EasyChair Preprint

№ 600

Challenges and proposals for enabling dynamic heterogeneous execution of Big Data frameworks

Maria Xekalaki, Juan Fumero and Christos Kotselidis

October 31, 2018

Challenges and proposals for enabling dynamic heterogeneous execution of Big Data frameworks

Maria Xekalaki
School of Computer Science
The University of Manchester
Manchester, UK
maria.xekalaki@manchester.ac.uk

Juan Fumero
School of Computer Science
The University of Manchester
Manchester, UK
juan.fumero@manchester.ac.uk

Christos Kotselidis
School of Computer Science
The University of Manchester
Manchester, UK
christos.kotselidis@manchester.ac.uk

***Abstract*—The efficient execution of Big Data applications requires a large quantity of compute and memory resources. Typically, these resources are in the form of data centres with numerous processing elements connected through a computer network. Although initially the majority of data centers were utilizing only CPU resources, nowadays we can find heterogeneous accelerators such as GPUs and FPGAs. Ideally, Big Data frameworks and applications should exploit those diverse hardware resources in order to push their performance boundaries or increase resource utilization. Despite ongoing work to enable such functionality, the majority of the solutions revolve around external libraries that provide pre-compiled kernels for heterogeneous accelerators. This fact imposes programmability and code fragmentation challenges that can only be addressed by enabling Big Data platforms to dynamically compile and execute their code on such devices.**

In this paper we analyze and discuss the major challenges for programming and executing Big Data processing applications on distributed systems with heterogeneous hardware. In addition, we present our work-in-progress towards providing a heterogeneous programming framework for running Big Data applications on systems that include diverse hardware resources including CPUs, GPUs, and FPGAs. In contrast to existing approaches, our envisioned solution employs JIT compilation and runtime support, integrated in the data flow engine, enabling the automatic acceleration of Big Data platforms completely transparently to the user and without sacrificing programmability.

***Index Terms*—Big Data Frameworks; Apache Flink; GPGPUs**

I. Introduction

The advent of data intensive applications in the form of social media analytics, machine learning, and the Internet of Things (IoT), has resulted in a plethora of Big Data processing frameworks [1] including, but not limited to, MapReduce [2], Apache Flink [3], Spark [4] and Storm [5]. These systems enable programmers to express computations in high-level programming languages such as Java, Scala, Python and R which are executed on commodity clusters or cloud environments. The performance of the aforementioned systems is directly linked to the processing capacity of the underlying hardware configuration currently following a scale-out or a scale-up approach.

In order to tackle the ever-increasing need for extreme compute capabilities, cloud providers (e.g. Google [6], Amazon [7], etc.) have recently started investing in equipping their clusters with heterogeneous hardware, such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), to increase their computational power and energy efficiency.

Although such heterogeneous cloud deployments are public and ready-to-use, Big Data platforms can not currently exploit them in a transparent manner. Any solutions or proposals, mainly derived from academia, tackle the challenge of heterogeneous Big Data acceleration by introducing external libraries that contain specific pre-defined and pre-compiled kernels. By *offloading* specific parts of the computation, reflected by those kernels, Big Data platforms can exploit heterogeneous hardware in a non-transparent manner by combining multiple programming models. By following this approach, the programming models become fragmented since well-known data-flow engines execute code of managed programming languages (such as Java and Scala) while GPUs and FPGAs are only programmed with C-based languages like OpenCL and CUDA. Since the majority of Big Data frameworks are Java-based, and hence execute on top of a Java Virtual Machine (JVM) [8], adding JVM support for heterogeneous execution could potentially solve that problem. Although there is a limited number of projects that perform GPU compilation for Java programs (e.g. Aparapi [9], IBM-J9 [10] and Marawacc [11], [12]), both their proper integration with Big Data frameworks and execution capabilities pose challenges in enabling transparent and dynamic Big Data acceleration on heterogeneous hardware.

