[This doc is shared publicly.]

Accelerator-aware scheduling in Apache Spark 3.0[1]

Authors:
Xingbo Jiang, Xiangrui Meng, Xiao Li, Wenchen Fan (Databricks)
Andrew Feng, Jason Lowe, Madhukar Korupolu, Thomas Graves (NVIDIA)
Yinan Li (Google)
Yu Jiang (Alibaba)

Revisions:

- 2019/02/14: initial draft mainly on standalone mode
- 2019/02/20: added section for YARN, reviewed the draft
- 2019/02/21: added section for k8s, reviewed the draft
- 2019/02/21: re-organized stories and addressed remaining comments

Table of contents

[1] Apache Spark 3.0 is not yet an official release. We expect it to be the next major release.

[This doc is shared publicly.]

Terminology

- P0: MUST have in Spark 3.0. P0s define the MVP.
- P1: SHOULD have in Spark 3.0.
- P2: COULD have in Spark 3.0 (after P0s/P1s and w/ additional resources).
- P3: WON'T have in Spark 3.0 but could be added in later releases.
- "GPU": We use "GPU" for simplicity where general accelerators might apply.

Goals

- Make Spark 3.0 GPU-aware in standalone, YARN, and Kubernetes.
- No regression on scheduler performance for normal jobs.

Non-Goals

- Fine-grained scheduling within one GPU card.
    - We treat one GPU card and its memory together as a non-divisible unit.
- Support TPU.
- Support Mesos.
- Support Windows.

Personas

- Admins who need to configure clusters to run Spark with GPU nodes.
- Data scientists who need to build DL applications on Spark.
- Developers who need to integrate DL features on Spark.

User Workflow

Admin:

1. Configure standalone/YARN/k8s cluster managers to discover GPUs on worker nodes.
2. Provide users instructions on how to request GPU in their applications.

User/Developer:

[This doc is shared publicly.]

1. Use spark-submit to start a Spark application and request GPU resources per executor, optionally set the default number of GPUs per task.
2. Request number of GPUs to use (if non-default) for a task (RDD stage, Pandas UDF).
3. In the task code, retrieve the logical indices of assigned GPUs and use them.

User Scenarios

Story 1 - Describe GPU resources

(P0) As an admin, I can configure Spark cluster managers to list available GPU resources on worker nodes. So Spark applications can request GPUs and use them in scheduling.

(P0) As an admin, I can expose GPUs to users using some name alias, e.g., “gpu”.

(P0) As an admin, I don’t expect any breaking changes in relevant Spark 3.0 settings, e.g., spark.task.cpus, or performance regression in scheduling if I don’t need GPU scheduling.

Story 2 - Auto discovery of GPU resources.

(P0) As an admin, I can let the workers automatically discover GPU resources and report them to Spark cluster managers.

NOTE:

- Auto discovery already exists in YARN/k8s.
- The features requires nvidia-smi. We might need to update Spark Jenkins for tests.

Story 3 - Start application w/ default GPUs per task.

(P0) As a user/developer, I can start a Spark application and optionally set default number of GPUs to use per task for my application. So I don’t need to request GPUs for each of my task.

(P1) I expect Spark cluster manager to wait for allocating valid resources or warn me if no existing resources (idle or not) can meet the requirement.

Story 4 - Retrieve and use assigned GPUs

(P0) As a user/developer, I can retrieve the logical indices of assigned GPUs at runtime and I know how to map them to the physical GPU devices to use for computation.

Story 5 - Request GPUs in RDD operations

(P1) As a user/developer, I can specify how many GPUs to use per task in an RDD operation, e.g., for DL training and inference. This overrides the default setting of GPUs per task.

[This doc is shared publicly.]

(P2) If no active executors have the requested number of GPUs, I should see an error.

(P1) If tasks in the same RDD stage require different GPU resources, the allocated resources per task should meet all requirements.

- NOTE: It means some tasks might receive more GPUs than requested.

(P1) I can see the requested resources and assigned device indices in Spark UI.

Story 6 - Request GPUs in Pandas UDF.

(P1) As a user/developer, I can specify how many GPUs to use in my Pandas UDF, e.g., for model inference.

Story 7 - Prefer GPUs but not required.

(P2) As a user/developer, I can specify how many GPUs I prefer to use, but I'm also okay with the task fallback to CPUs if GPUs are not available at runtime. So the worse job time is predictable assuming CPUs are always available.

Story 8 - Request CPU cores per task.

(P1) As a user/developer, I can specify how many CPU cores, if non-default, to use in my task. So I can do multi-thread data loading/preprocessing to match the speed of GPU computation.

Story 9 - Heterogeneous resources

(P2) As an admin, I can list GPUs of different types, e.g., k80, p100 on a cluster of heterogeneous nodes. So I can smoothly upgrade the cluster to use new generation of GPUs.

(P2) As a user/developer, I can specify which GPU type and how many to use for my application on a heterogeneous cluster.

- NOTE: Within one application, we expect the executors are homogeneous.

