Scaling Up Data-Parallel Analytics Platforms: Linear Algebraic Operation Cases

Luna Xu*, Seung-Hwan Lim†, Min Li‡, Ali R. Butt*, Ramakrishnan Kannan†
*Virginia Tech, †Oak Ridge National Laboratory, ‡IBM Almaden Research
*{xuluna, butta}@cs.vt.edu, †{lims1, kannanr}@ornl.gov, ‡minli@us.ibm.com

***Abstract*—Linear algebraic operations such as matrix manipulations form the kernel of many machine learning and other crucial algorithms. Scaling up as well as scaling out such algorithms are key to supporting large scale data analysis that require efficient processing over millions of data samples. To this end, we present, ARION, a hardware acceleration based approach for scaling-up individual tasks of Spark, a popular data-parallel analytics platform. We support both linear algebraic operations of between two dense matrices, and between sparse and dense matrices in distributed environments. ARION provides a flexible control of acceleration according to matrix density, along with efficient scheduling based on runtime resource utilization. We demonstrate the benefit of our approach for general matrix multiplication operations over large matrices with up to four billion elements by using Gramian matrix computation that is commonly used in machine learning. Experiments show that our approach achieves more than $2\times$ and $1.5\times$ end-to-end performance speed-ups for dense and sparse matrices, respectively, and up to $57.04\times$ faster computation compared to MLlib, a state of the art Spark-based implementation.**

I. INTRODUCTION

High-dimensional data is commonplace both in science [1], [2] and enterprise [3] applications. In order to discover patterns and relationships in such data, machine learning (ML) is widely used, typically through a core set of linear algebraic operations, called the analysis kernel [4]. The fundamental need of scalable data analysis, thus, is the ability to process a large number of data samples timely, i.e., to have a high analysis kernel computing throughput for matrix operations.

Extant practices for fast analysis kernel computing fall into two broad groups: (1) enabling scaling out on commodity hardware via the use of data parallel computation software platforms such as MPI [5], Hadoop[6], and Apache Spark [7], [8], [9]; and (2) enabling scaling up the computational power of individual nodes/machines for data analysis [10], [11]. Scaling up approaches have been proven to be beneficial to data analysis algorithms, since the approaches entail optimization algorithms [4] that can exploit hardware accelerators, e.g., GPUs, ASICs, FPGAs, and specialized CPU instructions [12] for matrix operations [13], [14], [15].

However, advantages from both scale-out and scale-up techniques come at a price to achieve higher performance. Scale-out options can suffer communication networking overheads and higher costs (energy, labor, machine failures, etc.) as the size of a cluster grows. Due to the constantly growing data acquisition throughput, such overheads lead to a constant pressure on scale-out options. On the other hand, scale-up options face a hard-wall of scalability, limited by the resources available in a single machine. Hence, *how best to combine and reconcile the scale-out and scale-up approaches* has become an important topic in processing matrix operations at scale.

Although it is promising to combining scale-out and scale-up techniques, it is non-trivial to realize the full potential of both approaches, due to the fundamental design choices made under each approach. Scale-out approaches, such as Spark, often focus on scalability, fault tolerance, cluster utilization, and workload balancing, but do not typically factor in techniques for improving an individual node's performance via hardware acceleration to support a wide range of distributed environments. For instance, Spark maps a job to a number of tasks that can later be run on any participating nodes without any regard to the unique hardware capabilities of the nodes or the ability of a task to leverage the available capabilities.

Let us look into the details of the challenges in exploiting node-local accelerators for matrix operations in the context of Spark. Spark MLlib [16], a widely used machine learning library built on top of Spark, often applies an ad-hoc Scala implementation for performing matrix operations without regards to node-local accelerators. Thus, users have to understand and implement system-level details if they want to scale-up Spark tasks. Moreover, scale-up approaches are not designed to consider the wide variance of tasks within Spark applications.

Scale-up solutions focus on parallelism at a fine granularity, such as SIMD and SIMT models, but with limited consideration of distributed environments. Each scale-up solutions for matrix operations, also known as Basic Linear Algebra Subroutine (BLAS) [17] operations, typically target specialized operations for the specific target hardware acceleration,

This work is sponsored in part by the NSF under the grants: CNS-1565314, CNS-1405697, and CNS-1615411. The manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan). This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

with minimal consideration, if any, on distributing tasks across systems. For example, OpenBLAS [18] is optimized for BLAS operations for dense matrices on CPU; cuBLAS [19] is optimized for BLAS operations on GPUs; and spBLAS [20] is optimized for BLAS operations between a sparse matrix and a dense matrix. It is because the maximal performance for each BLAS operation under each hardware configuration can significantly vary according to the density and size of given matrices. Hence, it is challenging to integrate existing scale-up solutions to the Map-Reduce programming paradigm, where the same operation is to be applied over each partition (or element) in the data set, through a single run-time solution when different tasks form a directed acyclic graph.

Therefore, we propose ARION, a dynamic hardware acceleration framework atop Spark. ARION is carefully designed for scaling up BLAS operations in a MapReduce like scale-out environment where multiple map tasks are concurrently running. ARION supports multiple hardware accelerated solutions for BLAS operations on both sparse and dense matrices. In addition, ARION treats available GPUs as additional cluster slots to run tasks. Thus, we allow to run tasks on both GPUs and CPUs on the condition that the overheads to schedule tasks on each processing unit does not exceed the benefits. The ability to leverage multiple hardware resources increases the overall system utilization. ARION bridges the mismatch of the design between local accelerators and scale-out platforms by utilizing techniques such as stream processing and kernel batching. More specifically:

- We extend our previous work [21] by designing and implementing ARION, a dynamic hardware acceleration framework atop Spark, and effectively combine scale-up hardware acceleration for distributed linear algebraic operations with scale-out capabilities of Spark.
- We design distributed matrix manipulation support for both dense and sparse matrices in Spark for scale-out matrix operations. In contrast, the current state of the art, e.g., MLlib [16], considers only sparse matrices in a distributed setting.
- We design a dynamic algorithm that chooses the best hardware accelerator at runtime by considering key factors such as matrix size, density, and system utilization. Our approach does not require changes to the Spark applications on user side, instead it leverages already-available libraries to support the target operations.
- We design a dedicated GPU component along with the BLAS libraries to support concurrent usage of both CPU and GPU to increase overall system utilization. The GPU component is optimized to maximize the GPU utilization by streaming processing and kernel batching techniques.

On top of our design and implementation, for computing Gram matrix (XX^T [22]), ARION shows more than $2\times$ and $1.5\times$ speed-up for end-to-end performance for dense and sparse matrices, respectively. Most notably, for dense matrices, ARION achieves $57.04\times$ of speed-up in the computation time. Finally, ARION performs faster than or comparable to a larger scale setup with default Spark.

