HeteroSpark: A Heterogeneous CPU/GPU Spark Platform for Machine Learning Algorithms

Peilong Li, Yan Luo Ning Zhang, Yu Cao

Dept. of Electrical and Computer Engineering Dept. of Computer Science
University of Massachusetts Lowell

***Abstract*—Analytics algorithms such as machine learning, image processing and bioinformatics on big data sets require tremendous computational capabilities. Spark is a recent development that addresses big data challenges with data and computation distribution and in-memory caching. However, as CPU only framework, Spark cannot leverage GPUs and a growing set of GPU libraries to achieve better performance and energy efficiency. We present HeteroSpark, a GPU-accelerated heterogeneous architecture integrated with Spark, which combines the massive compute power of GPUs and scalability of CPUs and system memory resources for applications that are both data and compute intensive. We make the following contributions in this work: (1) we integrate the GPU accelerator into current Spark framework to further leverage data parallelism and achieve algorithm acceleration; (2) we provide a plug-n-play design by augmenting Spark platform so that current Spark applications can choose to enable/disable GPU acceleration; (3) application acceleration is transparent to developers, therefore existing Spark applications can be easily ported to this heterogeneous platform without code modifications. The evaluation of HeteroSpark demonstrates up to 18x speedup on a number of machine learning applications.**

I. Introduction

Big data and data-centric science discovery have become the new research paradigm, deriving a wave of new knowledge in many domains such as medical, social and engineering. Yet both the scale of data and the computational complexity of analytics algorithms challenge the architecture and software framework of big data applications.

The analytics of big data sets such as user generated videos, query logs and purchase transactions helps us understand the context of the videos[1, 2], or optimize indexing and query responses [3], or predict user purchasing behavior [4], all of which require an enormous scale of computational capabilities in addition to storage capacity.

Recent research advance in machine learning and deep learning algorithms and data analytics tools imposes new requirements on existing computing systems and architectures. Machine learning algorithms such as singular vector decomposition (SVD), support vector machine (SVM), principal component analysis (PCA), clustering, and neural networks are applied to extremely large data sets to extract data information and build a knowledge base. New deep learning algorithms such as autoencoder [5], RBM [6], both computing and memory intensive, have become infeasible because of the scale of data [7]. To help address the challenges, MapReduce [8] takes advantage of the parallelism of distributed computation framework to split "big data" into manageable granularity in order to gain better speedup. In this direction, the recent development of Spark [9] significantly boosts the performance of Hadoop framework [10] with in-memory data computation.

We observe that there is a gap in CPU-based cluster computing frame such as Spark, for computationally intensive applications. On one hand, while Spark framework can partition and distribute data, the complex algorithms applied on the data unit in a single node still consume a large number of CPU cycles, therefore have difficulty in meeting real-time requirements. On the other hand, GPUs have been leveraged as accelerators in speeding up complex workloads such as K-Means and SVM [11], thanks to the density of the cores and their power efficiency. However, the memory space is generally limited to a single GPU node. Therefore, the marriage of GPUs to CPU-based cluster computing framework is a particularly interesting approach to address both big data and intensive computation challenges.

We are motivated to address the challenges with heterogeneous architecture where GPUs work side by side with CPUs at the worker nodes of a compute cluster. Such heterogeneous architecture has three design objectives: (1) to accelerate complex algorithms. The heterogeneous architecture will be utilized to speed the workloads on the best suitable processors, which often means that the GPUs will execute the phases of an application with massive data parallelism, and the CPU cores carry out the sequential phases; (2) to support "plug-n-play" where GPUs can be dropped in or removed flexibly. The configuration of the system resources should be elastic; and (3) to reduce programming complexity and achieve transparent acceleration, which we believe are the most important factors to encourage adoption and benefit a broad range of real user applications.

In this work, we present HeteroSpark, a middleware framework to achieve the aforementioned objectives. We augment the vanilla Spark framework by placing GPUs in Spark's worker nodes and employing Java Virtual Machines to exchange data between CPU and GPU domains. We design a mechanism to encapsulate the GPU capability in the form of callable functions, and present to the user applications through a well-designed APIs. The function calls from Spark applications are intercepted by our HeteroSpark, and then directed to invoke GPU kernel for the acceleration of chose functions, if the GPU resources are available. The HeteroSpark framework is effective and extensible, demonstrating great promise in performance, portability and transparent acceleration.

