
# CHALLENGES AND TECHNIQUES FOR TRANSPARENT ACCELERATION OF UNMODIFIED BIG DATA APPLICATIONS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2022

Maria Nektaria Xekalaki

Department of Computer Science

# Contents

**Word Count: 27454**

# List of Tables

# List of Figures

# Abstract


CHALLENGES AND TECHNIQUES FOR TRANSPARENT ACCELERATION OF UNMODIFIED BIG DATA APPLICATIONS

Maria Nektaria Xekalaki

A thesis submitted to The University of Manchester for the degree of Doctor of Philosophy, 2022

The ever-increasing demand for high-performance Big Data analytics and data processing has paved the way for heterogeneous hardware accelerators, such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), to be integrated into modern Big Data platforms. Currently, this integration comes at the cost of programmability, as the end-user Application Programming Interface (API) of Big Data frameworks must be altered in order to access the underlying heterogeneous hardware. In some cases, it is even required by developers to provide their application code in a low-level programming language that targets specific hardware accelerators (e.g., CUDA, OpenCL, etc.).

The purpose of this thesis is to identify the current barriers in the automatic acceleration of Big Data applications and to propose techniques that can lift the emerged restrictions. Specifically, this thesis presents the first Big Data platform that can dynamically take advantage of GPUs and FPGAs for the acceleration of unmodified applications in a completely agnostic manner to the user. This novel heterogeneous platform has been prototyped in the context of Apache Flink, a widely used Big Data platform, and TornadoVM, an open-source framework that automatically compiles and executes Java applications on GPUs, FPGAs, and multi-core CPUs. The techniques that will be presented are not bound to the frameworks used, and can also be applied to other software platforms with slight modifications. The performance evaluation of the proposed solution has been conducted on both standard benchmarks and industrial use cases,

showcasing performance speedups of up to 65$x$ on GPUs and 184$x$ on FPGAs, against vanilla Apache Flink running on traditional multi-core CPUs.

# Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

# Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420`), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see `http://www.library.manchester.ac.uk/about/regulations/`) and in The University's policy on presentation of Theses

# Acknowledgements

First and foremost, I would like to thank my supervisor *Christos Kotselidis* for his immense support over the last four years. He was always generous with his time and helped me expand my horizons in ways I never expected. Pursuing this PhD would not have been possible without him and I will always be grateful to him.

Moreover, I would like to thank *Juan Fumero* for teaching me everything I know about compilers and for always supporting me, and *Thanos Stratikopoulos* for his constant encouragement and for the help he offered so generously in times I needed it the most. I was very fortunate to be a member of such a qualified and supporting team. Thank you for always having my back.

Additionally, I am grateful to all the members of the *APT* lab for the interesting conversations we shared over the last few years and for making me feel part of the group from my very first day there. I will always be grateful for having the opportunity to collaborate with *Michalis Papadimitriou* and for all the insightful discussions we had. Moreover, I would like to thank the *University of Manchester*, the *E2Data Project* and *HiPEAC* for proving the necessary means to support this PhD.

Furthermore, I would like to thank *Dr. Paris Carbone* and *Dr. Andre Freitas*, for agreeing to be the reviewers of this thesis and for the invaluable feedback they provided.

On a more personal note, I would like to thank *Nikita Dewani* and *Jane Soo*, who were some of the very first friends I made in Manchester. I will always treasure our time together. Additionally, I would like to thank *Eleni Mataragka*, *Rigina Skeva*, *Eleni Maragkoudaki*, *Mark Kynigos*, *Katerina Papakyriakopoulou* and *Christos Hadjiarapis* for everything we shared together and for making Manchester my home away from home. Your love and support played a significant part in the completion of this PhD. I will always be grateful for my friends in Greece, *Epaminondas Koutavelis*, *Zoe Papadopoulou*, *Evdokia Chrysagi* and *Francesca Tsironi*. Thank you for being so loyal and for always believing in me. Furthermore, I would like to extend a special thank you

to *Vasso Ampatsidou*, who was always kind and supportive, and kept me sane through some very challenging times.

I feel very fortunate to have such a loving and supporting family. I am grateful for my siblings, *Evi* and *Chronis*, for my niece, *Konstantina*, and my nephews, *Iasonas* and *Filippos*, who always put a smile on my face. Last but not least, I would like to thank my parents *Kostas* and *Tasoula* for raising me and for providing their unconditional support throughout my life. Everything I ever achieve will always be thanks to you.

# Chapter 1

# Introduction

Over the last few years, the amount of data that is being created, captured and replicated globally has been increasing at a rapid pace. Specifically, it is estimated that from 33 zettabytes, in 2018, the amount of data the requires processing will reach the staggering size of 163 zettabytes by 2025 [RGR18]. This trend imposes several new challenges and opportunities regarding high-performance and energy-efficient data analytics. To adhere to the performance Service Level Agreements (SLAs) defined for Big Data applications despite this trend, heterogeneous hardware accelerators, such as Graphic Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) have been put forward as a means to achieve higher data processing throughput and energy efficient execution. Specifically, hardware accelerators can now be found in almost all modern cloud providers, such as AWS [Ser], Google Cloud [Clo], and Microsoft Azure [Azu], complementing the conventional CPU-only execution for accelerating suitable workloads.

In order to exploit these hardware accelerators, developers must write their code in non-managed programming languages and frameworks, such as CUDA [Cor21b], OpenCL [SGS10], and OneAPI [INT22]. However, in the domain of Big Data analytics, established systems that are typically written in managed programming languages (e.g. Java or Scala) and run on top of the Java Virtual Machine (JVM) [Ora22] are traditionally used. Since the development of Big Data platforms preceded the heterogeneity of clusters, they were designed with CPU-only execution in mind.

## 1.1 Challenges

With the exception of NVIDIA GPU acceleration for Apache Spark 3.x [RE20] via the RAPIDS API [rap21] and for Flink [Fou22b] via JCuda [Hut22b] and/or JCublas [Hut22a], the remaining of existing works are mostly academic efforts to bring heterogeneous hardware acceleration on various Big Data frameworks (e.g., [SCN+15, CLOL18, GBS16]). A common denominator of the current state-of-the-art, both on the academic and industrial side, is that new APIs, which are often specific to the target accelerator, have to be used and/or an implementation of the computation in a low-level programming language has to be provided. These characteristics have several disadvantages.

First of all, having multiple implementations of the user code, both in high-level and low-level languages, leads to *code fragmentation*. Naturally, a fragmented codebase is difficult to maintain and expand. For instance, with multiple implementations of the user function existing, for every minor change in the algorithm, all the versions of the code have to be updated. This is an error-prone and time-consuming process. Moreover, on systems that rely on pre-compiled kernels, the user function support is limited, as it is not viable to have a kernel implementation for every possible user function. (It is important to note at this point that, throughout this thesis, the word kernel will be used to refer to a method compiled for high throughput devices, such as GPUs and FPGAs.) Furthermore, in many cases, the kernels used by the system (generated or pre-compiled) might not be portable across different platforms. This could be the case if, for example, the kernels contain optimizations that are specific to their target device. Last but not least, some kernel implementations might lead to *vendor lock-in* (e.g. CUDA kernels can only target NVIDIA GPUs).

By studying the current approaches for heterogeneous execution of Big Data applications, the vision that motivated this thesis emerged: *to enable Big Data frameworks to utilize heterogeneous accelerators transparently, without breaking the programming norms or requiring any hardware knowledge from the developers*.

However, at the initial development stages of this vision platform, several challenges were identified. Firstly, the API exposed by Big Data frameworks is in high-level programming languages such as Java, Scala or Python, which cannot be directly used to program devices like GPUs and FPGAs. One way to target such devices from Java is to make a JNI call to precompiled C/C++ code. However, this approach requires multiple versions of the code to be implemented and developers have to become familiar with low-level architectural details for each target platform in order to write efficient

kernels. Thankfully, there are some frameworks that enable the compilation of Java programs on heterogeneous devices, with the most prevalent being Aparapi [apa16] and TornadoVM [FPZ+19, tor]. Nevertheless, the APIs of both of these frameworks are not compatible with the APIs of Big Data platforms. Exposing either the API of TornadoVM or Aparapi to the developers would break our vision to provide seamless acceleration.

Moreover, both have some crucial limitations. Aparapi is limited to multi-cores and GPUs and does not support Java Objects, which are the cornerstone of Object Oriented Programming (OOP). TornadoVM can target a plethora of heterogeneous devices but it is also restricted to primitive data types. Therefore, it is evident that in order to use any of these frameworks in combination with Big Data platforms - without imposing programming restrictions - they have to be extended first.

## 1.2 Research Questions

The research questions that this thesis explores are the following.

**RQ1.** What are the main challenges in transparent acceleration of Big Data applications?
**RQ2.** Is it possible to accelerate Big Data applications without breaking the existing Big Data programs or introducing new APIs?
**RQ3.** What are the preconditions of hardware acceleration and the performance trade-offs of seamless execution of Big Data applications on GPUs and/or FPGAs?

## 1.3 Research Methodology

The methodology presented below was followed in order to successfully answer the research questions of this thesis.

1. A comprehensive survey of the literature was performed, to identify the techniques that are currently used for acceleration of Big Data applications.

2. The architecture and execution models of the frameworks that were selected for this implementation were thoroughly analysed. Through this analysis, the main incompatibilities and challenges for transparent acceleration were identified.

3. A set of techniques that bridge the gap between the Big Data stack and heterogeneous computing were designed and implemented.

4. The proposed system was evaluated across a wide set of benchmarks to identify whether speedup can be obtained against scale-out CPU execution and under what conditions.

## 1.4 Contributions

The contributions of this thesis, classified based on the research questions they answer, are presented below.

- **Research Question 1.**
  - It performs a comprehensive analysis of the whole execution stack of the Big Data framework of choice and discusses the challenges of heterogeneous execution for each layer.
- **Research Question 2.**
  - It presents a novel approach for enabling automatic and transparent GPU and FPGA acceleration for existing Big Data applications written in Java. To achieve that, two novel techniques are introduced: 1) automatic code and data morphing, and 2) Just-In-Time (JIT) compilation for heterogeneous hardware, in the context of TornadoVM.
- **Research Question 3.**
  - It performs a comprehensive performance evaluation of the proposed system across a variety of benchmarks and real-world industrial use cases against the CPU-only version of the Big Data framework.
  - It discusses the merits of heterogeneous hardware acceleration, while also highlighting the pre-conditions that must exist in order to achieve performance improvements when running on GPUs and FPGAs compared to scale-out CPU only configurations.

## 1.5 Thesis Layout

The remainder of this thesis is organized as follows:

- **Chapter 2** introduces background concepts and technologies that will be used throughout this thesis.
- **Chapter 3** presents how the map-reduce model, which is adopted by most Big Data platforms, has been used in heterogeneous programming over the years. Additionally, it provides an in-depth analysis of the current advances in augmenting Big Data frameworks with heterogeneous devices.
- **Chapter 4** discusses the challenges of implementing the vision platform of this thesis and presents an initial prototype that was developed. This prototype does not have much innovation compared to the state-of-the-art, however, its development illuminated the reasons why the gaps in the state-of-the-art exists. Moreover, it was used as a stepping stone for the implementation of the final platform.
- **Chapter 5** proposes an innovative Big Data platform, that can seamlessly utilize various hardware accelerators by performing JIT compilation. Following a thorough research on the relevant literature, it is believed that this is the first time that a Big Data framework can automatically target different types of devices (i.e. GPU and FPGAs) without providing new APIs or requiring any intervention from the user.
- **Chapter 6** presents an exhaustive experimental analysis of the proposed platform. During this analysis, the pre-conditions that should exist so that Big Data applications can gain performance are highlighted, by running on heterogeneous hardware devices. This thesis aspires to provide insights that will be invaluable for any future development of Big Data frameworks designed with heterogeneity in mind.
- **Chapter 7** provides a summary of the insights gained throughout this work, as well as proposals for future research.

## 1.6 Publications

The work presented in Chapter 4 has been published in the following venue.

- **Maria Xekalaki**, Juan Fumero, and Christos Kotselidis. Challenges and proposals for enabling dynamic heterogeneous execution of big data frameworks. In 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pages 335–341, 2018. [XFK18a]

In addition, part of the work of Chapter 4 and Chapter 5 has been presented in the following poster sessions.

- **Maria Xekalaki**, Juan Fumero, and Christos Kotselidis. Dynamic acceleration of big data applications on heterogeneous hardware resources. 14th International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems, ACACES 2018, Poster abstracts, 2018. [XFK18b]

- 6th Compiler and Programming Language Summit in Munich, Germany, 3-5 December 2018.

- 7th Compiler and Programming Language Summit in Munich, Germany, 9-11 December 2019.

Moreover, the following papers have been produced in the process of maturing TornadoVM.

- James Clarkson, Juan Fumero, Michail Papadimitriou, **Maria Xekalaki**, and Christos Kotselidis. Towards practical heterogeneous virtual machines. In Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Programming'18 Companion, page 46–48, New York, NY, USA, 2018. Association for Computing Machinery. [CFP+18a]

- James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, **Maria Xekalaki**, Christos Kotselidis, and Mikel Luján. Exploiting high-performance heterogeneous hardware for java programs using graal. In Proceedings of the 15th International Conference on Managed Languages Runtimes, ManLang '18, New York, NY, USA, 2018. Association for Computing Machinery. [CFP+18b]

- Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, **Maria Xekalaki**, James Clarkson, and Christos Kotselidis. Dynamic application reconfiguration on heterogeneous hardware. In Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2019, page 165–178, New York, NY, USA, 2019. Association for Computing Machinery. [FPZ+19]

Finally, the paper below, which contains the contributions of Chapter 5, has been accepted for publication in PVLDB vol. 15 and will be presented in VLDB 2023.

- **Maria Xekalaki**, Juan Fumero, Athanasios Stratikopoulos, Katerina Doka, Christos Katsakioris, Constantinos Bitsakos, Nectarios Koziris, Christos Kotselidis. Enabling Transparent Acceleration of Big Data Frameworks Using Heterogeneous Hardware.

## 1.7 Summary

This chapter motivated this thesis by elaborating on the necessity to take advantage of heterogeneous devices to perform Big Data computations. Moreover, it summarized the four key contributions presented this thesis, which are: (i) an extensive analysis of current Big Data platforms and the design features that make the adaption of heterogeneous execution challenging, (ii) the introduction of a novel Big Data platform that can dynamically run existing applications on multiple different devices in a user-agnostic way, (iii) detailed evaluation of the proposed platform using multiple use cases, and (iv) an in-depth analysis of the preconditions that should exist to get optimal performance out of heterogeneous execution compared to the CPU scale-out. Additionally, a list of the publications produced in the scope of this work was provided as well as a brief outline of the chapters of this thesis.

# Chapter 2

# Background

This chapter covers the concepts and programming models that will be used throughout this thesis. Specifically, Section 2.1 provides a general overview of Big Data frameworks, focusing on Apache Flink, Apache Spark, and Hadoop, three of the most widely used platforms among Big Data developers. Emphasis is given on their processing models (subsection 2.1.1) and their architecture (subsection 2.1.2). Additionally, subsection 2.1.3 specifies the positioning of the work presented in this thesis in the universe of these artefacts. Furthermore, Section 2.3 introduces heterogeneous computing and provides information about hardware accelerators, specifically GPUs and FPGAs. Finally, Section 2.4 overviews the concepts introduced in this chapter.

## 2.1 Big Data Frameworks

As the amount of data that is being produced globally keeps growing, processing it in single-machine, monolithic systems is becoming increasingly inefficient - if not impossible. Distributed computing is an efficient solution to this bottleneck, as the computational load is shared among several machines. However, it imposes plenty of challenges, including load balancing among the machines, fault tolerance, scheduling, synchronization and more. Big Data frameworks were developed as a way to formalize solutions to these challenges.

Some of the most popular Big Data frameworks at the moment are Hadoop, Apache Spark and Apache Flink. All three of these systems are written in managed programming languages, i.e. languages that are built on top of a managed runtime system (e.g., the Java Virtual Machine). The next subsections will provide a general overview of their processing models and their architecture. Scheduling and fault tolerance are out

of the scope of this thesis.

### 2.1.1 Processing Models

The term processing model refers to the way that the computation is modeled and the data is processed. Khalid et al. [KY21] classify the data processing models of Big Data frameworks into three categories. The first category consists of those frameworks that process data in batches. Batch processing refers to computations applied on a set of data collected over a specific time frame. Hadoop belongs in this category. The second contains the frameworks that are used for stream processing, e.g. Apache Storm [Fou21c] and Samza [Fou22a]. In stream processing, the computations are performed on the data in real time, as it is generated. Finally, the third category includes frameworks that support both batch and stream processing such as Apache Spark and Apache Flink.

Regarding the way that the computation is modeled among Big Data frameworks, the task are most commonly expressed through map-reduce functions (e.g. Hadoop) or they can by represented by a directed acyclic graph (DAG) (e.g. Apache Spark, Apache Flink). Next, a brief overview of each computational model is provided.

#### Map-Reduce Model

The *map* and the *reduce* programming patterns were originally introduced by Murray Cole [Col89]. Cole presented a new way of parallelizing distributed computations, which was to use a set of templates, referred to as *algorithmic skeletons*. By abstracting the parallelism, these skeletons shift the focus from the way that each system facilitates parallel execution to the actual computation to be performed. Several skeleton functions were presented by Cole (e.g. *stencil*, *scan*, etc.) which are out of the scope of this thesis.

A formal description of the map and reduce skeletons is the following. The map skeleton applies a function `f` on an input set of type `x` to produce an output set of type `y`.

$$map(f) : x[] \rightarrow y[]$$

Similarly, the reduce skeleton applies a function `f` on an input set of type `x` to produce a single element of type `y`.

$$reduce(f) : x[] \rightarrow y$$

For both the map and the reduce skeletons, the types `x` and `y` can be identical.

Several years later, Google advocated using the map and reduce skeletons for processing large datasets [DG04]. The *map-reduce programming model*, as proposed in this work, consists of three parts:

- A map function that applies a user-defined computation on each element of a `<key, value>` input set and produces a new set of `<key, value>` intermediate results.
- A process called *shuffling*, during which the results of the map are split into groups, one for every unique key. Each group is then provided as input to the reduce function.
- A reduce function that merges the values of each key group, in a way that is specified by the user, and returns a dataset containing a single element.

Figure 2.1 illustrates how a word count computation is executed with the map-reduce programming model, following the three stages described above. In the first stage, each word is mapped with a counter that has the value of one. In this example, the words are considered the `keys`, so during the shuffling eight groups are created, one for each word. Then, each of these groups is passed to a reduce function which increments the counters. The final output contains the words and their number of occurrences in the text.

Hadoop [Fou21a] has been among the frameworks that have adopted the map-reduce programming model.

### Dataflow Model

Although the map-reduce processing model is established in parallel computing, it can be restrictive, as computations are only expressed using two operators, map and reduce. An extension of the map-reduce programming model is the dataflow [Tha18]. In the dataflow model, the computations are expressed using a directed acyclic graph. The vertices represent the computational tasks (which can be operators like map,reduce filter, join etc.) and the edges represent the flow of the data across the operators.

Figure 2.2 illustrates an exemplary dataflow graph. In the computation represented by this graph, the input dataset is consumed by two map operators. The result of each map operator is then passed as input to a filter operator. Finally, the results of the two filter operators are aggregated using a join operator.

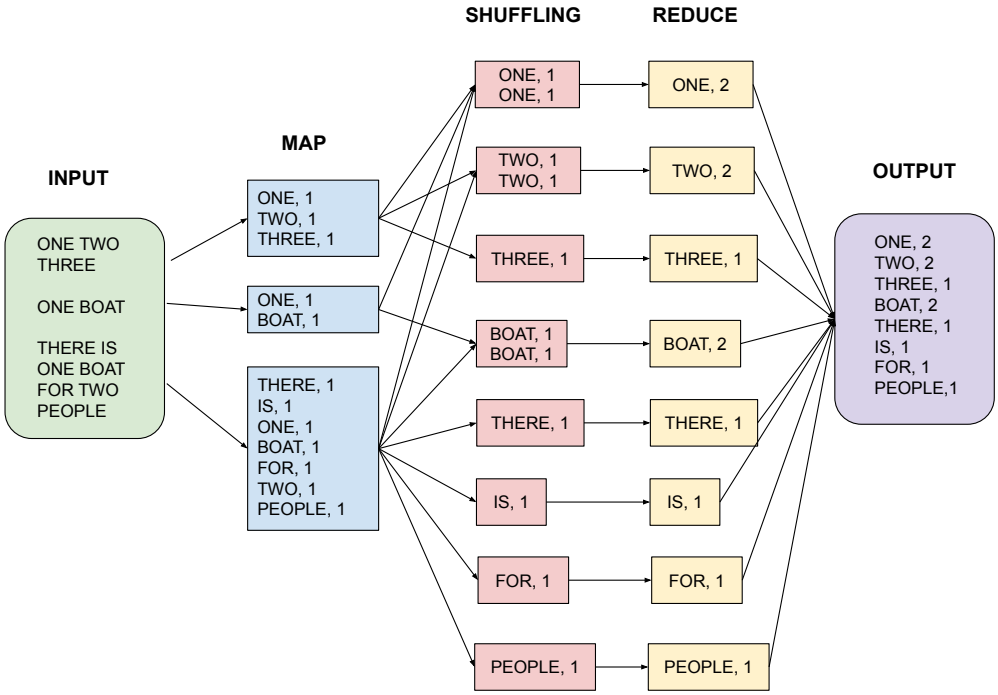


Figure 2.1: A WordCount Map-Reduce Example.

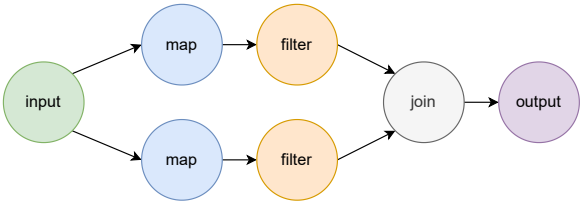


Figure 2.2: A Dataflow Graph Example.

The distributed dataflow system (e.g. Apache Flink) can parallelise the tasks represented in the dataflow graph by creating a parallel version of the initial dataflow graph, with one vertex per parallel task. The edges of the parallel dataflow graph represent the flow of the partitioned data across the parallel instances of the operators. Figure 2.3 presents the parallel version of the dataflow of Figure 2.2, with a degree of parallelism of two.

As illustrated in Figure 2.3, there are operators (e.g. map) that can process data independently, while others (e.g. join) require the whole dataset to be present. The distinction is based on the computation that each operator performs. Parallel instances of a map operator can be applied on random partitions of the dataset, as each element

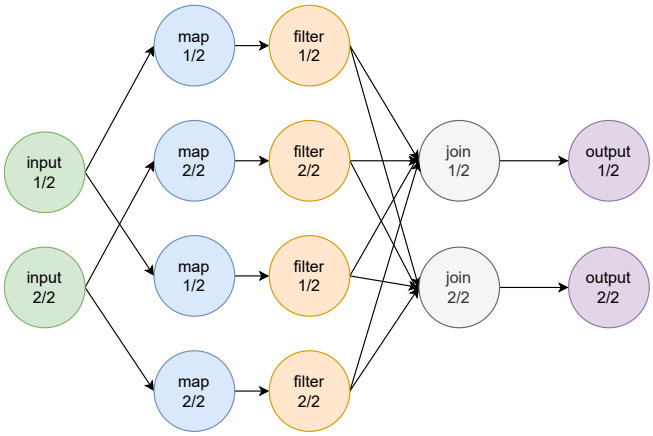


Figure 2.3: A Parallel Dataflow Graph Example.

is processed individually. In contrast, reduce and join operators, merge elements based on a key value, so they require the whole dataset in order to perform the merging correctly. Such operators are known as *pipeline-breaking operators*.

Although pipeline-breaking operators require synchronization, since all previous tasks have to be completed before they can be executed, once the whole dataset is available they can run in parallel just like any other task. However, the data has to be partitioned so that each parallel instance operates on data with the same key value, using a partitioning algorithm.

### 2.1.2 Architecture

Big Data frameworks typically apply the *master-worker* model of execution on a *cluster*. The term cluster refers to a set of computers (*nodes*) that are connected in the same network and communicate to perform a computation. Figure 2.4 illustrates a description of this execution flow. As shown on the left side of this figure, developers express the workflow of their applications using an Application Programming Interface (API) that is provided by the framework (*Client* side). The API contains multiple operators, such as map, reduce, groupBy, etc. The master node of the cluster (*Cluster Manager*) is primarily responsible for monitoring and coordinating the execution among the worker nodes. YARN [VMD+13] is one of the most popular choices for scheduling among Big Data frameworks, however, some frameworks offer additional scheduling mechanisms in standalone mode (e.g. FIFO and Fair Scheduler for Spark [Spa], eager/all at once scheduling for Flink [Zag20] etc.). Finally, each compute node performs the execution assigned to it and returns the results.

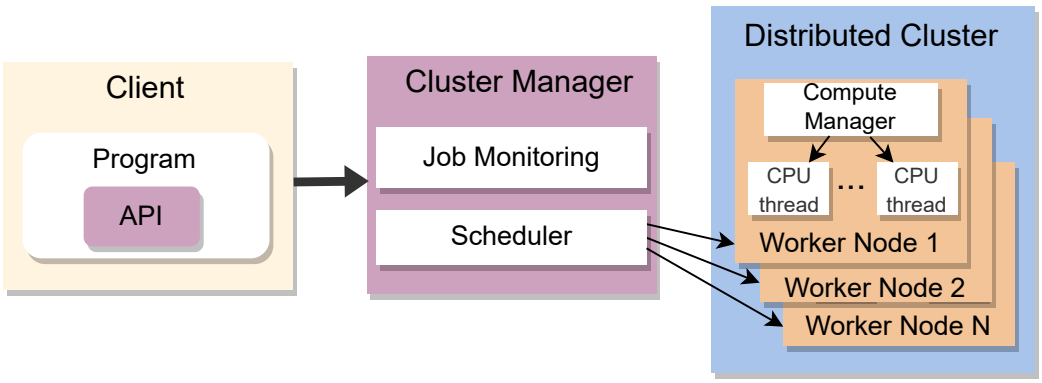


Figure 2.4: Overview of Big Data Frameworks.

### 2.1.3 Defining the Basis of this Thesis and Discussions

The Venn diagram in Figure 2.5 classifies Apache Flink, Apache Spark and Hadoop based on their processing model (computation representation and data processing) and places the work presented in this thesis. As shown in the diagram, the focus of this work is on distributed dataflow systems on batch processing mode. Specifically, the novel techniques for transparent acceleration that will be presented in Chapter 5, were prototyped on Apache Flink on batch processing mode.

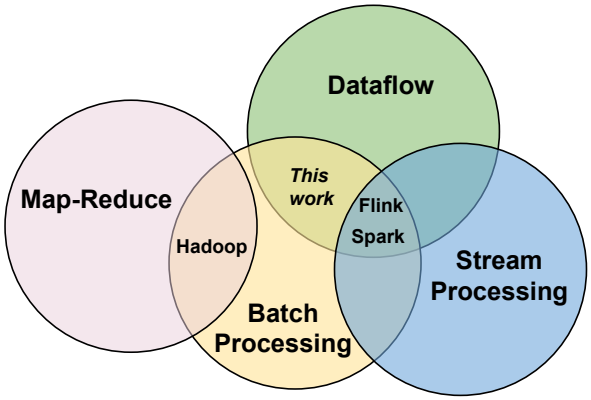


Figure 2.5: Venn Diagram Illustrating the Positioning of this work

Batch processing was chosen as it is a more natural fit for hardware accelerators. In batch processing, the data is bounded, therefore, the dataset can be provided in bulk to hardware devices such as GPUs and FPGAs making it easier to limit underutilization and to gain performance (e.g. by configuring the number of threads suitable for the data load). In contrast, in unbounded data streams, efficiently allocating resources

among heterogeneous hardware is a more challenging task. The data again would have to be split in bulks, to avoid hardware underutilization, however, deciding the bulk size without knowing the full dataset size can be arduous. Nevertheless, enabling hardware acceleration in stream processing could be a rewarding and exciting field for future research. For example, configuring the bulk size in such a way that optimal utilization is obtained, without the data becoming obsolete in the meantime, is not trivial and could offer great value to the community.

## 2.2 Apache Flink

Apache Flink follows the same execution model that was presented in Figure 2.4. In Flink terminology, the Big Data Engine is called *Job Manager* and the Compute Nodes are referred to as *Task Managers*. It is a highly sophisticated system and it consists of multiple layers that ensure fault tolerance, communications handling, and other necessary operations. This subsection focuses on two aspects of Flink which are the most relevant to this work. The first one is the programming model of Flink for batch processing (Dataset API) and the second is how the computation is represented throughout the execution via Flink's dataflow model.

### 2.2.1 DataSet API

Using the DataSet API of Flink, developers specify the transformations (map, reduce, filtering, etc.) that will be applied on their data. Some of these transformation operators, including map and reduce, require a *user-defined function* (*UDF*), which expresses how they will be applied to their input dataset. Since this thesis focuses on map and reduce, from this point on, the word operator will be used interchangeably for these two transformations.

Typically, the Flink UDF is written in a user class that implements an operational interface (e.g., `MapFunction`, `ReduceFunction`, etc.). On the left side of Figure 2.6, the UML for such a user class is presented. Nevertheless, for certain applications, the user function might require additional information (e.g., extra parameters, broadcasted datasets, etc.). In this case, the user-defined class can extend an abstract class instead, which is characterized as *rich* (right side of Figure 2.6). Apart from the user function, rich classes also contain a runtime context, which consists of information such as the

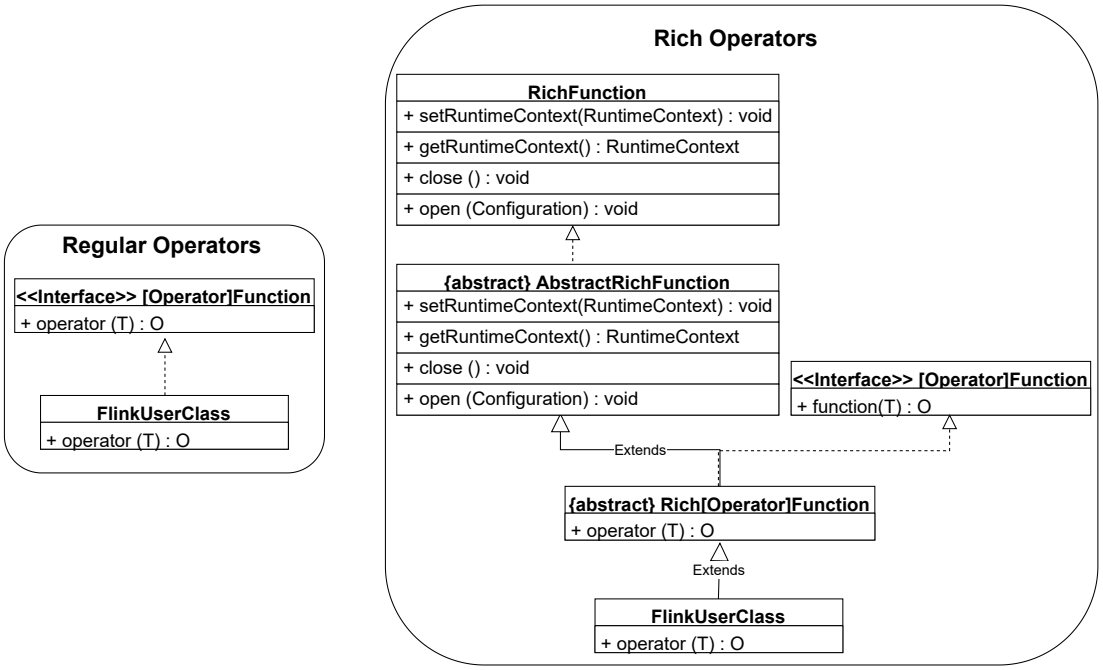


Figure 2.6: UML Representing the Flink API.

degree of parallelism, the additional parameters, the broadcasted datasets etc. Moreover, two auxiliary functions can be implemented by a rich user class in order to setup the context (`open()`) and to perform the cleanup after the computation is completed (`close()`).

