The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-05924-1

Parallel programming in mobile devices with FancyJCL

Sergio Afonso[1] · Óscar Gómez-Cárdenes[1] · Paula Expósito[1] · Vicente Blanco[1] · Francisco Almeida[1]

Accepted: 21 January 2024

Abstract
Mobile devices and handheld systems, such as the smartphones and tablets universally extended, are becoming increasingly powerful. Their basic hardware configuration is usually state-of-the-art heterogeneous architectures consisting of multicore processors and some kind of accelerator such as GPUs or DSPs. Specific code adapted to the architecture is mandatory if high-performance computation is required and low-level libraries and parallelism are needed, which constitutes an important barrier for the usual developer in such devices. In this context, we propose the FancyJCL framework. It provides a high-level abstraction layer that hides implementation details and allows to develop parallel programs for mobile devices. The target platform for FancyJCL is mainly Android and Java developers due to their high market penetration. A very simple, seemingly sequential encoding results in parallel efficient OpenCL code. FancyJCL is itself based on the Fancier framework, which enables optimal memory management across memory spaces on unified memory systems. Benchmarks of FancyJCL code developed for a wide range of image processing algorithms show good performance with low development effort.

Keywords Application programming interfaces · Hardware acceleration · Heterogeneous systems · Image processing · Mobile computing · Parallel programming · Performance analysis

1 Introduction

Since public adoption of mobile platforms is pervasive and their unused performance potential is so high, it is important to consider compatibility with these platforms in designing new parallel programming models and tools. Most modern mobile devices run Android or iOS Operating Systems (OS), the former being much more widespread. The development of native Android applications is done mainly in

Sergio Afonso, Óscar Gómez-Cárdenes, Paula Expósito, Vicente Blanco and Francisco Almeida have contributed equally to this work.

Extended author information available on the last page of the article the Java and Kotlin programming languages, compiled into Java bytecode and seamlessly interoperating and running on top of the same runtime. Using this type of managed high-level programming language simplifies the development of interactive applications at the cost of adding overhead that hinders execution performance. The potential of providing parallel and accelerated execution capabilities to Java-based applications 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 higher levels of abstraction from the hardware and OS. However, some of these shortcomings have improved over time due to improvements in virtual machine optimizations and Just-in-Time (JIT) compilation techniques [2].

Nevertheless, Java execution on accelerators still requires specialized Java Virtual Machines (JVMs) or libraries or code translation tools that integrate Java bytecode execution and native libraries with low-level access to accelerators. As a consequence, the programmability effort is still very high. Alternatives as language bindings like JCUDA [3] or JOCL [4] are geared toward experts and do not significantly reduce the programming effort. There are multiple approaches to address the issue of accelerating Java applications trying to reduce programming complexity. Projects like Sumatra [5] and TornadoVM [6] propose implementation features or extensions to JVMs and standard libraries that require a reduced effort of developers. Although working at the JVM level has the benefit of being able to decide at runtime, through a JIT compilation process, the processor to target for each task or a dynamic task migration is not always possible to rely on the use of custom JMVs, such as in Android devices.

Tools such as Paralldroid [7], Aparapi [8], Rootbeer [9], or ParallelME [10], on the other hand, can work on standard JVMs, and they can translate Java code or compiled bytecode into native or accelerated implementations. Each alternative defines its parallel programming model on top of Java, adding certain restrictions on 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. Despite these efforts, none of the existing approaches has been widely adopted by the developer or scientific community, so there is no standard system that implements hardware acceleration of Java code yet.

In the Fancier framework [11], we proposed 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 the value semantics used in native languages, allowing a simpler mapping of Java code to C/C++ or OpenCL for accelerators. Fancier provides containers with transparent zero-copy memory support on unified memory systems while providing direct access to the memory from the Java, native, and OpenCL contexts. This is particularly relevant on mobile SoCs where memory copy overhead can easily become a performance bottleneck. The approach does not require modification to the Java compiler or JVM; it only depends on OpenCL.

In this paper, we present the FancyJCL framework,[1] and we propose a high abstraction layer that hides implementation details when developing parallel Java programs for mobile devices. Genericity, flexibility, and efficiency are basic issues in the design strategy. Parallelism is provided transparently through a sequential interface so that sequential users may access the parallel system. Our proposal is close to Aparapi [8]. However, they operate at the bytecode layer which may generate dependences on the VMs and could penalize the efficiency. FancyJCL is based on the Fancier framework and provides a set of "sequential" classes, methods, and data structures that encapsulate access to the parallel context, providing easy development and fast prototyping. No previous knowledge is needed about JNI and OpenCL since all the glue code for JNI and OpenCL is wrapped into FancyJCL. A very simple, apparently sequential encoding produces parallel and efficient OpenCL code. This ease of programming increases development productivity, improves the delivery of new parallel applications due to the rapid development time, and contributes to the efficient exploitation of new emerging architectures. FancyJCL also benefits from some of the Fancier advantages since OpenCL has been chosen as the hardware acceleration backend, support for general-purpose computations and widespread adoption among all types of architectures, from low-power SoC to state-of-the-art high-performance computing accelerated distributed systems, is provided. The efficient memory management avoids unnecessary copies among the different memory spaces and increases the performance. The decoupled design of the FancyJCL classes is introduced without loss of efficiency. The benchmarks of FancyJCL-generated code show good speedups and ease of use on different mobile devices tested over a wide range of representative image processing kernels.

The rest of the paper is structured as follows: Section 2 introduces the process by which Android applications are compiled and executed. Section 3 presents a summary of the Fancier framework and its advantages. Section 4 describes in detail the building of FancyJCL on top of Fancier. Section 5 includes the performance analysis developed to validate our approach, and in Sect. 6, we give our conclusions and future related lines of work.

