Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2017) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

7th International Scientist Conference on Computational Science, 2-6 July 2018, Heraklion, Greece

HetSpark: A Framework that Provides Heterogeneous Executors to Apache Spark

Klodjan Klodi Hidri[a], Angelos Bilas[a], Christos Kozanitis[a]

[a]*FORTH-ICS, N. Plastira 100, V. Vouton, Heraklion 70013, Greece*

Abstract

The increasing computational complexity of Big Data software requires the scale up of the nodes of clusters of commodity hardware that have been used widely for Big Data workloads. Thus, FPGA-based accelerators and GPU devices have recently become a first class citizen in data centers.This is not a trivial task, however, from an engineering effort perspective because developers of distributed computing frameworks, such as, Apache Spark, that are used to writing simple code in a high level language, such as Python, or Scala to use accelerators requires development with low level APIs, such as CUDA, and OpenCL.

Fortunately, recent developments on Accelerator Virtualization, attempt to simplify the use of accelerators developing VineTalk [8] a software layer between FPGAs , GPU devices and applications that reduces the complexity of communication between application software and accelerator hardware.

This paper presents HetSpark, a heterogeneous modification of Apache Spark. HetSpark enables Apache Spark to operate with two classes of executors: an accelerated class, and a commodity class. HetSpark applications are expected to use VineTalk for their entire interaction with accelerators. The schedulers of HetSpark are sophisticated enough to detect the existence of VineTalk routines in the java binary code. Thus, they take decisions as to which tasks require the use of accelerators, and they send them only to executors of the former class.

Finally, we evaluated thoroughly the performance of HetSpark with different mixes of executors of the two different classes. When applications run linear tasks, we observed that the use of CPU-only accelerators is preferable to GPU enhanced accelerators, while for applications with computationally challenging tasks, the time savings from the use of GPUs compensate for data transfers between commodity and accelerated executors.

Peer-review under responsibility of the scientific committee of the 6th International Young Scientist conference in HPC and Simulation.

Keywords: Apache Spark; Apache Mesos; GPUs; heterogeneity

* Klodjan Klodi Hidri

E-mail address: hidri@ics.forth.gr

Peer-review under responsibility of the scientific committee of the 6th International Young Scientist conference in HPC and Simulation.

1. Introduction

Apache Spark [11] is a distributed computing framework that benefits from the parallel execution of simple tasks to all nodes of a cluster. Spark partitions a large dataset into small partitions and distributes them into all nodes of a cluster. Spark programs consist of a number of jobs, each of which contains a series of tasks that the Spark master broadcasts them to all executors to run them in their individual partitions of the data.

For many years, disk and network comprised the main bottlenecks of Spark applications. Applications typically consisted of computationally simple linear tasks, and Spark spent most of the time loading the data and shuffling them between executors. To deal with these bottlenecks, Spark users could 1) increase the parallelism of their deployment by adding more compute nodes, and 2) write their applications carefully to minimize data shuffles.

The increasing computational complexity of Big Data applications, however, adds an extra bottleneck to Apache Spark: CPU. Given the abundance of non linear routines that modern pipelines require to execute, little benefits are expected from scaling out cluster deployments. Instead, a more productive practice is the scale up of the nodes of clusters of commodity hardware. Thus, FPGA-based accelerators and GPU devices have recently become first class citizens in modern datacenters.

Given that Spark sends the same code to run to all executors, the scale up of compute nodes of Apache Spark introduces a difficult trade off. On one hand, having a cluster with hundreds or thousands of accelerator enhanced executors is not sustainable, given the high costs of accelerators. On the other hand, a small cluster, where all executors are equipped with accelerators becomes less capable to handle large datasets, because data loads dominate.

In order to address this trade-off, users of Apache Spark break their applications into simpler sub-programs, run the individually in appropriate cluster, depending on the processing needs of each sub-program, and they use permanent storage as a means of communication. For example consider a Deep Learning application that receives as input unstructured data. A Spark program would first enforce some structure to the data through the use of Extract-Transform-Load (ETL) routines. Then it could take the structured transformation of the input and use it to train a Neural Network through its ML library [9] or through its TensorFlow interface [1]. However, those two stages have different processing needs. The ETL stages benefits from parallelism, and thus it requires the scale our of distributed storage and memory to run fast. On the other hand, the Neural Network stage needs the use of GPUs to reduce training time. Unfortunately, given that Spark expects that all executors have identical configurations, the entire application would need unsustainably large numbers of GPUs to run. Thus, the common practice for developers today is to run ETL as a standalone program in a highly parallel cluster, and then run training individually in a smaller cluster where all nodes are equipped with GPUs.

