Enabling Transparent Acceleration of Big Data Frameworks Using Heterogeneous Hardware

Maria Xekalaki±, Juan Fumero±, Athanasios Stratikopoulos±, Katerina Doka⊤, Christos Katsakioris⊤, Constantinos Bitsakos⊤, Nectarios Koziris⊤, Christos Kotselidis±
±The University of Manchester, UK
{first}.{last}@manchester.ac.uk
⊤National Technical University of Athens, Greece
{doka,ckatsak,kbitsak,nkoziris}@cslab.ece.ntua.gr

ABSTRACT

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 since the end-user Application Programming Interface (APIs) must be altered to access the underlying heterogeneous hardware. For example, current Big Data frameworks, such as Apache Spark, provide a new API that combines the existing Spark programming model with GPUs. For other Big Data frameworks, such as Flink, the integration of GPUs and FPGAs is achieved via external API calls that bypass their execution models completely.

In this paper, we rethink current Big Data frameworks from a systems and programming language perspective, and introduce a novel co-designed approach for integrating hardware acceleration into their execution models. The novelty of our approach is attributed to two key design decisions: a) support for arbitrary User Defined Functions (UDFs), and b) no modifications to the user level API. The proposed approach has been prototyped in the context of Apache Flink, and enables unmodified applications written in Java to run on heterogeneous hardware, such as GPU and FPGAs, transparently to the users. The performance evaluation of the proposed solution has shown performance speedups of up to 65x on GPUs and 184x on FPGAs for suitable workloads of standard benchmarks and industrial use cases against vanilla Flink running on traditional multi-core CPUs.

PVLDB Reference Format:
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. PVLDB, 15(13): 3869 - 3882, 2022.
doi:10.14778/3565838.3565842

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/mairooni/Flink-TornadoVM-Artifact.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 13 ISSN 2150-8097.
doi:10.14778/3565838.3565842

1 INTRODUCTION

The staggering increase in the generation rate of data - which is predicted to reach 163 zettabytes by 2025 [59] - has posed new challenges and opportunities regarding high-performance and energy efficient data analytics. To address these challenges, 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. Hardware accelerators can now be found in almost all modern cloud providers, such as AWS, Google Cloud, and Microsoft Azure, complementing traditional CPU-only execution for accelerating suitable workloads.

In order to exploit these hardware accelerators, developers must write their code in specific programming languages and frameworks, such as CUDA [14], OpenCL [26, 66], and OneAPI [34]. In the domain of Big Data analytics, developers typically use already established systems that are typically written in managed programming languages (e.g., Java) and run on top of the Java Virtual Machine (JVM) [52]. This three-layered execution model (user application - Big Data framework - runtime system) makes the integration of heterogeneous hardware accelerators very complex, as the majority of the frameworks and runtime systems have been designed with CPU-only execution in mind. Except for Nvidia GPU acceleration for Spark 3.x [60] via the RAPIDS API [1], and for Flink [18] via JCuda [32] and/or JCublas [31], the remaining of existing works are mostly academic efforts to bring heterogeneous hardware acceleration on various Big Data frameworks (e.g., [7, 23, 62]). A common denominator of all the aforementioned approaches is the introduction of specific APIs that must be used in order to access the underlying hardware accelerators. This characteristic has several disadvantages, such as code fragmentation, vendor lock-in, lack of portability, and most importantly limited UDF support.

In this paper, we take a step back and we rethink the three-layered execution model that current Big Data frameworks utilize. We approach the challenge of hardware acceleration from the systems and programming language perspective (bottom up) following a co-designed approach. The result of this methodology is a proposed system with the ability to run unmodified Big Data applications across multiple hardware accelerators, transparently to the users. We prototyped our solution in the context of Flink [6] and TornadoVM [19], thereby enabling developers to transparently run their existing vanilla code on GPUs and FPGAs. It is important

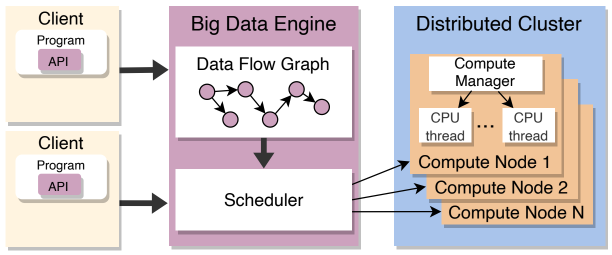


Figure 1: Overview of Big Data Frameworks.

to note that the techniques described in this paper are generally transferable to other frameworks and programming environments.

In detail, this paper makes the following contributions:

- It elaborates on the whole execution stack of the Big Data framework of choice and discusses the challenges of heterogeneous execution for each layer (user application, Big Data framework, and runtime system).
- It presents a novel approach for enabling automatic and transparent GPU and FPGA acceleration of existing Flink user programs written in Java. To achieve that, two novel techniques are introduced: 1) automatic code and data morphing, and 2) application of Just-In-Time (JIT) compilation for heterogeneous hardware, in the context of TornadoVM.
- It performs a performance evaluation of the proposed system across a variety of benchmarks and real-world industrial use cases against the vanilla CPU-only Flink and current GPU support in Flink. The performance analysis showcases speedups of up to 65x when running on GPUs, and up to 184x when running on FPGAs compared to Flink.
- It discusses the merits of heterogeneous hardware acceleration, while also highlighting the pre-conditions that must exist in order to observe performance improvements when running on GPUs and FPGAs compared to scale-out CPU only configurations.

2 BACKGROUND

This section presents the background on Big Data platforms (Section 2.1) and hardware acceleration (Section 2.2). Additionally, it discusses the state-of-the-art approaches for integrating hardware acceleration into Big Data frameworks (Section 2.3). Finally, Section 2.4 presents the challenges that hinder the utilization of hardware acceleration from within existing Big Data frameworks.

2.1 Overview of Big Data Frameworks

Big Data frameworks have as ultimate goal to scale out execution on multiple compute nodes, as a means to increase performance. Figure 1 presents the execution model of such frameworks. As shown in the left part of the figure (*Client* side), developers can express the workflow of their applications as a chain of various operators. Typically, each framework exposes an API that contains operators, such as `map`, `reduce`, `groupBy`, etc. As soon as developers express their applications in the form of operators, a data-flow graph that models the dependency between those operators is constructed.

Figure 2 presents an example that shows a `map` function written in Java with the Flink API to express a simple computation; in this case, a multiplication. Line 1 defines a Java class, named `C`, that

```
1 class C implements MapFunction<Tuple2<Double,Integer>,Double>{
2    Double map(Tuple2<Double, Integer> t) {
3         return t.f0 * t.f1;
4    }
5 }
```

Figure 2: Multiplication in Flink Using a map Function.

implements the `MapFunction` Flink interface which enables code to be expressed as a lambda function or a method reference. The class signature indicates that a pair of `Double` and `Integer` values which belong to a `Tuple2` object will be consumed to perform a map operation and the outcome will be of `Double` type. Lines 2-4 present a map function that multiplies the first field of the input `Tuple2` which is a value of type `Double` with the second field which is a value of type `Integer`. Note that the computation expressed in Flink operators, such as *map*, *reduce*, etc., will be performed per element of the input data.

Near the Clients, is the *Big Data Engine* which utilizes information within the data-flow graphs that have been created by the *Clients*. The information obtained from the analysis of a graph is consumed by the *Scheduler*, which creates an execution plan that distributes data across multiple compute nodes in a cluster (right part of Figure 1). Each compute node within a cluster contains a compute manager and several CPU threads. A compute manager deploys the user-defined operators onto a number of CPU threads based on the plan that is orchestrated by the *Scheduler*.

Since this paper uses Flink for prototyping, we present the Flink terminology that corresponds to the entities depicted in Figure 1, as follows. The programs that are being developed within the clients employ the Flink API along with the supported user-defined operators. These operators are intercepted by the *Job Manager*, the Flink Big Data engine that distributes the computation across the available *Task Managers* (i.e., one or more Task Managers per compute node). Each *Task Manager* can deploy multiple *Task Slots* which can match the number of CPU cores in a compute node.

2.2 Hardware Acceleration

To program hardware accelerators, several programming models can be used (e.g., OpenCL, CUDA, and OneAPI). The prime goal of these models is to ease programming by exposing a unified way of coding that is applicable to every device type. 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 device memory (e.g., GPU 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. Depending on the device type (GPU or FPGA), different workloads and/or algorithmic patterns can perform better or worse. For example, GPUs deliver fine-grain execution as they comprise thousands of threads that execute the same instructions with different input data items (SIMD) [53]. In contrast, FPGAs offer coarse-grain execution as they combine various on-device resources (i.e., blocks of memory, registers, logic slices) to compose diverse hardware blocks [13].

2.3 Acceleration of Big Data Frameworks

To integrate the various heterogeneous programming models described in the previous subsection, Big Data frameworks both at

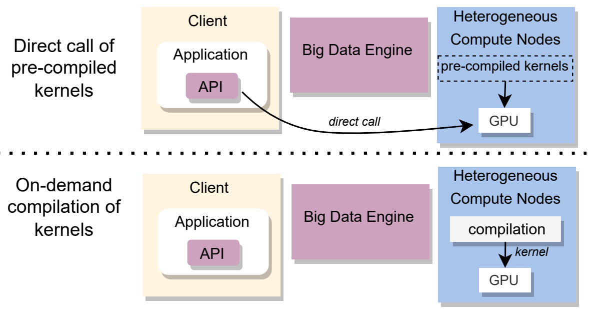


Figure 3: Approaches for the Integration of Heterogeneous Hardware Execution within Big Data Frameworks.

the industry and academia have mainly adopted two approaches. As shown in Figure 3-up, the first approach is to directly call a compute kernel from within the user-level API of the frameworks. This way assumes the presence and installation of a pre-compiled kernel (usually written in CUDA or OpenCL) at every compute node that acceleration will take place.

