FatMan vs. LittleBoy: Scaling up Linear Algebraic Operations in Scale-out Data Platforms

Luna Xu*, Seung-Hwan Lim†, Ali R. Butt*, Sreenivas R. Sukumar† and Ramakrishnan Kannan†
*Virginia Tech, †Oak Ridge National Laboratory
*{xuluna, butta}@cs.vt.edu, †{lims1, sukumarsr, kannanr}@ornl.gov

***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 highly desirable to enable efficient processing over millions of data points. To this end, we present a matrix manipulation approach to effectively scale-up each node in a scale-out data parallel platform such as Apache Spark. Specifically, we enable hardware acceleration for matrix multiplications in a distributed Spark setup without user intervention. Our approach supports both dense and sparse distributed matrices, and provides flexible control of acceleration by matrix density. We demonstrate the benefit of our approach for generalized matrix multiplication operations over large matrices with up to four billion elements. To connect the effectiveness of our approach with machine learning applications, we performed Gramian matrix computation via generalized matrix multiplications. Our experiments show that our approach achieves more than $2\times$ performance speed-up, and up to 96.1% computation improvement, compared to a state of the art Spark MLlib for dense matrices.**

I. Introduction

Modern advances in data collection have led to the proliferation of high-dimensional data in both science [9], [39] and business [35]. Such high-dimensional data can be most effectively represented as a matrix. This is mainly because matrix representation makes the data amenable to linear algebraic operations that form the basis of machine learning algorithms such as Singular Value Decomposition (SVD) and Principal Component Analysis (PCA) [16], [37]. The importance of the targeted linear algebraic operations stems from the fact that many popular data analysis methods can be constructed through small core computations, so called as kernels [12]. closely related to matrix operations [12].

To perform scalable data analysis, it is desirable to have a high kernel computing throughput, i.e., to be able to compute the kernels for each of a large amount of data samples in a fast and timely fashion. The extant practices largely fall into two groups: (1) raising the throughput of matrix operations by enabling scaling out on commodity hardware via the use of data parallel computation software platforms such as Apache Spark [14], [25], [41], Hadoop [22], and MPI [33]; and (2) scaling up the computational power of individual machines for data analysis [11], [13], as data analysis algorithms entail matrix operations and optimization algorithms [12] that can benefit from hardware accelerators, e.g., GPUs [15], ASICS [18], FPGAs [36], and specialized instructions in CPUs [19] for high-throughput matrix operations [23], [29], [30], [40].

While beneficial, both the scale-out and scale-up techniques have some shortcomings. Scale-out options can suffer communication networking overheads when the size of a cluster becomes too large. Moreover, maintaining more machines entails higher costs (energy, manpower, machine failure, etc.). Scale-up options have a hard-wall of scalability, limited by the resources available in a single machine. The challenge is *how to best combine and reconcile the two scaling approaches in the service of scalable matrix operations?*

Combining the scale-out and scale-up techniques is nontrivial, given the fundamental design choices made under each approach. Scale-out approaches, such as Spark, focus on scalability, fault tolerance, cluster utilization, and workload balancing, but give less attention to hardware acceleration at single node scale. For example, the map-reduce model-based Spark maps a job to a number of tasks that can later be run on a number of nodes; utilization of hardware capabilities by each task is not managed. On the other hand, scale-up solutions focus on parallelism at a fine granularity, such as SIMD and SIMT models. To better utilize the scale-up solutions, it is important that the tasks are amenable to hardware acceleration.

However, most of the scale-out platforms like Spark *do not generate tasks in a way that can utilize specific local accelerators*. For instance, distributed matrices in Spark MLlib [27], a widely used machine learning library built on top of Spark, are often considered sparse matrices, and operations on sparse matrices are implemented with an ad-hoc Scala implementation. Although matrix solutions are beginning to be developed [43], they still face challenges of constructing and manipulating distributed dense matrices efficiently. This is because the general support for dense matrices could entail refactoring a substantial portion of MLlib. Thus, the implementations of matrices in Spark focus on sparse matrices and cannot utilize accelerators and other optimization available for dense matrices. To enjoy the benefit brought by both scale-out and scale-up capabilities of Spark, users have to understand and implement system-level details. The overhead of doing this is counter productive for domain scientists and data analysts due to the system-level complexity of these tasks. To address the above issues, we present an approach to generate distributed tasks so as to better utilize scale-up and scale-out methods with an emphasis on matrix operations for data analysis applications.