### An Example of A Flink Program

Listing 2.1 presents an example of a Flink computation using the DataSet API. First, the input, which can either be read from a file or a *Java Collection* (e.g. ArrayList, LinkedList, etc.), is stored in a DataSet object (lines 2 and 3). Next, two transformations are applied on this data; a map along with a reduction.

As discussed above, Flink provides interfaces and abstract classes that developers can implement or extend to express their computation for each transformation. In this example, the classes that contain the user implementation (`MapUserClass` and `ReduceUserClass`, presented in Listings 2.2 and 2.3) extend the `MapFunction` and `ReduceFunction` interfaces respectively. As shown in line 4 of Listing 2.2, the map operator of the `MapUserClass` receives an input a set of Tuple2 objects and returns Tuple3 objects that consist of two Double fields and one Integer field. Then, for every input Tuple2, the map operator creates a new Tuple3 object, which contains the two

Listing 2.1: DataSet Transformations using the Flink API.

```
1  public static void main (String[] args) {
2      List<Tuple2<Double, Double>> list;
3      DataSet<Tuple2<Double, Double>> input = getInput(list);
4
5      DataSet<Tuple3<Double, Double, Integer>> res = input
6      .map(new MapUserClass())
7      .reduce(new ReduceUserClass());
8
9      res.collect();
10 }
```

fields of the input Tuple and, additionally, the Integer value of 1 (line 5 of Listing 2.2). The reduce operator of the `ReduceUserClass`, as presented in Listing 2.3, receives a set of Tuple3 objects, which consist of two Double fields and one Integer field, and sums up all the Integer fields of the input data set (line 5 of Listing 2.3).

An instance of each of the `MapUserClass` and the `ReduceUserClass` classes is passed as a parameter to the corresponding DataSet transformation function (lines 6 and 7 of Listing 2.1).

Listing 2.2: MapUserClass Implementation.

```
1  public static final class MapUserClass implements MapFunction
       <Tuple2<Double, Double>, Tuple3<Double, Double, Integer>>
       {
2
3      @Override
4      public Tuple3<Double, Double, Integer> map(Tuple2<Double,
           Double> value) {
5          return new Tuple3<>(value.f0, value.f1, 1);
6      }
7
8  }
```

After all the transformations are declared, a function that triggers the execution and indicates how the developer wishes to receive the output data has to be called. This function is known as *sink*. There are many different sink functions, with some of the most common being `collect()`, which returns a list with the results and `print()` that prints the results on the console. In the example of Listing 2.1 the sink that is called is `collect()`, in line 9.

Listing 2.3: MapUserClass Implementation.

```
1 public static class Red implements ReduceFunction <Tuple3<
    Double, Double, Integer>> {
2
3     @Override
4     public Tuple3<Double, Double, Integer> reduce(Tuple3<
        Double, Double, Integer> value1, Tuple3<Double, Double
        , Integer> value2) throws Exception {
5         return new Tuple3<>(value1.f0, value1.f1, value1.f2 +
              value2.f2);
6     }
7 }
```

### 2.2.2 The Representation of a Flink Computation Throughout Execution

As Flink is dataflow system, it represents the computation by creating a graph of dependencies between the operations, in order to distribute them among the compute nodes. This subsection will describe how this graph, named *Execution Graph* in Flink, is derived from the user program.

First of all, following the data sink call, three graphs are constructed on the Client side.

1. The **Plan**: contains a description of all the data sources, data sinks and operators that compose the computation.

2. The **Optimized Plan**: contains the nodes for the input/output data and the operators expressed in the Plan. Each node additionally stores information about how the execution will take place, such as shipping policies that define how data will be distributed to the next operator (e.g. `forward`, `redistribute`, `shuffle` etc.). The nodes of the Optimized Plan do not have a one-to-one relationship with those of the Plan, meaning that, in some cases, new nodes might be introduced (the example presented later illustrates one such case).

3. The **Job Graph**: is a directed acyclic graph that is constructed from the Optimized Plan and it has two different types of nodes. The first category has nodes that represent the operators (*Job Vertices*), whereas the second consists of nodes that represent the intermediate datasets (*Intermediate Datasets*) which are produced by each operator. The Job Graph Generator, using information such as the

shipping strategy, might *chain* some operators together. If two or more operators are chained, it means that the computations they represent will be executed by the same thread, reducing communication costs, unnecessary context switching, serialization and other overheads.

Next, the Job Graph is deployed to the Job Manager to create the Execution Graph. The Execution Graph is essentially the parallel version of the Job Graph. Each Job Vertex is represented in the Execution Graph by an *Execution Job Vertex*. To express the parallelism of the computation, every Execution Job Vertex encapsulates one or more *Execution Vertices*. Each Execution Vertex, therefore, represents one parallel task. Finally, the Intermediate Dataset nodes of the Job Graph are represented in the Execution Graph by *Intermediate Result* nodes, that contain the results of each parallel partition.

The tasks that are described in the Execution Graph are then scheduled to be executed on the Task Managers. The number of parallel tasks that each Task Manager can execute is defined through the *task slots*. The task slots express how the hardware resources will be divided within a Task Manager. For example, if a Task Manager has four task slots, then each will obtain 1/4 of the managed memory for its computation. Moreover, by specifying the number of task slots, the users specify how the subtasks are isolated from each other. For example, if multiple task slots are used, then multiple subtasks share the same JVM instance. If only one task slot is used, then each group of tasks is executed on a separate JVM instance.

Figure 2.7 presents the workflow of a Flink user program, with the graphs discussed above in their corresponding components of the Flink cluster.

### Example

To gain a better understanding of the graphs described above, the dataflow graphs (Plan, Optimized Plan, Job Graph and Execution Graph) that would be generated for the code in Listing 2.1 will be presented.

This example assumes that the parallelism of the map and reduce operators is set to two, meaning that the computation will be split among two parallel Java threads.

The first dataflow graph is the Plan (Figure 2.8). It is a minimal representation of the computation, containing information such as the parallelism of each operator and the execution flow between them. As seen, the Plan consists of four nodes, a node that represents the reading of the input (DataSource), a node for the map function, a node for the reduction and finally one for the output (DataSink).

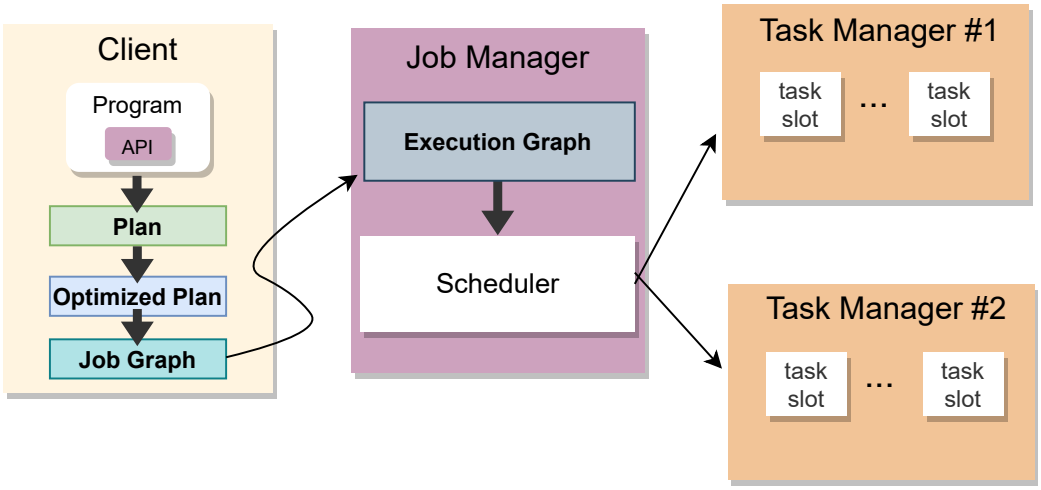


Figure 2.7: An Overview of the Workflow in a Flink Cluster.

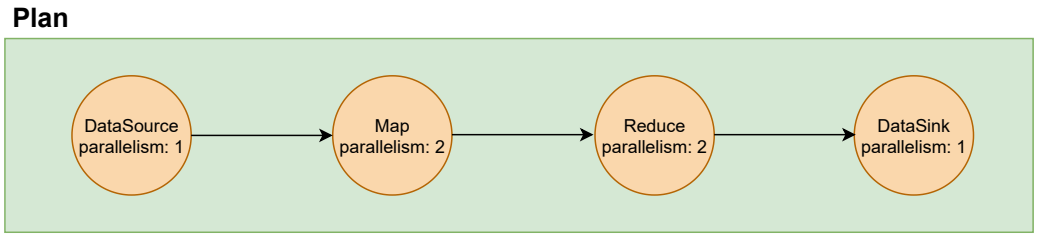


Figure 2.8: The Flink Plan for the Operations in Listing 2.1.

Next, the Optimized Plan (Figure 2.9) is constructed. During the creation of this plan, the operators are analyzed and the distribution strategies are decided. Given the operators of the example and their parallelism, it is assumed that Flink would choose the following distribution strategies. The first one is *Redistribution*, which indicates to the system to partition randomly the dataset among the subtasks. The second is *Forward*, which signifies that the data will be send locally to memory. The last strategy is typically chosen to transfer data between tasks that have the same degree of parallelism. Next, a possible way that these strategies could be assigned among the vertices of the Optimized Plan will be discussed, and the purpose of each vertex will be described.

As shown in Figure 2.9, the Optimized Plan for this example contains five nodes instead of four that the Plan had. The first node is the DataSource, which represents the input source, as in the Plan. The next node expresses the map operator. Since this transformation will be executed by two parallel threads, the input data of this operator will be distributed among each parallel instance (redistribution policy). The

node that follows represents the reduction. The reduction has the same parallelism as the previous task, so in this case the results of the map operator can be forwarded in a one-to-one fashion, from each map subtask to a reduce subtask (forward policy). The next node in the Optimized Plan is an extra reduction node. This node indicates that after the parallel reduction has been executed, a final reduction has to be performed to merge the results of each thread. This is defined by the nature of the computation, since reductions return a single element on Flink. As the parallelism of the two reduction nodes is not the same, the redistribute policy is used to send the data. Finally, the single-threaded reduction forwards the final results to the DataSink node.

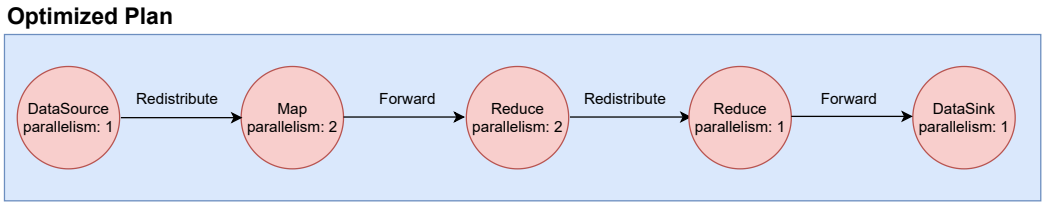


Figure 2.9: The Flink Optimized Plan for the Operations in Listing 2.1.

Using the Optimized Graph described above, the Job Graph (Figure 2.10) is constructed. The Job Graph consists of two types of nodes, the Job Vertices (purple) that represent the operators expressed in the Optimized Plan and the Intermediate Datasets (yellow) that depict the datasets that are produced after each operator. In this example, it is assumed that the map and first reduce can be chained, since they have the same degree of parallelism. This means that each subtask of these two operators will be executed by the same thread, on the same task slot of the Task Manager.

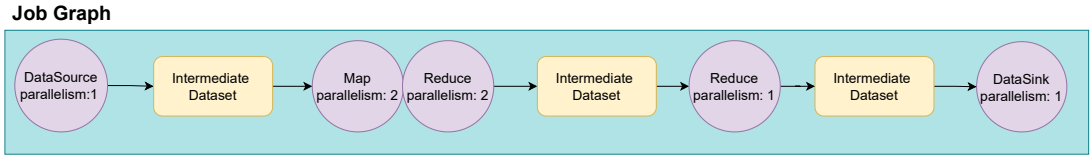


Figure 2.10: The Flink Job Graph for the Operations in Listing 2.1.

Finally, the Job Manager, using the Job Graph, constructs the Execution Graph, as presented in Figure 2.11. As mentioned above, this is a parallel representation of the Job Graph. It consists of Execution Job Vertices and Intermediate Result nodes. Each of these two types of nodes encapsulates their degree of parallelism by having multiple Execution Vertices and Intermediate Result Partitions, respectively.

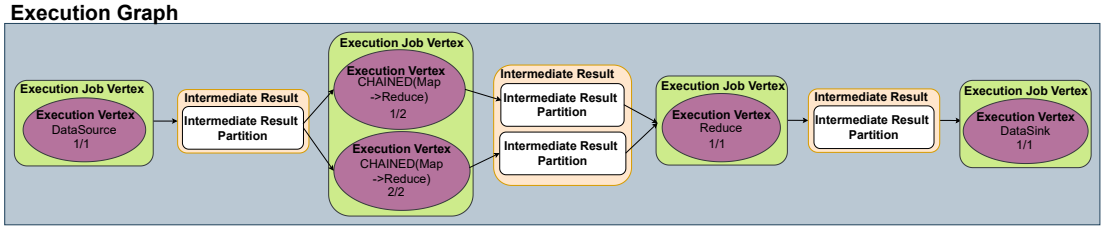


Figure 2.11: The Flink Execution Graph for the Operations in Listing 2.1.

## 2.3 Hardware Accelerators and Heterogeneous Execution

*Hardware accelerators* are devices that appear to be particularly efficient in executing certain tasks. This thesis examines how Big Data applications can gain performance by exploiting two types of accelerator devices, GPUs and FPGAs. The architecture of GPUs differs from the architecture of FPGAs, so different workloads and/or algorithmic patterns perform better on one or the other. For example, GPUs deliver fine-grain execution and data parallelism due to their ability to utilize thousands of threads concurrently, in which physical cores execute the same instructions with different input data items (SIMD) [OHL+08]. On the other hand, FPGAs offer coarse-grain execution and pipeline parallelism, as they can combine various on-device resources (i.e., blocks of memory, registers, logic slices) to compose diverse hardware blocks [CH02]. Therefore, GPUs are suitable for accelerating applications, such as computer vision [FM05, KCR+17] and deep learning (DL) [GCS+, GZYE20, SOV+20], that inherently offer a large number of operations that can execute in parallel; whereas FPGAs are used in applications, such as financial technology [Xil21], for accelerating math operations (e.g., sine and cosine).

The term *heterogeneous execution* refers to computations that take place using different types of processing devices (e.g., CPUs, GPUs, FPGAs, etc.). In order to program heterogeneous hardware accelerators several programming models with different characteristics can be used (e.g., OpenCL [Gro21, SGS10], CUDA [Cor21b], OneAPI [INT22]). The prime goal of these models is to ease programming by exposing a unified way of coding that is applicable to every device type. Typically, the execution of a program is separated in two code segments: (i) the host code that runs on the main CPU; and (ii) the *kernel* code that is offloaded on the hardware co-processors that are connected to the CPUs via PCIe. Through these programming models, developers can explicitly orchestrate the execution of the two code segments in three steps: (i) the

host code copies the input data to the on-device memory (e.g., GPU DRAM, FPGA DRAM); (ii) the kernel code is launched to perform a computation over the input data; and (iii) the result of the computation is copied from the on-device memory to the CPU main memory. Additionally, these programming models allow developers to explicitly utilize the memory hierarchy of a device type in order to increase the performance of memory accesses for compute kernels (e.g., global, local, and private memory on GPUs).

## 2.4 Summary

This chapter introduced the main concepts that constitute the background of this thesis. First, an overview of Big Data frameworks was provided, focusing on their processing model and architecture. The processing models that were presented were the map-reduce (used by Hadoop) and the dataflow (used by Apache Flink and Apache Spark). Then a discussion section followed, that introduced the focus of this thesis, which is distributed dataflow systems in batch processing mode. Furthermore, Apache Flink, which is the distributed dataflow system that was used for prototyping, was presented. Two parts of Flink were discussed in detail, its programming model and the representation of the computation throughout its layers. Finally, some basic terminology regarding the heterogeneous execution and hardware accelerators was established. The next chapter will present the most prominent advances in accelerating map-reduce computations and Big Data applications using hardware accelerators.

# Chapter 3

# Related Work

This chapters presents the related work of this thesis. Specifically, Section 3.1 elaborates on the most notable map-reduce frameworks that use GPUs and/or FPGAs for acceleration, with subsection 3.1.1 providing a critical overview for each of them. Additionally, Section 3.2 presents the current state-of-the-art Big Data Platforms that support execution on devices such as GPUs and FPGAs. This thesis separates these platforms into two categories. The first category consists of frameworks that rely on pre-compiled kernels (subsection 3.2.1) and the second category has platforms that generate the kernels dynamically (subsection 3.2.2). Subsection 3.2.3 reflects on these implementations and evaluates them in terms of their programming overhead and device coverage. Finally, Section 3.3 summarizes the presented frameworks.

## 3.1 Map-Reduce on GPUs and FPGAs

Due to the highly parallel nature of the map-reduce programming model, a lot of research has been conducted on incorporating it to GPU and FPGA programming.

Catanzaro et al. [CSK08] presented a framework that supports the execution of map-reduce functions on GPUs. To use the proposed framework, the developers have to write the map-reduce functions in CUDA and to tune low-level parameters, including the number of threads per block, the number of registers used for every function etc. Glasswing [EHHB14a, EHHB14b] exposes a map-reduce OpenCL API to developers. To optimize performance, the authors proposed a pipeline that coordinates between all the activities involved in the map-reduce work-flow (computation, communication between cluster nodes, memory transfers etc.).

Grex [BK13], is another significant map-reduce framework for GPU execution. In

Grex, the applications are written in CUDA. The developers are responsible for memory management and can specify if the data is read-only or read-write. Additionally, the authors considered several challenges for efficient execution (including optimal data split, memory management, etc.), and introduced the *lazy emit*. The lazy emit is an optimization during which the `values` of `<key, value>` pairs are stored in memory only if the developer indicates that they will be used during the execution. This was implemented based on the observation that, in some cases, only the `keys` are utilized.

MapCG [HCC+10] exposes a C-like map-reduce API to developers. The memory allocations and the communications are handled by the system. Mars [HFL+08] is an optimized framework for GPU execution that provides a C/C++ map-reduce API. MATE-GC [JA12] provides a CUDA map-reduce API. The data management between the host and the device is handled by the developers. Additionally, this framework uses *generalized reductions*. During the generalized reductions, as defined in this work, for each input element, the map and the reduce are performed in a single step, in order to avoid the overheads due to sorting and grouping the data. MGMR [CQJ+13] was developed in CUDA and C++. This framework can target NVIDIA Fermi GPUs. NVIDIA GPUDirect [Cor21a] is used to enable remote GPU memory access without going through CPU memory.

Stuart et al. [SCMO10, SO11] also worked extensively on accelerating map-reduce operations using GPUs. In [SCMO10], a map-reduce volume rendering implementation was proposed, which can be executed on multi-GPU clusters. Then, in [SO11] the same authors presented GPMR, a map-reduce library written in C++ and CUDA. In order to obtain optimal performance, the authors performed several optimizations, such as applying the maps and reductions on large chunks of data to get optimal GPU utilization and overlapping communication with computation. Chen et al. [CA12] and Ji et al. [JM11] also provided a map reduce framework for GPU execution. The main innovation of their work was that they utilized GPU's shared memory for optimal performance.

StreamMR [ELcFS11] is an OpenCL map-reduce framework optimized for AMD GPUs. Soren [MAKA11] is a another novel map-reduce framework for GPU execution. It uses a training dataset to extract monitoring information (i.e. number of `<key, value>` pairs, number of unique keys etc.) which is used for system tuning. The authors also proposed a technique to incrementally combine reduction results and a

mechanism that handles memory overflows. Ravi et al. [RMCA10] created an LLVM-based [LA04] compiler and a runtime to run reductions on GPUs. Their system provides an annotated-C API to the developers. Moreover, the authors evaluated various load distribution schemes between GPUs and CPUs.

SkePU [SI09] and SkelCL [SKG11] are C++ template libraries that target multi-GPU systems. SkelCL generates OpenCL code for a set of skeleton functions (i.e. map, reduce, scan and zip). SkePU exposes an API that consists of a set of macros, which are used to generate CUDA and OpenCL code. Lift [SRD17] is a compiler that generates OpenCL code for algorithmic skeletons which are written in C/C++. To achieve that, it uses an intermediate representation (named *LiftIR*).

Chen et al. [CHA12] introduced an OpenCL map-reduce API that targets integrated CPU-GPU chips. Xie et al. [XKB13] presented Moim, a map-reduce framework that distributes the execution among CPUs and GPUs. It utilizes Thrust [BH12], which is a parallel programming library for CPU and GPU execution. In Moim, the user code has to be written in CUDA. Additionally, the authors of Moim explored how to perform efficient load balancing among the reducers and among the mappers.

Tsoi et al. [TL10] implemented a map-reduce framework which targets clusters with heterogeneous nodes, i.e. nodes that potentially consist of multiple processing elements of different types, such as CPUs, GPUs, and FPGAs. OpenMPI [Pro21] is used to transfer data between the cluster nodes. Furthermore, the kernels for each different type of device have to be provided by the developer. In [YTT+08] the authors presented a C map-reduce library that can run on both GPUs and FPGAs. The functions are manually translated to HyperStreams for FPGA execution and CUDA for GPU execution.

The FPMR framework [SWY+10] enables developers to target FPGAs by providing RTL map and reduce modules. Task scheduling, communication, and data synchronization are performed automatically. Additionally, Choi et al. [CS14] presented a version of KMeans developed specifically for FPGAs using the map-reduce programming model. The map-reduce functions are implemented using FPGA fabrics. The system has a inter-FPGA communication channel to enable data movements between FPGAs and a management layer that performs tasks like handling job requests and monitoring the cluster. Finally, Spatial [KKO+18] is a Domain Specific Language (DSL) and compiler that provides high-level abstractions for FPGAs and CGRAs. The API of Spatial consists of dataflow operators (e.g. Reduce) and methods that developers can implement for memory management.

### 3.1.1 Discussion

Although, the works discussed above, enable developers to take advantage of the map-reduce abstraction of parallelism for hardware acceleration, they focus more on performance and less on programmability. Specifically, in many of the implementations proposed, the developers have to write code in a low-level programming language (e.g. [CSK08, BK13, JA12, CQJ+13, CHA12, SWY+10]). Moreover, in some works, developers have to handle low-level details (e.g. to define the number of registers used [CSK08] or to perform memory management [BK13, JA12, KKO+18]).

Furthermore, most of the works presented are solely focused on GPU execution [CSK08, EHHB14a, EHHB14b, BK13, HCC+10, HFL+08, JA12, CQJ+13, SCMO10, SO11, CA12, JM11, ELcFS11, MAKA11, RMCA10, SI09, SKG11, SRD17], while some of them [CSK08, BK13, JA12, CQJ+13, ELcFS11, XKB13, YTT+08, SO11] can only target devices from specific vendors. In some implementations, even though the proposed frameworks can target both GPUs and FPGAs, they require developers to write and maintain multiple versions of the applications, one for each target device (i.e. [TL10]).

The target audience for all the works discussed above, is developers that are familiar with the architecture and programming model of hardware accelerators. Making heterogeneous devices accessible to high-level developers is one of the main challenges that this thesis strives to overcome.

## 3.2 Big Data Frameworks on GPUs and FPGAs

This section includes the most prominent research conducted on accelerating Big Data Platform using GPUs and/or FPGAs. Table 3.1 summarizes the current state-of-the-art and presents for each implementation whether it fulfills the criteria presented in Section 1.1, specifically: (i) if it leads to code fragmentation, (ii) if the low-level code is generated on-demand or if pre-compiled kernels are required, (iii) if a vendor lock-in is imposed, and finally (iv) what the possible target devices are. As shown in Table 3.1, there is currently no Big Data framework that can target CPUs, GPUs and FPGAs on demand, without fragmenting the codebase or locking on a specific vendor. The framework that will be presented in this thesis will differentiate from the current state-of-the-art, as it will be able to dynamically target CPUs, GPUs and FPGAs without any restrictions or new APIs.

Table 3.1: The related work on hardware acceleration of Big Data frameworks.

| Implementation | Big Data Framework | Code Fragmentation | Code Generation | Vendor lock-in | Device Coverage |
|---|---|---|---|---|---|
| [HCJ17], [LLZC15], [LHWW21], [MT16], [OMM16], [HCJ21], [YSH+16] | Spark | Yes | Pre-compiled | Yes | GPUs (NVIDIA) |
| [GC16], [GC19], [HZK+18], [HWY+16] | Spark | Yes | Pre-compiled | Yes | FPGAs (Xilinx) |
| [SKKS18] | Spark | Yes | Pre-compiled | No | FPGAs |
| [VAA19] | Spark | Yes | Pre-compiled | No | CPUs, GPUs, FPGAs |
| [GS16] | Spark | Yes | On-demand | No | CPUs, GPUs, FPGAs |
| [SCN+15] | Spark | Yes | On-demand | No | CPUs, GPUs, FPGAs, APUs, DSPs |
| [GIS15] | Spark | Yes | On-demand | No | GPUs |
| [CJ15], [RE20] | Spark | Yes | On-demand | Yes | GPUs (NVIDIA) |
| [HCL+15] | Hadoop | Yes | Pre-compiled | No | GPUs |
| [RSA+17], [ZLH+14] | Hadoop | Yes | Pre-compiled | Yes | GPUs (NVIDIA) |
| [AKA12] | Hadoop | Yes | Pre-compiled | Not defined | GPUs |
| [FVCC09] | Hadoop | Yes | Pre-compiled | Yes | GPUs (NVIDIA) |
| [TLHC12] | Hadoop | Yes | Pre-compiled | Yes | CPUs, GPUs (NVIDIA) |
| [LC13], [NMGH15], [NMG+15], [NMH15], [NSH16] | Hadoop | Yes | Pre-compiled | Yes | FPGAs (Xilinx) |
| [GBS13], [GBS16], [LOR12] | Hadoop | Yes | On-demand | No | GPUs |
| [SSE15] | Hadoop | Yes | On-demand | Yes | GPUs (NVIDIA) |
| [CLO+16, Fou22b] | Flink | Yes | Pre-compiled | Yes | GPUs (NVIDIA) |
| [COTL17] | Flink | Yes | Pre-compiled | Yes | CPUs, GPUs (NVIDIA) |
| [CLOL18] | Flink | Yes | On-demand | No | CPUs, GPUs |
| [CXT+15] | Storm | Yes | Pre-compiled | Yes | GPUs (NVIDIA) |
| [WHI+19] | Storm | Yes | Pre-compiled | Yes | FPGAs (Intel) |
| [SBK20] | Ignite | Yes | Pre-compiled | Yes | GPUs (NVIDIA) |
| **This thesis** | **Flink (although transferable to other frameworks)** | **No** | **On-demand** | **No** | **CPUs, GPUs, FPGAs** |

The next two subsections describe in detail the works presented in Table 3.1, classifying them into two categories. Specifically, subsection 3.2.1 presents the implementations that require a pre-compiled kernel to be provided by the developer and subsection 3.2.2 all the works where the code is compiled at runtime. In both subsections, the implementations that are presented, are further categorized depending on their basis framework. Finally, subsection 3.2.3 provides further discussion on the current state-of-the-art in Big Data acceleration, in the context of the criteria presented in Table 3.1

### 3.2.1 OpenCL/CUDA Pre-built kernel

#### Apache Spark

Below are presented some of the most notable efforts of accelerating Spark using pre-built kernels. Hou et al. [HZK+18] proposed using Python *ctypes* on Apache Spark to invoke pre-compiled OpenCL kernels on Xilinx FPGAs. Stamelos et al. [SKKS18] enabled FPGA acceleration on Spark for machine learning applications. In this work, JNI is used for kernel invocations. Ghasemi and Chow [GC16, GC19] also accelerated Spark with FPGAs but, in their framework, the map and reduce functions have to be written in RTL. Ohno et al. [OMM16] augmented Spark workers to launch CUDA kernels using JCUDA for data-intensive operators.

Hong et al. [HCJ21, HCJ17] also worked extensively on enabling GPU execution on Spark. In [HCJ21] the authors augmented Spark with an OpenGL/CUDA API. Additionally, several inter and intra node communication optimizations were deployed and MPI was used to enable direct communication between nodes. In [HCJ17], the authors exposed a Python API on Spark. In this implementation, the GPU kernels are launched using pyCUDA. To further optimize the execution, it was proposed to store the data off-heap, in order to skip the deserialization/deserialization process. Manzi and Tompkins [HCJ21] utilized pyCUDA wrappers to distribute the execution on the GPUs.

In HeteroSpark [LLZC15], the developers provide pre-built CUDA kernels, which are deployed on NVIDIA GPUs using JNI. Blaze [HWY+16] is a framework that uses Hadoop YARN as the resource management tool. In this work, the software and the hardware code implementations are decoupled, meaning that a software developer can utilize an API to indicate the request of performing a computation to an FPGA, while

a hardware expert is responsible for implementing the actual FPGA design. Additionally, this work provides custom serializers/deserializers for primitive data types and provides the functionality for users to implement their own serializers in order to use arbitrary data types.

Chen et al. [CCF+16] also suggest abstracting FPGAs as a service. In their proposed framework, FPGA execution is triggered using JNI. Rathore et al. [RSA+17] presented a real-time stream system that integrates Hadoop with Spark and GPUs. Spark is used for capturing the data and distributing it among the Hadoop nodes and each worker node is equipped with a GPU that executes a pre-compiled CUDA kernel.

ShadowVM [LHWW21] proposes the decoupling of the Spark control plane from the data plane. The control plane consists of a virtual RDD (vRDD) that specifies the pipeline of computations and the dependencies between them, while the data plane contains the actual data and performs the computation on either a CPU, a GPU or both. ShadowVM supports kernel implementations written in CUDA, and the loading of the data does not go through the JVM. Instead, it takes place directly on the data plane via the invocation of CPU native code and system calls. To further improve performance the authors also propose a *passive prefetcher*, that overlaps data transfers with computation and loads data from the main memory to the GPU memory only when they are about to be consumed by the kernel. This was implemented is based on the observation that, in some cases, not the entire dataset is consumed by certain computations (e.g. queries that are not applied on all columns).