The second approach, depicted in Figure 3-down, uses a form of compilation to create the kernel on-demand. Depending on the framework, different levels of integration exist. For example, Spark [60] with RAPIDS [1] offers the most comprehensive solution where multiple operators are supported. On the other hand, FlinkCL [7] that uses Aparapi [3] for compilation, is limited by the capabilities of the Aparapi framework that includes user level annotations, explicit memory management, etc. Regardless of the type (pre-compiled or on-demand kernel generation) and the state (production or academic) of integration, current solutions have several limitations, as follows:

- **Code fragmentation:** Developers need to decide apriori which parts of their code should run on a hardware accelerator and code them with specific APIs. This leads to code fragmentation since programmers need to maintain multiple versions of their code (traditional scale-out CPU-only and hardware accelerated versions).
- **Lack of transparency:** Since developers must use specific APIs to access hardware accelerators, they need to reason about performance, code suitability, and device selection in advance.
- **Vendor lock-in:** In some cases (e.g., Spark/RAPIDS), hardware acceleration comes at the cost of vendor lock-in since only particular devices from particular vendors are integrated into the system of choice. Hence, migration across devices becomes a challenging task.
- **Device coverage:** Similarly to the previous challenge, the selection of a specific solution for hardware acceleration might exclude a particular type of devices that could be more suitable for a specific workload. For example, Spark/RAPIDS supports only Nvidia GPUs. Therefore, to use an FPGA to accelerate some specific suitable functions, another approach must be seeked.

Thus, a natural question that arises is: *Can we utilize hardware acceleration in the same seamless way that we execute on CPUs?* To answer this question a software analysis of the three-layered architecture that current frameworks utilize is required, to highlight the challenges that must be addressed at each layer in order to enable seamless heterogeneous hardware acceleration.

2.4 Challenges

This section presents the challenges at each layer of the software architecture that currently inhibit transparent hardware acceleration of Big Data frameworks:

2.4.1 User Level API. Regardless of the framework and its custom API, all operators (built-in or UDFs) at some stage in the execution pipeline will be JIT compiled by the underlying JVM to machine code to be executed. Hence, in order to transparently accelerate such operators, it is essential that hardware acceleration support is built natively inside the JVM; similarly to how automatic vectorization is supported. Currently, such support is very limited with only a handful of frameworks or JVMs supporting it; namely, TornadoVM, Aparapi, and IBM J9 [33]. All aforementioned frameworks provide their own Java-level custom API to enable hardware acceleration which must be exposed to developers of Big Data frameworks. Unfortunately, the API exposure contributes to code fragmentation and lack of transparency out-of-the-box. In addition, each state-of-the-art heterogeneous JVM framework has its own limitations (e.g., limited device support, limitations in Java-supported features, etc.).

Naturally, someone could completely bypass the JVM and offload custom pre-built kernels (Figure 3-up), but this solution does not fulfil the requirements of transparent hardware acceleration that has been presented in Section 2.3.

2.4.2 Big Data Framework. Currently, most Big Data frameworks have been designed to operate under the scale-out CPU-only paradigm. In addition, they typically employ a resource scheduler (e.g., Apache Yarn) to inquire about available resources and schedule tasks for execution according to the preferred scheduling policy. In order for a hardware accelerator to be visible - and hence utilized by a framework - the auxiliary software components must be also altered in order to expose those devices to the framework. Fortunately, work into this direction is underway [4], with latest versions of Yarn, for example, exposing GPUs to the resource scheduler. Despite the work being done into this direction, a number of challenges remain:

(1) **Fall-back mechanisms:** If we operate under the assumption that operators or UDFs will be JIT compiled at the node-level for execution on GPUs or FPGAs, we must account for fallback mechanisms in case compilation fails or hardware acceleration does not result in increased performance.
(2) **Checkpointing:** Current Big Data frameworks employ checkpoint functionalities [17, 64] to save the work that has been done up to a point of an unforeseen failure. To achieve that, they save the execution state at a user-defined granularity and restore execution from that point upon node recovery or migration. However, in the case that code is executed on a GPU, it is imperative that large amounts of data will be processed in batches in order to benefit from the high throughput of such devices. Hence, in the presence of a node failure, the risk of losing a large amount of work is higher compared to fine-grained execution on CPUs. Of course, developers could add checkpoint functionality inside the GPU executed code, but the cost of control flow divergence inside the accelerated kernels will negate all achieved performance benefits [15].
(3) **Scheduling algorithms:** Similarly to the previous challenge, when hardware accelerators are employed, we must ensure that

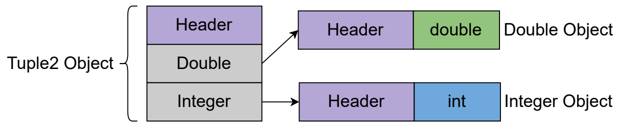


Figure 4: Layout of the Flink *Tuple2<Double, Integer>* **Object.**

they are provisioned with large data sets to counteract the costly data transfer times from the CPU main memory via PCIe. Hence, the execution time of the accelerated task is less predictable compared to a CPU-only execution due to the uniformity of resource and data distribution. In addition, if the data size fluctuates, it might be the case that hardware acceleration may or may not outperform CPU execution. To account for all those cases, the resource scheduler must be modified and become reactive to performance fluctuations.

2.4.3 Runtime System. Java Virtual Machines that host the execution of the majority of Big Data frameworks are designed around the Java specification which in some cases inhibits hardware acceleration. Two prime factors that negatively influence the ability to compile and execute user code on hardware accelerators are:

(1) **Memory Management and Object Representation:** Big data frameworks offer various custom data structures which are implemented as memory objects in Java. However, Java is a managed programming language, and therefore, it relies on its runtime system to perform automatic memory management of the Java applications. This feature contradicts the semantics of static memory allocation, which is supported by current heterogeneous devices, thereby hindering the compilation of custom objects, such as Flink's `Tuples` and Java `Collections`, from Java to OpenCL and CUDA.

Furthermore, the memory layout of the data structures in Java is different than in C-based languages (e.g., OpenCL C). Figure 4 shows the memory layout of a Java `Tuple2` object that contains two objects of types `Double` and `Integer`. For each object, there is a header that includes a `klass` pointer, flags, and locks. Following that, there are two references stored, one to a `Double` wrapper and one to an `Integer` wrapper. In turn, these wrapper classes contain their own headers along with the `double` and `int` values. On the contrary, in C-based languages there is not such memory indirection because the data are stored in a contiguous memory area, and they are managed *by value* rather than *by reference*.

(2) **Irregular Data Endianness:** Typically, Big Data frameworks employ the Java serialization mechanism [51] to transfer data during execution. Java serializers by default use Big Endian, while the vast majority of hardware co-processors use Little Endian. Therefore, if someone attempts to copy a serialized byte buffer from a node's CPU main memory to a GPUs memory, the execution will fail due to the incompatibility of the data layout. Hence, a performance penalty due to *marshaling* will be imposed, because data must be transformed from Big Endian to Little Endian before being copied to the device memory; and vice versa when the result is copied back to the main memory.

3 ENABLING TRANSPARENT ACCELERATION

This section presents our approach for addressing all four aforementioned limitations (namely; code fragmentation, lack of transparency, vendor lock-in, and device coverage - Section 2.3) with respect to hardware acceleration of Big Data frameworks. To achieve

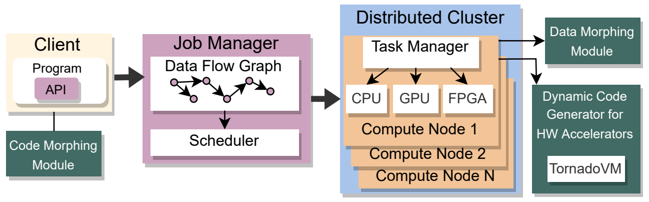


Figure 5: Architecture of the Proposed System.

that, we follow a co-designed approach in which we co-engineer the software layers of a Big Data framework to tackle the majority of the challenges listed in Section 2.4. This section describes the proposed modifications in Flink that enable seamless hardware acceleration on GPUs and FPGAs. Nonetheless, the proposed solution can be applied to other Big Data engines (e.g., Spark, Hadoop).

Figure 5 presents the overall architecture of the proposed system. As shown, the proposed extensions are performed in three steps: 1) the *code morphing module* that resides in the Flink Client and dynamically adapts Flink UDFs to TornadoVM-compatible code; 2) the *data morphing module* that resides in the Task Manager and modifies the data layout to be accessible by the generated kernels; and 3) the dynamic code generation extensions to TornadoVM that automatically generate GPU and FPGA kernels for execution.

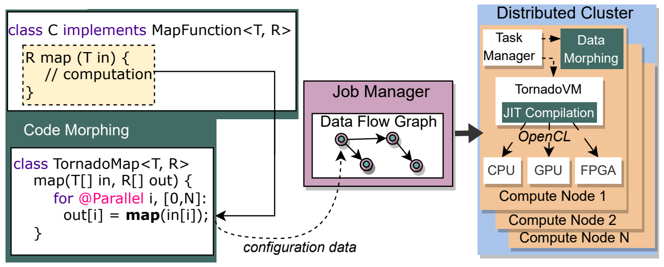
3.1 TornadoVM

To enable transparent hardware acceleration, we employ and augment the TornadoVM [19] framework. TornadoVM is an open-source plugin to various JVM distributions (e.g., OpenJDK) for accelerating applications on multi-core CPUs, GPUs and FPGAs [12, 54, 55, 57, 67]. Unlike other heterogeneous programming frameworks, such as Aparapi [3] or IBM J9 [33], TornadoVM offers automatic code specialization at the compiler level. Additionally, TornadoVM offers the ability to seamlessly migrate the execution from one heterogeneous device (e.g., a GPU) to another (e.g., an FPGA) [19], and orchestrate multiple tasks to run concurrently on multiple heterogeneous devices [57].