In this paper we present the current challenges and considerations when programming Big Data applications on heterogeneous distributed systems (e.g., clusters of computers with CPU, GPUs, and FPGAs connected through a network). In addition, we present our work-in-progress towards providing a common programming paradigm and framework for accelerating Big Data applications dynamically and transparently to the users. To showcase our solutions, we modified the existing Apache Flink, for batch data processing, to allow programmers to execute workloads on CPUs and GPUs transparently. Our proposal includes a runtime layer integrated into the Apache Flink runtime that allows automatic compilation and execution on the heterogeneous cluster. To allow GPU execution, we use Tornado [13], [14], a practical heterogeneous programming framework that executes Java programs on OpenCL-compatible hardware.

In detail, this paper contributes the following:

- It presents the main challenges for enabling Big Data applications and frameworks to exploit heterogeneous hardware resources dynamically and transparently to the users.
- It demonstrates our work-in-progress towards integrating a batch and stream data processing engine (Apache Flink) with a heterogeneous programming framework (Tornado), in order to enable heterogeneous execution of Big Data applications.

II. Challenges

Enabling the automatic and transparent acceleration of Big Data frameworks on heterogeneous hardware entails a number of challenges which are outlined in the following subsections:

a) Programmability: Developing Big Data applications on distributed memory systems (e.g., clusters of compute nodes and data centers) is a challenging task that combines multiple programming paradigms. Structured parallel programming, combined with Map/Reduce operators, is the basis of the most recent and extended parallel programming frameworks for Big Data analytics and processing. In addition, the successful Map/Reduce programming paradigm is often combined with ideas borrowed from functional programming, facilitating the development process of parallel and distributed applications. For example, Apache Flink exposes a set of parallel operations such as `map`, `reduce`, `filter`, and `group`. This programming style(s) is commonly expressed with high-level programming languages like Java, Python and R. The combination of the high-level languages, the use of structured parallel patterns, and the operator-style processing of Big Data frameworks enables programmers and researchers to prototype their ideas very rapidly since they are not expected to possess specific set of skills for parallel programming.

Unfortunately, when combining these ideas and programming styles with heterogeneous computing, applications are becoming increasingly complex. For example, GPUs are usually programmed in C-like, low-level programming models and languages, with the most common ones being CUDA and OpenCL. When programming from high-level Big Data frameworks such as Flink or Spark, users need to mix Java and Scala code with low-level CUDA and OpenCL. This creates not only programmability challenges, since programmers of various expertise are required, but also results in code fragmentation and code maintenance issues. Ideally, the runtime system supporting the Big Data framework should provide compilation and runtime support for arbitrarily compiling any code segment to any hardware device completely transparently to the user. By using the same parallel and structured patterns such as map and reduce, developers should benefit from the underlying hardware capabilities without having to manually code their application in a different programming language.

In addition, users need to compose map/reduce operations in a different way to fully utilize the GPUs. For example, Apache Flink processes items from input data collections (called *data sets*) at the granularity of a single element. When computing on heterogeneous and parallel hardware such as GPUs, computation needs to be performed in a coarse grain rather than a fine grain manner. This is due to the fact GPUs or other devices have different and distinct memory and address spaces. Hence, any data used during execution must be explicitly allocated and then transferred from the main host (e.g., a CPU) to the target compute-device (e.g., a GPU). Therefore, the Big Data framework should factor in the execution time (or the anticipated performance) the time required to perform a bulk copy of data to the heterogeneous device.

b) Data Partitioning: Batch and stream data processing are the two main computational models that the most common frameworks support. For example, although Apache Spark was originally designed for batch processing, it recently added support for micro-batch data processing where data is first accumulated into a buffer and then processed at once. Stream data processing, currently supported by Apache Flink, enables computations as soon as data becomes available. Depending on the input application and business' requirements, developers can select between the two modes of execution.