Story 10 - Placement at application level

(P0) As a user, I can request my Spark application to be launched on fewer nodes if possible to improve the inter-task throughput, e.g., for model training.

- NOTE: YARN supports affinity at request time.

(P1) As a user, I can request my Spark application to be launched on more nodes if possible to improve the total I/O bandwidth, e.g., for model inference.

[This doc is shared publicly.]

Story 11 - Placement at task level

(P2) As a user, I can request my tasks to be launched on fewer nodes if possible to improve the inter-task throughput, e..g, during model training.

- NOTE: This is useful when there are concurrent training tasks running within the same Spark application, e.g., parallel tuning. The feature should be considered together with barrier execution mode.

(P3) As a user, I can request my tasks to be launched on more nodes if possible to improve the total I/O bandwidth, e.g., during model inference.

- NOTE: This is useful for streaming model inference where we have long-running tasks loading data from an external streaming source.

Story 12 - GPU resource isolation

(P0) As an admin, I can configure the cluster manager such that a Spark executor process can only access the GPU resources allocated to it. GPU memory is allocated proportional to GPU cards. This helps prevent users oversubscribing the resources on the same host.

(P0) As a user, I know that I should only use the assigned GPUs and corresponding GPU memory in my task, while the task process might be able to access unassigned GPUs.

Story 13 - Monitor GPU utilization

(P0) As an admin, I can set up Ganglia to monitor GPU utilization on my cluster based on Spark documentation.

(P2) As an admin, I can monitor GPU utilization at application level. So I can notify users if they requested GPUs but they didn't use them efficiently. Users should be able to view GPU utilization at application level to help tune performance.

(P3) As a user, I can view GPU utilization in Ganglia for my task stage by matching the stage start/end times and the allocated GPUs. So I can fine tune its performance.

- NOTE: This requires a mapping from the logical index to the physical device in Ganglia.

(P3) As a user, I can view GPU utilization in Spark UI for my task stage. So I don't need to go to Ganglia and do extra work.

- NOTE: If we have resources to do it, we should do it for CPU first.

Story 14 - YARN support

(See Appendix for detailed notes on YARN support.)

[This doc is shared publicly.]

(P0) As an admin, I need to upgrade YARN to 3.1.2+ to use GPU support, details here.

- NOTE: This means we will discuss whether Spark 3.0 can depend on YARN 3.1.2.
- NOTE: YARN 3.1.2 supports the following that match our stories above:
    - Auto discovery of GPU resources.
    - GPU isolation at process level.
    - Placement constraints.
    - Heterogeneous device types via node labels.

(P0) As a user, I can request GPU resources in my Spark application via spark-submit.

(P1) As a user, I can use YARN+Docker support to launch my Spark application w/ GPU resources. So I can easily define the DL environment in my Dockerfile.

- NOTE: YARN 3.1.2 also supports docker containers.

(P1) As a user, I can use YARN dynamic allocation in my Spark application, where new containers allocated should have the GPU resources as initially requested.

Story 15 - Kubernetes support

(See Appendix for detailed notes on K8s support.)

(P0) As an admin, I can prepare the GPU nodes for a K8s cluster, details here.

- NOTE: K8s already supports the following that match our stories above:
    - Auto discovery of GPU resources.
    - GPU isolation at executor pod level.
    - Placement constraints via node selectors.
    - Heterogeneous device types via node labels.

(P0) As a user, I can specify GPU requirements for my Spark application on K8s.

- NOTE: Which interface we expose is pending design discussion. Possible choices:
    - spark-submit w/ the same GPU configs used by standalone/YARN.
    - spark-submit w/ pod template (new feature for Spark 3.0).
    - Spark-submit w/ mutating webhook confs to modify pods at runtime.

(P0) As a user, I can run Spark/K8s jobs using nvidia-docker to access GPUs.

Story 16 - Testing framework

(P0) As a Spark developer, I can run Spark GPU tests on a local machine with GPUs installed and configured.

(P1) Spark Jenkins runs GPU tests as a nightly tests on machines with GPUs. So if the GPU scheduling feature is broken, a list of owners will receive notifications.

[This doc is shared publicly.]

- NOTE: It is hard to run GPU tests on every PR because it requires many GPU cards given Spark PR frequency, and more importantly, GPU scheduling on Jenkins.
- NOTE: If needed, NVIDIA can donate GPU cards to Berkeley to run Spark Jenkins.

(P2) As a Spark developer, I can manually trigger GPU tests for my PR. So I can confidently merge my PR that touches GPU scheduling features.

(P3) Spark Jenkins automatically triggers tests on GPU nodes if code changes might affect GPU scheduling.

Story 17 - FPGA support

(P3, TODO) This story is currently a placeholder.

[This doc is shared publicly.]

Appendix (some raw tech notes, okay to skip)

YARN support

GPU scheduling support will require YARN 3.1 or later (likely 3.1.2+). Details on how users can configure their YARN cluster for GPU scheduling is covered here. YARN supports automatic detection of GPUs or admins can explicitly specify the GPU devices exposed to YARN if desired.