Spark-GPU [YSH+16] provides a custom RDD data structure, named *GPU-RDD*, that consists of two interfaces: (i) a standard interface that enables the integration of *GPU-RDD* with existing Spark data flows; and (ii) a block interface that returns the address of the buffered data. The buffered data is stored in the native memory instead of the Java heap. Furthermore, Spark-GPU offers a Scala interface that wraps a kernel implementation written in OpenCL or CUDA by the users. The kernel is invoked via JNI. Another framework that can facilitate the execution offloading from Spark on CPUs, GPUs, or FPGAs, is SparkJNI [VAA19]. This is a Spark plugin that enables developers to submit blank Java user functions that can be used for native execution and integration with user-defined pre-built kernels.

### Apache Flink

This class of research work consists frameworks built on top of Flink that enable GPU acceleration using pre-compiled kernels. Currently, Apache Flink offers developers

the option to run their applications on NVIDIA GPUs. This is achieved by invoking CUDA kernels through JCuda [Hut22b] or by using JCublas [Hut22a] to perform linear algebra operators on vertices and matrices. Developers have to explicitly handle data transfers from the host to device and vice versa using JCuda. Chen et al. [CLO+16] propose GFlink, which allows users to write their code in Java using a specific API of the platform. In this work, developers have to provide the pre-compiled CUDA kernels that will be executed, while also utilizing a particular data structure for defining the input data. Chen et al. [COTL17] present an extension of GFlink that showcases the acceleration of variants of the extreme machine learning algorithm (ELM). This work augments GFlink with a heterogeneous task manager in order to enable efficient hybrid execution between CPUs and GPUs.

### Hadoop

Next, the most prominent efforts on accelerating Hadoop with a user-provided kernel are presented. Pamar [TLHC12] supports the execution on CPUs or GPUs and offloads the execution of functions on GPUs via JCUDA. Users can annotate their code to indicate to which device it should be executed (CPU or GPU). Apart from that, the paper also presents a heterogeneous scheduler which uses a first-come-first-served policy.

Surena [AKA12] uses JNI code to execute the user functions on GPUs and to copy the data (in a byte array form) to the GPU memory. To determine how the computation should be distributed among the CPU and the GPU, the platform monitors the execution on both units by deploying each computation with a small subset of data on them. Then, the rest of the computation is deployed on the hardware unit that resulted in higher performance. Surena uses JNI to deploy the kernels on the GPU. This framework also supports several scheduling optimizations to limit GPU idleness.

Neshatpour et al. [NMGH15, NMH15, NSH16, NMG+15] use Hadoop Streaming to write C-like map/reduce functions which are converted to RTL using the Xilinx Vivado HLS. Zhu et al. [ZLH+14] enable Hadoop users to target GPUs by writing their code on Java and CUDA. Hadoop+ [HCL+15] focuses on managing resources between CPUs and GPUs in heterogeneous execution. This work provides two metrics (execution time and cost) as well as a formula to aid users tuning the migration of the execution between a CPU and a GPU. Nonetheless, users are responsible for providing the implementation of the kernels in OpenCL or CUDA.

Shirahata et al. [SSM10] propose a platform built on top of Hadoop. They present three ways to deploy the user-provided CUDA kernel, Hadoop Streaming, Hadoop

Pipes and JNI but eventually choose Hadoop Pipes. Finally, in Mithra [FVCC09] the developers have to also provide an implementation using Hadoop's API in order to take advantage of the data distribution mechanism of Hadoop and provide the CUDA kernel which is invoked using Hadoop Streaming.

#### Apache Storm & Apache Ignite

The works described below have studied the integration of heterogeneous execution on other distributed frameworks, such as Apache Storm [Fou21c] and Apache Ignite [Fou21b]. G-Storm [CXT+15] targets GPUs via JCUDA to offload PTX kernels provided by the users. F-Storm [WHI+19] requires developers to write their OpenCL kernels which are then passed to the FPGAs via a JNI function call. Finally, Ignite-GPU [SBK20] uses JCUDA to load and execute CUDA kernels on GPUs from Apache Ignite.

### 3.2.2 Runtime Compilation

This subsection presents Big Data frameworks that do not rely on pre-built kernels, but compile the code at runtime instead. Again these frameworks are classified based on their basis platform (i.e., Spark, Flink and Hadoop).

#### Apache Spark

This category consists of Big Data frameworks, implemented using Spark, that can dynamically target heterogeneous devices. Grossman et al. [GIS15] provide support for JVM object types (e.g., `Tuple2`, `SparseVector` and `DenseVector`) on GPUs by modifying Aparapi [apa16] to handle objects as structs in C. In addition, this work provides custom serializers that serialize objects so that their byte layout matches that of the corresponding structs. However, the fields of the serialized objects have to be primitives. As an extension, Grossman and Sarkar [GS16] present SWAT, a system that compiles JVM bytecodes to OpenCL kernels that can execute on hardware accelerators. SWAT also provides support for multi-GPU memory handling by implementing an internal library that performs caching of data on OpenCL devices.

SparkCL [SCN+15] provides a Java API, in which the users have to implement two functions that corespond to how the pre-processing and post-processing of the data should be performed. The system uses a fork of Aparapi, Aparapi Ucores, which supports binary execution flow accelerator types (for FPGAs) and enables execution

on FPGAs, GPUs, APUs and DSPs. Furthermore, Choi and Jeong [CJ15] propose Vispark, a Python-like language that translates the source code to execute on GPUs. The authors introduce an extension of RDD that instead of applying the computation iteratively, it copies the chunk data into the GPU and launches a CUDA kernel that can operate in parallel. Lastly, Apache Spark 3.x is accelerated on NVIDIA GPUs [RE20] using the RAPIDS API [rap21].

#### Apache Flink

Chen et al. [CLOL18] present FlinkCL, which requires developers to use a Java-based API, however they do not need to provide the CUDA kernel. Instead, the system uses Aparapi to JIT compile the Java user functions to OpenCL kernels. Moreover, this work proposes a data mapping scheme to avoid serialization and deserialization between the JVM objects and the OpenCL structs. This scheme allows users to define the preferred memory layout (Structure-of-Arrays or Arrays-of-Structures). Moreover, an auto-tuning partitioning scheme and a dynamic load balancing scheme for GPU-CPU execution are proposed. Some extra optimizations compared to GFlink include the ability to execute the same tasks on both CPUs and GPUs simultaneously, and the memory optimizations regarding the avoidance of transferring intermediate results to main memory.

#### Hadoop

Lastly, heterogeneous frameworks that are based on Hadoop and can generate low-level code on-demand are presented. Okur et al. [LOR12] introduce Hadoop+Aparapi (HAPI), a framework that requires developers to implement three functions: (i) `preprocessing` which makes data GPU-compatible, (ii) `gpu()`, which contains the computation, and (iii) `postprocessing()`, which is responsible for retrieving the results of a kernel. HAPI provides this interface only for mappers. HadoopCL [GBS13] is a framework that employs Aparapi to compile Java map and reduce functions to OpenCL. In addition, extra communication threads are used to maximize the utilization of bandwidth.

HadoopCL2 [GBS16] is another framework that provides a Java map-reduce-combine API. The system uses a modified Aparapi software to target GPUs, which supports dynamic memory management. HadoopCL2 provides a runtime layer that manages memory as well as an offline debugger to check for correctness and performance. Another work on running Hadoop operations on GPUs is HeteroDoop [SSE15]. In this work, the developers have to annotate the code with HeteroDoop directives to enable

source-to-source translation by using the Cetus compiler. This work also proposes a tail scheduling scheme. A GPU-first policy is used until the final tasks are deployed, which are forced on the GPU.

### 3.2.3 Discussion

The research works discussed in the previous subsections share one crucial shortcoming; they all lead to *code fragmentation*. This is self-evident for the implementations that require prebuilt kernels (e.g., [HZK+18, SKKS18, GC16, GC19, CLO+16, Fou22b, COTL17, TLHC12, AKA12]), as the existing programs have to be translated to a low-level representation, be it CUDA, OpenCL or PTX. However, even though solutions that provide dynamic code generation alleviate Big Data developers of the burden to familiarize themselves of the programming language and architecture of each target device, the existing programs still have to be rewritten to adhere to the API of each of these frameworks. Therefore, utilizing these systems leads once again to fragmented code bases.

Additionally, some implementations can only target one type of hardware accelerator (i.e. only GPUs [HCJ21, HCJ17, LLZC15, LHWW21, HCJ21, OMM16, RE20, CJ15, RSA+17, FVCC09, SSE15, ZLH+14, CLO+16, Fou22b, CXT+15, SBK20, YSH+16, HCL+15, AKA12, GBS13, GBS16, LOR12] or only FPGAs [SKKS18, WHI+19, GC16, GC19, LC13, NMGH15, NMH15, NSH16, NMG+15]), which is restrictive, as these frameworks cannot harvest the full potential of heterogeneous computing. Making matters worse, some works can only target devices from specific vendors (i.e. only NVIDIA GPUs [HCJ21, HCJ17, LLZC15, LHWW21, HCJ21, OMM16, RE20, CJ15, RSA+17, FVCC09, SSE15, ZLH+14, CLO+16, Fou22b, CXT+15, SBK20] or only Xilinx [GC16, GC19, LC13, NMGH15, NMH15, NSH16, NMG+15] or Intel [WHI+19] FPGAs), leading to *vendor lock-in*.

By studying the current state-of-the-art, it becomes evident that there is a big gap in Big Data acceleration that needs to be filled, since there is currently no uniform solution for fully transparent and hardware-agnostic acceleration of Big Data applications. This thesis strives to fill this gap by investigating the main challenges in integrating heterogeneous accelerators on Big Data frameworks (Chapter 4) and by proposing a set of techniques that tackle these challenges and enable automatic acceleration of Big Data applications on both GPUs and FPGAs, without breaking the programming norms or requiring any user intervention (Chapter 5).

## 3.3 Summary

This chapter presented efforts to offload map-reduce computations on GPUs and FPGAs as well as the most notable advancements on enabling heterogeneous execution for Big Data execution. The map reduce frameworks that were presented rely on the developers to provide an implementation in a low-level programming language and require expertise in the architectural details of each target device. Regarding accelerated Big Data Frameworks, which use the map-reduce programming model, the current literature consists of two classes of works.

In the first category, belong platforms that require from developers to write their implementation in OpenCL or CUDA. The disadvantages of this approach are the same as above. The second category consists of platforms that enable allow developers to express their implementation in a high-level language and use a compilers (mostly Aparapi) to deploy the computation on different devices. However, in all of these approaches the developers have to write their implementation in an API that is enforced by the platform leading to code fragmentation. The proposed framework of this thesis, which will be presented in Chapter 5, goes beyond the state-of-the-art, as it does not require from the developers neither to write their code using a new API nor to provide a CUDA or OpenCL implementation. The next section will present the challenges that were identified when trying to develop the proposed platform.

# Chapter 4

# Understanding the Challenges

As explained in the related work presented in Chapter 3, to enable GPU and/or FPGA execution it is necessary to provide a low-level representation of the code, either by writing the kernel by hand or by generating a kernel via JIT compilation. Providing an efficient pre-compiled kernel implementation in languages like OpenCL or CUDA requires a deep understanding of the architectural details of each target device. Moreover, multiple versions of the kernels have to be provided to allow targeting different accelerators. Naturally, this leads to having a code-base that is difficult to maintain and to expand, since a lot of effort is required to support new use cases.

Using a tool that can directly target hardware accelerators from a high-level programming language alleviates these issues. Looking at the current state-of-the-art, the most popular framework for JIT compilation of Big Data applications to low-level code is Aparapi. However, Aparapi cannot target FPGAs and is limited to multi-core systems and GPUs instead.

An alternative to Aparapi is TornadoVM. TornadoVM has being gaining traction over the last few years with plenty of research revolving around it [PFSK21, PMF+21, BSFK22, PFS+21, SOV+20]). In contrast to Aparapi, it provides support for multiple backends (OpenCL, SPIR-V, PTX) and can target FPGAs, which is crucial for the purposes of the proposed implementation. Moreover, it offers several features which could be used to optimize acceleration of Big Data applications in future works. One of these features is its ability to perform live mitigation of tasks on the best device based on specific policies [FPZ+19]. Consequently, TornadoVM was chosen for the implementation of the proposed heterogeneous Big Data platform.

This chapter provides an introduction of TornadoVM and explores the feasibility of integrating it with Flink. Specifically, Section 4.1 presents the architecture and API

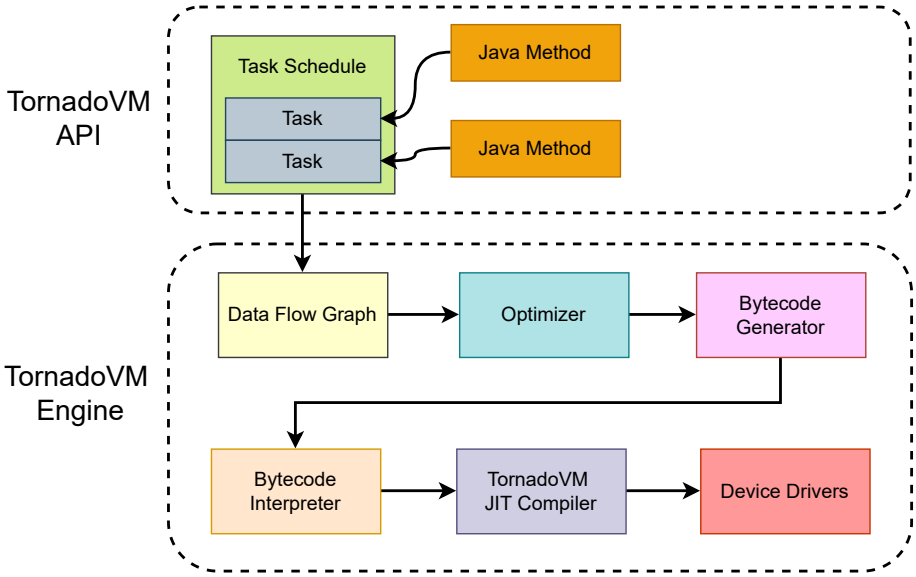


Figure 4.1: TornadoVM Execution Flow.

of TornadoVM. Section 4.2 identifies the incompatibilities between Flink and TornadoVM and highlights the challenges of integrating the two. Section 4.3 presents an initial, naive integration of Flink and TornadoVM. This first version of the integration does not have much novelty, as it relies on techniques used by the current state-of-the-art (i.e. providing a new API). However, developing it served two purposes. The first was to prove that enabling Flink to dynamically target GPUs and FPGA via TornadoVM is attainable. The second was to gain the insight necessary to develop the final platform, which can seamlessly offload Big Data computations on heterogeneous devices in a user-agnostic manner (presented in Chapter 5). Section 4.4 pinpoints the limitations of this initial system and identifies the gaps that need to be filled to develop the end-goal platform. Finally, Section 4.5 summarizes the contents of this chapter.

## 4.1 TornadoVM

TornadoVM is a plugin to OpenJDK that enables programmers to dynamically offload and run Java programs on multi-device platforms. As depicted in Figure 4.1, it consists of a specialized Java API and a runtime engine that analyzes the user programs, compiles them with a heterogeneous JIT compiler and deploys them for execution.

**TornadoVM API:** The API exposed by TornadoVM is *task-based*, meaning that the methods to be accelerated are written in tasks. The Java programs that can be accelerated with TornadoVM have to use primitive types, since Java objects are not supported (with the exception of some specific types provided by the framework). Moreover, the API exposes two annotations that can be used. The first is `@Parallel`, which is used to indicate that there are no dependencies in the loop code and, therefore, that it can be parallelized. The second annotation is `@Reduce`, which marks the variable that will store the results of a reduction. This serves as a hint to the TornadoVM compiler, since reductions have dependencies that need to be taken into account during the parallelization. The TornadoVM tasks are encapsulated in *Task Schedules*. If multiple tasks are grouped in the same Task Schedule then the runtime can analyze the dependencies between them and perform data transfer optimizations automatically.

Listing 4.1: Computations Using TornadoVM API.

```
1  public class Computation {
2      private static void mult(int[] in1, int[] in2, int[] out)
           {
3          for (@Parallel int i = 0; i < out.length; i++) {
4              out[i] = in1[i] * in2[i];
5          }
6      }
7
8      private static void sum(int[] in, @Reduce int[] out) {
9          for (@Parallel int i = 0; i < out.length; i++) {
10             out[i] += in[i];
11         }
12     }
13
14     public static void main(int[] in1, int[] in2, int[] tmp,
           int[] out) {
15         TaskSchedule ts = new TaskSchedule("s0")
16           .task("t0", Computation::mult, in1, in2, tmp)
17           .task("t1", Computation::sum, tmp, out)
18           .streamOut(out)
19           .execute();
20     }
21 }
```

Listing 4.1 provides an example of a Java program written with TornadoVM's API. There are two Java methods in this example. The first one, in lines 2-6, performs a multiplication of two `int` vectors. The second method in lines 8-12 sums up the results

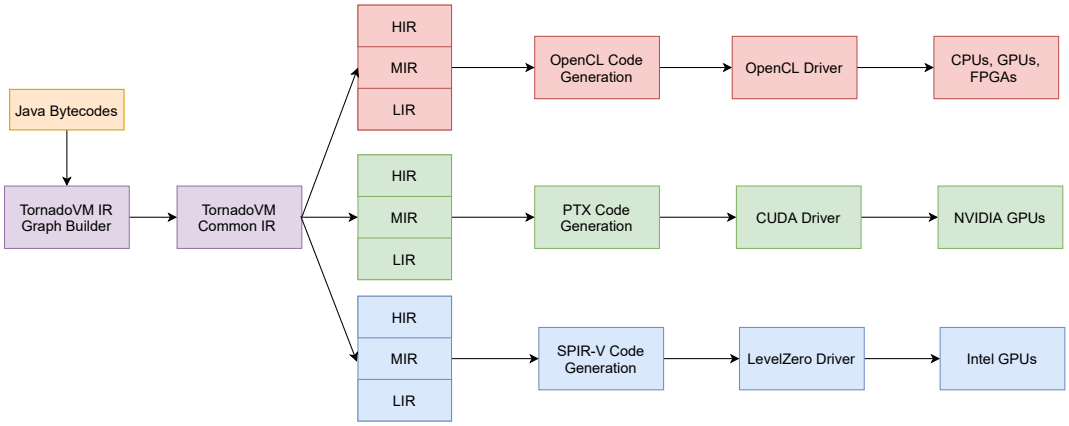


Figure 4.2: TornadoVM Compiler Overview.

of the first function into a single value. As this function essentially performs a reduction, the `@Reduce` is used to annotate the output array (line 8). Both methods use the `@Parallel` annotation in their for-loops (lines 3 and 9), to indicate to the TornadoVM compiler that the enclosed code is a candidate for parallelization. In line 15, a Task Schedule which contains two tasks, each for every Java method, is declared. Both the Task Schedule and each task have a String identifier (`"s0"` for the Task Schedule, `"t0"` for the first task and `"t1"` for the second task). The purpose of these identifiers is so that they can be referenced at runtime. The `streamOut` in line 18 indicates to TornadoVM that the `out` array has to be copied from the device memory to the host memory. Finally, the `execute` command in line 19 triggers the execution of the Task Schedule.

**TornadoVM Engine:** Next, using the Task Schedule, a data-flow graph is constructed, which is passed to the *TornadoVM Optimizer* to identify any read-write data dependencies between the tasks of the Task Schedule. This analysis is performed mainly for two reasons. The first is to prevent unnecessary data transfers from the device to the host and vice-versa. The second is to ensure that the scheduling order will respect any read-write dependencies [CFP+18b]. For instance, in the example of Listing 4.1, the results of the task `"t0"` do not have to be copied to host memory. Instead, they can persist in the device memory to be used directly as an input for the task `"t1"`. Moreover, since there is a read after write (RAW) dependency, TornadoVM ensures that the second task will not be executed before the first.

Following the dependency analysis of the Optimizer, the TornadoVM bytecodes [FPZ+19] are generated and interpreted to produce an *Intermediate Representation* for

each Java method in a task. TornadoVM extends GraalVM [WWS10, WWW+13] for heterogeneous compilation. Specifically, the Graal IR [DWS+13] is expanded with information essential for parallel execution (e.g. whether global, shared or local memory is used, which loop is marked with the annotation `@Parallel` etc.) to build a common TornadoVM IR. Then, TornadoVM, following the three-tier optimization model of Graal, includes phases in each tier (high, mid and low) to construct an optimized IR per target architecture. Finally, specialized code (OpenCL C, PTX or SPIR-V binary) is generated and sent to the corresponding driver to be dispatched for execution. Figure 4.2 summarizes the compilation process explained above.

## 4.2 Identifying the Incompatibilities

Integrating Flink with TornadoVM would enable the former to target a plethora of heterogeneous devices transparently. Nevertheless, there are some key incompatibilities between the two frameworks that have to be dealt with to make this integration possible.

Regarding their programming models, the API of Flink (presented in Chapter 2, section 2.2) differs from that of TornadoVM in the following ways. Firstly, in the TornadoVM user methods, the parallelism is expressed explicitly (with the for-loop inside the method). However, the parallelism of Flink is implicit, since user specifies the computation that will be applied per element. Secondly, Flink operates on Java objects, while TornadoVM only supports primitive types.

To highlight these differences, Listing 4.3 illustrates how a user function that is implemented using Flink's API (labeled (a)) would have to be rewritten to be compatible with TornadoVM's API (labeled (b)). The TornadoVM-compatible method has as input two `float` arrays, since the `Tuple2` type is not supported. Additionally, a for-loop with the `@Parallel` annotation surrounds the computation to indicate that this code is a candidate for parallel execution. Finally, in (b), the array that will store the results is passed as a function argument, and the return type is `void` instead of `Float`.

However, these are not the only incompatibilities between Flink TornadoVM. As mentioned above, Flink executes its operations in a fine-grained way. This means that, when a Flink computation is scheduled to the Task Manager, the pipeline of operators is defined to be executed element by element. This flow of execution allows Flink to have fine-grain control over checkpoints and failures. Nevertheless, this does not agree with the execution model of TornadoVM, which is coarse-grained. Last but not least,

(a)
```
public Float map (Tuple2<Float, Float> in) {
    return in.f0 + in.f1;
}
```

(b)
```
public void map (float[] in1, float[] in2, float[] out) {
    for (@Parallel int i = 0; i < out.lenght; i++) {
        out[i] = in1[i] + in2[i];
    }
}
```

Figure 4.3: A Flink User Function Written with the API of TornadoVM.

since Java objects are not supported by TornadoVM, the data has to be converted from an object representation to primitive types in a process known as *marshalling*.

Therefore, the challenges to overcome in order to integrate the two frameworks can be summarized as follows:

- **#1 Challenge:** The APIs of TornadoVM and Flink are not compatible. TornadoVM exposes a task-based API, while, in Flink, the computations are expressed in the context of the DataSet, as explained in 2.2.1. Additionally, the parallelism is expressed on a different level through the two APIs. Specifically, in TornadoVM the parallelism is explicit, contrary to Flink, where the parallelism is implicit.

- **#2 Challenge:** Flink relies heavily on Java objects but TornadoVM supports only primitive types.

- **#3 Challenge:** Flink provides a fine-grained model of execution which does not match that of TornadoVM.

Even though, initially, it might seem that the challenges described above are imposed only due to the specific frameworks of choice, this is actually not the case. These challenges stem from the design decisions that were followed when these frameworks were developed, so as to better fulfill their objectives. Similar design principles are shared among their equivalent systems as well.

Specifically, Flink was developed to execute large applications on CPU clusters in a scale-out manner. Consequently, both its API and execution model were designed with that consideration. A fine-grained model of computation was deemed suitable for

several reasons, such as that it is efficient for high-throughput devices (i.e. CPUs), it makes recovery from failures easier and so on. Since essentially all Big Data platforms share the same objectives as Flink, these design decisions are also followed by other frameworks, such as Spark. Furthermore, most Big Data platforms expose APIs in high-level programming languages, with Java being a quite common choice, so Java objects are used frequently in Big Data applications. Thus, even if a different Big Data framework were selected, the challenges of integrating it with TornadoVM, or any other similar framework, would be equivalent.

Similarly, the limitations and design decisions of TornadoVM are actually imposed by its target devices. When executing a kernel on hardware accelerators, the size of the dataset has to be known in advance to determine the parallelism of the computation, i.e the number of threads with which the kernel will be launched. Moreover, without knowing the exact size of the input dataset it is difficult to decide if an application is actually a good candidate for heterogeneous execution. For instance, if a very light computation is deployed on a GPU, the device will be underutilized and, thus, no performance will be gained. Additionally, Java object allocation and management is not supported on accelerators such as GPUs and FPGAs, so in reality, this restriction is also not imposed by TornadoVM but by the target platforms (and for that reasons it is shared among other heterogeneous frameworks as well, such as Aparapi).

Hence, a more generic description of the challenges discussed above could be the following:

- **#1 Challenge:** How to enable the execution of applications that are written using a specific API of a Big Data framework on GPUs and FPGAs.
- **#2 Challenge:** How to handle Java object types on GPUs and FPGAs.
- **#3 Challenge:** How to use a coarse-grained model of execution on a Big Data platform, while developers write their code in a fine-grained model.

Figure 4.4 presents a high-level overview of the challenges described above. In the example illustrated in the figure, a Big Data framework deploys the execution on two worker nodes. The first worker targets a CPU and the second a hardware accelerator (i.e. GPU or FPGA). The first worker node receives the task, and executes the user code on the CPU. The code is executed in a fine-grained way, for each element of the input dataset. However, the second worker cannot execute the code on the available hardware because it is written in a high-level language (challenges 1 and 2). Additionally, launching the execution on a hardware accelerator on an element-to-element

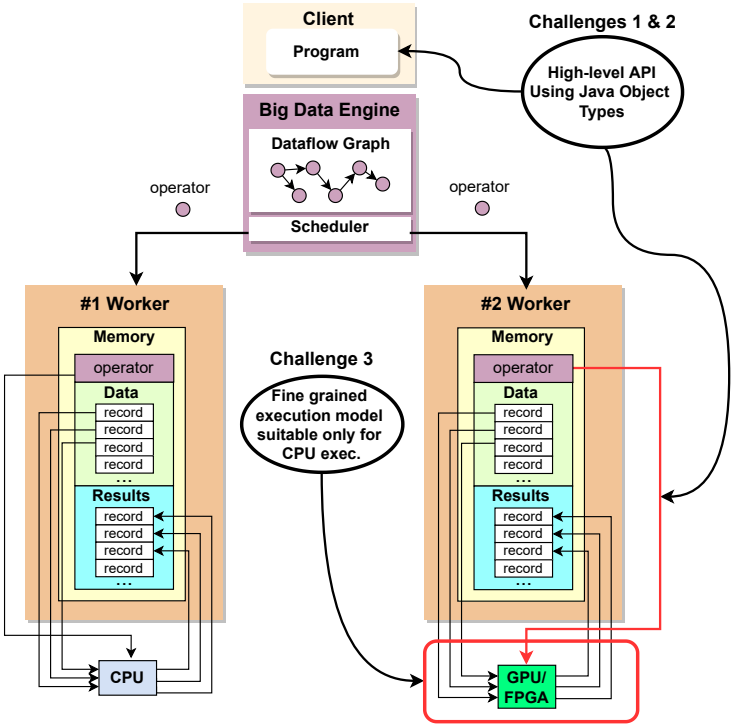


Figure 4.4: Challenges of Enabling Heterogeneous Execution on Big Data Frameworks.

basis (i.e. following the fine-grained model of execution) would severely impact performance, as the costs of initializing the device and copying the data would strip any performance gains (challenge 3).

## 4.3 Enabling Heterogeneous Execution: A Prototype

To verify that these, in fact, are the incompatibilities between the two frameworks, a quick integration of Flink and TornadoVM was implemented as proof-of-concept. Figure 4.5 illustrates the extensions that were applied on Flink (colored green) to construct this prototype. Subsections 4.3.1 and 4.3.2 will provide a description of how each of these extensions handles the aforementioned challenges.

### 4.3.1 Extending the API of Flink

In this initial version of the integration, the API of Flink was extended, as demonstrated on the Client component of Figure 4.5. Specifically, a new set of classes and interfaces was provided to bridge the gap between the API of Flink and that of TornadoVM. The

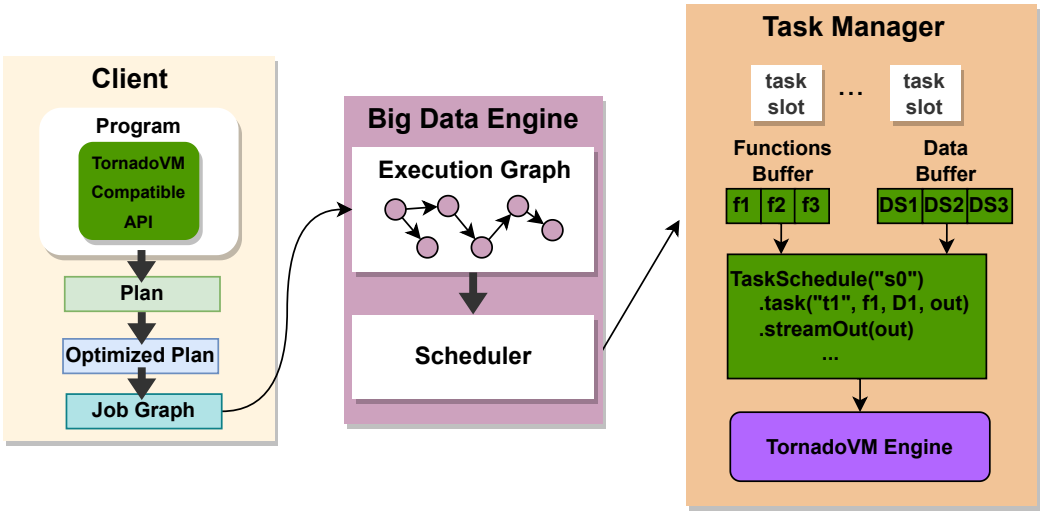


Figure 4.5: The First Version of the Flink-TornadoVM Integration.

UML diagram in Figure 4.6 (left side) presents a new class hierarchy for TornadoVM-compatible Flink operators, while on the right side of Figure 4.6, is the class hierarchy of the Flink operators as presented in 2.2.1. The purple components of the left-side UML diagram shows the extensions over the existing operator interfaces of Flink. For simplicity, in the discussed UMLs, `p[]` represents primitive arrays and `[Operator]` is used to represent either a map or a reduce operator, as these were the ones mainly studied in this thesis.

Specifically, for each operator, a new interface called `TornadoFunction` is provided with a set of typed and non-generic new methods, named `tornadoOperator`. The abstract class `TornadoFunctionBase` implements the `tornadoOperator` methods with a common template. Therefore, if a user class extends the `TornadoFunctionBase`, the code of the application can be written - in a TornadoVM-compatible way - inside the `compute` function that is exposed. Note that the whole stack of Flink operates under the assumption that user functions are applied on Java objects, not on primitives. Therefore, the DataSets are declared as in the programs written in the original Flink version.