A more productive alternative to Spark users would be an enhancement of Spark core to be able to operate on executors with heterogeneous characteristics. As it is shown in [6], there are specific classes of workloads that benefit from the deployment of compute intensive tasks into remote GPUs, as processing - and not network- comprises the main bottleneck. Thus in the example of the previous paragraph, one could run both the ETL and the Neural Network training component as one application on a cluster that consists of a big number of commodity workers, and a small number of GPU enhanced workers.

We introduce *HetSpark*, a modification of Apache Spark which enables it operating with heterogeneous executors. In more detail:

- *HetSpark* depends on VineTalk [8] software package, which provides an abstraction layer between application software and physical accelerators. VineTalk enables cluster managers, such as Apache Mesos [3] to offer fractions of accelerator resources to frameworks.
- *HetSpark* uses Mesos as a cluster manager to negotiate resources to launch executors. *HetSpark* accepts two different classes of offers: A class that consists of VineTalk abstract units, and a class that does not. For the executors of the former class, *HetSpark* also initiates the back end of VineTalk and imports its API into the namespace of the executor process.
- *HetSpark* expects that a Spark application submits tasks to GPUs through the use of VineTalk API. The modified driver of *HetSpark* inspects the contents of the *closure* of tasks that Spark submits to workers to detect the presence of VineTalk related routines. In case VineTalk routines are detected in the class, the scheduler uses a VineTalk-equipped executor to dispatch the task. Otherwise it uses a conventional one.

We thoroughly evaluate *HetSpark* through a series of experiments with both compute intensive and I/O intensive tasks. We plot the performance of each application as it runs in Apache clusters with a varying mix of GPU-enhanced and conventional executors. We show that compute intensive tasks are insensitive from the fact that they have to transfer data to only a few nodes that run GPU-enhanced executors, and they still achieve a speedup of more than 6× even with the presence of a single GPU-enabled executor in a cluster of mostly commodity executors, and the running task is as demanding as Monte Carlo simulation.

The rest of this paper is organized as follows. Section 2 introduces the different technologies on which *HetSpark* depends, Section 3 provides the description of *HetSpark*, and Section 4 describes the evaluation methodology and the results that *HetSpark* achieves.

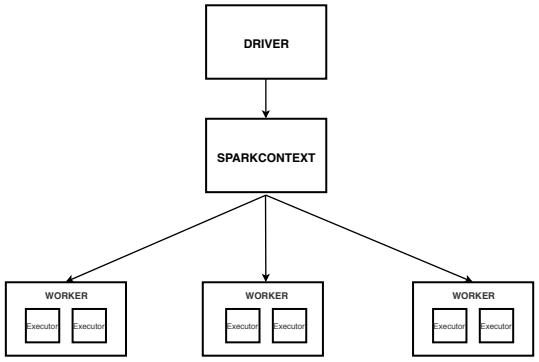
2. Background

2.1. Apache Spark

The main abstraction of Spark is called Resilient Distributed Dataset (RDD), and it represents a distributed collection of elements of the same type. At a high level, Spark applications create RDDs out of some input, apply a series of transformations on them using their API, and finally they call actions to collect or store data.

Figure 1 displays the general architecture of Apache Spark, which follows the master-slave model. The Spark *driver* runs the main Spark program, and the workers perform transformations and actions on RDDs. RDDs divide their elements into partitions, and distribute these partitions to the memory of all executors. Then, whenever a user program specifies a function to run on all elements of RDDs, the driver sends the respective code binaries with all necessary dependencies to executors, and the latter run them across all elements of the partitions that they own.

Fig. 1: Spark Architecture



The ecosystem of Apache Spark consists of the Spark core, which is the main Spark engine, and a rich suite of tools that have been implemented on top of it, such as, Spark SQL [2], Spark Structured Streaming [12], Spark MLlib [9] and Spark GraphX [5] that operate on top of Spark Core.

The rest of this subsection, describes the internals of Apache Spark in more detail. Figure 2 displays a high level architecture of Spark core.