TornadoVM API: The TornadoVM API expresses parallelism at the task level, where a task is a reference to a Java method. The API is designed to enable programmers to chain tasks in groups (called `TaskSchedules`). These groups can either execute all tasks in consecutive order on the same device (e.g., the same GPU) [19], or employ the multiple-tasks on multiple-devices (MTMD) mechanism to enable concurrent execution across all available devices [57]. TornadoVM exposes two annotations (**`@Parallel`** and **`@Reduce`**) to express loop parallelism and parallel reduce operations, respectively. The former annotation is used to inform the TornadoVM JIT compiler that a loop can be executed in parallel, while the latter indicates that a variable will contain the result of a reduction. The main characteristic of the TornadoVM API is that it allows Java programmers to exploit hardware parallelism without requiring any knowledge of OpenCL/CUDA or hardware architecture.

3.2 Code Morphing

To enable transparent hardware acceleration in Flink, the user defined code (hereafter referred to as *UDFs*) must be adapted to the TornadoVM API. To achieve this, we employ a code morphing technique that dynamically adapts UDFs to TornadoVM-compatible


```
class C implements MapFunction<T, R>
 R map (T in) {
    // computation
 }
```

```
class TornadoMap<T, R>
   map(T[] in, R[] out) {
      for @Parallel i, [0,N]:
         out[i] = map(in[i]);
   }
```

Figure 6: The Workflow of the Proposed System that Combines Code Morphing, Data Morphing and JIT Compilation for Heterogeneous Hardware.

code via on-the-fly bytecode rewriting. In a nutshell, the code morphing module performs two key operations: a) it adapts the method signatures of UDFs, and b) it transforms the implicit parallelism of Flink to explicit parallelism that is used by TornadoVM (and all other heterogeneous programming frameworks, such as OpenCL).

Figure 6 depicts the code morphing operations (left) as well as the remaining stages of the execution pipeline (compilation and execution). As shown in Figure 6-left-top, a vanilla Flink map operator expresses parallelism in an implicit and fine-grain manner (per element). This way of expressing parallelism is incompatible with current heterogeneous programming frameworks in which parallelism is explicit and various operations are performed in primitive arrays instead of Java objects. Hence, the code morphing module dynamically rewrites Flink expressions as illustrated in Figure 6-left-bottom. The key modifications are the replacement of generic input and output objects to generic array types, and the transition from implicit parallelism to explicit parallelism (`@Parallel` annotated explicit loop). The code rewriting is performed with the help of skeleton classes and dynamic code manipulation via ASM [49].

The *code morphing* module provides specific templates of skeleton functions for various Flink operators, such as `map` and `reduce` which are automatically selected and used during the bytecode rewriting phase. The first step of the bytecode rewriting phase includes the creation of an ASM *ClassReader* instance to retrieve the input and output types of the user function; this information is extracted from the signature of the user function. In the next step, a new ASM *ClassWriter* instance utilizes the retrieved information and places the Flink UDF inside the `for` loop of the matched skeleton function (Figure 6-left-bottom). In this case, the UDF performs a mapping from `T` to `R` as exemplified in Figure 2 (`T` ⟶ `Tuple2<Double, Integer[] input>`, `R` ⟶ `Double`). Finally, the ASM *ClassWriter* returns the skeleton classes in a byte form.

To send this information along with the types of inputs and outputs for each operator to a *Task Manager*, the proposed system stores them as configuration data within the vertices (i.e., `Job Vertex`) of the *Job Graph*. The *Job Graph* is a data-flow graph generated by a Flink Client and consumed by the `Job Manager` for the creation of an execution plan. Furthermore, the *Job Graph* is forwarded to the *Task Managers*, which deploy the execution of the scheduled tasks. Each *Task Manager* contains multiple Flink drivers (e.g., `MapDriver`, `ReduceDriver`, `ChainedMapDriver` etc.) which are responsible for: (i) retrieving the skeleton code along with the input/output types, and (ii) creating the TornadoVM `TaskSchedule` object. This object is used to: a) orchestrate the JIT compilation of the Java UDFs to OpenCL and the data transfers of inputs/outputs, and b) launch the execution on a heterogeneous device (CPU, GPU, FPGA). To provide coverage for all drivers, we have attached a `TaskSchedule` creation module in all supported Flink drivers within a *Task Manager*. Upon the creation of a `TaskSchedule`, a *Task Manager* triggers the JIT compilation from Java to OpenCL, as shown in Figure 6.

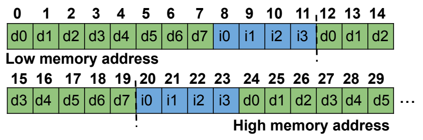


Figure 7: Serialized Array Performed by Flink.

Note that the current implementation of the code morphing module operates on each Flink operator individually; therefore, a separate `TaskSchedule` object is created for each Flink operator. In future, we plan to optimize the efficiency of data movements by fusing chained Flink operators in a shared `TaskSchedule` object.

3.3 Data Morphing

As mentioned in Section 2.4, heterogeneous programming languages, such as OpenCL or CUDA, do not support Java objects or dynamic memory allocation. In addition, the majority of commercial hardware accelerators are Little Endian architectures in contrast to the Big Endian data serialization mechanism of the JVM. To circumvent both of these challenges, we employ a data morphing module which allows the direct execution of kernels on serialized byte buffers generated by Flink. In detail, the data morphing module performs the marshaling and padding of input data and the unmarshaling of output data. In addition, the TornadoVM compiler has been co-designed with several extensions to allow instruction generation for reading/writing data (Section 3.4).

3.3.1 Data Serialization. Since Flink is a distributed system, the data is serialized in order to be transferred between the *Flink Client*, the *Job Manager*, and the *Task Managers* over the network. Similarly to all Java-based frameworks, Flink also uses Java serializers to transform objects to byte arrays. In the case of Flink, the serialized data is transferred in byte format and it is stored in a pre-defined memory region. Once the data arrives at its destination and is ready to be processed, it is deserialized.

In essence, prior to the execution of the UDFs, the Flink *Task Manager* has to access the serialization region, deserialize the data to the form of Java objects, and pass the objects as input to the Java UDF. As explained in Section 2.4.3, several metadata is stored inside each Java object including the header, flags and locks. To avoid the data type conversions from the byte array to Java objects (due to deserialization) and from Java objects to primitive array types (to compile to OpenCL), we have extended the TornadoVM API to accept the byte arrays directly from the Flink memory region. Hence, the proposed system obtains the byte array in the *Task Manager* and performs some offset manipulation at the compiler level, as will be discussed in Section 3.4.

3.3.2 Data Marshaling & Padding. To unify the memory layout of data between the Java representation and the OpenCL C representation, used by a kernel on heterogeneous hardware, we apply *marshaling*; a transformation that keeps only the values of the

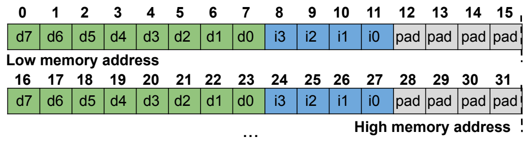


Figure 8: Serialized Array after Marshaling and Padding.

data in primitive types. For example, a `Tuple` object composed of a `Double` and an `Integer` Java object, is transformed to keep only the values of the fields in primitive types (e.g., `double` and `int`). Figure 7 shows the representation of a byte array that corresponds to the serialized values of multiple Flink `Tuple` objects that have the two above-mentioned fields. In this example, each `Tuple` object consists of 12 bytes, eight bytes for the `double` field (depicted in green) and four bytes for the `int` field (depicted in blue). The first `Tuple` object corresponds to the range from byte 0 to byte 11, while bytes 12 to 17 belong to the second `Tuple` object.

As shown in Figure 7, the serialized buffers use Java's big-endian layout which contradicts the little-endian architectures of modern hardware accelerators. To address this incompatibility, the data morphing module automatically reverses the bytes of the byte array for each field within an object, as shown in Figure 8. Furthermore, the byte arrays that store the results of the computation are created using the same semantics. Excluding marshaling, a secondary operation that is performed by the data morphing module is padding in order to avoid unaligned memory accesses. Since the byte buffer to be processed may consist of multiple types (e.g., interleaved `double` and `int` values), it may be the case that an access to a value inside the byte buffer is unaligned with respect to the memory alignment requirements of the underlying architecture. For example, in Nvidia GPUs unaligned memory accesses can yield undefined results [48], while Intel GPUs can handle unaligned memory accesses; albeit, with reduced performance. To solve this problem, we perform padding on different types within the byte buffer, as shown in Figure 8 for the `int` and `double` values.