### 4.3.2 Invoking Execution

To make the execution flow of Flink compatible with TornadoVM the following approach is used. Whenever a task is deployed to the Task Manager, its data (i.e. its function if it is an operational task, or its input dataset if is it a DataSource task) is

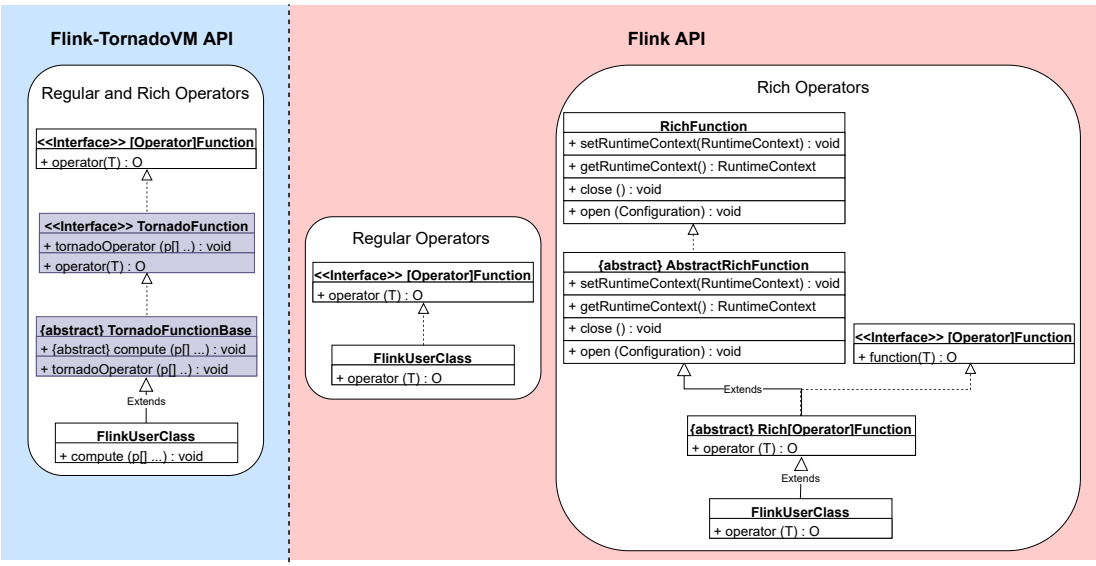


Figure 4.6: UMLs for the Flink-TornadoVM API and the Flink API.

stored in internal buffers - which in Figure 4.5 are marked as *Function Buffers* and *Data Buffers*. When the data sink task is deployed, the heterogeneous execution takes place. The input datasets stored in the internal buffers are converted to primitive arrays and the Task Schedules for the user functions are created and executed. Finally, the results of execution are converted to the object type that is expected by Flink and are collected. Listing 4.2 illustrates this process for a program with a single Flink operator. If the deployed task is an operational task or a data source task, its information is stored. Otherwise, if the task is a `DataSinkTask`, the input dataset is converted to primitive arrays (line 6) and the Task Schedule is created and executed (lines 7-10). The results of the Task Schedule are then converted to Objects (line 11) and are passed to the output collector (lines 12-14).

## 4.4 Limitations of The Initial Integrated Platform

As mentioned before, the version of the Flink-TornadoVM integration presented in this chapter is very close to the current state of the art. A new API is exposed to the developers, meaning that they need to be familiar with the programming model of TornadoVM and the limitations of it. Moreover, the classes and interfaces provided, have to be extended for every new benchmark to contain methods with signatures that have the required input and output arguments. Therefore, this implementation is difficult to maintain and restricts greatly the programmability.

Listing 4.2: Computations Using the TornadoVM API.

```
public class Task implements Runnable {
    run() {
        ...
        if (this instanceof DataSinkTask) {
            float[] in, in2, out;
            marshalling(data, in, in2, out);
            TaskSchedule ts = new TaskSchedule("s0")
                .task("t0", compute, in, in2, out)
                .streamOut(out)
                .execute();
            List<T> results = unmarshalling(out);
            for (T record : results) {
                collector(record);
            }
        } else {
            // store input data and functions in internal
               buffers
        }
    }
}
```

In Table 4.1, this integrated framework is evaluated using the criteria that were deemed essential for a modern heterogeneous Big Data platform, as they were described in Chapter 3. It is evident that the main challenge to overcome in order to construct a hardware-agnostic Big Data platform, is to lift any programming restrictions that currently exist in this version. Specifically, the gap between the two APIs needs to be filled without any user intervention and object type support has to be provided, to enable the heterogeneous execution of existing applications.

Table 4.1: The criteria that the prototype fulfills.

| Implementation | Big Data Framework | Code Fragmentation | Code Generation | Vendor lock-in | Device Coverage |
|---|---|---|---|---|---|
| Flink-TornadoVM Integration-v1 | Flink | **Yes** | **On-demand** | **No** | **CPUs, GPUs, FPGAs** |

Moreover, in this version, all the computations are performed when the data sink task is deployed, in order to change the granularity of the execution. However, this approach is not suitable for distributed execution, during which it should be possible to split the computational tasks among different Task Manager nodes, equipped with various hardware resources.

## 4.5 Summary

This chapter introduced the main challenges in accelerating Big Data applications with heterogeneous devices. Section 4.1 presented TornadoVM, which is the framework that is used in this work to enable heterogeneous execution of Big Data applications. Section 4.2 identified the main challenges in developing a heterogeneous Big Data platform. As it was discussed in this section, the obstacles to overcome are independent of the platforms of choice. Then, Section 4.3 presented a naive approach of integrating Flink with TornadoVM. This development had mostly educational purposes, as it verified that the connection of the two frameworks is feasible and that the challenges identified were correct. Finally, Section 4.4 highlighted the limitations of the presented integration and illuminated the main challenges for developing the vision platform. In the next chapter a set of novel techniques that were used to dynamically offload and accelerate existing Big Data applications without any user intervention are presented.

# Chapter 5

# Seamless Heterogeneous Execution on Big Data Frameworks

This chapter proposes a set of techniques to enable the transparent execution of Big Data applications on heterogeneous hardware accelerators, i.e., CPUs, GPUs and FPGAs. The methodology that was followed will be presented in the context of Flink and TornadoVM, however, with slight modifications, it could be applied to other Big Data frameworks and heterogeneous compilers as well.

In the platform presented in this chapter, the challenges discussed in Section 4.2 are addressed using three modules. First of all, regarding the API incompatibilities between Flink and TornadoVM, a *code morphing module* is introduced, which is responsible for transforming the Flink UDF into a TornadoVM-compatible function. Secondly, to change the granularity of execution, the data is extracted in bulk from the Flink serialization buffers by a *data morphing module*. This module is also responsible for performing the essential transformations on the data to turn it into a format that is compatible with hardware accelerators. Finally, a *code generation module* is responsible for dynamically JIT-compiling the user code for heterogeneous devices.

The proposed platform, containing the aforementioned modules, is presented in Figure 5.1. Each extension is marked with a number, that indicates the order in which it is invoked during heterogeneous execution as well as a label that states the section in which it will be described. Specifically, Section 5.1 introduces the code morphing module, which receives as input the Flink user function and transforms it into a TornadoVM-compatible function. Section 5.2 presents how the data morphing module extracts the input data from the serialization buffers and manipulates it to turn it into a format compatible for heterogeneous accelerators. Section 5.3 presents the dynamic

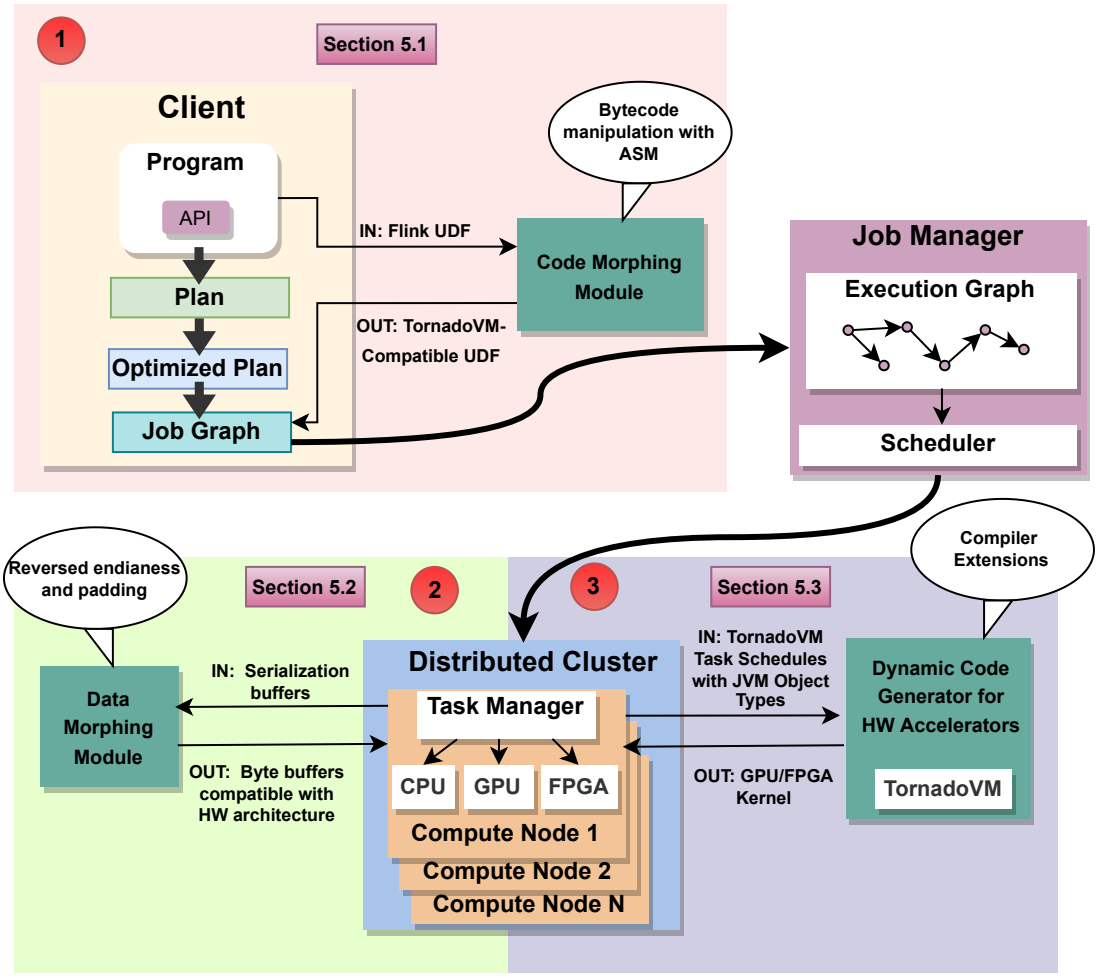


Figure 5.1: Proposed Heterogeneous Big Data Framework.

code generator, which contains an extended version of TornadoVM to enable transparent compilation. Each of these three sections follows the same structure. First, an high-level overview of the module presented. Then, any essential background information is provided. Finally, the details of the implementation are discussed. Following, Section 5.4 elaborates on how the proposed systems handles operators that are not at the moment supported for heterogeneous execution. Finally, Section 5.5 provides an overview of the contributions presented in this chapter and how the proposed framework compares with the current state-of-the-art.

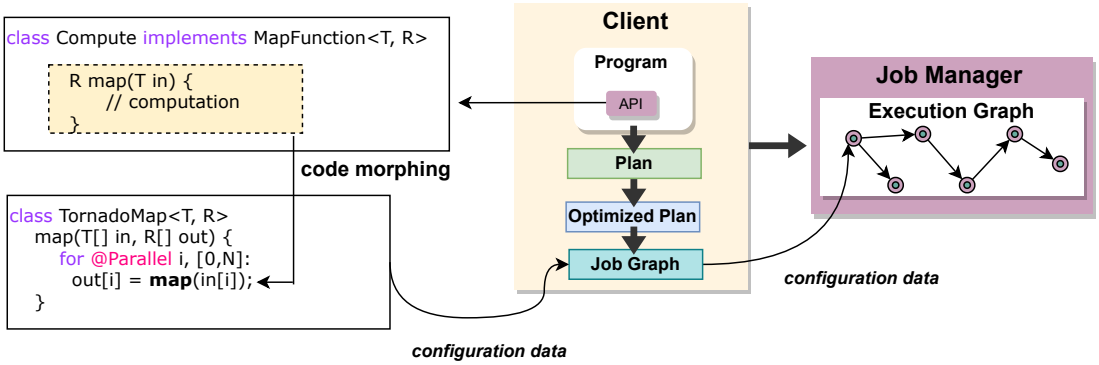


Figure 5.2: An Overview of the Code Morphing Module.

## 5.1 Code Morphing

As explained in Chapter 4, even though both Flink and TornadoVM have Java APIs, there are some key differences between the user functions that are executed on each system. The first version of the integration followed a naive approach to address this challenge, which was to extend the API of Flink so that developers can express their computations in a TornadoVM-compatible way. However, changing the API of Flink has a number of disadvantages.

First of all, this technique reduces usability, since Big Data developers have to be familiar with TornadoVM and its API. Secondly, the codebase gets fragmented, as existing programs have to be rewritten with this new API. Finally, this API does not allow developers to write their applications using Java objects, which are extremely significant in OOP; therefore it is very limiting.

To alleviate these drawbacks, a code morphing module is employed, that dynamically adapts Flink user functions to TornadoVM-compatible code via on-the-fly bytecode rewriting. An overview of this module is presented in Figure 5.2. As shown in Figure 5.2, the code morphing module receives the Flink user function and transforms it into a method that is compatible with the API of TornadoVM. This transformation is performed using ASM [Obj22]. ASM is a Java bytecode manipulation framework that is used for the generation, transformation, or analysis of Java classes. An ASM-generated class is stored in a byte array, which can either be loaded for immediate use or it can be written in a file for future executions. The proposed implementation goes for the former option, therefore, the byte array is transferred to the Task Manager to be dynamically loaded. The way that this generated class reaches the Task Manager is by taking advantage of the Flink configuration data. Specifically, after the class

transformation is completed, the byte array containing the user class is stored in the configuration data of the Job Vertices. As this configuration data is transferred by Flink from the Job Graph to the Execution Graph and eventually the to the Task Managers, the altered user function reaches the Flink driver class (e.g. MapDriver, ChainedMapDriver, ReduceDriver, etc.) that has been assigned to execute the operator. At this point, it can be loaded using a classloader instance and passed to a TornadoVM Task Schedule for heterogeneous execution.

The next subsections will provide background information about the framework used for class bytecode manipulation and details about the internals of the code morphing module. Specifically, subsection 5.1.1 will briefly introduce ASM and subsection 5.1.2 will explain how ASM was used in this context.

### 5.1.1 ASM

ASM is a widely used bytecode manipulation library that operates on compiled classes. Its API is based on the *Visitor* design pattern and consists of three core components, a *ClassVisitor*, a *ClassReader* and a *ClassWriter*.

The abstract class ClassVisitor is the basis of this API. It contains visitor functions for each of the components of a compiled class (methods, fields, annotations etc.).

Listing 5.1: The ClassVisitor class.

```
1  public abstract class ClassVisitor {
2      public ClassVisitor();
3
4      // Visit the header of a class
5      public void visit();
6      // Visit the source file of the compiled class
7      public void visitSource();
8      // Visit the annotations of a class
9      public AnnotationVisitor visitAnnotation();
10     // Visit the class fields
11     public FieldVisitor visitField();
12     // Visit the class methods
13     public MethodVisitor visitMethod();
14     // Visit the end of the class. It is used to inform the
           visitor that all the class attributes have been
           visited
15     void visitEnd();
16     ...
17 }
```

A simplified version of the ClassVisitor class, containing only some of the most important methods, is presented in Listing 5.1. The complete class with a detailed description about each method can be found in the ASM guide [Bru]. By providing an implementation to the visit methods of this class, the user can extract information about the compiled class elements or specify a set of actions that will be applied to them. Apart from the three main classes (ClassVisitor, ClassReader and ClassWriter), the ASM API contains specialized visitor classes for some of the class components. For instance, there is the *FieldVisitor* class, which has its own methods for visiting the annotations and attributes of class fields.

The ClassReader class is used for analyzing a compiled class. It accepts a ClassVisitor instance and executes its visit methods to inspect the compiled class components. The ClassWriter extends the ClassVisitor class and is used for generating classes from scratch or for modifying existing classes. The resulting class is stored in a byte way which can be instantiated with a class loader instance or written in a file for future use.

### 5.1.2 Class Rewriting at Runtime

This subsection will provide the implementation details about how the ASM library is used in this work to bridge the gap between the APIs of TornadoVM without user intervention.

To facilitate the code morphing process, a set of classes is introduced, which adhere to the programming model of TornadoVM. At runtime, the Flink user function is patched automatically into the appropriate function of these classes using ASM. Then, the TornadoVM-compatible function, that contains the Flink user code, is provided to a Task Schedule.

These new classes are *TornadoMap*, *TornadoReduce*, *MiddleMap*, *MiddleReduce*, *MapASMSkeleton* and *ReduceASMSkeleton*. The classes TornadoMap and TornadoReduce, which are presented in Listings 5.2 and 5.3, contain functions that match the programming model of TornadoVM. In each of these two classes, there are map/reduce functions for various input and output types (e.g. primitive, Flink Tuples etc.). In addition, the TornadoMap class contains a field of type MiddleMap, while the TornadoReduce class a field of type MiddleReduce. This field is used to invoke, inside the for-loop of the class methods, a MiddleMap/MiddleReduce function of matching input and output types (as can be seen in lines 10 and 16 of Listing 5.2 and in line 11 of Listing 5.3).

Listing 5.2: The TornadoMap class.

```
1  public class TornadoMap {
2      private MiddleMap mdm;
3
4      public TornadoMap(MiddleMap mdm) {
5          this.mdm = mdm;
6      }
7
8      public void map(int[] in, int[] out) {
9          for (@Parallel int i = 0; i < in.length; i++) {
10             out[i] = mdm.mapintint(in[i]);
11         }
12     }
13
14     public void map(Tuple2[] in, Tuple2[] out) {
15         for (@Parallel int i = 0; i < in.length; i++) {
16             out[i] = mdm.maptuple2tuple2(in[i]);
17         }
18     }
19
20     ...
21 }
```

Listing 5.3: The TornadoReduce class.

```
1  public class TornadoReduce {
2      public MiddleReduce mdr;
3
4      public TornadoReduce(MiddleReduce mdr) {
5          this.mdr = mdr;
6      }
7
8      public void reduce(int[] input, @Reduce int[] result) {
9          result[0] = 0;
10         for (@Parallel int i = 0; i < input.length; i++) {
11             result[0] = mdr.redintint(input[i], result[0]);
12         }
13     }
14
15     ...
16 }
```

The abstract classes MiddleMap and MiddleReduce (depicted in Listing 5.4), contain typed methods that match fine grained API of Flink. Specifically, the functions

Listing 5.4: The MiddleMap and MiddleReduce classes.

```
public abstract class MiddleMap {
    public abstract int mapintint(int in);
    public abstract Tuple2 maptuple2tuple2(Tuple2 t);
            ...
}
```

```
public abstract class MiddleReduce {
    public abstract int redintint(int x, int y);
        ...
}
```

Listing 5.5: The MapASMSkeleton and ReduceASMSkeleton classes.

```
public class MapASMSkeleton {
    public int mapintint(int in) {
        return 0;
    }

    public Tuple2 maptuple2tuple2(Tuple2 t) {
        return null;
    }
            ...
}
```

```
public class ReduceASMSkeleton {
    public int redintint(int x, int y) {
        return 0;
    }
            ...
}
```

of the MiddleMap class have a single input and output value, similarly to Flink map functions. Similarly, the MiddleReduce methods have two input arguments and return a single value just like the reduce function of Flink.

Finally, the classes MapASMSkeleton and ReduceASMSkeleton, as presented in Listing 5.5 extend MiddleMap and MiddleReduce respectively. Both classes provide a dummy implementation for the abstract methods, which is to return zero or null depending on the output type. The MapASMSkeleton and ReduceASMSkeleton classes are the ones that are manipulated with ASM. Specifically, at runtime, the MapASMSkeleton or ReduceASMSkeleton function that matches the Flink user code

is identified using an ASM ClassReader. Then, this function is altered by an ASM ClassWriter instance to return the Flink user function call instead of zero or null.

At this point, it is natural to wonder why a more straight-forward approach was not followed; what is the purpose of having the MiddleMap/MiddleReduce and MapASMSkeleton/ReduceASMSkeleton classes, instead of directly modifying the TornadoMap/TornadoReduce classes with ASM? To explain this, some insight into the way that Java performs class loading has to be provided. During the JVM initialization all classes are loaded with the system classloader. For every class, the system classloader makes sure that it has not already been loaded and, only then, loads it. Since the TornadoMap class is loaded upon the JVM initialization, it cannot be reloaded at runtime with the system classloader. Therefore, to reload a class at runtime, a different classloader instance has to be utilized.

However, loading the ASM-generated class with a classloader other than the system classloader causes the following issue. Every Java class is identified by its full name (package and class name) and the classloader that loaded it. Therefore, attempting to load the ASM-generated class like below would cause an illegal casting exception.

```
AsmClassLoader loader = new AsmClassLoader();
TornadoMap tmap = loader.loadClass("org.apache.flink.api.
    asm.TornadoMap", ASMbytes);
```

The reason is that the variable `tmap` is not considered by the system to be of the same type as the loaded class instance.

As a work-around, the intermediate classes MiddleMap/MiddleReduce and MapASMSkeleton/ReduceASMSkeleton were introduced. The ASMSkeleton classes extend the Middle classes. This means that, for example, even though an ASM-generated MapASMSkeleton instance cannot be cast to a variable of type MapASMSkeleton due to the classloader issue, the following cast is legal, since both versions extend the same MiddleMap class.

```
AsmClassLoader loader = new AsmClassLoader();
MiddleMap md = loader.loadClass("org.apache.flink.api.asm
    .MapASMSkeleton", ASMbytes);
TornadoMap tmap = new TornadoMap(md);
```

Next, an example will be used to showcase how ASM transforms the provided classes.

### Example

The Flink user class that will be used in this example is presented in Listing 5.6. This class contains a map function that has an input and an output of type Tuple2. The map function multiplies the second field of each Tuple by two. Since, in this example, the user function is a map, the class that will be manipulated with ASM will be the MapASMSkeleton.

Listing 5.6: A Flink Map User Class.

```
1 public static final class UserClass implements MapFunction
      <Tuple2<Double, Double>, Tuple2<Double, Double>> {
2
3     @Override
4     public Tuple2<Double, Double> map(Tuple2<Double, Double>
          value) {
5         return new Tuple2<>(value.f0, value.f1 * 2);
6     }
7
8 }
```

Before any changes are applied to the MapASMSkeleton class, some preprocessing steps have to be followed.
**Step 1:** A ClassReader instance `cr` examines the Flink user class to deduce the input and output types of the Flink user function. In this example the function signature is `public Tuple2<Double, Double> map(Tuple2<Double, Double>`, so the input and output types are identified to be Tuple2.
**Step 2:** A new ClassReader instance, using the input/output types of the Flink user function, identifies the function of the MapASMSkeleton that matches the types and therefore will be modified. Since the ClassReader `cr` deducted that the input and output types of the Flink UDF are Tuple2, the MapASMSkeleton function that will be transformed in this example is `maptuple2tuple2`, as it has matching types.

Figure 5.3 illustrates how the MapASMSkeleton class is changed to trigger the Flink user function inside the maptuple2tuple2 method. At the top of the figure (labeled (a)), the original Flink class and its bytecode representation is presented. At the bottom, (labeled (b)) is the resulting class, both in Java and bytecode format. The bytecodes in blue are the ones inserted by the code morphing module. The process of transitioning from (a) to (b) can be summarized in the following steps.

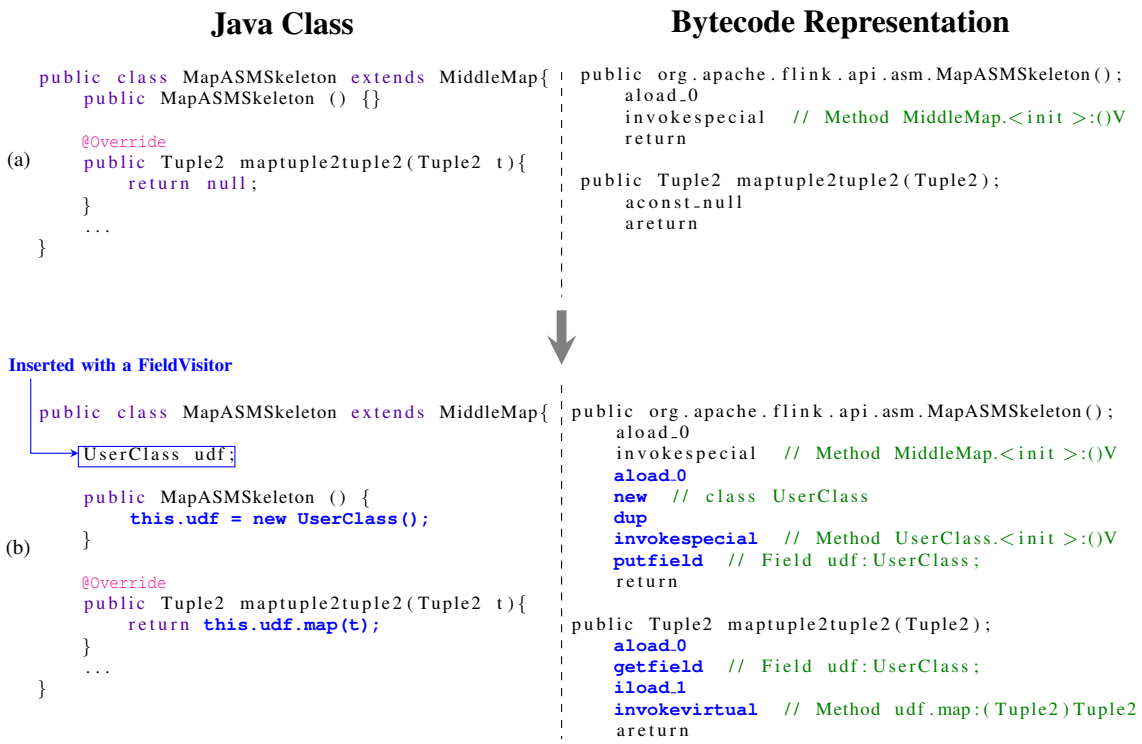


Figure 5.3: Transforming the Skeleton class.

**Step 1:** A ClassWriter instance `cw` inserts a new public field named `udf` in the MapASMSkeleton class, which will have the same type as the Flink user class. In this case the field will be of type `UserClass` (blue box in Figure 5.3).

**Step 2:** Then, the constructor of the MapASMSkeleton class is visited by `cw`. The visit method initializes the `udf` field with an instance of the Flink user class. To achieve that, the visitor inserts the bytecode instructions that represent the initialization of the `udf` field with an instance of the Flink user class.

**Step 3:** Finally, the `maptuple2tuple2` function is visited by `cw`. The `aconst_null` bytecode, which represents the null value, is replaced by a set of bytecode instructions that represent the invocation of the `udf` map. The input of the `maptuple2tuple2` function is also provided as input to the `udf` map.

Finally, this transformed instance of the MapASMSkeleton class is stored in a byte array. This byte array is then saved in the configuration data of the Job Graph vertices to be distributed across the cluster. Note that all the steps described above are generic, meaning that they are followed for all input and output type combinations. Regarding Flink reductions, equivalent actions are applied to the ReduceASMSkeleton class. This whole transformation process takes place without any user knowledge or intervention.

### 5.1.3 Invocation of Heterogeneous Execution

After the deployment of a task to the Task Manager, the execution is directed to the appropriate Flink driver class. This example assumes that the driver that is used is the ChainedMapDriver, presented in Listing 5.7.

Listing 5.7: Executing the Flink User Class on TornadoVM.

```
1  public class ChainedMapDriver {
2      void collect () {
3          byte[] bytesMsk =  this.configuration.
               getSkeletonMapBytes();
4          AsmClassLoader loader = new AsmClassLoader();
5          MiddleMap md = loader.loadClass("org.apache.flink.
               api.asm.MapASMSkeleton", bytesMsk);
6          TornadoMap tmap = new TornadoMap(md);
7
8          ...
9
10         TaskSchedule ts = new TaskSchedule("s0")
11           .task("t0", tmap::map, in, out)
12           .streamOut(out)
13           .execute();
14     }
15
16 }
```

Firstly, the byte array containing the transformed MapASMSkeleton class is retrieved from the configuration data (line 3). Using a classloader instance, this class is loaded and assigned to a MiddleMap variable `md` (lines 4 and 5). Then, a TornadoMap variable `tmap` is initialized, which takes `md` in its constructor. It is reminded that the TornadoMap functions have a for loop that is marked with the TornadoVM `@Parallel` annotation and iterates over an input array. Inside each loop, the MiddleMap function of the same input/output type is called on each input element (Listing 5.2). This means that, by passing `md` to the TornadoMap class, the appropriate TornadoMap map method will call the Flink user function in its for-loop for all the input elements. This map method can be provided to a Task Schedule to be deployed for heterogeneous execution.

This section presented a way that the Flink user code can be deployed to TornadoVM in a completely agnostic way to the user and without any changes to the existing API of Flink. In the next section, the data morphing module will be presented.

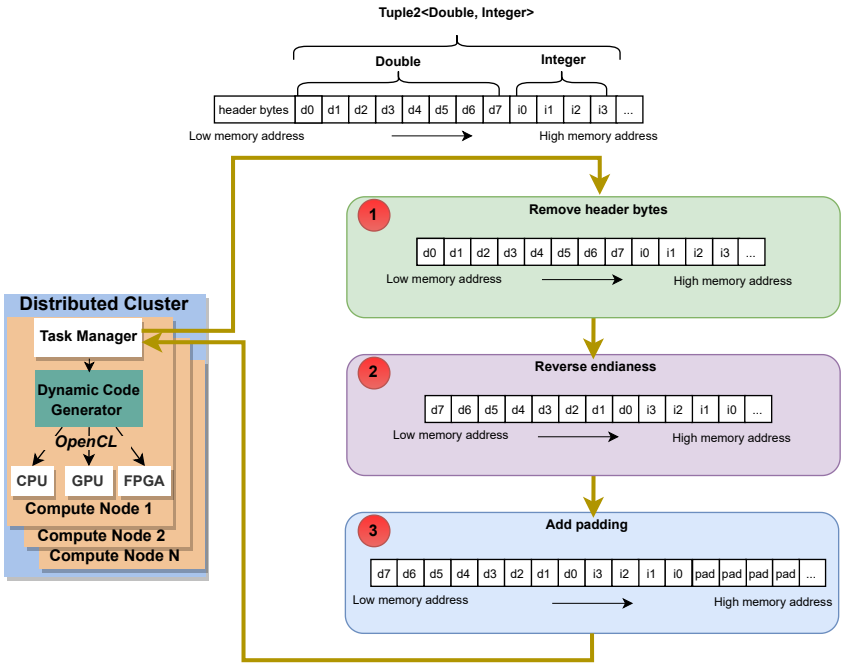


Figure 5.4: An Overview of the Data Morphing Module.