2 Application execution on android

Android is a Linux-based operating system mainly designed for mobile devices, such as smartphones and tablets. Several development models are supported to manage the resources allocated to each application. Each model has different features that have to be used in different parts of the application to get the best

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

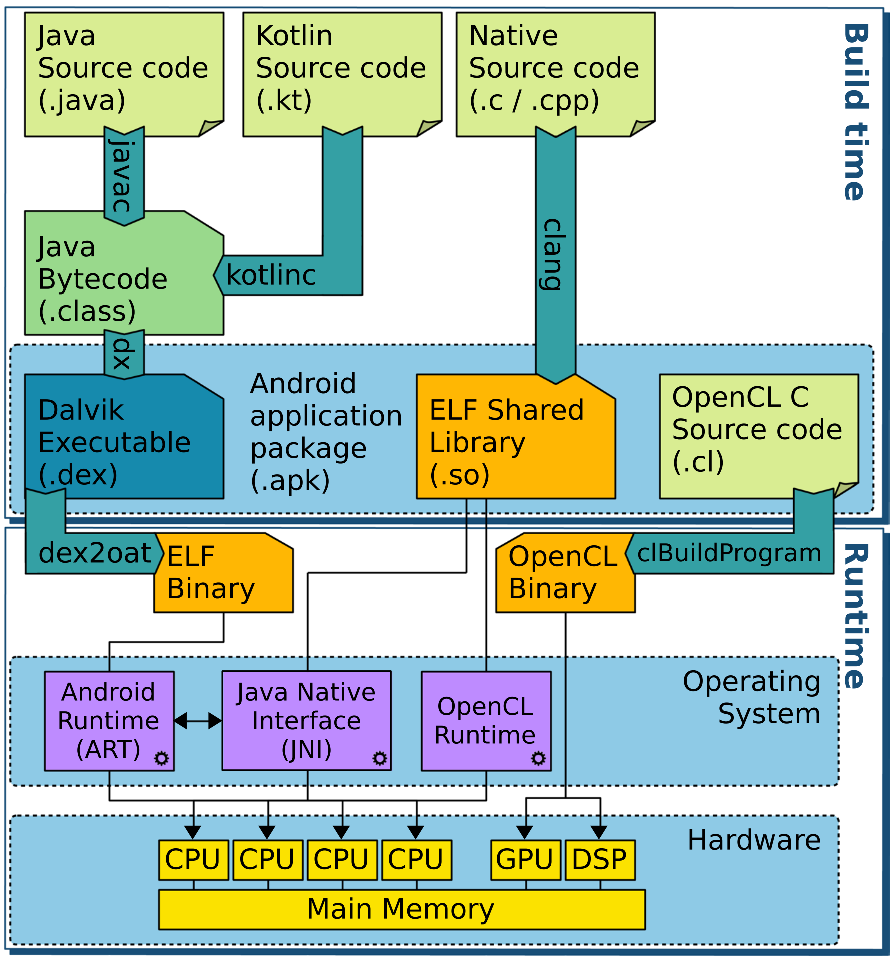


Fig. 1 Compilation and execution of an Android application

compromise between performance and development cost. These development models and their implementation trade-offs are outlined in this section.

Managed execution Android native applications are developed mainly using the Java and Kotlin managed programming languages, providing simple development environments with reasonable performance. Both are compiled into Java Bytecode, allowing transparent interoperability between those languages. However, their higher level of abstraction comes with a performance cost over native programming languages. To mitigate this, since Android 5.0, Java Bytecode is transformed and optimized in various stages until it is compiled ahead of time (AOT) into binaries that run natively with help from Android Runtime (ART) [12], as shown in Fig. 1. Recent Android releases use a hybrid just-in-time

(JIT) and AOT profile-guided optimization process [13]. Its performance, in many cases, is still far from that of the Native code.

Native execution Java applications can interface with native code by using the Java Native Interface (JNI) [14], also supported by Android [15]. The native implementation of the method require some JNI API calls as a bridge between Java and C to the actual work to be done by it, adding some development complexity and execution overhead. As an advantage, it allows to link and use Android-supported native system libraries (OpenGL, Vulcan...) or any other native independent library, and existing native code can be integrated instead of re-implementing it in Java or Kotlin. Compute or memory-intensive codes can be accelerated.

Accelerated execution The acceleration of regular Android applications is possible through the use of the officially supported Renderscript language [16]. Conversely, OpenCL is a multi-platform standard 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 or Qualcomm. Although this alternative has not received official support from Android, its widespread adoption, higher performance potential, and finer-grained control make it an interesting alternative. Additionally, the officially supported framework for Android acceleration from version 12 is Vulkan instead of RenderScript, for which work is being done to allow OpenCL kernels to run [17]. 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 they demand a higher development effort to be integrated into a managed application written in Java or Kotlin and a steeper learning curve. This is why approaches like those presented in this paper are still relevant and necessary.

3 The Fancier framework

In this section, we describe the Fancier framework [11], a multi-platform, modular, and extensible programming interface to accelerate Java applications without requiring any special features from the JVM they are running in. FancyJCL is supported on top of the Fancier execution model.

3.1 Overview

Fancier uses OpenCL as the high-performance multi-platform backend for accelerated execution. It targets Linux and Android platforms and defines a Java and Native API modelled after the OpenCL C programming language for accelerators. Fancier provides several utilities that simplify the integration of C/C++ and OpenCL execution to a Java application efficiently. Two main features from Fancier have been decisive in building FancyJCL:

- Fancier defines a Java and C/C++ subset of features and functions where the mismatch between the reference semantics of Java and the value semantics of C/C++ and OpenCL is addressed. Fancier forces value semantics by returning a new result object when operating. The pressure on the Java memory management is reduced by pre-allocating objects for all intermediate results. The Fancier unification of the Java, Native, and OpenCL languages greatly reduces development costs and has no performance penalty. Fancier facilitates the translation of a standardized Java subset to C/C++ and OpenCL for accelerated execution into a Java application and their runtime integration by creating a unified interface and library.
- The information flow between Java, C/C++ and OpenCL is difficult and requires manual management. There are multiple ways of performing this, but most incur in performance penalties that may go unnoticed. Fancier implements optimal strategies for container data types that work transparently and significantly reduces the development cost. Unified memory systems allow 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. The Fancier memory management strategy features efficient and transparent zero-copy read and write access from all contexts.