To scale-up matrix operations on distributed data processing platforms, we propose an approach that utilizes hardware accelerators like GPUs and newer optimized instructions in CPUs through highly optimized libraries, which, in turn, accelerates the execution of data analysis tasks. Our approach enables hardware acceleration for both dense and sparse matrices. The major anticipated benefit from our approach is that we can use a relatively smaller sized cluster to deliver the same performance as that of a bigger (more expensive) setup. As with [13] enabling large scale data processing using a smaller system footprint democratizes large scale machine learning capabilities to small–medium research groups.

Specifically, this paper makes the following contributions.

- We introduce the support for distributed dense matrix manipulations in Spark for scale-out matrix operations. In contrast, the current state of the art, e.g., MLlib [27], considers only sparse matrices.
- We adopt scale-up hardware acceleration for BLAS-3 operations [31] of distributed matrices. Our approach of enabling accelerator support inside Spark does not require changes to the Spark applications on user side by leverage existing libraries to support the target operations, which are available for the accelerators.
- We design a flexible control of deciding when and whether to use hardware accelerators based on the density of matrices.
- Finally, we evaluate our approach with Gramian matrix calculation, a typical example of matrix multiplication, on a 2-node cluster each with 2 Intel Xeon CPUs and 2 NVIDIA K80 GPUs attached as accelerators to demonstrate the benefit of our scaling-up approach. Our experiments show that hardware accelerators can achieve more than 2× speed-up than the state-of-the art MLlib implementation for end-to-end performance, improving computation time by up to 96.1%.

II. Related work

The surge of high dimensional data sets generated by scientific instruments, experiments, sensors, and supercomputer simulations [9] has made scalable data analysis an essential component of the scientific discovery process. A large number of studies have employed scalable data analysis frameworks for scientific discoveries. ADAM [32] shows a significant performance speed-up and cost reduction by using Spark for astronomy image processing. Thunder [17] is a library built on Spark that provides scalable data analysis for neuroscience. Kira [45] is yet another toolkit built on Spark for astronomy image processing, which achieves performance speed-up for big data sets. These works show promising results for utilizing big data processing frameworks such as Spark for high-dimensional scientific data analysis in terms of both performance and costs.

Together with scale-out data parallel frameworks, scaling up individual machines also has been explored for enhancing the throughput of matrix computations. For instance, numerous works have focused on optimizing GEMM operations [23], [29], [30], [40] and, in turn, machine learning algorithms [4], [20], [26], through exploiting GPU's capability in providing high throughput of matrix operations. In addition, studies on the optimal usage of multi-core CPUs or new hardware introductions attempted to fill the deficits of GPU-based approaches such as inefficiency of GPUs for a small amount of data [21] and overcoming challenges in processing large scale data over the memory capacity of a single GPU is also studied [34]. The use of both CPUs and GPUs for matrix operations has been proposed as well [2], [11].

In light of the above developments, the Spark community has initiated discussions regarding utilizing hardware accelerations from within the Spark [6], [8] platform. 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 [38]. Similarly, HeteroSpark [24] showed that RMI can be used to reduce communication overheads between CPU and GPU in Spark. SparkNet [28] provides a Spark interface to use Caffe framework [20] 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. Still, there are many challenges that remain. Some of the above approaches may not be applicable to general data analysis [28], 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 still not widely available in popular distributed matrix computation packages such as ScaLaPACK [10], PLAPACK [5], and elemental [33].

Complementary to the aforementioned studies, our proposed approach aims to reconcile the Spark philosophy of providing general-purpose data analysis with benefits of using hardware optimized libraries as one of the accelerators for extracting higher performance.

