Efficient Compilation and Execution of JVM-Based Data Processing Frameworks on Heterogeneous Co-Processors

Christos Kotselidis
The University of Manchester
christos.kotselidis@manchester.ac.uk

Sotiris Diamantopoulos
Exus Ltd.
s.diamantopoulos@exus.co.uk

Orestis Akrivopoulos
SparkWorks ITC Ltd.
akribopo@sparkworks.net

Viktor Rosenfeld
German Research Center for Artificial Intelligence
viktor.rosenfeld@dfki.de

Katerina Doka
National Technical University of Athens
katerina@cslab.ece.ntua.gr

Hazeef Mohammed
Kaleao Ltd.
hazeef.mohammed@kaleao.com

Georgios Mylonas
Computer Technology Institute & Press Diophantus
mylonasg@cti.gr

Vassilis Spitadakis
Neurocom Luxembourg
v.spitadakis@neurocom.lu

Will Morgan
IProov Ltd.
will.morgan@iproov.com

This paper addresses the fundamental question of how modern Big Data frameworks can dynamically and transparently exploit heterogeneous hardware accelerators. After presenting the major challenges that have to be addressed towards this goal, we describe our proposed architecture for automatic and transparent hardware acceleration of Big Data frameworks and applications. Our vision is to retain the uniform programming model of Big Data frameworks and enable automatic, dynamic Just-In-Time compilation of the candidate code segments that benefit from hardware acceleration to the corresponding format. In conjunction with machine learning-based device selection, that respect user-defined constraints (e.g., cost, time, etc.), we enable dynamic code execution on GPUs and FPGAs transparently to the user. In addition, we dynamically re-steer execution at runtime based on the availability of resources. Our preliminary results demonstrate that our approach can accelerate an existing Apache Flink application by up to 16.5x.

I. Introduction

Several Big Data frameworks have emerged to address the ever increasing data generation and satisfy the processing needs. Regardless of the programming model or features that each framework offers, their scalability is mainly achieved through the following techniques: (1) scale-up by increasing the resources of a single node (e.g., Saber [24], Streambox [29]), (2) scale-out by increasing the number of nodes (e.g., Apache Spark [34], Apache Flink [14], Storm [6]), or (3) manual implementation of code optimizations specific to the underlying hardware, such as GPU offloading [22], [31].

Typically, the scale-up and scale-out approaches concern CPU-only deployments, while manual scalability in the form of ad-hoc optimizations and hardware acceleration targets the emerging heterogeneous data centres infrastructures. Hardware accelerators are constantly gaining popularity for Machine Learning and Big Data Analytics workloads, since they often outperform general purpose CPU-based implementations, due to their massive capabilities for parallel execution [26] and energy efficiency. As a result, cloud vendors have been motivated to include specialized hardware accelerators in their offerings, along with general purpose resources (e.g., Amazon's EC2 Elastic GPUs [3] or FPGA instances [4], Google's TPU [7]). Recently, cloud/cluster management software systems such as Apache Yarn [32] and Mesos [5] have provided support for heterogeneous hardware through their API over bare metal, Virtual Machines or even Docker containers [8]. The exploitation of heterogeneous hardware accelerators by Big Data applications is a challenging task. This is, mainly, attributed to: (1) the CPU-only homogeneous design assumptions of Big Data frameworks; (2) the fragmentation of programming models across different devices; and (3) the lack of compiler and runtime support for heterogeneous hardware, by the underlying execution engines of the Big Data frameworks - mainly Java Virtual Machines (JVMs).

To enable the exploitation of heterogeneous hardware accelerators by Big Data applications, we propose novel pluggable extensions to the existing software components of a Big Data stack. The proposed stack is capable of adapting itself to the underlying heterogeneous hardware resources, while retaining a unified, high-level programming model. Finally, the proposed extensions are technology-independent and can be adopted by any technology vendor.

II. Challenges

This section presents, in detail, the challenges in exploiting heterogeneous hardware accelerators by Big Data frameworks.

A. Programmability

Contemporary Big Data frameworks such as Apache Spark [34], Apache Flink [14], and Storm [6] are implemented on top of the Java Virtual Machine (JVM) due to their portability across different platforms and operating systems, and interoperability with high-level programming languages like Java and Scala. However, the majority of production JVMs generate code only for CPUs. Consequently, software developers have to generate, ad-hoc, code suitable for execution on a heterogeneous device (e.g., a GPU or an FPGA). Although this approach is commonly found in the literature (e.g., HadoopCL [22], HeteroDoop [31], Glasswing [19], and HeteroSpark [27]), it has a number of disadvantages that limit its general applicability.

