Stochastic Gradient Descent on Modern Hardware: Multi-core CPU or GPU? Synchronous or Asynchronous?

Yujing Ma
University of California Merced
yma33@ucmerced.edu

Florin Rusu
University of California Merced
frusu@ucmerced.edu

Martin Torres
University of California Merced
mtorres58@ucmerced.edu

***Abstract*—There is an increased interest in building data analytics frameworks with advanced algebraic capabilities both in industry and academia. Many of these frameworks, e.g., TensorFlow, implement their compute-intensive primitives in two flavors—as multi-thread routines for multi-core CPUs and as highly-parallel kernels executed on GPU. Stochastic gradient descent (SGD) is the most popular optimization method for model training implemented extensively on modern data analytics platforms. While the data-intensive properties of SGD are well-known, there is an intense debate on which of the many SGD variants is better in practice. In this paper, we perform a comprehensive experimental study of parallel SGD for training machine learning models. We consider the impact of three factors – computing architecture (multi-core CPU or GPU), synchronous or asynchronous model updates, and data sparsity – on three measures—hardware efficiency, statistical efficiency, and time to convergence. We draw several interesting findings from our experiments with logistic regression (LR), support vector machines (SVM), and deep neural nets (MLP) on five real datasets. As expected, GPU always outperforms parallel CPU for synchronous SGD. The gap is, however, only 2-5X for simple models, and below 7X even for fully-connected deep nets. For asynchronous SGD, CPU is undoubtedly the optimal solution, outperforming GPU in time to convergence even when the GPU has a speedup of 10X or more. The choice between synchronous GPU and asynchronous CPU is not straightforward and depends on the task and the characteristics of the data. Thus, CPU should not be easily discarded for machine learning workloads. We hope that our insights provide a useful guide for applying parallel SGD in practice and – more importantly – choosing the appropriate computing architecture.**

I. INTRODUCTION

Stochastic gradient descent (SGD) is the most popular optimization method to train analytics models, e.g., the back-propagation algorithm for deep neural networks [4], in a wide variety of application domains ranging from image [15] and speech [12] recognition to finance [8]. SGD is implemented in a form or another by every modern analytics system, including Google's Brain [16], Microsoft's Project Adam [36] and Vowpal Wabbit [1], IBM's SystemML [2], Pivotal's MADlib [17], and Spark's MLlib [10]. Since these billion-dollar enterprises depend on processes which rely on SGD, it is important to understand its optimal behavior and limitations on modern computing architectures.

Motivation. Over the past decade, CPU design has been moving towards highly-parallel architectures with tens of cores on a die. The culmination of this trend is best exemplified by the current Graphics Processing Units (GPU) having thousands of cores[1]. GPUs are assumed to be the ideal platform for analytics model training due to the compute-intensive nature of the task. This is exemplified by the extensive GPU support across many analytics frameworks, e.g., Caffe[2], TensorFlow[3], MXNet[4], BIDMach[5], SINGA [37], Theano[6], and Torch[7]. Published results that compare CPU and GPU implementations, however, do not support the assumption that GPU is always superior [16], [22], [24]. Quite the opposite, it is often the case that the CPU optimizer outperforms the GPU implementation, even though the degree of parallelism is much lower. A possible reason is the choice of the SGD algorithm. The asynchronous Hogwild-family of algorithms [7], [11], [26], [31], [33], [40] are the preferred SGD implementation on multi-core CPUs due to their simplicity – the parallel code is identical to the serial one, without any synchronization primitives – and near-linear scaling across a variety of analytics tasks [9], [23], [32]. The SGD solutions on GPU resort to a synchronous implementation in which only the highly-optimized linear algebra kernels are offloaded to the GPU. The reasons behind this strategy are the original role GPUs had as accelerators for certain classes of computations and the intricate data access pattern incurred by asynchronous execution. The algorithmic difference in model update strategy – which is an open debate both in theoretical circles [5], [34] and practice [24] – makes a direct comparison between SGD on CPU and GPU challenging. As far as we know, there is no work that performs an in-depth comparison across architectures and SGD algorithms. Given its central role in analytics, we believe it is imperative to identify which SGD algorithm performs better on which architecture and what data.

Problem. We briefly present the setup for model training using SGD. The input data is a matrix in $\mathbb{R}^{N \times d}$ containing N d-dimensional training examples. The goal is to find a d-dimensional vector that minimizes the (convex) loss function over the examples specific to each model. SGD makes several complete passes over the input data and updates the model one or several times in each pass. SGD performance is measured by the time it takes to reach the loss function minimum.