3.4 JIT compilation

As described in Section 3.3.1, the TornadoVM API has been enhanced to access data directly from a serialized byte array. Although this decision has mitigated any further copies that could impact performance due to deserialization, it requires further modifications at the compiler level to ensure that the generated OpenCL kernels will access the correct data. Thus, the proposed solution introduces several transparent compiler phases in the TornadoVM JIT compiler as follows:

(1) **Object Replacement:** This phase 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. This compiler phase takes place during the high-tier compilation stage and prepares the compiler graph for the later stages (low-tier) in which the offset calculations of the load/store accesses of the arrays will be performed.
(2) **Padding Offset Calculation:** This phase calculates the offsets of the primitive values that are stored in the byte array. The serialized byte arrays are collections of Java objects derived from Flink (e.g, Tuples). To know at which offset a specific value is held, extra information about the fields is required (e.g., type). This information is transmitted as configuration data within a `Job Vertex` (Section 3.2). Then, to obtain the position of each input/output field within a `Tuple` object, the following formula is used: $field = (tupleIndex * numberOfFields + fieldPos) * fieldSize$. The formula uses the following inputs: (i) *tupleIndex* indicates which `Tuple` in the dataset is being accessed; (ii) the *numberOfFields* value, which specifies how many fields exist in the indexed `Tuple` object; (iii) *fieldPos*, which corresponds to the position of the field that we are trying to access inside the `Tuple`, counting from zero (e.g., if we are accessing the first field this value is 0, for the second field is 1 and so on); and (iv) *fieldSize*, that stores the size of the field in bytes. For example, the *fieldSize* value for a field of `int` type is 4, while for a field of `double` type it would be 8. For simplicity, we revisit the example of the `Tuple2` object that has two primitive fields (one field of type `double`, and a second field of type `int`) once padding has been applied (Figure 8). In this case, each field is represented by 8 bytes since the `int` field is padded to satisfy memory alignment (Section 3.3.2). By using the formula above, the first field within a `Tuple` object indexed by `i`, is accessed by reading from offset `(i * 2) * 8`; and, similarly, the second field is accessed by reading from offset `(i * 2 + 1) * 8`. The calculation of the offset for write-operations follows the same formula.
(3) **Array Handling:** In some cases, the Flink `Tuple` object can contain arrays of primitive values. To accommodate these cases, a compilation phase is added which copies the contents of the arrays directly from within the serialized byte array. In turn, the original fields accessing the contents of these arrays are being replaced with the corresponding array accesses by identifying the memory area in which the input/output fields should be read/written. Excluding the handling of the arrays referenced by Flink objects, an extra compiler phase is added to perform the offset calculation for accessing the memory addresses of the arrays, similarly to phase (2).
(4) **Replacement of Java Collections:** A second group of objects that is widely used in Flink, and other Java-based frameworks, is derived from the `Java Collections` library. Dynamic data structures (e.g., `LinkedList`) 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, the `size()` method that is common in Java Collections is replaced in the compiler graph with a `ConstantNode` that represents the size of the corresponding array. Similar actions are performed for node removal (e.g., de-optimization nodes) that cannot be translated to OpenCL/CUDA and executed on a GPU.
(5) **Matrix Flattening and Offset Calculation:** Apart from `Tuples`, we also provide support for matrices. To be consistent with the way that `Tuples` are handled by the compiler, we perform matrix *flattening*; a technique that transforms multi-dimensional arrays to equivalent ones of a single dimension (linear). Thus, the bytes that correspond to the rows and columns of the array are stored consecutively. 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. The 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 of a matrix `m` indexed by `i` and `j`, the following formula is used: $matrixElement = rowByteSize * i + elementSize * j$. `RowByteSize` corresponds to the size of bytes that each matrix row has (for a example, if each row consists of 8 float values, then the value of the `rowByteSize` is 32 bytes). The `elementSize` stores the size the matrix element in bytes (i.e., 4 bytes for representing `int` and `float` values, whereas 8 bytes for representing `long` and `double` values).

3.4.1 Reverse Marshaling & 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. Thus, the output data can be directly patched to the serialized byte array to be distributed over the network by Flink.

3.5 Handling Hybrid Execution

The compiler extensions explained in the previous section essentially bridge the gap between the API of Flink (or other Java-based Big Data frameworks) and the computational capabilities of heterogeneous hardware accelerators. However, the ability to transparently JIT compile UDFs 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. Such an example is the `groupBy` operator (used in the KMeans pipeline - Figure 9) which is currently partially supported by TornadoVM.

A closer inspection of the `JobGraph` generated by Flink, reveals that the `groupBy` and the consequent reduce operator are being chained together in one function called `SortAndCombine` via the `ChainedReduce-CombineDriver`. When trying to compile the `SortAndCombine` function with TornadoVM, a failure occurs during the QuickSort function that is internally used. In contrast, the reduce operation can be compiled and executed on the accelerator.

Hence, to run KMeans in the proposed platform two options exist: a) manually code a GPU/FPGA compatible `QuickSort` kernel and plug it into 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 since all execution would take place on a hardware accelerator, it violates the code portability and transparency challenges (Section 2.3). Thus, we opted for the solution which enables transparent hybrid execution of a data pipeline between a CPU and hardware accelerators. Consequently, all the operators of KMeans, besides the sorting function, are executed on the accelerator. To perform sorting, data is copied back to the CPU, unmarshaled, and then marshaled again for continuing execution of the reduce and consequent operators on the accelerator.

```
DataSet <> compute = ds
 .map() //Assigns each point to closest centroid
 .map() //Appends a counter to the results of map
 .groupBy()   //Groups data based on the centroid ID
 .reduce()   //Computes num points per centroid
 .map();  //Calculates new centroid coordinations
```

Figure 9: The KMeans Pipeline.

4 EXPERIMENTAL EVALUATION

We evaluate the proposed system against the baseline CPU-only Flink using seven benchmarks (Table 1). The baseline implementation is *scaled-out* to operate on multiple nodes and multiple threads per node for the first five applications (Matrix Multiplication, Logistic Regression, Discrete Fourier Transformation, KMeans, IoT Analytics), while for the last two (Pi Estimation and Vector Addition) we compare against single-threaded CPU execution. The reason that a different configuration is used for Pi Estimation and Vector Addition is that these are mainly used to evaluate our system against the current state-of-the-art support for GPU execution that Flink provides [18] (hereafter referred as *Flink-GPU*). Our experiments are executed following the same methodology (Section 4.1) on two testbeds which exhibit different software and hardware characteristics, as presented in Tables 2a and 2b, respectively. 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 Nvidia GP100 GPU and an Intel FPGA.

4.1 Experimental Methodology

For every experiment we apply the following methodology: 1) As a first step we launch the Flink cluster which, in essence, initiates the Flink *Job Manager* and the *Task Manager* nodes, 2) During the second step, the benchmarks are executed and performance is measured. To ensure fair comparison between the proposed and baselined systems, all reported results are the averages of ten consecutive end-to-end executions. The execution times are obtained by the Flink API. Thus, 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 our system, we execute each benchmark using various input sizes, as presented in the last column of Table 1.

To distinguish the configurations of all systems used in our experiments, we apply the following naming convention: (N)-Number of physical nodes - (TS-CPU/GPU)-Number of Task Slots per node. For example, the *N-2 TS-CPU-4* configuration corresponds to a deployment on two physical nodes, each running four CPU threads. Similarly, *N-2 TS-GPU-1* signifies that two physical nodes, each running with one CPU thread are utilized and the execution is deployed to a GPU. Following that semantics, the six different configurations used for the baseline execution are: *N-1 TS-CPU-1*, *N-1 TS-CPU-2*, *N-1 TS-CPU-4*, *N-2 TS-CPU-1*, *N-2 TS-CPU-2*, *N-2 TS-CPU-4.*

4.2 Performance Evaluation on GPUs

Section 4.2.1 presents the performance comparison of two benchmarks that are executed by the proposed system on a GPU (without

Table 1: Benchmarks along with their Configuration regarding the Utilized Flink Operators, Hybrid Execution and Data Sizes.

Name	Description	Operators Used	Accelerated Operators	Hybrid Execution Enabled	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, reduceGroup	reduce	yes	346 MB (Data from IoT Sensors)
Pi Estimation	Provides an estimation of the value of pi	map, reduce	map, reduce	no	1048576-16777216 samples
Vector Addition	Adds the elements of two vectors	map	map	no	1048576-16777216 elements per vector

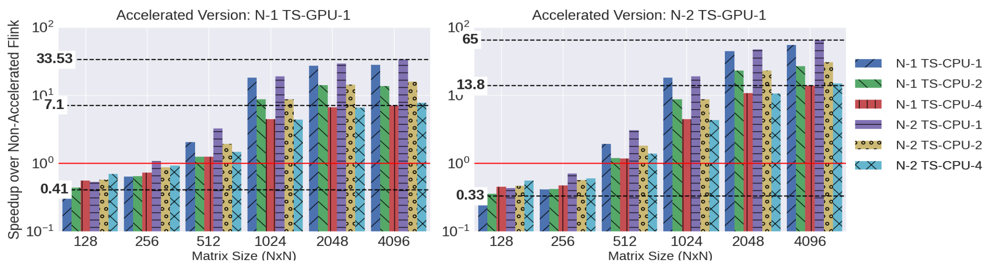


Figure 10: MxM: Performance of the Proposed System against the Baseline Flink Configurations. The higher, the better.

hybrid mode) versus CPU-only Flink (referred in plots also as non-accelerated Flink). The first benchmark is matrix multiplication between two two-dimensional matrices. The second benchmark is an industrial use case that predicts the likelihood of patients to be readmitted in public hospitals by using logistic regression on historical patient data. Section 4.2.2 compares the proposed system against Flink-GPU, using Pi Estimation and Vector Addition.