III. Design

In this section, we first present the factors that affect the design of our approach. Then, we give an overview of our architecture, followed by how we employ hardware acceleration for matrix multiplications in Spark. The key design considerations for our approach are as follows:

User transparency: Spark has been widely adopted and has a big user base with thousands of existing applications. To ensure that our approach will not disrupt such applications, we aim to provide hardware acceleration without requiring application changes, and with minimal modifications to the Spark framework.

Scalable matrix multiplication support: Existing BLAS hardware accelerations are designed for single machines, where the calculation of large matrices (input matrix and intermediate data) cannot fit into the memory. We aim to support big matrix multiplications at scale so as to handle crucial data sets that cannot fit into a single-node's memory.

Dense and sparse matrix support: We have experimentally observed that it is not always beneficial to enable hardware

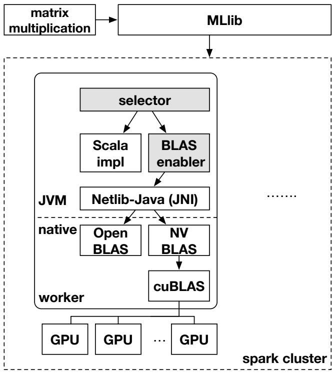


Fig. 1: System architecture.

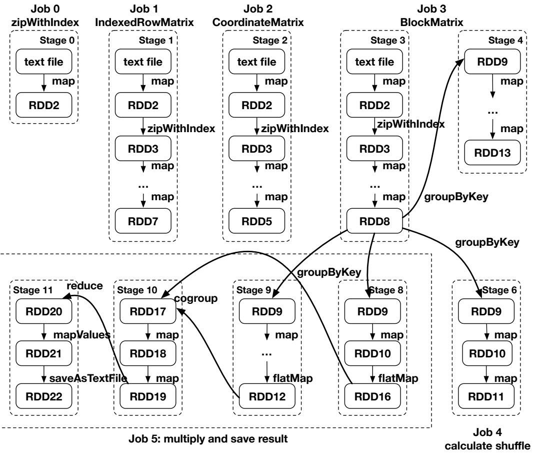


Fig. 2: DAG of Gramian matrix calculation using BlockMatrix in Spark MLlib.

acceleration for matrix operations. Hardware acceleration of BLAS helps dense matrix operations. However, for highly sparse matrices, the optimized algorithm provided by Spark MLlib are comparable or faster than native BLAS libraries. We aim to support both dense and sparse matrix multiplications in our system.

A. System architecture

Figure 1 shows the overall architecture of our approach. We start with a Spark cluster setup comprising multiple worker nodes. Each node is equipped with one or more GPUs. We design two new components, namely **selector** and **BLAS enabler**, which both run in a Spark executor. Each worker node runs one or more such executors.

Matrix multiplication can be handled by linear algebra classes in Spark MLlib [27]. When a matrix multiplication application is submitted, MLlib distributes the matrices on each node as standard Spark resilient distributed datasets (RDDs) [44]. Based on different matrix abstractions provided by MLlib, Spark generates tasks to perform multiplications of matrix partitions on worker nodes locally, i.e., each task performs multiplication of sub-matrices in a single node. Currently, Spark adopts a self implemented algorithm, i.e., a Scala implementation of matrix product, to compute local tasks. To adopt native libraries to scale-up the task computation, we design a **BLAS enabler** component to enable the use of underlying native BLAS libraries through a JNI interface (Netlib-Java [1]). In addition, to also support the original implementation for sparse matrices, we design a **selector** to decide whether to use the native BLAS libraries for hardware acceleration or use the default implementation of MLlib.

B. Hardware acceleration in Spark

Spark supports four distributed matrix abstractions: *RowMatrix*, *IndexedRowMatrix*, *CoordinateMatrix*, and *BlockMatrix*. Only *BlockMatrix* supports the multiplication of two distributed matrices [7], while others only support the multiplication of a distributed matrix and a local matrix, where a vector–matrix multiplication (BLAS-2) is performed locally, i.e., the distributed matrix is partitioned as row vectors, and each task multiplies a vector and the local matrix on a single node. When training a machine learning model, the multiplication of a distributed matrix and a local matrix can be useful if the model fits into the local memory. The training data can then be distributed in the cluster automatically by standard Spark operations. However, it is becoming more and more common to train a model that does not fit into a single machine's memory [13]. For training such a large model, a user would need to use *BlockMatrix* to fully scale-out the operations. Thus, we focus on *BlockMatrix* in this study.