[1] https://en.wikipedia.org/wiki/Nvidia_Tesla
[2] http://caffe.berkeleyvision.org/
[3] https://www.tensorflow.org/
[4] https://mxnet.incubator.apache.org/
[5] https://github.com/BIDData/BIDMach
[6] https://github.com/Theano/Theano
[7] http://torch.ch/

This depends both on the number of passes over the data and the time per pass. In parallel SGD, the input data is partitioned across threads which share a single common model. While this reduces the time per pass, it has the potential to increase the number of passes. The overall effect depends on several factors—parallelization strategy, model update, data characteristics, and loss function.

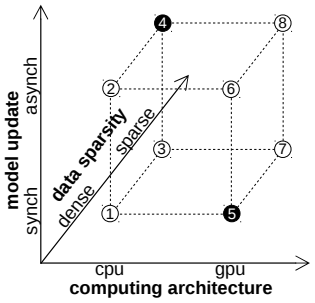

Fig. 1: Exploratory axes.

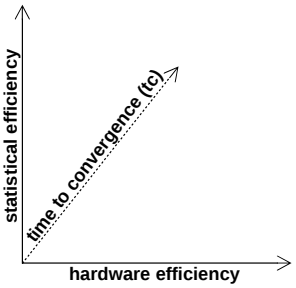

Fig. 2: Performance axes.

In this paper, we perform the first comprehensive experimental study of parallel SGD that investigates the combined impact of three axes – computing architecture, model update strategy, and data sparsity (Fig. 1) – on three measures—hardware efficiency, statistical efficiency, and overall time to convergence (Fig. 2).

Exploratory axes. On the *computing architecture* axis, we consider multi-core CPUs with Non-Uniform Memory Access (NUMA) and many-core GPUs with wide Single Instruction Multiple Data (SIMD) processing units. The specific representatives of the two architectures we use in our work are a dual-socket machine with two 14-core 28-thread Intel Xeon E5-2660 v4 CPUs (56 threads overall, 256 GB memory) and an NVIDIA Tesla K80 GPU with 2496 cores, a 32-wide SIMD unit, and 24 GB memory. The *model update* strategies we consider are synchronous and asynchronous. Synchronous updates follow a transactional semantics and allow a single thread to update the model. While this strategy limits the range of parallel execution inside the SGD algorithm, it is suitable for batch-oriented high-throughput GPU processing. In the asynchronous strategy, multiple threads update the model concurrently. Our focus is on the Hogwild algorithm which ignores any synchronization to the shared model. *Data sparsity* represents the third axis. At one extreme, we have dense data in which there is a non-zero entry for each feature in every training example. This allows for a complete dense 2-D matrix representation. When the model is large, it is often the case that the examples have only a few non-zero features. A sparse matrix format, e.g., Compressed Sparse Row (CSR), is the only alternative that fits in memory. Out of the eight possible combinations, a limited set is implemented in practice—the dark circles in Fig. 1. The GPU solutions implement synchronous model updates over dense data, while the CPU implementations use asynchronous Hogwild which is suited for sparse data. We explore the complete space and map the remaining combinations experimentally.

Performance axes. The *hardware efficiency* measures the average time to do a complete pass – or iteration – over the training examples. Ideally, the larger the number of physical threads, the shorter an iteration takes since each thread has less data to work on—thus, higher hardware efficiency. In practice, though, this holds only when there is no interaction between threads—even then, the size and location of data can be limiting factors. In asynchronous SGD, however, the model is shared by all (or a group of) the threads. This poses a difficult challenge both in the CPU and GPU case. For CPU, the implicit cache coherency mechanism across cores can decrease the hardware efficiency dramatically. Non-coalesced memory accesses inside a SIMD unit have the same effect on GPU. The *statistical efficiency* measures the number of passes over the data until a certain value of the loss function is achieved, e.g., within 1% of the minimum. This number is architecture-independent for synchronous model updates which are executed at the end of each pass. In the case of asynchronous model updates during a data pass, however, the number and order of updates may have a negative impact on the statistical efficiency. The third performance axis is represented by the *time to convergence*. This is, essentially, the product between the hardware and statistical efficiency. The reason we include it as an independent axis is because there are situations when two algorithms have reversed hardware and statistical efficiency – algorithm A has better hardware efficiency than algorithm B and worse statistical efficiency – and only the time to convergence allows for a full comparison.