## 5.2 Data Morphing

Following the code transformation, the next challenge that needs to be overcome has to do with the data types. Java objects are an integral part of OOP, however, they are not supported in accelerator devices like GPUs and FPGAs. In the first implementation (Chapter 4), the object types (e.g. Tuples) were converted to primitive arrays - one array for each field. Nevertheless, on closer inspection of the Flink system, the following was observed. On Flink the data already undergoes a type of marshalling in order to be distributed among the cluster. This marshalling regards the transformation of object types to bytes through serialization. Since byte arrays can be allocated on GPU/FPGA memory, this implementation explores the idea of directly getting the data in byte form and writing to device memory instead of deserializing it and converting it to primitive arrays. Moreover, if Big Data systems were developed with heterogeneity in mind, the marshalling could be avoided altogether by getting the data in byte form.

However, the data stored in the serialization buffers cannot be directly forwarded to the hardware accelerator memory for the following reasons. First of all, the serialization buffers contain header bytes which are essential to maintain the serialization semantics. Nevertheless, in this context, they increase significantly the memory footprint, without adding any value as they do not contain actual computational data.

Secondly, the majority of commercial hardware accelerators are Little Endian architectures, in contrast to the Big Endian data serialization mechanism of the JVM. Therefore, in order to read data correctly, the bytes of each value have to be reversed. Lastly, in case objects consist of fields that have a different size (e.g., an int value is represented with four bytes, while a double value with eight), padding has to be included in order to avoid unaligned memory accesses. Unaligned memory accesses can yield undefined results [NVI22] in NVIDIA GPUs, and reduce the overall performance in Intel GPUs.

To transform the serialized data in a format that is suitable for hardware devices, the data morphing module was introduced. Figure 5.4 illustrates an overview of this module and how it transforms the byte representation of a set of Tuple2 elements, that consist of a Double and an Integer field. As it is depicted in the figure, the module operates as follows:

1. It extracts raw data, excluding the header bytes.
2. It reverse the endianess of the bytestream so that the little-endian ordering is followed.
3. If the serialized values are of different size (which they are in this example), it adds extra bytes for padding.

The remainder of this section is split into three subsections. Subsections 5.2.1 and 5.2.2 provide essential background information by presenting how various data types are serialized on Flink and what the format of the serialized data is when it reaches the Flink operators. Subsection 5.2.3 describes in detail how the data morphing module accesses the serialized data and modifies it to be valid for heterogeneous execution.

### 5.2.1 Flink Serialization

Flink has specific software components that are dedicated to handling data types. Initially, when an operator is declared on a dataset, the user code is analyzed by Flink to extract the input and output types. Next, Flink examines these types and stores information relevant to their serialization in an instance of the *TypeInformation* class. All Flink data types are associated with a TypeInformation class. For instance, Tuple objects store type information in a *TupleTypeInfo* class instance, boxed primitives in a *BasicTypeInfo* class instance, etc.

Each TypeInformation class has a function which assigns a serializer class to the data type. There are several different serializer classes in Flink which contain functions that define how the data should be copied, serialized, deserialized, etc. In this thesis, the way that Tuples, boxed primitives, arrays of primitives, and arrays of objects are serialized will be discussed.

### Serialization of Boxed Primitives

Flink provides serializer classes (e.g., IntSerializer, FloatSerializer, DoubleSerializer, etc.) for all boxed primitive types. Before diving into the Flink serialization details, some background on how Java serializes primitives and boxed primitives has to be provided. Depending on their type, primitive values are represented with one (boolean, byte), two (short, char), four (int, float), or eight (long, double) bytes. Nevertheless, all boxed values regardless of their type require 16 bytes in total. The first 8 bytes store information that is essential to memory management. The rest are dedicated to the actual data. Even though most primitive values can actually be represented with less than 8 bytes, boxed values are always stored using 8 bytes due to JVM specifications (object sizes must always be a multiple of 8 for aligned memory accesses). Moreover, it is important to note that all Java types are serialized in a Big Endian order, meaning that the most significant byte is stored at a lowest memory address compared to the least significant.

However, in order to optimize memory allocations, Flink uses a different approach for boxed primitive serialization. Specifically, for each boxed value, only the raw primitive data is serialized. This means that in Flink, a set of 16 Integer values takes up 64 bytes (4 bytes for each value * 16), while with the original Java serialization process it would take 256 bytes (16 bytes for each value and the header * 16). These primitive values are serialized using Java serialization, so the same semantics discussed above apply (about their size, their endianess etc.). Figure 5.5 illustrates how a set of Integer records is serialized in Flink. For each record the IntSerializer class converts its int value to four bytes, using Java serialization.

### Serialization of Tuples

For Tuple serialization, Flink tries once again to serialize the data as compactly as possible. Therefore, no extra header bytes are included. For a Tuple to get serialized, the serializer of each field is called. Figure 5.6 presents how a Tuple2 record that contains an Integer and a Double field is serialized. First the record is sent to the

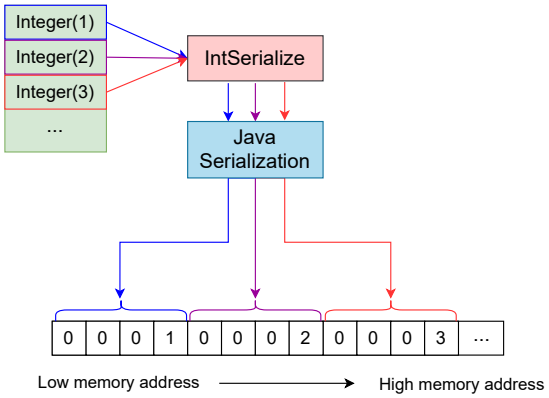


Figure 5.5: Serialization of Integers in Flink.

TupleSerializer. For each field, the TupleSerializer invokes the serialization function of its data type class. For Integers, the IntSerializer is called like in the previous example. For Doubles, the DoubleSerializer is called. These serializers, in conjunction with Java serialization functions, convert the Integers and Doubles to four and eight bytes respectively. Therefore, in this example each serialized record comprises of 12 bytes (four for the Integer and eight for the Double).

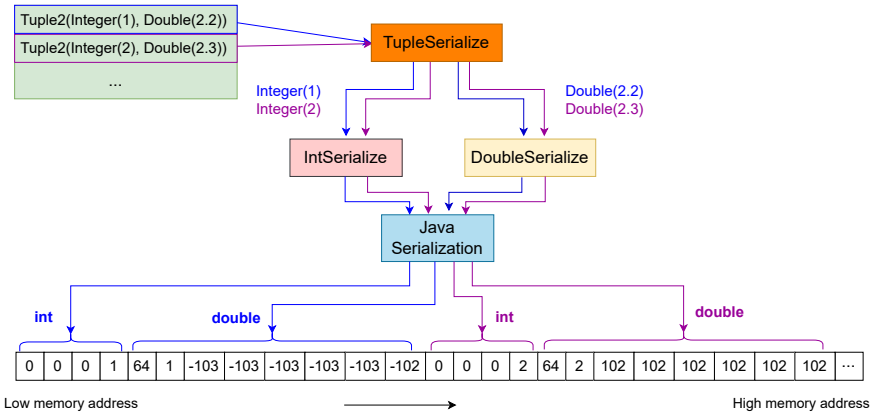


Figure 5.6: Serialization of Tuple2<Integer, Double>types on Flink.

### Serialization of Primitive Arrays

Flink also provides serializers for primitive arrays. Each type of primitive arrays has its own dedicated serializer class. For example, integer arrays are serialized using the *IntPrimitiveArraySerializer* class, double arrays are serialized using the *DoublePrimitiveArraySerializer* and so on. In all of these classes, prior to the serialization the data,

an integer value with the length of the array is serialized. Then, each primitive array element is serialized using the default Java serialization. Figure 5.7 illustrates how a dataset that consists of integer arrays is serialized. First, the length of each array is converted to four bytes. In this example each array has three elements, so the length value is three. Then, each integer is converted to four bytes. This is repeated for all array records.

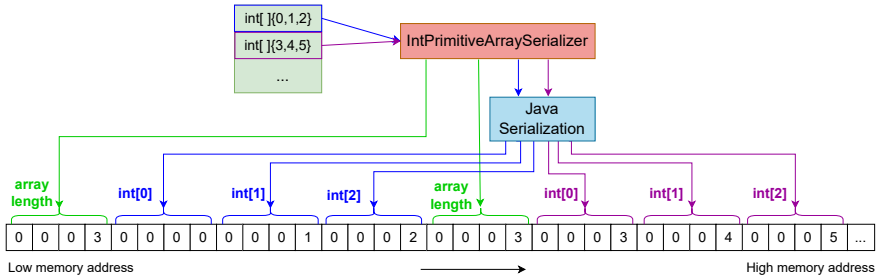


Figure 5.7: Serialization of int arrays on Flink.

#### Serialization of Arrays of Objects

The last data type examined in this thesis is the object arrays type. The dedicated class for this serialization type is the *GenericArraySerializer* class. Just like for the primitive arrays, the length of the array is serialized first. Then a bit that is set to zero if the array element is null and one otherwise is written. Finally each array element is serialized with its corresponding serialization technique. If an element is null, zeros are inserted in the serialized buffer instead. Figure 5.8 presents how arrays of Integer values are serialized. First the length of the arrays is serialized (which is this case is two). Then, one byte set to 1 (true) signifies that the value that follows is not null. Finally, the Integer is serialized using the IntSerializer class like in the previous examples.

### 5.2.2 How The Flink Tasks Are Accessing Serialized Data

The reason Flink serializes the data is to distribute it the across the cluster, therefore, next it is be described how the serialized data is actually accessed by the operators.

DataSource tasks are the tasks that represent the input dataset of a computational pipeline. If the input comes from a file, e.g., a csv file, then the DataSource stores information about how the data should be read (how the values are seperated, the type of the file etc.). However, if the input is a Java Collection, the serialized data is stored in its configuration. Specifically, in this case the serialized data is written in a Java

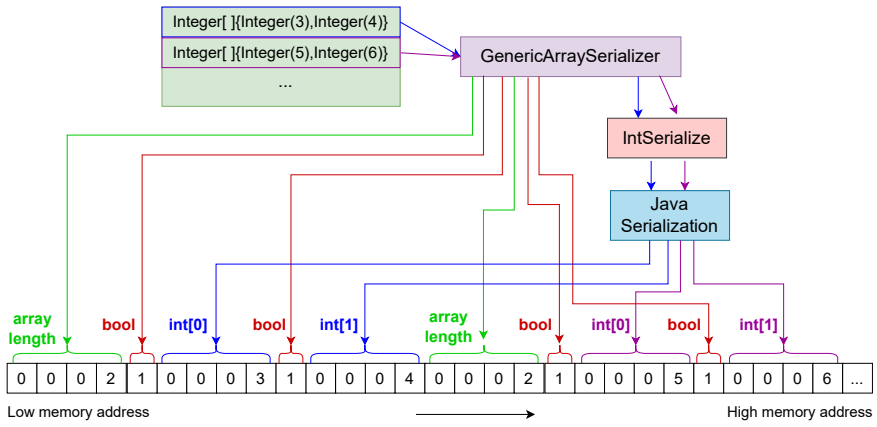


Figure 5.8: Serialization of Integer arrays on Flink.

*ObjectOutputStream* stream. The first few bytes of the stream contain some header bytes for the collection class. Next, four bytes are written to indicate the size of the collection.

Then, the data follows. For primitive data, the ObjectOutputStream splits serialized bytes in blocks of 1024 bytes. Each block contains a header and the serialized data. The header consists of five bytes unless the data block has less than than 256 bytes. In this case the block is prefixed with a two-byte header. Since in Flink serialization the serialized buffers, for all the types we discuss above, contain data in primitive form (the unboxed values) the output stream is split in blocks as described above. Figure 5.9 presents the format of an ObjectOutputStream that contains a serialized dataset of Tuple2 records. As can be seen in the figure, the stream consists first of some header bytes for the collection. Then, the size of the input is written in the next four bytes. After that follow the serialized data by Flink, split among block of 1024 bytes which are preceded by two or five header bytes.

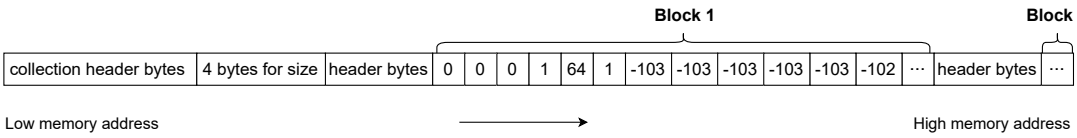


Figure 5.9: DataSource Bytes.

When a pipeline of tasks is distributed among the cluster, the first task that gets executed is the DataSourceTask. In the original Flink implementation, this task extracts the ObjectOutputStream data from its configuration, deserializes it and then provides it

to a collector. If a computational task (map, reduce etc.) is chained to the DataSourceTask, then the collector is this task and, therefore, the materialized data is directly passed to the user function of this operator. This is safe to do, in this case, because the DataSourceTask instance is executed by the same thread as the computational task (since they are chained).

If the DataSourceTask is not chained to another, then this data is written in memory. Flink uses the `SpanningRecordSerializer` class, which writes the data in the following way. First, it writes the number of bytes of each record and then serializes again the data and writes them in byte format. Figure 5.10 illustrates how a Tuple2 with an Integer and a Double field would be written. The first four bytes represent the size of the Tuple, which is 12 in this case (4 for the Integer and 8 for the Double). Then the serialized bytes are written.

The reason why, in this case, the number of bytes per record is written is explained by the Flink memory management. In Flink, memory is represented by `Memory Segments`, which essentially are regions of memory with a fixed size. Depending on the size of the records and the size of the memory segments, records might span across multiple memory segments. The memory system needs to be aware of how many bytes there are per record so that the complete data is read even if they are stored in separate memory segments.

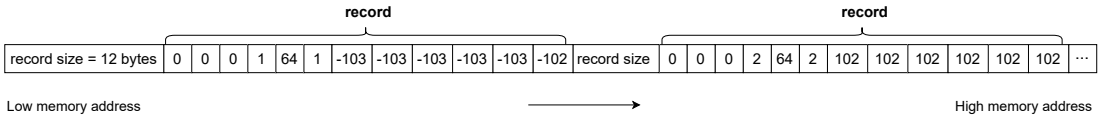


Figure 5.10: Memory Record Bytes.

Since chained and non-chained operators access the serialized buffers in different formats, we need to perform different actions in each case to get the byte data.

### 5.2.3 Data Conversion

As explained in Subsection 5.2.2, the way that the data is accessed by each computational task depends on whether it is chained with its predecessor or not. If it is, the task gets its input data materialized (either from a DatasourceTask or from a previous operational task). If it is not, it reads data from memory. Therefore, in order to extract the data in byte form to provide it as input to a TornadoVM TaskSchedule, different approaches have to be followed for chained and non-chained operators.

Consequently, the data morphing module operates in the following way. If it identifies that the DataSourceTask is chained with an operational task, it gets the byte stream from the configuration data and, instead of materializing it, it extracts the raw serialized data, without the extra bytes that were inserted during the creation of the ObjectOutputStream buffer. This data will then be forwarded to the computational operator in byte format. Otherwise, the DataSourceTask writes the data in memory like the original implementation of Flink so that they can be consumed by the next operator.

If a task reads the data from memory, instead of materializing each individual record from memory, all the data is extracted from the memory segments and stored in a single buffer. The extra bytes that specify the size of each record are removed since they are not necessary for the computation and they only lead to larger memory allocation.

Even though, by following the steps described above, the raw serialized data can be extracted, there are still two main obstacles to overcome before being able to delegate this data to TornadoVM. First of all, as mentioned in 5.2.1, the data is written in the serialization buffers using Big Endian ordering. However, the vast majority of commercial hardware accelerators have Little Endian architectures. Therefore, the bytes of each value have to be reversed. Secondly, as explained in 5.2.1, Flink serialized buffers can contain data of different sizes. For instance, if the data consists of Tuple2 records and the first field is an Integer while the second a Double, then the first field takes up four bytes while the second eight. In a byte array like that the accesses of the fields inside the byte buffer would be unaligned with respect to the memory alignment requirements of the underlying architecture. To solve this problem, padding has to be included in the byte data when the sizes of the object fields are not homogeneous.

Figure 5.11 presents an example of the transformations described above. The array at the top contains the raw serialized data for Tuple2 records. In this example the first field is an Integer and the second is a Double. The array at the bottom is the array that is passed to TornadoVM. As can be seen in the figure, the ordering of the bytes of each field has been reversed and four extra bytes have been added to the Integer field as padding. It is important to note, that in order to have a uniformed representation of data and also to make our byte arrays as compact as possible, all the extra bytes that exist in the serialized buffers for object and primitive array types are removed. Specifically, the bytes that are removed are the boolean bytes that specify if an array of objects has null data and the bytes that indicate the size of serialized arrays.

After the data morphing is completed, the byte array is ready to be passed as input

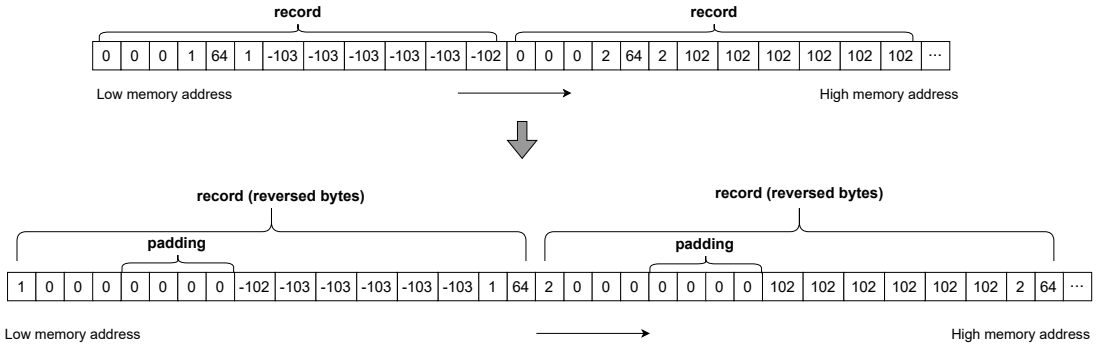


Figure 5.11: A Data Morphing Example.

to TornadoVM. However, as mentioned in Section 5.1, the TornadoMap functions are typed as the Flink user functions. The reason it has to be this way because the input of the TornadoMap function is provided to the Flink user function that is inside the for-loop, so the types have to match. Therefore, the Task Schedule API of TornadoVM was augmented with an extra function. This function receives an object that contains the input array in bytes and a byte array that will store the output. If the computation comes from Flink, TornadoVM will ignore the arguments of the Task Schedule and use these byte arrays for reading and writing instead. Listing 5.8 presents such a Task Schedule. In lines 5 and 6 the byte arrays are initialized. Then two dummy arrays are created to keep the TornadoVM task semantics. Then in line 11 the byte data is stored in an instance of the `FlinkBytes` class. This is a class that resides on the TornadoVM side and is dedicated to store byte data coming from Flink. This FlinkBytes object is passed to the Task Schedule through a new function called `flinkData`.

## 5.3 Dynamic Code Generation for Hardware Accelerators

By transforming the Flink user function into a TornadoVM-compatible method and by getting the data in a format that is suitable for heterogeneous execution, a Flink computation can be deployed to heterogeneous devices through TornadoVM. However, at this point the code would not compile. The reason is that, the developers write their application under the assumption that they are operating on objects and not byte arrays. Therefore, modifications at the compiler level have to be made to ensure that the generated OpenCL kernels will access the data correctly.

Listing 5.8: Executing the Flink User Class on TornadoVM with Extended Task Schedule.

```
1  public class ChainedMapDriver {
2      void collect () {
3              ...
4          // make the byte dataset ready for TornadoVM
5          byte[] inputBytes = datamorphing();
6          byte[] outBytes = new byte[];
7
8          Tuple2[] in = new Tuple2[1];
9          Tuple2[] out = new Tuple2[1];
10         // store the data in a new TornadoVM class
11         FlinkBytes fdata = new FlinkBytes()
12
13         TaskSchedule ts = new TaskSchedule("s0")
14             // provide the byte arrays to TornadoVM
15             .flinkData(fdata)
16             .task("t0", tmap::map, in, out)
17             .streamOut(out)
18             .execute();
19     }
20
21 }
```

For this reason, the code generation module was integrated with Flink. This module contains TornadoVM, extended with a set of compiler phases specialized for Flink user functions. An overview of the code generation module is presented in Figure 5.12. As shown in the figure, this module receives three inputs: (i) the TornadoVM-compatible user function, (ii) the byte datasets and, (iii) the TypeInformation for the function. The TypeInformation it is essential to perform the the compiler transformation that will be described next. As discussed in subsection 5.2.1, the TypeInformation is produced by Flink during the serialization phase. To access it from the Task Manager, the same technique that was applied for the ASM-generated functions is followed. Specifically, the TypeInformation objects for all the methods are placed in a HashMap, which is serialized and stored in the configuration data of the Job Graph along the byte arrays that contain the ASM generated classes.

The TornadoVM compiler has been augmented with six compiler phases. Half of these phases are included in the high-tier and the other half in the low-tier. These phases are responsible for making the data accesses in the user code match the data layout imposed by the data morphing module. Specifically:

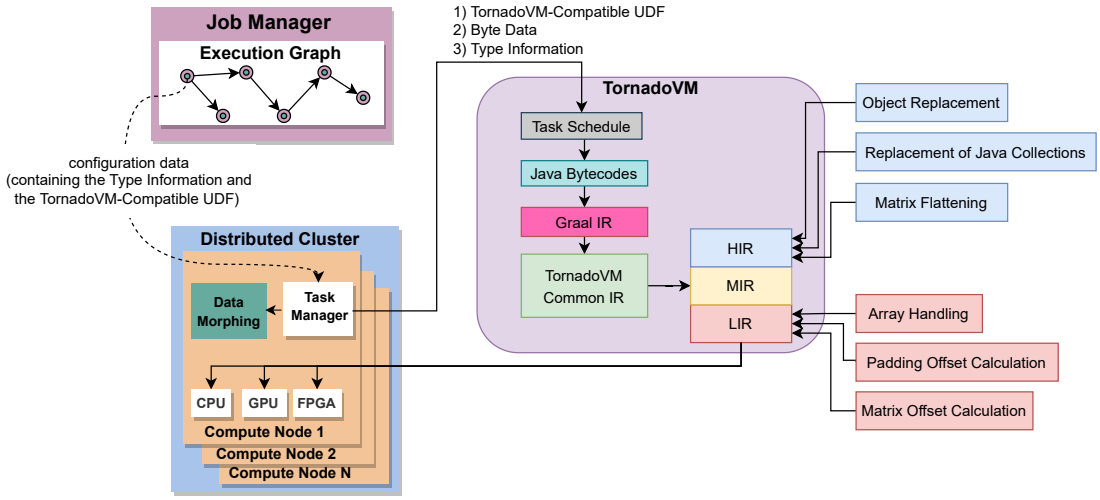


Figure 5.12: Dynamic Code Generation for Heterogeneous Accelerators.

1. The **Object Replacement** phase replaces the loads/stores for accessing object field, with the equivalent loads and stores for accessing the data from a byte array.

2. The **Padding Offset Calculation** phase calculates the exact addresses from which each value should be read, if the byte array has padding.

3. The **Replacement Of Java Collections** phase replaces the actions of common Java Collection function calls with the corresponding array operations. For instance, a call to the get() function is replaced with an array load.

4. The **Array Handling** phase provides support for objects where one of their fields is an array (e.g. types such as Tuple2<double[], Long>). This phase specifies how such data is accessed and copied.

5. The **Matrix Flattening** phase is functions that operate on matrices. As mentioned in Subsection 5.2.3, for all data types, the data morphing module removes all header bytes to keep the data as compact as possible, and returns a linear byte array. Therefore, this compiler phase transforms the multi-dimensional matrix accesses to one-dimensional array accesses.

6. The **Matrix Offset Calculation** phase specifies the exact offsets of the byte array from which each matrix element should be read.

Subsection 5.3.1 will provide background information about the GraalVM compiler, which is the basis of the TornadoVM compiler, and describe the format of the

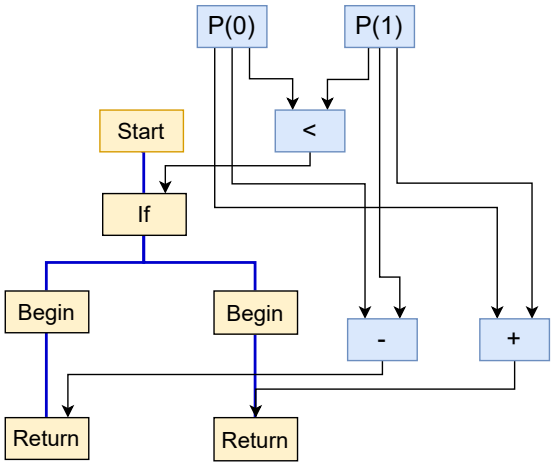


Figure 5.13: The Generated GraalVM IR for the code in Listing 5.9.

Graal IR graphs. In Subsection 5.3.2, each of the aforementioned compiler phase will be presented in detail, with examples of the changes on the IR that each phase applies. The nodes of the IR that are colored red are nodes that are being deleted by the compiler phase, while the green ones are newly inserted nodes. Finally, Subsection 5.3.3 will discuss how the output results are distributed across the system after the computation has completed.

### 5.3.1 GraalVM Compiler

The GraalVM compiler [WWS10, WWW+13] is a high-performance JIT compiler that replaces or complements the JVM HotSpot existing compilers. It is integrated to the JVM through the JVM Compiler Interface (JVMCI) [Ros]. One of its main advantages is that it is written in Java, unlike the *client* (*C1*) and *server* (*C2*) compilers, which are written in C++. Due to this fact, the GraalVM compiler is less prone to crashes, since there are no segmentation faults and errors can be handled with exceptions. Moreover, it is easier to maintain and extend using the numerous tools of the Java ecosystem (e.g. profilers, debuggers etc.). Just like the other JIT compilers, the GraalVM compiler generates machine code from the JVM bytecodes. To carry out this task, it creates an intermediate representation (IR) using the bytecodes, which is known as *Graal IR*. Several tiers of compilation are then applied to the IR to generate machine code that is optimized for the target device.

#### Graal IR

The GraalVM IR consists of *control-flow* and *data-flow* nodes. The control-flow nodes are connected to their successor and represent the flow of execution, while the data-flow nodes are connected to their input. The nodes can also be categorized as *fixed* or *floating*. Floating nodes can be moved around the graph, as long as the semantics of the program do not change. On the other hand, fixed nodes cannot change positions in the graph. Nodes that represent an If condition (e.g. `IfNode`) or a loop (e.g. `LoopBegin`) are examples for fixed nodes.

Figure 5.13 presents the IR that would be produced for the code in Listing 5.9. This code receives two input parameters and returns their difference, if the first value is larger than the second, or their summation otherwise. In this IR graph (Figure 5.13), the data-flow nodes are colored blue and the control-flow nodes yellow.

Listing 5.9: A Simple Java Program.

```
1 public int m (int in, int out) {
2     if (in > out) {
3         return in - out;
4     } else {
5         return in + out;
6     }
7 }
```

### 5.3.2 JIT Compilation Phases

#### Object Replacement

This phase resides at the high tier of the compilation and replaces the default way that TornadoVM loads/stores data from/to the fields of a Java object with a memory access to a byte array. In essence, the load/store operations emitted by the TornadoVM JIT compiler correspond to the instructions that load/store data from global memory to a physical register on the device. Then, to obtain the position of each input/output field within a Tuple object, the following formula is used:

$$field = tupleIndex * numberOfFields + fieldPos$$

The formula uses the following inputs: (i) *tupleIndex* indicates which Tuple in the dataset is being accessed; (ii) the *numberOfFields* specifies how many fields exist in

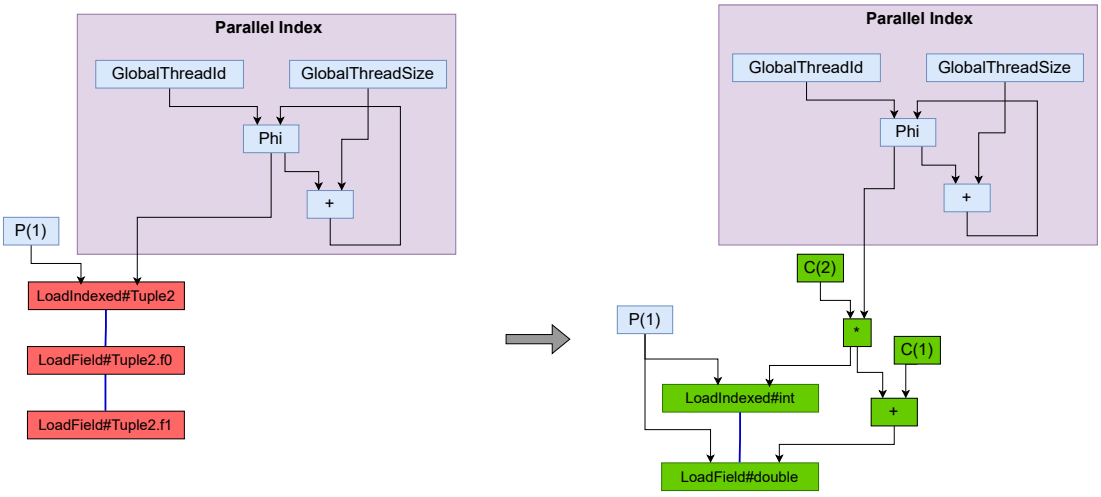


Figure 5.14: Object Replacement for Load Nodes.

the indexed Tuple object and (iii) *fieldPos*, corresponds to the position of the field that is being accessed from the Tuple, counting from zero (e.g., if the first field is accessed this value is 0, for the second field is 1 and so on).

Figure 5.14 illustrates the changes that this compiler phase applies to the IR for the loading of a Tuple2. Specifically, on the left side of the figure the original representation is provided while on the right side is presented the graph after the actions of this compiler phase are applied. The `LoadIndexedNode#Tuple2` node signifies the loading of the Tuple2 from an array of the same type. The index of the Tuple array (purple box) is constructed by TornadoVM and it essentially signifies how the computation will be split among parallel threads. From this point on, this will be referred as `Parallel Index`. The two LoadField nodes that follow represent the loading each Tuple field. During the Object Replacement compilation phase, these loading nodes are removed and replaced with one LoadIndex node per field (right side of the figure). These new load nodes are indexed with the formula provided above. Specifically, the first and the second value are accessed with the index `parallelIndex * 2` and `parallelIndex * 2 + 1` respectively.

Regarding the store nodes, on the left side of Figure 5.15 is presented the creation of a new Tuple2 object that is being stored in a Tuple2 array. The nodes that represent this allocation and store in the object array are replaced with one store node per field of the return Tuple. Since in this example a Tuple2 is returned, there are two new StoreIndexed nodes which are indexed using the same formula as the load nodes described in the previous example.

