Accelerating Spark Workloads using GPUs

Rajesh Bordawekar, Minsik Cho, Wei Tan, Benjamin Herta, Vladimir Zolotov, Alexei Lvov, Liana Fong, and David Kung

IBM T. J. Watson Research Center

Outline

- Spark Background
- Opportunities for GPUs in Spark
- Spark GPU Integration Issues
- Our Approach
- GPU-enabled Spark in use

What is Spark?

- An in-memory distributed computing infrastructure
  - Implemented in Scala, uses JVMs for execution
- Parallel computations encoded using a fundamental data structure, Resilient Distributed Dataset (RDD)
  - Work on RDD gets distributed as per RDD *partitions*
- Supports high-level APIs in Java, Scala, Python, and R
- Provides libraries for SQL (Spark SQL), Machine Learning (MLlib/ML), Graph Analytics (GraphX), and Streaming (Spark Streaming)
- Supports data transfer to/from file systems such as HDFS

Spark Programming Model: RDDs

- A immutable distributed in-memory collection of elements
- Distributed-shared memory view of the cluster environment
- Computations on RDDs parallelized by default
  - Split into multiple partitions (each partition -> data subset)
  - Embarrassingly parallel execution on individual partitions
- Base type for other specialized data structures: Key-Value Pairs, Data Frames, Distributed Matrices, DStream, Triplets.
- RDD operations
  - Element-wise transformations from one RDD to another
  - Actions to compute results (actions do not generate RDDs)
  - Transformations triggered by actions in a lazy manner
    - Data “pulled” and “transformed” by actions

Spark Execution Flow

- Four core components
  - RDDs: Parallel Data Collections with Partitions
    - Moving towards DataFrames that are built over RDDs
  - DAG:
    - Logical graph of RDD operations of the entire program connecting different stages
  - Stages:
    - Each stage is a set of tasks that run in parallel
    - Ordering between different stages
  - Tasks:
    - Fundamental units of works; decided by the RDD partitions

Spark Execution Model: Drivers and Executors

- Spark application: A driver and multiple executors
- Overall execution split into stages, each with potentially different number of partitions
  - Data needs to be *shuffled* to create new partitions
- A Spark Driver invokes Executors to execute operations on the RDDs in embarrassingly-parallel manner
  - Each executor can use multiple threads
  - Transformations are element-wise data parallel over elements in a partition
  - Actions are task-parallel, one job per partition
- Spark application invoked by an external service called a cluster manager which uses one of the following schedulers
  - Spark Standalone, YARN, Mesos,..

Spark Memory Management

- The driver runs its own Java process and each executor is a separate Java process
- Executor memory is used for following tasks
  - RDD partition storage for persisting or caching RDDs. Partitions are deleted in LRU manner under memory constraints
  - Intermediate data for shuffling
  - User code
- 60% allocated for RDD storage, 20% for shuffling, 20% for user code
- Default caching uses MEMORY_ONLY storage level. Use persist with MEMORY_AND_DISK storage level
- Spark can support OFF_HEAP memory for RDDs

GPU Opportunities in Spark

- Computationally intensive workloads
  - Machine Learning/Analytics kernels in native Spark codes
  - *Sparkifying* existing GPU-enabled workloads (e.g., Caffe)
- Memory-intensive in-memory workloads
  - GraphX (both matrix and traversal based algorithms)
  - Spark SQL (mainly OLAP/BI queries)
- Two approaches: Accelerate an entire kernel or a hotspot
- System implications
  - A few nodes with multiple GPUs can potentially out-perform a scale-out cluster with multiple CPU nodes
    - Reduce the size of the cluster
    - Inter-node communication replaced by inter-GPU communication within a node

GPU Execution and Deployment Issues

- GPU execution inherently hybrid
  - GPU kernel invoked by CPU host program
  - Multiple kernels can be concurrently invoked on the GPU
  - “push” functional execution managed by the CPU(s)
- GPU memory separate than the host memory
  - Usually much smaller than the host CPU system
  - Data needs to be explicitly copied to/from the device memory
  - No garbage collection, but memory region can be reused across multiple kernel invocations
- Spark is a homogeneous cluster system
  - Spark resource manager can not exploit GPUs

GPU Integration Issues: Executing Spark Partitions on GPUs

- A Spark partition is a basic unit of computation
- Mapping a partition on GPUs:
  - A kernel executing on a GPU
  - A single GPU
  - Multiple GPUs
- A Spark instance can use one of these mappings during its execution
  - Need a specialized hash function to reduce data shuffling
- Spark partition can hold data larger than a GPU device memory
  - Out-of-core execution or re-partitioning?