II. BACKGROUND

In abstract, most of machine learning algorithms can be understood as the operations between a weight matrix that represents the training model and feature vectors that represents the data samples. Thus, linear algebra operations forms the foundation of machine learning algorithms, making MLlib, the machine learning package of Spark, the default package for distributed linear algebra operations in Spark. MLlib wraps the lower level Spark Resilient Distributed Dataset (RDD) generation and operations needed to perform matrix operations and machine learning algorithms. A common assumption across components in MLlib is that a matrix will be mostly likely to be sparse, tall, and thin [23]. Assuming each row vector of a matrix will fit into a reasonable size of machines, most of linear algebra operations in MLlib does not support fully distributed linear algebra operations, along with limited support for dense matrix operations, except *BlockMatrix*.

Such a design choice leads to challenges in accelerating linear algebra operations in Spark. Assuming sparse matrix, MLlib heavily exploits sparse matrix representation to compactly store non-zero terms for the sake of efficiency. A consensus on accelerating BLAS operations is that GPUs are typically less efficient in performing sparse matrix operations than dense matrix operations [13]. Thus, in most cases, to support sparse operations, a user inflates the sparse matrices into a dense one and offloads the processing to the accelerators. Otherwise, accelerator inflates internally. In addition to data representation, Spark partitions the matrix into concurrent multiple tasks per node, assigning in a relatively smaller chunk to each task (e.g., default partition size for *BlockMatrix* is $1K \times 1K$). The execution of each task often forms multiple waves, resulting in a limited number of concurrent tasks per node. As a result, a small number of concurrent matrix operations arrive at GPU at the unit of small chunks, leading to a low utilization of GPU. Accordingly, a straightforward drop-in solution of using BLAS libraries from Spark MLlib, including *BlockMatrix* class, cannot guarantee a full utilization of hardware capabilities.

III. DESIGN

This section describes the design overview of ARION, along with detailed explanation on our design choices.

A. System architecture

Figure 1 illustrates the architecture of ARION and its key components that all run in Spark executors on participating worker nodes. **Selector** is responsible for selecting the optimal processing variant such as BLAS and GPU implementation (**GPU Impl**) based on matrix characteristics. **Selector** addresses the problem of the standard MLlib always using the Scala implementation for multiplications in *BlockMatrix*. We employ MLlib to partition the input datasets per given block size, and generate RDDs and associated tasks, which are then send to the executor. **Selector** calculates the density of

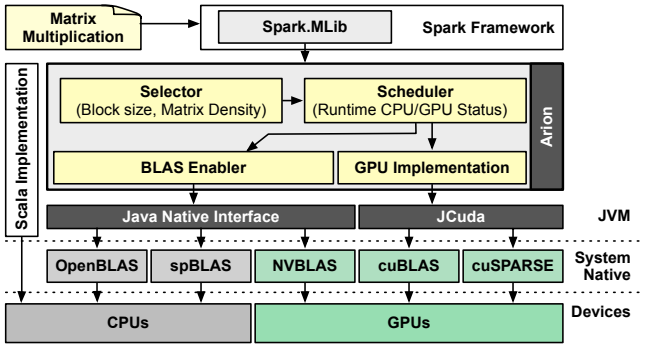


Fig. 1: System architecture of ARION.

assigned sub-matrices (this also catches skewed density cases that can be missed if examining a matrix as a whole), retrieves the block size information, and decides which processing variant to use to perform the multiplication. The tasks are then submitted to the **Scheduler**.

Our **Scheduler** aims for high resource utilization of both CPUs and GPUs. Thus, tasks are scheduled based on the system runtime utilization using the processing variant chosen by the **Selector** with a high priority. **Scheduler** works with node-local hardware accelerators, which can help the case of a heterogeneous cluster where each node has different hardware configurations. Moreover, to increase resource utilization, **Scheduler** negotiates with Spark task scheduling system to dynamically allocate more tasks if hardware resources become idle. This is in contrast to the current Spark process, which only considers CPU core resource.

B. Hardware acceleration for dense and spare matrices

Table I lists the processing variants supported in ARION. The ad-hoc Scala implementation is preserved in our system as one of the options. For others, we proceed as follows. Given the block size, the matrices are partitioned into block sub-matrices. Each task performs the multiplication of two block sub-matrices. ARION supports multiplications of both ddGEMM and spGEMM. As shown in Figure 1, ARION enables hardware acceleration via two channels: **BLAS Enabler** and **GPU Impl**. **BLAS Enabler** is able to adopt underlying hardware optimized BLAS libraries through a Java native interface (JNI). Currently we adopt OpenBLAS [18], NVBLAS [24], and spBLAS [20]. Here, OpenBLAS and NVBLAS support acceleration for ddGEMM on CPU and GPU, respectively. Both are allowed in MLlib with the integrated Netlib-Java [25] as the JNI layer. To support hardware accelerated sdGEMM, we adopt the spBLAS library under **BLAS Enabler**. For this purpose, we extend Netlib-Java to support linking to the spBLAS library so that it is transparent to MLlib. Note that our design of **BLAS Enabler** is flexible and support plugging in of any external libraries via JNI. While detailed discussion is out of the scope of this paper, we do not prevent the adoption of SpGEMM (sparse-sparse matrix multiplication) libraries without hardware acceleration. We also implement a dedicated GPU matrix multiplication component,

TABLE I: Summary of all processing variants supported in ARION. Methods on "GPU (direct)" platform can co-execute with methods running on "CPU" platform.

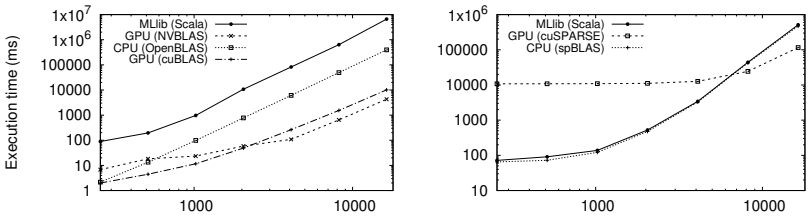
Methods	Matrix type	Platform
OpenBLAS	Dense-dense	CPU
NVBLAS	Dense-dense	GPU
cuBLAS (GPU Impl)	Dense-dense	GPU (direct)
spBLAS	Sparse-dense	CPU
cuSPARSE (GPU Impl)	Sparse-dense	GPU (direct)

GPU Impl, with GPU runtime dense matrix operation libraries such as cuBLAS [19] and spark matrix operation libraries such as cuSPARSE, which bypasses initializing BLAS libraries through JNI. Thus, ARION supports co-execution using both CPU and GPU, i.e., using **GPU Impl** without JNI BLAS library linking, while using CPU BLAS libraries through JNI. This can yield more effective scale-up by leveraging both CPU and GPU accelerator resources.

C. Choice of processing variants

Once **Selector** determines the matrix type of a given multiplication task, it recommends the methods to **Scheduler**, which are then given priority for processing the matrix. Note that the **Scheduler** can also ignore the recommendation if it conflicts with resource availability and utilization. To suggest the right option for performing multiplication on a given environment, we adopt a user configurable system parameter Th_{block} (size of a block matrix) to decide the best compute variants, given environments and workloads. We choose block size as a parameter since other parameters such as density can skew in a matrix, and with a fixed block size, we can calculate the density of each block. Note that the Th_{block} is for both dense and sparse matrix.