Although these two models work well for computations on homogeneous architectures (CPUs), they face challenges when applied in a heterogeneous context due to data partitioning. The main challenge in data partitioning for Big Data applications is to find the right balance in order to maximize the performance and utilization of the underlying heterogeneous hardware resources. The decision on which device a particular task should execute must be accompanied by the correct data granularity upon which the execution will be performed. For example, if the runtime system selects a GPU for execution, the data partitioning should dynamically be divided in such as way that maximizes resource utilization on GPUs. Otherwise, the performance will degrade compared to CPU-only execution, in which an even data distribution between cores suffices both for load-balancing and non-starvation of the hardware resources.

c) Fault Tolerance: Big Data frameworks utilize checkpointing [15] in order to provide fault-tolerant execution. Checkpoints are global asynchronous snapshots of the state of an application and they are used to recover the computation from a predefined point, in case an error occurs. For example, if a server crashes or goes offline, a main server can re-schedule the job amongst the rest of the operational servers from the last checkpoint. The frequency of checkpointing typically can be user-defined (for example via a time-stamp) and poses a trade-off between execution time and replay actions upon restoring a checkpoint. The more frequently we checkpoint our application, the less work we will have to repeat upon a failure in the expense of time spent to perform the checkpointing.

Although this trade-off is evident and can be calculated in advance for CPU-only execution, it poses significant challenges when executing on heterogeneous hardware. For example if we consider GPUs, both the parallel execution of code and the coarse granularity of data they operate on, can influence the efficiency of checkpoint operations due to the following reasons: 1) checkpointing in GPUs is a more expensive operation

compared to CPUs since data must be copied back to the host's memory, and 2) the infrequent checkpointing of GPU executed code, as a means to tackle the performance overhead, can result in significant replay operations due to the bulk execution granularity they operate on.

d) Maintaining the Data Value: This challenge correlates directly with the processing granularity that heterogeneous devices, such as GPUs, require. With respect to time-critical data where its value is highly dependent on its lifespan, it is imperative to process it as soon as possible. However, in GPU processing this may not be feasible due to the fact we have to perform bulk computations on coarse-grain data since they are designed for high-throughput, rather than low-latency.

Finding the right balance between the amount of data to be processed on distributed memory systems and keeping the data value by processing it in real-time is a significant challenge. Ideally, an optimizing Big Data processing runtime should be aware of these trade-offs and adapt itself to the best combination in order to satisfy the user requirements.

III. Background

In this paper we attempt to tackle the challenge of programmability by enabling dynamic compilation and execution of Big Data frameworks on heterogeneous devices. As a proof-of-concept, we prototype our solutions by combining the state-of-the-art Apache Flink execution engine with the Tornado framework.

We considered several aspects regarding platform capabilities and implementation details before selecting the Big Data framework for our proof-of-concept implementation. Currently, the most popular frameworks are Apache Spark, Flink, and Storm [16]. As discussed by Karimov et. al. [17], the selection of a platform highly depends on the nature of the running application. Apache Flink and Storm are predominately streaming engines that also support batch processing, unlike Spark, which is mostly used for batch processing with its streaming capabilities being in the form of micro-batching [18]. Our aim is to enable heterogeneous execution for both stream and batch data applications and hence we opted for Apache Flink to showcase our solutions. In addition, the code base of Apache Flink is closer to a pure Java implementation which makes the integration with Tornado easier compared to the rest (e.g. Scala-based in Apache Spark). Nevertheless, our proposed solutions are orthogonal to the Big Data framework and can be applied to any JVM-based frameworks and platforms.

This section provides an overview of the frameworks used, while Section IV describes our work-in-progress in enabling GPU execution on Apache Flink.