A key challenge to scaling-up computation in Spark is the dependency between internal components in MLlib. As a result, it is non-trivial for the current MLlib to utilize optimized BLAS libraries. For example, *BlockMatrix* is designed to be transformed only from *CoordinatedMatrix*, which represents sparse matrices. With such a dependency, *BlockMatrix* is considered a sparse matrix abstraction and calculation for sparse matrices is performed by a self implemented algorithm, which takes away the opportunity of using hardware optimized BLAS libraries.

To address this problem, we first change *BlockMatrix* in MLlib to also support dense matrices. We extend the original `toBlockMatrix()` API by adding an argument for specifying matrix density. When generating a *BlockMatrix*, we keep the density information to differentiate dense *BlockMatrix* from sparse *BlockMatrix*. When a computation is performed in *BlockMatrix*, matrix density is sent to **selector** to decide whether to use the ad-hoc Scala implementation or native BLAS libraries.

Secondly, we design a **BLAS enabler** that leverages Netlib-Java as a JNI interface to delegate BLAS routines to underlying native BLAS libraries. With this software stack, we are able to use native libraries without user application changes. The goal is to design a so-called drop-in BLAS solution for Spark to optimally utilize underlying hardware. The main benefit of a drop-in solution is that there is no need to re-build Spark in order to utilize heterogeneous computing resources in the cluster. However, it is not straight forward to use this drop-in library solution for distributed matrix multiplication in Spark. In our implementation, we use OpenBLAS [3] to perform matrix multiplications on CPU, and NVBLAS [38]

TABLE I: Studied matrix sizes and density.

# of Rows (Columns)	Density	File Size (GB)
4873	1	0.34
14684	1	3.1
24495	1	8.4
66331	1	77
4873	0.05	0.104
24495	0.05	2.6
66331	0.05	19
97708	0.05	41

to perform matrix operations on GPU. NVBLAS is a BLAS-3 library provided by NVIDIA built on top of cuBLAS-XT [2], which supports a multi-GPU capable host interface. However, our design is flexible and we can use any library as needed in our approach.

Following section shows that hardware accelerations may not always perform better than the ad-hoc Scala implementation, depending on the sparcity of matrix. To support matrices with varying sparcity, we accept a system configuration of a density threshold with a default value of 0.5. **Selector** compare the aforementioned density of *BlockMatrix* with this value to decide whether or not to enable native libraries for the matrix. The threshold is exposed as a user-configurable parameter to suit different use cases.

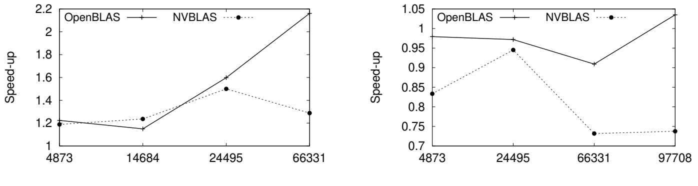
IV. Evaluation

This section evaluates our approach to use hardware acceleration in Spark for speeding up matrix multiplication. To this end, we use large scale Gramian matrix ($\mathbf{XX^T}$) computation kernel. This kernel is common among machine learning algorithms such as SVD and PCA [12]. Gramian matrix also plays critical role in popular data analysis techniques, e.g., all-pair similarity [42]. We use Spark version 1.6.1, the latest version at the time of this evaluation. We implemented Gramian matrix calculation using MLlib APIs. We divided the calculation into six jobs and 12 stages in Spark as shown in Figure 2. First four jobs create a *BlockMatrix* type matrix from input data files through intermediate data types (from *RowMatrix* to *IndexedRowMatrix* and *CoordinateMatrix*). Matrix multiplication happens in job-4 and job-5. Job-4 calculates the blocks to shuffle for the two input matrices. Job-5 groups input matrices by block, performs multiplication, and finally writes the result to output files. Each of the 12 stages represents a collection of tasks generated by the job without data shuffle. The shuffle occurs between the stages, in jobs 3, 4, and 5. Job-5 has 5 shuffle operations as it consists of four stages, while jobs 3 and 4 have only one shuffle operation each.