When YARN containers for executors are requested, Spark will need to map the number of GPUs needed for an executor to “yarn.io/gpu” resources in the YARN container request. YARN has GPU isolation, either with or without Docker containers, so an executor will be prevented from using GPUs that were not assigned to the YARN container assuming the admin configured that properly.

(P1) For affinity/anti-affinity support, Spark can leverage YARN Placement Constraints to help YARN know Spark would like containers co-located or explicitly not co-located. However the YARN cluster will need to be configured to support this, as it is disabled by default.

(P2) For heterogeneous clusters where GPU types vary and the job requires a specific GPU, users can leverage YARN’s node labels to separate the node types and setup queues that will allocate containers only on those node types. Starting in Hadoop 3.2, YARN supports node attributes which could be used to tag the type of GPUs on a node, and those tags can be used with placement constraints on the request to ensure the proper node type is requested.

(P2) YARN+Docker support. YARN supports running application tasks in a Docker container if the cluster admins have configured the runtime to be allowed. Details are covered here. The primary appeal for Spark is providing a simpler story for providing dependencies, like ML libraries, specific Python version, etc. at standard installation paths like `/usr/bin` and `/usr/lib` instead of shoe-horning those dependencies into the YARN distributed cache as .tgz archives and modifying the application’s `PATH` and `LD_LIBRARY_PATH` to find them.

Docker containers can run with the application, log, and distributed cache directories bind-mounted into the container under the expected paths, so applications can decide which dependencies to ship via the distributed cache (e.g.: job configs, user jars, etc.) and which to ship via the Docker image (e.g.: Python, R, ML libs, etc.). Since the log directories are bind-mounted into the container, the application provides the same logs to YARN as a non-Docker task. By default the Docker container runs with the host network, so from a networking perspective it runs just like a non-Docker task. Bridge networking is supported, but it may not be stable in Hadoop 3.1.

[This doc is shared publicly.]

As YARN assigns GPUs to an executor, the selected GPUs will be exposed into the Docker container. The Spark executor will need to discover the assigned GPUs in the same manner as in the non-Docker case. From a GPU-scheduling perspective, YARN Docker support should be an orthogonal issue which is primarily the responsibility of the YARN cluster admin to configure. Spark should not need to explicitly support the feature outside of potentially providing reference Docker images for common Spark-on-YARN use cases as examples.

Kubernetes Support

Kubernetes uses device plugin model to support accelerator devices on a node. The link shown gives a list of currently available accelerator plugins in K8s including those from Nvidia, AMD, Intel. Each device plugin has its unique set of requirements to be satisfied while preparing the nodes. The steps needed to prepare nodes for the Nvidia GPU device plugin are listed here: including installing necessary GPU drivers and setting nvidia as the default run time for docker.

As a Spark+K8s user, I can use spark-submit to submit an application to the Kubernetes cluster. The link running Spark on Kubernetes describes how this works for non-GPU workloads. In particular,

- Upon spark-submit, Spark first creates a driver running within a Kubernetes pod.
- The driver then creates executors which are also running within Kubernetes pods and connects to them, and executes application code.
- When the application completes, the executor pods terminate and are cleaned up
- Driver pod persists logs and remains in “completed” state in the Kubernetes API until it’s eventually garbage collected or manually cleaned up.

Specify number of GPUs to use for a task (RDD stage, Pandas UDF) in K8s will be similar to Standalone and YARN modes: either global defaults from spark.task.gpus and spark.executor.gpus or per task overrides at RDD stage etc.

- In both cases: gpu, cpu, mem resources will be mapped to pod spec of tasks and executors
- For gpu resources, k8s enables them as an extended-resource, and can be specified under a “vendor-name/gpu” tag (scheduling GPUs page for details).

(P0) For auto discovery of GPU resources support, with the above components installed on nodes during preparation, GPUs on each node will be auto-discovered and conveyed to the K8s API server. One can run “kubectl describe node” commands to verify that the correct number of gpus are being reported for each node.

(P0) For GPU device isolation, in the task code, Spark should retrieve the indices of assigned GPUs and use them for computation. Since Kubernetes runs with cgroups, each executor will only see the gpu devices that are assigned to it. For Spark tasks running inside the executor, we’ll need to verify the same behavior.

[This doc is shared publicly.]

(P2) For locality and affinity support, partial affinity can be provided in Kubernetes via node selectors using labels for placement choices. Further locality and affinity will need topology information of racks etc which will be added later.

(P2) K8s+Docker support. Kubernetes orchestrates containers and supports some container runtimes including Docker. Spark (version 2.3+) ships with a dockerfile that can be used for this purpose and customized to specific application needs. With Kubernetes this is the primary mode of running Spark apps.

- The config vars *spark.kubernetes.container.{driver/executor/}.image* can be used to specify the image location in repository.
- For running GPU apps using Docker, additional components will be needed on the worker nodes: for e.g., nvidia-docker for running on Nvidia GPUs using docker.

Specifying **GPU type** in addition to number of gpus will be a P1 requirement. Since we only support homogeneous gpu types as P0, this is not needed for 3.0. This story will be required for supporting heterogeneous GPU types later.