Spark cluster launching. We run Spark over Mesos [3] because Mesos enables fine-grained sharing cluster resources such CPU cores, GPU cores and memory by giving to Spark a common interface for accessing resource offers. A Spark program launches executors immediately upon its submission to the master. Without loss of generality, we describe here the launching procedure when a computer cluster is managed by Apache Mesos. Mesos first creates MesosCoarseGrainedSchedulerBackend, a SchedulerBackend that launches Spark driver, prepares the number of executors, runs tasks on Mesos using coarse-grained tasks, sends repeatedly every few seconds offers of available resources to Spark driver. MesosCoarseGrainedSchedulerBackend reads the preferences of user such overall number of cores of cluster, number of cores per executor, amount of memory per executor from config file and computes resources for launching executors. It then compares acquired CPU cores from user with available CPU cores ,acquired memory from user with available memory and number of executors with executors limit. If all comparisons are true, then the Mesos driver launches a CoarseGrainedExecutorBackend that sends a request with executor resources to spark driver to register executor and than decreases the number of available CPU cores by acquired CPU cores from user, reduces the amount of available offered memory by acquired memory from user. Each created CoarseGrainedExecutorBackend communicates with spark driver and calls a Remote Procedure Call (RPC) with a request to register executor. When the Spark driver receives the request, it registers executor on executorDataMap (a hashmap data structure where key is a executor id and value are executor data CPU cores,memory) and finally order CoarseGrainedExecutorBackend to launch the executor.