Let us show our approach to decide Th_{block} used in our evaluation, which also can highlight the differences between the three supported methods. We first performed ddGEMM with both NVBLAS and **GPU Impl** using cuBLAS as the two methods for GPU acceleration, and OpenBLAS as CPU acceleration on a GPU node in $Rhea$ [26] (Table II shows the specifications). For simplicity, we measured the end-to-end computation time for the multiplication of two matrices of the same size. In this measurement, we included the data transfer time between the host and GPU into the computation time since this time is included in the computation time of tasks in Spark. We used block sizes within the range of 256×256 to $16K \times 16K$ in the experiment, which is within the reasonable range for Spark. The default block size in Spark is $1K \times 1K$. Figure 2(a) shows the execution time using different options with different block sizes. The reported execution time is an average of five runs for each block size. For small block sizes up to $4K \times 4K$, we observe that **GPU Impl** performs better than the other two. As the block size increases, NVBLAS starts to outperform **GPU Impl** due to NVBLAS design (§ III-D). OpenBLAS performs reasonably within the block size of $1K \times 1K$, but the performance degrades dramatically as the size grows more than $2K \times 2K$, when OpenBLAS starts to perform worse than NVBLAS. Based on this observation, we identify



(a) Dense-dense matrix.

(b) Sparse-dense matrix.

Fig. 2: Execution time of matrix multiplication using different implementations. X-axis represents b in a block matrix $b \times b$. Both axes are log-scaled.

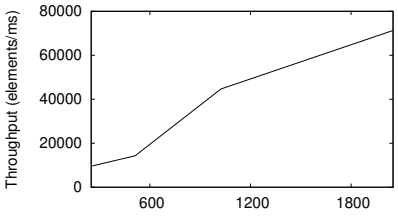


Fig. 3: Throughput of NVBLAS. X-axis represents b in a block matrix $b \times b$.

Th_{block} for dense matrix in our system to be $2K \times 2K$ (the cross point of the NVBLAS and **GPU Impl**). In our system, **Selector** would either choose NVBLAS or **GPU Impl** based on the threshold as they outperform OpenBLAS in any cases.

Next, we repeat the experiment with a dense matrix and a sparse matrix with density of 0.05. We recorded the end-to-end time of multiplication with **GPU Impl** using cuSPARSE, and CPU acceleration using spBLAS. As shown in Figure 2(b), GPU does not accelerate the calculation until the block size reaches $8K \times 8K$. Thus, we chose the Th_{block} to be $8K \times 8K$ for sparse-dense matrix multiplications. The thresholds are configurable by the users, which they can set based on their specific environment profiling. The values remain unchanged if the environment does not change. The default in our system is set based on our experiments above. Moreover, we found in our experiment that there is an upper limit size for a single GPU bounded by the GPU memory size, $32K \times 32K$ in our case, thus Th_{block} should be below this limit. In practice, a user's environment may have multiple decision points, for which we use a configuration file. The file is used by the **Selector** after it has determined the computation type (ddGEMM or sdGEMM) to select appropriate processing variants.

Discussion: To show the impact of hardware acceleration, we also study the performance of the Scala implementation of MLlib as our baseline case for our studied scenarios. We can see that for ddGEMM, hardware acceleration significantly speeds up the execution time. However, spBLAS performs slightly better or similar to MLlib. This is because ddGEMM is more computation intensive than multiplication with highly sparse matrices, thus ddGEMM offers more opportunity for performance improvement.

D. Improving system utilization

Our design aims to improve resource utilization with the goal to maximize performance and efficiency. We achieve this via a simple, yet effective, scheduling algorithm. If a GPU has to be employed, we also prevent node-level waves of tasks. This is because the number of tasks assigned per node can be much larger than the concurrency limit of the available GPU. We propose to batch task executions to improve the concurrency efficiency of GPUs.

1) Task scheduling: The goal of node-level task scheduling is to increase resource utilization whenever possible. First, **Scheduler** over-provisions the task slots and enables tasks to run on both GPUs and CPUs simultaneously, instead of just utilizing one type of resource at a time. Note that the data parallel design of Spark guarantees the data independency among Spark tasks so there is no data communication between CPU tasks and GPU tasks. A side effect of this approach can be that tasks are scheduled on suboptimal resources. To remedy such problems, **Scheduler** speculatively executes lagging tasks on available resources.

To over-provision the resources, we assign the task slots n_{slots} with the number of CPU cores plus the GPU capacity, i.e., the maximum number of block sub-matrices the GPUs attached to a node can hold. For example, in our experimental setup described in § III-C, $n_{slots} = 56$ (CPU cores)+32 (GPU streams [19] of 2 GPUs per node). Among the n_{slots} task slots, 56 are for the CPUs and the others for the GPUs. This allows ARION to request more tasks per node from Spark DAG scheduler to execute concurrently.

The scheduling algorithm works as follows. Given the processing option, and the occupied GPU and CPU slots, **Scheduler** obtains the resource type used by the processing variant as determined by **Selector**. If the resource type is GPU and all the GPUs are currently occupied, **Scheduler** changes the current processing variant to the method for the same matrix type in CPU platform shown in Table I to use the available CPU slots, i. e., OpenBLAS for ddGEMM and spBLAS for sdGEMM. On the other hand, if the resource type is CPU and all CPU slots are occupied, then depending on the block size, **Scheduler** changes the processing option to NVBLAS or to cuBLAS for ddGEMM. **Scheduler** simply changes to cuSPARSE for spGEMM. Here, to support running tasks on both OpenBLAS and NVBLAS simultaneously, we preload and link the NVBLAS library as the BLAS library, and automatically modify NVBLAS configuration to redirect the BLAS operation to OpenBLAS. Similarly, we modify the configuration back to NVBLAS when necessary. Note that this configuration switch requires each thread to maintain a thread local configuration file to avoid conflicts. In practice, the block size is usually small due to the limitations mentioned in § III-C. As a result, ARION opts for cuBLAS over NVBLAS, and thus is able to avoid this switching and initiation overhead. Furthermore, recall that we extended Netlib-Java to support spBLAS (§ III-B). Our extension does not interfere with the existing OpenBLAS and NVBLAS libraries, instead we internally load spBLAS library from our extended Java code without the need of preloading as long as the spBLAS library can be loaded in the system library path.

Scheduler respects the variant choice of **Selector** as long as appropriate resources are available, choosing an alternate only if needed resources are occupied. However, in cases with few number of waves of tasks, executing on the suboptimal alternate can lead to suboptimal task execution time, which in turn exacerbates the performance of entire stage execution. For instance, for an application that launches 88 tasks per node as a single wave requesting cuBLAS, **Scheduler** may change the processing option of 56 tasks to OpenBLAS and executes them on CPUs. Since GPUs run much faster for dense matrices, the other tasks executed on CPUs become stragglers. Thus, **Scheduler** speculatively re-launches the tasks on idle GPUs and terminates the stage when all the results are available. Consequently, ARION is able to mitigate any suboptimal **Selector** decisions.

2) GPU optimization: ARION adopts a number of optimization techniques for **GPU Impl** to ensure that all of the GPUs within a node are fully utilized.

