SparkJNI
A Reference Design for a Heterogeneous Apache Spark Framework

Tudor Alexandru Voicu
SparkJNI

A Reference Design for a Heterogeneous Apache Spark Framework

by

Tudor Alexandru Voicu

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday 31st of August, 2016 at 14:00.

Student number: 4422066
Project duration: December 1, 2015 – August 31, 2016
Thesis committee: Dr. Zaid Al-Ars, QE, TU Delft, supervisor
Dr. Claudia Hauff, ST, TU Delft
Dr. Carmina Almudever, QE, TU Delft
Dr. Koen Bertels, QE, TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

The image used on the cover has been taken from
Abstract

The digital era's requirements pose many challenges related to deployment, implementation and efficient resource utilization in modern hybrid computing infrastructures. In light of the recent improvements in computing units, the defacto structure of a high-performance computing cluster, ordinarily consisted of CPUs only, is superseeded by heterogeneous architectures (comprised of GPUs, FPGAs and DSPs) which offer higher performance and lower power consumption. Big Data, as a younger field but with a much aggressive development pace starts to exhibit the characteristic needs of its archetype and the development community is targeting the integration of specialized processors here, as well. The benefits do not come for granted and could be easily overshadowed by challenges in implementation and deployment when considering development time and cost. In this research, we analyze the state-of-the-art developments in the field of heterogeneous-accelerated Spark, the current Big Data standard, and we provide a reference design and implementation for a JNI-accelerated Spark framework. The design is validated by a set of benchmarked micro-kernels. The JNI-induced overhead is as low as 12% in access times and bandwidth, with speedups up to 12x for compute-intensive algorithms, in comparison to pure Java Spark implementations. Based on the promising results of the benchmarks, the SparkJNI framework is implemented as an easy interface to native libraries and specialized accelerators. A cutting-edge DNA analysis algorithm (PairHMM) is integrated, targeting cluster deployments, with benchmark results for the DNA pipeline stage showing an overall speedup of $\sim 2.7$ over state-of-the-art developments. The result of the presented work, along with the SparkJNI framework are publicly available for open-source usage and development, with our aim being a contribution to current and future Big Data Spark shift drivers.
Dedicated to my family, friends and to my girlfriend, Ioana.
List of Figures

1.1 Big data and HPC framework compared in terms of K-Means clustering performance [1]. ................................................................. 3

2.1 Native-Java application overview. ................................................. 6
2.2 Apache Spark most used transformations .................................. 13
2.3 Spark processing flow overview .................................................. 13

3.1 HeteroSpark simplified data flow concept ................................. 20

4.1 Spark infrastructure[2]. ................................................................. 25
4.2 The Spark-JNI data representation concept ................................ 27
4.3 Dataflow representation for a JNI-enabled Spark application 1 .... 28
4.4 The SparkJNI framework functional hierarchy ............................ 33
4.5 The SparkJNI CodeGen framework functional flow overview. Dashed line represents user code, whereas solid line represents code generated by the framework. .............................................. 35

5.1 Collect time for the access micro-kernel ..................................... 40
5.2 Average and cumulative results for native access time breakdown, for variable number of workers .................................................. 41
5.3 Memory copy micro-kernel benchmark ...................................... 42
5.4 Convolution speedup over Java execution time for the generated noise images batch. ................................................................. 43
5.5 Convolution speedup over Java execution time, for the real-world image batch. ................................................................. 44
5.6 Pi calculation based on random points. ...................................... 44
5.7 Benchmarks for SparkJNI-OpenCL vs SparkCL against the workload scenario (a) and OpenCL launch (b) ................................ 46
5.8 PairHMM alignment state machine in (a) and sequence alignment example in (b) [3] ................................................................. 48
5.9 PairHMM SparkJNI execution and data flow diagram. ............... 49
5.10 Analysis of the SparkJNI-induced overhead, for both online and offline runtimes, on the PairHMM FPGA-accelerated algorithm . .... 51
5.11 SparkJNI kernel call-stack representation ................................ 51
5.12 PairHMM performance evaluation of the three implementations .... 52
List of Tables

2.1 Comparison between JNI and JNA .......................... 11
3.1 Comparison between heterogeneous-enabling Java frameworks ............. 21
5.1 Test setup for the Spark-JNI micro-kernels benchmarking .................. 38
5.2 Test setup for OpenCL Pi algorithm benchmarking .......................... 45
5.3 Test setup for the PairHMM SparkJNI benchmarking ..................... 50
## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD</td>
<td>Advanced Micro Devices</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>APU</td>
<td>Accelerated Processing Unit</td>
</tr>
<tr>
<td>AVL</td>
<td>Georgy Adelson-Velsky and Evgenii Landis</td>
</tr>
<tr>
<td>CAPI</td>
<td>Coherent Accelerator Processor Interface</td>
</tr>
<tr>
<td>COM</td>
<td>Component Object Model</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>DLT</td>
<td>Doubly Logarithmic Tree</td>
</tr>
<tr>
<td>DNA</td>
<td>Deoxyribonucleic Acid</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast-Fourier Transform</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>GFLOP/s</td>
<td>Float Point Operations per Second</td>
</tr>
<tr>
<td>HDD</td>
<td>Hard Disk Drive</td>
</tr>
<tr>
<td>HMM</td>
<td>Hidden Markov Model</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>HSA</td>
<td>Heterogeneous System Architecture</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System</td>
</tr>
<tr>
<td>IDC</td>
<td>International Data Corporation</td>
</tr>
<tr>
<td>Java EE</td>
<td>Java Enterprise Edition</td>
</tr>
</tbody>
</table>
JDK  Java Development Kit
JIT  Just-in-time
JNA  Java-Native Access
JNI  Java Native interface
JVM  Java Virtual Machine
LFSR  Linear-Feedback Shift Register
MB/s  Megabytes per Second
MLLib  Machine Learning Library
OOP  Object-Oriented Programming
OpenCL  Open Computing Language
OpenMP  Open Multi-Processing
POV  Point-Of-View
RDD  Resilient Distributed Dataset
RMI  Remote Method Interface
SDK  Software Development Kit
SIMD  Single-Instruction Multiple-Data
SQL  Structured Query Language
UMA  Uniform Memory Access
VHDL  Very High Speed Integrated Circuit Hardware Description Language
Contents

List of Figures vii
List of Tables ix
List of Acronyms xii

1 Introduction 1
1.1 High Performance Computing vs Big Data 2
1.2 Project goal 3
1.3 Thesis outline 4

2 Background 5
2.1 Bridges from Java to native code 5
  2.1.1 The Java-Native Interface 6
  2.1.2 The Java-Native Access framework 9
  2.1.3 Discussions and performance evaluation 10
2.2 OpenCL 11
2.3 Apache Spark 12

3 Related Work 15
3.1 OpenCL bindings for Java 15
  3.1.1 Aparapi 15
  3.1.2 Java-OpenCL bindings 16
3.2 Performance-enriching frameworks for Spark 17
  3.2.1 Project Tungsten 17
  3.2.2 SparkCL 18
  3.2.3 HeteroSpark 19
3.3 Discussions and evaluation 21

4 The SparkJNI framework 23
4.1 Design Considerations 23
4.2 Spark architecture 24
  4.2.1 Data model 24
  4.2.2 Execution model 25
4.3 A JNI-accelerated Spark design 26
4.4 Implementation details 28
  4.4.1 Code generation engine 30
  4.4.2 Functional overview 34
4.5 Limitations 36
## 5 Experiments

5.1 JNI-accelerated Spark micro-kernels ........................................... 38
5.2 Results and discussions .......................................................... 40
5.3 Comparison with SparkCL ..................................................... 43
5.4 A SparkJNI FPGA PairHMM algorithm ................................. 47
  5.4.1 DNA analysis and the PairHMM ....................................... 47
  5.4.2 Implementation details .................................................. 48
  5.4.3 Evaluation ............................................................... 49

## 6 Conclusions and future perspectives

6.1 Conclusions ............................................................................. 53
6.2 Future perspectives ............................................................. 54

## A Publications

55
Introduction

The information era, our era, is a concept only coined recently due to the expansion of the Internet to nearly every household and person in the developed world. If until the late 90s there were only a few coarse-grained information streams reaching across continents, the picture today reveals that fine-grained Internet end-points (each user with its interaction - smartphone, laptop, smartwatch etc) are generating most of the data in a more unstructured form, increasing sizes and complexities than ever before.

While IDC [4] estimates a rate of 25% increase in data volume by 2020, with the advent of consumer-tailored content, complex modeling and machine learning integrated in consumer services and Web 2.0 applications, this figure can be amplified even more. Until the next "Moore's law" revolution, the increase in computational power of processors is not enough to keep up with these high demands, especially in traditional computing architectures.

At the rise of the Big-Data field, it was a completely distinguished domain that was targeting simple predictions, information extraction and market research, mostly serving commercial purposes. However, today we are witnessing a more aggressive expansion, since this field emulates to a more vast field of application. The paradigm serves today the needs of scientific and research fields, companies’ commercial and business purposes and end users as well.

Since we are relying more and more on Big Data tools and analysis to push forward our information era, we are spending exponentially more on building datacenters, clusters and burning more energy than ever before to keep them alive. While datacenters start to pose a great threat to nations’ sustainable energy growth, big players in the industry are looking to reduce consumption as much as possible, while also keeping costs under control (e.g. Facebook's open-air arctic server farms in Lulea, Sweden).

While once the Big Data standard, MapReduce [5] was initially targeted at "commodity hardware", when deployed by Google. Now, we cannot afford burning power in commodity hardware to keep up with the digital era's demands. The only solution is to retarget our development efforts towards energy efficiency in terms of computation (joule/operation). For this, the trend needs to be reverted a bit as we need to uncover a part of the abstraction
layer that has been installed over the hardware and low-level details with the advent of the Big Data field.

Developments in non-conventional computing units, such as FPGAs, GPUs and DSPs, offer a possibility in reducing the energy footprint of large-scale computing clusters, while also pushing forward in terms of performance. For achieving this purpose, the intersection of Big Data’s scalability, reach and ease-of-deployment, on one side and the specialization, expertise and efficiency in resource utilization of HPC, on the other side, needs to materialize.

1.1. High Performance Computing vs Big Data

For a better understanding of how the convergence of the two fields would look like, we need to deviate from evaluating the future and take a glimpse at the fundamental differences between the two fields, when the current trend was in incipient stages [6].

First, one fundamental difference between the two is regarding their target market: while HPC is targeting and accessible mostly to scientific communities, Big Data frameworks were meant to be deployable on commodity hardware, as emphasized by Google’s approach a decade ago with using Hadoop. Next, as the name suggests, Big Data emphasizes on large amounts of storage handled by inexpensive infrastructures, while HPC focuses on Big Compute: as much raw computing power as possible, with lower redundancy and failover options [6].

Big Data was not initially meant to leverage high-complexity algorithms, but instead handle data and information extraction as cost-efficient as possible. Then, it naturally lead to less constrained requirements on processing capabilities, while emphasizing on dense storage options.

Second, applications and infrastructure in Big Data were commonly allowed to fail, especially hardware, and handled properly. On the other side, HPC infrastructures are not designed as fault-tolerant and, usually, they’re not about redundancy, but high efficiency in resource utilization.

Regarding performance, the authors in [1] have realised a simple comparison between Spark, MPI and other programming paradigms based on a commonly-used kernel in machine learning applications (K-Means). While the MPI implementation leads the pack in terms of performance, Spark is 2x-10x slower, while Hadoop MapReduce takes 20x more execution time.

Last, because Big Data has always been targeted at commercial applications, and developed by the open-source community, the programming paradigms that it brought up focus on ease-of-deployment, have a wide-spread reach and aggressive development pace. In contrast, the high performance frameworks are targeting lower-level implementation details for achieving optimal resource usage and are only developed in academic and scientific environments. The specialization of HPC programming makes a non-introduced programmer’s task non-trivial.

Returning at the current state-of-the-art, we have to note another gap that is further increasing between the two fields: computing power and resource specialization, while the areas of application are overlapping more and more. Especially with data analytics, pattern recognition and machine learning, Big Data infrastructures are pushed to new limits in terms of computing power requirements. Whereas HPC infrastructures have naturally included massively parallel processors (GPUs) and reconfigurable computing units (FPGAs).
1.2. Project goal

In this thesis, we aim at providing a proof-of-concept framework design and implementation that accelerates Spark Java, with the usage of lower-level programming paradigms, such as OpenCL, CUDA, FPGA programming etc., by providing a JNI-enabled bridge between Spark's Java and C++. Furthermore, we aim to demonstrate the benefits of the framework by accelerating a series of micro-kernels. Finally, we target deployment of a novel PairHMM algorithm with Spark, through our reference framework (SparkJNI).

Since our research involves bridging the gap that appeared between the High Performance Computing and Big Data fields (by design and conception eras), there is need for clear identification of the foreseeable challenges before the conceptual foundation is layered. Therefore, we need raise the following questions:

- Can we get low native-access overhead in a heterogeneous Big Data-managed system? More formally, are there foreseeable solutions that can bring the performance of a Spark cluster close to HPC infrastructures?

- Assuming the existence of a Big Data framework that enables good acceleration on heterogeneous hardware, can we assure good compatibility with legacy infrastructures and libraries?