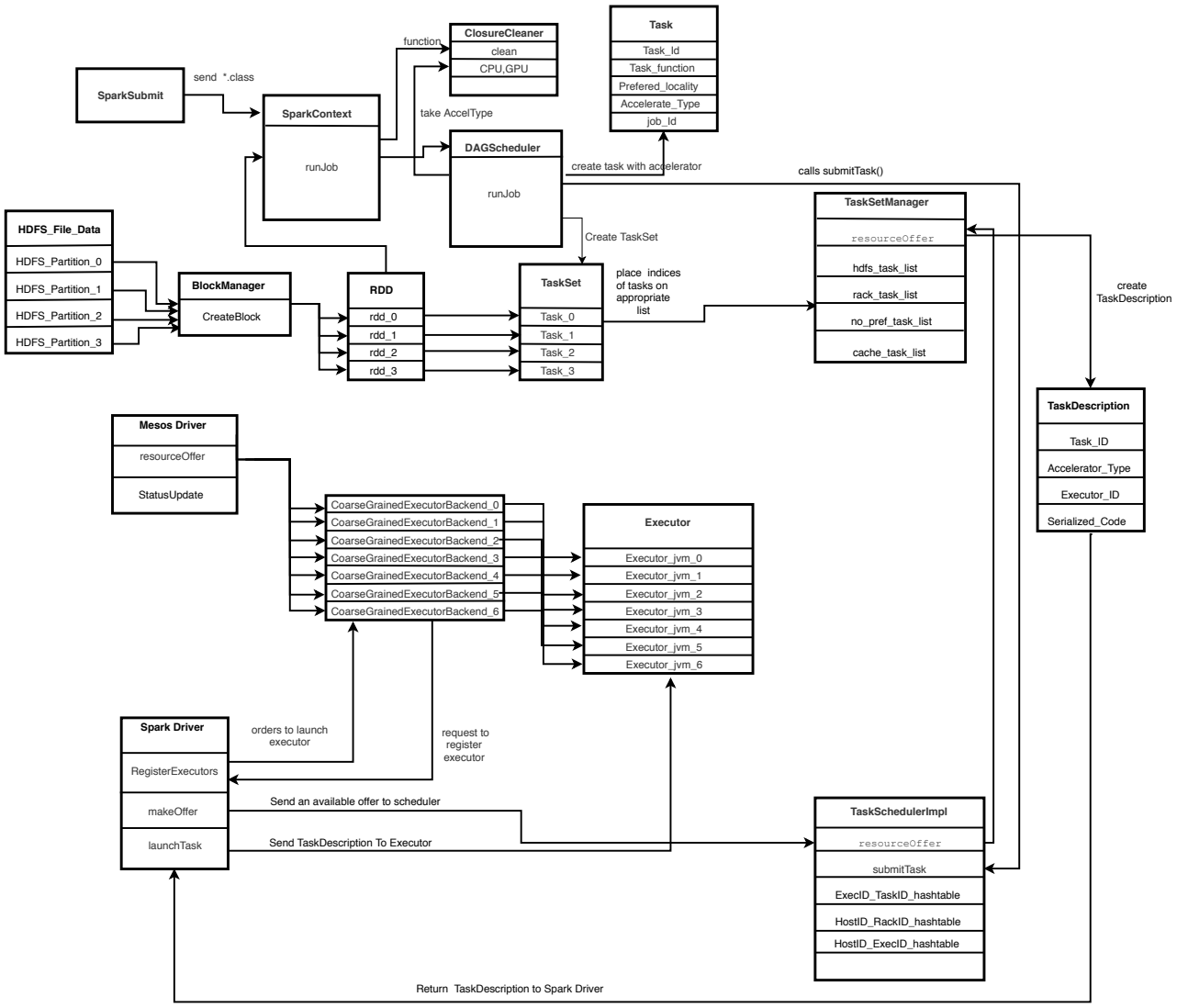


Fig. 2: Spark Core Architecture

Spark Scheduling. Spark uses two levels of scheduling for its tasks. The first level of scheduling is the DAGScheduler, which groups tasks into stages. Stages contain tasks with no dependencies between each other which run in parallel. DAGScheduler computes an execution Directed Acyclic Graph of stages in topological order for a job, determines the preferred locations as we can see in Table 1, handles failures, creates a collection of tasks a TaskSet. A TaskSet belongs to a single stage, contains an independent sequence of tasks that can run on parallel over the data on the cluster. DAGScheduler submits TaskSets to TaskScheduler. TaskScheduler determines a scheduling order across jobs. It creates a TaskSetManager that manages scheduling of tasks in a TaskSet. TaskSetManager registers the task as pending execution, matches each task of a TaskSet collection by an executor that receives as offer from driver creating a TaskDescription. TaskDescription is a class that contains serialized task code, the id of task and id of executor and is traveled on Spark Driver. Spark Driver than checks the id of executor from TaskDescription to send the serialized task code over RPC to this executor for execution .

Cleaning of the Spark Closure. Functions in Scala are object instances where the compiler creates an anonymous class for each function instance. For each method and function literal that are passed as function arguments, the

Process_Local	data is in the same JVM as the running code
Node_Local	data is on same node but not in executor_JVM
No_Pref	data is accessed anywhere
Rack_Local	data is on same rack
ANY	data is elsewhere on the network

Table 1: Preferred Locations

Scala compiler generates a binary file (anonym$$N.class N=0,1,2,..,N). Before sending to executors code to run as anonymous classes, Spark cleans up all their unnecessary references from their closure.

2.2. Java Virtual Machine

Java code is executed inside threads. Each thread has its own execution stack, which consists of frames that represent methods invocations. When methods are invoked, their frames are pushed on the current threads execution stack, and when methods return, frames are popped from the stack. A frame contains a local variables array, an operand stack and a constant pool. Local variable array holds the values of method's parameters and local variables. Operand stack fetch/stores the values from/to local variable array and constant pool using pushing/popping. Constant Pool is a data structure that contains all references of variables and methods as a symbolic reference, and is created when a java class is compiled. Java compiler translates the source code into a bytecode. Bytecode is an instruction set of the Java virtual machine. A java bytecode instruction consists of an opcode for identifying instruction. As an example, Table 2 show some bytecode instructions :

ILOAD, LLOAD, FLOAD, DLOAD, ALOAD	read a value from local variable array and push it on the operand stack
ISTORE, LSTORE, FSTORE, DSTORE, ASTORE	pop a value from the operand stack and store it in a local variable
GETFIELD, GETSTATIC	field instructions that loads the value of a field of an object
PUTFIELD, PUTSTATIC	field instructions that stores the value of a field of an object

Table 2: Bytecode Instructions

To manipulate and analyze the java bytecode on cleaning procedure Spark uses ASM [7] framework. ASM is a framework that modifies classes, generates new classes, transforms and analyzes bytecodes before java code is loaded into the Java Virtual Machine. It manages the constant pool and the class structure such annotations, fields, methods.

2.3. VineTalk

VineTalk [8] is a transport layer between applications and accelerator devices. It consists of two components; a Communication Layer and a Software Controller. The communication layer implements and manages virtual accelerators, which are called Virtual Accelerator Queues (VAQ), while the controller schedules and executes tasks to the underlying physical accelerators. Applications use the VineTalk API to add their tasks to VAQs and two static variables as flags to select accelerator type VineAccerator.Type.CPU for CPU accelerators and VineAccelerator.Type.GPU for GPU accelerators. As soon as a task is issued and there are available executors, the controller fetches it from the corresponding VAQ queue and executes it in the physical accelerator. At the end of a task execution, the VineTalk controller adds the results of the execution to the appropriate VAQ, and it notifies the the issuer that the current task has finished.

3. HetSpark: An integration of Apache Spark with VineTalk

This section describes the main components of HetSpark. The rest of this section is organized as follows. Section 3.1 how HetSpark executors differ from the Apache Spark. Section 3.2 describes how HetSpark negotiates with Mesos executors with different characteristics. Section 3.3 describes the scheduler of HetSpark, and how it schedules tasks to different executors. Finally, Section 3.4 describes how the core of HetSpark detects which classes need GPU resources to route them exclusively to those executors that provide GPU support through VineTalk.