We first discuss the performance inefficiency of the tiling technique of NVBLAS. NVBLAS uses the cuBLAS-XT [19] API under the hood. cuBLAS-XT supports a multi-GPU enabled host interface. In cuBLAS-XT, matrices are divided into configurable square tiles of a fixed dimension, called *BlockDim*, with a default value of $2K$. Each tile is assigned to an attached GPU device in a round-robin fashion, and one CPU thread is associated with one GPU device for transferring data. This enables NVBLAS to pipeline tile transfer and computation, thus improving the overall GPU throughput.

While NVBLAS outperforms other processing variants by tile splitting when matrices are large, Spark tasks work on RDD partitions where each sub-matrix is usually not big enough to be further split into square tiles of size $2K$. Usually GPU throughput is low with small sizes of matrices. To verify this, we tested GPU throughput of matrix multiplication with different sizes. As shown in Figure 3, GPU throughput drops exponentially as the size of matrix decreases.

Another problem is that NVBLAS handles one matrix multiplication at a time, with multiple tiles of the matrices handled concurrently. This concurrency model is different from Spark concurrency model in which each task performs one sub-matrix multiplication, and multiple sub-matrix multiplications are performed by different tasks at the same time. It is equally important to optimize the performance of using GPU for small matrices. However, NVBLAS is inefficient in achieving high concurrency for small matrices multiplications. Thus, we propose to batch small matrices into a large one to improve the concurrency and utilization of GPUs in our **GPU Impl** design (for both cuBLAS instance and cuSPARSE instance). In particular, we adopt the CUDA Streams [19] technology to overlap computation of different tasks by batching the execution of small kernels. We associate one Spark task with one separate CUDA stream to make the GPU compute the tasks concurrently. This effectively batch multiple small matrices into a large matrix to run concurrently, consequently improving the GPUs' throughput.

While it is possible to have as many streams as needed for a single GPU, the CUDA community suggests to not have more than 16 concurrent kernels per GPU [19]. We further limit the degree of concurrency based on the GPU memory capacity, such that a GPU can hold no more than $GPU_{ram}/b^2 bytes/3$ streams concurrently, where GPU_{ram} represents the GPU memory capacity. This is to accommodate the two $b \times b$ input matrices and the result matrix. Although sparse matrices can be represented with compressed formats, we keep this limitation without loss of generality (e.g., GPU is used for a mixture of multiplication tasks including ddGEMM and sdGEMM). Using multiple streams may cause higher data transfer overhead between the CPU and GPUs. In this case, we arrange a fixed number of concurrent streams for each GPU, and once the streams in one GPU are all occupied, we dispatch the next batch tasks to the next GPU in a round robin fashion. Therefore, we can efficiently utilize multiple GPUs to overlap the data transfer and accelerate Spark tasks for small matrices with high concurrency.

TABLE II: System specification.

System name	Rhea GPU node	Rhea CPU node
CPU model	dual Xeon E5-2695	dual Xeon E5-2650
CPU cores	14×2 (28×2 HT)	16×2 (32×2 HT)
CPU memory	1TB	128GB
GPU model	dual NVIDIA K80	N/A
GPU (CUDA) cores	4992×2	N/A
GPU memory	24×2 GB	N/A
CUDA ver.	7.5	N/A
Network Interface	1G Ethernet	1G Ethernet

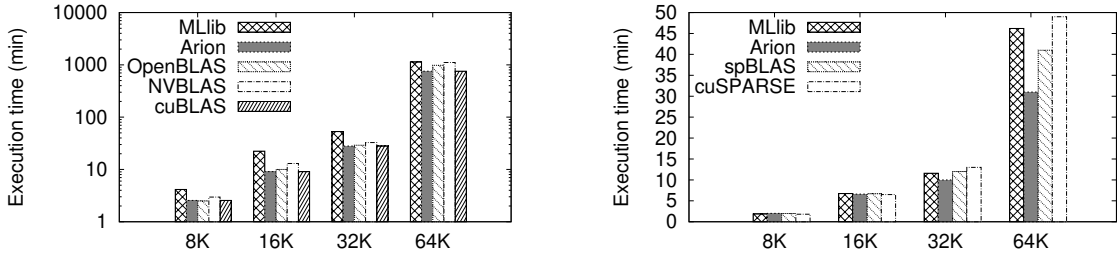
TABLE III: Studied matrix sizes, densities, and raw file size.

Matrix	8K	16K	32K	64K
Density=1.00	0.98 GB	3.93 GB	15.73 GB	80.82 GB
Density=0.05	0.30 GB	1.19 GB	4.77 GB	19.08 GB

IV. EVALUATION

We have integrated ARION in Spark by modifying MLlib (mainly small changes to *BlockMatrix* class). However, ARION can also be integrated with other scale-out platforms. In this section, we evaluate ARION's ability to efficiently accelerate matrix multiplication in Spark compared to vanilla MLlib implementation. To this end, we use a large scale Gramian matrix ($\mathbf{XX^T}$) computation kernel. This kernel is common in ML algorithms such as SVD and PCA [4]. Gramian matrix computation also plays a critical role in popular data analysis techniques, e.g., all-pair similarity [22].

We use Spark 1.6.1, the latest version at the time of this work. We manually compiled OpenBLAS and spBLAS library version 0.2.19 with acceleration instruction flags set including AVX2 and FMA3 [27]. We use the default NVBLAS library included in CUDA toolkit 7.5. We implemented **GPU Impl** with cuBLAS and cuSPARSE APIs from the CUDA toolkit. Our experiments are conducted on six highly-scalable GPU nodes assigned from a super computing cluster, $Rhea$, as described in Table II. We configure Spark with one master and six worker nodes. One worker is co-located with the master. Each worker node runs one executor. We configure each



(a) Dense matrix.

(b) Sparse matrix.

Fig. 4: End-to-end execution time with default Spark MLlib, ARION, and the naive adoption of other processing variants. (Note log-scaled Y-axis in (a).)

executor to have $800\ GB$ memory and 56 cores. We repeat each experiment five times and report the average results.

A. Overall performance

In our first experiment, we study the overall performance gain obtained from ARION. For this purpose, we generate random square matrices of different orders with uniform densities as shown in Table III. We store both dense and sparse matrices in text files with same format to maintain the same data preprocessing process of Spark. We use density 1.0 for dense matrices and 0.05 for highly sparse matrices. We see that our raw input file sizes scale up to $80\ GB$ with computation of up to 4.3 billion data points. We perform generalized matrix multiplication from an input matrix, $\mathbf{X}$, to compute Gramian Matrix, $\mathbf{XX^T}$. Due to the nature of the computation of Gramian Matrix, the pair of matrices of multiplication would be either both dense or both sparse. However, ARION may still employ acceleration of sparse-dense matrix multiplication on block sub-matrices where there may be skewness within the matrix. To study the performance of the different processing variants, we consider several scenarios: (i) default Spark MLlib, ARION for both dense and sparse matrix $\mathbf{X}$; (ii) ARION using OpenBLAS only (with **Selector** and **Scheduler** disabled), ARION using NVBLAS only, ARION using **GPU Impl** with cuBLAS only for dense matrix $\mathbf{X}$; (iii) ARION using spBLAS only and **GPU Impl** with cuSPARSE only for sparse matrix $\mathbf{X}$. We use the default block size of $1K \times 1K$ for our experiments.