Code fragmentation: Developers must integrate different programming models and languages in their code bases [22], [19], [27]. For example, developers have to mix Java and Scala code with low-level CUDA, OpenCL, or similar APIs [2] for GPU acceleration. This creates not only programmability challenges, since programmers with high expertise are required,

978-3-9819263-4-7/DATE20/©2020 EDAA

but also negatively impacts code maintainability by requiring familiarization with different concepts, APIs, and toolchains.

Lack of code portability: The accelerated code segments are developed for a particular device or family of devices [31]. Migrating to a different cloud provider, hardware vendor or even between devices of the same type and vendor [12], [30] requires porting of the corresponding code segments to new devices. This is attributed to the fact that the low-level programming models used for programming heterogeneous accelerators do not adhere to the "write-once-run-anywhere" paradigm of Java.

Lack of dynamism: The underlying JVM cannot reconfigure, at runtime, the accelerated code segments [31], [19], [27]. The lack of dynamism impacts both the performance of the application as well as the potential cost of deployment. Applications are essentially limited to use only the resources they have been programmed for, as they are incapable of dynamically reconfiguring the accelerated code segments.

Ideally, Big Data frameworks should support runtime systems capable of arbitrarily compiling any code segment to any hardware device transparently to the user. Unfortunately, though, adding heterogeneous support on JVMs is a complicated task as several language features (e.g., dynamic code dispatch, automatic memory management, de-optimization) must be properly handled in order not to violate the semantics of the JVM. As a result, with the exception of IBM's J9 GPU support [23], [9], the majority of JVM-based solutions for heterogeneous execution are research prototypes. IBM J9 essentially translates Java 8 parallel streams to GPU code [23], thus limiting the code that can be accelerated to a strict subset of Java. To increase the range of applications that can be transparently accelerated, IBM J9 recently started accelerating Spark workloads on GPUs [9]. Future heterogeneous JVMs for Big Data frameworks should support not only GPUs but also application-specific accelerators (e.g. FPGAs), while allowing the dynamic migration of different parts of the running applications across these devices without restarting them.

B. Task Composition and Scheduling

On homogeneous systems, simple heuristics, such as one worker per core, or other best practices[1] are used to schedule worker threads onto the available physical nodes. Therefore, task composition and scheduling are related to monitoring the task queues of the available workers and selecting those with the lighter load. However, on heterogeneous systems, such scheduling techniques are not applicable, since the computational capacity of the available devices might differ by up to three orders of magnitude [21]. In addition, apart from some cases [10], most of the hardware accelerators can only be managed by a single worker, since they do not allow application virtualization like CPUs do.

To fully utilize heterogeneous hardware, Big Data frameworks will have to non-uniformly distribute the computation of their applications across the heterogeneous resources that exhibit different computational characteristics. For instance, CPUs and GPU accelerators have disjoint memory spaces. As a result, the data must be explicitly allocated and transferred from a CPU to the targeted GPU in order to be accessible by the latter. Additionally, the decision to exploit GPU acceleration is made during scheduling; therefore, the device availability must have been guaranteed till the moment at which execution is actually performed. Otherwise, unpredictable performance behaviours will arise, since the task has to be rescheduled on another GPU (if available) or re-composed for CPU execution. Thus, future Big Data frameworks should factor in the time required to perform bulk copies of data across the heterogeneous hardware resources, as well as being able to dynamically react to load imbalances.

C. Data Processing Granularity

Apart from task composition and scheduling across the whole cluster or the cloud deployment, a heterogeneous Big Data framework should also account for the data partitioning within a node. Different hardware accelerators not only feature different processing capabilities, but may also perform differently under different workloads [15], [16], [20]. Consequently, heterogeneous Big Data systems need to dynamically choose the best data partitioning scheme for each task on a per device basis.

Processing Timeliness: Most hardware accelerators require data to be transferred from the host memory space to their memory space in order to get processed. However, this data transfer incurs a significant performance overhead. To circumvent this inefficiency, software engineers prefer to transfer sufficient amounts of data in advance, to offset this overhead from execution time.