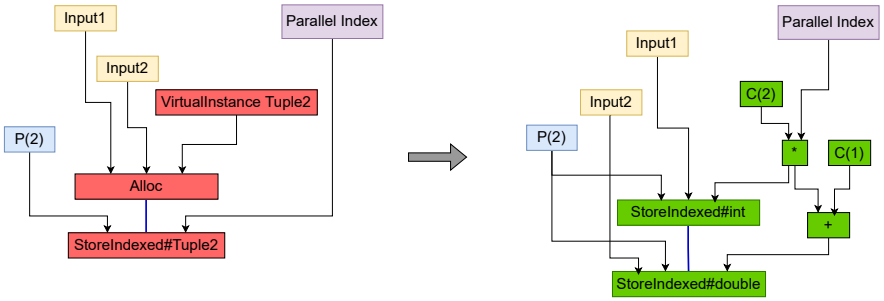


Figure 5.15: Object Replacement for Store Nodes.

### Padding Offset Calculation

The aforementioned *Object Replacement* does not take into account the padding that is inserted if a byte array consists of heterogeneous elements (values that are represented with a different size of bytes). The reason is that this phase resides at the high tier, where the specific read/write addresses are not specified yet. Therefore, a new phase named *Padding Offset Calculation* was included at the low tier of the compilation, which is only invoked if the input or output byte arrays are padded.

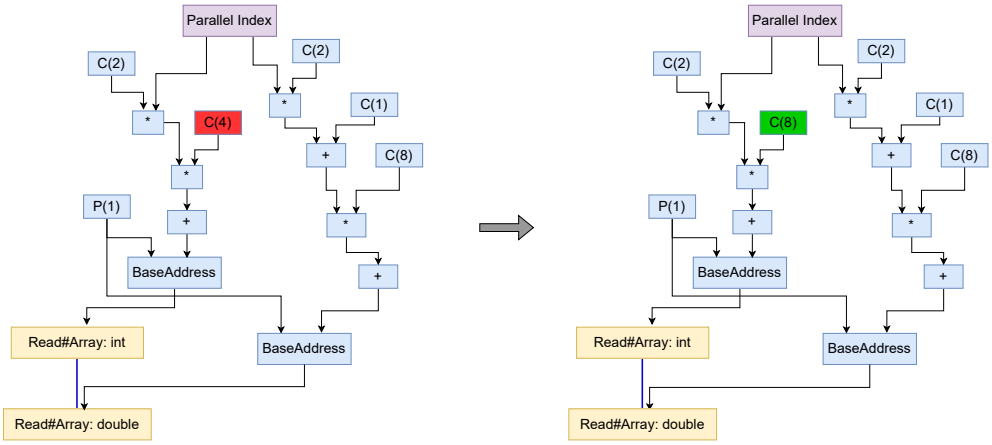


Figure 5.16: Offset calculation for Fields of Different Sizes.

During the lowering, the specific offsets from where to read and write data are calculated by the compiler. If the compiler comes across a loading node for an int or a float value, it specifies that four bytes should be read/written, so the indexing is multiplied by 4. Similarly, for a double and a long, it is calculated that eight bytes should be read/written from memory, therefore the index is multiplied by 8 in this case.

The left graph on Figure 5.16 illustrates how the graph on the right side of Figure 5.14 would look like after lowering. The LoadIndexed nodes have been replaced with `ReadArray` nodes that have as input the memory base address (in the accelerator) where the data resides in memory. In this example, and in accordance with the indices that were calculated in the *Object Replacement* phase, the int value is being read from the position `parallelIndex * 2 * 4`, while the double value is being read from the position `(parallelIndex * 2 + 1) * 8`. However, since with the padding all array elements take up eight bytes, the size four is replaced with eight (right side of the graph in Figure 5.16).

### Array Handling

In some cases, the Flink Tuple object can contain arrays of primitive values as fields. Therefore, a new compiler phase was created, that takes into account the following scenarios.

If this array is directly copied to the output (as in the left side of Figure 5.17), the load and the store nodes that represent this action are replaced with a new node named `CopyArrayTupleField`. This node generates code that perform this copying, taking into account the specific offsets in the byte buffer.

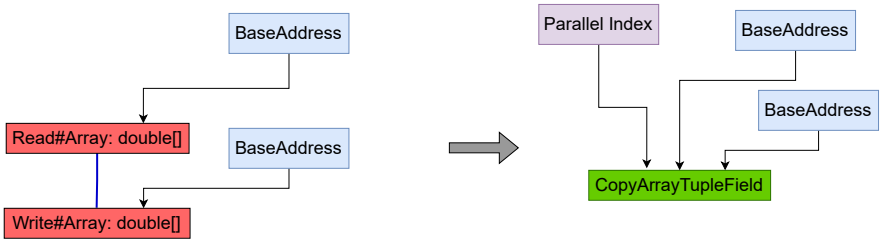


Figure 5.17: Introduce Node that Copies The Array Values of a Tuple Field.

In case specific elements of the array field are being accessed, similar actions as in the phase *Padding Offset Calculation* are performed. The formula to calculate the positions from/to which the data will be read/written is the following:

$$arrayElement = sizeOfPreviousFields + sizeOfArrayElement * innerIndex + totalSizeOfTuple * outerIndex$$

$$regularField = sizeOfPreviousFields + totalSizeOfTuple * outerIndex$$

In these formulas, the `sizeOfPreviousFields` represents the number of bytes of

all the Tuple fields that proceed it. For instance, if the field in question were the third field in a Tuple3<Integer, Double, long[]> then, the value of the `sizeOfPreviousFields` would be 4 + 8 = 12. The `sizeOfArrayElement` represents the number of bytes each array element takes in the byte buffer (eight if it is a double or a long array or if there is padding between the array elements, four otherwise). The `totalSizeOfTuple` is the size in bytes for all the elements of each Tuple. For example, in a Tuple2<double[], Double> with the double array having 4 elements, the value of this variable would be 8*4 + 8 = 40. Finally, the `innerIndex` represents the index that iterates over the inner array, while the `outerIndex` the index that iterates over the whole data.

Figure 5.18 presents how the default offsets (produced from the indexing performed during the *Object Replacement* phase) are replaced with offsets that adhere to the formulas described above. In this example the input consists of Tuple2<double[], Double> values. The value `totalBytes` in the graph represents the total size of the Tuple, while the `arrayFieldBytes` the number of bytes that each array element has. Since this is a double array this value is actually eight. In this example the outer index is the ParallelIndex.

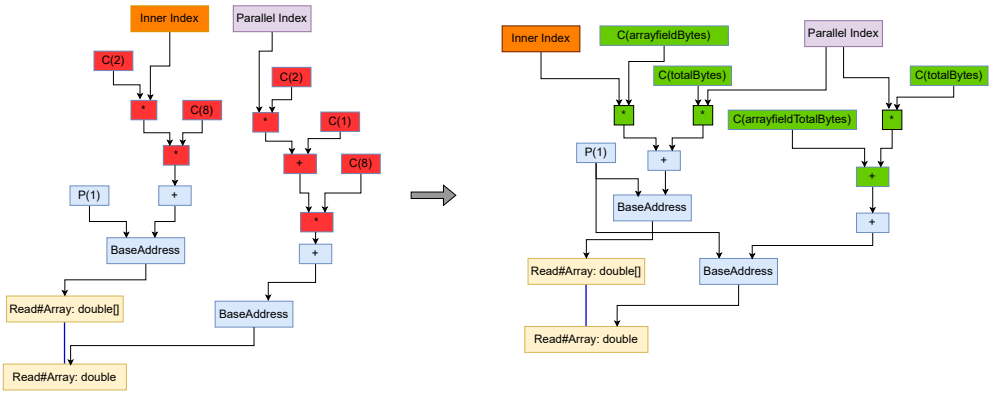


Figure 5.18: Calculate Array Offsets.

### Replacement of Java Collections

A second group of objects that is widely used in Apache Flink, and other Java-based frameworks, is derived from the Java collections library. Dynamic (e.g., `LinkedList`) or semi-dynamic (e.g., `ArrayList`) data structures of this library can potentially generate function calls or dynamic memory allocation during runtime (e.g., `size()`, `resize()`, etc.). Since hardware accelerators do not support such dynamic calls or memory allocations, a special compilation phase is added to replace all those calls with equivalent

operations on static arrays.

For example, as seen in Figure 5.19, the nodes that represent a get() operation on a collection of Tuples will be replaced with a load node. During the *Object Replacement* phase, this load node will then also be removed and replaced with loads for each raw Tuple value as has been explained.

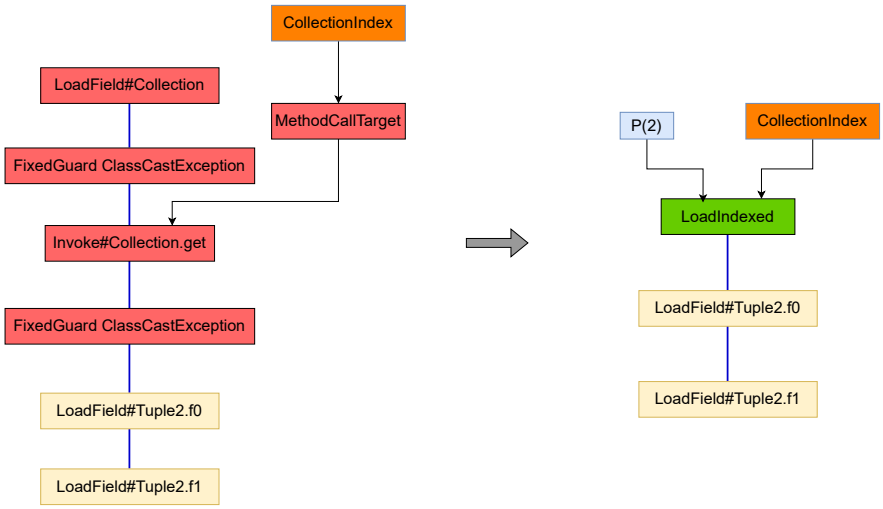


Figure 5.19: Replace a Collection get() function call with a Load.

Figure 5.20 illustrates how a function call to get the collection size is replaced with a single constant node with the size of the collection.

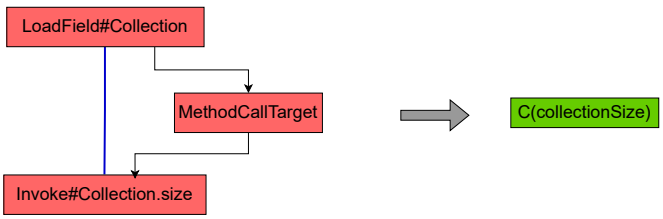


Figure 5.20: Replace a Collection size() function call with the actual size.

### Matrix Flattening

Apart from Tuples, support for matrices has also been introduced. As explained in Subsection 5.2.3, the data morphing module strives to make the data as compact as possible. For matrices, this translates to having the data *flattened*, meaning that the raw values of the matrix dataset are stored in an one-dimensional array. Therefore, the memory accesses have to be adapted to be compatible with this data layout. To

that end, this compiler phase replaces all the nodes that correspond to the original memory accesses, with nodes that access one-dimensional arrays stored in the global memory of a heterogeneous device. Figure 5.21 illustrates how the loading nodes that are generated to access a single matrix element are replaced in the graph with a loading node that accesses the same data in the linear byte array.

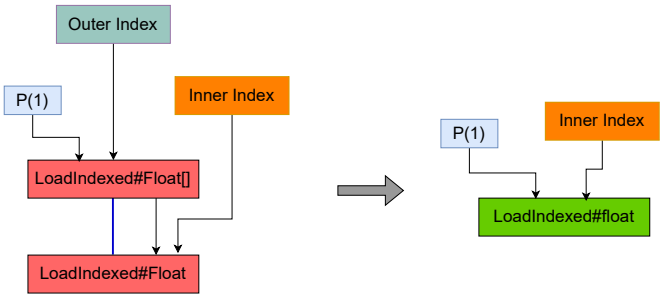


Figure 5.21: Matrix Flattening.

### Matrix Offset Calculation

The Matrix Offset Calculation phase computes the offsets for accessing the memory addresses of the flattened matrices and it is used for both writing and reading operations to/from a matrix. For example, to access an element (`m[i][j]`) of a matrix `m` pointed by indices `i` and `j`, the following formula is used:

$$matrixElement = rowByteSize * i + elementSize * j$$

In this formula, the `rowByteSize` corresponds to the size of bytes that each matrix row has (for a example, if each row consists of eight float values, then the value of the `rowByteSize` is 32 bytes). The `elementSize` stores the size of the matrix element in bytes (i.e., 4 bytes for representing `int` and `float` values, whereas 8 bytes for representing `long` and `double` values). Finally, `i` and `j` are the indices that point to a particular row and column to be accessed.

Figure 5.22 illustrates how the default offsets are replaced using the formula presented above. In this example, the matrix consists of float values, so the `elementSize` is four.

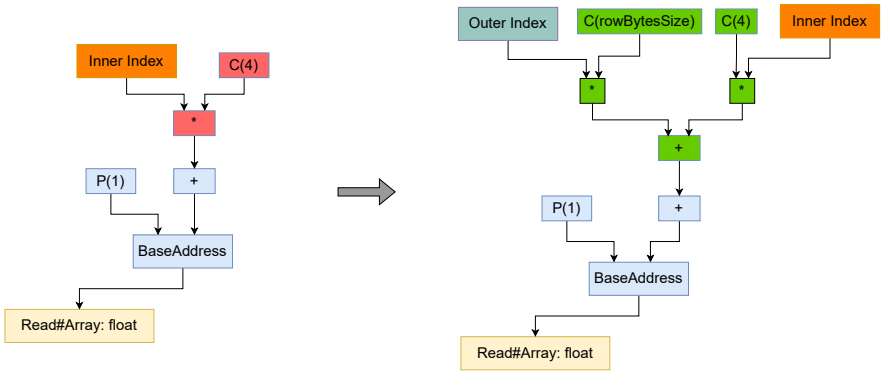


Figure 5.22: Calculating the Matrix Offsets.

### 5.3.3 Reverse Endianess & Padding of Data

Once a generated OpenCL kernel is executed, the output data which resides in the serialized byte array is returned from TornadoVM to the Flink runtime. At this stage, the order of the bytes is reversed to cope with the endianness (i.e., Big Endian) of Flink and the padding is removed. Since the results are in byte format, they can be directly patched to the serialized byte array in order to be distributed over the network by Flink.

## 5.4 Handling Hybrid Execution

The compiler extensions explained in the previous subsection essentially bridge the gap between the API of Apache Flink (or other Java-based Big Data frameworks) and the computational capabilities of heterogeneous hardware accelerators. However, the ability to transparently JIT compile Flink user functions to heterogeneous accelerators correlates with the capability of TornadoVM (or any other JVM of similar nature) to identify computational patterns (e.g., map, reduce, etc.) and generate functionally correct high performance OpenCL or PTX code. To that end, there are cases (operators) that either do not have inherent parallelism or they are not currently supported by TornadoVM. For example, in KMeans, which is one of the applications that we used to evaluate the proposed system (the evaluation will be presented in Chapter 6), one such operator exists. This operator is the `groupBy`, which is part of the computational pipeline as presented in Listing 5.10.

A closer inspection of the Flink Job Graph reveals that the groupBy and the consecutive reduce operator are being chained together in one function called `SortAndCombine`

Listing 5.10: KMeans pipeline

```
DataSet<> results = data
    .map() //Assign each point to closest centroid
    .map() //Appends a counter to the results of the first
       map
    .groupBy() //Groups data based on the centroid ID
    .reduce() //Adds the counters to calculate num points
       per centroids
    .map(); //Calculates new centroid coordinates for each
       cluster
```

via the `ChainedReduceCombineDriver`. When trying to compile the SortAndCombine function with TornadoVM, a failure occurs during the QuickSort function that is being internally used. On the other hand, the reduce operation can be compiled and executed on the hardware accelerator.

Hence, to be able to run KMeans - and other use cases that contain a groupBy - in the proposed platform two options exist: a) manually code a GPU/FPGA compatible `QuickSort` kernel and plug it in to TornadoVM, or b) support hybrid execution where a pipeline can be transparently executed partially on the CPU and on the hardware accelerators. Although the first solution would yield the best performance results since all execution would take place on a hardware accelerator, it would violate the code portability and transparency, which have been very crucial during the development of heterogeneous Big Data system. In addition, this solution is not universal as there are kernels which should not be executed at all on a GPU or an FPGA due to their limited parallelism.

Hence, the second solution was selected, which enables transparent hybrid execution of a data pipeline between a CPU and hardware accelerators. Consequently, all the operators of KMeans, besides that sorting function, are executed on the accelerator. To perform sorting, data is being copied back to the CPU, unmarshaled, and then marshaled again for continuing execution of the reduce and consequent operators on the accelerator.

Naturally, as more operators are supported by TornadoVM the need to transition execution between the CPU and the hardware accelerators will be minimized.

## 5.5 Summary

This chapter presented the proposed Big Data system, which can seamlessly and dynamically target hardware accelerators. To enable Flink to transparently deploy execution on hardware accelerators, it was extended with three modules.

1. A code morphing module, which uses a bytecode manipulation library (ASM) to transform the Flink user function call into a skeleton function that is compatible with the API of TornadoVM.

2. A data morphing module, that extracts the input data from the Flink serialization buffers and performs some essential modifications on it to make it compatible with hardware architectures.

3. A code generation module, which includes the TornadoVM compiler extended with a number of compiler phases that help accessing the data in the correct way.

The proposed platform can execute existing user Flink programs that contain various data types, (e.g., Tuples, matrices) and can easily be extended to support more types following the techniques discussed above.

Table 5.1: The criteria that our proposed implementation fulfills.

| Implementation | Big Data Framework | Code Fragmentation | Code Generation | Vendor lock-in | Device Coverage |
|---|---|---|---|---|---|
| Flink-TornadoVM Integration - Final | Flink | **No** | **On-demand** | **No** | **CPUs, GPUs, FPGAs** |

As shown in Table 5.1, the proposed platform fulfills all the criteria that were deemed crucial for modern heterogeneous Big Data systems. Specifically, it does not fragment the API since the same user code that is executed by the original Flink can be executed by the proposed system. The generated kernels are JIT-compiled, therefore no pre-compiled code is necessary. Finally, because TornadoVM can target a plethora of different hardware accelerators there is no vendor lock-in.

# Chapter 6

# Experimental Evaluation

This chapter presents the evaluation of the proposed framework against the baseline CPU-only Apache Flink. A brief description of the benchmarks that were used for this evaluation is provided in Table 6.1, along with information about the operators they consist of, whether they were executed on hybrid mode and their input data sizes.

The remainder of this chapter is organized as follows. Section 6.1 elaborates on the methodology followed to evaluate each use case. Sections 6.2 and 6.3 present the performance evaluation of benchmarks that are executed by the proposed system on GPUs and FPGAs respectively. Section 6.4 provides the performance assessment of KMeans and IoT Analytics, which utilize the hybrid execution of operators (described in Section 5.4). Finally, Section 6.5 summarizes the conclusions that were drawn from this evaluation.

## 6.1 Experimental Methodology

For the experimental evaluation two testbeds were used, which exhibit different software and hardware specifications, as presented in Tables 6.3 and 6.2. Testbed-1 is a cluster that contains two compute nodes with two discrete NVIDIA GPUs; one per compute node, while Testbed-2 is a server that contains an Intel FPGA.

The experimental methodology that was applied for all benchmarks is the following:

1. An Flink cluster was launched to initiate the Job Manager and the Task Manager nodes.

2. The benchmarks were executed and performance is measured. To ensure fair

Table 6.1: The benchmarks along with their configuration with regards to the utilized Apache Flink operators, hybrid execution and data sizes.

| Name | Description | Operators Used | Accelerated Operators | Hybrid Execution | Data Sizes & Ranges |
|---|---|---|---|---|---|
| Matrix Multiplication (MxM) | Mathematical operation widely used in Machine Learning and Deep Learning workloads. | map | map | no | Range: 128x128 - 4096x4096 |
| Logistic Regression (LR) | Standard regression analysis, which, in this paper, is used to predict the likelihood of patients' re-admission in public hospitals. | map, reduce | map, reduce | no | Training DS: 1 GB, Test DS: 147 MB |
| Discrete Fourier Transformation (DFT) | Commonly found in digital signal processing applications. | map | map | no | Range: 2048 - 65536 |
| KMeans | Widely used clustering algorithm. | map, groupBy, reduce | map, reduce | yes | Centroids: 2, Points: 16K - 16M |
| IoT Analytics | Encompasses standard operations (e.g., sums, average, etc.) on IoT collected data which, in this paper, have been collected by sensors operating on buildings providing metrics (e.g., temperature, humidity, etc.) | reduce, groupReduce | reduce | yes | 346 MB (Data from IoT Sensors) |

Table 6.2: Software Setup.

| Common Software Characteristics | |
|---|---|
| **Flink** | Apache Flink 1.11 |
| **JVM** | OpenJDK 1.8.0_308, JVMCI 21.2.0 |
| **TornadoVM** | TornadoVM v0.11 |
| **JVM Heap Size** | 32 GB |
| **Testbed-1** | |
| **OS** | Debian 10, Linux-4.19.0-11 |
| **OpenCL Version** | OpenCL 1.2 |
| **NVDIA Driver** | 418.152.00 |
| **Testbed-2** | |
| **OS** | CentOS 7.4.1708 |
| **OpenCL Version** | OpenCL 1.0 |
| **FPGA Driver** | Intel FPGA SDK 17.1 |

comparison between the proposed and baselined systems, all reported results are the averages of ten consecutive end-to-end executions. In addition, the execution

Table 6.3: Hardware Setup.

| Testbed-1 Characteristics | |
|---|---|
| Node 1, Node 2 | |
| **CPU** | Intel(R) Core(TM) i7-4820K @ 3.70GHz |
| **Main Memory** | 64 GB |
| **GPU** | NVIDIA GeForce GTX 1060 |
| **GPU RAM** | 4 GB |
| **Testbed-2 Characteristics** | |
| **CPU** | Intel(R) Core(TM) i7-7700K @ 4.20 GHz |
| **Main Memory** | 64 GB |
| **FPGA** | Nallatech 385a |
| **FPGA RAM** | 4 GB |

times are obtained by the Flink API (Flink *Job Runtime*). That means that the end-to-end execution time of an application that runs via the proposed system on a heterogeneous device encompasses: a) the time for performing the data transfers from the host (CPU) to the device (GPU or FPGA), and vice-versa; b) the kernel execution time on the device, and c) the time spent in the Flink runtime to orchestrate the execution. Furthermore, to analyze the scalability of the proposed system, each benchmark was executed using various input sizes, as presented in the last column of Table 6.1.

The baseline Flink was tested in six different configurations, three were run on a single Task Manager, with the task slots (parallel CPU threads) scaling up from one to two and four, and remaining three configurations were run on two Task Managers, where each of them scaled up again from one up to four CPU threads. Regarding the accelerated system, the performance was analyzed using two configurations, in which the number of Task Managers scaled up from one to two and each Task Manager deployed one parallel CPU thread. Table 6.4 summarizes the configurations used to test the two systems. From this point on, when discussing the different configurations, the following naming convention will be applied:

*(N)-Number of physical nodes - (TS-CPU/GPU)-Number of Task Slots per node*

For example, a configuration marked as *N-2 TS-CPU-4*, indicates that two physical nodes were utilized, each running four CPU threads, and the execution was deployed to the CPU. Similarly, a configuration labeled *N-1 TS-GPU-1* signifies that one physical node with one CPU thread was utilized and the execution was deployed to the available

Table 6.4: Execution Configurations.

| Configuration Name | Physical Nodes | CPU Threads Per Physical Node | Target Device |
|---|---|---|---|
| N-1 TS-CPU-1 | 1 | 1 | CPU |
| N-1 TS-CPU-2 | 1 | 2 | CPU |
| N-1 TS-CPU-4 | 1 | 4 | CPU |
| N-2 TS-CPU-1 | 2 | 1 | CPU |
| N-2 TS-CPU-2 | 2 | 2 | CPU |
| N-2 TS-CPU-4 | 2 | 4 | CPU |
| N-1 TS-GPU-1 | 1 | 1 | GPU |
| N-2 TS-GPU-1 | 2 | 1 | GPU |

GPU. The terms <CPU parallel thread>and <task slot>will be used interchangeably.

## 6.2 Performance Evaluation on GPUs

This section presents the performance comparison of two benchmarks that were executed by the proposed system on a GPU (without hybrid mode) compared to the baseline Apache Flink implementations. The first benchmark is a matrix multiplication between two two-dimensional matrices (Subsection 6.2.1). The second benchmark is an industrial use case that predicts the likelihood of patients to be re-admitted in public hospitals (Subsection 6.2.2) by using logistic regression on patient data.

### 6.2.1 Matrix Multiplication

The matrix multiplication benchmark uses a map operator to multiply two matrices, of two dimensions, on Testbed-1. Figure 6.1 presents the relative performance of matrix multiplication executed by the proposed system against the six configurations of the baseline system, which scale out the number of physical nodes and the number of threads per node. The left-size plot of Figure 6.1 presents the speedups of the Flink-TornadoMV integration while running on the GPU of a single Task Manager node. On the right size of Figure 6.1 are illustrated the speedups acquired when running the integration on two Task Manager nodes, each equipped with a GPU. The horizontal axis groups the experiments for different matrix sizes ranging from 128 to 4096 elements per dimension of each matrix. Specifically, the size of the input datasets ranges from 171KB to 170MB. The vertical axis shows the relative performance speedup of the proposed system against the baseline implementations (six bars) in logarithmic scale.

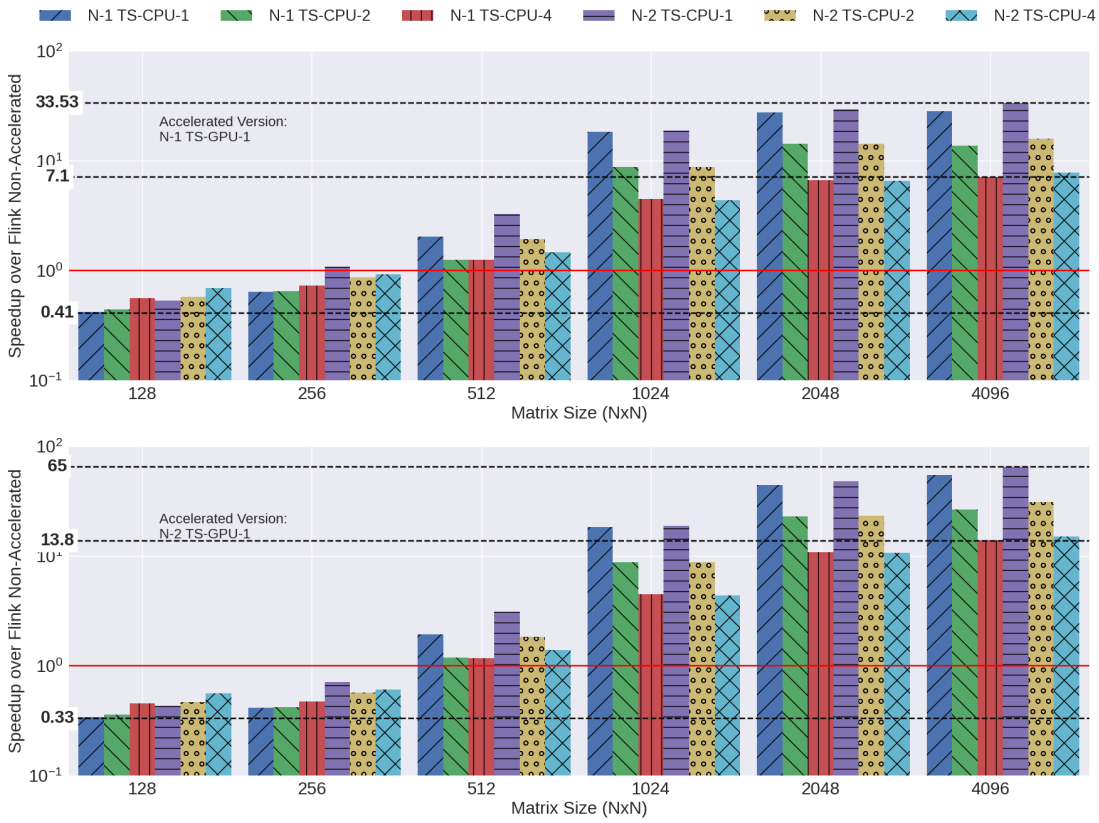


Figure 6.1: Matrix Multiplication: Performance speedup of the proposed system against the baseline Apache Flink configurations. The higher, the better.

The red horizontal line illustrates the reference point that shows equal performance between the compared systems.

As shown in Figure 6.1, the performance of matrix multiplication in the proposed system for sizes smaller than 512 elements (per dimension) does not outperform the execution of the baseline Flink non-accelerated implementation. However, as the data size increases, the proposed system outperforms the non-accelerated Flink by up to 33.53$x$ (Figure 6.1-left) and 65$x$ (Figure 6.1-right). The highest performance improvement is observed against the Flink non-accelerated configuration that runs with two Task Managers and one task slot per Task Manager (N-2 TS-CPU-1) and is depicted in purple. On the other hand, the least performance improvement, for the largest dataset, is 7.1$x$ (Figure 6.1-left) and 13.8$x$ (Figure 6.1-right); and it is observed against the baseline configuration that operates with four threads on a single node (N-1 TS-CPU-4

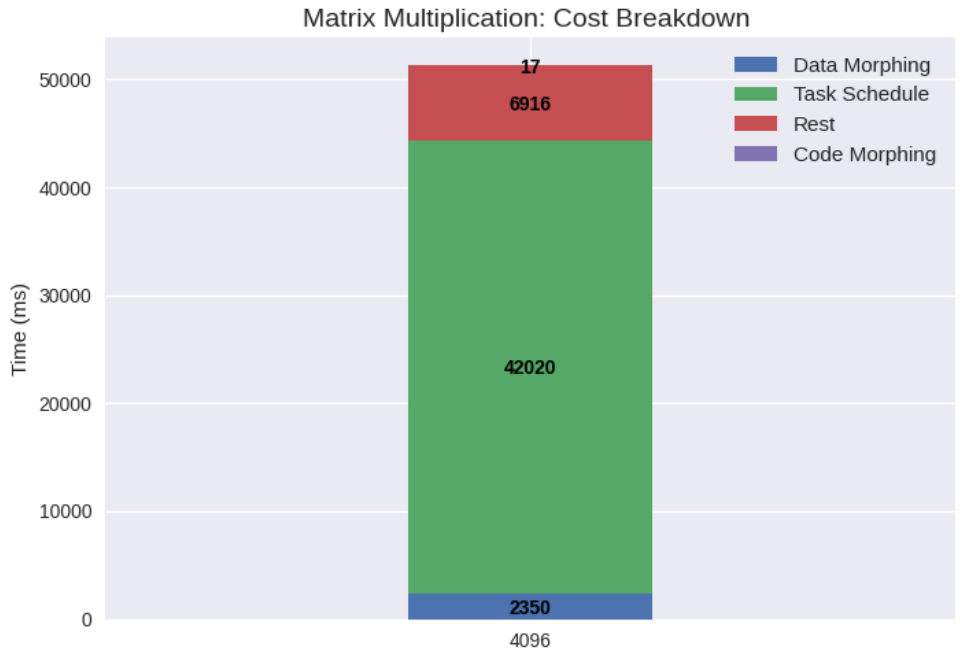


Figure 6.2: A Breakdown Analysis of The Matrix Multiplication Use Case for Matrices of Dimensions 4096x4096.