3.1. HetSpark executor

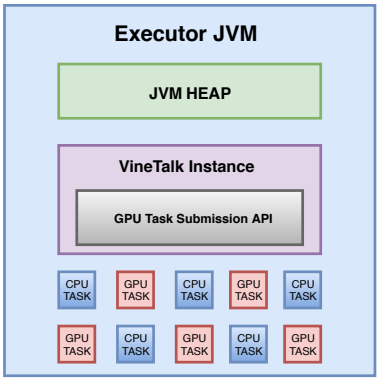


Fig. 3: Executor's Architecture with VineTalk.

Figure 3 displays the position of VineTalk on a Spark executor. A VineTalk instance runs as a service on every executor and serves those Spark tasks that call the VineTalk API.

The main reason behind our decision to have VineTalk run as a service on each executor is overhead minimization. The alternative and more obvious approach would be to have the Spark driver pack VineTalk into the closure of a task, together with commands for its start-up and termination, and then have all tasks deserialize the software, initialize VineTalk in isolation, and destroy it at the end. However, as multiple tasks run the same code on each executor for different partitions, and given that the initialization and termination of VineTalk involves significant overheads due to the allocation and release of large memory segments, the expected overheads lead to prohibitive performance overheads. Thus, VineTalk should be initialized only once for all tasks that are going to use it, and since it also has to be included in the namespace of each task, the best possible component that should initialize it is the Spark executor itself.

3.2. How HetSpark creates heterogeneous executors

Although in the default configuration of Apache Mesos, the cluster manager that we use in our deployment, the only offered resources include CPU cores, Memory, and when available, whole GPUs, we have used a modified Mesos installation, which offers fractions of GPU resources as well. The reasoning behind this approach is to enable multiple executors of a worker node to have shared access to GPUs. The modified installation launches the back end of VineTalk on each Mesos agent that provides GPU resources, and it replaces the offer currency with Virtual Accelerator Queues (VAQs) which are VineTalk-specific buffers.

HetSpark operates with two classes of executors concurrently. One class is the class of commodity executors, and it expects from users to use the same variables as the ones that the Apache Spark provides, such as *spark.executro.cores*, to specify the proper executor configuration, in terms of CPU cores, and memory. The other class is the class of accelerated executors, which are permitted to use GPU resources. For this class of executors, HetSpark provides to users a different set of variables for their configuration. These variables are called spark.executorAcc.memory, spark.executorAcc.cores, spark.executor.vaqs, and they specify the configuration of accelerated executors with respect to memory, CPU, and GPU resources expressed as VineTalk VAQs, respectively.

The HetSpark Master negotiates and creates through Mesos executors that match the configuration of both sets of variables. For each Mesos offer that does not contain VAQ resources, but covers the resource requirements of commodity cluster variables, the Spark Master creates a commodity executor, while it creates an executor of the accelerated class when it receives an offer that covers the desired VAQ resources, together with CPU, and memory.

3.3. How HetSpark schedules tasks to different executors

DAGScheduler submits TaskSet collection on TaskScheduler. For each submitted TaskSet, a new TaskSetManager is created. TaskSetManager contains sets of pending task indices of TaskSet to keep track of the tasks pending execution per executor, host, rack or with no locality preferences. These collections are actually treated as stacks, in which new task indices are added to the end of the sets and removed from the end. Table 3 summarizes these sets.

pendingTasksForExecutor	set of pending tasks for each executor (PROCESS_LOCAL tasks)
pendingTasksForHost	set of pending tasks for each host (NODE_LOCAL tasks)
pendingTasksForRack	set of pending tasks for each rack (RACK_LOCAL tasks)
pendingTasksWithNoPrefs	set containing pending tasks with no locality preferences (NO_PREF tasks)
allPendingTasks	set containing all pending tasks (ANY tasks)

Table 3: Sets of Tasks

TaskSetManager adds all indices of tasks from TaskSet to the above pending sets in reverse order to get launched first tasks with low indices. Checks the preferred locality of each task to write indices on suitable pending set.