4.2.1 Comparison against CPU-only Flink.

Matrix Multiplication. The MxM benchmark uses a map operator to perform the multiplication between two two-dimensional matrices on Testbed-1. Figure 10 presents the relative performance of MxM executed by the proposed system against six configurations of the baseline system that scale out the number of physical nodes and the number of threads per node. The difference between Figure 10-left and Figure 10-right is related to the configuration of the proposed system. In the left figure the proposed system operates with one *Task Manager* and one *Task Slot* on a GPU (N-1 TS-GPU-1), while in the right figure it uses two *Task Managers* and one *Task Slot* on a GPU (N-2 TS-GPU-1). The horizontal axis groups the experiments for different matrix sizes ranging from 128 to 4096 elements per dimension of each matrix. The vertical axis shows the relative performance speedup of the proposed system against the baseline implementations (six bars) in logarithmic scale. The red horizontal line illustrates the reference point that shows equal performance between the compared systems. As shown in Figure 10, 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. The reason is that the input data size is small and 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. As the data size increases, the proposed system outperforms the CPU-only Flink by up to 33.53*x* (Figure 10-left) and 65*x* (Figure 10-right). The results indicate that in order to benefit from GPU acceleration, a particular data size threshold (e.g., 512 elements per dimension) must be met to offset the overheads of data transfers to the GPU. In addition, as the number of CPUs is scaled in the baseline configuration, the relative speedup achieved by a single GPU decreases.

(a) Software Setup.

Common Software Characteristics	
Flink	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
Nvidia Driver	418.152.00
Testbed-2	
OS	CentOS 7.4.1708
OpenCL Version	OpenCL 1.0
Nvidia Driver	384.111
FPGA Driver	Intel FPGA SDK 17.1

(b) Hardware Setup.

Testbed-1	
Node 1, Node 2	
CPU	Intel i7-4820K @ 3.70GHz
Main Memory	64 GB
GPU	Nvidia GeForce GTX 1060
GPU RAM	4 GB
Testbed-2	
CPU	Intel i7-7700K @ 4.20 GHz
Main Memory	64 GB
GPU	Nvidia Quadro GP100
GPU RAM	16 GB
FPGA	Nallatech 385a
FPGA RAM	4 GB

Table 2: Software and Hardware Characteristics of Testbeds.

Logistic Regression (LR). The LR 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 11 presents the overall end-to-end time for both systems, as reported by Flink on Testbed-1. The configurations of Flink are grouped in the horizontal axis based on the number of *Task Managers* and the device that they target. For instance, the

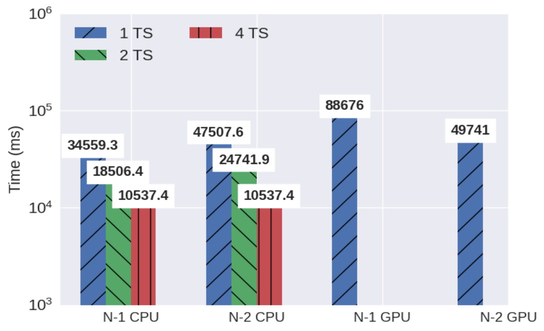


Figure 11: Execution Time of Logistic Regression for the Baseline Flink Configurations and the Proposed System that Executes on a GPU. The lower, the better.

baseline implementations run on the CPU with one (N-1 CPU) or two physical nodes (N-2 CPU), while the proposed system executes the computation on a GPU using one (N-1 GPU) or two physical nodes (N-2 GPU). The Flink configurations that we tested for the baseline implementations use 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). As shown in Figure 11, the proposed system performs slower than all baseline configurations. The N-1 TS-CPU-1 configuration is up to 2.5*x* faster than the equivalent configuration that is executed on the GPU (N-1 TS-GPU-1). Additionally, the N-2 TS-CPU-1 configuration that runs on two physical nodes outperforms the equivalent configuration that is executed on the GPU (N-2 TS-GPU-1) by up to 4%.

To understand why the GPU implementations perform significantly lower than the baseline implementations, we performed a complementary study. This study includes the breakdown analysis of the end-to-end job time for the proposed system that is configured to run on a single physical node (N-1 TS-GPU-1). Figure 12 presents the breakdown of the overall job time for two GPUs (GTX 1060 and Tesla V100). The former GPU is a commodity GPU integrated in Testbed-1, while the latter is a high-end GPU that offers more compute capabilities. 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 in the left bar of Figure 12, the marshaling cost along with the execution time of the `map` and `reduce` operators dominate the overall time of the Flink job. The marshaling time takes up to 30.7 seconds, while the execution time of the two operators on the GPU takes up to 33.3 and 24.7 seconds for `map` and `reduce`, respectively. The breakdown shown in the right bar of Figure 12 indicates that the job time is dominated by the marshaling cost which takes more than 80% of the overall time (almost 47 seconds). The comparison between the two systems shows that despite the large difference in performance between the two GPUs, the marshaling cost poses a significant overhead that can severely impact performance. The marshaling cost is attributed to the transformation of the serialized buffer in order to preserve endianness and aligned memory accesses. This issue is present to any acceleration solution

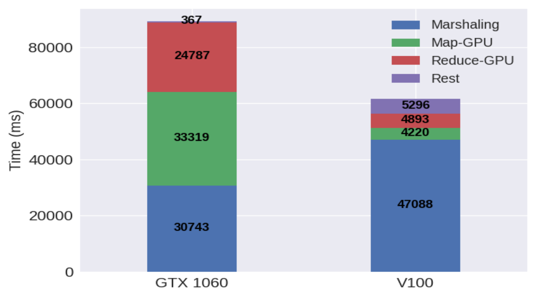


Figure 12: Execution Breakdown of Logistic Regression on two GPUs, a GTX 1060 (left bar) and a Tesla V100 (right bar).

Table 3: Evaluation of Logistic Regression for Various 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%

(pre-compiled CUDA/OpenCL kernels, or dynamically compiled via TornadoVM) that attempts to use direct serialized buffers that derive from a JVM. 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 for native buffers that could also be directly used by accelerators, the marshaling process would be circumvented.

Nonetheless, the performance on the GTX 1060 GPU would still be lower compared to the baseline even if the marshaling cost was completely eliminated. The total execution time on the GTX 1060, removing the marshaling cost is approximately 57933 milliseconds, which is still larger than the equivalent baseline configuration execution time, which is 34599 milliseconds. A hypothesis on why in this particular case the computation is slow, even though the application is highly-parallelizable, is that the GPU is overutilized [43]. To investigate this hypothesis, we conduct an additional experiment. In this experiment, the use case is executed on the GTX 1060 GPU of Testbed-1 for data sizes ranging from 1GB to 56MB, and the performance of the GPU computation deployed by TornadoVM is measured via the TornadoVM profiler. For each data size, the TornadoVM profiler reports the overall execution time of a TornadoVM TaskSchedule (TaskSchedule Time) in a granularity that 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 memory (Copy-Out Time). Table 3 presents the breakdown analysis of the TaskSchedule execution time for all data sizes. All times are reported in milliseconds. As shown in Table 3, the execution time of the kernels for all data sizes is significantly lower than the overall TaskSchedule time. In fact, the kernels' execution times range from 2726 milliseconds (1 GB) to 143 milliseconds (56 MB). Transferring data from the main memory to the GPU memory takes up to 77.4% and 56.2% of the

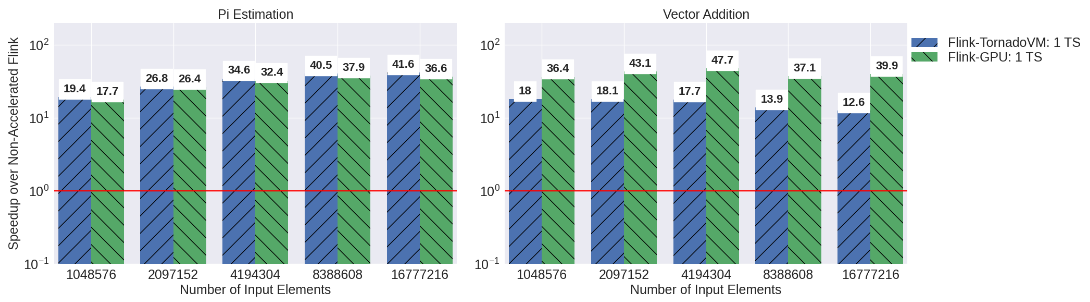


Figure 13: Performance of the Proposed System and Flink-GPU against Baseline Flink for Pi Estimation (left) and Vector Addition (right). The higher, the better.

overall TaskSchedule time for large (1 GB) and small (56 MB) sizes, respectively. The high cost of data transfers is attributed to the fact that the current implementation utilizes a separate TornadoVM TaskSchedule for every operator. This results in transferring data (back and forth) every time an operator is executed. In our future work, we plan to analyze this overhead and optimize it by grouping operators within the same TaskSchedule.

4.2.2 Comparison against the state-of-the-art Flink-GPU. Currently, Flink allows developers to target GPUs using JCuda and/or JCublas. Specifically, JCuda can be used in Flink applications to invoke prebuilt CUDA kernels on Nvidia GPUs. The developers have to explicitly perform memory management, write the kernel and transform data from object types to primitives. Figure 13 presents the speedup that Flink-GPU and the proposed system obtain over single-threaded Flink on Testbed-2 for two benchmarks, a Pi Estimation algorithm (Figure 13-left) and the Vector Addition computation (Figure 13-right). To ensure a fair comparison, both implementations exclude compilation times and execute prebuilt kernels. As illustrated in Figure 13, the proposed implementation consistently outperforms Flink-GPU for the Pi Estimation benchmark, with the highest performance obtained for the largest dataset (16777216 samples), while Flink-GPU achieves the best performance for 8388608 samples. In the Vector Addition benchmark, it is observed that even though the proposed framework outperforms CPU-only Flink, the obtained speedup decreases as the amount of data increases. The same trend is also observed for Flink-GPU. The performance of Flink-GPU over CPU-only Flink drops after the size of 4194304 input elements. This is attributed to the cost of marshaling. Nevertheless, if data was serialized in a format compatible for acceleration and no data morphing was involved, the presented framework could yield even higher performance gains. Moreover, a pivotal advantage of the proposed implementation, contrary to Flink-GPU, is that it does not require any code rewriting or user intervention.