We compare Gramian matrix computation under three different settings: the current Spark MLlib implementation as our baseline case; Spark with CPU acceleration using OpenBLAS; and Spark with GPU acceleration using NVBLAS. We disabled **selector** in our experiments to show the performance impact of using hardware acceleration in sparse matrices. Note that we use hand-compiled OpenBLAS library with hardware acceleration instruction flags (e.g. AVX2, FMA3, etc.) set. For NVBLAS, we use the NVBLAS library shipped with CUDA toolkit 7.5. We generate random square matrices in various sizes and densities for our tests as shown in Table I. All experiments are performed in 2 GPU nodes in Rhea cluster. Table II shows the machine specifications. We configure Spark with one master node and 2 worker nodes, where one worker co-exists with the master node. Each worker node runs one executor. We configure the executors to have $800\ GB$ memory and 56 cores. We repeat each experiment five times and report the average results, except for the very-long running experiments with 66331×66331 and 97708×97708 matrices, where we report the average of two runs.

TABLE II: System specification for the evaluation.

Parameter	Value
System name	Rhea GPU node
CPU model	dual Intel Xeon E5-2695 @ 2.3 GHz
CPU cores	14×2 (28×2 HT)
CPU memory	1TB
GPU model	dual NVIDIA K80
GPU (CUDA) cores	4992×2
GPU memory	24×2 GB
CUDA ver.	7.5



(a) Dense Matrix

(b) Sparse Matrix

Fig. 3: Overall performance on GPU (NVBLAS) and CPU (openBLAS) normalized to Spark (MLlib).

A. Overall performance

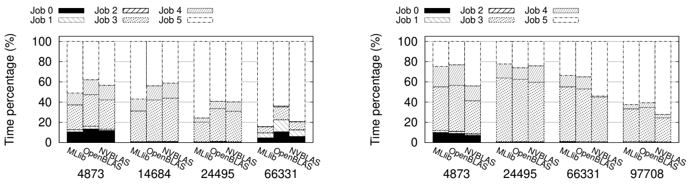
Figure 3 shows the end-to-end performance normalized to the case of MLlib for dense matrices and sparse matrices, We can see from the graph that, for dense matrices, both CPU and GPU acceleration achieved the overall speed-up of up to $2.2\times$ and $1.7\times$, respectively. For OpenBLAS, we observe that the speed-up increases as the matrix size increases within the tested range. However, for NVBLAS, the speed-up peaks at 24495×24495 matrix and diminishes after that. We investigate this behavior further in later sections.

On the other hand, we observe that, for sparse matrices with 5% of non-zero terms, hardware acceleration may not increase the performance much. Scala implementation of matrix multiplication in Spark is faster or comparable to OpenBLAS, and much faster than NVBLAS, as reported in [43]. This is mainly because NVBLAS pads zero terms into the matrix and sends dense matrix blocks to the GPU, which causes extra data transfer between GPU and CPU, as well as extra computation in GPU and thus affects performance.

Discussion: The overall execution time results show that hardware acceleration accelerates dense matrix multiplications but degrades the performance for highly sparse matrices.

B. Performance breakdown

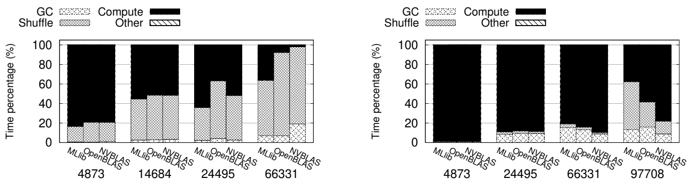
As shown in Figure 2, the overall performance includes execution time of every job, while only job-4 and job-5 are



(a) Dense matrix.

(b) Sparse matrix.

Fig. 4: Performance breakdown by job.



(a) Dense matrix.

(b) Sparse matrix.