This work has several contributions:

- We present the first scalable computing architecture with heterogeneous processors (CPU + GPU) as an effective extension to Spark on computationally intensive workloads.
- We design APIs to seamlessly glue exiting and newly developed GPU kernels to HeteroSpark so that the growing set of GPU libraries can be leveraged.
- We conduct experiments to demonstrate the effectiveness of HeteroSpark in scalability and portability.

II. Related Work

This paper covers the topic of heterogeneous system architecture of distributed computing framework for large-scale analytics. In this section, we briefly summarize some related works in the same area and compare our work with the existing ones.

As MapReduce draws increasing attention from both academic and industrial side, many recent work investigates methods to accelerate MapReduce with GPU. The earliest works in this direction stays in the scope of single GPU MapReduce [12, 13]. Both of these works have their own implementation of MapReduce and only support single GPU execution. As stated in the introduction section, GPU memory resource is limited and expensive. In the big data world, a single GPU is not capable of handling the increasing large dataset.

A. Mooley et al. propose DisMaRC [14], a MapReduce framework on a network of GPUs. DisMaRC demonstrates scalable speedup over Mars with more number of GPUs. However, since DisMaRC uses MPI and is not built on top of Hadoop, it lacks fault tolerance and process scheduling based on data locality. In addition, since all map and reduce processes are executed on GPUs, this architecture will finally meet the dilemma of limited GPU memory resources versus huge GPU cluster cost.

J. Stuart et al. designed GPMR [15], a standalone MapReduce library written in C++ and CUDA to leverage the power of GPU clusters for large-scale computing. GPMR requires users to have their own implementations on every part of the MapReduce pipeline with provided operations. The big drawback of GPMR is that since all map and reduce workloads are exclusively executed on GPU rather than CPU, the data communication overheads cast a big challenge for GPMR applications to gain benefits.

HAPI (Hadoop+Aparapi) [16] is a library that is built on top of Hadoop and Aparapi [17]. S. Okur et al. propose HAPI with programming APIs to simplify the implementation of MapReduce on Hadoop that takes advantage of GPUs. HAPI provides mapper APIs to split map operation into three stages for acceleration, *preprocessing*, *gpu*, and *postprocessing*. Though showing promising speedup on the benchmark, HAPI still lacks the capability of scaling up since it is targeting on single machine acceleration.

Similar with HAPI, HadoopCL [18] also proposes an automatic code conversion method by using Aparapi to generate GPU kernels within the Hadoop framework. HadoopCL has demonstrated better performance than Hadoop by investigating parallelism in both mapper and reducer functions. However, confined by the data representation of Aparapi, the expression of mappers and reducers of HadoopCL application is limited.

J. Canny et al. propose BID Data Suite (BDS) [19], a single machine solution to compete with cluster implementations for common machine learning and data mining algorithms. The BDS consists of an accelerated matrix computing backbone, and a library of machine learning models that build on top. The major differences between BDS and HeteroSpark can be two folds. Firstly, HeteroSpark believes cluster computing provides more scalability theoretically and practically in handling increasing size of big data. In contrast, BDS is targeting using less computing resource on a moderate size of big data to reduce cost and power consumption. Secondly, while BDS provides its own computing platform, HeteroSpark aims at accelerating the increasing number of Spark applications since Spark is now a mature and industrial level distributed computing framework.

Glasswing [20] is another MapReduce framework that takes advantage of OpenCL-enabled heterogeneous clusters. Glasswing proposes a pipelined fashion of executing map and reduce and thus to mitigate communication and memory transaction overheads. Fault tolerance is one big issue in this framework, since Glasswing does not support the Hadoop-like task failure recovery mechanism.