4.3 Performance Evaluation on FPGAs

As `reduce` operators are not currently supported by TornadoVM for FPGA execution, we evaluate the performance of the DFT and MxM benchmarks implemented with a `map` operator in Flink.

Discrete Fourier Transformation (DFT). As stated in previous studies [56], the JIT compilation time for FPGAs can take significantly longer time than the actual execution of a particular compute kernel. The compilation time for the kernel that is generated from the implemented DFT algorithm takes more than one hour. Thus, the rest of this section employs the ahead-of-time execution mode of TornadoVM which focuses exclusively on the kernel execution on FPGAs. Figure 15 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 to 65536 elements for the input arrays. As shown in Figure 15, 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 1.56x and 1.57x 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 184x (N-2 TS-CPU-1 for 65536 elements). Additionally, the performance trend shown in Figure 15 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 performed by N-2 TS-CPU-4 (light blue bar), which outperforms up to 3.07x and 3.1x the N-1 TS-CPU-1 (blue bar) and N-2 TS-CPU-1 (purple bar) configurations. Like other works [56], 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. Applications, such as DFT, that can utilize specific hardware units integrated on the FPGA hardware for digital signal processing are good FPGA candidates.

Matrix Multiplication (MxM). Figure 16 presents the performance of the Matrix Multiplication application compared to CPU-only Flink. Even though, this application showcased performance improvement of up to 65x over Flink (Section 4.2) when executed on the GPU, it demonstrates consistent slowdowns when executed on the FPGA. This reaffirms the performance portability problems of OpenCL [5, 35]. Due to hardware limitations, this application could not be evaluated for matrices exceeding the 512x512 size.

Such benchmarks demonstrate that the capability of the proposed system to transparently target both GPUs and FPGAs, from the same unmodified Flink application, is one of its greatest strengths.

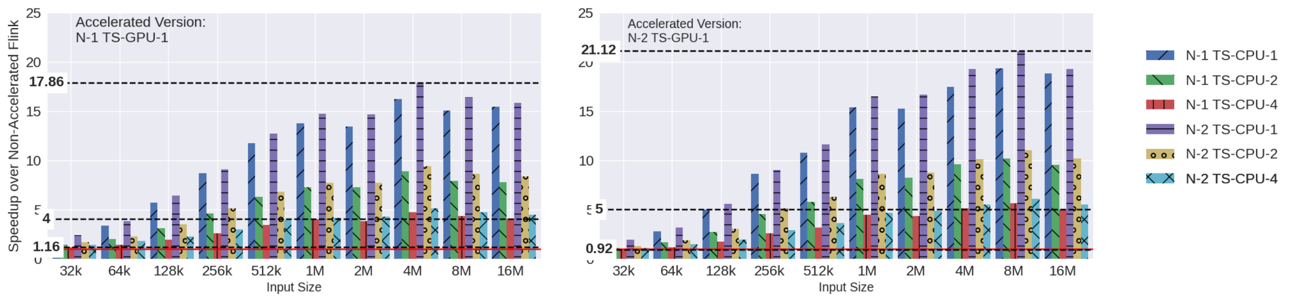


Figure 14: KMeans: Performance of the Proposed System against the Baseline Flink Configurations. The higher, the better.

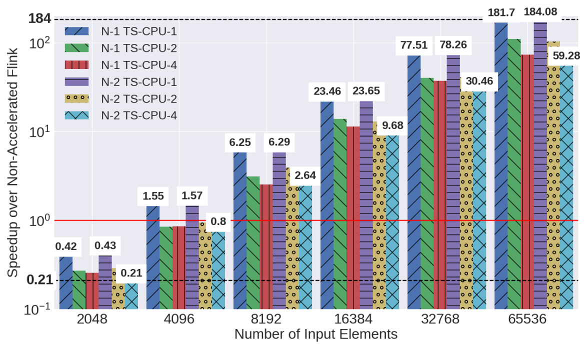


Figure 15: DFT on FPGAs: Performance of the Proposed System against Baseline Flink. The higher, the better.

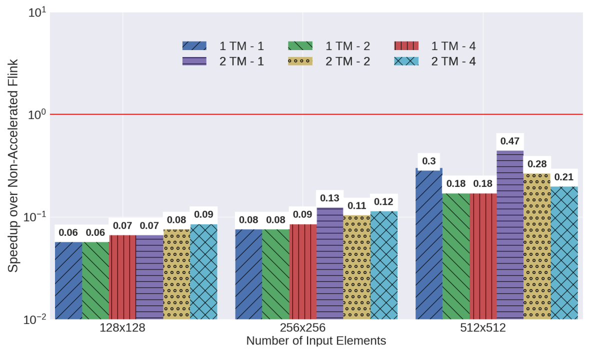


Figure 16: MxM on FPGAs: Performance of the Proposed System against Baseline Flink. The higher, the better.

4.4 Evaluation of Hybrid Execution

This section presents the performance comparison of KMeans and IoT Analytics that use the hybrid execution of Flink operators on both the CPU and GPU, against the baseline Flink implementations.

KMeans. Figure 14 shows the performance of KMeans on the proposed system against six configurations of the baseline Flink Non-Accelerated implementations. As shown in Figure 14, 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 ~ 1.3MB) to large (i.e., 16777216 elements ~ 649MB). The Flink accelerated version on the GPU performs up to 17.86*x* (Figure 14-left, 4194304 elements, purple bar) and 19.29*x* (Figure 14-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

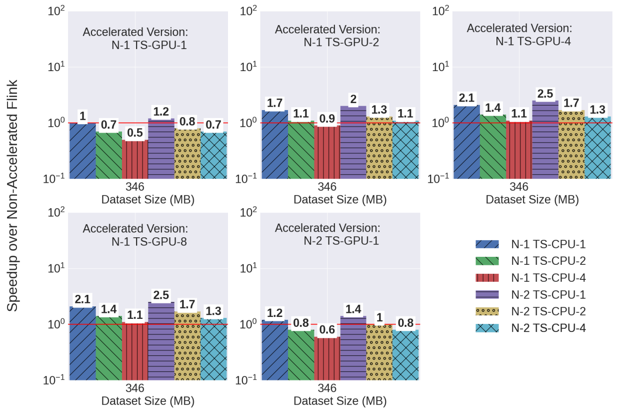


Figure 17: IoT Analytics: Performance of the Proposed System against Baseline Flink. The higher, the better.

maximum input size by 4*x* (Figure 14-left, red bar) and 5*x* (Figure 14-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`. The proposed approach can accelerate `map` and `reduce` operators, however the `groupBy` operator was executed on the CPU (Section 3.5).

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

Figure 17 contains five plots; each plot presents the comparative evaluation of 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 solution is tested in three additional configurations compared to the previous experiments. These additional configurations operate on one Task Manager and run on a GPU with two (N-1 TS-GPU-2), four (N-1 TS-GPU-4) and eight (N-1 TS-GPU-8) parallel threads (i.e., *Task Slots*). The reason that we added these extra configurations is that the initial configurations that run using one thread on a single node (N-1 TS-GPU-1) and two nodes (N-2 TS-GPU-1) do not yield high performance improvement. In fact, N-1 TS-GPU-1 and N-2 TS-GPU-1 present up to 1.23*x* and 1.43*x* performance speedups, respectively. Thus, the additional configurations can indicate whether the GPU under-utilization is the reason of the low performance improvement.

Figure 17 shows that performance increases, but it does not scale when running with more than four *Task Slots*. In particular, it is shown that the maximum performance improvement is 2.54x and it is achieved by N-1 TS-GPU-4 against the N-1 TS-CPU-1 baseline (purple bar). When the number of parallel threads on the same node is increased to eight (N-1 TS-GPU-8), the overall performance is increased up to 2.48x against N-1 TS-CPU-1 which indicates saturation. Upon further inspection of the GPU utilization using *nvidia-smi*, it has been observed that the GPU is overutilized (up to 100% utilization) even when running with two *Task Slots*.

5 RELATED WORK

Table 4 groups the related work in two categories: i) pre-compiled kernels; and ii) on-demand kernel compilation, based on the characteristics and limitations discussed in Section 2.3.

Pre-compiled Kernel Implementations Plenty of research has been conducted on enabling the execution of pre-compiled kernels on heterogeneous hardware devices for Spark [20, 21, 28–30, 38, 39, 42, 50, 60, 65, 69, 71], Hadoop [2, 16, 27, 41, 44–47, 58, 68, 72], Flink [8, 9], Storm [10, 70], and Ignite [63]. Several works expose APIs to Java [2, 8, 10, 38, 39, 50, 63, 65, 68–72] or Python [28, 29, 42] for developers to interface with pre-compiled kernels. Most works use JCuda or PyCUDA [36] for launching heterogeneous pre-compiled kernels on GPUs. MITHRA [16] is a framework that enables users to write their functions using Hadoop's API and launch pre-compiled CUDA kernels on GPUs. Other studies have investigated the integration of FPGAs in Big Data platforms by utilizing either custom hardware designs in the Register-Transfer-Level (RTL) [20, 21, 41] or via High-Level-Synthesis (HLS) tools that generate RTL designs from C/C++ and OpenCL [44–47]. Furthermore, work has focused on the hybrid utilization of accelerators (i.e., CPUs and GPUs) for Flink [9] and Hadoop [27].