A. Apache Flink

Apache Flink is a Big Data framework for both batch and stream processing. Figure 1 depicts the main components of the Apache Flink architecture. A typical Flink cluster consists of a client, a Job Manager, and at least one Task Manager. The client creates a data-flow graph that models how the data flows between Flink operators, such as `map` and `reduce`, and deploys

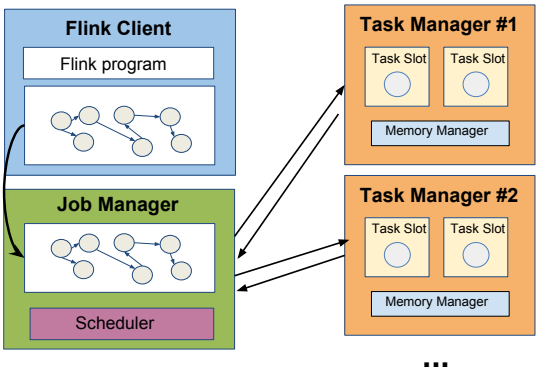

Fig. 1: Overview of the Apache Flink Architecture

it to the Job Manager. The Job Manager is the main process (normally allocated in a dedicated server) that uses round-robin scheduling to distribute the computation among the available Task Managers. It is also responsible for tracking the status of each operation and coordinating the various checkpoints.

The Task Managers are the processes that are executed on the compute nodes of the cluster. These processes define their available resources through task slots that can run pipelines of successive tasks. Additionally, the memory within a Task Manager is equally split among all task slots. For instance, if a Task Manager has four task slots, each pipeline that runs on these task slots will utilize 1/4 of its managed memory.

Apache Flink has two core Application Programming Interfaces (APIs): the data set API, which is used for finite data sets and the data stream API which is used for real time data processing. Furthermore, Apache Flink user functions are regular Java or Scala methods that implement a set of transformations (e.g., `map`, `reduce`, `group`, `join`) on data sets or data streams.

B. Tornado

Tornado is a practical heterogeneous programming framework for Java programs generating OpenCL C code through JIT compilation. One of the main advantages of Tornado is that it supports dynamic configuration allowing developers to identify data-parallel code and the device it should execute on. Tornado consists of three software layers as shown in Figure 2:

1) The Tornado API: Tornado uses a task-based API that allows developers to group tasks together so that the runtime can perform data transfer optimizations automatically. In addition, developers can identify a data-parallel loop by adding a Parallel annotation (@Parallel) before its induction variable for auto-parallelization.
2) The Tornado Runtime: The Tornado Runtime is responsible for performing data management optimizations and coordinating the execution between the host and the hardware accelerators.
3) The Tornado Compiler: Tornado is equipped with a Just-In-Time compiler that dynamically produces machine code for each target device. Moreover, the compiler

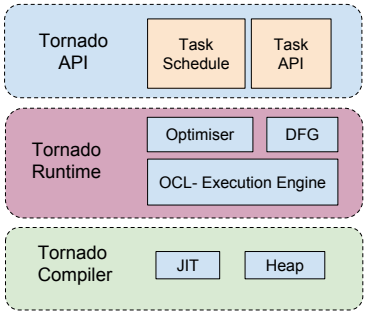

Fig. 2: Tornado Overview

Listing 1: An array multiplication on Tornado

```
1  public void mult(int[] a, int[] b, int[] c) {
2    for (@Parallel int i = 0; i < c.length; i++) {
3      c[i] = a[i]*b[i];
4  }}
5  // Tornado task-composition
6  TaskSchedule s = new TaskSchedule("s0")
7       .task("t0", myClass::mult, a, b, c)
8       .streamOut(c)
9       .execute();
```

communicates with the OpenCL drivers to install the code on the device, perform data allocation, and transfer data between the host and the target device.

Listing 1 shows a multiplication example of two integer arrays in Tornado. As already mentioned, since Tornado has a task-based API we create a *task-schedule*. This schedule contains a single task that stores the `mult` function, the input arrays `a` and `b`, and the output array `c`. Since Tornado does not automatically copy back the result from the accelerator to the main host, a `streamOut` operation is necessary. This way Tornado minimizes the amount of data transfers since only the necessary data will be copied back using this operation. Finally, the execution begins with the `execute` call in line 9.