- red bar). In addition, as the number of CPUs is scaled in the baseline configuration, the measured relative speedup achieved by a single GPU is decreasing.

Additionally, Figure 6.2 presents a breakdown analysis of the execution for the largest dataset (4096 elements per matrix dimension). As shown in this figure, the cost of the code morphing module (purple segment) is almost insignificant (17 ms). In this experiment, the Task Schedule time dominates the execution (green segment), while the transformations applied by the data morphing module (blue segment) take up approximately 4.5% of the total job time. The red segment in the figure represents the remaining job time.

**Conclusion:** As expected, the results of this evaluation indicate that in order to benefit from GPU acceleration, a particular data size threshold must be met to offset the overheads of data transfers to the GPU. The reason is that, when the input data size is small, the cost of transferring low volume of data to the GPU exceeds the actual execution gains of the computation on the GPU, thereby penalizing the overall performance.


Figure 6.3: Execution runtime of Logistic Regression for the baseline Apache Flink configurations and the proposed solution that executes on a GPU. The lower, the better.

### 6.2.2 Logistic Regression (LR)

The Logistic Regression use case deploys a Machine Learning model that is trained with a data set of 1 GB size, while the testing of the trained model is performed with 147 MB of data. The benchmarking of this use case includes both the training and testing phases of the ML model. Each phase consists of three operators, which are executed in the following order: map, reduce, and map. Figure 6.3 presents the overall end-to-end execution time for both systems, as reported by Flink. Similar to the previous experiment, the Flink baseline configurations that were tested using various threads per node, as follows: a) one task slot (blue bar), b) two task slots (green bar), and c) four task slots (red bar). This figure groups the systems in the horizontal axis based on the number of Task Managers and the device that they target. For instance, execution times of the baseline configurations that were run on the CPU of one physical node belong to the group labeled *N-1 CPU*, the same configurations that were executed on two physical nodes belong to the group labeled *N-2 CPU* and so on.

As shown in Figure 6.3, in this benchmark it was observed that the proposed system performs slower than all baseline configurations. In particular, the N-1 TS-CPU-1

Table 6.5: Testbed-3.

| Testbed-3 Characteristics | |
|---|---|
| **CPU** | Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz |
| **Main Memory** | 256 GB |
| **GPU** | NVIDIA Tesla V100-SXM2 |
| **GPU RAM** | 32 GB |

configuration is up to approximately 2.5$x$ faster than the equivalent configuration that is offloaded on the GPU (N-1 TS-GPU-1). Additionally, the fastest configuration of the baseline (which is in this case is four task slots on either one or two Task Managers) is up to 4.7$x$ faster compared to the best configuration of the proposed system (two Task Manager nodes).

### Further Investigation

To understand better the reasons why the GPU implementations perform significantly worse than the baseline implementations, a complementary study was performed. This study includes the breakdown analysis of the end-to-end job time for the proposed system while running on a single physical node (N-1 TS-GPU-1). Figure 6.4 presents the breakdown of the overall job time while running (i) on Testbed-1, which is equipped with the GTX 1060 GPU and, (ii) on a testbed equipped with a Tesla V100 GPU, which a is high-end GPU that offers more compute threads and higher memory capacity. The characteristics of this testbed (Testbed-3) are presented in 6.5.

The execution breakdown consists of four main segments: a) the cost of data marshaling (blue segment), b) the execution time of the map operator on the GPU (green segment), c) the execution time of the reduce operator on the GPU (red segment), and d) the rest of the job run time (purple).

As shown on the left bar of Figure 6.4, the marshaling cost along with the execution of the map and reduce operators dominate the overall time of the Flink job. In particular, the marshaling time takes up to 30743 milliseconds, while the execution time of the two operators on the GPU take up to 33319 and 24787 milliseconds for map and reduce, respectively. However, the breakdown shown on the right bar of Figure 6.4 indicates that, when running on a faster GPU, the job time is dominated by the marshaling cost, which takes more than 80% of the overall time (47088 milliseconds).

Even though the previous breakdown analysis identified the mashalling as a big overhead, this was not the only bottleneck; high execution times were also observed. Specifically, for this benchmark, the performance on the GTX 1060 GPU would still

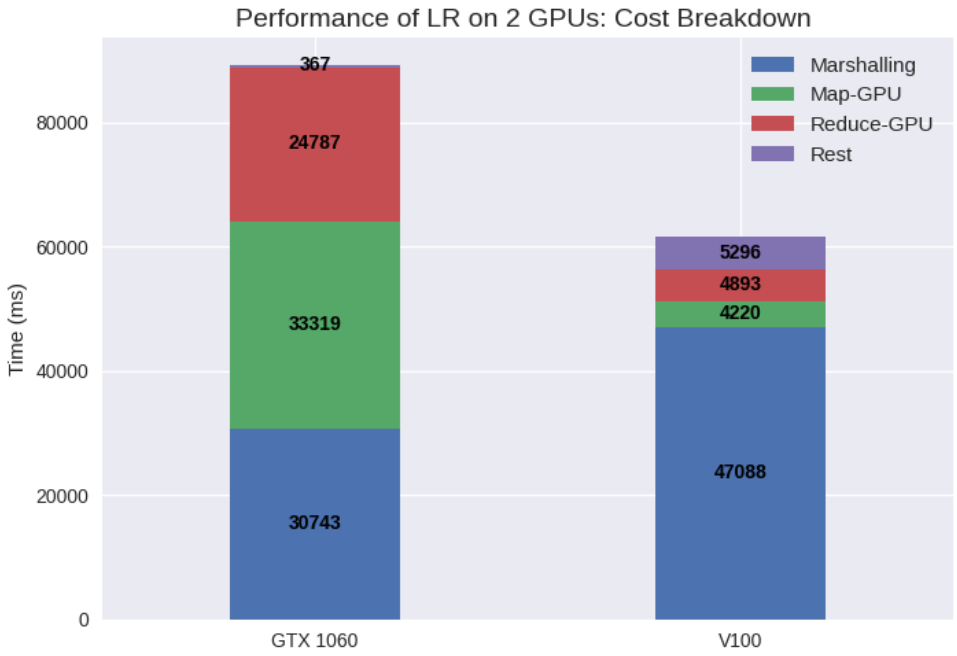


Figure 6.4: Performance breakdown of Logistic Regression on two GPUs a GTX 1060 (left bar) and a Tesla V100 (right bar).

be lower compared to the baseline even if the marshalling cost was completely eliminated. The total execution time on the GTX 1060, removing the marshalling cost is approximately 57933 milliseconds, which is still larger than the equivalent baseline configuration execution time, which is 34599 milliseconds. An initial hypothesis on why in this particular case the computation is slow, even though the application is highly-parallelizable, could be that the GPU is overutilized due to the size of the load (1 GB), which the largest used in this evaluation.

To examine this hypothesis, an additional experiment was implemented. In this experiment, the use case was run on the GTX 1060 GPU of Testbed-1 for datasets ranging from 1GB to 56MB, and the performance of the computation deployed on TornadoVM was measured using the *TornadoVM profiler*. The TornadoVM profiler returns the total time of all Task Schedules - which includes data deployment and kernel execution times. Table 6.6 presents the results of this experiment. Specifically, this table contains the total TaskSchedule time, which encompasses: a) the time for transferring data from the main memory to the GPU memory (Copy-In Time), (b) the kernel execution time (Kernel Time), and (c) the time to move data from the device memory back to the main

Table 6.6: Evaluation of Logistic Regression for various Input Sizes.

| | Data Size | | | |
|---|---|---|---|---|
| | **1 GB** | **888 MB** | **111 MB** | **56 MB** |
| | TaskSchedule Breakdown | | | |
| **TaskSchedule Time** | 58021 | 41603 | 5748 | 3577 |
| **Copy-In Time** | 44922 | 31827 | 4000 | 2012 |
| **Copy-Out Time** | 9731 | 6816 | 850 | 425 |
| **Kernel Time** | 2726 | 2255 | 284 | 143 |
| **Copy-In over TaskSchedule (%)** | 77.4% | 76.5% | 69.6% | 56.2% |
| | Overall Comparison | | | |
| **Flink N-1 TS-CPU-1** | 34559 | 25506 | 3286 | 1792 |
| **Proposed System N-1 TS-GPU-1** | 88876 | 64407 | 8801 | 5336 |
| **Slow-down** | 0.39 | 0.39 | 0.37 | 0.33 |

memory (Copy-Out Time). Moreover, it contains the overall execution time of the proposed system (N-1 TS-GPU-1) and the baseline implementation (N-1 TS-CPU-1). All times are reported in milliseconds.

As shown in Table 6.6, for 1 GB of input, the kernel execution time is 2726 milliseconds, which is very low compared to the end-to-end times reported. However, it appears that what dominates the execution time is the time to transfer data from the main memory to the GPU memory (Copy-In Time). This takes up around 77% of the total Task Schedule execution time. The same trend was observed for smaller datasets as well. The kernels' execution times are consistently low, ranging from 2726 milliseconds (1 GB) to 143 milliseconds (56 MB). On the other hand, the transferring of data from the main memory to the GPU memory takes up to 77.4% and 56.2% of the overall TaskSchedule time for large (1 GB) and small (56 MB) sizes, respectively.

**Conclusion:** The performance evaluation of this benchmark illuminated two crucial factors that should be taken into consideration when striving to achieve the highest performance.

First of all, it showcased that the marshaling cost poses a significant overhead that can severely impact performance. The marshaling cost is attributed to the transformation performed by the data morphing module (presented in Section 5.2), that is, the changes made on the serialized buffer in order to preserve the endianess and aligned memory accesses. This overhead is present in all the experiments discussed in this chapter, however, it becomes visible here due to the large size of the input dataset (1GB). The design decision of the JVM to use big endian serialized buffers while the majority of acceleration hardware uses little-endian can be only addressed via changes to the core runtime system itself. However, if Flink could process information directly

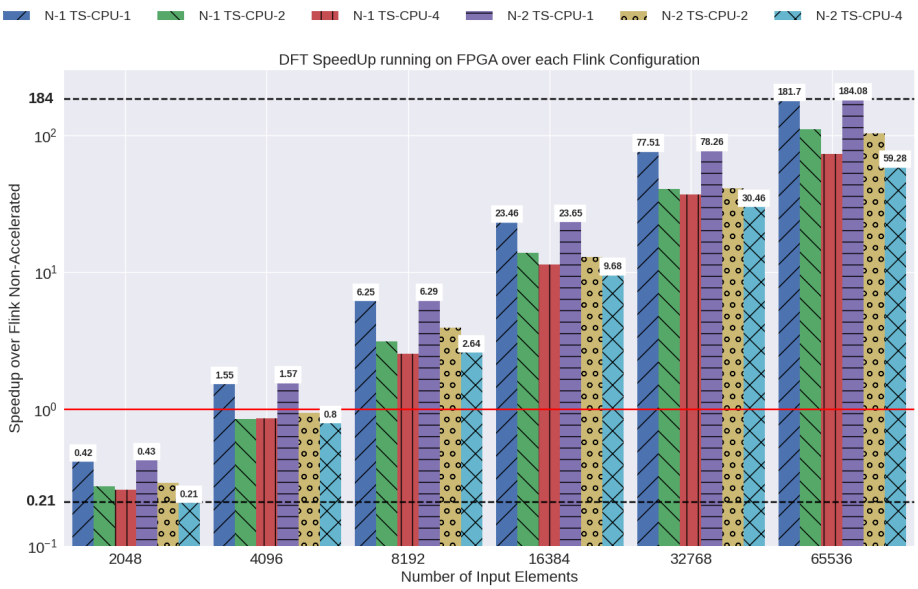


Figure 6.5: DFT: Performance speedup of the proposed system against the baseline Apache Flink configurations. The higher, the better.

for native buffers (not JVM heap-allocated) which can also be directly used by accelerators, the marshaling process would be circumvented and this potential performance gains would be higher. A more immediate solution to this problem would be to use off-heap memory allocation via Project Panama [aia], for example, in order to format data appropriately. Nevertheless, such a solution would impose programmability and memory maintenance challenges at the Apache Flink (or any Big Data framework) side.

However, the data marshalling was not the only overhead contributing to the performance of this benchmark. High data transfer cost was also observed. One possible factor that could be contributing to the fact that the data transfer cost is so high in this use case - but was not in the others benchmarks - could be that this benchmark consists of the most operators. At the current implementation of the proposed platform, each operator is executed by a separate TornadoVM Task Schedule, therefore, there are more data exchanges between operators compared to other use cases. In a future iteration of the proposed system, if these operators were placed on the same Task Schedule, this cost could be much lower.

## 6.3 Performance Evaluation on FPGAs

Due to the interoperability with TornadoVM, the proposed system can target not only GPUs but also FPGAs. This section evaluates the performance of a *Discrete Fourier Transformation (DFT)* algorithm implemented with a map operator in Apache Flink.

### 6.3.1 Discrete Fourier Transformation (DFT)

Figure 6.5 presents the relative performance of the proposed system running on an Intel FPGA versus six baseline configurations. This experiment is executed on a single node of Testbed-2. The horizontal axis shows six bars that correspond to a baseline configuration for various input sizes ranging from 2048 $\sim$ 50KB to 65536 $\sim$ 1.6MB elements for the input arrays. As shown in Figure 6.5, the execution on the FPGA for small input sizes (up to 4096 elements) does not result in performance improvement over the baseline implementations. The only exception where the FPGA execution outperforms the baseline for 4096 input size is observed against the single thread configurations that operate on one (N-1 TS-CPU-1, blue bar) or two (N-2 TS-CPU-1, purple bar) nodes. In this case, the performance improvement is up to 1.55$x$ and 1.57$x$ higher than N-1 TS-CPU-1 and N-2 TS-CPU-1, respectively. For input sizes greater than 4096, the proposed system accelerates all baseline configurations by up to 184$x$ (N-2 TS-CPU-1 for 65536 elements). Additionally, the performance trend shown in Figure 6.5 for all baseline configurations is consistent across all input sizes. For example, the maximum performance between all baseline configurations for the maximum input size is achieved by N-2 TS-CPU-4 (light blue bar), which outperforms up to 3.07$x$ and 3.1$x$ the N-1 TS-CPU-1 (blue bar) and N-2 TS-CPU-1 (purple bar) configurations.

**Conclusion:** Similarly to other works [PFS+21], the execution of parallel workloads that have small data size on FPGAs may not result in acceleration. However, when the amount of data to be processed is sufficient, acceleration is possible. In particular, applications, such as DFT, that can utilize specific hardware units integrated on the FPGA hardware for digital signal processing are good candidates. The capability to transparently utilize both GPUs and FPGAs from the same unmodified Apache Flink code base enables device selection based on the workload characteristics which is the strength of the proposed system.

## 6.4 Performance Evaluation of Hybrid Execution

This section presents the performance comparison of two benchmarks that use the hybrid execution of Apache Flink operators on both the CPU and GPU, against the baseline Apache Flink implementations. The first benchmark is KMeans (Section 6.4), while the second benchmark is an industrial use case that performs analytics on data obtained from IoT sensors (Section 6.4).

### KMeans

Figure 6.6 shows the performance of KMeans on the proposed system against six configurations of the baseline Flink non-accelerated implementations. As shown in Figure 6.6, the execution with the proposed solution on the GPU of Testbed-1 outperforms all the baseline Flink configurations for various input sizes ranging from small (i.e., 32768 elements $\sim$ 1.3MB) to large (i.e., 16777216 elements $\sim$ 649MB). The Flink accelerated version on the GPU performs up to 17.86$x$ (Figure 6.6-left, 4194304 elements, purple bar) and 21.12$x$ (Figure 6.6-right, 16777216 elements, purple bar) faster than the N-2 TS-CPU-1 Flink configuration. Additionally, the proposed system outperforms the fastest baseline Flink configuration for the maximum input size by 4$x$ (Figure 6.6-left, red bar) and 5$x$ (Figure 6.6-right, red bar) against the N-1 TS-CPU-4 and N-2 TS-CPU-4 configurations, respectively.

Note that the KMeans benchmark uses various operators, including map, reduce and group-by. In this experiments, the map and the reduce were deployed on the GPU, while the groupBy operator was executed on the CPU. Figure 6.7 presents a breakdown analysis of the total execution time for the largest dataset (16777216 points). In this figure, the green segment represents the GPU execution time (Task Schedule time), the red segment the CPU execution time (groupBy time), the blue segment the data marshalling cost and finally the purple segment the remaining of the job time. For this use case, it appears that Flink has a lot of internal overhead, since the purple segment dominates the total job time.

**Conclusions:** As shown in Figure 6.6, the proposed system is faster than the baseline for all the configurations and datasets used in this evaluation. This showcases that the cost of having hybrid execution is not necessarily prohibitive, which is a very positive outcome. Through mixed execution of operators across different types of devices (e.g., by using the recommendations of a heterogeneous scheduler [MBD+18]) the system

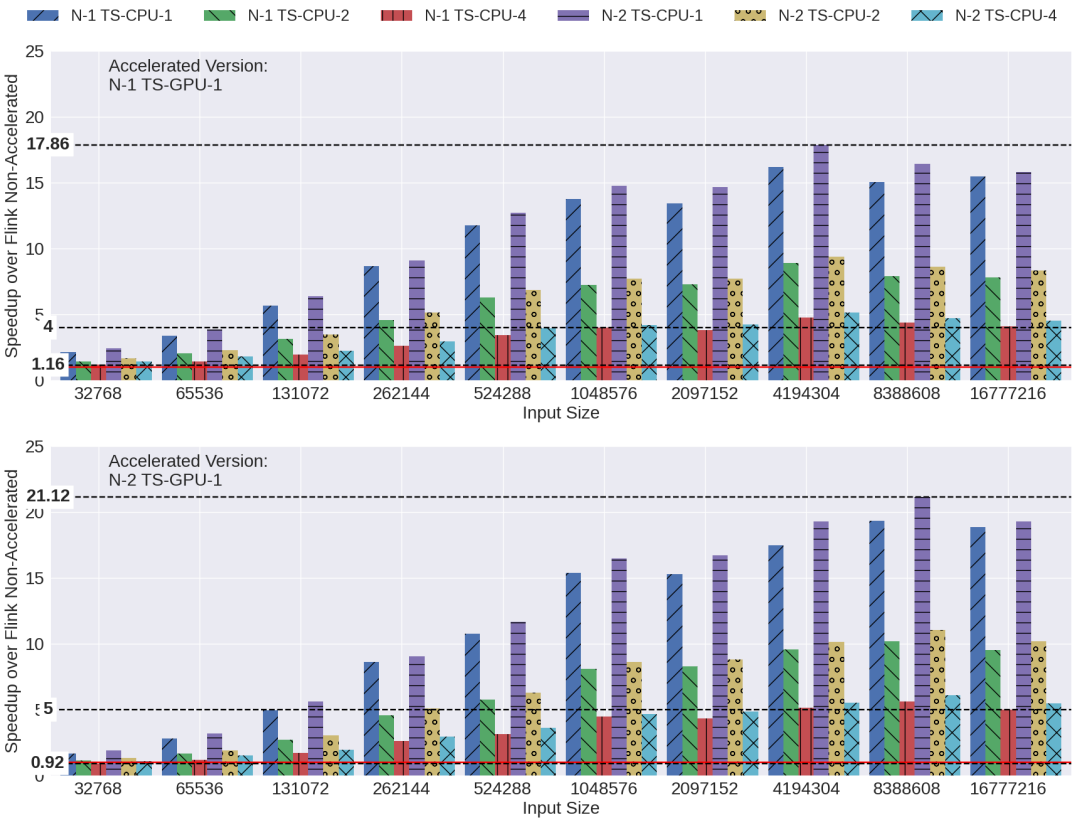


Figure 6.6: KMeans: Performance speedup of the proposed system against the baseline Apache Flink configurations. The higher, the better.

can achieve a higher level of heterogeneity.

### IoT Analytics

The IoT Analytics use case is a benchmark that deploys four reduce operators to calculate `min`, `max`, `mult`, and `sum` operations on the input datasets. This benchmark contains also a *reduceGroup* operator, which is currently not supported for acceleration, and therefore, was executed on the CPU. The datasets are obtained from a network of IoT sensors.

Figure 6.8 contains five plots; each plot presents the comparative evaluation of

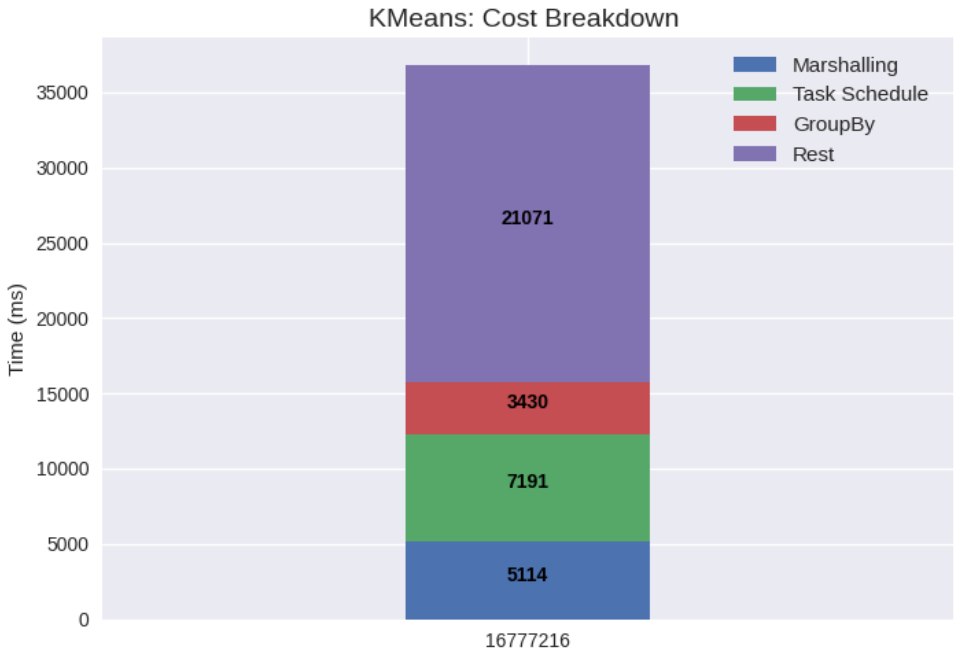


Figure 6.7: A Breakdown Analysis of the KMeans Benchmark for Matrices for 16777216 Points.

a configuration that uses the proposed system to execute on GPUs, against six non-accelerated Flink configurations that have been used as baseline in previous experiments. In this benchmark, the proposed platform was tested in three additional configurations compared to the previous experiments. These additional configurations are on one Task Manager running on the GPU with two (N-1 TS-GPU-2), four (N-1 TS-GPU-4) and eight (N-1 TS-GPU-8) parallel threads. The reason that these extra configurations were examined was that with the original configurations, not enough performance was reported (up to 1.23$x$ on the configuration N-1 TS-GPU-1 and 1.43$x$ on the configuration N-2 TS-GPU-1). Therefore, they served to prove whether under-utilization of the GPU was the root of the problem.

The results presented in Figure 6.8 show that although performance increases with more threads, it does not scale further for more than four threads. In particular, it is shown that the maximum performance improvement is 2.54$x$ and it is achieved by N-1 TS-GPU-4 against the N-1 TS-CPU-1 baseline (purple bar). When the number of threads on the same node is increased to eight threads, the overall performance is increased up to 2.48$x$ against N-1 TS-CPU-1 which indicates saturation.

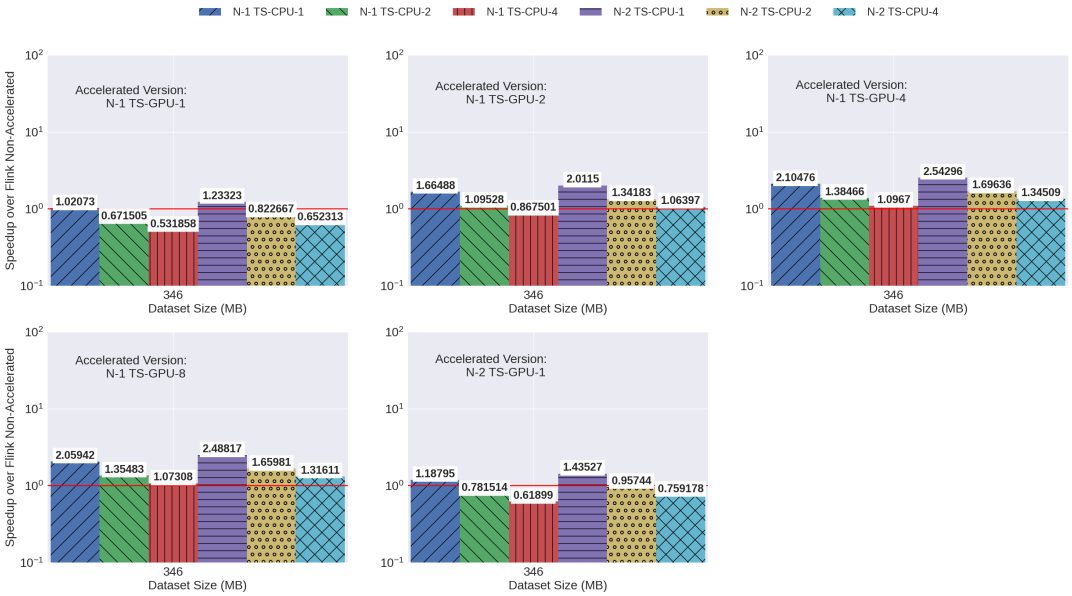


Figure 6.8: IoT Analytics: Performance speedup of the proposed system against the baseline Apache Flink configurations. The higher, the better.

Table 6.7: GPU Utilization on Testbed-1 for the IoT Use Case.

| Task Slots | Utilization Percentage |
|---|---|
| 1 | 100% |
| 2 | 100% |
| 4 | 88% |
| 8 | 92% |

Table 6.8: Testbed-4.

| Testbed-4 Characteristics | |
|---|---|
| **CPU** | Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz |
| **Main Memory** | 64 GB |
| **GPU** | NVIDIA Quadro GP100 |
| **GPU RAM** | 16 GB |

### Further Investigation

Upon further inspection of the GPU utilization using *nvidia-smi*, it was observed for this use case, the GPU was getting overutilized (up to 100% utilization) even when running with one and two threads. Table 6.7 presents the utilization percentages as calculated by nvidia-smi for each thread.

The same experiment was repeated on a testbed equipped with a Quadro GP100

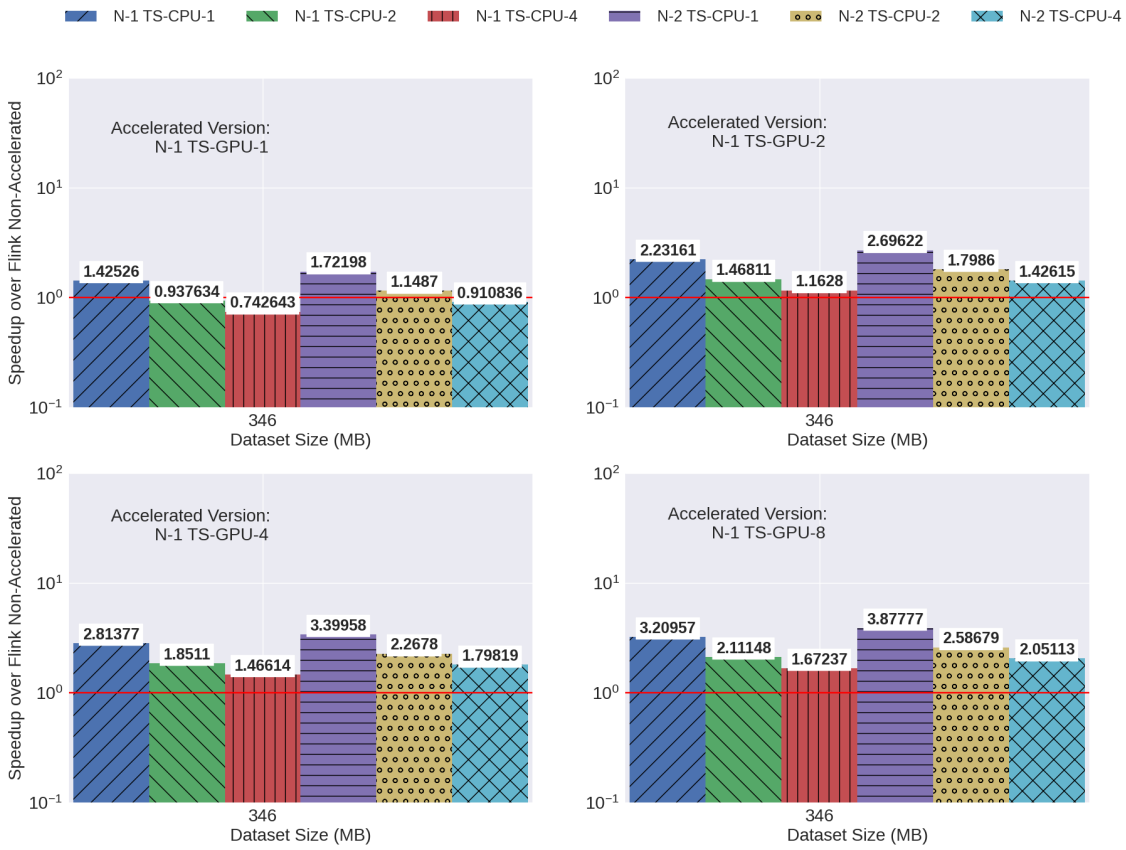


Figure 6.9: IoT Analytics: Performance speedup of the proposed system against the baseline Apache Flink configurations, while running on a faster GPU. The higher, the better.

GPU (Testbed-4, presented in Table 6.8). The GPU of Testbed-4 is considerably faster compared to GTX 1060. The speedups of this experiment are presented in Figure 6.9. As seen in this figure, the accelerated platform performs consistently better compared to the previous results observed (Figure 6.8), with up to 3.8$x$ speedup for eight threads (N-1 TS-GPU-8). With this faster GPU, the utilization is much lower and, therefore, the performance can scale up with more than four threads. Table 6.9 presents the percentage of the GPU utilization on Testbed-4, with the task slots ranging from one to eight. As presented in Table 6.9, the GPU utilization remains below 10% despite the increase in the number of task slots. This could be attributed to the computing power of the GP100 GPU.

Table 6.9: GPU Utilization on Testbed-4 for the IoT Use Case.

| Task Slots | Utilization Percentage |
|---|---|
| 1 | 5% |
| 2 | 9% |
| 4 | 4% |
| 8 | 4% |

**Conclusions:** Overutilization of GPUs can cause performance degradation [MO16]. As the amount of threads increased to four and eight, the utilization percentage dropped to 88% and 92% respectively. The reason that this drop of utilization was observed could be that the GPU driver put pending work in a queue to handle saturation.

## 6.5 Summary

This section presented the evaluation of the proposed platform across a diverse set of applications. The presented framework was tested in three execution different setting; on a GPU, an FPGU and mixed CPU/GPU execution. The largest speedup observed against the Flink baseline on GPUs was 65$x$ for the Matrix Multiplication case, and on the FPGA 184$x$ for the Discrete Fourier Transformation.