- Furthermore, can we achieve all of the above, with little to no compromise in terms of maintainability, scalability, fault-tolerance and ease-of-deployment?
1.3. Thesis outline

The remainder of the thesis will cover the following:

- Chapter 2. **Background** will present important details regarding the frameworks on which the reasoning is built on. A brief description and comparison is done on the field of Java-to-native bridges.

- Chapter 3. **Related work** offers a summary and analysis of existing research targeting integration of lower-level application processors into Spark and Java-managed systems.

- Chapter 4. **The SparkJNI framework** presents the steps taken and design choices for the proposed Spark-JNI integration architecture. Furthermore, the CodeGen SparkJNI framework is presented, in terms of architectural design, implementation and limitations.

- Chapter 5. **Evaluation and results**. Here, three micro-kernels are implemented and benchmarked for the purpose of evaluation of the JNI-induced overhead in the reference heterogeneous Spark design. Next, the SparkJNI framework is validated by a real-world integration of a novel PairHMM algorithm that is accelerated on FPGAs.

- Chapter 6. **Conclusions and future research**. In the last chapter, we will try to summarize our opinions on the field and its prospective options, as well as our view on the future of the SparkJNI framework. Last, the questions raised in Section 1.2 will be answered based on the outcomes of the presented research.
In this chapter, we are going to introduce the reader with outlines of relevant work on which this design is being built. We will start with Java-to-C/C++ integration bridges and continue with frameworks designated to leverage the use of the heterogeneous clustered compute infrastructures: OpenCL targets parallel acceleration in C/C++, while Apache Spark is the most common Big Data framework for cluster computing, deployed with Java.

2.1. Bridges from Java to native code

The launch of the Java programming language and its fast increase in popularity have instantly created a paradigm gap between legacy codebase and applications that were newly-developed under Java. Since the industry wanted to leverage already existing code for various reasons (performance, cost-efficiency, etc.), the need of a Java-to-native bridge became natural.

In this thesis, we are going to consider and evaluate this hybrid approach as a Java-managed application with:

- the high-level structure and implementation done in Java,
- managed by the Java Virtual Machine,
- with different extents of native sections, encapsulated and handled by Java code.

A simple overview of a hybrid Java-C/C++ application flow is highlighted in Figure 2.1. A JNI-enabled Java application running in its master thread, hands control to the native interface at the call site, stalls until the native thread returns and then continues execution.

With the launch of the Java programming language, there has been immediate pressure on all major vendors to offer means of bridging the gap created to the native code. At the beginning, the solution offered with JDK 1.0, the native method interface did not offer lenient access to C, and the competitors started building their own solutions, some of the most important being Netscape’s Java Runtime Interface, Microsoft’s Raw Native Interface and the Java/COM interface etc.
However, none of the above-mentioned could deliver in all areas, but they proved to be good case-studies and building blocks for Sun JVM’s JNI, that was shipped with JDK 1.1. Solving most of the critical issues, the Java Native Interface became the standard in building Java-native applications[7].

2.1.1. The Java-Native Interface

The JNI framework is an API shipped with all the major JDK implementations. It allows both integration of native code into Java and the other way around. The integration is done via function calls (of native code from Java main application or viceversa). The targeted use cases of the JNI are:

- Encapsulate an existing C++ functionality into a higher-level Java application, without being forced to rewrite the code.
- Reuse legacy native libraries in modern applications.
- Access platform-dependent features needed by the application that are not available with the standard Java libraries.
- Accelerate portions of code by taking advantage of the lower-level characteristic of native code.

In our implementation we are focused on the Java-to-C++ function calls, since this yields the smallest overhead and allows full control from the Spark executors.

Introduced with the JDK version 1.1, the JNI became the defacto native programming standard for most of the Java community. The overview of the functional flow of a JNI-accelerated Java program is as follows:

- The developer implements the top-level architecture of the target application in Java, leaving out portions that are suitable for integration through JNI: legacy code or user-implemented functionality.
- The user defines native prototypes in Java of the form:

  ```java
  public native return_type nativeFunc(type1 arg1, type2 arg2,..);
  ```
• A command-line tool shipped with the JDK (javah) is used to create JNI function headers in C/C++ from user-defined, compiled Java classes that contain native functions. This tool defines specialized prototypes in native code of the following form:

```
JNICALL JNICALL Java_SomeUserClass_nativeFunc(JNIEnv *, jobject, jobject, jobject,..);
```

Analyzing the generated prototype, it is important to note that, in addition to the arguments passed from Java, the native code receives a reference of JNIEnv – a handler of the caller (JVM) that is used to access any member of the Java implementation; and a reference to the calling object/class (jobject or jclass - if the caller function is static).

• It is the developer’s duty to implement the function in C/C++, by including the automatically generated JNI header file(s).

• The C/C++ code base then needs to be packaged into a native library, by means of native compilation tools (e.g. GCC).

• Last, the developer points the JVM to the previously-packaged library, allowing the application to populate the call-stack around the native function call. It is important to note that, in case of the missing library, there will be no static warning, but a RuntimeException, resulting into an application crash.

A simple implementation of a memory copy application is showcased:

```java
class Bean{
    int[] dataArr;
    // .. Getters and setters ..
}
class Main{
    public static void main(String[] args){
        Bean sourceArray = generateRandArr(256);
        Bean copiedArray = memcopyNative(sourceArray);
        // deep - compare the objects for equality
    }
    public static native Bean memcopyNative(Bean input);
    public static Bean generateRandArr(int arrSize){
        // Generate a random array
        return null; // in case of error
    }
}
```

The java classes are then compiled:

```
javac Bean.java Main.java
```

and we obtain Bean.class and Main.class files. Then,

```
javah Main
```
which generates Main.h JNI header based on the memcopyNative function prototype:

```c
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Main */

#ifndef _Included_Main
#define _Included_Main
#ifdef __cplusplus
extern "C" {
#endif

/* Class: Main
* Method: memcopyNative
* Signature: (LBean;)LBean;
*/
JNIEXPORT jobject JNICALL Java_Main_memcopyNative
(JNIEnv *, jclass , jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Below, a simplified implementation of the native function that performs the array copy is presented. To be noted: the number of statements needed for retrieving the data array from the our previously-defined Bean object. Creation of the returned object is symmetric and as lengthy as the presented one:

File Main.cpp:

```c
JNIEXPORT jobject JNICALL Java_Main_memcopyNative(JNIEnv *env,
jclass callerClass , jobject source){
    jclass beanClass = env->GetObjectClass(source);
    jfieldID dataArrField =
        env->GetFieldID(beanClass, "dataArr", "[I");
    jobject dataArrObj = env->GetObjectField(source, dataArrField);
    jintArray dataArr = reinterpret_cast<jintArray>(doubles_obj);
    jint arrLength = env->GetArrayLength(dataArr);
    int* data = env->GetIntArrayElements(dataArr, NULL);
    /*
    * Data copy section and target object creation..
    */
    return target;
}
```

Last, the presented source files are compiled into a shared library and loaded in the JVM with `System.load(libraryName)`. At runtime, when the execution flow reaches the native Java function, control is handed to the above-implemented C++ function.
2.1. Bridges from Java to native code

2.1.2. The Java-Native Access framework
The JNA is a community-developed API with the purpose of offering an alternative to the JNI. The target of the JNA, primarily mentioned by its developers, is to allow Java developers to access native code by abstracting over the cumbersome implementation details of the JNI.

In contrast to JNI, JNA is mostly developed in Java and it encapsulates a JNI stub meant to realize the actual native calls.

From the developer's point-of-view, the JNA offers access to native calls from within Java, by simple Java interface parametrization.

Authors in [8] have concluded that, on average, native calls to native libraries perform up to 5 times worse when using JNA in comparison with native calls through the JNI. One of the considerable sources of slowdown in JNA is the dynamic characteristic of method and parameter bindings: these operations are performed at runtime.

However, significant performance improvements can be seen if the methods are directly mapped. This procedure is enabled when the methods of a class are marked with the "native" modifier. In this case, the performance can be similar to the JNI, since the invocation path is the same. However, since every reference type on the Java call stack requires additional steps for conversion for the native call, the performance is greatly influenced by the number of non-primitive arguments that are passed.

A simple example [9] of a JNA implementation, that prints a "Hello, World" to the console, is outlined below:

```java
public class HelloWorld {
    public interface CLibrary extends Library {
        CLibrary INSTANCE = (CLibrary)
            Native.loadLibrary((Platform.isWindows() ? "msvcrt" : "c"), CLibrary.class);

        void printf(String format, Object... args);
    }

    public static void main(String[] args) {
        CLibrary.INSTANCE.printf("Hello, World
n");
    }
}
```

By analyzing the above snippet, we observe that calling a native function with JNA is far simpler than with the JNI. The procedure follows:

- Load the jna.jar library in the application's classpath.
- Inherit the Library abstract class from the jna package, by creating own interface.
- Provide the function candidate prototype that should be bound to the native library function by type and name matching.
2.1.3. Discussions and performance evaluation

The development of a mixed Java-C/C++ involves working with one of the alternatives presented earlier, or with derivatives of them. The choice should satisfy programming requirements and flexibility, and, if any, performance constraints. In the remainder of this section, we will compare the two variants, while also emphasising their case-to-case relation to Java.

Researchers in [10] have realized a comparison between pure Java and JNI implementations, based on a couple of benchmarked micro-kernels. The purpose was to identify characteristics of the approaches that make them suitable for accelerating specific parts of code/algorithms.

Their findings show that, for memory-bound algorithms as well as for micro-kernels with data input smaller than the cache size, the performance is roughly the same, if not degrading when using JNI. These cases are not justifiable for native code acceleration, at least on the CPU. Compute intensive kernels, on the other hand, are significantly accelerated by native implementations, in comparison with their Java counterparts (consistent speedup up to 7x).

The debate on ease-of-implementation vs performance has its roots in the "Java vs C" discussion. Authors in [11] also identify interesting scenarios where Java is indeed faster than native (C) implementations. Their conclusion is more clear, as they do not only focus on raw computational kernels, but also on more complex data structures. The Java Virtual Machine can outperform the GNU C compiler in areas where there is no memory allocation and for complex enough computational problems (matrix multiply, FFT and LFSR). The result is not surprising, as the novel JIT (just-in-time compilation) feature of the JVM can optimize the bytecode better than the statically compiled C code. In case of iterative algorithms which allocate memory more aggressively (e.g. Red-Black tree sorting algorithm), Java's garbage collector will increase the execution time with approximatively 1 to 15 %.

Due to the constant evolution of the managed Java language and it's JVM, we can expect the performance gap between native code and Java to decrease asymptotically. The work in [10] and [11] shows that the JVM can already outperform native compilers in certain scenarios. However, we can conclude that in higher complexity algorithms which involve complex branching and explicit memory management, native implementations that are closer to machine language will top higher-level languages.

Although JNA and JNI target the use-case of allowing native integration into Java, their detailed purposes vary. A simple comparison between the two frameworks is outlined in Table 2.1.

We observe that a simple micro-kernel like the one presented in Section 2.1.1 grows very much in implementation size and complexity, if accelerated by JNI. Aside from the development effort, JNI applications are more complicated due to difficult native code debugging and to the more fault-prone characteristic of C/C++ code. Usually, errors that are not seen at compile time in native code lead to segmentation faults and core dumps, leading to very time-consuming inspections.

In contrast, JNA provides the alternative in terms of implementation effort: coding is faster, requires less C/C++ knowledge and offers closely similar advantages. In terms of performance, JNI allows lower-level code-tuning, enabling faster execution [8], mostly by reducing the latency in function and parameter mapping.

None of the approaches limits the compatibility spectrum when considering legacy code integration or user-developed applications.
2.2. OpenCL

The Open Computing Language framework is an open-source parallel programming standard targeted at cross-platform compatibility. At the time of writing, almost all major vendors support OpenCL deployment on their products, from all the general processor categories, such as CPUs, GPUs, FPGAs and DSPs.

Initially targeting C applications, the latest release (version 2.2) allows programming in the C++14 standard as well. En route to becoming the standard for parallel computing, OpenCL has abstracted over the most challenging task of parallel programming: a code tree can be executed with almost no changes on any OpenCL-enabled device. Furthermore, in heterogeneous environments, the work can be dynamically balanced across all supported processors.

Conceptually, the programming model is almost identical to CUDA (an NVIDIA proprietary API targeted at GPUs) and comprises characteristics of the task-parallel, bulk synchronous and data-driven models.

In contrast to higher-level parallel frameworks, OpenCL contains well-encapsulated parallel code, defined as kernels. Being initially meant for deployment on GPU platforms, in comparison to OpenMP, OpenCL is a more lower-level framework. This translates to increased development time with the benefits of better performance and larger platform compatibility, whereas OpenMP can only run parallelized code on CPUs.

A simple overview of the control flow of an OpenCL-accelerated application:

1. Launch C application.
2. Parameterize kernel and launch it on the device.
3. Wait or transfer data back to main memory after kernel execution has finished.

The current trend targets unified memory architectures (UMA) or virtually unified memory architectures: the developer cannot distinguish between the devices’ address spaces.
This trend is felt in both hardware and software deployments and frameworks are integrating the concept in their implementation with the aim of making the data copy mechanisms transparent to the developer.

Originally targeted at GPUs, expressing high-efficiency in resource utilization was a non-trivial task for the developer. However, recent implementation changes and evolution of underlying hardware reduced the knowledge requirements and the development effort, helping for a faster convergence towards seamless interoperability in heterogeneous architectures.

2.3. Apache Spark

Currently the Big Data development standard, Apache Spark is a Big Data programming framework developed for allowing fault-tolerant, resilient and distributed computing. Although it is not the first of its kind, the framework is vastly gaining market share because it enables the acceleration of processing pipelines with the use of in-memory computing, in contrast with Hadoop which uses disk storage for inter-task results serialization.

The programming, data and control-flow paradigms are a mixture of the following legacy paradigms:

- Fork-join, from a programmatic point-of-view, as the execution flow is a sequential chaining of data-parallel transformations.

- Bulk-synchronous parallel (at transformation level), where the following characteristic steps can be identified:
  - Execute algorithm in parallel on data partitions.
  - Communicate – arrange data (reshuffle).
  - Barrier – synchronize all the concurrent operations in a given transformation.
  - Communicate the data back to the driver (gather) or reshuffle into a new distributed dataset.

The framework’s main building element is the RDD (Resilient Distributed Dataset), an immutable generic partitioned data wrapper built in Java that enables all the major benefits of Spark.

There are two general types of operations allowed on an RDD:

- Actions: allow retrieval and storage of data contained in RDD’s. Basically they return data to the user on the Spark Driver.

- Transformations of RDDs in new RDDs (e.g. map, filter, join, etc). Each transformation needs personalization with user-defined behavior (i.e. personalized by a function applied on elements of the collection stored in the RDD).

The most common transformations are presented in Figure 2.2. The functional flow characteristic of the map (Figure 2.2a) and filter (Figure 2.2c) transformations resemble closely to the SIMD paradigm: a unique operation is applied on all elements of the RDD and processing of individual elements is independent of others. This opens the way for easier acceleration on massively-parallel processors.
2.3. Apache Spark

Figure 2.2: Apache Spark most used transformations

The reduce transformation involves shuffling of data within the RDD, involving stronger dependencies between the elements.

A simplified control flow representation is given in Figure 2.3.

Distributed storage

Transformation (e.g. Map)

RDD

Transformation (e.g. Filter)

RDD

Distributed storage

Figure 2.3: Spark processing flow overview

For evaluating a simple application, let’s consider the example taken from the official website[12]:

```scala
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Now, by analogy, we will identify the visual steps presented in Figure 2.3, with the statements in the above code snippet:

1. First an RDD is created by a Spark-specific action, `textFile`, which creates an RDD<String> (collection elements are lines from a given textfile).

2. The lines are splitted by white spaces creating a two-layered distributed collection and the `flatMap` flattens the result in a new RDD<String>. 
3. Each word gets assigned a 1-weight.

4. The last transformation is a reduce: in parenthesis, the function specifies that for elements with colliding keys, output a new element with the summed weight.

5. Finally, another Spark-specific action is meant to store the results to disk.

Since the Spark framework is of high importance in our research and implementation, we will dedicate a significant portion for its architectural details in Chapter 4.
Related Work

In this section, we are going to review some implementations and directions that resemble our research ideas closely. Conceptually, we are interested in academic and cutting-edge work that targets acceleration of the Apache Spark framework by means of heterogeneous computing platforms. Some implementations are interesting even though they target pure Java, since the integration of Java APIs is simple with Spark.

3.1. OpenCL bindings for Java

3.1.1. Aparapi

The framework[13], internally developed by AMD and published for open-source development is meant to allow developers to write target-agnostic code, that can run on heterogeneous architectures. While the initial destination of Aparapi was to run code on AMD APU (accelerated processing unit) devices and dedicated graphics cards, it has been ported on accelerated devices from other vendors [14] and also on FPGAs and DSPs.

Aparapi is a materialization of the HSA (Heterogeneous System Architecture), a design and functional AMD concept that "provides a unified view of fundamental computing elements" [15]. The target of Aparapi was originally to leverage the use of the new HSA. In this sense, the new concept would allow developers to abstract over computing unit-specific implementations, by providing a single unified programming platform. A recurrent theme with the industry, HSA also defines a unified virtual memory space (as targeted as well by IBM's CAPI [16] at a hardware level and OpenCL on the software level), meant to alleviate programming effort, by abstracting over the inter-device data transfer intrinsics.

With the aim of supporting Java's "write once, run everywhere" motto, Aparapi is built on top of OpenCL, the previously-mentioned portable parallel programming framework. Acceleration is enabled in Java by data-parallel portions of code, defined here as kernels.

The user-defined Java kernels are accelerated on OpenCL-enabled devices by a bytecode-to-OpenCL translation engine provided with Aparapi: the user writes data-parallel kernels in Java, which are then translated to normal OpenCL kernels. It is important to note that code translation works only at a very simple layer, managing the fundamental semantics of Java that is similar to C. For this reason, the focus is on primitive types and primitive arrays,
whereas complex data types are not supported. Last, a JNI wrapper bridges the function calls and data transfer between the Java application and the OpenCL kernels.

AMD’s framework lays a solid foundation for transparent parallel execution that is target-agnostic. For these reasons, it has been used and referenced later for deployment of more targeted frameworks.

Consistent development of Aparapi (according to its public code repository commit chart) has been halted or finished at the end of 2015. At the time of writing, it is unclear whether AMD’s developers are switching to a re-branded and enhanced version of Aparapi or if they are dropping the initiative altogether.

3.1.2. Java-OpenCL bindings

Soon enough after OpenCL’s emergence as a competitive parallel computing framework, the scientific community has taken note of the crucial similarity between the two platforms: portability. Naturally, there have appeared bindings between the two paradigms (generically named JoCL or JavaCL), some of them addressing mot-a-mot access from Java, while others were targeting enriching OpenCL with Object-Oriented features.

Because there is no definitive winner of the bunch, we will outline their characteristics briefly:

- **Java OpenCL**[17] is full-platform-compatible with OpenCL 1.1. and work-in-progress shows portability to the OpenCL 2.0 standard. The bindings are both higher- and lower-level, abstracting over some of the intricacies of OpenCL, by providing easier-to-use Java handlers and object-oriented encapsulation.

- **JavaCL** is an abstraction tier built over OpenCL4Java which offers lower-level bindings to OpenCL code. Full compatibility is targeted at OpenCL 1.2, while the last version [18] is still a release candidate.

- **JOCL**[19] the most natural flavor of OpenCL’s translation to Java, represents a mot-a-mot implementation of the OpenCL C API. The developer needs to be aware of all intricacies of OpenCL, since there is no beautify-layer in-between, with the exception of the data passing and conversion, done via Java’s java.nio byte buffers. The library has different versions for different platforms.

The authors of the mentioned frameworks claim close-to-OpenCL performance, while we expect that JOCL[19] to offer the most unconstrained performance due to its lower-level characteristic and JNI-enabled acceleration (whereas JavaCL uses JNA and Java OpenCL features object-oriented abstractions).
3.2. Performance-enriching frameworks for Spark

Although the majority of Java APIs should provide good compatibility with Spark, when involving native implementations or specialized processors, a Java developer can be overwhelmed by the programming experience:

- Spark is Java-based, offering full portability, whereas native code is not. The developer needs to manually code and compile for a specific architecture.

- If the implementation targets a cluster-wide infrastructure, an additional layer should take care for automatic deployment of native code/specialized accelerators.

In this direction, we will have an overview of Spark-enriching research that should alleviate some of the challenges.

3.2.1. Project Tungsten

Starting with the release of Spark 1.6 in 2015, the authors are targeting the increase in performance of the Spark framework, by bringing the execution and programming frameworks closer to the *bare metal* abstraction[20].

The mentioned project, developed by Databricks (company who currently maintains Spark) aims to extend Spark with the following performance-enriching features:

- Reduce the impact of the JVM object model and garbage collection by means of explicit memory management and binary processing. This would also imply a proper adaptation of the Spark-specific semantics.

- Accelerate processing by implementing cache-aware computation. The algorithms and data structures would be designed for taking advantage of the memory hierarchy details in computing systems.

- Optimize executable code by exploiting modern compilers and CPUs through code generation.

- Include GPU-specific and SIMD instructions for accelerating massively parallel mathematical computations.

At the time of writing, Spark 2.0 comprises implementations featured in phase 2 of Project Tungsten. While the concrete details regarding specific improvements are scarce, the developers market runtime code optimization and native bridges for commonly-used higher level libraries[21]. Currently, there is also ongoing work in allowing off-heap memory usage for C++ applications.

Utopically, the presented directions of the Tungsten project would encompass many of the necessary optimizations and enhancements needed for a heterogeneous distributed system deployed with Apache Spark. A key fact that is missing from Codebricks’ Tungsten project overview is the means of accelerating Spark with reconfigurable hardware (FPGAs) or specialized processors (DSPs).
3.2.2. SparkCL
Built on top of Aparapi, the framework allows execution of Spark transformations in OpenCL: instead of applying Spark transformations on parallel data, the framework leverages the use of native-accelerated code for processing the input. The data-parallel portions of native code are supported by OpenCL kernels that are automatically generated from user-defined Java code. The generated OpenCL kernels are then accelerated on heterogeneous hardware that support OpenCL compilation. Since this solution resembles the most to our own initiative, a thorough analysis of the authors’ paper[22] and code base will help us understand better our directions, in terms of conceptual and implementation details:

- The design chosen by the authors enriches the data flow and structure of the original Spark framework, by defining specific constructs that mimic the functionality of the original implementation (e.g. creating specific RDDs and transformations through inheritance, adding functions to map data from Spark to Aparapi-specific kernels etc). While this could make integration more lenient, there are downsides in terms of ease-of-programmability. Since the users should be able to seamlessly integrate Java and accelerated code (from Spark’s point-of-view), the implementation and coding styles should remain as much as possible untouched.

- The data model employed by SparkCL relies solely on raw arrays and does not allow fine-grained user personalization. In reality, complex applications contain complex data structures that are not all well suited for the framework. In the case of kernels which need multiple parameters and arrays, the programming task evolves in complexity for the Java-side and can also lead to additional overhead (data serialization done for passing to the Spark kernel - possibly very costly operation). Moreover, the paper does not show ways for using complex parameter types (reference types). The developer, most probably an OOP programmer should be able to use encapsulation for containing the entire information spectrum in atomic elements (beans).

- There are stability issues that reside both in the framework’s implementation and also in the combination with needed OpenCL or Aparapi libraries. Moreover, full code translation in the scope defined by the authors is not possible as well (e.g. double-precision floating point operations and data types are not supported).

- As implemented, changes into the Spark framework can lead to compatibility issues later as SparkCL has deeper roots into the Spark codebase. With future developments in the Spark framework, SparkCL can become version dependent and, if not maintained, obsolete.

- Since the framework leverages Aparapi’s OpenCL code generation engine, developers are enabled to write kernel-like transformations in Spark, relying on SparkCL for device-specific acceleration details. In regard to this, the debate in the scientific community argues that code generation is suboptimal even in the less-constrained environments. In java, there are not too many commercially available options to translate functionality to C/C++. The code translation feature of the Aparapi framework does a simple C-level translation of Java, providing functionality mostly with primitive data types and arrays. From here on, cumbersome design choices are needed in order to seamlessly enable SparkCL into mainstream applications without significant changes in already coded functionality.
Considering the above-mentioned arguments, for such a direction to be feasible in the short term, we need to debate the following questions:

1. Do we need/can we have a good enough compiler for Java-to-C/C++ so we can compete in performance with manually-developed native code?

2. Are OpenCL-generated HDL bit streams good enough to satisfy requirements and alleviate the need for experienced HDL programmers?

In regard to the above-mentioned concerns, although the field is showing consistent improvements in hardware Just-In-Time compilation [23] and better compilers (Altera SDK), we consider that current developments are not mature enough for the moment. Moreover, the differences in performance between manual implementation and automatic code generation are not justifiable in the high performance computing or Big Data fields. If the above questions would have competitive solutions, other areas would allow trade-offs (e.g. specialized Spark distributions) for taking benefit of the reduced development effort.

While the authors set a cornerstone on the path to enabling OpenCL-accelerated Spark, the design constraints imposed raise doubts about ease-of-programmability and performance. Last, the ties to specific implementations of external APIs (Spark, OpenCL and Aparapi) contribute to the variety of debatable design choices and darken the future of this framework.

### 3.2.3. HeteroSpark

The authors of HeteroSpark[24] are targeting acceleration of Spark with the help of GPUs. The execution architecture can be described as composed of master-slave couples:

- The Spark environment (on the executor nodes).
- The accelerated environment (GPU side) with its dedicated JVM.

Information transfer between the two environments is done via the Java RMI (Remote Method Invocation). By using the RMI as bridge between the two environments, the architectural constraints are softened: an executor node can either run accelerated GPU code locally (on the same node), remotely – through the network or not at all. A cluster-level configuration file is needed to set the execution mode of each Spark executor.

Information needed to be processed is serialized on the Spark executor JVM, transferred via the RMI and deserialized on the "GPU" JVM. The latter calls a JNI-accessible native kernel for processing the received data, by means of a precompiled native library. The computed results follow the same path back to the Spark environment.

The framework can be deployed in two modes:

- **Black-box** mode (from Spark's POV): only the remote RMI socket is provided, assuming that on the other side (the remote node that runs the GPU-JVM) the deployment has been done separately and accepting connections from an executor.