GPU Integration Issues: RDDs and Persistence

- Hybrid RDDs
  - RDD stored on the CPU, but stores data computed by the GPU
- Native GPU RDDs
  - RDDs created by GPU by transformations on hybrid RDDs
  - Data stored in device memory and not moved to the CPU
  - Native RDD have space limitations
- Actions can be implemented as GPU kernels
  - Operate on hybrid or native RDDs and return results to the CPU
  - Results of actions can be cached on the device memory
  - Any RDD operated by an GPU kernel must be (at least partially) materialized before GPU kernel execution
- GPU RDD Persistence
  - DEVICE_MEMORY
  - GPU device memory not garbage collected.

GPU Integration Issues: Supporting Spark Data Structures

- Spark uses a variety of data structures derived from RDDs
  - Data Frames, Key-Value Pairs, Triplets, Sparse and Dense matrices
- GPU performance depends on how data laid out in memory
  - Data may need to be shuffled to make it amenable for GPU acceleration
  - GPU-based RDDs can have specific memory layout options
    - Columnar RDD from IBM (Kandasamy and Ishizaki)
- Spark memory manager needs to be extended to enable GPU memory allocation and free

GPU Integration Issues: Clustering and resource Management

- Usually, the number of GPUs less than the available CPU virtual processors (== #nodes*SMT*#cores)
- Spark's view of GPU resources
  - Access restricted to CPUs of the host node?
  - All nodes can access any GPU?
- Visibility to the Spark Cluster Manager
  - Number of threads used in a GPU kernel is usually very large
  - How does cluster manager assign executors to the GPUs (related to partition definition)
- Integration into Spark resource manager necessary

Spark GPU Integration: Three Key Approaches

- Use GPUs for accelerating Spark Libraries and operations without changing interfaces and underlying programming model. (Our approach)
- Automatically generate CUDA code from the source Spark Java code (K. Ishizaki, Thur 10 am, S6346)
- Integrating Spark with a GPU-enabled system (e.g., Spark integrated with Caffe)

Spark GPU Integration: Our Approach

- *Transparent* exploitation of GPUs without modifying existing Spark interfaces
  - Current Spark codes should be able to exploit GPUs *without any user code change*
  - Only need to update the Spark library being linked
  - Code runs using CPUs on nodes that do not have GPUs
- Focus on accelerating entire kernels
- Supports multiple node, multiple GPU execution
- Support for out-of-core GPU computations
- Initial focus on Machine learning kernels in Spark MLlib and ML directories

Spark GPU Integration: Our Assumptions

- A Spark partition covers single GPU (no concurrent execution of partitions)
  - A GPU kernel will run over only one GPU
  - Spark partitions from an executor mapped to different GPUs in a round-robin manner
- Native GPU RDDs have default DEVICE_MEMORY persistence
- RDDs can not be larger than the device memory
- Large datasets handled by using more smaller partitions
- GPU host memory will be allocated in a Java Heap
- GPU kernels use both cublas/cusparse and native code
- Support for both RDDs and DataFrames