Although this execution characteristic can be seamlessly applied to batch execution models, the same does not hold for stream analytics. In particular, delaying data processing in order to gather enough data to get the best performance out of a hardware accelerator comes at the cost of increased latency. In turn, the increased latency can reduce the validity or value of the returned results, when processing time-critical data that their value is highly dependent on their lifespan. Therefore, heterogeneous Big Data frameworks should consider such trade-offs and adapt themselves to the best combination in order to satisfy the applications' requirements. Choosing enough data to benefit from acceleration while maintaining the latency requirements of applications is a significant challenge.

Fault Tolerant Operation: Guaranteeing the fault tolerance in large-scale clusters or in cloud deployments is of paramount importance. As a result, modern Big Data frameworks exploit checkpointing mechanisms to tolerate nodes' failures [13]. On homogeneous systems, checkpointing is achieved with minimal overhead since the data to be checkpointed is already in the host's memory. On the contrary, on heterogeneous deployments the checkpointing process triggers a data transfer from the accelerator's memory to the host's memory, before the application's state is stored. This operation is quite discouraged in hardware acceleration, since it negatively impacts performance. Thus, identifying the optimal checkpointing granularity in heterogeneous deployments is a first-order challenge.

III. The Proposed Architecture

To address the aforementioned challenges, we are developing a novel Big Data execution framework capable of exploiting heterogeneous resources, dynamically and

[1] https://tinyurl.com/ul83glj

transparently to the user. Figure 1 illustrates the workflow of the proposed framework. As shown, the Big Data Framework receives the user applications and constructs a Job Graph. Next, it identifies on which device these tasks should execute.

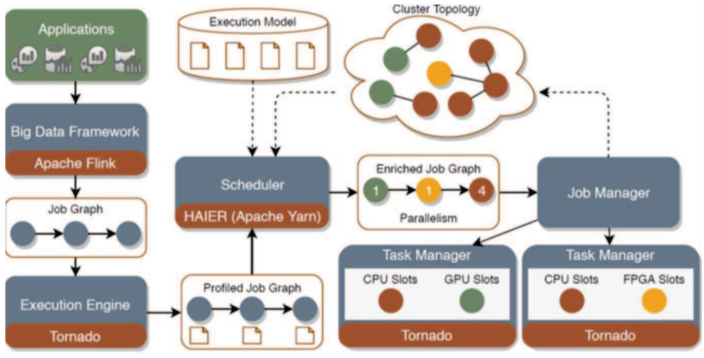

Figure 1: The workflow of the proposed framework.

To do so, it employs the front-end compiler of the Execution Engine to perform a mock partial compilation of each task in order to extract code features (branches, loops, floating point operations, etc.). Then, the Scheduler feeds this information to its Execution Model to predict the hardware device on which tasks should execute to meet the users' requirements. Finally, the Job Manager sends the tasks to the Task Managers that host the desired hardware accelerators. Note that execution is continuously monitored to detect any hardware failures which may trigger a new data partitioning and scheduling to different types of devices. The following subsections present the key software components of the proposed framework in more detail along with their interoperability.

A. Execution Engine

The proposed framework enables heterogeneous execution at the JVM level, where the worker nodes of the Big Data engine runs. The employed technology that we use is based on the TornadoVM [17], [21], [25]; a JVM capable of executing vanilla Java code on heterogeneous devices. TornadoVM works in cooperation with standard JVMs (e.g., HotSpot [28]) allowing the seamless integration of the acceleration functionality to existing deployments.

TornadoVM consists of the following components: 1) **The TornadoVM API** which enables developers to identify code that can be accelerated, as well as composing and building pipelines of multiple tasks, where dependencies and optimizations between the tasks are automatically managed by the runtime layer; 2) **The TornadoVM JIT Compiler** that dynamically generates optimized machine code for heterogeneous devices; and 3) **The TornadoVM Runtime** that performs data dependence analysis, optimizes data transfers, and orchestrates the parallel execution between the host and the heterogeneous target device.

By utilizing TornadoVM, the proposed software stack is able to dynamically –at run-time– compile Java bytecode to machine code targeting CPUs, GPUs, and FPGAs. Furthermore, it is able to profile the executed code and discover the best performing hardware device for each code segment, according to a user-defined optimization policy (maximum performance, minimum cost, etc.).