- **Dev** mode – the developer writes the wrapper for the GPU kernel.

In both deployment modes, it is assumed that the GPU kernel is provided, either as source code or as a dynamic library (*.so).
HeteroSpark has a strong advantage when coming across cluster customization. However, in terms of design, there are a couple of debatable research points that do not convince.

First, by using two instances of the Java Virtual Machine for a single executor node, the development task can be parallellized itself, the only constraint being the data transfer interface which needs agreement from both development sides. The issue that arises with this separation is related to the data transfer channel, the Java RMI. The use of this framework enforces a round trip in terms of data serialization, which automatically includes memory copies. We expect, that for big invocation sizes, the RMI-induced overhead to become significant.

Second, although we agree with the idea of freedom in terms of customization, the possibility of allocating a dedicated GPU-only node for network-remote access would not appear efficient in any circumstance: even the lowest-end graphics cards would be severely bottlenecked by transfer latency and bandwidth, in comparison with the standardized GPU bus, PCI Express 3.0.

As pointed in the paper, the framework was designed and targeted at heterogeneous Spark acceleration. Although the architectural design would allow easier integration with other accelerated units (e.g. FPGAs), there is no hint that a deployment different to GPUs has been investigated.

In terms of performance, the authors claim up to 18x speedup when using their GPU-accelerated framework, in comparison with CPU-only Spark. However, the code base is not pointed to in the research, and we could not validate their findings. As of August 2016, the framework appears to be unmaintained.
3.3. Discussions and evaluation

For evaluating the presented frameworks, we will try to quantify their properties in Table 3.1, by comparing both Spark-enabled and the Java-targeted frameworks. The Tungsten Project is also included in the comparison, for reference purposes, even though development is in progress and not all mentioned features are currently integrated in the Spark release.

Conceptually, the Tungsten project would be the most suited for enabling a high performance Spark implementation, from the evaluated batch. Even though the consistent acceleration details are only provided on paper, the concept sets a good direction for others to follow, and, as of Spark 2.0, development is picking up pace, with phase 2 of the Tungsten Project. The only caveat is the lack of mentions regarding support for more specialized native accelerators (e.g. DSPs or FPGAs).

Table 3.1: Comparison between heterogeneous-enabling Java frameworks

<table>
<thead>
<tr>
<th>Framework</th>
<th>Performance</th>
<th>Ease-of-deployment</th>
<th>Compatibility &amp; Portability</th>
<th>Active Development</th>
<th>Integrated with Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aparapi [13]</td>
<td>++</td>
<td>+++</td>
<td>++</td>
<td>No, stable, unmaintained</td>
<td>No</td>
</tr>
<tr>
<td>SparkCL [22]</td>
<td>++</td>
<td>+++</td>
<td>+</td>
<td>Research phase, public repo</td>
<td>Yes</td>
</tr>
<tr>
<td>JoCL [19]</td>
<td>+++</td>
<td>+</td>
<td>++</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>HeteroSpark [24]</td>
<td>+++</td>
<td>+</td>
<td>++²</td>
<td>Unknown, research phase</td>
<td>Yes</td>
</tr>
<tr>
<td>Spark Tungsten³</td>
<td>++</td>
<td>+++</td>
<td>+++</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Our concept</td>
<td>+++</td>
<td>++</td>
<td>+++</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

From the evaluated set of qualities, we are interested the most in the performance aspect. Here, the most unconstrained implementations are the highest performing: JoCL should post the best results, being a Java representation of OpenCL, while the researchers of HeteroSpark [24] also claim high performance for their framework. Researchers in [25] show that Aparapi imposes significant overheads in performance (up to 2x slowdown), when comparing to pure multi-threaded Java implementations. The research on SparkCL does not give reference benchmarks and our findings have shown that their framework is

---

1 Since the mentioned frameworks have different target use-cases and divergent implementation directions, the performance comparison has been evaluated empirically and is the personal opinion of the author.
2 The assessed compatibility and portability degrees of HeteroSpark are reliant only on impressions obtained from the author’s article.
3 The Spark Tungsten project is currently in the development phase, and has been included for comparison purposes.
marginally slower than Aparapi, on a case-to-case comparison (up to two times slower, depending on system and software configuration).

From the suite of OpenCL bindings, Aparapi and JoCL have been listed for their proven historic and good compatibility with Java, as well as community-driven or consistent development efforts. Since Spark is mainly written in Java and Scala, these libraries’ integration with Spark should not prove an insurmountable development task. However, none of them are complete solutions, since Aparapi is targeted at AMD products and JoCL, while being target-agnostic, is cumbersome in terms of implementation effort.

While SparkCL and HeteroSpark offer concrete Spark integration in their design, there are some weak points such as lack of full native support and no publicly available codebase (HeteroSpark) or external dependencies and restricted compatibility with Spark (SparkCL).

The best examples as ease-of-implementation are clearly Aparapi, and the Aparapi-based SparkCL. Ideally, full native code generation would reduce the development effort considerably, while requiring less programming and hardware-specific knowledge. However, as previously-mentioned, the solution is not complete and market developments do not offer a better alternative.

Finally, the last row of Table 3.1 represents our conceptual evaluation of how a heterogeneous Spark framework should be designed. Based on our research, a brief list of important remarks for a JNI-accelerated Spark framework is presented:

- Since the entire Big-Data field is community-driven, there should be no steep learning curve when using the solution.

- Performance should be kept as much as possible unhindered, both in computation throughput and memory access, regardless of the target hardware.

- Compatibility should be assured with future Spark versions, by avoiding deep Spark integration.

- In the absence of smarter code translation engines, code generation should be used restrictively, only to reduce the native development effort.

The previous observations represent our qualitative assertions on the related work and are meant to better understand the design choices that will be faced when conceptualising a heterogeneous Big Data Spark framework. After the proposed framework implementation has been done based on these assertions, the resulting framework will be evaluated quantitatively in Section 5.3 against its most similar counterpart: SparkCL, which is the only Spark-enabled framework with publicly available source code.

In the upcoming chapter, we will present a design that closely matches our findings and conclusions.
In this chapter, we will aim at quantifying the results of the analysis on the available research done in the field of heterogeneous Spark frameworks (Chapter 3). First, a more in-depth research will be done into the Spark architecture. Next, a reference design is sketched, implemented and tested with three micro-benchmarks (in the next chapter). In the last section of this chapter, the design choices and implementation details of the SparkJNI framework are presented.

4.1. Design Considerations

As mentioned in the previous chapter, the Spark community and Codebricks have started to integrate HPC features and plugins into the Spark distribution (e.g. Spark Machine Learning library). However, these efforts target particular applications and do not offer means of personalization for general application requirements: the design space is limited and contained within the space provided by the accelerated libraries (e.g. MLlib targets only machine learning applications).

For the presented reasons, the scientific community has started to search for optimal starting points for this integration. The consensus is currently as follows:

- Use Spark as the cluster deployment, management and high-level programming environment.
- Accelerate specific portions of Big Data code with the help of native or device-specific library calls.

As seen in works by [14], [22] and [24], all of the implementations use the JNI interface to enable the Spark framework with native libraries and specialized accelerators. There is however no consensus in architectural details, as each author proposes different approaches and solutions with variable exposure to the developer of the native interface (i.e. the programmer has more or less control over the implementation of the native code).

Research done in the most interesting framework to ours, SparkCL, the authors prefer full abstraction over the native implementation details, leaving the developer with the job
of writing OpenCL-like kernels in Java (in the Spark environment). As discussed in Section 3.2.2, the solution does not solve the major issues and imposes further constraints on ease-of-implementation and performance.

Our research has found common ground for the design considerations detailed in the related research, and based on the evaluation of the authors’ design choices, a candidate for a heterogeneous Spark reference design is showcased in Section 4.3.

4.2. Spark architecture

A reference design for a JNI-optimized Spark framework should first be conceived as a conceptual evaluation of requirements and existing possibilities. For this reason, we consider the current Spark architecture to be a crucial building block in any Spark-native application.

For the mentioned reasons, in this section, after the overview presented in Chapter 2, we will enrich the information background of the Apache Spark framework, by focusing on the components that will be exposed in and by our design.

4.2.1. Data model

As mentioned in [26], the RDD (Resilient Distributed Dataset) is the main layer of data abstraction recognized in Spark. This model represents a read-only (immutable), partitioned collection of data elements (i.e. the RDD uses generics for collection personalization) stored locally or distributed across Spark worker nodes. The RDDs can be created either from other RDDs (through transformations), or by specific operations on stored data (either from Hadoop, HDD, Amazon S3 etc.)

There are two general types of operations that can be applied on RDDs:

- **Transformations** are coarse-grained updates to the shared state of the RDD’s elements. These updates are represented by different functions: map(), reduce(), filter(), join() etc, which compute a new RDD from the base RDD, by applying the same operation in parallel too many data items [2]. These transformations are chained in a DAG (directed acyclic graph) and they are evaluated in stages (each stage can contain one more pipelined transformations). The transformations are *lazy evaluated*: they are triggered when needed by their direct successor in the DAG. The result of logging the transformations needed to build a dataset from a base dataset is called lineage. This logging concept of the coarse-grained transformations lessens the implementation effort of a fault-tolerant mechanism, as it allows recovering failed stages of a DAG by just repeating the failed operations, not the entire application pipeline.

- **Actions** are Spark operations that return a result to the Spark driver (application), by triggering computation of the unprocessed segment of the application DAG. Examples of actions include `first` (get the first element of an RDD), `collect` (get all elements of an RDD into a Java List), `count` (return the number of elements in the RDD). There are also actions which output data to the storage system (e.g. `save`)[2].

As mentioned, the greatest advantage of Spark comes from its data-management engine. Before, the legacy standard in Big Data, the MapReduce framework, employed data serialization to storage, even during intermediate application pipeline stages. Hence, for
each transformation, data has to be retrieved and stored to permanent storage and redundancy needs to be rebuild across an entire cluster.

The mentioned operations are on the critical path of every MapReduce execution flow and the result are, as expected, high access and storage time, increased bandwidth and IO/s requirements for disks and asymptotically increased network congestion that heavily constrain cluster performance. In contrast, Spark’s in-memory computing feature alleviates some if not all of the mentioned performance bottlenecks, by allowing intermediate results to be stored in application memory [27].

There are multiple APIs that offer access to distributed data. While HDFS [5] is the most deployed for storing raw files, there are also SQL (Hive [28]), columnar (Parquet [29]) and other storage solutions for distributed datasets. Practically, almost all distributed storage formats in the Hadoop ecosystem are also available to Spark. The storage options are of significant importance, since most Spark applications start and end with actions that involve disk access.

In Figure 4.1, a simple overview of the Spark infrastructure is presented. The Spark driver is the processing node which launches the Spark application. In turn, each worker node announces its availability to the Spark driver through a management layer. Once a worker is available, it can be assigned work (transformations), in form of RDD partitions based on resources and data locality. After all the parallel jobs are executed on the worker nodes, the results are either returned to the Spark driver or serialized to storage, through an action.

### 4.2.2. Execution model

In traditional HPC clustered systems, the Message Passing Interface is the defacto standard for running large-scale jobs. For employing this mechanism, developers are involved in complex network programming. Although the paradigm has been successfully deployed for decades, it starts to show limitations in comparison to modern field-related frameworks:

- Problem and data partitioning becomes increasingly difficult and scaling sometimes concerns a multitude of factors.

- Failures and straggler nodes (slower nodes) have to be manually dealt with programmatically by means of complex schedulers and fallback mechanisms.
In contrast, Spark, as well as its archetype, MapReduce [30], employs a data flow model as its architectural characteristic. In this case, the programming interface is restricted in order to increase the scheduler’s transparency to the developer.

In more detail, applications are structured in coarse-grained jobs, which, in turn, are constructed from transformations. These building blocks allow the conception of graphs of high-level operators (DAG), which can be handled more easily by schedulers, allowing for better allocation of processing tasks and data partitioning to available resources. Moreover, due to the lineage characteristic of the DAG, fault recovery is automatically handled with insignificant overheads [2].

By sacrificing programmability freedom in terms of code specialization (by restricting it to higher-level programming languages and coarse-grained tasks), Spark offers ease-of-programmability, better scalability and a more active development community in comparison to its aforementioned HPC counterpart.

4.3. A JNI-accelerated Spark design

The foundation of a well-defined framework that intersects the Spark and high-performance applications is represented by valid mappings between the elements that need to be welded together. In our case, we want to be able to create a seamless transition between the Spark environment and native code, in terms of data and control interfaces.

As mentioned in Section 2.1.1, we need to assure Spark compatibility for a longer timeframe. Therefore, we need to design a framework that can use the Spark distribution JAR as it is. In this way, we can avoid cumbersome compilation, integration and deployment issues.

Next, we want to be able to guide developers towards efficient implementation, by means of a parametrizable and templated framework design: the building blocks of any Spark JNI-native application should be provided and personalization should be done by the end-user. The guidelines will be, therefore, enforced by abstract classes and interfaces. However, the deliverable should be implementation-agnostic with the help of Java Generics and the Java Reflection API: development, in terms of creativity, should be left unhindered, in comparison to the original specification.