We record the end-to-end execution time of the application, which includes reading input data file, creating a *BlockMatrix* matrix type from the input file through intermediate data types (from *RowMatrix* to *IndexedRowMatrix* and *CoordinateMatrix*), and performing matrix multiplication on *BlockMatrix*. Figure 4 shows the results. The x-axis represents execution time against the order of matrices as shown in Table III.

We first compare the performance of ARION to default Spark MLlib. Here, we observe that ARION performs better than MLlib for dense matrices with an average speed-up of $1.87\times$, and up to $2.47\times$. For sparse matrices, ARION speeds up performance by $1.15\times$ on average and up to $1.5\times$ for $64K \times 64K$ matrix. This performance improvement mainly comes from the hardware acceleration for performing matrix multiplications (further investigated in § IV-B. The performance of ARION over MLlib is higher in the case of dense matrices (y-axis is only log scaled in the case of dense matrix) due to the fact that the computation of dense matrices is more intensive than the case of sparse matrices, and dense matrices can benefit more from the GPU computation power.

Next, we analyze the impact of different implementation variants. From the figure, we see that for dense matrices, all hardware acceleration methods (OpenBLAS, NVBLAS, cuBLAS) finish earlier than MLlib. For sparse matrices, hardware acceleration methods (spBLAS, cuSPARSE) perform similar to, or only slightly better than MLlib. This is because for highly sparse matrices with 5% of non-zero elements, the overhead of random memory access outweights the benefit of GPU acceleration where the overhead is incurred by storing non zero entries non-sequentially in memory due to the sparsity of the matrices. This observation is consistent with the results reported in [23]. Overall, ARION outperforms other implementation variants for both dense and sparse matrices. Among all matrices sizes, ARION increases the performance by 10.75% and 20.28%, compared to the naive use of OpenBLAS and NVBLAS, respectively. ARION performs similar to cuBLAS case because ARION selects cuBLAS for dense matrices in our system, and most tasks are scheduled to GPU over CPU due to the short task execution times. The performance improvement reaches up to 31.89% for dense matrices. Similar trend is observed in sparse matrix case where ARION further improves performance by 10.59% compared with cuSPARSE, and 10.13% compared with spBLAS. The main reason for this is that ARION dynamically chooses the optimal variant per block based on density and order of block sub-matrices and schedules tasks to otherwise idled compute resources, consequently increasing resource utilization significantly.

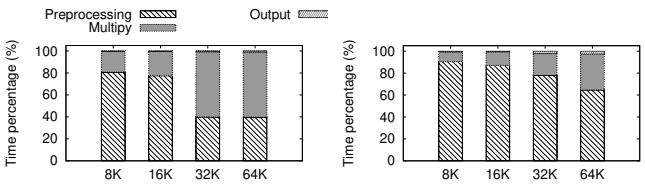
B. Performance breakdown

The end-to-end performance includes data preprocessing (reading input file and converting to *BlockMatrix*), matrix multiplication, and persisting results to storage. Figure 5 shows the breakdown of the end-to-end time for these three stages under ARION. The time fraction for the "Multiply" phase suggests how much time is spent on computation, which is the component where hardware acceleration can speed up. As seen in the figure, in general, dense matrices spend more time on multiplication than sparse matrices, thus dense matrices benefit more from hardware acceleration. Moreover, larger matrices spend more time on computation while smaller

matrices spend more time in data preprocessing. We thus conclude that hardware acceleration favors bigger and denser matrices. The benefit of hardware acceleration on computation tends to be discounted by large data preprocessing time for smaller and more sparse matrices.

In the "Multiply" stage, Spark tasks shuffle the block matrices to get the right sub-matrices, and then perform the computation. To further investigate the acceleration of computation, we eliminate the data shuffle phase, and record only the computation time. Note that we use aggregated compute time as a simplified metric due to the complexity of Spark task scheduling for this set of experiments. Figure 6 shows the speed-ups of aggregated compute time of ARION and other variants compared to MLlib. The observed speed-up for dense matrices increases with matrix size, and peaks at $57.04\times$ with the matrix of $64K \times 64K$. This is because ARION fully utilizes both CPUs and hardware accelerators. As the matrix grows, hardware accelerators demonstrate greater speed-ups. Using hardware accelerators improves the computation performance by up to 57 times. However, the overall performance decreases due to the data preprocessing and other framework overheads. For sparse matrices, we still see a slight speed-up (up to $1.5\times$) under ARION over the Scala implementation of MLlib, which is consistent with the overall results in Figure 2(a). Again, we observe that ARION performs similar to or better than other implementations for both dense and sparse matrices, similar to the overall performance. As the matrix size grows, the performance improvement of ARION and **GPU Impl** increases, up to 88.2% and 84.9% for dense and spare matrices, respectively, compared to BLAS libraries. We expect the performance improvement of ARION to increase with growing matrix sizes against just vanilla hardware acceleration.

Next, we compare the computation speed-ups of different implementation variants. Among the three hardware solutions for dense matrices (OpenBLAS, NVBLAS, and cuBLAS), cuBLAS outperforms the other two variants, which is consistent with results in Figure 2(a). Moreover, OpenBLAS slightly surpasses NVBLAS with an average improvement of 9.07%, which is different from the pure measurement of Figure 2(a). This is because the default block size of Spark falls out of the sweet spot of NVBLAS. As described in § III-D, NVBLAS is designed to opt for large matrices. However, Spark divides matrices to $1K \times 1K$ block sub-matrices, which hinders NVBLAS to benefit from its optimizations such as matrix tiling and pipelining of data transfers. Furthermore, NVBLAS does not support stream processing and kernel batching as in **GPU Impl**, falling short in meeting the demand of high concurrency of Spark and resulting in low GPU utilization. Finally, the overhead of NVBLAS in initializing the NVBLAS library, reading a configuration file, and selecting GPUs is not negligible. In contrast, **GPU Impl** avoids this overhead by directly operating through GPU function calls. As a result, we see that in Figure 4, for dense matrices, NVBLAS always performs the worst among the hardware acceleration variants. For the libraries for sparse matrices (spBLAS and cuSPARSE), spBLAS performs similar to the Scala implementation of MLlib, and cuSPARSE degrades computation heavily. This is also in line with our hardware profiling results in § III (Figure 2(b)). However, we do not see this impact on the overall performance in Figure 4. This is because the computation portion is not dominant for sparse matrices (Figure 5), thus the impact is again negated by the other overheads. Nevertheless, ARION's selection of the best processing variants and utilizing multiple resources still results in overall performance improvement.



(a) Dense matrix. (b) Sparse matrix.