On-demand Compiled Kernels Several research studies [7, 22–24, 40, 62] have utilized Aparapi as a way to compile Java bytecodes to OpenCL kernels for CPUs, GPUs or FPGAs. Additionally, Grossman and Sarkar [25] have presented SWAT, a system that compiles JVM bytecodes to OpenCL kernels that can execute on hardware accelerators. SWAT provides support for multi-GPU memory handling by implementing an internal library that performs caching of data on OpenCL devices. HeteroDoop [61] exposes specific directives that enable source-to-source translation by using the Cetus compiler. Vispark [11] is a Python-like language that translates the source code to execute on GPUs. Finally, Kotselidis et al. [37] presents a system architecture that employs TornadoVM as a means to offer transparent compilation and acceleration of Flink applications on heterogeneous devices. However, that work does not focus on the challenges regarding the transparent integration of hardware acceleration in Flink and it presents a preliminary performance evaluation for k-means on a single node with a GPU.

The work presented in this paper goes beyond the state-of-the-art as, to the best of our knowledge, it is the first work that proposes a co-designed approach which enables on-demand compilation of arbitrary unmodified user-defined functions on a variety of hardware devices (e.g., CPUs, GPUs and FPGAs).

Table 4: Work on HW Acceleration of Big Data Frameworks.

Code Gen.	References	Big Data Framework	Code Fragm.	Vendor lock-in	Device Coverage
Pre-compiled	Hong et al. [28], HeteroSpark [38], ShadowVM [39], Manzi&Tompkins [42], Ohno et al. [50], DataBricks [60]	Spark	Yes	Yes	GPUs (Nvidia)
	Spark-GPU [71]	Spark	Yes	No	GPUs
	Ghasemi&Chow [20, 21], Hou et al. [29], Huang et al. [30]	Spark	Yes	Yes	FPGAs (Xilinx)
	Stamelos et al. [65]	Spark	Yes	No	FPGAs
	SparkJNI [69]	Spark	Yes	No	CPUs, GPUs, FPGAs
	Hadoop+ [27]	Hadoop	Yes	No	GPUs
	Rathore et al. [58], GPU-in-Hadoop [72]	Hadoop	Yes	Yes	GPUs (Nvidia)
	Abbasi et al. [2]	Hadoop	Yes	N/D	GPUs
	MITHRA [16]	Hadoop	Yes	Yes	GPUs (Nvidia)
	Tan et al. [68]	Hadoop	Yes	Yes	CPUs, GPUs (Nvidia)
	ZCluster [41], Neshatpour et al. [44–47]	Hadoop	Yes	Yes	FPGAs (Xilinx)
	GFlink [8]	Flink	Yes	Yes	GPUs (Nvidia)
	Chen et al. [9]	Flink	Yes	Yes	CPUs, GPUs (Nvidia)
	G-Storm [10]	Storm	Yes	Yes	GPUs (Nvidia)
	Wu et al. [70]	Storm	Yes	Ys	FPGAs (Intel)
	Ignite-GPU [63]	Ignite	Yes	Yes	GPUs (Nvidia)
On-demand	SWAT [25], SparkCL [62]	Spark	Yes	No	CPUs, GPUs, FPGAs
	HJ-OpenCL [24]	Spark	Yes	No	GPUs
	Vispark [11]	Spark	Yes	Yes	GPUs (Nvidia)
	HadoopCL [22], HadoopCL2 [23], Hadoop+Aparapi [40]	Hadoop	Yes	No	GPUs
	HeteroDoop [61]	Hadoop	Yes	Yes	GPUs (Nvidia)
	FlinkCL [7]	Flink	Yes	No	CPUs, GPUs
	Kotselidis et al. [37]	Flink	No	No	GPUs
	This work	**Flink**	**No**	**No**	**CPUs, GPUs, FPGAs**

6 CONCLUSIONS AND FUTURE WORK

In this paper, we discuss the main challenges for integrating hardware acceleration within current Big Data frameworks. To tackle those challenges, we propose a co-designed approach that enables the execution of unmodified Big Data applications on multiple hardware accelerators, transparently to the users. The proposed approach is prototyped in the context of Flink and TornadoVM, and introduces two novel techniques regarding: a) automatic code and data morphing, and b) JIT compilation for heterogeneous hardware. The experimental evaluation across a variety of benchmarks and industrial use cases, demonstrates speedups of up to 184x when running on FPGAs and 65x when running on GPUs, compared to CPU-Flink. Furthermore, the proposed implementation is evaluated against the current state-of-the-art support for GPU execution that Flink provides, showcasing competitive performance against highly-optimized CUDA kernels. As future work, we aim to increase the coverage of the Flink operators that can be accelerated.

ACKNOWLEDGMENTS

This work was partially funded by grants from Intel Corporation and the European Union Horizon 2020 E2Data 780245, ELEGANT 957286, and Horizon Europe/Innovate UK ENCRYPT 101070670 and TANGO 101070052 projects.