On the native side, we have to give full freedom to the developer, in terms of implementation details. Although solutions described in [22] and [13] offer possibilities of full code generation from Java kernels to OpenCL kernels, as discussed, some of the drawbacks that come with this feature are not suitable for production-level applications. Moreover, we want to allow users to use the framework even with newer versions of native libraries.

While most of the high-performance applications are traditionally coded in C, the overall trend suggests a sensitive separation with a more stronger shift towards OOP and functional programming. Apache Spark includes both aspects, allowing developers to use Java, Scala and more API-restricted versions of Python and R.

Since C++ offers compatibility with legacy C code and libraries (i.e. C++ can incorporate C code) and because of the previously-mentioned OOP constraints in the Spark environment, we will define our design as well based on OOP (in C++), with a stronger focus on data encapsulation.

An overview of the OOP-based abstraction in the JNI-Spark design is represented in Figure 4.2. The Bean class representation is a general specification of data containers in our design, derived from the JavaBeans concept used mostly within Java EE (Enterprise
4.3. A JNI-accelerated Spark design

Figure 4.2: The Spark-JNI data representation concept

Edition) field. In our case, as well, Bean classes have to conform to a set of design rules:

- Expose a zero-argument constructor (mandatory) and other user-defined constructors.

- Implement java.io.Serializable for allowing encapsulation in RDDs.

- Define fields, and, optionally, getter and setter methods: methods that retrieve or assign a value to/from a field.

The overarching statement in Figure 4.2 is the symmetry of the Java and C++ class representations in terms of data abstraction: the Bean-derived user class should have a similar representation in C++, at least for the core components that are exposed both to the Java and the C++ environment. Therefore, JavaType, CppType and their defined objects have the same fundamental data representation.

This representation convention allows for a standardized JNI data link and favours ease-of-programmability, by means of a simplified application architecture overview.

In Figure 4.3, we are highlighting a simplified application data flow at the Spark functional level. The encapsulated data elements are represented on their path from the Java dataset (the List of Bean objects on the Spark driver), through the RDD encapsulation layer and, then, to the native method invocation layer.

At the intermediate layer, the RDD keeps the data partitioned (split in sub-arrays), each partition being sent to a Spark executor for sequential processing. The dashed double-lines in Figure 4.3 represent Spark operations:

- The Spark parallelize operation converts a standard Java list to a distributed dataset of the original elements. As mentioned before, there are multiple ways of creating RDDs from storage (or from other RDDs), here we exemplify the simplest one: creation of an RDD based on a user-provided Java collection.
• The second Spark operation is a transformation. In this case, a map or reduce transformations can be pictured, showing no pairwise element dependencies. These transformations invoke kernel processing on partitions of data elements, on Spark executors.

In our JNI-enabled model, a Spark executor is accelerated by JNI code through native calls from the JVM. For a true heterogeneous architecture, the target execution platform should be parametrizable for each individual node, where a node can deploy one or more executors.

4.4. Implementation details

A JNI-accelerated Spark application can grow significantly in complexity, as a developer needs advanced skills for efficiently programming in C/C++ and Java concurrently. Moreover, as shown in Section 2.1.1 (JNI analysis), integrating native calls from Java grows the code size and development efforts considerably, even for the simplest kernels.

In order to maintain Spark's ease-of-programming, we need to be able to make the native development less cumbersome, by abstracting over the JNI implementation. Since the JNI package contains a somehow limited base of interface functions, we will try to create an execution link and data-level translator between Spark Java and C++. In detail, we will be focusing in generating C++ counterparts of the types that parametrize RDDs in Spark. Moreover, the framework should be built as a template that allows for cleaner code both in Java and C++, without constraining the user's development freedom.

As mentioned before, a first step would be to allow primitive types and primitive arrays to be translated and encapsulated transparently in native classes, based on their counterparts definitions in Java. The translation target should also enable access to the raw arrays in the JNI, without additional memory copies.

Specialty high performance algorithms sometimes rely on more complex data structures, that cannot be efficiently expressed in terms of first-class primitive arrays and fields. The most used data structures that are non-efficiently representable by arrays are:

- Different flavors of trees (binary, AVL, DLT) - used in algorithms such as Red-Black ordering, prefix computations, pointer jumping, divide and conquer, etc.

- Graphs - used in symmetry breaking, resource allocation (graph coloring problem), etc.

For the mentioned reasons, a further natural improvement in the code generation engine is the permission of reference types and arrays of references in Bean classes. The developer should be able to define complex data structures in Java, and use them seamlessly in the native environment. However, without imposing too many constrains, the feature needs to be targeted towards efficient use-cases, without overlapping in generality with the field of full language translators.

Additionally to the data-level translator (fields mapping), we need to implement the dynamic translation as well: aside from translation of data encapsulated in the Java beans, we can also generate useful methods and constructors in C++ that would further reduce the developer's interaction with the JNI. These methods would contain dynamic behavior that extracts data from the Java object and store it into its C++ counterpart that is visible to the native programmer.

In regard to the discussed features that would greatly improve the SparkJNI engine, we have to reason about realisation as well. It is widely accepted that, because of design incompatibilities between Java and C++, the task of full code translation between the two Object Oriented languages is deemed almost impossible. A couple of examples that expose mismatches in the two languages' designs:

- The lack of automatic garbage collection in C++, leaving the memory management task to the writer (in our case, the code generation engine), whereas Java's JVM uses a separate thread to automatically handle this process.

- Java Generics are a form of meta-programming, while in C++, template parametrization offers a complete level of polymorphism, making the two concepts inherently incompatible.

- Inheritance rules are different, and family syntactic trees cannot be translated seamlessly from one language to another.

- While C++ is a target-compiled language, Java is portable and uses an intermediate language (Java bytecode) that is either compiled or interpreted at runtime, by the JVM. These differences also bring major variations in performance, as discussed in Section 2.1.3.

- C++ does not offer a reflection library, like the Reflection API provided with Java. For this reason, retrieving runtime information about types in C++ is much more difficult and in-depth translation would need a language parser.
4.4.1. Code generation engine

The mentioned differences and the general consensus of the software engineering community makes full Java-to-C/C++ interoperability only available at the data level (through data streams and pure JNI invocations). However, if the stringent incompatibilities can be avoided by use-case specialization, the goal of code generation can be achieved with little to no compromise.

The following sections build up towards the SparkJNI framework, created in the direction of the above-mentioned guidelines.

The Java Reflection API

The effort in translating Java code to C++ is lessened with the use of the Reflection API, a package of libraries that allows dynamic information retrieval about classes and their members: fields and methods. While regular language parsers perform syntactic analysis of code, the Reflection API is not meant for understanding and decomposing Java programs: simply, it only provides the possibility of learning about definitions and their types.

A simple example of the Reflection API in practice is listed below:

```java
class Bean{
    int [] dataArr;
    Bean(int [] dataArr){
        this.dataArr = dataArr;
    }
}

/*
 * Method which retrieves runtime information about an object.
 * If we pass a previously-defined Bean object, we will get:
 */
static void reflection(Object bean){
    Class beanClass = bean.getClass();
    String className = beanClass.getSimpleName(); /* className = "Bean"
    Field [] fields = beanClass.getDeclaredFields();
    String field = fields[0].getType().getName(); /* field = "int[]"
    Constructor [] constructors = beanClass.getConstructors();
}
```

Due to Java’s *dynamic dispatch*, an object with an ambiguous type definition (the root type in Java - Object) in method reflection, the bean object, can become more resourceful if handled with the Reflection API. In the example, the class name of the passed parameter is retrieved, and also the type of its first field (in our case, we have what we expected: an int[] type).

Based on the mentioned functionality offered with the standard JDK, we can generate C++ interfaces for Java classes that only define type-compatible fields. The feat is achieved by creating type mappers, according to the original JNI specification in [7]. Most primitive types have identical names and memory representation (e.g. int, long, char, float etc.), while others do not and need special handling when mapped:

- C++ does not have a built-in boolean type, hence an integer (int) is used for the correct mapping.
4.4. Implementation details

- Static array declaration is very similar in both languages (if the array size is known at compile time - i.e. a constant):

  ```java
  int[] dataArr = new int[7] (Java) vs (C++) int dataArr[7]
  ```

  The above initialization of an array with a predefined size is the same in Java for both static and dynamic initialization. However, in C++, dynamic initialization needs a different type declaration:

  ```c++
  int* dataArr so we can dynamically-allocate it later with dataArr = new int[7]
  ```

  However, there are cases where an existing mapping of a C++ primitive type in Java does not exist and needs to be explicitly offered to the developer and handled accordingly. One such example is the `unsigned` types in C/C++, which do not have correspondents in Java.

  In order to alleviate and handle certain incompatibilities between the languages, like the one described above, the framework will heavily rely on Java annotations.

Java annotations

Java annotations are elements of meta-programming that also come bundled with the standard JDK 7. Although they can offer features such as code generation (like preprocessor directives in C/C++), their functionality is richer and allows us to label specific parts of code without altering the functionality. Examples where such labeling enriches the development experience, would be:

- An already-implemented Java class needs to have a native counterpart. Since we cannot translate most of the reference fields in Java, the user either has to split the class in two (Spark-enabled and native-enabled) or to tag the elements that should be exposed in C++.

- Our design enforces declaration of Java constructors which populate all the native fields of interest (e.g. for the purpose of object creation). How do we know which constructor to implement in C++?

- In the same case mentioned above with the Java-native constructor: which of the constructor’s arguments needs to be assigned to which native-exposed field?

  Below, an example of a user-defined annotation (as opposed to the Standard Java annotations):

  ```java
  @Retention(RetentionPolicy.RUNTIME) // Standard Java annotation
  @Target(value = ElementType.FIELD) // Standard Java annotation
  public @interface JNI_field {} // Java user-defined annotation
  ```

  We can see that the above code defines the annotation as interface with a “@” symbol in front. In addition to the annotation definition, we can personalize it more with built-in predefined annotations. In our case, we have mentioned:

- The retention policy: RUNTIME. Since we need to dynamically extract information about user-defined code, we have to make the annotation available at runtime, by patching it in the compiled bytecode.
• The target specifies which syntax members can be annotated with this type. In our case, we enforced it on fields, since we will later rely on it for mapping elements from Java to C++. If a developer tries to annotate a class definition like this, the compilation will fail.

Like with all Java classes, if we want to make annotations types public, we need to define them in their own Java file.

Annotations can be made much more complex than this as they can contain members of their own for further personalization. E.g. `@JNI_field(target = "FPGA")` would be a possibility to say that the field should be made available to native code only for FPGA target platforms. Therefore, by using such annotations, the developer can express design choices for native code, while keeping the Spark code-base unchanged.

Let’s consider our simplistic Java bean used for information passing to the C++ code. We will continue using and enriching the `Bean` class example since it represents the core data flow representation for the framework.

```java
class Bean{
    @JNI_field int[] dataArr;
    @JNI_method Bean(int[] dataArr){
        this.dataArr = dataArr;
    }
    //.. Getters and setters../
}
```

In addition to the previous example of the `Bean` class, here we have annotated the field and the constructor. The framework comes with 5 built-in annotations that serve different purposes and should be used by developers accordingly:

• `@JNI_class` indicates which class is a bean container. This is useful to let the framework know for which class (in the user’s classpath) should do the translation and dynamic code generation.

• `@JNI_field` marks a field to be translated and its data made available to native code. This annotation influences how data is initialized and accessed in the native code and it contains three members offered for user personalization:

  – `nativeTypeTarget` allows the Java developer to specify a particular type for the target member field. This is meant for using with user-defined types or types that are incompatible with Java (e.g. unsigned integer).

  – `alignment` specifies if the target field should be initialized in the constructor to a memory aligned location. Values from 8 up to 256 byte alignments are possible.

  – `safe`\(^2\) enables the use of off-heap un-managed memory usage for a particular array. The feature is meant for memory-constrained algorithms which need aligned memory access (e.g. CAPI-enabled accelerated applications).

• `@JNI_method` marks the method to be translated into native code. For the moment, the only possible translation is for constructors that initialize fields in the target class.

\(^2\)At the time of writing, the feature is under development and not tested extensively for stability. It has been included in the immediate implementation roadmap. Unadvised use may lead to program crashes.
4.4. Implementation details

Figure 4.4: The SparkJNI framework functional hierarchy.

- `@JNI_param` is used for labeling parameters in definitions of setter methods or constructors, within user-defined Java code. Usually, setters and constructors are written by developers and allow initialization of parameters. In our case, since we are automatically generating field assignments, we need to know which field the user was intending to initialize with a specific parameter. For this reason, in addition to previous annotations, a parametrization of the annotation field is needed:

```java
@JNI_method Bean (@JNI_param (target = "dataArr") int dataArr) {
    this.dataArr = dataArr;
}
```

- `@JNI_function` marks a class in the classpath that should call a JNI function from within a Spark transformation.

In Figure 4.4, the functional hierarchy of the SparkJNI framework is overviewed in a mixed representation. Solid lines represent composition, dashed line represents inheritance (subtyping), whereas the dotted lines represent representational connection: CppClass objects are generated based on the Bean-derived (user-defined) classes. Colored blocks represent Java classes from the SparkJNI API that are exposed to the developer.

The task of annotation and class runtime type extraction for Java members in the user codebase is accomplished with the use of the aforementioned Reflection API, which is used in the CodeGen engine at each layer in the functional hierarchy.