B. Scheduler

The Hardware-Aware Intelligent Elastic Resource Scheduler (HAIER) maps the tasks to the available heterogeneous resources. To produce an optimal execution plan HAIER analyses input data from 1) compiler extracted code features, 2) Big Data framework's task graphs, and 3) resources' availability. Then, the scheduler allocates each task to the most beneficial set of hardware resources, in order to conform to a user-defined optimization policy. The optimization process is based upon detailed models of the cost and performance characteristics of tasks over various underlying hardware, such as CPUs, GPUs or FPGAs. The models are stored and updated in a model library and whenever a new task graph is scheduled for execution through HAIER, they are used in order to intelligently assign workflow parts to the available hardware according to the user optimization policy. Once the optimal execution plan is available, it is delegated back to the Heterogeneous-aware Big Data framework and be enforced through a cluster management framework that can handle heterogeneous resources (e.g., Apache YARN [33]). The workflow execution is monitored for failures and/or performance degradation, at runtime, allowing HAIER to dynamically adapt to the current conditions by creating a new execution plan for the remaining tasks.

The HAIER scheduler is comprised of the following components. 1) **Planner**: Determines, in real-time, where each task should be scheduled and whether data needs to be moved to/from their current locations and between processing units. Such a decision must rely on the characteristics of the involved tasks, which derive as code features by the compiler, and the underlying hardware they execute upon. 2) **Models Library**: This module consists of machine learning models that describe the behaviour of each hardware processing unit in terms of performance, cost, energy efficiency, etc. 3) **Profiler/Modeller**: The initial task models result from the offline profiling of this module, that directly interacts with the pool of physical resources and the monitoring layer in-between. 4) **Model Refinement**: While the workflow is being executed, the initial models are refined online by this module, by using monitoring information of the actual executions.

Use Case	Processing	Data Type	Volume	Key Algorithms
Health Analytics	Batch	Text	Up to TBs	Alternating Least Squares, Linear Algebra
Natural Lang. Proc.	Batch/Stream	Text	Up to TBs	Lexicographical/Statistical fuzzy Matching
Green Buildings	Stream	Text	Up to 100s of GBs	Reductions, Linear Algebra
Biometric Security	Batch/Stream	Video/Images	Up to TBs	ColorMorph, Computer Vision Algorithms

Table 1: Characteristics of the use-cases.

This mechanism facilitates dynamic adjustments of the models and enables the planner to base its decisions on the most up-to-date knowledge. 5) **Execution Monitor**: monitors the execution to detect possible failures and/or performance degradation and acts accordingly.

C. Big Data Framework

The TornadoVM and the HAIER scheduler are integrated with Apache Flink [14] to form the proposed heterogeneous-aware Big Data framework. Flink follows the common programming model popularized by MapReduce [18]. The developer encapsulates the functionality of the data analysis task into user-defined functions (UDFs) which are passed to operators that model second-order functions, such as map or reduce. Whereas each UDF call processes a single tuple, these operators determine how UDFs process data in parallel [11]. The operators are assembled into a Job Graph which represents a complete data analysis task. To orchestrate the execution of a Flink job on a cluster, Flink transparently ships the application logic to different worker nodes, partitions the data, and initiates and monitors the execution of the operators.

To accelerate the execution of the data analysis task, we fuse UDFs with their second-order operators, so they can be executed in parallel on hardware accelerators. Supporting operations, such as data shuffling, partitioning, or joins, are also fused and executed on the accelerators, so that data is only touched once. Note that we aim to generate the code that runs on accelerators transparently, without any additional intervention by developers.

In Flink, the Job Manager orchestrates the execution of the data analysis task. Each worker node is represented by a Task Manager which offers task slots as a unit of execution. Flink assumes that worker nodes are uniform, and schedules operators on any free task slot. Machines with different capabilities are supported by scaling the number of task slots, that each worker node provides.

However, in heterogeneous systems, the execution capabilities of different devices can differ by orders of magnitudes. This is particularly true when generating custom hardware configurations running on FPGAs. Therefore, we extend Flink to explicitly model the hardware capabilities of the cluster. Task Managers communicate the presence of hardware accelerators as well as their performance and power characteristics to a central cluster manager. Furthermore, Task Managers offer GPU and FPGA specific task slots in addition to CPU-based task slots that Flink already provides. The *Job Manager* in turn distributes work to the appropriate *Task Managers* as determined by the HAIER scheduler.

IV. Preliminary Evaluation

To demonstrate the potential of our proposal we have built a prototype that integrates TornadoVM in the Apache Flink framework. To evaluate its performance we use the kmeans algorithm and run 10 times in batches of 10 jobs each. The batches of 10 jobs ensure that the JVMs have stabilized, while the 10 iterations allow us to observe the reproducibility of the results. TornadoVM extracts a set of eight acceleratable kernels from the k-means Java source code [1] and creates three groups of kernels that can run as pipeline on an accelerator without interacting with the host. Kernel loading, synchronization, and data transfers from and to the host are handled automatically.