Accelerated Java applications using Fancier for OpenCL execution are able to take full advantage of all the features of the OpenCL 1.1 standard, with the advantage that passing data between the Java, C/C++ and OpenCL layers is much simpler, and the control flow between Java and C/C++ is more seamless. As a result, Fancier enables the design and implementation of parallel accelerated programming models on top of Java, and automated transformation tools can easily generate high-performance Fancier Native or OpenCL C code.

3.2 Fancier: APIs and memory management

The Fancier framework provides three layers: the Fancier Java API, the Fancier Native API, and the Fancier OpenCL API. The three layers implement all the functionality to allow an algorithm to be expressed similarly in Java, C++ or OpenCL so its translation from and to any of them is trivial. Arrays, vectors of sizes 2, 3, 4 and 8 and basic types `Byte`, `Short`, `Int`, `Long`, `Float`, and `Double` are supported in the three layers. All Fancier containers are allocated through OpenCL API calls. The functions for managing buffers are much simpler than manually writing JNI and OpenCL API calls. The Fancier Native API allows to obtain and compile OpenCL C kernels at runtime that add math functions supported by Fancier Java and Fancier Native, which are not included in the OpenCL C standard library. In the case of `DirectBuffer` Java objects, Fancier uses the native address to provide zero-copy operations.

Data transfers between contexts have great implications for performance. An OpenCL-accelerated Java application deals with three independent memory spaces:

the Java-managed heap, the OS-managed native heap, and the OpenCL device memory. Fancier handles the memory buffers efficiently, taking advantage of unified memory and avoiding unnecessary copies. The Fancier functions also manage memory synchronizations. Moreover, Fancier includes methods to get the `ByteBuffers` and copies of their content when it is assumed that the copies will perform more efficiently than the direct use of the `ByteBuffer`. In summary, Fancier enables transparent use of the unified memory architecture, avoids unnecessary memory movements, and allows reading and writing data from the three layers. Java, Native, and OpenCL.

4 The FancyJCL framework

This section presents the FancyJCL framework, a high-level tool oriented to Java developers willing to accelerate their applications on mobile devices. Core ideas as ease of use, genericity, flexibility, and efficiency are basic goals of the design strategy in FancyJCL.

4.1 Overview

FancyJCL provides a high-level sequential abstraction layer so that users without any expertise in native or parallel programming can use it in a simple way. FancyJCL collects a package of *sequential* classes, methods, and data structures that encapsulate access to native and parallel contexts, allowing easy development and fast prototyping. No prior knowledge is needed about parallelism, JNI or OpenCL since all the glue code for JNI and OpenCL is wrapped into FancyJCL. A very simple, apparently sequential encoding produces parallel efficient OpenCL code. Since FancyJCL is based on Fancier, it is also multi-platform, extensible, and does not require any special features from the JVM they are running in, so the generated parallel code can be executed into a mobile device but also on server or desktop computers.

Genericity appears naturally since OpenCL is the hardware acceleration backend, with support for general-purpose computations, fine-grained hardware control, and widespread adoption among all types of architectures. The object-oriented design makes FancyJCL a flexible tool that can be easily extended to new scenarios. For example, optimization techniques such as vectorization or tiling could be introduced in a transparent manner. The Fancier parallel structures are decoupled from the FancyJCL classes so no loss of efficiency is introduced by the approach. The zero-copy memory support provided by Fancier avoids unnecessary copies among the different memory spaces, increasing the performance. Improved application quality, increased use of parallel architectures by non-expert users, rapid inclusion of emerging technology benefits into their systems, and dynamic and transparent optimization enhancements are some of the benefits that can be reached.