The code generation flow, defined conceptually as the CodeGen engine, is started by the JniFrameworkLoader (root node in Figure 4.4 which creates CppClass objects for each registered Java bean class. Following, CppClass objects take care of generating their fields, methods and headers, based on the reflection data extracted from their assigned Java bean. Therefore, the CodeGen engine execution flow can be regarded as a depth-first parse of the functional tree described above.
4.4.2. Functional overview

The SparkJNI framework is built as a plug-in for the Spark execution environment and it is meant to generate the JNI-wrappers and helper C++ classes without disrupting the Spark executors. Therefore, the framework is meant to be integrated in the user code, before the actual Spark initialization. A functional overview of an application developed with SparkJNI is sketched in Figure 4.5.

The first block in this flow represents the Spark application codebase implemented by the user. The development process, provided as a guideline for programmers using the provided framework and its API, is outlined in the steps below:

1. The developer implements the application in Spark Java (or Scala), leaving out transformations meant to be accelerated by native implementations. Existing application implementations can be easily adapted as well, by replacing already-implemented transformations.

2. The developer defines the special classes that form the link and data-transport elements in relation to the native code:

   - **JniFunction** is an abstract class which inherits Function interfaces from Spark. It overrides the Function’s `call()` method by redirection to a native function call (which is dynamically invoked). Simply, this class category conceptualizes the behavior that a developer wants to personalize a Spark transformation with. In functional programming, this would be a function; here it is a class which must provide a `call()` method for the Spark engine to invoke. The parent class, JniFunction allows for dynamic-method invocation, based on user defined requirements: if multiple JNI links (native methods) are implemented for the same jni function class, the developer can allow for different behaviors at runtime, by means parametrization with the target method name. This powerful feature allows for executor-specific specialization: if the cluster configuration is heterogeneous, each node can be configured to run the same algorithm, but on different hardware, from the same transformation in the Spark application.

   - **Container** classes provide data encapsulation for information that needs to be exchanged between Java and native code. As well as with the jni functions, containers are Bean-derived classes (they extend the SparkJNI-defined abstract class Bean).

     The Bean abstract class defines behavior for execution time logging and implements the Serializable interface for creating Spark transformation-compatible classes.

     This inheritance does not put additional constraints on the implemented functionality. Therefore, containers can be any user-defined types, personalized with SparkJNI features, as needed. These classes are subjected to translation from Java to C++, based on the above-mentioned rules and user-personalization.

3. Next, it is the developer’s task to identify proper personalization of the previously-defined classes:

   - **JNI functions** need to be annotated accordingly.
• **Container** or **Bean**-derived classes should be analyzed for personalization of the members that need to be exposed to the native side (incorporated into the translated C++ counterpart).

4. *(Optional)* If the previously-defined classes are not in the Java classpath, the developer needs to manually register the above-mentioned class definitions with the framework.

---

Figure 4.5: The SparkJNI CodeGen framework functional flow overview. Dashed line represents user code, whereas solid line represents code generated by the framework.

After the codebase has been adapted for integration with SparkJNI, the second block in Figure 4.5 represents the CodeGen engine. The engine, triggered by the SparkJNI framework, generates JNI-wrappers, translated C++ containers and a kernel template, based on the user-provided codebase.

Last, if the target kernel is implemented (if the file exists), the whole native project is automatically built, the native library loaded on the Spark Driver and Executors and execution proceeds. If not, the Spark application exits, allowing the developer to implement the kernel files.
4.5. Limitations

As we observed in the analysis performed in Chapter 3, a perfect heterogeneous Spark design is hard to achieve. In our case, as opposed to the solution presented in [22], we have decided not to include kernel code generation into the presented framework.

In regard to functionality, the framework only offers support for the *map* Spark transformation: we have directed our efforts into proving that the concept works for the most-used-cases. However, conceptually, we don't expect any implementation difficulties when generalizing the linkers for all possible Spark transformations (e.g. reduce, filter), since only the number of input and the type of the output argument changes.

A decision was taken to avoid time-consuming code parsing and generation at kernel-level. Therefore, the entire generation is done at runtime, before launching the Spark job on the executors. A caveat of this decision was to limit the generalization of the types of the user-defined *JniFunction*: the developer needs to parametrize the function with concrete types (not generics), making the information available at compile-time.

Due to undocumented behavior of a Spark-defined *ByteBuffer* wrapper, usage of memory-aligned and fast-accessible fields is only possible in restricted circumstances. We expect a resolution in further incremental Spark updates, that would allow proper handling of this use case.

Currently, the developer needs to manually turn on and off the code generation and native build feature, leading to a slightly disruptive implementation process.

Originally implemented in Java 8, the CodeGen framework has been ported to Java 7, due to poor support on dedicated clusters and older operating systems. We expect to be able to push the implementation further, with the transition back to Java 8.
Experiments

In this chapter, we will concentrate on evaluating the performance of our hybrid Spark-JNI design. It is assumed from the beginning that not all use cases can be accelerated efficiently by a native implementation. In this sense, the developer needs to make correct assumptions about what to expect when using the JNI and SparkJNI for building Spark applications.

Conceptually, the addition of the Java-native data and control links should impose a quantifiable overhead in execution, in some areas at least. Although there are many situations to take into account, the focus needs to be on use-cases that are naturally unconstrained from other environmental aspects, such as network overhead (inter-node data shuffle), disk access time (parallelizing data for RDD creation) etc.

By evaluating the proposed JNI-accelerated Spark design, we have identified the following metrics as sensitive to our changes:

- **Transformation latency.** Since the native access is only possible in Spark transformations, we have to analyze and find the overhead induced by JNI invocations. Moreover, since the library load in Java is only done for the calling JVM, the per-worker library load duration should be benchmarked as well.

- **Memory transfer.** Ideally, there should be no difference between pure Spark and our design in terms of memory transfer. However, we need to take into consideration that data movement that happens in the native code cannot be handled as efficiently as in the Java environment (by the JVM). Moreover, in the case of arrays, some mechanisms actually do stop the JVM’s garbage collector for optimizing the heap’s structure.

- **Computational throughput.** Naturally, this metric should post better results than Java, even when running plain C/C++ code (without the help of dedicated accelerator hardware). We are expecting that, for an increased computational complexity of the accelerated algorithms, the Spark-JNI hybrid approach should outperform a pure Java implementation.
Additionally, for a quantitative analysis of the implementation's performance against other solutions from the ecosystem, SparkJNI needs to be evaluated against the competing heterogeneous Spark implementation (SparkCL), for revealing potential bottlenecks and advantages/bottlenecks in the design. The evaluation and a brief reflection will be covered in Section 5.3.

Aside from the above-mentioned evaluated metrics, applications deployed with the SparkJNI framework inherently contain another layer of complexity, through code generation and dynamic invocation. The induced overhead, separated in offline (at the deployment phase) and online (at native kernel invocation) segments, will be benchmarked in the last section of this chapter, along with a real-life algorithm implementation and evaluation.

5.1. JNI-accelerated Spark micro-kernels

For the purpose of benchmarking the original Spark framework against the hybrid implementation (with JNI), three main use cases have been defined: a native access micro-kernel, a memory-copy micro-kernel and a 2D convolution micro-kernel. The general setup is presented in Table 5.1.

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster configuration</td>
<td>4 + 1 nodes &amp; 2 sockets per node</td>
</tr>
<tr>
<td>Node model</td>
<td>IBM 8246-L2T</td>
</tr>
<tr>
<td>CPU</td>
<td>IBM Power 7 8-cores, 4 threads/core, 3.6 GHz, 64 MB L3 Cache</td>
</tr>
<tr>
<td>RAM</td>
<td>128 GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>CentOS Linux 7</td>
</tr>
<tr>
<td>Build tools</td>
<td>OpenJDK 64-Bit Java 1.7.101 &amp; gcc version 4.8.5</td>
</tr>
<tr>
<td>Spark deployment</td>
<td>Spark v1.6.1 with YARN, Hadoop 2.5.1</td>
</tr>
</tbody>
</table>

Table 5.1: Test setup for the Spark-JNI micro-kernels benchmarking

Native invocation micro-kernel

We are benchmarking the access to the native code from a map transformation: a call of the C function that should implement functionality, but is left unpopulated on purpose. Aside from the actual function call, each Spark executor needs to load the native library from disk (in our case, the file has been added to Spark cache). Putting the overall focus on the overhead of the JNI access, the previously-mentioned two events are timed with the following test setup:

- A variable sweep is performed on the number of executors/partitions (we keep the same number of Spark executors and RDD partitions). Furthermore, for one benchmark launch, we are performing 16 iterations of the native access-mapper on our target RDD.
In the end, we are measuring the total *access time* as observed from within the transformation and its components:

\[ T_{access} = T_{libLoad} + T_{invocation} \]

**Memory copy micro-kernel**

This benchmark targets the evaluation of raw memory copy performance and of the overall performance overhead as seen from the Spark environment (is the overhead significant, as seen from *outside* of a transformation?): loading the native library, calling the native function which copies the atomic object’s content (a primitive array) to a new Java object which is then returned to the Spark JVM through a newly-created Java atomic object.

Serialization and deserialization of data to/from RDDs is a costly operation and since we are targeting an in-memory object clone, we should abstract as much as possible over creating RDDs and collecting the data from them. For this reason, we are performing 64 iterations of the copy-mapper on the data-RDD (moving the data 64 times), simulating typical iterative processes (e.g. K-Means). Each data container has a 256 KB array, and a sweep is performed on the number of containers to be copied in each iteration.

The memory copy metric includes the access overhead evaluated in the previously-presented kernel. The breakdown of the memory-copy function is:

\[ T_{memcpyFunc} = T_{libLoad} + T_{invocation} + T_{raw_memcpy} \]

The returned data is stored in a secondary array in the bean that has been passed through the function. This choice avoids including the object creation overhead in the memory copy overall result.

**Convolution micro-kernel**

The convolution algorithm performs a blurring effect (homogeneous kernel) on a given image, stored as a raw 1D array in an atomic object in Java. This case provides, basically, an extra step over the previous benchmark: the addition of the processing layer to the suite of library load, native access and memory copying.

Since this metric targets evaluation of many cumulative factors, here we are timing both the *collect* Spark action and native function invocation time after the convolution-mapper transformation has been performed and the resulting RDD, cached. Because the Spark programming model relies on lazy-evaluation, we need to call an action for triggering the execution flow of the previous stages in the DAG, in order to retrieve the results on the Spark driver (or to permanent storage).

The combined metric is defined as follows:

\[ T_{convolutionFunc} = T_{libLoad} + T_{invocation} + T_{convolutionProcessing} + T_{newObject} \]

This use case has been further split in two:

- The first use case concentrates on raw computation and memory access, with a homogeneous set of 64 noise images (random grayscale pixels) with constant size, leading to, basically, symmetric processing on all atomic objects (image containers). The number of images is a multiple of the partition size, assuring similar workload for all executors.
• A more realistic benchmark where the test batch is a set of 64 real-world images of
different sizes (ranging from 64x64 up to 60 megapixels). This case simulates a stage
of an image processing pipeline and should be significant to the "Spark Java vs JNI-
enabled Spark" assessment.

For each micro-kernel, the reference implementation was done in Java and the evaluated
JNI-enabled implementations are coded in C\(^1\).

For the convolution micro-kernel, for the sake of the comparison, we have added a third
use-case, where the *parallelism* is achieved with the help of OpenMP, not Java executors:
there is only one Spark executor per node, which launches the native kernel, which spawns
OpenMP-managed threads for the image convolution task.

## 5.2. Results and discussions

### Kernel access

Evaluating the invocation overhead involves assessing the important functional compo-
nents that contribute to the overall metric. As mentioned, we have benchmarked the shared
library load time, the native invocation cost and the total collect() time (done each iteration
after the mapper transformation).

![Graphs showing collection time vs iteration number and first collect vs number of workers](images/graphs.png)

**Figure 5.1: Collect time for the access micro-kernel**

In the Figure 5.1, the *collect()* action is evaluated as follows:

• **Figure 5.1a.** The measured time of the collect action in each iteration. The sharp
decrease in the action execution time after the first iteration shows the effects of
instruction caching and the JVM’s *warming up* leads to the constant improvement
thereafter. Except for the first iteration, the JNI micro-kernel outputs a 5-12% slower
time than its Java counterpart.

• **Figure 5.1b.** The measured time of the first iteration, for a number of workers and
partitions starting from 1 up to 64. It is noticeable that the first native access has

\(^1\)The native invocation micro-kernel does not have a Java implementation, since it is meant to test the JNI-
induced overhead.
lower latency when running the optimal number of cores (8 cores/workers in this case). Starting from 16 executor cores, the launch and management routine of the JVM starts to show its overhead, as the collect() duration starts increasing for both implementations.

In this scenario, both implementations perform similarly, with inconsistent differences in the range of -5.3% to 2.5% for the JNI-Spark vs pure Java Spark.

The breakdown of a transformation performed through JNI reveals surprising results in Figure 5.2: in both invocation (5.2a) and library load times (5.2b), the average metric decreases exponentially with the number of executors, even when exceeding the optimal number of hardware managed threads. Additionally we have to note that the invocation cost goes as low as 2 us, while the library load takes two orders of magnitude more (minimum 225 us).