III. The Design of HeteroSpark

A. Design Goals

The design goals of HeteroSpark are three folds. Firstly, we intend to establish a heterogeneous platform which consists of general purpose CPUs and GPU accelerators to accelerate computational intensive workloads such as machine learning applications, taking advantage of the salient features of both. Therefore, we need to design the communication paradigm in this heterogeneous architecture for data and computation. We decide to extend the Spark framework due to its scalability and performance. Secondly, Spark users should have the flexibility of applying GPU acceleration if GPUs are available and carrying out original workloads on CPUs if not. Our design ought to have the minimum revisions to the original Spark for augmenting GPU resources. Third, HeteroSpark will have better programmability and attract users if GPU acceleration is transparent to programmers. Thus, implementations on the GPU side should provide the same programming interface as in Spark supported libraries. The HeteroSpark is responsible for the workload scheduling onto GPUs and CPUs without needing user intervention. In addition, we aim to leverage the existing open source machine learning CUDA implementations or documented libraries, therefore design the wrapper

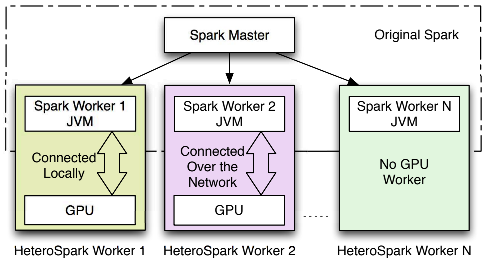


Fig. 1. HeteroSpark architecture

TABLE I
COMMUNICATION SOLUTIONS

Solutions	Pros	Cons
Java RMI	Secure, faster, lightweight	Java-specific
COBRA	Language independent	No garbage collection supported
SOAP	XML-based web service	Heavy and slow
Spring+JMS	Message queuing; simple programming interface	Steep learning curve; extra dependencies

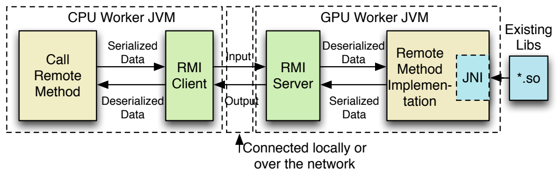
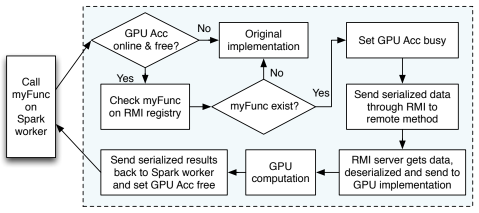


Fig. 2. HeteroSpark CPU-GPU Communication



Call myFunc on Spark worker
GPU Acc online & free?
No
Original implementation
Set GPU Acc busy
Yes
Check myFunc on RMI registry
No
myFunc exist?
Yes
Send serialized data through RMI to remote method
Send serialized results back to Spark worker and set GPU Acc free
GPU computation
RMI server gets data, deserialize and send to GPU implementation

Fig. 3. HeteroSpark Glue Logic

layer that contains Java native interface to make GPU libraries callable from a Spark worker.

B. Architecture Overview

HeteroSpark clusters can be regarded as Spark clusters with GPUs connected within some or all of Spark worker nodes. Figure 1 gives the overview of the HeteroSpark architecture. We can see that HeteroSpark extends original Spark with GPU acceleration option on Spark worker nodes. HeteroSpark currently supports three ways to connect GPUs with Spark workers: "local GPU" (GPU resides on the same Spark worker machine and interacts with the CPU via the PCIe bus), "remote GPU" (GPU resides on different Spark worker over the network), or "no GPU". All GPU enable/disable and connection options are configured in the cluster configuration file which is read on starting.

C. CPU-GPU Communication

We have compared different possible communication implementations for the CPU-GPU interface to find the best suitable solution. Among the options listed in Table I Java Remote Method Invocation (RMI) is chosen due to its secure, faster, and lightweight characteristics.

Using Java RMI technique, we implement the CPU-GPU communication layer on Spark worker nodes. The idea is to offload the workload from Spark worker node's CPU to the "remote method" that is implemented on the coupled GPU. As demonstrated in Figure 2, there are four major components in the communication layer: the CPU worker JVM, the input/output streams, the GPU worker JVM and the existing libraries. Upon calling the remote method from the CPU JVM, the CPU worker will serialize the data partition on this particular worker and send the serialized data to remote method on the GPU JVM side through the RMI communication interface. On the GPU JVM, a RMI server will get the incoming data, deserialize it, and send the data to the remote method implementation. Since one of our design goals is to utilize existing GPU libraries, thus the true logic of the remote method is in the dynamically linked library (*.so). The library is integrated with GPU JVM through Java Native Interface (JNI).