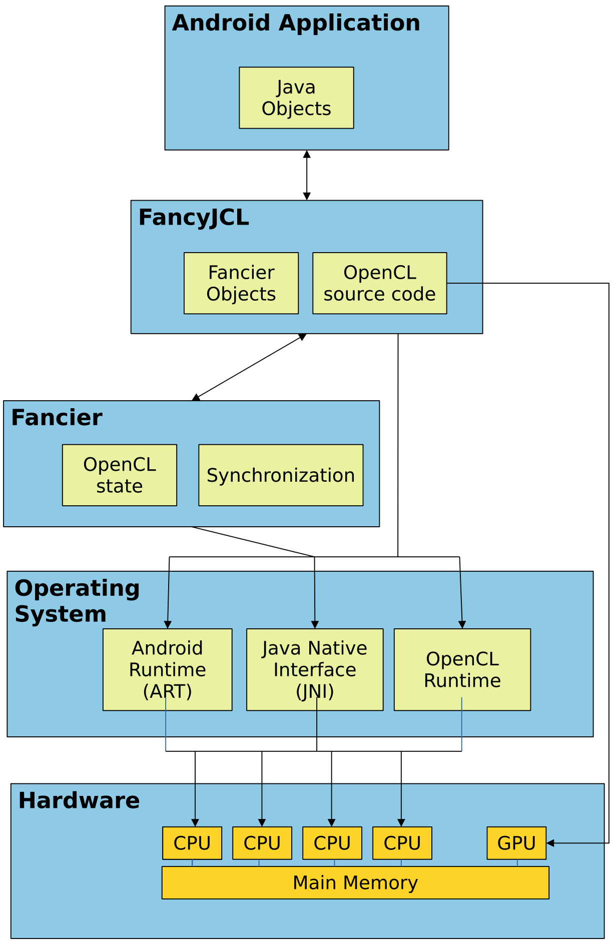
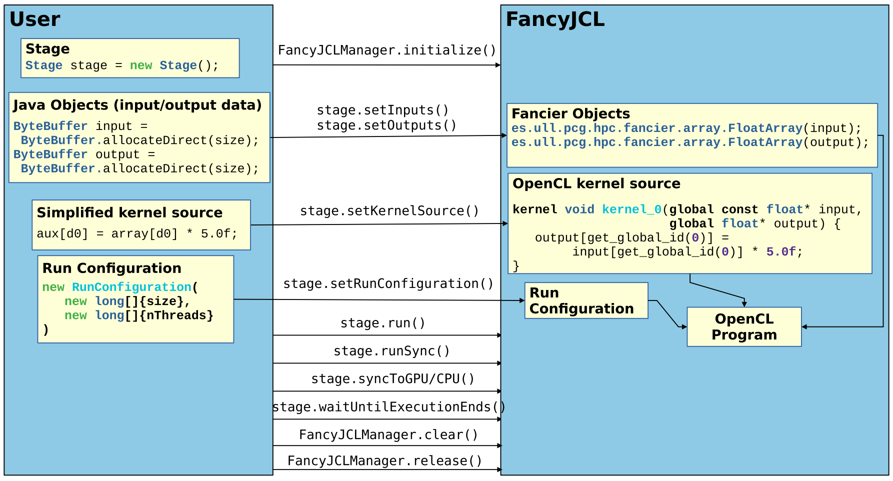


Fig. 2 The FancyJCL architecture

4.2 Design

Figure 2 shows a general overview of the FancyJCL architecture. FancyJCL builds a layer on top of Fancier that encapsulates the Fancier initialization, the glue code necessary for the JNI and OpenCL generation, the mapping to Fancier data structures and objects, the generation of the functional Kernel including inputs and outputs, and also the full OpenCL program and the synchronization among the different contexts. Some of these actions are supported by Fancier, and the FancyJCL interface wraps up and adapts the user calls but, in some others, more elaborated extra code is added internally in FancyJCL to generate new code or before a call to the Fancier framework is performed.

Figure 3 goes deeper in the architecture and shows a set of representative FancyJCL classes, objects and methods. The FancyJCL environment is initialized by



```
Stage stage = new Stage();
```

```
ByteBuffer input =
 ByteBuffer.allocateDirect(size);
ByteBuffer output =
 ByteBuffer.allocateDirect(size);
```

```
aux[d0] = array[d0] * 5.0f;
```

```
new RunConfiguration(
    new long[]{size},
    new long[]{nThreads}
)
```

```
es.ull.pcg.hpc.fancier.array.FloatArray(input);
es.ull.pcg.hpc.fancier.array.FloatArray(output);
```

```
kernel void kernel_0(global const float* input,
                     global float* output) {
   output[get_global_id(0)] =
        input[get_global_id(0)] * 5.0f;
}
```

Fig. 3 FancyJCL use case diagram

calling the initialize() method of the FancyJCLManager static class. This initializes the FancyJCL and Fancier contexts at the same time and creates the JNI, the OpenCL context and the OpenCL queues. The list of namespace and attributes that will be passed through the different contexts is created.

The FancyJCL Stage class is designed to create execution units that manage information about an algorithm, its parameters, and the configuration run, i.e., the range of the index variables and how many GPU threads to use, among others. Objects declared in the Stage class are intended to run a parallel process on input data objects to produce output data objects. The setInputs and setOutputs methods of a Stage object will manage the corresponding java objects. FancyJCL converts them into Fancier objects; if the declared object belongs to the DirectBuffer class (or a derived from it), only pointer reassignments to the Fancier objects will be needed, and there will not be data copies among the different contexts. This has an important impact on overall performance. Otherwise, a copy is performed into a Fancier object, and from that moment on, no additional copies among the different contexts are needed. The parallel procedure is assumed to be a simplified kernel, defining the operation to be applied to the input and output data, passed through the setKernelSource method. The predefined variables $d_0, \dots, d_k, \dots, d_{n-1}$ manage the index on dimension $k-th$, which eases and encapsulates the call to the get_global_id(k) OpenCL kernel function. Before an execution is launched the length and number of threads to use per dimension must be fixed (setRunConfiguration()). This method calls to the OpenCL functions to create the OpenCL kernel. The Stage class provides methods to run the kernel, including synchronous blocking and asynchronous executions of several kernels, and the corresponding methods to ensure synchronization among the different contexts. An attribute list created in FancyJCLManager keeps a list of references to the Stage which addresses the attributes. This allows the output to be the input of the same kernel or a different kernel.

```
// Initialize FancyJCL
FancyJCLManager.initialize(
               getApplicationContext()
                   .getCacheDir()
                   .getAbsolutePath()
               );
// Memory allocation
ByteBuffer input0 = ByteBuffer
                    .allocateDirect(size);
ByteBuffer input1 = ByteBuffer
                    .allocateDirect(size);
ByteBuffer output = ByteBuffer
                    .allocateDirect(size);
// or
// int[] input0 = new int[size];
// int[] input1 = new int[size];
// int[] output = new int[size];

// Constants declaration
float k0 = 0.3f;
float k1 = 0.7f;

// Create an execution Unit
Stage stage = new Stage();

// Set inputs and outputs
stage.setInputs(Map.of("input0",input0,
                       "input1",input1,
                       "k0", k0, "k1", k1));
stage.setOutputs(Map.of("output", output));

// Create a simplified Kernel
stage.setKernelSource("""
  // Get pixels and multiply by constant
   float p0 = input0[d0] * k0;
   float p1 = input1[d0] * k1;
   // Store pixel
   output[d0] = (int) round(p0 + p1);
  """);

// Fix size elements in the first dimension
// and numThreads threads to be launched
stage.setRunConfiguration(
  new RunConfiguration(new long[]{size},
  new long[]{numThreads}));

// Run the kernel and synchronize
// data among the contexts
stage.runSync();

// Clear the FancyJCL state
FancyJCLManager.clear();
```