IV. Enabling GPU Execution within Flink

This section presents our proposal to automatically enable GPU execution in Apache Flink. In our implementation we utilize Tornado in order accelerate Flink applications using heterogeneous hardware resources. The integration of Flink and Tornado takes place in two components of Flink, the Client and the Task Manager. An overview of the integration is illustrated in Figure 3. Our contributions to the existing Flink platform are represented in light-yellow.

a) Extending the Flink-API: On the Client side, we extended Flink with a set of interfaces and classes that enable programmers to exploit Tornado and run on GPUs, while adhering to Flink's programming model.

The UML diagram of Figure 4 presents the new class hierarchy for a Flink `map` transformation in our proposed framework. The purple components of the UML diagram shows our extensions over the existing `MapFunction` interface in Flink. We provide a new interface called `TornadoMapFunction` with a

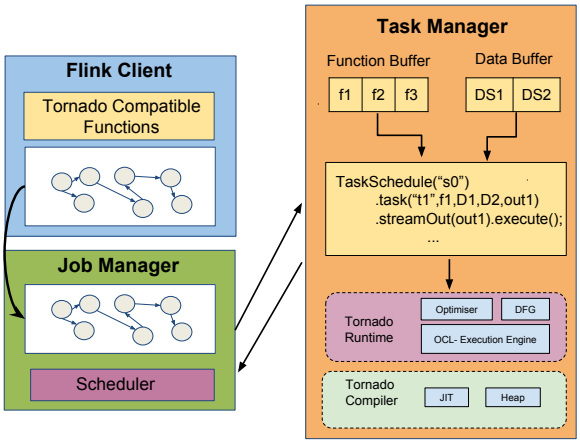

Fig. 3: Flink-Tornado integration overview

set of typed and non-generic new methods `tmap` (Tornado maps). Because Tornado does not currently support Java generics, we define a set of methods with common data types that users can override and execute.

Additionally, we provide a new abstract class called `TornadoMapFunctionBase`, which implements the `tmap` methods with a common template. This is because the Tornado API exposes task-based parallelism while Flink exposes data-parallelism. We provide a small snippet as a template to convert from data parallelism to task-based parallelism in this abstract class. In this way, the full method is compatible with the Tornado programming model while keeping the Flink's semantics.

b) Extending the Task Manager: The second part we have extended to enable Flink's execution on heterogeneous devices is the Task Manager. Flink currently executes its operations in a fine-grained way. This means that when a Flink computation is available in the task manager for final execution, the execution pipeline of operations is defined to be executed element by element. This architecture allows Flink to have fine-grain control over checkpoints and failures. However, this is not suitable for computation on heterogeneous hardware such as GPUs.

To make the execution flow compatible with Tornado and GPUs, we extended the Task Managers to store task information (data sets and input functions) into internal buffers. When a data sink task is deployed to the Task Manager, the Tornado Task Schedules are created and perform all the computations on GPUs using the stored information. The final output is then collected by the data sink similarly to the original Flink implementation. As a result, the user obtains the output in the existing and expected Flink's objects, while the execution has been performed on a GPU.

To illustrate our extensions, we present K-means as an example that demonstrates how a Flink application can be adapted to run on our proposed framework.