D. The Glue Logic

There are two usage methods for HeteroSpark: 1) use HeteroSpark with provided acceleration libraries, and 2) use HeteroSpark with user developed GPU acceleration libraries. We will discuss the first option in this subsection and cover the second option in the next subsection.

As shown in Figure 3, if using HeteroSpark provided acceleration libraries, developers can just treat the acceleration process as a black box, which means that Spark worker feeds the data into HeteroSpark, and gets the accelerated results back. We call the black box the "Glue Logic". The glue logic works as follows.

Assume we want to accelerate *myFunc* on a Spark worker. Upon calling *myFunc*, Spark worker will first check the "alive" flag of GPU accelerator to see if it is online and free. If not, logic flow will go back to the original CPU implementation without acceleration; if yes, logic flow will check whether *myFunc* exists on the RMI registry (available accelerators). If *myFunc* can be accelerated, then *myFunc* will take exclusive control on the coupled GPU and set the GPU "busy" flag. Data then can be serialized and sent from CPU JVM to the GPU JVM for computation. After the batch of data is computed, GPU JVM will again serialize the results and send it back to Spark CPU worker. The major overhead of the glue logic lies in the RMI communication step, where we show that the overhead is minimal IV.

E. GPU Accelerator Development

Since GPU accelerator libraries in HeteroSpark still keep spawning, chances are that developers may want to develop their own accelerators that could be integrated to HeteroSpark. The porting of GPU kernel to HeteroSpark requires implementing a wrapper ("Wrapper.c") and performing several compiling steps as follows.

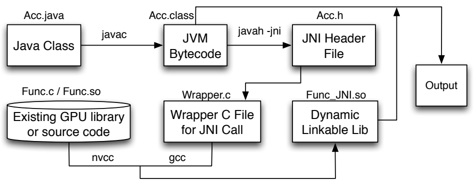


Fig. 4. GPU Accelerator Development

The development process is illustrated in Figure 4. Firstly, we should start with declaring a native method which contains no actual implementation in "Acc.java" file. Secondly, we will use Java compilation tools "javac" and "javah" to generate our required header file "Acc.h" which contains the method primitives. Thirdly, since the existing GPU libraries or CUDA source code contains the actual native implementation, we need to implement a wrapper layer "Wrapper.c" to glue the existing implementation with JNI. We can use the primitives in "Acc.h" to write the wrapper layer of the native implementation. Fourthly, we need to compile the CUDA implementation and the C implementation separately and then compile them together to generate the final dynamically linked library "Func_JNI.so". We finish all development till this step. To use the accelerator library, one can simply load it in the "Acc.java" file. We provide a script to automate all the compiling steps, thus only the "Wrapper.c" file requires extra effort.

We show the following code snippets as a reference of GPU library development.

```
/**** Acc.java ****/
......
File sharedLibrary = new File("Func_JNI.so");
System.load(sharedLibrary.getAbsolutePath());
public native int func(float[] para1, int para2 ...);
......
int retArray = obj.func(para1, para2, ...);
......

/**** Wrapper.c ****/
#include "Acc.h" // Generated by javah -jni
......
// Read all input parameters
jdouble *para1 = (*env)->GetFloatArrayElements(env, array1,
    0);
// Call native function which is implemented on GPU
func(para1, para2, ...);
// Return data back to Java env
(*env)->ReleaseFloatArrayElements(env, array1, para1, 0);
......
```

F. Workload Scheduling

For a heterogeneous system with both host CPUs and GPU accelerators, a good workload scheduling policy can be very beneficial to maximizing the system performance. Since the performance of different applications may be decided by application characteristics and dynamic computing environments, to find an optimized static scheduling policy is difficult. Before achieving an adaptive scheduling policy for dynamic workloads, we implement as the first step a basic scheduling policy - GPU aggressive scheduling which strives to fully utilize GPU at all times. A more intelligent way of scheduling will be studied in the near future where we plan to carry out profiling on both CPU and GPUs as well as applications.