Fig. 4 Computing in FancyJCL a weighted average of elements of two buffers and storing it in a third buffer

4.3 FancyJCL example

An example illustrating the use of FancyJCL appears in Figs. 3 and 4, where a simple code is shown that computes the average of two buffers. The flow control follows the steps previously described. No native code is included by the user into their application nor glue code. Every declaration, definition, and method call adopt the natural coding style in Java. The user only needs to know two FancyJCL classes, `FancyJCLManager` and `Stage` and some of their methods. The only difference is the notion of a simplified kernel. The user must understand how to express the computation of an input data item to produce the output. This is straightforward if the user is familiar with the CUDA or OpenCL streaming programming model, but, in any case, it involves a quite simple semantic and does not require too much effort from the user side. Some attention must also be paid by the user to the synchronization of the results. This piece of program (about 20 lines) encapsulates a code of about 56 lines.

Another important element that usually goes unnoticed is the effort involved in the cross-compilation for this case, where java and native code are mixed. Since FancyJCL is delivered in a precompiled package, the extra compilation stage and CMake management are removed from the development cycle. Compiling is faster, and the knowledge required to do it is less.

Table 1 Hardware platforms

	Xiaomi Mi Mix 2	Snapdragon 865 HDK	Vivo iQOO 7
OS	Android 9	Android 10	Android 11
SoC	Qualcomm Snapdragon SD845	Qualcomm Snapdragon SD865	Qualcomm Snapdragon SD888
CPU	4 Kryo 385@1.7GHz + 4 Kryo 385@2.8 GHz	4 Kryo 585@2.1GHz + 3 Kryo 585@2.42 GHz + 1 Kryo 585@2.84GHz	4 Kryo 680@1.8GHz + 3 Kryo 680@2.42 GHz + 1 Kryo 680@2.84GHz
GPU	Adreno 630@670–710 MHz	Adreno 650@250–587MHz	Adreno 660@700MHz
RAM	6GB LPDDR4x@1866MHz	6GB LPDDR5@2750MHz	8GB LPDDR5@3200MHz
Year	2017	2019	2021

5 Evaluation

Three Android devices have been used to evaluate the performance improvements of the FancyJCL API over Java reference implementations (Table 1). Their results are representative of a range of recent SoCs present in modern Smartphones. Two of the devices are off-the-shelf products, and the other is an HDK (mobile Hardware Development Kit). FancyJCL takes advantage of two main key aspects: the existence of a GPU that is programmable using OpenCL and the unified memory model that allows for implementing zero-copy strategies when executing code in CPU and GPU contexts.

Several image filter kernels have been developed to evaluate the performance of the FancyJCL library. The kernels have been programmed naively, meaning that no parallel strategy has been applied to the FancyJCL implementation, such as loop unrolling, vectorization, or sharing memory in a workgroup (local memory). The only strategy followed is to establish a large global workgroup size. The image processing kernels have been implemented using a 32-bit RGBA pixel format. The collection of kernels implemented is the following:

- **Bilateral** An edge-preserving smoothing filter is a stencil code; each neighbor is weighted according to its color and distance to the center pixel.
- **Median** A median filter 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** A filter that applies pre-selected colors to given ranges of input pixel intensity is a pixel-wise kernel that is called multiple times, one per range. We use five ranges in our testing.
- **Levels** A pixel-wise kernel applies a saturation and contrast levels change to an image.
- **Fisheye** A distortion kernel applies a fisheye 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.

- **Contrast** A pixel-wise kernel applies a parameterized contrast enhancement of an image.
- **Convolution 5×5; 3×3** Convolution kernels using a 5×5 or 3×3 mask implemented without loops.
- **GrayScale**: A pixel-wise kernel converts a color image to gray scale.
- **GaussianBlur**: A smoothing filter based on the Gaussian function 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 because it is separable.

Each of these kernels has been implemented in Java and FancyJCL, the Java reference implementation uses Java arrays (`byte []`) to fetch, process, and write pixel data and the FancyJCL is a parallel OpenCL implementation using `DirectBuffer` Java Buffers and taking advantage of the FancyJCL library.

Due to the highly dynamic performance characteristics of these SoCs and because of the power and thermal constraints they have, the testing algorithms must be repeated enough times so that the standard deviation is low enough for the average to be considered a valid measurement. Also, in the multiple-context application such as the use case of executing in the GPU, the benchmarking process must be aware of when to synchronize with the CPU. If `DirectBuffer` Objects are used in an architecture that implements unified memory, synchronization of the input and output memory buffers is not necessary. To achieve fairness, the measurement process in GPU implementations is the following:

- Start measuring time
- Enqueue execution of the algorithm on GPU `N` times
- Wait until the OpenCL queue finishes
- End `time_measurement`
- Return `time_measurement / N`

The class `FancyJCL.Benchmark` allows for executing a benchmark a number of times and includes a synchronization prologue that will only be executed once but will be included in the measured time. The benchmarking application was compiled in release mode, producing native binaries in arm64-v8a mode (64-bit). The implementation of the image-processing kernels was evaluated over the following set of image inputs of different sizes: **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 number of executions was variable for each experiment. A fixed execution time of at least 30 s was set as `experiment_time`. The number of executions `N` is therefore the amount needed to take the `experiment_time`. In all cases, `N` was greater than 10 and the measured standard deviation was lower than `0.01`.

