.221%		3.258%
	QHD		68.776 s	9.743 s	34.791 s	83.127 s	47.654 s	5.421 s	12.610 s		17.838 s
			1.855%	2.768%	2.520%	2.920%	2.233%	3.710%	3.376%		2.518%
	UHD		155.215 s	21.807 s	78.600 s	187.468 s	107.605 s	12.155 s	28.446 s		40.125 s
			3.141%	2.988%	3.177%	1.785%	1.759%	4.092%	2.284%		2.251%
Bitmap Native	VGA	34.299 s	0.097 s	0.021 s	0.049 s	0.120 s	0.142 s	0.014 s	0.070 s	2.543 s	0.111 s
		0.296%	6.682%	16.103%	2.692%	1.019%	0.589%	11.362%	1.431%	1.298%	6.870%
	XGA	95.440 s	0.441 s	0.056 s	0.129 s	0.352 s	0.366 s	0.036 s	0.171 s	8.265 s	0.300 s
		0.597%	1.775%	24.048%	3.251%	0.261%	0.441%	10.025%	12.076%	0.843%	0.895%
	HD1	116.672 s	0.307 s	0.065 s	0.146 s	0.362 s	0.429 s	0.043 s	0.209 s	7.925 s	0.348 s
		0.752%	3.544%	17.470%	6.713%	0.716%	0.242%	26.034%	3.003%	1.034%	3.170%
	HD2	129.334 s	0.356 s	0.070 s	0.168 s	0.416 s	0.490 s	0.048 s	0.237 s	9.564 s	0.408 s
		0.255%	3.681%	17.568%	3.177%	1.479%	1.484%	30.259%	3.702%	0.907%	2.932%
	HD+		0.440 s	0.104 s	0.229 s	0.560 s	0.672 s	0.066 s	0.314 s		0.550 s
			3.834%	19.734%	5.353%	1.864%	0.895%	17.561%	4.250%		0.738%
	FHD		0.657 s	0.146 s	0.329 s	0.816 s	0.969 s	0.095 s	0.454 s		0.790 s
			1.454%	11.580%	4.153%	2.146%	0.887%	15.819%	2.650%		1.058%
	QHD		2.043 s	0.264 s	0.594 s	1.466 s	1.720 s	0.168 s	0.807 s		1.359 s
			1.276%	1.004%	0.483%	1.280%	0.985%	8.352%	0.578%		1.820%
	UHD		2.722 s	0.588 s	1.316 s	3.210 s	3.854 s	0.376 s	1.814 s		3.089 s
			1.570%	0.583%	0.589%	1.169%	0.969%	0.707%	0.618%		1.564%
Fancier Native	VGA	37.470 s	0.098 s	0.020 s	0.048 s	0.125 s	0.335 s	0.014 s	0.046 s	2.518 s	0.106 s
		0.763%	1.064%	34.621%	1.712%	1.048%	1.120%	27.092%	32.161%	1.223%	1.636%
	XGA	103.426 s	0.421 s	0.050 s	0.127 s	0.377 s	0.852 s	0.036 s	0.118 s	8.464 s	0.292 s
		0.663%	0.791%	27.253%	5.004%	2.876%	1.804%	38.006%	5.179%	1.047%	1.308%
	HD1	125.574 s	0.304 s	0.061 s	0.144 s	0.382 s	0.999 s	0.041 s	0.138 s	8.072 s	0.342 s
		0.330%	3.266%	22.819%	4.047%	2.833%	1.213%	30.774%	4.305%	0.822%	1.531%
	HD2	139.367 s	0.349 s	0.065 s	0.161 s	0.435 s	1.139 s	0.046 s	0.158 s	9.666 s	0.399 s
		0.247%	1.873%	1.384%	6.838%	2.266%	0.849%	2.993%	3.808%	1.098%	2.188%
	HD+		0.439 s	0.090 s	0.228 s	0.596 s	1.564 s	0.063 s	0.217 s		0.537 s
			2.271%	12.059%	1.686%	2.385%	0.584%	10.809%	4.086%		2.881%
	FHD		0.647 s	0.131 s	0.328 s	0.858 s	2.252 s	0.091 s	0.311 s		0.771 s
			2.333%	9.416%	2.118%	1.544%	0.554%	17.343%	3.590%		2.023%
	QHD		1.952 s	0.232 s	0.593 s	1.531 s	4.012 s	0.165 s	0.551 s		1.334 s
			1.254%	0.668%	0.582%	0.979%	0.469%	9.358%	1.528%		1.239%

TABLE 4. *(Continued.)* Summarized results of all performance evaluations of Java implementations on the SXZ device. The median execution times in seconds (s) are shown, and the magnitude of the inter-quartile range relative to the median is given to represent the relative spread of measurements.