Spark driver receives all resource offers from Mesos driver and than calls resourceOffers method of TaskScheduler to make a TaskDescription. ResourceOffers method extracts the localities of each task of TaskSet and than calls resourceOffer method of TaskSetManager by passing executor id, host id and task localities as parameters. The resourceOffer method of TaskSetManager responds to an offer of a single executor from the TaskScheduler by finding a task that will be executed by this executor. Figure 4 represents the scheduling of four tasks. First checks preferred localities to search on appropriate set for task_index. Second gets the value (task_index) from the end of set and than takes a task using task_index as index to TaskSet collection to create a description of a task (TaskDescription) that will be passed onto executors to be executed.

We change resourceOffer method of TaskSetManager to direct tasks_gpu on accelerated_executors. First get type of executor that sends TaskScheduler. If an executor is accelerated than algorithm is the same as we described above because accelerated_executors can execute two kinds of tasks. If an executor is default than we iterate all TaskSet and check the type of each task. If task is task_cpu than get this task from TaskSet by task_index and break the iterate but if task is task_gpu iterate on next tasks of TaskSet until find task_cpu. If doesn't find any task_cpu than waits for receiving accelerated_executor from TaskScheduler in next offers.

3.4. How HetSpark detects which tasks need GPUs

The mechanism that enables HetSpark to detect which tasks contain calls to GPU kernels is based on the closure clean up mechanism of Spark. We use ASM [7] framework to represent java bytecode of binary files before the application code is loaded into the Java VM. HetSpark writes the sequential bytecode of each binary file in a bytearray and than creates an instance ClassReader passing as argument the array (new ClassReader(bytearray[])). By calling the accept() method, ClassReader fires all visiting events corresponding to the bytecode structure. VineAccelerator.Type.GPU variable is constant and it's value is known at compile time where compiler replaces the constant name everywhere in the code with its value on constant pool. We use MethodVisitor class to trace all bytecode of methods and check all types of parameters in invoked methods to detect the GPU flag. MethodVisitor manages automatically to fetch and return VineAccelerator.Type.GPU's value from constant pool. The value of flag will be send on DAGScheduler when creates tasks.

After cleaning the function code SparkContext orders DAGscheduler to begin a job giving RDD and cleaned function as arguments where every job corresponds to an action. DAGScheduler begins creating a RDD graph to solve dependencies between tasks. It first computes the number of partitions of RDD to know the number of tasks will