The second plot in the graphs of Figure 5.2 expresses a cumulative value of the measured event, for all Spark workers. Here we can find the point of maximum efficiency, as expected, for 8 workers (each with its own partition). The general remark of both cases is that, probably, instruction caching and JVM dynamic optimization helps in reducing the overhead of accessing a shared resource, in a multi-threaded scenario.

![Graph](image)

(a) Invocation time  
(b) Library load time

Figure 5.2: Average and cumulative results for native access time breakdown, for variable number of workers

The access time metric imposes constrains on the native implementation: if we want to reduce the invocation cost to a negligible value (e.g. <1%) the algorithm complexity in terms of raw execution time needs to take at least 200 ms if only one iteration is performed; or at least 25 ms if the Spark transformation is iterative (more than 16 looped transformations).

**Memory copy kernel**

In Figure 5.3, we have plotted the results of the memory copy micro-kernel. The metric presented is the average transfer speed, as measured from the outside of the transformation. The complete formula is:

\[
\text{Transfer speed (MB/s)} = \frac{(64 \times \text{dataSize})}{T_{\text{collect}}}
\]
As shown in the formula above, it is important to note that the chunks being copied each iteration are 64 times smaller than the values represented on x axis, which represents the total data movement.

Although the overall metric yields fairly unimpressive results, much lower than what can be achieved over the POWER 7 memory bus (at least an order of magnitude lower), the JNI-induced overhead is reduced at $\sim 10\%$ and it decreases with the increase in data size. We can only speculate on the small metric, with possible reasons being un-aligned memory access due to the JVM, cache trashing etc.

![Pure Spark vs JNI transfer speed](image)

Figure 5.3: Memory copy micro-kernel benchmark

**Convolution kernel**

As mentioned, the first convolution test case stressed a correlation between the atomic element size (an image which will be sequentially convoluted), number of executors and speedup vs Spark Java. The results are presented in Figure 5.4. In this case there are 2 variable sweeps:

- The dimension of the images that forms the batches ranges from 64 up to 4096 (e.g. 64x64, 128x128, .. 4096x4096).
- The number of executors (when running pure Java or native-C++ code) or OpenMP threads (when parallelization is handled with the help of OpenMP, inside the native kernel).

The results as presented are speedup of the cases-of-interest (Native C and OpenMP kernels) over the Java Spark implementation. It is easily distinguishable that the speedup is increasing for the JNI implementation, as we can see that the JNI-enabled runs overcome the invocation overhead (presented previously) and outperforms pure Java Spark in performance even at the smallest image size (64 x 64 pixels).

With the increase of image sizes, the invocation and library load overhead becomes insignificant to the total compute time, and the native user code becomes up to 6 times faster than the Java-implemented kernel.
The results of the real-world, predefined batch convolution experiment can be investigated in Figure 5.5. The layout, legend and methodology is the same as in the previous case (Figure 5.4, with the exception of the secondary variable sweep: now, we are also stressing the dependency to the computational intensity of the accelerated algorithm.

From a pure performance standpoint, the native kernel reaches a computational throughput of approx. 26 GFlop/s (16 threads) which is comparable to what could be possible on the POWER 7 machine, with un-vectorized code. These quantities shows that, for this simple case, running native code contained in the Spark environment does not significantly impact performance in comparison to full-implementation native code (C code).

We can also identify a sensitive correlation between the number of executors and the JNI’s speedup over Spark Java. A possible explanation for this trend is the JVM’s reduced capability to optimize multi-threaded Java code.

For the realistic image batch, we can see that native C outperforms Java by 5 up to 12 times, showing better results than the previous test case. Even though OpenMP is faster the Java Spark, it’s almost twice as slow to the native C implementation.

5.3. Comparison with SparkCL

Following, the SparkJNI framework is compared against the SparkCL framework, the sole competing solution which offers heterogeneous integration with Spark.

First, as mentioned in section 3.2.2, SparkCL is an extension of the Aparapi framework, offering a Java-OpenCL bridge with automatic code generation. The simplicity of use constrains the expressivity of the underlying native implementation: code generation is restricted to OpenCL and the OpenCL code translation is restricted as well by lack of C++ supported features that would support a natural transition from the OOP characteristic of Java.

The algorithm implemented in Spark transformations is the one being translated to OpenCL and can be inspected and improved, if necessary. However, setting up the OpenCL workspace is transparent to the user and can impact the final result in terms of perfor-
mance, by limiting the configuration freedom to the API-exposed OpenCL functions. On the other side, SparkJNI does not constrain expressivity by fixing less points in the final native design, by only building the data and control links. In result, for accommodating applications identical to ones developed in SparkCL, the developer needs to take care, additionally, of writing the OpenCL kernel and setting up the OpenCL environment. The paid price allows for a more unconstrained tuning of the implementation for more algorithm types and higher performance.

Performance and framework overheads assessments: Pi calculation
For assessing a quantitative difference between the two frameworks, in term of performance, a sample algorithm provided with the SparkCL repository has been considered, regarding it to be an optimized and tuned example for this framework: a calculation of the Pi number based on random numbers. The original implementation has been translated to a SparkJNI-backed OpenCL implementation.
A simple representation of the algorithm’s concept is shown in Figure 5.6 and it can be described as consisting of three sequential steps:

1. Generate random points (pairs of random numbers) in a square interval \([-1, 1]\).

2. Decide whether each point is in the circle or not.

3. Calculate the percentage of points that were drawn in the circle tangent to the enclosing square. By multiplication of this percentage by 4, an approximation of \(\pi\) will be obtained.

An important observation is that the presented algorithm is memory bounded, since computations in each step are reduced in complexity. From the above-mentioned steps, only the second step (deciding whether a point is or not in the circle) has been accelerated within an OpenCL kernel, following the design decision in the original implementation of the SparkCL version.

The test setup is presented in Table 5.2.

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computing configuration</td>
<td>1 node</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Core i5-6300U, 2-core, 2 threads/core, 3.0 GHz, 3 MB L3 Cache</td>
</tr>
<tr>
<td>RAM</td>
<td>8 GB</td>
</tr>
<tr>
<td>OpenCL device</td>
<td>Intel GPU HD520 (24 compute units), 1.0 GHz, OpenCL 1.2 enabled</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 14.04</td>
</tr>
<tr>
<td>Build tools</td>
<td>OpenJDK 64-Bit 1.8.0_111 &amp; gcc version 4.9.2</td>
</tr>
<tr>
<td>Spark deployment</td>
<td>Spark v2.0.1, Hadoop 2.7</td>
</tr>
</tbody>
</table>

Table 5.2: Test setup for OpenCL Pi algorithm benchmarking

The benchmarked times for both implementations consist of end-to-end runtimes of the three stages of the algorithm and, also, the compilation and build times, since they cannot be separated easily in the SparkCL framework.

The first benchmarked scenario consists of a workload evaluated on a local configuration of Spark, with a swipe on the number of executors, number of partitions and partition size. Since the major constraint in this application is the memory access, this scenario stresses the efficiency in memory utilization for both frameworks.

The results can be assessed in Figure 5.7a. Here we have plotted the execution time for the algorithm with inputs ranging from \(2^{12}\) (4096) up to \(2^{27}\) (130.8 million) points, on RDDs with 4 partitions. The results are surprising with similar results for the evaluated implementations:

- For small sizes, the SparkJNI implementation is up to 18.5% faster (for \(2^{12}\) points) than its SparkCL counterpart.
• Towards the end of the input size spectrum, the SparkJNI-OpenCL Pi implementation becomes up to 8.9% slower than the SparkCL implementation (for $2^{27}$ points).

The second evaluated scenario is the framework overhead and OpenCL launch time for the evaluated accelerator. Since the two frameworks work differently, but both execute common steps before execution start (e.g. source code generation, build, library load etc.), the metric for this scenario has been decided to measure the total execution time, for the same algorithm, for the smallest possible work-group size (in this case, 8 points).

The results for the second benchmarked scenario can be inspected in Figure 5.7b. In this case, surprisingly, the SparkJNI implementation shows faster 35% execution time than SparkCL. Assuming that the compile time of OpenCL kernels is constant, we can conclude that the Java implementation of SparkCL and Aparapi create a considerably greater overhead than SparkJNI’s native link.

When benchmarking against a variable number of Spark executors per node, a disappointing observation reveals that, natively, OpenCL devices do not easily offer multi-tenancy: dividing the work in more native threads should keep the execution time approximately constant, but, the result, in turn, shows an inexplicable consistent reduction in performance, for both implementations. A thorough research shows that partitioning of OpenCL devices, called device fission, needs manual configuration, from a single starting point (master thread), a design which does not naturally fit into the design of a Spark-driven implementation, where native threads are independent of eachother.

In regard to this issue, SparkCL does not offer such solution and the example provided is impacted if more executors are allowed to use the same OpenCL resource. Since SparkCL is tuned for OpenCL, we would have expected device fission to be implemented for an efficient use of these types of accelerators. The lack of such a adaptation results in a constrained use of Spark, by limiting the number of executor threads to one per OpenCL node.
Reflection
Both SparkJNI and SparkCL frameworks share the commonality of allowing high performance accelerators integration with Spark, with a significant distinction in the ease-of-programming and expressivity department: SparkJNI targets more freedom in native implementations, while SparkCL allows for easier and fast prototyping of parallelized OpenCL transformations in Spark applications.

The previous observation about device fission calls for a possible extension to the current design of the SparkJNI framework, with a singleton native component which would take care of the device partitioning, universal setup and clean of the OpenCL environment. From there, independent Spark executors would be able to queue to launch jobs concurrently on the accelerator.

In terms of performance, as evaluated by the memory-intensive algorithm, SparkJNI shows better execution times for smaller input sizes, while asymptotically targeting the performance of SparkCL for big input sizes, with a maximum overhead of 8.9%. The figure represents a worst-case scenario, since most algorithms present higher computational intensity, a metric which is agnostic of the higher-level implementation (it depends only on the OpenCL accelerator).

The obtained results of the comparison with SparkCL confirm that the SparkJNI framework is universally suited for native hardware acceleration, by competing in terms of performance with a OpenCL-targeted implementation of a Apache Spark extension.

5.4. A SparkJNI FPGA PairHMM algorithm
The SparkJNI framework can considerably reduce development effort and allow for a more stable implementation when integrating Java and native code. Since the framework is meant, by design, to enable any kind of native code, we have decided to show-case its power with the use of true heterogeneous integration in Spark.

For this reason, following the trend of high-performance DNA research in academia on big data enabled solutions, such as GATK[31], Halvade[32], BWA[33]. and others, a novel PairHMM algorithm [34] implemented for reconfigurable hardware has been taken and encapsulated into a Spark application, through the SparkJNI framework.

5.4.1. DNA analysis and the PairHMM
DNA analysis applications are used for a better understanding and treatment of specific genetically-defined diseases (e.g. cancer, hereditary diseases, etc.)

While DNA sequencing costs are decreasing exponentially for the past 20 years, the amount of generated data still poses great challenges for their thorough analysis. The processing complexity, combined with the increased size require massive mainframes or even cluster computers to extract meaningful information from DNA sequences within a reasonable amount of time. For this reason, this field too falls under the incidence of the large computing scale challenges, in terms of efficiency, cost and power consumption.

Current research is targeting easier deployment and specialized acceleration for different stages of the DNA analysis pipeline [34] [3] [35]. However, there is no current framework that would help with deployment of a large scale computing cluster for this application, by means of heterogeneous acceleration.

For the mentioned reasons, along with researchers in [34], we are targeting both an in-
crease in efficiency and cluster-level deployment of a specific stage in the DNA processing pipeline: pairwise alignment of DNA sequences - PairHMM.

Although the PairHMM algorithm is similar to the Smith-Waterman algorithm [36], while the latter outputs the best possible alignment string of two sequences, PairHMM calculates the overall probability of alignment. A brief description of the algorithm's flow follows:

Given two DNA sequences (e.g. X and Y), the alignment probability needs to be calculated based on their similarity. The comparisons between symbols in the sequences can lead to three outcomes:

- Current symbols are the same, i.e. a match.
- Symbols are different, an insertion is evaluated in sequence X.
- Symbols are different, an insertion is evaluated in sequence Y.

Based on the possible allowed symbol comparison outcomes, a state machine parses the sequences until both of them are consumed. Each state passed by the machine has a predefined probability. The state-machine model is provided in Figure 5.8a, along with a sequence alignment example 5.8b.

For evaluating the overall alignment probability, a forward algorithm is used and implemented through dynamic programming [3], with a computation complexity of $O(N^2)$, where $N$ is the size of a sequence.

The quadratic complexity of the algorithm along with an increased execution pipeline and reduced dependencies between computation steps allow for easier parallelization on FPGAs, more efficiently than on CPUs.

In the next section, we will focus on the implementation details of the PairHMM algorithm with the SparkJNI framework.

5.4.2. Implementation details

The algorithm, as implemented in [34], can be split in three functional steps, for better use of resources:

- Load the sizes of the read and haplotype sequence pairs and create a workload descriptor. For maximum resource utilization, the values representing the sizes of the input data need to be sorted.
- Batches of atomic computations (a predefined number of pair alignment tasks) are formed with pair sequence content.
5.4. A SparkJNI FPGA PairHMM algorithm

- The workload descriptor and pair sequence content is fed to the processing algorithm, implemented on the FPGAs.

For implementing the above-mentioned pipeline in SparkJNI, we need to define the two categories of components used by the framework: the jni functions and the data containers (beans). This results in the following Spark execution flow as presented in Figure 5.9.