IV. Performance Evaluation

A. Benchmark Applications

We evaluate the performance of HeteroSpark with widely used machine learning applications including Logistic Regression, K-Means and Word2Vec. We explore the impact of GPU acceleration by benchmarking the performance of HeteroSpark and vanilla Spark with varying CPU and GPU configurations. The benchmarks are briefly described as follows.

Logistic Regression (LR) is a used to estimate the parameters of a qualitative response model. In our experiments, we use Criteo Lab provided Kaggle Display Advertising Challenge Dataset [21] (11 GB training and 1.5 GB testing)to benchmark LR. This dataset represent many click prediction tasks in industry.

K-Means is used for cluster analysis in data mining. We benchmark K-Means for image classification by applying MNIST-8M [22] dataset. The training dataset of MNIST is a collection of hand-written digits which has 8 million of 28x28 digits labeled 0-9 with the size of 24.8 GB.

Word2Vec is a preprocessing stage in most natural language processing applications to prepare raw text into vector format. We use the latest Wikipedia page article dataset [23] (11.8 GB) to benchmark our implementation.

B. System Setup

HeteroSpark has been tested on Amazon EC2 instances with various cluster setups. We choose to use m3.xlarge instances for the pure CPU cluster, and use g2.2xlarge instances for CPU/GPU heterogeneous cluster. CPU type in both m3.xlarge and g2.2xlarge is Intel Xeon E5-2670 v2, which runs at 2.6 GHz of frequency with 8 cores per processor. We scale up/down the number of CPU cores and the number of GPUs to show the performance of HeteroSpark versus Spark.

C. Experiments

We compare the performance of HeteroSpark versus Original Spark with various cluster configurations. In Spark, we use a 32 core (8 nodes) cluster as the baseline setup, and increase the number of cores to 64 and 128 to see how our applications perform when cluster scales up. In HeteroSpark, we use "N CPU cores: M GPUs" configuration, where N and M stand for number of CPU cores and number of GPUs separately. We show the normalized experimental results in Figure 5.

We observe that HeteroSpark platform with the configuration of "8 CPU cores: 2 GPUs" achieves comparable performance with the"128 cores: no GPU" Spark configuration. This demonstrates the performance advantages of GPU accelerations. If we continue scaling up the heterogeneous cluster size to "32 CPU cores: 8 GPUs", we will see a speedup as much as 18.6x comparing to "32 CPU cores: no GPU". We can see a 9.0x and 4.0x speedup comparing with the "64 CPU cores: no GPU" and "128 CPU cores: no GPU" setup. While the improvement on performance is not surprising because of GPU acceleration, it is worthy noting that the applications gain such benefits without sacrificing the programmability and portability. In addition, the HeteroSpark middleware enables

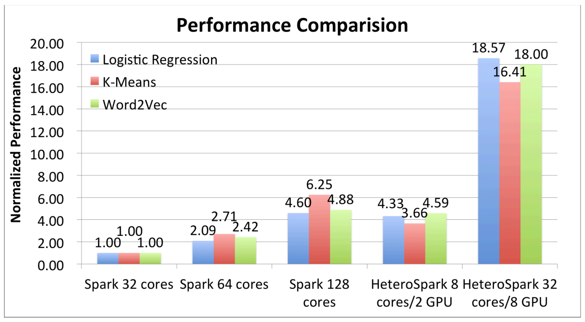


Fig. 5. Performance of HeteroSpark v.s. Spark

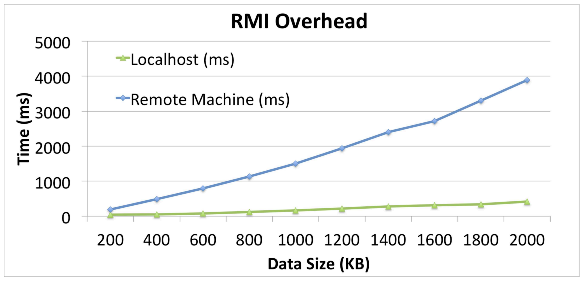


Fig. 6. HeteroSpark RMI Overhead

transparent acceleration as HeteroSpark manages the GPU resources and dispatches workloads.