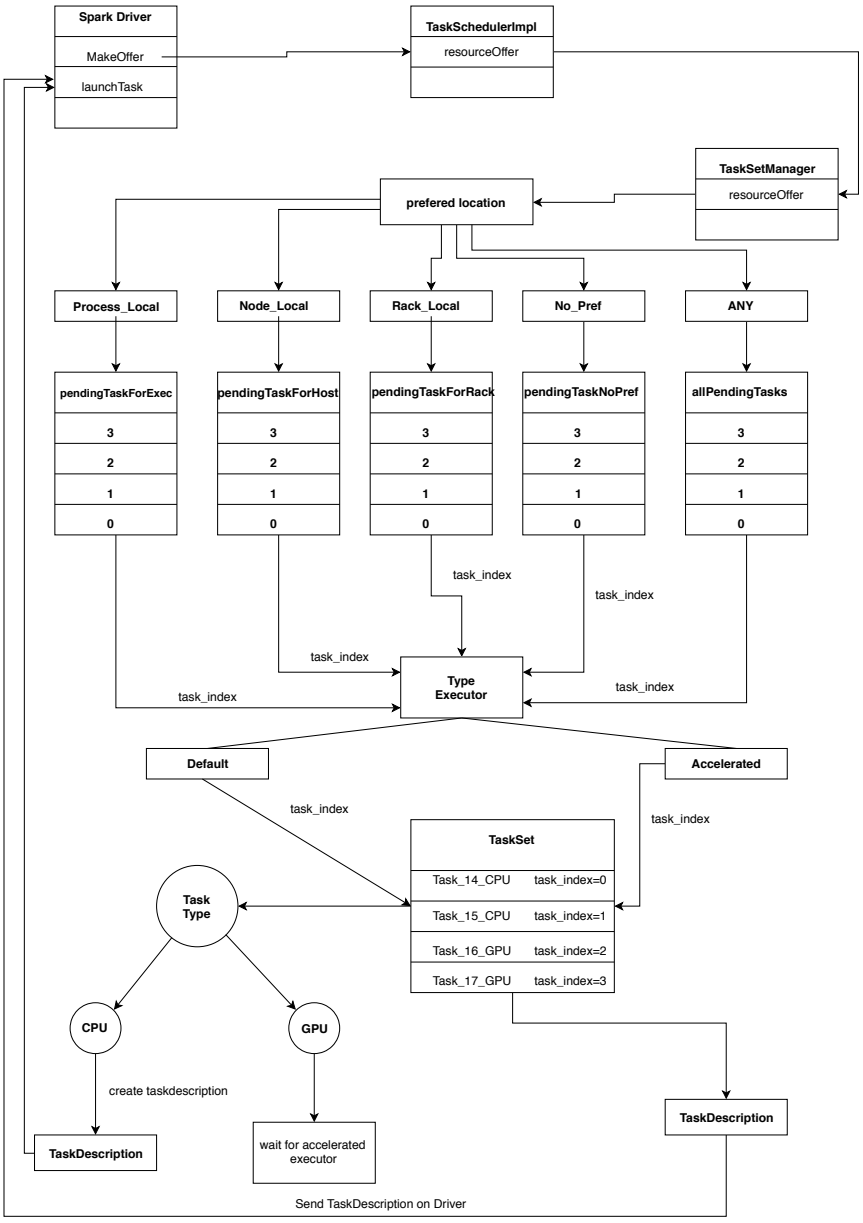


Fig. 4: Task Scheduling with four tasks.

be created and second computes preferred locations, where to run each task in a stage based on the preferred locations. Preferred locations are useful for making better performance on Spark jobs succeeding function code that operates on data to be together on same node.

For each RDD partition, HetSpark creates a task using the partition number of RDD as id of task, the serialized function code, a preferred location and job id. At this point we use a new field, with name Accelerator_Type that is determined in clean procedure of function code. If a spark program does not use VineTalk, the value of Accelerator_Type will be null, and it means that the respective function code of the task will be executed on a CPU. If, on the other hand, the value of Accelerator_Type's is *GPU*, the respective function code will run on a GPU accelerator.

4. Results

We evaluated HetSpark on two benchmarks. The first benchmark is mostly I/O intensive, and it transforms every partition of an input dataset of size 99 GB into the result of the application of the Black&Scholes [10] algorithm. The second benchmark is a compute intensive benchmark, and it consists of Monte Carlo [4] simulations on a dataset of 2 GB. We ran the experiments on our local infrastructure, which consists of 5 identical servers, each of which consists of 32 Intel CPU cores *Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz* , and 250GB of RAM. Two of the nodes were also equipped with an NVidia GPU device *NVIDIA GeForce GT 720*.

We ran experiments using two distinct cluster configurations. A configuration of two compute nodes, and one with all 5 nodes of our infrastructure. In each configuration, as a baseline we used an Apache Spark cluster, which consists of identical executors with 4 CPU cores, and 50GB of memory.

Overall, the configurations that we used are the following.

1. A cluster with two compute nodes with the following two configurations. The point of this experiment is to enable Apache Spark to operate with a fully homogeneous GPU-enabled cluster.

   (a) As a baseline of the 2-node experiment, we used a configuration that consists of 8 commodity executors (4 executors per node).
   (b) As a homogeneous GPU-accelerated cluster, we used a cluster that consists of 8 executors, each of which has access to a GPU through VineTalk, and they are spread out evenly between the two nodes.

2. A total of 5 compute nodes of our infrastructure. This configuration comprises a heterogeneous cluster, since only two of the nodes are equipped with a GPU.

   (a) A baseline of 20 commodity executors, which servers as the baseline.
   (b) 16 commodity executors (4 executors per node similar to baseline), and four GPU accelerated executors, which run on the same node and share a GPU through VineTalk.
   (c) 12 commodity executors (4 executors per node similar to baseline), and 8 GPU-enhanced executors, which are split evenly between the two nodes that are equipped with a GPU.

In the case of the running of the Black&Scholes algorithm in the cluster of 5 compute nodes (Figure 5(a)), the baseline achieves the best performance, as it takes only 7.52 minutes to complete, while the best GPU configuration takes 9.74 minutes. This is not surprising as the most demanding task in the computation is the transfer of large amounts of data towards the GPU executors, while the actual computation itself is of negligible complexity.

In order to verify that the computational demands of the Black&Scholes algorithm are negligible, we ran the execution on a small heterogeneous cluster with 2 nodes. In such a cluster, both the CPU baseline, and the homogeneous GPU configuration have identical data transfers. In this setup, the GPU-enhanced cluster achieves a speedup of only 1.09, which verifies the hypothesis that the processing requirements of the Black&Scholes algorithm are too small. Thus, the gains from a possible acceleration of such a routine cannot dominate over overheads from data transfers.