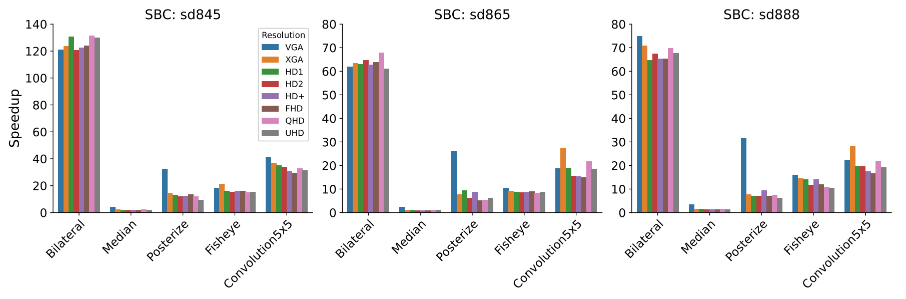


Fig. 5 Speedups of the Java averaged execution time by the FancyJCL equivalent implementation of 5 groups of kernels with 8 resolutions. One subplot per SoC

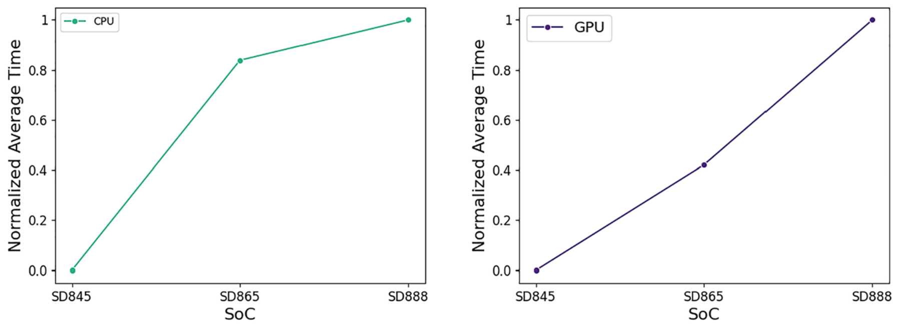


Fig. 6 Normalized average of time measurements

Table 2 shows the average execution times and speedups obtained from the computational experience developed. Figure 5 shows the speedups for a selected group of kernels on each SoC. For almost every experiment, the average execution time of the FancyJCL version was shorter than that of its Java reference implementation. The average speedup is of 23. This means that by using this library, not only is a significant increase in speed gained, but also the complexity of the code is greatly reduced by hiding all the OpenCL library calls and the native C++ code. The worst case is the **Mean** filter, since it fetches a large number of pixels to be sorted, and a local memory strategy should be used to achieve a significant speedup. The best case is the **Bilateral** filter, where most of the computational cost lies on floating point mathematical operations such as multiplication and power because their Adreno implementations are much faster.

The speedups achieved in the SoC SD845 are about twice those of the SD865 or SD888. This is explained since the CPU and the GPU did not evolve at the same rate between SD845 and SD865, while they did between SD865 and SD888. The difference in performance between the CPU and GPU is greater for the SD845 than for the SD865 or SD888. This can be better appreciated using our data in Fig. 6, where the normalized execution times are compared for the three SoC and for CPU and GPU. The GPU evolution behaves almost linearly, while the CPU evolution behaves exponentially.

Table 2 Mean of execution times in milliseconds

Filter	Res.	sd845			sd865			sd888		
		Java	FancyJCL	SpeedUp	Java	FancyJCL	SpeedUp	Java	FancyJCL	SpeedUp
	VGA	407.36	3.36	121.24	192.26	3.10	62.02	146.87	1.96	74.93
	XGA	1013.65	8.20	123.62	488.66	7.69	63.54	351.58	4.96	70.88
	HD1	1204.78	9.22	130.67	563.48	8.94	63.03	399.11	6.16	64.79
	HD2	1318.84	10.94	120.55	633.75	9.79	64.73	448.24	6.65	67.40
Bilateral	HD+	1792.37	14.61	122.68	864.35	13.74	62.91	621.09	9.49	65.45
	FHD	2580.64	20.81	124.01	1242.93	19.43	63.97	889.58	13.60	65.41
	QHD	4748.36	36.14	131.39	2287.54	33.67	67.94	1654.15	23.68	69.85
	UHD	10388.87	79.79	130.20	5023.37	82.05	61.22	3595.64	53.16	67.64
	VGA	392.08	88.97	4.41	170.81	70.51	2.42	154.98	43.75	3.54
	XGA	559.73	231.53	2.42	214.89	179.54	1.20	180.36	111.21	1.62
	HD1	601.30	270.88	2.22	238.55	209.40	1.14	198.28	130.71	1.52
	HD2	648.80	307.32	2.11	225.85	236.03	0.96	193.20	147.46	1.31
Median	HD+	866.83	423.69	2.05	304.20	327.12	0.93	264.97	203.02	1.31
	FHD	1223.02	613.13	1.99	434.74	475.13	0.91	374.80	291.39	1.29
	QHD	2514.40	1087.86	2.31	1006.75	848.10	1.19	787.04	517.95	1.52
	UHD	5182.31	2421.49	2.14	2026.95	1895.44	1.07	1616.36	1160.93	1.39
	VGA	13.00	0.40	32.5	9.34	0.36	25.94	7.33	0.23	31.87
	XGA	14.03	0.96	14.61	6.41	0.82	7.82	4.14	0.53	7.81
	HD1	14.92	1.14	13.09	9.25	0.98	9.44	4.90	0.68	7.21
	HD2	15.93	1.31	12.16	9.81	1.56	6.29	5.50	0.77	7.14
Posterize	HD+	21.72	1.74	12.48	16.97	1.91	8.88	9.21	0.98	9.4
	FHD	32.59	2.43	13.41	18.11	3.49	5.19	10.31	1.45	7.11
	QHD	53.51	4.36	12.27	28.06	5.18	5.42	18.94	2.53	7.49