For the testbed we use a single node with a 4-core (8-threads) Intel Core i7-7700K CPU and an NVIDIA Quadro GP100 GPU. In our experiment we measure end-to-end execution time, including all data transfers, and data marshalling and un-marshalling to and from the GPU.

Figure 2 illustrates the performance for varying workload sizes. On the y-axis is the average end-to-end execution time of the 10 batches in logarithmic scale, while on the x-axis is the workload size in bytes. The black lines on the bars indicate the standard deviation of the runs. We observe that up to 256 KiB the integrated version performs slightly worse (~10%) than the original Flink. This is due to the fact that at the current state of the integration, the code is always run on the GPU, even if it is not the best option. Note that the proposed framework is able to dynamically choose the best device. After the 256 KiB point and as the workload size increases the integrated version outperforms the vanilla Flink system by up to 16.5x. This is attributed to the fact that by increasing the workload size, we exploit more GPU parallelism and the performance gains compensate for the data transfer overhead. With the whole stack in place and with more effort on optimizing the code generation, we anticipate performance to further improve.

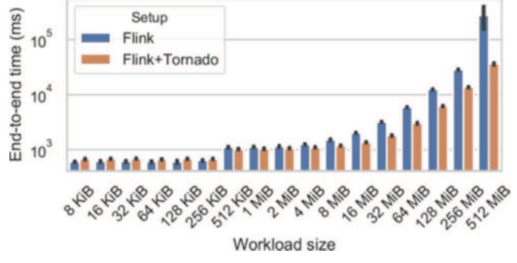

Figure 2: Execution times (Flink+Tornado vs Flink).

Ultimately the proposed stack is going to be evaluated using four applications from four different domains: (1) **Health Analytics**: A real-time streaming use case for predicting patients' hospital re-admissions; (2) **Natural Language Processing**: A sentiment analysis and opinion mining to enable fraud detection; (3) **Green Buildings**: Enabling energy efficient buildings based on analytics of data derived from Internet of Things deployed sensors; and (4) **Biometric Security**: Real-time video recognition to enable biometric authentication. Table 1 presents the features of the use cases that will be used to assess the proposed heterogeneous Big Data deployment. Additionally,

a wide spectrum of hardware configurations will be exploited, including: 1) **x86-based systems with NVIDIA and AMD GPUs, and Intel FPGAs**; and 2) **ARM based systems with Xilinx FPGAs and Mali GPUs**.

V. Conclusions

In this paper we describe the main research challenges towards achieving heterogeneous Big Data frameworks. We identify key software components at all layers of the overall software stack that are necessary to enable true heterogeneous execution of Big Data applications. We propose an integrated software stack that cooperatively implements a heterogeneous Big Data framework. Our preliminary results demonstrate the potential of our approach by accelerating an existing Apache Flink application up to 16x.

Acknowledgments

This work is funded by the EU Horizon 2020 E2Data 780245 programme grant.

Additional Authors

Additional authors: Juan Fumero, Foivos S. Zakkak, Michail Papadimitriou, Maria Xekalaki, Nikos Foutris, Athanasios Stratikopoulos (The University of Manchester, first.last@manchester.ac.uk), Nectarios Koziris, Ioannis Konstantinou, Ioannis Mytilinis, Constantinos Bitsakos (National Technical University of Athens, nkoziris@cslab.ece.ntua.gr, ikons@cslab.ntua.gr, gmytil@cslab.ece.ntua.gr, kbitsak@cslab.ece.ntua.gr), Christos Tsalidis (Neurocom Luxembourg, c.tsalidis@neurocom.lu), Christos Tselios, Nikolaos Kanakis (SparkWorks ITC Ltd., tselioschristos@sparkworks.net, nkanakis@sparkworks.net), Clemens Lutz (German Research Center for Artificial Intelligence, clemens.lutz@dfki.de), Sebastian Breß, Volker Markl (Technische Universität Berlin, sebastian.bress@tu-berlin.de, volker.markl@tu-berlin.de).

References

[1] Kmeans, Apache Flink. https://github.com/apache/flink/blob/release-1.7/flink-examples/flink-examples-batch/src/main/java/org/apache/flink/examples/java/clustering/KMeans.java. [Online;accessed 7. Jun. 2019].