The opposite occurs in a more computationally demanding benchmark. In the Monte Carlo runs (Figure 5(b)), the gains from the usage of even a single GPU (shared among 4 executors) are indisputable. The CPU-only cluster was taking too long to run and we interrupted its execution after 9 hours, while even the use of a single GPU manages to reduce the execution time dramatically. Thus, in this case, it makes perfect sense to have a small number of GPU-enhanced executors, and let HetSpark move its partitions towards those executors to benefit from the speedup.

5. Conclusions

Unlike previous work, where Apache Spark became able to use GPU accelerators, but only through executors with identical configuration, this is the first work that allows Apache Spark to work with truly heterogeneous executors. HetSpark builds on recent work that enables cluster managers, such as Apache Mesos, to offer to frameworks fractions of GPU resources. The task manager of HetSpark manages two classes of executors: a class with commodity executors,

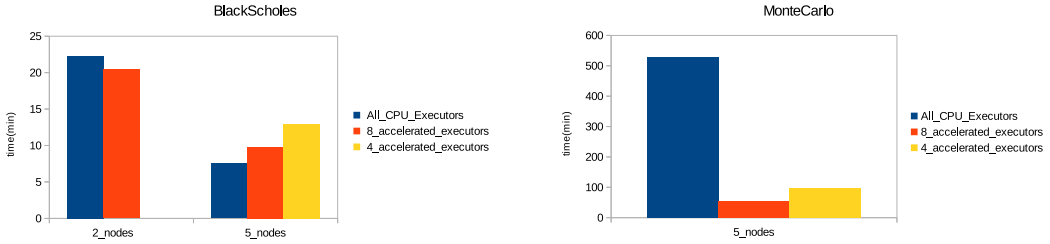


Fig. 5: (a) BlackScholes (b) MonteCarlo.

and a class with GPU-enhanced ones. It inspects the Java closure that the driver sends to executors, and in case it detects GPU kernel invocations, it sends the tasks to run to the latter class. HetSpark is useful to complex Spark applications with tasks of different needs. Data intensive Spark tasks that benefit from high parallelism can use a large number of low cost executors, while at the same time compute intensive tasks can send their code and data to run to a privileged pool of accelerated executors.

Acknowledgments

We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the VINEYARD (H2020-ICT-687628) project. We are thankful to Polyvios Pratikakis for valuable advice on the internals of the Java Virtual Machine and the Spark core. We also thank Stelios Stavridis, and Manos Pavlidakis for their support with the Vinetalk software, Christi Symeonidou for her implementation of Spark programs that use Vinetalk, Efstratios Gelagotis for enhancing resource offers of Apache Mesos with Vinetalk VAQs, and Theodoros Zois for valuable advice on Mesos deployments.

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al., 2016. Tensorflow: A system for large-scale machine learning., in: OSDI, pp. 265–283.

[2] Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., et al., 2015. Spark sql: Relational data processing in spark, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM. pp. 1383–1394.

[3] Benjamin Hindman, Andy Konwinski, M.Z.A.G.A.D.J.R.K.S.S.I.S., . Mesos: A platform for fine-grained resource sharing in the data center .

[4] Bortz, A.B., Kalos, M.H., Lebowitz, J.L., 1975. A new algorithm for monte carlo simulation of ising spin systems. Journal of Computational Physics 17, 10–18.

[5] Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I., 2014. Graphx: Graph processing in a distributed dataflow framework., in: OSDI, pp. 599–613.

[6] Kanellou, Eleni adn Chrysos, N., Mavridis, S., Sfakianakis, Y., Bilas, A., . Gpu provisioning: The 80 - 20 rule, in: Proceedings of the 24th International European Conference on Parallel and Distributed Computing.

[7] Kuleshov, E., 2007. Using the asm framework to implement common java bytecode transformation patterns. Aspect-Oriented Software Development .

[8] Mavridis, S., Pavlidakis, M., Stamoulias, I., Kozanitis, C., Chrysos, N., Kachris, C., Soudris, D., Bilas, A., 2017. Vinetalk: Simplifying software access and sharing of fpgas in datacenters, in: Field Programmable Logic and Applications (FPL), 2017 27th International Conference on, IEEE. pp. 1–4.

[9] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al., 2016. Mllib: Machine learning in apache spark. The Journal of Machine Learning Research 17, 1235–1241.

[10] Turnbull, S.M., Wakeman, L.M., 1991. A quick algorithm for pricing european average options. Journal of financial and quantitative analysis 26, 377–389.

[11] Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I., 2010. Spark: Cluster computing with working sets. HotCloud 10, 95.

[12] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I., 2013. Discretized streams: Fault-tolerant streaming computation at scale, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM. pp. 423–438.