Table 2 (continued)

		sd845			sd865			sd888		
Filter	Res.	Java	FancyJCL	SpeedUp	Java	FancyJCL	SpeedUp	Java	FancyJCL	SpeedUp
	UHD	117.55	12.54	9.37	61.97	9.99	6.20	35.42	5.68	6.24
	VGA	49.99	0.67	74.61	7.37	0.44	16.75	6.82	0.21	32.48
	XGA	23.79	0.89	26.73	12.72	0.83	15.33	6.05	0.49	12.35
	HD1	23.66	1.18	20.05	15.63	1.43	10.93	7.75	0.58	13.36
	HD2	26.75	1.20	22.29	17.27	1.92	8.99	7.98	0.65	12.28
Levels	HD+	35.04	1.73	20.25	24.77	2.48	9.99	12.28	1.07	11.48
	FHD	52.43	2.56	20.48	27.64	3.51	1.87	17.13	1.43	11.98
	QHD	86.30	4.40	19.61	56.21	4.84	11.61	29.12	2.89	10.08
	UHD	223.13	9.65	23.12	113.49	9.91	11.45	67.82	4.83	14.04
	VGA	76.47	4.19	18.25	40.50	3.85	10.52	38.55	2.40	16.06
	XGA	235.93	11.12	21.22	90.57	9.71	9.33	87.35	6.01	14.53
	HD1	209.92	13.01	16.14	100.75	11.34	8.88	99.32	7.04	14.11
	HD2	227.91	14.86	15.34	113.01	13.00	9.69	95.17	8.00	11.90
Fisheye	HD+	330.27	20.56	16.06	152.66	17.16	8.90	154.96	10.97	14.13
	FHD	465.10	28.95	16.07	219.47	24.46	8.97	188.64	15.83	11.92
	QHD	776.28	51.44	15.09	380.27	45.62	8.34	312.67	28.74	10.88
	UHD	1735.60	113.28	15.32	855.84	97.16	8.81	690.15	65.57	10.53
	VGA	16.37	0.52	31.48	11.10	0.46	24.13	8.54	0.34	25.12
	XGA	20.04	1.34	14.96	9.27	1.67	5.55	4.24	0.74	5.73
	HD1	24.26	1.40	17.33	14.06	1.88	7.48	9.03	0.88	10.26
	HD2	24.94	1.55	16.09	15.85	1.38	11.49	10.14	1.03	9.84
Contrast	HD+	35.27	2.17	16.25	18.01	2.80	6.43	13.12	1.53	8.58
	FHD	51.82	3.22	16.09	21.41	5.41	3.96	16.98	2.26	7.51

Table 2 (continued)

		sd845			sd865			sd888		
Filter	Res.	Java	FancyJCL	SpeedUp	Java	FancyJCL	SpeedUp	Java	FancyJCL	SpeedUp
	QHD	92.52	6.09	15.19	37.69	6.24	6.04	25.19	4.00	6.30
	UHD	205.35	14.82	13.86	80.95	11.66	6.94	47.73	8.35	5.72
	VGA	89.88	2.19	41.04	36.72	1.95	18.83	26.58	1.19	22.34
	XGA	202.51	5.47	37.02	112.46	4.09	27.50	91.58	3.24	28.27
	HD1	231.60	6.58	35.20	108.34	5.71	18.97	76.72	3.87	19.82
	HD2	253.70	7.46	34.01	93.27	5.98	15.60	78.72	3.99	19.73
Convolution 5×5	HD+	339.92	10.89	31.21	129.88	8.44	15.39	100.93	5.77	17.49
	FHD	453.16	15.36	29.50	182.59	12.13	15.05	138.67	8.33	16.65
	QHD	890.39	27.11	32.84	456.13	20.90	21.82	334.08	15.25	21.91
	UHD	1903.40	60.22	31.61	913.72	49.04	18.63	647.45	33.71	19.21
	VGA	13.32	0.44	30.17	5.63	0.38	14.82	11.00	0.23	47.83
	XGA	9.33	0.92	10.14	5.23	0.83	6.30	3.19	0.54	5.91
	HD1	12.10	1.15	10.52	7.57	1.06	7.14	3.78	0.64	5.91
	HD2	13.44	1.27	10.58	9.11	1.92	4.74	5.25	0.75	7.00
Grayscale	HD+	15.66	1.76	8.90	10.88	2.12	5.13	6.42	1.09	5.89
	FHD	22.34	2.47	9.04	15.35	3.20	4.80	9.07	1.69	5.37
	QHD	46.79	4.38	10.68	24.33	5.35	4.55	13.53	3.03	4.47
	UHD	92.05	10.48	8.78	51.63	9.45	5.46	28.95	6.39	4.53
	VGA	44.63	1.55	28.79	16.30	0.95	17.16	12.72	0.63	20.19
	XGA	84.15	3.99	21.09	38.66	3.40	11.37	30.01	1.72	17.45
	HD1	92.65	4.48	20.68	40.13	3.80	10.56	26.72	2.18	12.26
	HD2	129.85	5.09	25.51	40.79	4.22	9.67	27.62	2.38	11.61
Convolution 3×3	HD+	164.50	7.04	23.37	55.35	5.52	10.03	41.43	3.37	12.29

Table 2 (continued)