	UHD		2.673 s 1.778%	0.519 s 0.870%	1.311 s 0.417%	3.410 s 1.077%	9.051 s 1.079%	0.360 s 0.297%	1.238 s 0.160%		3.022 s 1.463%
Fancier OpenCL	VGA	0.372 s 0.107%	0.047 s 11.274%	0.008 s 48.894%	0.022 s 18.240%	0.039 s 9.827%	0.009 s 41.776%	0.008 s 49.908%	0.010 s 44.368%	16.149 s 25.993%	0.012 s 32.064%
	XGA	0.995 s 1.617%	0.119 s 0.993%	0.011 s 37.501%	0.047 s 8.544%	0.089 s 0.479%	0.016 s 24.879%	0.011 s 37.364%	0.016 s 26.985%	51.387 s 19.687%	0.021 s 18.500%
	HD1	1.104 s 0.044%	0.132 s 0.507%	0.012 s 32.717%	0.053 s 7.829%	0.110 s 0.417%	0.018 s 20.224%	0.012 s 32.257%	0.017 s 22.451%	61.210 s 13.897%	0.024 s 17.345%
	HD2	1.287 s 0.804%	0.189 s 2.437%	0.016 s 26.794%	0.074 s 3.236%	0.141 s 0.780%	0.020 s 20.564%	0.017 s 24.028%	0.027 s 16.697%	77.385 s 24.183%	0.036 s 13.409%
	HD+		0.196 s 0.693%	0.015 s 26.292%	0.079 s 0.500%	0.168 s 0.362%	0.027 s 14.446%	0.015 s 25.491%	0.024 s 18.145%		0.036 s 10.933%
	FHD		0.290 s 0.680%	0.019 s 21.165%	0.116 s 0.284%	0.243 s 0.283%	0.035 s 10.634%	0.019 s 19.974%	0.031 s 12.017%		0.047 s 0.851%
	QHD		0.513 s 0.431%	0.029 s 2.028%	0.197 s 0.387%	0.423 s 0.191%	0.058 s 0.635%	0.028 s 13.097%	0.050 s 0.678%		0.078 s 0.544%
	UHD		1.137 s 0.405%	0.059 s 2.146%	0.431 s 0.305%	0.956 s 0.279%	0.128 s 0.424%	0.059 s 0.871%	0.106 s 1.136%		0.173 s 0.401%

It can be clearly seen that native implementations take more effort than pure Java ones. This is due to the need to use JNI as a bridge between the main Java application and the C/C++ implementation of an image-processing kernel. *Fancier Java* implementations are between 10% to 60% smaller than the reference *Bitmap Java* implementation, which is a very significant improvement. *Fancier Java Perf.* versions take more lines of code due to the need to explicitly declare and create a temporary object for each operation, making it more verbose than the reference Java counterparts on shorter kernels. However, for longer kernels it tends to improve due to the relative simplicity of the Fancier Java API. Regarding native implementations, the differences between *Fancier Native* and *Bitmap Native* versions of each kernel are more pronounced than their respective Java implementations. This is mainly due to the significant programming overhead of JNI, which the *Fancier Native* set of structures, functions, and macros reduces to the bare minimum.

OpenCL is known to be a verbose programming model, which it requires in order to provide its low-level control over the accelerated execution. It demands the definition of independent source code for the accelerator and the host processor, adding a significant development overhead. Through the use of the Fancier Native API, the interaction between Java and OpenCL accelerators is greatly simplified, but it does not remove the need to properly set up kernel parameters and kernel launches. Compared to the reference C/C++ implementation of the same kernels, *Fancier OpenCL* needs extra work, but it is significantly less than manually integrating OpenCL execution within a Java application. Crucially, *Fancier Java* code is very close to OpenCL C, so a translation layer can be designed to eliminate the added development cost of this alternative.