Fig. 5: Performance breakdown of stage 10.

related to matrix multiplication. Moreover, hardware acceleration only takes place in stage 10, where two block matrices are co-grouped to perform multiplication. To further investigate the behavior of Spark when calculating multiplication on distributed matrices, we study the breakdown of the end-to-end times by jobs and tasks. First, we examined the overall execution time per job. Figure 4 shows the percentage of run time for each job. We see that jobs 4 and 5 take more than 40%, and up to 84%, of total execution time. We also see that small and sparse matrices spend less time in actual multiplication, but spend more time on data transforming. This limits the overall performance improvement by adopting acceleration on computation for those matrices.

To further identify the impact of using hardware acceleration for matrix multiplication, we collect task-level run time of stage 10, and group them into 4 categories: garbage collection, shuffle read/write, computation, and others (schedule delay, serialization, etc.). We record the aggregated run time of all tasks by category and then calculate the percentage of time consumed under each category. Figure 5 shows the the percentage of run time in each category. We observe that both dense matrices with small sizes and sparse matrices have a larger portion of computation times. However, it is not against our previous observation about overall performance, as even though computation is dominant in Stage 10, job-5 has still smaller portion in end-to-end times. For dense matrices, as matrix size grows, shuffle overhead increases and becomes the dominant factor as shown in 66331×66331 matrix cases. Although the overall speed-up is less than 2.2× (Figure 3), CPU and GPU accelerate the compute time by, on average, 54.7% and 43.9%, respectively. For small matrices, OpenBLAS performs similar to NVBLAS or better, as GPU-CPU memory transfer is not involved. Also notice that for case of 66331×66331 of dense matrix, both CPU and GPU significantly improve the computation time, i.e., by 92.9% and 96.1%, respectively. In this case, NVBLAS performs better than hand-compiled OpenBLAS. However, this advantage fails to lead the overall performance improvement due to shuffle and other overheads, which gives computation time relatively less contribution towards overall performance.

For sparse matrices, we see that computation accounts for relatively big amount of time to other overheads, since the amount of shuffle data is small due to the high sparsity. We also continue to observe that native BLAS libraries take more time to compute than MLlib implementation with an average slowdown of 22% and 85.1% for CPU and GPU (Figure 5(b)), respectively, which aligns with the observation in Section IV-A.

Discussion: In Spark, the job of multiplication on distributed matrix is divided into a number of tasks, where each task calculates the multiplication of small block matrices. The default size of each block is 1024×1024. The number of tasks generated by Spark increases as the matrix size increases. As the number of tasks increases, data shuffle overhead increases, diminishing advantages of leveraging the advantage of hardware accelerators for performing high throughput computations. However, even with this shuffle and framework overhead, hardware acceleration can still speed-up overall performance up to more than 2× as shown in Section IV-A. On the flip side, we find that for matrix with small sizes, OpenBLAS performs similar to or faster than NVBLAS, but for the 66331×66331 matrix, NVBLAS actually spends less time on computation than OpenBLAS. Thus, we expect the benefits from utilizing high-throughput accelerators such as GPUs to grow with growing matrix sizes.

V. Conclusion

This paper shows an approach of applying scale-up accelerations for linear algebraic operations in scale-out data processing platforms. Specifically, we enable both CPU and GPU accelerations via native BLAS libraries inside Spark, and without requiring manual hacking or user-side intervention. We also provide control over acceleration choices, which are based on matrix density. Our initial experiments showed that hardware acceleration can achieve an overall up to 96.1%.

Acknowledgments

This work is sponsored in part by the NSF under the grants: CNS-1405697, CNS-1422788, and CNS-1615411. This research also used resources of the OLCF at the Oak Ridge National Laboratory and this 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, worldwide 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).

References

[1] High performance linear algebra. https://github.com/fommil/netlib-java.

[2] Nvidia cuda blas library. https://developer.nvidia.com/cublas.

[3] Openblas : An optimized blas library. http://www.openblas.net.

[4] S. R. Agrawal, C. M. Dee, and A. R. Lebeck. Exploiting accelerators for efficient high dimensional similarity search. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 3. ACM, 2016.

[5] 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 *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, SC '97', pages 1–16, New York, NY, USA, 1997. ACM.

[6] Apache Spark. Explore gpu-accelerated linear algebra libraries. https://issues.apache.org/jira/browse/SPARK-5705.