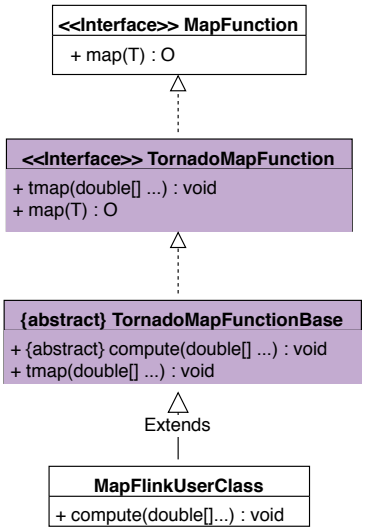

Fig. 4: Flink-Tornado Integration API

Listing 2: Kmeans on Apache Flink: main function

```
1  loop = centroids
2          .iterate(params.getInt("iterations", 10));
3  newCentroids = points
4      .map(new SelectNearestCenter())
5      .withBroadcastSet(loop, "centroids")
6      .map(new CountAppender())
7      .groupBy(0)
8      .reduce(new CentroidAccumulator())
9      .map(new CentroidAverager());
10 finalCentroids = loop.closeWith(newCentroids);
11 clusteredPoints = points
12     .map(new SelectNearestCenter())
13     .withBroadcastSet(finalCentroids, "centroids");
```

A. K-means

K-means is a popular clustering algorithm [19] that groups points with their nearest centroids. Initially, each point is assigned to the centroid closest to it. Then, for each centroid, the mean of the distance between itself and all the points assigned to it is calculated. In turn, the centroid is then moved to the calculated position. This process is repeated until no points change their centroid cluster or after a specified number of iterations.

a) K-means on Apache Flink: Listing 2 illustrates a sketch of the K-means application in Apache Flink. As shown, four transformations (three `map` and one `reduce`) are applied to the points and centroids data sets for a fixed number of iterations. After this pipeline of operations is executed, the points are assigned to the new centroids by applying a `map` transformation to the points and the new centroids data sets (line 12). The functionality of each transformation is defined by user functions.

b) K-means on Flink-Tornado: The process of creating a version of K-means that is Tornado compatible consists of two phases:

Listing 3: Kmeans on Apache Flink: SelectNearestCentroid

```
1  class SelectNearestCenter extends RichMapFunction {
2    @Override
3    public Tuple2<Integer, Point> map(Point p) {
4      for (Centroid centroid : centroids) {
5         closestCentroidId = computeDistance(...)
6      }
7      return new Tuple2<>(closestCentroidId, p);
8    }
9  }
```

Listing 4: Kmeans on Flink/Tornado: SelectNearestCentroid

```
1  class SelectNearestCenter extends
       ↪ TornadoMapFunctionBase {
2    public void compute(double[] centroids_id[], ...,
         ↪ double[] pointX ...) {
3      for (@Parallel int j = 0; j < numOfPoints; j++) {
4        for (int i = 0; i < numOfCentroids; i++) {
5          closestCentroidId = computeDistance(...)
6        }
7        result_id[j] = closestCentroidId;
8      }
9  }
```

1) Adapting the user functions to be compatible with the Tornado API.
2) Collecting all the input data and functions in the Task Managers to create the appropriate task schedules.

Listing 4 illustrates how the user function of Listing 3 has been adapted to be compatible with Tornado. Since Tornado operates on primitive arrays, the arguments of the function are two arrays for the points (one for each x, y coordinates) and three arrays for the centroids (one that stores their ids and two for their x, y coordinates). The code is inside a for loop in order to be executed for all the elements of the points' arrays. The remaining of the user functions were transformed in a similar manner.

The execution then proceeds like in a typical Flink cluster until the Job Manager starts deploying the Execution Graph among the Task Managers. Each task that is deployed to a Task Manager, it stores either data, if it is a data source task, or a user function. When the Task Manager receives the data sink task, all the necessary information to complete the execution is available.

The task schedules shown in Listing 5 represent the K-means control flow as presented in Listing 2. The first task schedule contains one task with the first `map` operation (line 4, Listing 2) that maps each point to its nearest centroid. To simulate the `groupBy` operation in line 7 of Listing 2, two new arrays are created for each centroid that store the coordinates of the points that are mapped to them. Then, a new Task Schedule applies a `reduce` and a `map` transformation on the points of each centroid. The execution of these schedules is repeated for the specified number of iterations. Finally, the Task Schedule in line 20 (Listing 5), maps the points to their new centroids. After the last Task Schedule is executed, the results are consumed by the data sink.