In order to understand RMI overhead, we test the round-trip time of our RMI interface with increasing size of the input data in Figure 6. As shown in this figure, both locally and remotely connected accelerators increases RMI overhead linearly. RMI overhead is less than 500 ms with 2 MB of data if accelerator is locally connected. However, overhead of the remote connection is much higher than that of the local connection. Thus in most design cases we suggest developers to use local GPUs which are connected via PICe to obtain the best performance.

V. Conclusion

In this paper, we have developed a heterogeneous CPU/GPU Spark platform for accelerating machine learning algorithms. We achieve the major goals in our design: acceleration, flexibility, portability and transparency.

HeteroSpark is a prototype under active development. In our future work, we plan to grow HeteroSpark with an increasing number of machine learning and deep learning algorithms accelerated on GPUs and faster serialization techniques for the data communication between CPU and GPU.

Acknowledgment

This work is supported in part by National Science Foundation (ACI-1440737, ACI-1450996, IIP-1530989).

References

[1] Y. Yang, G. Shu, and M. Shah, "Semi-supervised learning of feature hierarchies for object detection in a video," in *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, June 2013, pp. 1650–1657.

[2] M. Zhao, J. Yagnik, H. Adam, and D. Bau, "Large scale learning and recognition of faces in web videos," in *Automatic Face Gesture Recognition, 2008. FG '08. 8th IEEE International Conference on*, Sept 2008, pp. 1–7.

[3] A. Khalid, H. Afzal, and S. Aftab, "Balancing scalability, performance and fault tolerance for structured data (bspf)," in *Advanced Communication Technology (ICACT), 2014 16th International Conference on*, Feb 2014, pp. 725–732.

[4] L. Bhatia and S. Prasad, "Building a distributed generic recommender using scalable data mining library," in *Computational Intelligence Communication Technology (CICT), 2015 IEEE International Conference on*, Feb 2015, pp. 98–102.

[5] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng, "Building high-level features using large scale unsupervised learning," in *International Conference in Machine Learning*, 2012.

[6] G. Hinton, "A practical guide to training restricted boltzmann machines," in *Neural Networks: Tricks of the Trade*, ser. Lecture Notes in Computer Science, G. Montavon, G. Orr, and K.-R. Mller, Eds. Springer Berlin Heidelberg, 2012, vol. 7700, pp. 599–619. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35289-8_32

[7] X.-W. Chen and X. Lin, "Big data deep learning: Challenges and perspectives," *Access, IEEE*, vol. 2, pp. 514–525, 2014.

[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[10] "Hadoop project host." [Online]. Available: https://hadoop.apache.org/

[11] K. van de Sande, T. Gevers, and C. Snoek, "Empowering visual categorization with the gpu," *Multimedia, IEEE Transactions on*, vol. 13, no. 1, pp. 60–70, Feb 2011.

[12] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 260–269. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454152

[13] B. Catanzaro, N. Sundaram, and K. Keutzer, "A map reduce framework for programming graphics processors," in *In Workshop on Software Tools for MultiCore Systems*, 2008.

[14] A. Mooley, K. Murthy, and H. Singh, "DisMaRC: A Distributed Map Reduce framework on CUDA," 2008.

[15] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU clusters," pp. 1068–1079, 2011.

[16] S. Okur, C. Radoi, and Y. Lin, "Hadoop + Aparapi: Making heterogenous MapReduce programming easier," 2012.

[17] "Aparapi project host." [Online]. Available: https://code.google.com/p/aparapi/

[18] M. Grossman, M. Breternitz, and V. Sarkar, "HadoopCL: MapReduce on Distributed Heterogeneous Platforms through Seamless Integration of Hadoop and OpenCL," *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pp. 1918–1927, 2013.

[19] 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*, ser. KDD '13. New York, NY, USA: ACM, 2013, pp. 95–103. [Online]. Available: http://doi.acm.org/10.1145/2487575.2487677

[20] I. El-Helw, R. Hofman, and H. E. Bal, "Scaling MapReduce Vertically and Horizontally," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, pp. 525–535.

[21] "Kaggle display advertising challenge dataset." [Online]. Available: http://labs.criteo.com/downloads/2014-kaggle-display-advertising-challenge-dataset/

[22] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[23] "Wikipedia downloads page." [Online]. Available: https://dumps.wikimedia.org/