V. CONCLUSION

We have presented Fancier, a multi-language framework that simplifies the execution of native and accelerated OpenCL code within a Java application. This is achieved by the definition of various Java data structures with a C/C++ counterpart, and several utilities to seamlessly read and modify them from both programming languages. An optimized memory management system is implemented, which avoids memory copies of Fancier containers on unified memory systems while allowing access from Java, C/C++, and OpenCL code.

The Fancier Java API is highly streamlined, and it has been modeled after OpenCL C with the goal of defining a subset of Java that can be easily transformed into high-performance native or accelerated implementations. In our analysis, it requires less code than Android `Bitmap` for image processing kernels. If executed directly, it suffers from poor performance, so alternative Java methods are provided to avoid this overhead. This alternative still matches closely the simpler Java counterpart, making it useful for debugging purposes before porting it to C/C++ or OpenCL. We have found its performance to be, in many cases, greater than a reference Java implementation using `Bitmap` objects, making it a good alternative if a pure Java application is the goal.

The Fancier Native API greatly reduces the effort of integrating C/C++ into a Java application, by simplifying JNI calls, managing containers automatically and efficiently, and providing logging and exception handling features. By featuring a similar kernel API to Fancier Java, it can be easily derived from it. Our experimentation showed this API to be very high performance, even significantly outperforming our reference C/C++ implementations. The automatic management of memory provided by Fancier and the usage of its native API for the bridge between Java and OpenCL makes the acceleration of compute-intensive kernels in Java much simpler. This opens the way for the acceleration of more Java applications by reducing their development overhead.

Even though Fancier has promising features to help application developers integrate high-performance native execution into Java applications, its main strength is that native and OpenCL implementations can be automatically produced from the much simpler Fancier Java code. Parallel programming models built on top of Java can adopt this framework and benefit from its ease to program and optimize.

As a continuation of this work, Fancier is being adapted to and validated on Linux systems and non SoC-based architectures. Future work includes designing and developing a high-performance Java parallel programming model based on this framework, and further investigation should be carried out in order to understand the origin of performance edge cases of Fancier Native implementations. The interaction of native parallel programming models, such as OpenMP or OpenACC, with Fancier Native code could be explored as an interesting avenue for large-scale, desktop and embedded multicore or accelerated computing systems. We believe this would represent a promising backend to investigate for parallel programming models built on top of Fancier.

.

APPENDIX A PERFORMANCE RESULTS SUMMARY

See Tables 3 and 4.

REFERENCES

[1] L. A. Smith, J. M. Bull, and J. Obdrzálek, "A parallel Java Grande benchmark suite," in *Proc. ACM/IEEE Conf. Supercomputing (CDROM) Supercomputing*, 2001, p. 6.

[2] K. Shiv, R. Iyer, C. Newburn, J. Dahlstedt, M. Lagergren, and O. Lindholm, "Impact of JIT/JVM optimizations on Java application performance," in *Proc. 7th Workshop Interact. Between Compil. Comput. Archit. (INTERACT)*, Feb. 2003, pp. 5–13.

[3] OpenJDK. *Project Sumatra*. Accessed: Dec. 13, 2021. [Online]. Available: https://openjdk.java.net/projects/sumatra/

[4] J. Fumero, M. Papadimitriou, F. S. Zakkak, M. Xekalaki, J. Clarkson, and C. Kotselidis, "Dynamic application reconfiguration on heterogeneous hardware," in *Proc. 15th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*, 2019, pp. 165–178, doi: 10.1145/3313808.3313819.

[5] OpenJDK. *JEP 243: Java-Level JVM Compiler Interface*. Accessed: Dec. 13, 2021. [Online]. Available: http://openjdk.java.net/jeps/243

[6] C. Wimmer *et al.*, "Initialize once, start fast: Application initialization at build time," *ACM Program. Lang.*, vol. 3, pp. 1–29, Oct. 2019.

[7] A. Acosta, S. Afonso, and F. Almeida, "Extending paralldroid with object oriented annotations," *Parallel Comput.*, vol. 57, pp. 25–36, Sep. 2016.