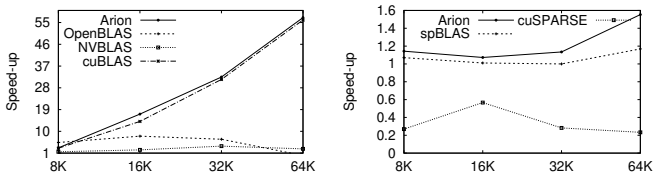
Fig. 5: Performance breakdown of studied matrices under ARION.

C. System utilization

Next, we repeat the experiments with $16K \times 16K$ dense matrix and $64K \times 64K$ sparse matrix using MLlib, ARION, OpenBLAS, NVBLAS, and **GPU Impl**. We observe differences in resource utilization across variants.

First, we compare the CPU utilization of MLlib, ARION, and OpenBLAS as shown in Figure 7. Here, x-axis represents the time sequences, and the y-axis depicts the average CPU utilization of the testbed nodes. Since the computation using MLlib takes longer than 250 seconds, we omit the tail results after $250sec$, where the average utilization is less than 1.3%. We observe that OpenBLAS has the highest CPU utilization, reaching 100%, since this variant heavily exploits CPUs to accelerate matrix multiplication. MLlib also exhibits medium CPU utilization, which reaches 37.24% due to the usage of Scala based implementation executing on CPUs. However, this Scala based implementation has much lower CPU utilization compared to OpenBLAS. In contrast, ARION shows the lowest CPU utilization with an average of 2.49%. For default $1K \times 1K$ dense blocks, ARION favors GPU accelerations for computation, and **GPU Impl** calculates fairly fast for each task and finishes earlier than OpenBLAS. Moreover ARION selects **GPU Impl** for all tasks as there is only one wave of tasks for $16K \times 16K$ matrix.

Figure 8 shows the aggregate GPU utilization of all nodes for ARION, ARION using **GPU Impl** only, and NVBLAS. Note that maximum GPU utilization reaches $6 \times 100\%$. Here we observe that NVBLAS demonstrates the lowest GPU utilization among the three. The tiling of NVBLAS is effective on block sub-matrices with size greater than $2K$ instead of small ones. Compared to NVBLAS, **GPU Impl** increases GPU utilization by associating each small task to a CUDA stream and batching the task execution. ARION shows a similar GPU usage with **GPU Impl** due to the main adoption of **GPU Impl** for the $16K \times 16K$ matrix case. The aggregated GPU utilization of both ARION and **GPU Impl** fails to reach the maximum capacity of the cluster. This is because in Spark, matrix multiplication tasks first shuffle data to retrieve needed



(a) Dense matrix. (b) Sparse matrix.

Fig. 6: Speed-up of aggregated compute time with ARION, and other processing variants with respect to the Scala implementation in MLlib.

sub-matrices. The actual computation happens after all the data has been obtained. In this case, the actual computation of different tasks may start at different times based on how long the shuffle takes for a task. In the case of multiple waves, the discrepancy in computation start time increases across tasks. This feature of MapReduce based scale-out platforms like Spark limits the chances of fully extracting GPU utilization with the batching task design of **GPU Impl** inside ARION, and also prevents the **Scheduler** of ARION to detect GPU resource contention. However, such shuffle behavior is nondeterministic so we can benefit from the flexibility of ARION to fully utilize idle resources when available. Moreover, when applied to a multi-tenant environment or multiple parallel phases of a job—where there may be resource contention, e.g., when a GPU intensive job runs alongside a matrix multiplication job—, **Scheduler** is able to avoid resource contention by scheduling tasks to resources with higher availability. For highly synchronized scale-out platforms such as Bulk Synchronous Parallel (BSP) based frameworks [28], [29], [30], ARION is expected to be able to fully utilize the hardware resources and schedule tasks to avoid resource contention.

To evaluate how ARION effectively utilizes both CPU and GPUs resources, we run an experiment with sparse matrix multiplication of size $64K \times 64K$. Figure 9 shows aggregate CPU and GPU utilization. Here, while CPUs are mostly fully utilized, GPUs exhibit a spike pattern throughout the execution. While **Selector** initially selects spBLAS, **Scheduler** notices idle GPU resources and fully occupied CPU slots. **Scheduler** then reassigns the tasks to GPU based variants (**GPU Impl** with cuSPARSE). The fluctuation in GPU utilization comes from ARION's decision to respect choices of **Selector**. That means tasks are assigned with spBLAS whenever a CPU slot becomes available. Thus, we see that ARION improves the overall resource utilization.

D. Impact of block size

In our next experiment, we perform Gramian matrix calculation with $16K \times 16K$ dense matrix using ARION with different block sizes. We record the end-to-end runtime of all cases and break it down into the three stages as illustrated in Figure 10. Here we can see that Spark spends most of time on data preprocessing and performing multiply computation, and the time on writing outputs is thus trivial. Therefore, we focus on preprocessing and multiplication times. The data preprocessing is minimum with block size of 512, but the matrix multiplication is minimum with block size of $1K$, the default case. The overall performance reaches minimum at 9.08 minutes with the block size of $1K$. Although suboptimal block sizes (such as 256) cause huge data preprocessing overhead from data partitioning and shuffling, in this paper we fix block size with default value and focus on optimizing computation time.

Next, we study the relationship between computation and block sizes, by repeating the experiment with hardware acceleration implementations (**GPU Impl**, OpenBLAS, and NVBLAS). We record the average compute time of each task. Figure 11 shows the results. For all the implementations, computation time increases as the block size increases. Larger block sizes result in larger matrices and longer computation time, which is consistent with our earlier observation (Figure 2(a)). However, in contrast to Figure 2(a), where NVBLAS performs better when the matrix size grows and finally outperforms other two implementations, here we see that NVBLAS performs worse as the block size grows and the gap between the other two variants also increases. This is because under the highly concurrent environment of Spark, NVBLAS cannot handle multiple task requests, and causes a delay to execute subsequent tasks. With smaller sizes of matrices, although the number of tasks increases, the computation finishes fast enough to overlap the shuffle of other tasks. However, when the matrix size grows, even if there are fewer tasks, the computation takes longer and tasks are prone to a higher chance and duration of waiting. On the contrary, the Spark-aware design of **GPU Impl** yields a consistent best performance among the processing variants in any block size.

E. Scalability of ARION

In our final experiment, we test the scalability of ARION. In this experiment, we scale ARION in a 128-node "Rhea CPU" cluster (4096 cores in total). Table II shows the specifications for both setups. Here, we configure Spark executor with $120\ GB$ memory and 64 cores for the CPU nodes, and repeat the experiments for dense matrices using default Spark MLlib. We record the end-to-end execution time of both default Spark MLlib and ARION. ARION is able to apply CPU hardware acceleration for dense matrix multiplications in this set-up. We are able to scale our input matrix up to $96K \times 96K$ with a raw file size of $177\ GB$ for ARION. However, default Spark MLlib fails to finish the $96K \times 96K$ matrix case due to *OutOfMemory* error after 7.2 hours running as plotted in the figure. This is because the computation with Scala implementation is executed inside JVM and thus requires on-heap memory space, while hardware acceleration uses out-of-heap memory (sometimes GPU memory) for calculation. Figure 12 shows the results. Note the log-scaled y axis. We see that ARION is also able to improve the performance by 13.35% (up to 40.9%) in a large scale system. This improvement is however limited by the CPU only infrastructure of the "Rhea CPU" environment. We expect ARION to have more performance improvement on emerging systems that have multiple hardware accelerators.