[7] Apache Spark. MLLib. http://spark.apache.org/docs/latest/mllib-data-types.html.

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

[9] 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*, 1(1):1–25, 2015.

[10] 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 *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pages 5–5. IEEE, 1996.

[11] J. Canny and H. Zhao. Big data analytics with small footprint: Squaring the cloud. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 95–103. ACM, 2013.

[12] 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.

[13] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *Proceedings of the 30th international conference on machine learning*, pages 1337–1345, 2013.

[14] T. Elgamal, M. Yabandeh, A. Aboulnaga, W. Mustafa, and M. Hefeeda. spca: Scalable principal component analysis for big data on distributed platforms. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 79–91. ACM, 2015.

[15] Facebook. Facebook to open-source ai hardware design. https://code.facebook.com/posts/1687861518126048/facebook-to-open-source-ai-hardware-design/.

[16] I. K. Fodor. A survey of dimension reduction techniques. Technical Report UCRL-ID-148494, Lawrence Livermore National Laboratory, 2002.

[17] J. Freeman, N. Vladimirov, T. Kawashima, Y. Mu, N. J. Sofroniew, D. V. Bennett, J. Rosen, C.-T. Yang, L. L. Looger, and M. B. Ahrens. Mapping brain activity at scale with cluster computing. *Nature methods*, 11(9):941–950, 2014.

[18] Google. Google supercharges machine learning tasks with TPU custom chip. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

[19] 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 *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, pages 57–64. ACM, 2014.

[20] 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 *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[21] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the bar for using gpus in software packet processing. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 409–423, 2015.

[22] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[23] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning gemm kernels for the fermi gpu. *Parallel and Distributed Systems, IEEE Transactions on*, 23(11):2045–2057, 2012.

[24] P. Li, Y. Luo, N. Zhang, and Y. Cao. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, pages 347–348. IEEE, 2015.

[25] J. Liu, Y. Liang, and N. Ansari. Spark-based large-scale matrix inversion for big data processing. *IEEE Access*, 4:2166–2176, 2016.

[26] N. Lopes and B. Ribeiro. Gpumlib: An efficient open-source gpu machine learning library. *International Journal of Computer Information Systems and Industrial Management Applications*, 3:355–362, 2011.

[27] 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*, 17(34):1–7, 2016.

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

[29] N. Nakasato. A fast gemm implementation on the cypress gpu. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):50–55, 2011.

[30] R. Nath, S. Tomov, and J. Dongarra. An improved magma gemm for fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24(4):511–515, 2010.

[31] Netlib. Blas (basic linear algebra subprograms). http://www.netlib.org/blas/.

[32] 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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 631–646. ACM, 2015.

[33] 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 Transactions on Mathematical Software (TOMS)*, 39(2):13, 2013.

[34] 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.

[35] H. Ringberg, A. Soule, J. Rexford, and C. Diot. Sensitivity of pca for traffic anomaly detection. In *ACM SIGMETRICS Performance Evaluation Review*. ACM, 2007.

[36] Ryft. Transforming high performance analytics: Converged storage and compute powered by fpgas unlocks 100x faster analytics. http://www.ryft.com/products.

[37] J. Shlens. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*, 2014.

[38] A. Ulanov. Nvblas:gpu usage with nvblas, 2016. https://github.com/fommil/netlib-java/wiki/NVBLAS.

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

[40] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16', pages 20:1–20:11, New York, NY, USA, 2016. ACM.

[41] J. Xiang, H. Meng, and A. Aboulnaga. Scalable matrix inversion using mapreduce. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 177–190. ACM, 2014.

[42] R. B. Zadeh and A. Goel. Dimension independent similarity computation. *The Journal of Machine Learning Research*, 14(1):1605–1626, 2013.

[43] 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. *arXiv preprint arXiv:1509.02256*, 2015.

[44] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[45] Z. Zhang, K. Barbary, F. A. Nothaft, E. Sparks, O. Zahn, M. J. Franklin, D. A. Patterson, and S. Perlmutter. Scientific computing meets big data technology: An astronomy use case. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 918–927. IEEE, 2015.