Spark GPU Integration: Implementation Details

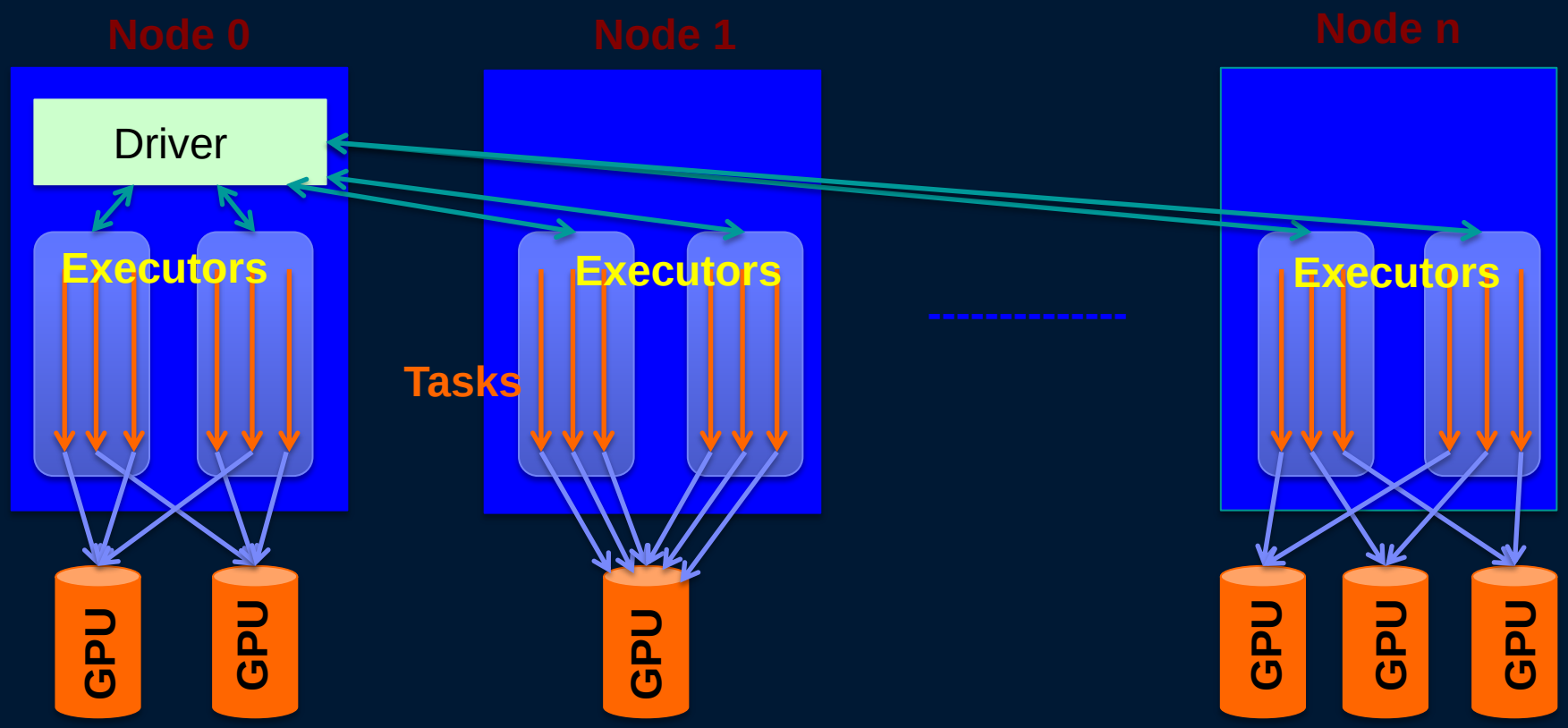
- Scala MLlib kernels modified without changing their interfaces
- Implementation supports multiple executors, each with multiple threads (each executor maps to a JVM)
- For each partition, data copied from Java heap to GPU device memory
  - GPU memory allocator uses CPU-based managed memory if GPU device memory allocation fails
- The GPUs are accessible to only one node and to the executors running on that node
- Partitions from different executors are mapped independently and using round-robin fashion
- Users turn on a Spark system variable to use GPU libraries

Spark Machine Learning Algorithms being accelerated

- Logistic Regression using LBFGS
- Logistic Regression Model Prediction
- Alternative Least Squares (W. Tan, S6211, Thur. 3.30 pm)
- ADMM using LBFGS
- Factorization Methods
- Elastic Net
- Word2Vec
- Nearest Neighbor using LSH and Superbits
- NNMF and PCA
- Investigating Deep Learning training within Spark

GPU-accelerated MLlib Kernel: ADMM

- Input data partitioned across executors using RDDs
- Each thread within an executor invokes a LBFGS solver
- The intermediate data is communicated to the driver for aggregation
- Each thread invokes a GPU kernel to implement the solver



Spark-GPU Integration: Some Observations

- GPUs are able to accelerate core kernels with substantial speedups over original code (e.g., 30X for Logistic Regression)
- End-to-end performance gain depends on performance of Spark functions
  - Performance of the LR affected by the costs of collating data (i.e., toArray()) in the Spark driver
- Effective mapping of Spark partitions on multiple GPUs is non-trivial
  - Can we coalesce partitions for reducing GPU calls?
- Managing large datasets from Java heap is not ideal
  - Data needs to be pinned, impacts GC,..
  - Off-heap memory exploitation should become more usable

Questions?

GPU-Accelerated MLlib code to released in open source soon.