Listing 5: Kmeans on Flink/Tornado: Task Schedules

```
1  public class Task implements Runnable {
2    run() {
3      ...
4      for (int i = 0; i < iterations; i++) {
5        new TaskSchedule("s0")
6          .task("t0", selectnearestcenter::tmap,
               ↪ centroidsId, centroidsX, centroidsY,
               ↪ pointsX, pointsY, sel_id)
7          .streamOut(sel_id)
8          .execute();
9        ...
10       for (int m = 0; m < numOfcentroids; m++) {
11         ...
12         new TaskSchedule("s1")
13             .task("t1", centroidaccum::treduce,
                   ↪ points_x_centr_m, points_y_centr_m,
                   ↪  accum)
14             .task("t2", centroidavg::tmap, accum,
                   ↪ newcentroids)
15             .streamOut(newcentroids)
16             .execute();
17       }
18       ...
19     }
20     new TaskSchedule("s2")
21         .task("t3", selectnearestcenter::tmap,
               ↪ centroidsId, newcentrx, newcentry,
               ↪ pointsX, pointsY, finalOutput)
22         .streamOut(finalOutput)
23         .execute();
24   ...
25   }
26 }
```

V. Related Work

Over the last decade, several research efforts have been made towards accelerating Big Data applications using heterogeneous hardware resources. HadoopCL [20], for example, generates OpenCL code using the Aparapi framework similarly to other state-of-the-art frameworks, like HeteroDoop [21], Glasswing [22] and HeterosSpark [23]. All of these frameworks rely on precompiled OpenCL and native kernels and to the best of our knowledge, no other heterogeneous Big Data framework supports dynamic configuration and JIT compilation in a hardware agnostic manner.

Regarding improving the programmability of heterogeneous hardware devices, Voodoo [24] achieves that by providing an algebraic programming model and a compiler that produces OpenCL C code. Although Voodoo has a similar vision to ours (that is, to tackle the programmability challenges on heterogeneous hardware resources), it targets databases instead of Big Data frameworks, and, consequently, its API focuses on a different kind of functionality. In addition, Voodoo does not support control flow operations, which is an essential part of Object Oriented programming; a vital feature required by the majority of the Big Data frameworks' user programs. Furthermore, Weld et. al. [25] propose a runtime that provides a programming abstraction between disjoint libraries and frameworks that can be integrated into Apache Spark. However, it does not yet target heterogeneous accelerators, unlike our proposed Big Data stack.

VI. Conclusions and Future Work

Enabling Big Data applications and frameworks to use a wide number of compute nodes and heterogeneous hardware resources is still under research. This paper discusses the major challenges to implement such types of applications and it presents our work-in-progress towards improving programmability of Big Data applications in order to enable the transparent use of GPUs and CPUs. Our solution is based on the state-of-the-art Apache Flink and Tornado frameworks, and we showcase their current integration status by using K-means as a use-case. The novelty of our solution is that, although we augmented the Flink's API, we preserve the semantics of original programs in Flink while enabling GPU execution.

In the future, we plan to address other challenges besides programmability such as making our integrated framework fault-tolerant by implementing checkpoints on heterogeneous devices and dynamically partitioning data between heterogeneous hardware efficiently. Additionally, we plan to enable autoparallelization, eliminating the need for the developers to manually mark the data parallel loops using the `@Parallel` annotation. Instead, an extra compiler phase will be added to Tornado that will automatically identify the data dependencies in the loops and try to parallelize the code if possible. Finally, we also aim to target other types of heterogeneous hardware, such as FPGAs, and to evaluate our system on CPUs, GPU and FPGAs.

Acknowledgment

This work is partially supported by the EU Horizon 2020 E2Data 780245. We would like to thank Viktor Rosenfeld, Sebastian Bress, Foivos Zakkak, Steffen Zeuch, and Clemens Lutz for their feedback and valuable discussions.