- **Workload** contains raw references to the haplotype and read sizes and number of batches. The workload is created based on the ordered sizes found in the input file `sizes.txt`, by the Create workload descriptor block.

- **RawByteArr** is created based on the workload specifications by the Load pair content and encapsulates the raw content of the haplotypes and reads.

- **PairHmmBean** encapsulates the above two mentioned reference types (workload and RawByteArr). The type is used as input for the last native call (PairHmm block), which calculates the pair alignment probability.

All map operations from Figure 5.9 are implemented in native code. The `zip` transformation (creation of the PairHmmBean, based on the Workload and RawByteArray objects) is performed in Java.

![Figure 5.9: PairHMM SparkJNI execution and data flow diagram.](image)

5.4.3. Evaluation

The algorithm provided by [34] has been adapted and integrated according to the steps mentioned previously and outlined in Figure 5.9. The first two map operations (creation of the workload descriptor and loading of the sequences content) have been implemented for the purpose of showcasing a full pipeline implemented with SparkJNI. The test setup is described in Table 5.3.

Considering the full native support of the map operations in SparkJNI, backed by implementations of the containers and jni functions presented in the previous section, we need to evaluate the overhead induced by a cold start of the SparkJNI framework. The steps are: generation of the JNI headers (`javah`), native code generation (`CodeGen`) through the SparkJNI CodeGen engine, native library build (`build`) and native library load (`libLoad`). The runtimes of the presented steps have been benchmarked and presented in Figure 5.10a.

We can observe that the library load and code generation for the native data containers take the least amount of time (1.3 milliseconds and 85 milliseconds). On the other side of
### Component Specification

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster configuration</td>
<td>1 node &amp; 2 sockets per node</td>
</tr>
<tr>
<td>Node model</td>
<td>IBM 8247-42L</td>
</tr>
<tr>
<td>CPU</td>
<td>IBM Power 8 10-core, 8 threads/core, 2.06 GHz, 80 MB L3 Cache</td>
</tr>
<tr>
<td>RAM</td>
<td>256 GB</td>
</tr>
<tr>
<td>FPGA</td>
<td>AlphaData ADM-PCIE-7V3</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 15.04</td>
</tr>
<tr>
<td>Build tools</td>
<td>OpenJDK 64-Bit Java 1.7.0_95 &amp; gcc version 4.9.2</td>
</tr>
<tr>
<td>Spark deployment</td>
<td>Spark v2.0.0, Hadoop 2.7</td>
</tr>
</tbody>
</table>

Table 5.3: Test setup for the PairHMM SparkJNI benchmarking

The spectrum, the JNI header generation (`javah`) and code build (`build`) take considerably more at 1.882 seconds and 2.128 seconds, respectively. Considering that only the `libLoad` is necessary for each Spark application run, in the context of applications that run for minutes and more, we can conclude that the SparkJNI framework imposes negligible overhead at startup.

The execution of the PairHMM algorithm (step PairHMM in Figure 5.9) has been benchmarked for the SparkJNI FPGA configuration. The baseline results are provided by benchmarking the implementation in [34] with the CPU run (20 OpenMP threads) and native FPGA execution. The benchmarks for all implementations have been based on a single variable sweep: the workload size, measured in number of pairs of sequences. This value ranges from a workload of 16 pairs, up to 32768 pairs.

For the biggest workload size, we have benchmarked the overall SparkJNI-induced overhead at the kernel level, and represented it in Figure 5.10b. Since the SparkJNI framework changes the call site environment, it is important to see how the overall quantifiable steps (in terms of runtime) compare in regard to the native implementation. The layered call stack can be visualized in Figure 5.11 and is briefly described in the following:

1. A `collect()` action is triggered by the application from the Spark environment for retrieving the PairHMM results to the Spark driver.

2. The above mentioned action, by triggering the execution on the unprocessed DAG, calls the PairHmmJniFunction (`jInvoke` in 5.11) by its inherited call method.

3. The PairHmmJniFunction calls the native FPGA-accelerated function (kernel), which retrieves the data from the received container, computes the PairHMM and returns the result on the same path back to the Spark driver (Step 1).

The composition of the total `collect()` action time is represented in Figure 5.10b. The common ground for both implementations is the actual PairHMM C++ kernel, which has been benchmarked as a whole and it is the sole representative of the cumulative execution time of native-FPGA implementation (left bar). In addition to the actual PairHMM kernel,
5.4. A SparkJNI FPGA PairHMM algorithm

![SparkJNI offline functional steps breakdown](image1)
(a) SparkJNI offline functional steps breakdown, (b) SparkJNI-PairHMM online kernel call stack times comparison with native-FPGA PairHMM

Figure 5.10: Analysis of the SparkJNI-induced overhead, for both online and offline runtimes, on the PairHMM FPGA-accelerated algorithm

slightly adapted for the SparkJNI implementation (right bar), we have Java invocation times which take around 16 milliseconds, and the Spark intrinsics (data reshuffling, serialization, management routines, etc.) which account for a considerable 381 milliseconds addition to the total collect() action runtime.

Based on the measurements displayed in Figure 5.10b, we can compute an overhead induced by the SparkJNI implementation against an ideal manually-implemented Spark-JNI application (following the guidelines of the reference design presented in Chapter 4). This overhead, representable here as the dynamic invocation time (16.3 milliseconds), values at around ∼4.2%.

![SparkJNI kernel call-stack representation](image2)

Figure 5.11: SparkJNI kernel call-stack representation

In Figure 5.12a, the throughput of the implementations is expressed in MCUP/s (mega cell updates per second), based on the following formula:
Throughput(MCUP/s) = \frac{Number_{CellUpdates}}{T_{runtime}(us)}

The number of cell updates is precomputed for each pair of sequence sizes, and divided by the kernel runtime for obtaining the throughput.

Based on the above metric, the speedup versus the CPU 20-threaded run is presented in Figure 5.12b, where FPGA\textsubscript{j} represents the SparkJNI implementation and FPGA\textsubscript{n} represents the native (C) implementation. The same notation convention applies to Figure 5.12a as well.

The throughput achieved by the SparkJNI PairHMM implementation (5.12a) reaches 2.183 GCUP/s, a figure 2.7 times higher than current state-of-the-art multi-threaded implementations on CPUs [37], which peak at 809 MCUP/s.

In our case, the results over the un-optimized C code are even more conclusive, with SparkJNI outperforming the CPU run by approximately 21 times, for the biggest workload size, while being less than 15% slower than the native FPGA implementation. However, for smaller workloads, we can observe the effect of the Spark-induced overhead, reducing the performance to the level of the CPU 20-threaded runs.

The results of the integration of the novel PairHMM algorithm implemented in [34] with SparkJNI show that performance remains mostly untouched in comparison to the baseline, with as low as 14.5% reduction in throughput. Considering the considerable gains imposed by the new systolic array design based on which this integration has been done, the result obtained by the SparkJNI implementation is still well ahead (2.7x) of the state-of-the-art implementations in the field.
Conclusions and future perspectives

6.1. Conclusions
The presented research aimed at quantifying a broad range of discussions and directions that are related to finding common ground between Big Data, HPC and heterogeneous systems.

The related work has shown that, even though no mature solution exists at the moment, research is picking pace and diversifying. The presented solutions have sparked interest and bring major contributions to the field. They also bring valuable assets to our framework as well, by creating controversy by their design decision, implementation details and result justification. Since there is no clear contender (neither by execution or by marketing) for taking into consideration for Spark integration, we considered that a more in-depth study is needed.

In this direction, we have proposed a reference design that can be used for accelerating Apache Spark with native/heterogeneous embodiments, through the JNI. Our conclusions will be evaluated against the initial research questions. The experiments conceived with this concept have shown low native-access overhead of the native implementation, in comparison to the Java code, in areas such as invocation time, native library load time and memory transfer speed. In terms of computing performance, our solution has proved to be up to 12x faster than pure Java-Spark solutions, with the help of the native convolution kernels.

Based on the results obtained with the JNI-accelerated Spark design, by finding solutions for good compatibility with specialized infrastructures and libraries, the SparkJNI framework has been implemented for means of reducing the development efforts and level the learning curve. The framework has been tested against SparkCL[22] for a quantitative comparison against an existing implementation. The result shows similar or better performance for the SparkJNI, confirming the framework as a valid prototyping toolchain, with open ends towards a dynamic plugin.

Validation of the SparkJNI framework has been done with a real-life application consisting of an FPGA-accelerated DNA analysis pipeline stage. The results show reduced overhead and pave the way to further improvements in terms of memory access.
The reduced overhead of as low as 14% of the PairHMM algorithm [34] and the successful integration with Spark open the way to possible deployments of fully heterogeneous-accelerated systems. Furthermore, taking into consideration the benefits offered by the Spark framework in terms of **maintainability, scalability, fault-tolerance** and **ease-of-deployment**, the SparkJNI would lessen the development effort of clustered computing architectures, set a cornerstone for a full pipeline acceleration of DNA analysis applications and considerably reduce analysis times for time-critical fields.

The work and outcome of this thesis, materialized in the form of the SparkJNI framework and its related examples (including the PairHMM implementation), are publicly available for open-source development on GitHub.

### 6.2. Future perspectives

This Apache Spark native code interface, as evolved from SparkJNI, will potentially have a big impact on code developed for big data applications as it allows developers to use existing high-performance native libraries and integrate them effortlessly within their Spark application. For many applications, this has the potential to significantly increase the performance of compute-intensive code, with the SparkJNI project being a strong candidate for many Big Data projects developed in the industry and within academia.

In the long run, we desire to enrich the current implementation towards a dynamic binding that can be integrated with the Spark runtime engine, for better resource allocation, code specialization and seamless integration with native libraries. Furthermore, we are working on additional data interface links that can optimize memory access and reduce the impact created by the JVM’s memory management engine.

Last, full code generation is desirable, by means of creating smarter parsers that are dedicated to the Spark and HPC environments. We consider that the current programming languages trend is converging in terms of expressiveness to natural languages, and if specialized use-cases like ours are targeted, an evolved SparkJNI framework would trigger a definitive paradigm shift in deployment of Big Data infrastructures and applications.
A

Publications
ABSTRACT

In light of the recent improvements in computing units, the de facto structure of a high-performance computing cluster, ordinarily consisted of CPUs only, is superseded by heterogeneous architectures (comprised of GPUs, FPGAs, and DSPs) which offer higher performance and lower power consumption. Big Data, as a younger field but with a much more aggressive development pace starts to exhibit the characteristic needs of its archetype and the development community is targeting the integration of specialized processors here, as well. In this research, we analyze the state-of-the-art developments in the field of heterogeneous-accelerated Spark, the current Big Data standard, and we provide a reference design and implementation for a JNI-accelerated Apache Spark application. The design is validated by a set of benchmarked micro-kernels. The JNI-induced overhead is as low as 12% in access times and bandwidth, with speedups up to 12x for compute-intensive algorithms, in comparison to the pure Java Spark implementations. Based on the promising results of the benchmarks, the SparkJNI framework is implemented as an easy interface to native libraries and specialized accelerators. A cutting-edge DNA analysis algorithm (Pair-HMM) is integrated, targeting cluster deployments, with benchmark results for the DNA pipeline stage showing an overall speedup of ∼2.7 over state-of-the-art developments. The result of the presented work, along with the SparkJNI framework are publicly available for open-source usage and development, with our aim being a contribution to current and future Big Data Spark shift drivers.

Keywords

Apache Spark, Java JNI, FPGA, Pair-HMM

1. INTRODUCTION

At the rise of the Big-Data field, it was a completely distinguished domain that was targeting simple predictions, information extraction and market research, mostly serving commercial purposes. However, today we are witnessing a more aggressive expansion, since this field emulates to a more vast field of application. The paradigm serves today the needs of scientific and research fields, companies’ commercial and business purposes and end users as well.

While once the Big Data standard, Hadoop MapReduce, was targeted at “commodity hardware”, now, we cannot afford burning power in commodity hardware to keep up with the digital era’s demands. The only solution is to retarget our development efforts towards energy efficiency in terms of computation (joule/operation). For this, the trend needs to be reverted a bit and uncover a portion of the abstraction barrier that has been installed over the hardware and low-level details with the advent of the Big Data field.

Developments in non-conventional computing units, such as FPGAs, GPUs, and DSPs, offer a reduction of the energy footprint of computing clusters, while also increasing performance. For achieving this purpose, the intersection of Big Data’s scalability, reach and ease-of-deployment, on one side, and the specialization, expertise and efficiency in resource utilization of HPC, on the other side, needs to materialize.

This paper is structured as follows: in Section 2, we will evaluate state-of-the-art developments in the field of heterogeneous Spark frameworks. In Section 3 the design of the SparkJNI framework is presented. In Section 4, a set of micro-kernels are benchmarked, while in Section 5, a novel DNA analysis pipeline stage is deployed on FPGAs, through SparkJNI and evaluated. Last, in Section 6, we will outline our conclusions and present future perspectives for the SparkJNI framework, as well as for heterogeneous Apache Spark cluster deployments.
Bibliography


[34] J. Peltenburg and Z. Al-Ars, “Maximizing Systolic Array Efficiency to Accelerate the PairHMM Forward Algorithm,” Quantum Engineering Lab, Delft University of Technology, [Unpublished].