Filter	Res.	sd845			sd865			sd888		
		Java	FancyJCL	SpeedUp	Java	FancyJCL	SpeedUp	Java	FancyJCL	SpeedUp
	FHD	194.53	10.82	17.98	74.79	7.20	10.39	53.64	4.89	10.97
	QHD	354.32	19.13	18.52	170.46	11.54	14.77	116.32	8.71	13.35
	UHD	719.95	41.73	17.25	284.46	24.46	11.63	199.14	19.21	10.37
	VGA	239.09	3.54	67.54	119.20	2.56	46.56	73.42	1.83	40.12
	XGA	415.69	9.26	44.89	142.87	6.35	22.50	105.44	4.45	23.69
	HD1	491.50	11.02	44.60	167.29	7.20	23.23	127.52	5.23	24.38
	HD2	572.96	12.35	46.39	189.30	8.27	22.89	139.29	5.94	23.45
GaussianBlur	HD+	757.25	17.36	43.62	254.75	11.36	22.43	190.78	8.10	23.55
	FHD	1106.58	24.60	44.98	365.29	15.86	23.03	278.61	11.68	23.85
	QHD	1927.03	43.09	44.72	648.62	27.24	23.81	488.22	20.45	23.87
	UHD	4348.09	103.44	42.03	1445.02	63.08	22.91	1086.51	46.71	23.26

6 Conclusion

We have presented FancyJCL, a framework that provides easy development and fast prototyping to generate parallel OpenCL codes in mobile devices. A set of Java classes is provided to the developer to fill code gaps in the apparently sequential methods of these classes. The set of classes made available to the user is quite reduced, so the learning curve is very low and the parallelism is offered in a transparent manner. Genericity and efficiency are guaranteed through the internal use of the Fancier Library. Fancier constitutes a promising backend for parallel programing models built on top. It is not constrained by any particular JVM and efficiently solves the problem of data movement among the various memory spaces involved in parallel computations in a mobile device. The ease of use and efficiency of FancyJCL are validated by benchmarking a wide range of representative image processing kernels on three different SoC architectures. The flexibility of the framework is inherent to the design of FancyJCL; for example, natural extensions to FancyJCL could provide support for Image2D, vectorization, or auto-tuning just by extending the `RunConfiguration` method. More elaborated extensions would allow us to generate OpenGL or Vulcan code, which is supported by a larger number of vendors, or generate code in a different language as JavaScript to cover a greater community of mobile developers.

Author Contributions These authors contributed equally to this work.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work has been partially funded by the Ministry of Science and Innovation of Spain through projects PID2019-107228RB-I00, TED2021-131019B-I00, and PDC2022-134013-I00; and by the CAPAP-H Spanish Network.

Data availability The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of interest The authors have no conflicts of interest to declare that are relevant to the content of this article.

References

1. Smith LA, Bull JM, Obdrizalek J (2001) A parallel java grande benchmark suite. In: SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, p 6. https://doi.org/10.1145/582034.582042

2. Shiv K et al (2003) Impact of JIT/JVM optimizations on JAVA application performance. In: 7th Workshop on Interaction Between Compilers and Computer Architectures (INTERACT-7), pp 5–13. https://doi.org/10.1109/INTERA.2003.1192351
3. Yan Y, Grossman M, Sarkar V (2009) JCUDA: a programmer-friendly interface for accelerating Java programs with CUDA. In: Sips H, Epema D, Lin H-X (eds) Euro-Par 2009 Parallel Processing. Springer, Berlin, pp 887–899
4. Hutter M. (2016) JOCL: Java Bindings for OpenCL. https://github.com/gpu/JOCL
5. OpenJDK: Project Sumatra. https://openjdk.java.net/projects/sumatra/
6. Fumero J et al (2019) Dynamic application reconfiguration on heterogeneous hardware. In: 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. ACM, New York, pp 165–178. https://doi.org/10.1145/3313808.3313819
7. Acosta A, Afonso S, Almeida F (2016) Extending Paralldroid with object oriented annotations. Parall Comput 57:25–36. https://doi.org/10.1016/j.parco.2016.04.003
8. Aparapi: API for data parallel Java. http://aparapi.com/
9. Pratt-Szeliga PC, Fawcett JW, Welch RD (2012) Rootbeer: seamlessly using GPUs from Java. In: 9th HPCC-ICESS Conference. IEEE, pp 375–380
10. Andrade G et al (2016) ParallelME: a parallel mobile engine to explore heterogeneity in mobile computing architectures. In: Dutot P-F, Trystram D (eds) Euro-Par 2016: Parallel Processing. Springer, Cham, pp 447–459
11. Afonso S, Almeida F (2021) Fancier: a unified framework for java, c, and opencl integration. IEEE Access 9:164570–164588. https://doi.org/10.1109/ACCESS.2021.3134788
12. Android Open Source Project: ART and Dalvik. https://source.android.com/devices/tech/dalvik/
13. Android Open Source Project: Implementing ART Just-In-Time (JIT) Compiler. https://source.android.com/devices/tech/dalvik/jit-compiler
14. Liang S (1999) The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley Professional, Palo Alto
15. Android Open Source Project: Get Started with the NDK. https://developer.android.com/ndk/guides
16. Android Open Source Project: RenderScript Overview. https://developer.android.com/guide/topics/renderscript/compute.html
17. Google: Clspv. https://github.com/google/clspv

Authors and Affiliations

Sergio Afonso[1] · Óscar Gómez-Cárdenes[1] · Paula Expósito[1] · Vicente Blanco[1] · Francisco Almeida[1]

✉ Vicente Blanco
vblanco@ull.es

✉ Francisco Almeida
falmeida@ull.es

Sergio Afonso
safonsof@ull.edu.es

Óscar Gómez-Cárdenes
ogomezc@ull.es

Paula Expósito
pexposit@ull.es

[1] Computer Science and Systems Department, Universidad de La Laguna (ULL), San Francisco de Paula s/n, 38270 La Laguna, Spain