References

[1] U. Sivarajah, M. M. Kamal, Z. Irani, and V. hanth Weerakkody., "Critical analysis of big data challenges and analytical methods." *Journal of Business Research 70*, pp. 263–286, 2017.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[3] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine." *In IEEE Data Engineering Bulletin, Volume 38*, 2015.

[4] A. Spark. https://spark.apache.org/. [Online]. Available: https://spark.apache.org/

[5] A. Storm. http://storm.apache.org/. [Online]. Available: http://storm.apache.org/

[6] G. C. P. B. J. Barrus. (2017) Gpus are now available for google compute engine and cloud machine learning. [Online]. Available: https://cloudplatform.googleblog.com/2017/02/GPUs-are-now-available-for-Google-Compute-Engine-and-Cloud-Machine-Learning.html

[7] A. W. Services. https://aws.amazon.com/hpc/. [Online]. Available: https://aws.amazon.com/hpc/

[8] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[9] G. Frost. Aparapi. [Online]. Available: https://code.google.com/archive/p/aparapi/

[10] IBM, "Ibm j9 virtual machine," 2018. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.win.70.doc/user/java_jvm.html

[11] J. J. Fumero, T. Remmelg, M. Steuwer, and C. Dubach, "Runtime Code Generation and Data Management for Heterogeneous Computing in Java," in *Proceedings of the Principles and Practices of Programming on The Java Platform*, ser. PPPJ '15. New York, NY, USA: ACM, 2015, pp. 16–26.

[12] J. J. Fumero, M. Steuwer, and C. Dubach, "A Composable Array Function Interface for Heterogeneous Computing in Java," in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ser. ARRAY'14. New York, NY, USA: ACM, 2014, pp. 44:44–44:49.

[13] J. Clarkson, J. Fumero, M. Papadimitriou, F. S. Zakkak, M. Xekalaki, C. Kotselidis, and M. Luján, "Exploiting High-performance Heterogeneous Hardware for Java Programs Using Graal," in *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ser. ManLang '18. New York, NY, USA: ACM, 2018, pp. 4:1–4:13. [Online]. Available: http://doi.acm.org/10.1145/3237009.3237016

[14] C. Kotselidis, J. Clarkson, A. Rodchenko, A. Nisbet, J. Mawer, and M. Luján, "Heterogeneous Managed Runtime Systems: A Computer Vision Case Study," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '17. New York, NY, USA: ACM, 2017, pp. 74–82.

[15] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink®: Consistent stateful distributed stream processing," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1718–1729, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137777

[16] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1789–1792, 2016.

[17] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream processing engines," *arXiv preprint arXiv:1802.08496 (2018).*, 2018.

[18] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams : A fault-tolerant model for scalable stream processing," in *ACM*, 2013, pp. 423–438.

[19] M. Verma, M. Srivastava, N. Chack, A. K. Diswar, and N. Gupta, "A comparative study of various clustering algorithms," in *Data Mining, International Journal of Engineering Research and Applications (IJERA)*, 2012, pp. 1379–1384.

[20] M. Grossman, M. Breternitz, and V. Sarkar., "Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl." *In 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW)*, 2013.

[21] A. Sabne, P. Sakdhnagool, and R. R. Eigenmann, "Heterodoop: A mapreduce programming system for accelerator clusters." *In Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015.

[22] I. El-Helw, R. Hofman, and H. Bal., "Glasswing: Accelerating mapreduce on multi-core and many-core clusters." *In Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, 2014.

[23] P. Li, Y. Luo, N. Zhang, and Y. Cao., "Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms." *In 2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2015.

[24] H. Pirk, O. Moll, M. Zaharia, and S. Madden, "Voodoo - a vector algebra for portable database performance on modern hardware," *PVLDB*, vol. 9, pp. 1707–1718, 2016.

[25] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab, "Weld : A common runtime for high performance data analytics," in *Conference on Innovative Data Systems Research (CIDR)*, 2016.