REFERENCES

[1] Accessed in October 2022. RAPIDS. https://rapids.ai/

[2] Amin Abbasi, Farshad Khunjush, and Reza Azimi. 2012. A preliminary study of incorporating GPUs in the Hadoop framework. *The 16th CSI International Symposium on Computer Architecture and Digital Systems (CADS)* (2012), 178–185.
[3] AMD. Accessed in October 2022. Aparapi project. https://aparapi.github.io/
[4] Apache. Accessed in October 2022. Apache Yarn GPU. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/UsingGpus.html
[5] Ignacio Bravo, Pedro Jimenez, Manuel Mazo, Jose Luis Lazaro, Jose J. de las Heras, and Alfredo Gardel. 2007. Different Proposals to Matrix Multiplication Based on FPGAS. In *2007 IEEE International Symposium on Industrial Electronics*. 1709–1714. https://doi.org/10.1109/ISIE.2007.4374862
[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.
[7] Cen Chen, Kenli Li, Aijia Ouyang, and Keqin Li. 2018. FlinkCL: An OpenCL-Based In-Memory Computing Architecture on Heterogeneous CPU-GPU Clusters for Big Data. *IEEE Trans. Comput.* 67 (2018), 1765–1779.
[8] Cen Chen, Kenli Li, Aijia Ouyang, Zeng Zeng, and Keqin Li. 2016. GFlink: An In-Memory Computing Architecture on Heterogeneous CPU-GPU Clusters for Big Data. *IEEE Transactions on Parallel and Distributed Systems* 29 (2016), 1275–1288.
[9] Cen Chen, Aijia Ouyang, Zhuo Tang, and Keqin Li. 2017. GPU-Accelerated Parallel Hierarchical Extreme Learning Machine on Flink for Big Data. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 47 (2017), 2740–2753.
[10] Zhenhua Chen, Jielong Xu, Jian Tang, Kevin A. Kwiat, and Charles A. Kamhoua. 2015. G-Storm: GPU-enabled high-throughput online data processing in Storm. *2015 IEEE International Conference on Big Data (Big Data)* (2015), 307–312.
[11] Woohyuk Choi and Won-Ki Jeong. 2015. Vispark: GPU-accelerated distributed visual computing using spark. In *LDAV*.
[12] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. 2018. Exploiting High-Performance Heterogeneous Hardware for Java Programs Using Graal. In *Proceedings of the 15th International Conference on Managed Languages and Runtimes (ManLang '18)*. Association for Computing Machinery. https://doi.org/10.1145/3237009.3237016
[13] Katherine Compton and Scott Hauck. 2002. Reconfigurable Computing: A Survey of Systems and Software. *ACM Comput. Surv.* (2002). https://doi.org/10.1145/508352.508353
[14] NVIDIA Corporation. Accessed in October 2022. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/
[15] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. 2011. SIMD re-convergence at thread frontiers. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO* (12 2011). https://doi.org/10.1145/2155620.2155676
[16] Reza Farivar, Abhishek Verma, Ellick Chan, and Roy H. Campbell. 2009. MITHRA: Multiple data independent tasks on a heterogeneous resource architecture. *IEEE International Conference on Cluster Computing and Workshops* (2009), 1–10.
[17] Apache Flink. Accessed in October 2022. Flink Checkpoints. https://nightlies.apache.org/flink/flink-docs-release-1.3/setup/checkpoints.html
[18] The Apache Software Foundation. Accessed in October 2022. Accelerating your workload with GPU and other external resources. https://flink.apache.org/news/2020/08/06/external-resource.html
[19] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic Application Reconfiguration on Heterogeneous Hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2019)*. Association for Computing Machinery. https://doi.org/10.1145/3313808.3313819
[20] Ehsan Ghasemi and Paul Chow. 2016. Accelerating Apache Spark Big Data Analysis with FPGAs. *International IEEE Conferences on Ubiquitous Intelligence and 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)* (2016), 737–744.
[21] Ehsan Ghasemi and Paul Chow. 2019. Accelerating Apache Spark with FPGAs. *Concurrency and Computation: Practice and Experience* 31 (2019).
[22] Max Grossman, Maurício Breternitz, and Vivek Sarkar. 2013. HadoopCL: MapReduce on Distributed Heterogeneous Platforms through Seamless Integration of Hadoop and OpenCL. *IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum* (2013), 1918–1927.
[23] Max Grossman, Maurício Breternitz, and Vivek Sarkar. 2016. HadoopCL2: Motivating the Design of a Distributed, Heterogeneous Programming System With Machine-Learning Applications. *IEEE Transactions on Parallel and Distributed Systems* 27 (2016), 762–775.
[24] Max Grossman, Shams Mahmood Imam, and Vivek Sarkar. 2015. HJ-OpenCL: Reducing the Gap Between the JVM and Accelerators. *Proceedings of the Principles and Practices of Programming on The Java Platform* (2015).
[25] Max Grossman and Vivek Sarkar. 2016. 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).
[26] Khronos OpneCL Working Group. Accessed in October 2022. The OpenCL C Specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_C.html
[27] Wenting He, Huimin Cui, Binbin Lu, Jiacheng Zhao, Shengmei Li, G. Ruan, Jingling Xue, Xiaobing Feng, Wensen Yang, and Youliang Yan. 2015. Hadoop+: Modeling and Evaluating the Heterogeneity for MapReduce Applications in Heterogeneous Clusters. *Proceedings of the 29th ACM on International Conference on Supercomputing* (2015).
[28] Sumin Hong, Woohyuk Choi, and Won-Ki Jeong. 2017. GPU in-Memory Processing Using Spark for Iterative Computation. *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2017), 31–41.
[29] Junjie Hou, Yongxin Zhu, L. Kong, Zhe Wang, Sen Du, Shijin Song, and Tian Huang. 2018. A Case Study of Accelerating Apache Spark with FPGA. *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)* (2018), 855–860.
[30] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. 2016. Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale. *Proceedings of the Seventh ACM Symposium on Cloud Computing* (2016).
[31] Marco Hutter. Accessed in October 2022. Java bindings for CUBLAS. http://javagl.de/jcuda.org/jcuda/jcublas/JCublas.html
[32] Marco Hutter. Accessed in October 2022. Java bindings for CUDA. http://www.jcuda.org/
[33] IBM. Accessed in October 2022. OpenJ9. https://github.com/eclipse-openj9/openj9
[34] INTEL. Accessed in October 2022. OneAPI. https://oneapi.io
[35] Xiaoxiao Jiang and Jun Tao. 2011. Implementation of effective matrix multiplication on FPGA. In *2011 4th IEEE International Conference on Broadband Network and Multimedia Technology*. 656–658. https://doi.org/10.1109/ICBNMT.2011.6156017
[36] Andreas Kloeckner. Accessed in October 2022. PyCUDA. https://documen.tician.de/pycuda
[37] Christos Kotselidis, Sotiris Diamantopoulos, Orestis Akrivopoulos, Viktor Rosenfeld, Katerina Doka, Hazeef Mohammed, Georgios Mylonas, Vassilis Spitadakis, and Will Morgan. 2020. Efficient Compilation and Execution of JVM-Based Data Processing Frameworks on Heterogeneous Co-Processors. In *Proceedings of the 23rd Conference on Design, Automation and Test in Europe (DATE '20)*. 175–179.
[38] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. 2015. HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms. *IEEE International Conference on Networking, Architecture and Storage (NAS)* (2015), 347–348.
[39] Zhifang Li, Mingcong Han, Shangwei Wu, and Chuliang Weng. 2021. 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).
[40] Yu Lin, Semih Okur, and Cosmin Radoi. 2012. Hadoop+Aparapi: Making heterogenous MapReduce programming easier.
[41] Zhongduo Lin and Paul Chow. 2013. ZCluster: A Zynq-based Hadoop cluster. *International Conference on Field-Programmable Technology (FPT)* (2013), 450–453.
[42] D. Manzi and David Tompkins. 2016. Exploring GPU Acceleration of Apache Spark. *2016 IEEE International Conference on Cloud Engineering (IC2E)* (2016), 222–223.
[43] Christos Margiolas and Michael F. P. O'Boyle. 2016. Portable and Transparent Software Managed Scheduling on Accelerators for Fair Resource Sharing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 82–93. https://doi.org/10.1145/2854038.2854040
[44] Katayoun Neshatpour, Maria Malik, Mohammad Ali Ghodrat, and Houman Homayoun. 2015. Accelerating Big Data Analytics Using FPGAs. *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines* (2015), 164–164.
[45] Katayoun Neshatpour, Maria Malik, Mohammad Ali Ghodrat, Avesta Sasan, and Houman Homayoun. 2015. Energy-efficient acceleration of big data analytics applications using FPGAs. *2015 IEEE International Conference on Big Data (Big Data)* (2015), 115–123.
[46] Katayoun Neshatpour, Maria Malik, and Houman Homayoun. 2015. Accelerating Machine Learning Kernel in Hadoop Using FPGAs. *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2015), 1151–1154.
[47] Katayoun Neshatpour, Avesta Sasan, and Houman Homayoun. 2016. Big data analytics on heterogeneous accelerator architectures. *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (2016), 1–3.
[48] NVIDIA. Accessed in October 2022. CUDA. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html
[49] ObjectWeb. Accessed in October 2022. ASM. https://asm.ow2.io
[50] Yasuhiro Ohno, Shin Morishima, and Hiroki Matsutani. 2016. Accelerating Spark RDD Operations with Local and Remote GPU Devices. *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)* (2016),

791–799.

[51] Oracle. Accessed in October 2022. The Java Tutorials. https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html

[52] Oracle. Accessed in October 2022. The Java Virtual Machine Specification. https://docs.oracle.com/javase/specs/jvms/se7/html/

[53] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. 2008. GPU computing. *Proceedings of the Institute of Radio Engineers* 96, 5 (May 2008), 879–899. https://doi.org/10.1109/JPROC.2008.917757

[54] Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. 2021. 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)*. Association for Computing Machinery. https://doi.org/10.1145/3453933.3454014

[55] Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, Foivos S. Zakkak, and Christos Kotselidis. 2020. Transparent Compiler and Runtime Specializations for Accelerating Managed Languages on FPGAs. *The Art, Science, and Engineering of Programming* 2 (Oct 2020). https://doi.org/10.22152/programming-journal.org/2021/5/8

[56] Michail Papadimitriou, Juan José Fumero, Athanasios Stratikopoulos, Foivos S. Zakkak, and Christos Kotselidis. 2021. Transparent Compiler and Runtime Specializations for Accelerating Managed Languages on FPGAs. *Art Sci. Eng. Program.* 5 (2021), 8.

[57] Michail Papadimitriou, Eleni Markou, Juan Fumero, Athanasios Stratikopoulos, Florin Blanaru, and Christos Kotselidis. 2021. 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)*. Association for Computing Machinery. https://doi.org/10.1145/3453933.3454019

[58] M. Mazhar Rathore, Hojae Son, Awais Ahmad, Anand Paul, and Gwanggil Jeon. 2017. Real-Time Big Data Stream Processing Using GPU with Spark Over Hadoop Ecosystem. *International Journal of Parallel Programming* 46 (2017), 630–646.

[59] David Reinsel, John Gantz, and John Rydning. 2018. *The Digitization of the World: From Edge to Core* . Technical Report. IDC.

[60] Jason Lowe Robert Evans. 2020. Deep Dive into GPU Support in Apache Spark 3.x. https://databricks.com/session_na20/deep-dive-into-gpu-support-in-apache-spark-3-x SPARK+AI SUMMIT 2020.

[61] Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann. 2015. HeteroDoop: A MapReduce Programming System for Accelerator Clusters. *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (2015).

[62] Oren Segal, Philip Colangelo, Nasibeh Nasiri, Zhuo Qian, and Martin Margala. 2015. SparkCL: A Unified Programming Framework for Accelerators on Heterogeneous Clusters. *ArXiv* abs/1505.01120 (2015).

[63] Amir Hossein Sojoodi, Majid Salimi Beni, and Farshad Khunjush. 2020. Ignite-GPU: A GPU-enabled in-memory computing architecture on clusters. *The Journal of Supercomputing* 77 (2020), 3165–3192.

[64] Apache Spark. Accessed in October 2022. Spoark Checkpoints. https://spark.apache.org/docs/latest/streaming-programming-guide.html

[65] Ioannis Stamelos, Elias Koromilas, Christoforos Kachris, and Dimitrios Soudris. 2018. A Novel Framework for the Seamless Integration of FPGA Accelerators with Big Data Analytics Frameworks in Heterogeneous Data Centers. *2018 International Conference on High Performance Computing and Simulation (HPCS)* (2018), 539–545.

[66] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering* (2010). https://doi.org/10.1109/MCSE.2010.69

[67] Athanasios Stratikopoulos, Mihai-Cristian Olteanu, Ian Vaughan, Zoran Sevarac, Nikos Foutris, Juan Fumero, and Christos Kotselidis. 2020. Transparent Acceleration of Java-Based Deep Learning Engines *(MPLR 2020)*. Association for Computing Machinery. https://doi.org/10.1145/3426182.3426188

[68] Yu Shyang Tan, Bu-Sung Lee, Bingsheng He, and Roy H. Campbell. 2012. A Map-Reduce Based Framework for Heterogeneous Processing Element Cluster Environments. *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)* (2012), 57–64.

[69] Tudor Alexandru Voicu and Zaid Al-Ars. 2019. SparkJNI: A Toolchain for Hardware Accelerated Big Data Apache Spark. *2019 IEEE 4th International Conference on Big Data Analytics (ICBDA)* (2019), 152–157.

[70] Song Wu, Die Hu, Shadi Ibrahim, Hai Jin, Jiang Xiao, Fei Chen, and Haikun Liu. 2019. When FPGA-Accelerator Meets Stream Data Processing in the Edge. *IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (2019), 1818–1829.

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

[72] Jie Zhu, Juanjuan Li, Erikson Hardesty, Hai Jiang, and Kuan-Ching Li. 2014. GPU-in-Hadoop: Enabling MapReduce across distributed heterogeneous platforms. *IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)* (2014), 321–326.