[2] Aparapi. http://aparapi.github.io, Dec 2016. [Online; accessed 5. Nov. 2018].

[3] Amazon EC2 Elastic GPUs. https://aws.amazon.com/ec2/elastic-gpus, Oct 2018.[Online; accessed 5. Nov. 2018].

[4] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1,Oct 2018. [Online; accessed 5. Nov. 2018].

[5] Apache Mesos and GPUs. https://www.nvidia.com/object/apache-mesos.html, Oct 2018. [Online; accessed 5. Nov. 2018].

[6] Apache Storm. https://storm.apache.org,Jun 2018. [Online; accessed 5. Nov. 2018].

[7] Cloud TPUs - ML accelerators for TensorFlow Cloud TPU. https://cloud.google.com/ tpu, Oct 2018. [Online; accessed 5. Nov. 2018].

[8] Enabling GPUs in the Container Runtime Ecosystem.https://devblogs.nvidia.com/ gpu-containers-runtime/, June 2018.

[9] Transparent GPU Exploitation on Apache Spark. https://tinyurl.com/yxdf4oqn, Apr 2018. [Online; accessed 27. May 2019].

[10] Virtual GPU Technology. https://www.nvidia.com/en-us/design-visualization/technologies/virtual-gpu/, Nov 2018. [Online; accessed 29. Nov. 2018].

[11] D. Battr´e, et al.,. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In Proceedings of the ACM Symposium on Cloud Computing, SoCC '10, pages 119–130, 2010.

[12] S. Breß et al., Generating custom code for efficient query execution on heterogeneous processors. The VLDB Journal, 2018.

[13] P. Carbone et al., Lightweight asynchronous snapshots for distributed dataflows. CoRR, abs/1506.08603, 2015.

[14] P. Carbone et al., Apache flinkTM: Stream and batch processing in a single engine. IEEE Data Eng. Bull., 38(4):28–38, 2015.

[15] S. Che et al., Rodinia: A benchmark suite for heterogeneous computing. In 2009 IEEE International Symposium on Workload Characterization (IISWC), pages 44–54, Oct 2009.

[16] S. Che et al., Accelerating Compute-Intensive Applications with GPUs and FPGAs. In 2008 Symposium on Application Specific Processors, pages 101–107, June 2008.

[17] J. Clarkson et al., Exploiting High-performance Heterogeneous Hardware for Java Programs Using Graal. In Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang '18, pages 4:1–4:13, New York, NY, USA, 2018. ACM.

[18] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. Commun. ACM, 51(1):107–113, Jan. 2008.

[19] I. El-Helw et al., 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.

[20] J. Fowers et al., A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-window Applications. In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '12, pages 47–56, New York, NY, USA, 2012. ACM.

[21] J. Fumero et al., Dynamic Application Reconfiguration on Heterogeneous Hardware. In Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2019.

[22] M. Grossman et al., Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl. In 27th International Parallel and Distributed Processing SymposiumWorkshops and PhD Forum (IPDPSW), 2013.

[23] K. Ishizaki et al., Compiling and optimizing java 8 programs for gpu execution. In 2015 International Conference on Parallel Architecture and Compilation (PACT), pages 419–431, Oct 2015.

[24] A. Koliousis et al., Saber: Window-based hybrid stream processing for heterogeneous architectures. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, pages 555–569, New York, NY, USA, 2016. ACM.

[25] C. Kotselidis et al., Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '17, pages 74–82, New York, NY, USA, 2017. ACM.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

[27] 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.

[28] T. Lindholm et al., The Java Virtual Machine Specification, Java SE 8 Edition. Addison-Wesley Professional, 1st edition, 2014.

[29] H. Miao et al., Streambox: Modern stream processing on a multicore machine. In Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '17, pages 617–629, Berkeley, CA, USA, 2017. USENIX Association.

[30] V. Rosenfeld et al., The operator variant selection problem on heterogeneous hardware. In ADMS@VLDB. VLDB Endowment, 2015.

[31] A. Sabne et al., Heterodoop: A mapreduce programming system for accelerator clusters. In Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, 2015.

[32] W. Tan and V. K. Vavilapalli. First Class GPUs support in Apache Hadoop 3.1, YARN&HDP 3.0, Aug. 2018.

[33] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing, page 5. ACM, 2013.

[34] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez,

[35] S. Shenker, and I. Stoica. Apache spark: A unified engine for big data processing. Commun. ACM, 59(11):56–65, Oct. 2016.