This experimental study also identifies the preconditions to gain the most performance out of the hardware accelerators. These can be summarized as follows:

1. The first precondition is to have the correct utilization of the hardware accelerator, both regarding the computational load and the parallel pipelines that are executed. Although this issue might seem trivial, there is no panacea in configuring the right data size and the number of threads to get optimal performance. Instead, the correct configuration highly depends on the application and the underlying hardware resources. As this experimental evaluation showcased, it is imperative to have integrated tools that can automatically perform data partitioning and configure the appropriate degree of parallelism for each application to take full advantage of heterogeneous Big Data platforms.

2. Secondly, it is crucial to limit the data copies from the CPU main memory to the device memory as much as possible, since this process can significantly deteriorate performance.

3. Lastly, it was observed that the marshalling cost, which is imposed by the data transformations of the data morphing module (Section 5.2) is a considerable

overhead for large datasets. To avoid this cost, the runtime system should be augmented to store the data in a format that is friendly to hardware accelerators so as to avoid these transformations.

# Chapter 7

# Conclusions and Future Work

As the amount of data that requires processing keeps increasing at a rapid pace, additional computational power is required to maintain low execution times. To that end, major cloud providers (e.g. AWS) have equipped their cluster nodes with high-throughput devices, such as GPUs and FPGAs, which are known to be performant in intensive computations. However, taking advantage of such hardware accelerators is not trivial, since the de-facto Big Data frameworks (e.g. Apache Flink, Apache Spark, etc.) were designed to run on CPU-only clusters.

Over the last few years, both the industry and the academia have worked extensively on filling the gaps between Big Data frameworks and hardware accelerators, and several solutions for heterogeneous execution of Big Data applications have emerged. However, these solutions have some characteristics that make their wide adoption by Big Data developers challenging. Specifically, in order to accelerate Big Data applications with any of the currently available frameworks, the developers have to either (i) provide a low-level implementation of their application in a programming language like OpenCL, (ii) rewrite their application to match a platform-specific API, or (iii) both. Apart from the programmability constrains, many of the existing heterogeneous Big Data frameworks are restricted to devices of specific vendors (e.g. NVIDIA) and/or they can only target specific accelerator devices (e.g. GPUs only).

This thesis provided an extensive analysis of the current obstacles in seamless heterogeneous execution of Big Data applications and introduced a set of techniques that enable the dynamic execution of Big Data loads on GPUs and FPGAs without user interference. Section 7.1 will provide a summary of the contributions made in this thesis, as they were presented in each chapter. Section 7.2 will present how this thesis addressed each of the research questions and draw the final conclusions. Finally,

Section 7.3 will suggest future research directions.

## 7.1 Summary

The content of this thesis is summarized as follows:

Chapter 2 provided the essential background on the core concepts of this thesis. Specifically, it introduced Big Data frameworks, presented a general overview of their architecture, and classified them based on their programming model (i.e., map-reduce and dataflow). Additionally, it defined the focus of this thesis, which is on distributed dataflow systems, and provided a deep analysis of Apache Flink, which is the platform that was used for prototyping. Lastly, it elaborated on the terms *hardware accelerators* and *heterogeneous execution* and provided a high-level description of GPUs and FPGAs.

Chapter 3 scrutinized the current state-of-the art and presented a detailed analysis of the most prominent efforts to introduce GPU and FPGA acceleration into both Big Data frameworks and map-reduce programming in general. The acceleration of map-reduce applications using GPUs and FPGAs has been widely studied. However, most the map-reduce frameworks in the relevant literature expose low-level APIs and often require a deep understanding of the hardware architectures from the developers. Regarding the works on accelerating Big Data applications, they can be classified into two categories. The first category consists of frameworks that rely on precompiled kernels. The second category contains frameworks that employ dynamic compilation. Regardless of the category they fall into, all these heterogeneous frameworks were evaluated based on the following criteria: (i) if they cause code fragmentation, (ii) if their kernels are pre-compiled or generated on-demand, (iii) if they impose any vendor lock-in, and (iv) if they can target CPUs, GPUs and FPGAs. As demonstrated in this chapter, there is currently no implementation that can target CPUs, GPUs and FPGAs on demand, without causing the fragmentation of the codebase. This is the gap that this thesis studies and fills.

Chapter 4 discussed the challenges of enabling transparent heterogeneous execution on Big Data frameworks. First, it elaborated on the need to use a heterogeneous programming framework in order to avoid pre-compiled kernels and presented TornadoVM, which is the heterogeneous framework of choice for this thesis. Following, three major challenges that stand in the way of transparent acceleration of Big Data applications were identified. The first challenge is to enable the execution of Big Data

applications, which are written using a specific API, on GPUs and FPGAs, without modifying the user code. The second challenge refers the granularity of execution followed by Big Data platforms versus the granularity of execution that is suitable for hardware accelerators. The last challenge is how to handle dynamic types (i.e. JVM objects) on GPUs and FPGAs so as not to break the OOP norms. Additionally, this section also presented a quick and naive integration of Flink and TornadoVM. This integration does not have much novelty value, however, it served two purposes. The first was to establish that an integration of Flink and TornadoVM is possible and the second was to verify the challenges discussed above.

Chapter 5 presented a set of techniques that enable dynamic execution of Big Data applications on heterogeneous hardware devices without breaking the programming norms. The techniques used were split into three modules. First, a code morphing module adapts the user functions at runtime to make them compatible with the API of the heterogeneous compiler (in this case, TornadoVM). Secondly, a data morphing module was presented, that collects the data from the Flink serialization buffers and performs a set of transformations to make it suitable for hardware acceleration. These transformations are performed with respect to endianess and the alignment of memory accesses. Lastly, a code generator module was introduced, which consists of an extended version of TornadoVM. Specifically, the compiler of TornadoVM was augmented with a set of compiler phases that enable the user function to access the data correctly. This chapter also discussed how the proposed framework handles operators that are not currently supported.

Finally, Chapter 6 presented an extensive evaluation of the proposed implementation using a wide set of benchmarks. During this evaluation it was showcased that indeed, in most cases, Big Data applications can benefit greatly from hardware accelerators (achieving speedups up $65x$ on GPUs and up to $184x$ on FPGAs over Flink CPU scaleout). This evaluation also highlighted a set of preconditions that need to be satisfied to ensure that a Big Data application gains the most out of GPU and FPGA execution. These are (i) to minimize the copying from the host to the device as much as possible, (ii) to ensure proper utilization of devices, because both under and over utilization can lead to performance degradation, and (iii) to design future Big Data platforms in such a way as to minimize or eliminate marshalling, since this can be a significant overhead.

## 7.2 Conclusions

This section presents the conclusions that can be drawn for each of the research questions presented in Section 1.2.

**RQ1. What are the main obstacles in transparent acceleration of Big Data applications?**

Through the analysis of the programming and execution models of the two basis systems, Apache Flink and TornadoVM, and the initial efforts to integrate them, the following challenges were identified:

- **#1 Challenge:** How to enable the execution of applications that are written using a specific API of a Big Data framework on GPUs and FPGAs.
- **#2 Challenge:** How to handle Java object types on GPUs and FPGAs.
- **#3 Challenge:** How to use a coarse-grained model of execution on a Big Data platform, while developers write their code in a fine-grained model.

**RQ2. Is it possible to accelerate Big Data applications without breaking the existing Big Data programs or introducing new APIs?**

Chapter 5 proved that it is indeed possible to offer hardware acceleration of Big Data applications, without introducing new APIs or requiring any user intervention. This thesis proposed three techniques to achieve that, (i) code morphing, with the user application being made compatible with the API of a heterogeneous compiler on-the-fly, (ii) data morphing, where the data is transformed automatically by the system into a format that is friendly to hardware accelerators and, (iii) a set of compiler phases that ensure correct memory accesses.

**RQ3. What are the preconditions of hardware acceleration and the performance tradeoffs of seamless execution of Big Data applications on GPUs and/or FPGAs?**

As demonstrated through the experimental evaluation of the proposed framework (Chapter 6), the ability to automatically target heterogeneous hardware devices can, in some cases, put a strain on performance instead of offering speedup.

The experimental analysis showcased that, sometimes, these costs can occur by the misusage of the hardware resources, for instance by running with the wrong number of threads leading to over/under-utilization, or by having multiple data transfers from the host to the device and vice versa. Therefore, to obtain performance and keep the system agnostic to the user, special tools that can automatically configure the execution should be integrated. Moreover, another large overhead that was observed had to do

with data marshalling. This overhead exists mainly because Big Data frameworks were not designed with hardware accelerators in mind. Therefore, even though their format of data is natural for high-level, object oriented programming, it is not suitable for low-level heterogeneous devices.

In conclusion, this thesis identified the following preconditions should be satisfied in order to make the most of the hardware accelerators:

1. First of all, the correct utilization of the hardware accelerator is essential, both regarding the computational load and the parallel pipelines that are executed. Tools that can automatically perform data partitioning and thread configuration should ideally be used, for the system to remain hardware-agnostic and to avoid human error.

2. Secondly, it is crucial to limit the data copies from the CPU main memory to the device memory as much as possible, since this process can significantly deteriorate performance.

3. Lastly, the runtime system should be augmented to store the data in a format that is friendly to hardware accelerators so as to avoid these transformations.

## 7.3 Future Work

The development of the proposed framework laid the ground for numerous exciting research questions to be explored. Moreover, if further extended, it could potentially revolutionize Big Data execution by making heterogeneous acceleration the new norm. Some of the future directions that could be followed to extend the presented heterogeneous platform are the following.

1. Identify more operators that can be benefited from hardware acceleration and enhance the compiler framework to support them.

2. Provide a mechanism that can automatically determine the optimal data partitioning strategy among hardware accelerators. As demonstrated in the experimental evaluation, combining the decision about on which device an operator should be executed with the correct granularity of data can have a huge impact on performance.

3. Develop a fault-tolerance mechanism suitable for heterogeneous execution. Most Big Data platforms are equipped with mechanisms that ensure recovery from failures on CPU clusters. Nevertheless, these mechanisms were not designed with heterogeneous accelerators in mind. As an example, if a Flink computation crashes, Flink checkpoints help resume the computation without losing the whole progress. However, if a GPU goes offline during a long-running computation, as there is no context switching between GPU and CPU threads, all the progress made will be lost. Since it is not realistic to assume that real-world Big Data applications will not suffer from crashes, it is imperative to provide an error-handling mechanism that is also efficient for hardware accelerators in the future.

4. Re-design the existing Big Data frameworks to eliminate the overheads the were identified during the experimental evaluation process. Specifically, first of all, enhance Big Data platforms to store the data internally in a format that is friendly to hardware accelerators. Secondly, include a hardware-aware scheduler, that distributes the computations among the hardware resources in a way that minimizes the copying costs.

# Bibliography


[aia] Oracle Corporation and/or its affiliates. Project Panama: Interconnecting JVM and native code, https://openjdk.java.net/projects/panama/.

[AKA12] Amin Abbasi, Farshad Khunjush, and Reza Azimi. A preliminary study of incorporating gpus in the hadoop framework. In The 16th CSI International Symposium on Computer Architecture and Digital Systems (CADS 2012), pages 178–185, 2012.

[apa16] Aparapi, https://aparapi.com/, Accessed in 2016.

[Azu] Microsoft Azure. https://azure.microsoft.com/.

[BH12] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. 2012.

[BK13] Can Basaran and Kyoung-Don Kang. Grex: An efficient mapreduce framework for graphics processing units. J. Parallel Distributed Comput., 73:522–533, 2013.

[Bru] Eric Bruneton. ASM 4.0 A Java bytecode engineering library, https://asm.ow2.io/asm4-guide.pdf.

[BSFK22] Florin Blanaru, Athanasios Stratikopoulos, Juan Fumero, and Christos Kotselidis. Enabling pipeline parallelism in heterogeneous managed runtime environments via batch processing. In Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2022, page 58–71, New York, NY, USA, 2022. Association for Computing Machinery.

[CA12] Linchuan Chen and Gagan Agrawal. Optimizing mapreduce for gpus with effective shared memory usage. In HPDC '12, 2012.

[CCF+16] Yu-Ting Chen, Jason Cong, Zhenman Fang, Jie Lei, and Peng Wei. When apache spark meets fpgas: a case study for next-generation dna sequencing acceleration. In CloudCom 2016, 2016.

[CFP+18a] James Clarkson, Juan Fumero, Michail Papadimitriou, Maria Xekalaki, and Christos Kotselidis. Towards practical heterogeneous virtual machines. In Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Programming'18 Companion, page 46–48, New York, NY, USA, 2018. Association for Computing Machinery.

[CFP+18b] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. Exploiting high-performance heterogeneous hardware for java programs using graal. In Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang '18, New York, NY, USA, 2018. Association for Computing Machinery.

[CH02] Katherine Compton and Scott Hauck. Reconfigurable Computing: A Survey of Systems and Software. ACM Comput. Surv., 2002.

[CHA12] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. 2012 International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–11, 2012.

[CJ15] Woohyuk Choi and Won-Ki Jeong. Vispark: Gpu-accelerated distributed visual computing using spark. In LDAV, 2015.

[Clo] Google Cloud. https://cloud.google.com/.

[CLO+16] Cen Chen, Kenli Li, Aijia Ouyang, Zeng Zeng, and Keqin Li. Gflink: An in-memory computing architecture on heterogeneous cpu-gpu clusters for big data. IEEE Transactions on Parallel and Distributed Systems, 29:1275–1288, 2016.

[CLOL18] Cen Chen, Kenli Li, Aijia Ouyang, and Keqin Li. Flinkcl: An opencl-based in-memory computing architecture on heterogeneous cpu-gpu clusters for big data. IEEE Transactions on Computers, 67:1765–1779, 2018.

[Col89] Murray Cole. Algorithmic skeletons: Structured management of parallel computation. 1989.

[Cor21a] NVIDIA Corporation. Nvidia gpudirect, https://developer.nvidia.com/gpudirect, 2021.

[Cor21b] NVIDIA Corporation. CUDA Toolkit Documentation, https://docs.nvidia.com/cuda/, Accessed in 2021.

[COTL17] Cen Chen, Aijia Ouyang, Zhuo Tang, and Keqin Li. Gpu-accelerated parallel hierarchical extreme learning machine on flink for big data. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 47:2740–2753, 2017.

[CQJ+13] Yi Chen, Zhi Qiao, Hai Jiang, Kuan-Ching Li, and Won Woo Ro. Mgmr: Multi-gpu based mapreduce. In GPC, 2013.

[CS14] Yuk-Ming Choi and Hayden Kwok-Hay So. Map-reduce processing of k-means algorithm with fpga-accelerated computer cluster. 2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, pages 9–16, 2014.

[CSK08] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A map reduce framework for programming graphics processors. In In Workshop on Software Tools for MultiCore Systems, 2008.

[CXT+15] Zhenhua Chen, Jielong Xu, Jian Tang, Kevin A. Kwiat, and Charles A. Kamhoua. G-storm: Gpu-enabled high-throughput online data processing in storm. 2015 IEEE International Conference on Big Data (Big Data), pages 307–312, 2015.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In OSDI, 2004.

[DWS+13] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages, VMIL '13, page 1–10, New York, NY, USA, 2013. Association for Computing Machinery.

[EHHB14a] Ismail El-Helw, Rutger F. H. Hofman, and Henri E. Bal. Glasswing: accelerating mapreduce on multi-core and many-core clusters. In HPDC '14, 2014.

[EHHB14b] Ismail El-Helw, Rutger F. H. Hofman, and Henri E. Bal. Scaling mapreduce vertically and horizontally. SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 525–535, 2014.

[ELcFS11] Marwa K. Elteir, Heshan Lin, Wu chun Feng, and Thomas R. W. Scogland. Streammr: An optimized mapreduce framework for amd gpus. 2011 IEEE 17th International Conference on Parallel and Distributed Systems, pages 364–371, 2011.

[FM05] James Fung and Steve Mann. OpenVIDIA: Parallel GPU Computer Vision. In Proceedings of the 13th Annual ACM International Conference on Multimedia, MULTIMEDIA '05. Association for Computing Machinery, 2005.

[Fou21a] Apache Software Foundation. Apache hadoop, https://hadoop.apache.org/, Accessed in 2021.

[Fou21b] Apache Software Foundation. Apache ignite, https://ignite.apache.org/, Accessed in 2021.

[Fou21c] Apache Software Foundation. Apache storm, https://storm.apache.org/, Accessed in 2021.

[Fou22a] Apache Software Foundation. Apache samza, https://samza.apache.org/, Accessed in 2022.

[Fou22b] The Apache Software Foundation. Accelerating your workload with GPU and other external resources, Accessed in July 2022.

[FPZ+19] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. Dynamic application reconfiguration on heterogeneous hardware. In Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2019, page 165–178, New York, NY, USA, 2019. Association for Computing Machinery.

[FVCC09] Reza Farivar, Abhishek Verma, Ellick Chan, and Roy H. Campbell. Mithra: Multiple data independent tasks on a heterogeneous resource architecture. 2009 IEEE International Conference on Cluster Computing and Workshops, pages 1–10, 2009.

[GBS13] Max Grossman, Maurício Breternitz, and Vivek Sarkar. Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl. 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, pages 1918–1927, 2013.

[GBS16] Max Grossman, Maurício Breternitz, and Vivek Sarkar. Hadoopcl2: Motivating the design of a distributed, heterogeneous programming system with machine-learning applications. IEEE Transactions on Parallel and Distributed Systems, 27:762–775, 2016.

[GC16] Ehsan Ghasemi and Paul Chow. Accelerating apache spark big data analysis with fpgas. 2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), pages 737–744, 2016.

[GC19] Ehsan Ghasemi and Paul Chow. Accelerating apache spark with fpgas. Concurrency and Computation: Practice and Experience, 31, 2019.

[GCS+] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. tiresias: A GPU cluster manager for distributed deep learning.

[GIS15] Max Grossman, Shams Mahmood Imam, and Vivek Sarkar. Hj-opencl: Reducing the gap between the jvm and accelerators. Proceedings of the Principles and Practices of Programming on The Java Platform, 2015.

[Gro21] Khronos OpneCL Working Group. The opencl c specification, https://www.khronos.org/registry/opencl/specs/3.0-unified/html/opencl_c.html, Accessed in 2021.

[GS16] Max Grossman and Vivek Sarkar. Swat: A programmable, in-memory, distributed, high-performance computing platform. Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, 2016.

[GZYE20] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU Kernels for Deep Learning. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, 2020.

[HCC+10] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. Mapcg: Writing parallel program portable between cpu and gpu. 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 217–226, 2010.

[HCJ17] Sumin Hong, Woohyuk Choi, and Won-Ki Jeong. Gpu in-memory processing using spark for iterative computation. 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pages 31–41, 2017.

[HCJ21] Sumin Hong, Junyoung Choi, and Won-Ki Jeong. Distributed interactive visualization using gpu-optimized spark. IEEE Transactions on Visualization and Computer Graphics, 27:3670–3684, 2021.

[HCL+15] Wenting He, Huimin Cui, Binbin Lu, Jiacheng Zhao, Shengmei Li, Gong Ruan, Jingling Xue, Xiaobing Feng, Wensen Yang, and Youliang Yan. Hadoop+: Modeling and evaluating the heterogeneity for mapreduce applications in heterogeneous clusters. Proceedings of the 29th ACM on International Conference on Supercomputing, 2015.

[HFL+08] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A mapreduce framework on graphics processors. 2008 International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 260–269, 2008.

[Hut22a] Marco Hutter. Java bindings for CUBLAS, Accessed in July 2022.

[Hut22b] Marco Hutter. Java bindings for CUDA, Accessed in July 2022.

[HWY+16] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. Programming and runtime support

to blaze fpga accelerator deployment at datacenter scale. Proceedings of the Seventh ACM Symposium on Cloud Computing, 2016.

[HZK+18] Junjie Hou, Yongxin Zhu, Linghe Kong, Zhe Wang, Sen Du, Shijin Song, and Tian Huang. A case study of accelerating apache spark with fpga. In 2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), pages 855–860, 2018.

[INT22] https://neapi.io INTEL. OneAPI, Accessed in 2022.

[JA12] Wei Jiang and Gagan Agrawal. Mate-cg: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters. 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pages 644–655, 2012.

[JM11] Feng Ji and Xiaosong Ma. Using shared memory to accelerate mapreduce on graphics processing units. 2011 IEEE International Parallel & Distributed Processing Symposium, pages 805–816, 2011.

[KCR+17] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '17. Association for Computing Machinery, 2017.

[KKO+18] David Koeplinger, Christos Kozyrakis, Kunle Olukotun, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, and Ardavan Pedram. Spatial: a language and compiler for application accelerators. ACM SIGPLAN Notices, 53:296–311, 06 2018.

[KY21] Madiha Khalid and Muhammad Murtaza Yousaf. A comparative analysis of big data frameworks: An adoption perspective. Applied Sciences, 2021.

[LA04] Chris Lattner and Vikram S. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. International Symposium on Code Generation and Optimization, 2004. CGO 2004., pages 75–86, 2004.

[LC13] Zhongduo Lin and Paul Chow. Zcluster: A zynq-based hadoop cluster. 2013 International Conference on Field-Programmable Technology (FPT), pages 450–453, 2013.

[LHWW21] Zhifang Li, Mingcong Han, Shangwei Wu, and Chuliang Weng. Shadowvm: accelerating data plane for data analytics with bare metal cpus and gpus. Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2021.

[LLZC15] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. 2015 IEEE International Conference on Networking, Architecture and Storage (NAS), pages 347–348, 2015.

[LOR12] Yu Lin, Semih Okur, and Cosmin Radoi. Hadoop+aparapi: Making heterogenous mapreduce programming easier. 2012.

[MAKA11] Reza Mokhtari, Amin Abbasi, Farshad Khunjush, and Reza Azimi. Soren: Adaptive mapreduce for programmable gpus. 2011.

[MBD+18] Ioannis Mytilinis, Constantinos Bitsakos, Katerina Doka, Ioannis Konstantinou, and Nectarios Koziris. The vision of a heterogenerous scheduler. pages 302–307, 12 2018.

[MO16] Christos Margiolas and Michael F. P. O'Boyle. Portable and transparent software managed scheduling on accelerators for fair resource sharing. In Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO '16, page 82–93, New York, NY, USA, 2016. Association for Computing Machinery.

[MT16] D. Manzi and David Tompkins. Exploring gpu acceleration of apache spark. 2016 IEEE International Conference on Cloud Engineering (IC2E), pages 222–223, 2016.

[NMG+15] Katayoun Neshatpour, Maria Malik, Mohammad Ali Ghodrat, Avesta Sasan, and Houman Homayoun. Energy-efficient acceleration of big data analytics applications using fpgas. 2015 IEEE International Conference on Big Data (Big Data), pages 115–123, 2015.

[NMGH15] Katayoun Neshatpour, Maria Malik, Mohammad Ali Ghodrat, and Houman Homayoun. Accelerating big data analytics using fpgas. 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, pages 164–164, 2015.

[NMH15] Katayoun Neshatpour, Maria Malik, and Houman Homayoun. Accelerating machine learning kernel in hadoop using fpgas. 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pages 1151–1154, 2015.

[NSH16] Katayoun Neshatpour, Avesta Sasan, and Houman Homayoun. Big data analytics on heterogeneous accelerator architectures. 2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pages 1–3, 2016.

[NVI22] NVIDIA. Cuda, https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#addresses-as-operands, Accessed in 2022.

[Obj22] https://asm.ow2.io ObjectWeb. ASM, Accessed in 2022.

[OHL+08] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Gpu computing. Proceedings of the Institute of Radio Engineers, 96(5):879–899, May 2008.

[OMM16] Yasuhiro Ohno, Shin Morishima, and Hiroki Matsutani. Accelerating spark rdd operations with local and remote gpu devices. 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS), pages 791–799, 2016.

[Ora22] Oracle. The java virtual machine specification, https://docs.oracle.com/javase/specs/jvms/se7/html/, Accessed in 2022.

[PFS+21] Michail Papadimitriou, Juan José Fumero, Athanasios Stratikopoulos, Foivos S. Zakkak, and Christos Kotselidis. Transparent compiler and

runtime specializations for accelerating managed languages on fpgas. Art Sci. Eng. Program., 5:8, 2021.

[PFSK21] Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. Automatically exploiting the memory hierarchy of gpus through just-in-time compilation. In Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2021, page 57–70, New York, NY, USA, 2021. Association for Computing Machinery.

[PMF+21] Michail Papadimitriou, Eleni Markou, Juan Fumero, Athanasios Stratikopoulos, Florin Blanaru, and Christos Kotselidis. Multiple-tasks on multiple-devices (mtmd): Exploiting concurrency in heterogeneous managed runtimes. In Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2021, page 125–138, New York, NY, USA, 2021. Association for Computing Machinery.

[Pro21] The Open MPI Project. Open mpi:open source high performance computing, https://www.open-mpi.org/, Accessed in 2021.

[rap21] Rapids, https://rapids.ai/, Accessed in 2021.

[RE20] Jason Lowe Robert Evans. Deep Dive into GPU Support in Apache Spark 3.x, 2020. SPARK+AI SUMMIT 2020.

[RGR18] David Reinsel, John Gantz, and John Rydning. The Digitization of the World: From Edge to Core . Technical report, IDC, 01 2018.

[RMCA10] Vignesh T. Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In ICS '10, 2010.

[Ros] John Rose. Jep 243: Java-level jvm compiler interface., https://openjdk.java.net/jeps/243.

[RSA+17] M. Mazhar Rathore, Hojae Son, Awais Ahmad, Anand Paul, and Gwanggil Jeon. Real-time big data stream processing using gpu with spark over hadoop ecosystem. International Journal of Parallel Programming, 46:630–646, 2017.

[SBK20] Amir Hossein Sojoodi, Majid Salimi Beni, and Farshad Khunjush. Ignite-gpu: a gpu-enabled in-memory computing architecture on clusters. The Journal of Supercomputing, 77:3165–3192, 2020.

[SCMO10] Jeff A. Stuart, Cheng-Kai Chen, Kwan-Liu Ma, and John Douglas Owens. Multi-gpu volume rendering using mapreduce. In HPDC '10, 2010.

[SCN+15] Oren Segal, Philip Colangelo, Nasibeh Nasiri, Zhuo Qian, and Martin Margala. Sparkcl: A unified programming framework for accelerators on heterogeneous clusters. ArXiv, abs/1505.01120, 2015.

[Ser] Amazon Web Services. https://aws.amazon.com/hpc/.

[SGS10] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. Computing in Science Engineering, 2010.

[SI09] Shigeyuki Sato and Hideya Iwasaki. A skeletal parallel framework with fusion optimizer for gpgpu programming. In APLAS, 2009.

[SKG11] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. Skelcl - a portable skeleton library for high-level gpu programming. 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pages 1176–1182, 2011.

[SKKS18] Ioannis Stamelos, Elias Koromilas, Christoforos Kachris, and Dimitrios Soudris. A novel framework for the seamless integration of fpga accelerators with big data analytics frameworks in heterogeneous data centers. In 2018 International Conference on High Performance Computing Simulation (HPCS), pages 539–545, 2018.

[SO11] Jeff A. Stuart and John Douglas Owens. Multi-gpu mapreduce on gpu clusters. 2011 IEEE International Parallel & Distributed Processing Symposium, pages 1068–1079, 2011.

[SOV+20] Athanasios Stratikopoulos, Mihai-Cristian Olteanu, Ian Vaughan, Zoran Sevarac, Nikos Foutris, Juan Fumero, and Christos Kotselidis. Transparent acceleration of java-based deep learning engines. MPLR 2020. Association for Computing Machinery, 2020.

[Spa] Apache Spark. Job scheduling, https://spark.apache.org/docs/latest/job-scheduling.html.

[SRD17] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A functional data-parallel ir for high-performance gpu code generation. 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 74–85, 2017.

[SSE15] Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann. Heterodoop: A mapreduce programming system for accelerator clusters. Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, 2015.

[SSM10] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Hybrid map task scheduling for gpu-based heterogeneous clusters. 2010 IEEE Second International Conference on Cloud Computing Technology and Science, pages 733–740, 2010.

[SWY+10] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. Fpmr: Mapreduce framework on fpga a case study of rankboost acceleration. 2010.

[Tha18] Lauritz Thamsen. Dynamic Resource Allocation for Distributed Dataflows. PhD thesis, Elektrotechnik und Informatik der Technischen Universität Berlin, 2018.

[TL10] Kuen Hung Tsoi and Wayne Luk. Axel: a heterogeneous cluster with fpgas and gpus. In FPGA '10, 2010.

[TLHC12] Yu Shyang Tan, Bu-Sung Lee, Bingsheng He, and Roy H. Campbell. A map-reduce based framework for heterogeneous processing element cluster environments. In 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), pages 57–64, 2012.

[tor] TornadoVM Run your software faster and simpler!, https://www.tornadovm.org/.

[VAA19] Tudor Alexandru Voicu and Zaid Al-Ars. Sparkjni: A toolchain for hardware accelerated big data apache spark. 2019 IEEE 4th International Conference on Big Data Analytics (ICBDA), pages 152–157, 2019.

[VMD+13] Vinod Vavilapalli, Arun Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: yet another resource negotiator. 10 2013.

[WHI+19] Song Wu, Die Hu, Shadi Ibrahim, Hai Jin, Jiang Xiao, Fei Chen, and Haikun Liu. When fpga-accelerator meets stream data processing in the edge. 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pages 1818–1829, 2019.

[WWS10] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for java. In Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, PPPJ '10, page 10–19, New York, NY, USA, 2010. Association for Computing Machinery.

[WWW+13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013, page 187–204, New York, NY, USA, 2013. Association for Computing Machinery.

[XFK18a] Maria Xekalaki, Juan Fumero, and Christos Kotselidis. Challenges and proposals for enabling dynamic heterogeneous execution of big data frameworks. In 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pages 335–341, 2018.

[XFK18b] Maria Xekalaki, Juan Fumero, and Christos Kotselidis. Dynamic acceleration of big data applications on heterogeneous hardware resources. 14th International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems, ACACES 2018, Poster abstracts, 2018.

[Xil21] Inc. Xilinx. Vitis Quantitative Finance Library, Accessed in 2021.

[XKB13] Mengjun Xie, Kyoung-Don Kang, and Can Basaran. Moim: A multi-gpu mapreduce framework. 2013 IEEE 16th International Conference on Computational Science and Engineering, pages 1279–1286, 2013.

[YSH+16] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. Spark-gpu: An accelerated in-memory data processing engine on clusters. 2016 IEEE International Conference on Big Data (Big Data), pages 273–283, 2016.

[YTT+08] Jackson H. C. Yeung, Chi Chiu Tsang, Kuen Hung Tsoi, Bill S. H. Kwan, Chris C. C. Cheung, Anthony Chan, and Philip Heng Wai Leong. Map-reduce as a programming model for custom computing machines. 2008 16th International Symposium on Field-Programmable Custom Computing Machines, pages 149–159, 2008.

[Zag20] Andrey Zagrebin. Improvements in task scheduling for batch workloads in apache flink, https://flink.apache.org/2020/12/15/pipelined-region-sheduling.html#the-new-pipelined-region-scheduling, 15 Dec 2020.

[ZLH+14] Jie Zhu, Juanjuan Li, Erikson Hardesty, Hai Jiang, and Kuan-Ching Li. Gpu-in-hadoop: Enabling mapreduce across distributed heterogeneous platforms. 2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS), pages 321–326, 2014.