We also compared ARION on a 6-node "Rhea GPU" with

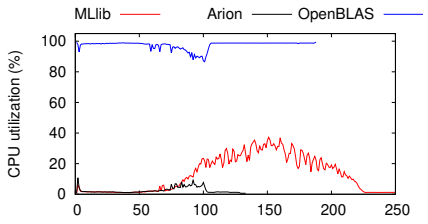


Fig. 7: CPU utilization of $16K \times 16K$ dense matrix using Spark MLlib, ARION, and OpenBLAS.

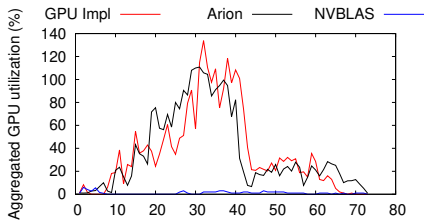


Fig. 8: Aggregated GPU utilization of $16K \times 16K$ dense matrix using ARION, **GPU Impl**, and NVBLAS.

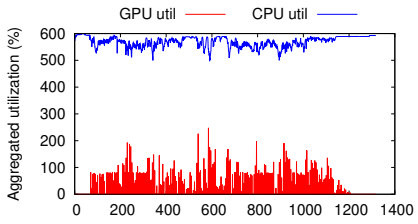


Fig. 9: Aggregated utilization of $64K \times 64K$ sparse matrix using ARION.

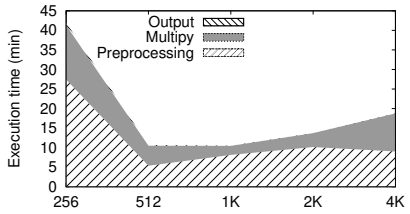


Fig. 10: Breakdown of end-to-end execution time with different block sizes.

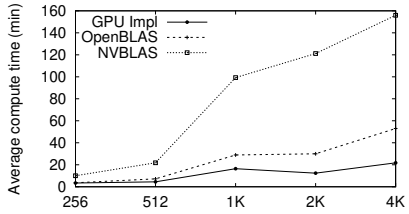


Fig. 11: Average computation time of *Block-Matrix* multiplication using different block sizes with **GPU Impl**, OpenBLAS, and NVBLAS.

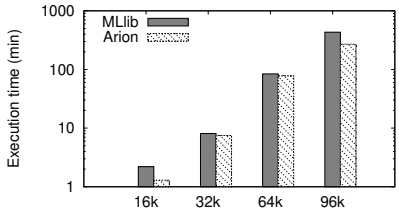


Fig. 12: Performance of default Spark MLlib and ARION in a 128-node scale. Y-axis is log-scaled.

default Spark on a scale-out 16-node CPU only cluster. Our results show that ARION on a smaller setup performs similar to default Spark MLlib on a larger setup (the detailed results are omitted due to page limit). Moreover, ARION improves performance by 17.45% for $16K \times 16K$ matrix, and 39.13% for $32K \times 32K$ matrix compared with default Spark MLlib on a larger setup. Since ARION effectively utilizes both CPUs and hardware accelerators in the cluster, we can achieve the same or better performance with a smaller cluster of fat machines than a larger cluster of less-endowed machines.

V. Related work

Advancing the computational capability of fundamental mathematical operations such as matrix operations is critical to brining large scale data analytics to bear upon knowledge discovery across domains [1], [31]. For instance, scalable matrix inversion has been studied both with Hadoop [7] and Spark [8]. Elgamal et al. [9] show that optimizing matrix operations can improve both scalability and performance of ML algorithms. Gittens et. al [32] investigated both performance and scalability of the randomized CX low-rank matrix factorization on Spark. Zadeh et. al [23] optimize Singular Value Decomposition (SVD)— which can be considered as a form of matrix factorization—in Spark by distributing matrix operations to the cluster while keeping vector computations local to the driver node. In addition, scaling up individual machines for enhancing the throughput of matrix computations is also promising. For instance, numerous works have focused on optimizing GEMM operations [13], [14], [15] and, in turn, ML algorithms [33], [34], [35], through exploiting GPUs.

The Spark community has also initiated discussions regarding utilizing hardware accelerations from within the Spark [36], [37] platform, so as to realize the benefits from hardware acceleration for increased throughput of matrix computations on individual nodes [38], [39], [10]. To this end, a preliminary study about using hardware optimized libraries for both CPU and GPU on a single machine for matrix multiplication in Scala has shown promising results [40]. Similarly, HeteroSpark [41] showed that RMI can be used to reduce communication overheads between CPU and GPU in Spark. SparkNet [42] provides a Spark interface to use Caffe framework [35] for training large-scale deep neural networks, where the instance of Caffe framework on each node can use GPUs, and Spark maintains data on system memory, managed by CPUs. However, many challenges remain when employing combining the above scale-out and scale-up techniques. Some approaches are not applicable to general data analysis [42], and further, utilizing hardware optimized libraries often requires an in-depth understanding of the hardware characteristics for each computation in data analysis pipeline, which tends to be cumbersome and impractical. Consequently, efficient utilization of hardware acceleration in a cluster is not available in popular distributed matrix computation packages such as ScaLaPACK [43], PLAPACK [44], CombBLAS [45], and Elemental [5]. While the focus of the above approaches is different, they essentially offer C/C++ programming libraries atop MPI library. Our approach is complementary to these works and aims to reconcile the Spark philosophy of providing a general-purpose, fault-tolerant data analysis platform with benefits of using hardware optimized libraries for extracting higher performance, specifically for crucial matrix multiplication operations.

VI. Conclusion

We have presented ARION, a dynamic hardware acceleration framework atop Spark. ARION supports scale-up accelerations for linear algebraic operations in scale-out data processing platforms. The approach adopts both drop-in solution of BLAS libraries and customized GPU acceleration in Spark for the best-effort processing of both dense and sparse matrices.

Moreover, we design a scheduler that maps tasks to different resources at runtime based on both matrix characteristics and system utilization. Our evaluation of ARION shows a 2× end-to-end speed-up compared to extant solutions in Spark. In addition, ARION is able to exploit hardware acceleration to enable the use of a smaller-sized cluster to achieve performance comparable to that of a larger Spark cluster. Although this work focuses on the usecases of matrix operations to provide a proof-of-concept, the result from this work can have a broad impact on various algorithms in machine learning and graph analysis, since the most of machine learning and graph algorithms are implemented in a layered approach on top of matrix operations.

REFERENCES

[1] A. Belianinov, R. Vasudevan, E. Strelcov, C. Steed, S. M. Yang, A. Tselev, S. Jesse, M. Biegalski, G. Shipman, C. Symons *et al.*, "Big data and deep data in scanning and electron microscopies: deriving functionality from multidimensional data sets," *Advanced Structural and Chemical Imaging*, vol. 1, no. 1, pp. 1–25, 2015.