[8] Aparapi. *API for Data Parallel Java*. Accessed: Dec. 13, 2021. [Online]. Available: http://aparapi.com/

[9] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch, "Rootbeer: Seamlessly using GPUs from Java," in *Proc. IEEE 14th Int. Conf. High Perform. Comput. Commun. IEEE 9th Int. Conf. Embedded Softw. Syst.*, Jun. 2012, pp. 375–380.

[10] G. Andrade, W. de Carvalho, R. Utsch, P. Caldeira, A. Alburquerque, F. Ferracioli, L. Rocha, M. Frank, D. Guedes, and R. Ferreira, "ParallelME: A parallel mobile engine to explore heterogeneity in mobile computing architectures," in *Euro-Par 2016: Parallel Processing*, P.-F. Dutot and D. Trystram, Eds. Cham, Switzerland: Springer, 2016, pp. 447–459.

[11] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA," in *Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H.-X. Lin, Eds. Berlin, Germany: Springer, 2009, pp. 887–899.

[12] Android Open Source Project. (2020). *ART and Dalvik*. [Online]. Available: https://source.android.com/devices/tech/dalvik/

[13] Android Open Source Project. (2020). *Implementing ART Just-in-Time (JIT) Compiler*. [Online]. Available: https://source.android.com/devices/tech/dalvik/jit-compiler

[14] B. Evans. (2014). *Understanding Java JIT Compilation With JITWatch—Part 1*. [Online]. Available: https://www.oracle.com/technical-resources/articles/java/architect-evan%s-pt1.html

[15] S. Liang, *The Java Native Interface: Programmer's Guide and Specification*. Reading, MA, USA: Addison-Wesley, 1999.

[16] Android Open Source Project. (2020). *Get Started With the NDK*. [Online]. Available: https://developer.android.com/ndk/guides

[17] Android Open Source Project. (2021). *Renderscript Overview*. [Online]. Available: https://developer.android.com/guide/topics/renderscript/compute.html

[18] Google. (2021). *Clspv*. [Online]. Available: https://github.com/google/clspv

[19] M. Bayer. (2021). *Mako Templates for Python*. [Online]. Available: https://www.makotemplates.org/

[20] R. M. Stallman and R. McGrath, "GNU make manual—A program for directing recompilation," Free Softw. Found., Cambridge, MA, USA, Tech. Rep. 0.28 Beta, 1991.

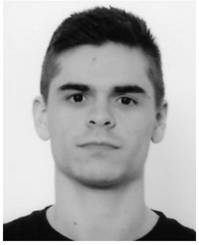
[21] Android Open Source Project. (2020). *The NDK-Build Script*. [Online]. Available: https://developer.android.com/ndk/guides/ndk-build

[22] B. Muschko, *Gradle in Action*, vol. 27. Shelter Island, NY, USA: Manning, 2014.

[23] Android Open Source Project. (2021). *Handling Bitmaps*. [Online]. Available: https://developer.android.com/topic/performance/graphics

[24] T. Petrick. (2016). *Convolution—Part Four: Separable Kernels*. [Online]. Available: https://taylorpetrick.com/blog/post/convolution-part4

[25] S. Afonso and F. Almeida, "Rancid: Reliable benchmarking on Android platforms," *IEEE Access*, vol. 8, pp. 143342–143358, 2020.



SERGIO AFONSO received the B.Sc. and M.Sc. degrees in computer science from the University of La Laguna, San Cristóbal de La Laguna, Spain, in 2015 and 2017, respectively. He is currently a Research Assistant with the Department of Computer Engineering and Systems, University of La Laguna. His research interests include the areas of programming models for parallel computing, performance optimization and portability, and code acceleration on mobile architectures and high performance computing systems.

FRANCISCO ALMEIDA received the degree in mathematics, the M.Sc. degree in mathematics, and the Ph.D. degree in computer science from the University of La Laguna, San Cristóbal de La Laguna, Spain, in 1989, 1992, and 1996, respectively. He is currently a Professor with the Department of Computer Engineering and Systems, University of La Laguna. His research interests include primarily in the areas of parallel computing, parallel algorithms for optimization problems, parallel system performance analysis and prediction, skeleton tools for parallel programming, and web services for high performance computing and grid technology.