[2] M. E. Wall, A. Rechtsteiner, and L. M. Rocha, "Singular value decomposition and principal component analysis," in *A practical approach to microarray data analysis*. Springer, 2003, pp. 91–109.

[3] H. Ringberg, A. Soule, J. Rexford, and C. Diot, "Sensitivity of pca for traffic anomaly detection," in *ACM SIGMETRICS*, 2007.

[4] C. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," *Advances in neural information processing systems*, 2007.

[5] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM TOMS*, vol. 39, no. 2, p. 13, 2013.

[6] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proc. of ICDM'09*, pp. 229–238.

[7] J. Xiang, H. Meng, and A. Aboulnaga, "Scalable matrix inversion using mapreduce," in *Proc. of the 23rd ACM HPDC*, 2014, pp. 177–190.

[8] J. Liu, Y. Liang, and N. Ansari, "Spark-based large-scale matrix inversion for big data processing," *IEEE Access*, vol. 4, p. 2166, 2016.

[9] T. Elgamal, M. Yabandeh, A. Aboulnaga, W. Mustafa, and M. Hefeeda, "spca: Scalable principal component analysis for big data on distributed platforms," in *Proc. of ACM SIGMOD ICMD*, 2015, pp. 79–91.

[10] J. Canny and H. Zhao, "Big data analytics with small footprint: Squaring the cloud," in *Proc. of the 19th ACM SIGKDD*, 2013, pp. 95–103.

[11] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proc. of the 30th ICML*, 2013, pp. 1337–1345.

[12] J. Hofmann, J. Treibig, G. Hager, and G. Wellein, "Comparing the performance of different x86 simd instruction sets for a medical imaging application on modern multi-and manycore chips," in *Proc. of WPMVP*. ACM, 2014, pp. 57–64.

[13] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning gemm kernels for the fermi gpu," *TPDS*, vol. 23, no. 11, pp. 2045–2057, 2012.

[14] R. Nath, S. Tomov, and J. Dongarra, "An improved magma gemm for fermi graphics processing units," *HPCA*, vol. 24, no. 4, p. 511, 2010.

[15] N. Nakasato, "A fast gemm implementation on the cypress gpu," *ACM SIGMETRICS*, vol. 38, no. 4, pp. 50–55, 2011.

[16] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016.

[17] Netlib, "Blas (basic linear algebra subprograms)," 2017, http://www.netlib.org/blas/.

[18] W. S. Zhang Xianyi, Wang Qian, "Openblas : An optimized blas library," 2011, http://www.openblas.net.

[19] NVIDIA, "Nvidia cuda blas library," 2007, http://docs.nvidia.com/cuda/cublas/#axzz4See8FUGO.

[20] I. S. Duff, M. Marrone, G. Radicati, and C. Vittoli, "A set of level 3 basic linear algebra subprograms for sparse matrices," *ACM Trans. Math. Softw*, vol. 23, pp. 379–401, 1995.

[21] L. Xu, S.-H. Lim, A. R. Butt, S. R. Sukumar, and R. Kannan, "Fatman vs. littleboy: scaling up linear algebraic operations in scale-out data platforms," in *Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*. IEEE Press, 2016, pp. 25–30.

[22] R. B. Zadeh and A. Goel, "Dimension independent similarity computation," *The Journal of Machine Learning Research*, vol. 14, no. 1, pp. 1605–1626, 2013.

[23] R. B. Zadeh, X. Meng, A. Staple, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Ulanov, and M. Zaharia, "Matrix computations and optimization in apache spark," in *Proc. of the 22nd ACM SIGKDD*. ACM, 2016.

[24] NVIDIA, "Nvidia blas library," 2007, http://docs.nvidia.com/cuda/nvblas/#abstract.

[25] netlib java, "High performance linear algebra," 2013, https://github.com/fommil/netlib-java.

[26] ORNL, "Rhea - oak ridge leadership computing facility," 2017, https://www.olcf.ornl.gov/computing-resources/rhea/.

[27] D. Kanter, "Intel's haswell cpu microarchitecture," *Real World Technologies, November*, 2012.

[28] Z. Wang, Y. Bao, Y. Gu, F. Leng, G. Yu, C. Deng, and L. Guo, "A bsp-based parallel iterative processing system with multiple partition strategies for big graphs," in *Proc. of IEEE Big Data*, 2013, pp. 173–180.

[29] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. of ACM SIGMOD ICMD*, 2010, pp. 135–146.

[30] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *Proc. of the 2nd IEEE CloudCom*, 2010, pp. 721–726.

[31] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman *et al.*, "Rethinking data-intensive science using scalable analytics systems," in *Proc. of ACM SIGMOD ICMD*, 2015, pp. 631–646.

[32] A. Gittens, J. Kottalam, J. Yang, M. F. Ringenburg, J. Chhugani, E. Racah, M. Singh, Y. Yao, C. Fischer, O. Ruebel, B. Bowen, N. Lewis, M. W. Mahoney, V. Krishnamurthy, and Prabhat, "A multi-platform evaluation of the randomized cx low-rank matrix factorization in spark," in *Proc. of the 5th ParLearning*, 2016.

[33] S. R. Agrawal, C. M. Dee, and A. R. Lebeck, "Exploiting accelerators for efficient high dimensional similarity search," in *Proc. of the 21st ACM SIGPLAN*, 2016, p. 3.

[34] N. Lopes and B. Ribeiro, "Gpumlib: An efficient open-source gpu machine learning library," *IJCISIM*, vol. 3, pp. 355–362, 2011.

[35] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. of the 22nd ACM MM*, 2014, pp. 675–678.

[36] Apache Spark, "Support off-loading computations to a gpu," https://issues.apache.org/jira/browse/SPARK-3785, 2014.

[37] ——, "Explore gpu-accelerated linear algebra libraries," https://issues.apache.org/jira/browse/SPARK-5705, 2015.

[38] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using gpus in software packet processing," in *Proc. of the 12th NSDI 15*, 2015, pp. 409–423.

[39] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "Virtualizing deep neural networks for memory-efficient neural network design," *arXiv preprint arXiv:1602.08124*, 2016.

[40] A. Ulanov, "Nvblas:gpu usage with nvblas," 2013, https://github.com/fommil/netlib-java/wiki/NVBLAS.

[41] P. Li, Y. Luo, N. Zhang, and Y. Cao, "Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms," in *Proc. of IEEE NAS*, 2015, pp. 347–348.

[42] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, "Sparknet: Training deep networks in spark," *arXiv preprint arXiv:1511.06051*, 2015.

[43] L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley *et al.*, "Scalapack: a portable linear algebra library for distributed memory computers-design issues and performance," in *Proc. of ACM/IEEE SC*, 1996.

[44] P. Alpatov, G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, R. van de Geijn, and Y.-J. J. Wu, "Plapack: Parallel linear algebra package design overview," in *Proc. of SC*, 1997, pp. 1–16.

[45] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *HPCA*, vol. 25, no. 4, pp. 496–509, 2011.