Summary of results. We organize the results based on the components of the exploratory axes. In the following, we present only the main findings, while we discuss the details in the experimental evaluation (Section IV) and in the extended version of the paper [25]:

- For synchronous SGD, GPU is always faster than parallel CPU in time per iteration and, thus, in time to convergence. The gap between GPU and parallel CPU is larger for complex deep nets models (≈5X on average), while super-linear speedup (>400X) is achieved over sequential CPU for LR and SVM. These results are on par or better than the synchronous solutions from TensorFlow and BIDMach. However, the speedup is nowhere close to the degree of parallelism gap between CPU and GPU.
- Asynchronous SGD on CPU always outperforms GPU in time to convergence, even when GPU has a speedup larger than 10X in hardware efficiency. The gap between CPU and GPU is higher than 5X on sparse data and deep nets.
- While GPU is the optimal architecture for synchronous SGD and CPU is optimal for asynchronous SGD, choosing the better of synchronous GPU and asynchronous CPU is task- and dataset-dependent. Thus, CPU should not be easily discarded. From a financial perspective, though, GPUs are likely the more cost-effective alternative.

II. COMPUTING ARCHITECTURES

We briefly present the two computing architectures considered in this work—multi-core NUMA CPU and GPU.

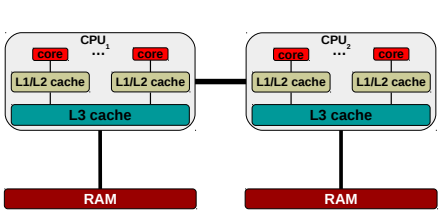

Fig. 3: NUMA CPU architecture.

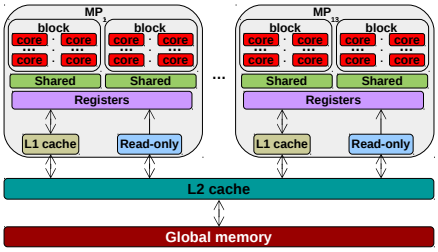

Fig. 4: GPU architecture.

	NUMA	GPU
CPU/MP	2	13
cores	14 per CPU	192 per MP
blocks	-	16 per MP
threads	28 per CPU	2048 per MP
L1 cache	32+32 KB	48 KB
L2 cache	256 KB	1.5 MB
L3 /shared	35 MB	48 KB
RAM/global	256 GB	12 GB

Fig. 5: Hardware specification.

NUMA CPU. The architecture of a NUMA machine is depicted in Fig. 3. It consists of several nodes which contain multiple cores and processor caches. The L1 and L2 caches are associated with each core, while the L3 cache is shared across all the cores in a node. Each node is directly connected to a region of the DRAM memory. NUMA nodes are connected to each other by high-bandwidth interconnects on the main board. To access DRAM regions of other nodes, data is transferred over these interconnects. However, this is slower than accessing the locally-associated memory. Cache-coherency is implicit on NUMA machines and is implemented in hardware. In the worst case, the coherency protocol requires transfer across nodes which can generate congestion on the interconnect and, thus, significantly reduce the speedup of parallel solutions.

GPU. As illustrated in Fig. 4, a GPU contains multiple streaming multiprocessors (MP). Each MP consists of a large number of specialized cores targeted at a limited subset of instructions. In the CUDA programming model, work is issued to the GPU in the form of a function, referred to as the kernel. A logical instance of the kernel is called a thread. The kernel code is parametrized by a logical thread identifier that allows each thread to operate on a different partition of the input data. Since thousands of threads can be executed concurrently across MPs, global thread synchronization is not available. Nonetheless, synchronization can be enforced at thread block level. Physically, all the threads in a block must reside on the same MP. To manage thousands of concurrent threads running on different parts of the data, the MP employs SIMT (single-instruction, multiple-thread) or SIMD (single-instruction, multiple-data) parallelism by grouping consecutive threads of a block into a warp. The MP issues instructions at warp level in vector-like fashion for all the threads in the warp at a time. Threads can access the various units of the deep memory hierarchy in Fig. 4 explicitly – in the code – during execution. Global memory (or device RAM memory) is persistent over multiple kernel invocations and can be accessed from all the threads across MPs. While the largest in size, global memory has the highest latency and lowest bandwidth. Shared memory is a low-latency high-bandwidth memory available to all the threads within a thread block. The read-only constant texture memory (or scratchpad memory) is a read-only cache populated from global memory. It is accessible by all the threads on an MP. The two levels of cache L1 and L2 are used to improve the latency to the global memory. L1 cache handles only local thread memory and does not cache global memory loads. As a result, there is no cache coherency implemented across MPs. When a global memory address is requested by a thread of a warp, aligned successive addresses are converted into a single memory transaction which is called memory coalescing. To move data efficiently from global memory, the threads in a warp have to access consecutive global memory addresses. If the requested addresses of the warp are sparse or unaligned, several memory transactions are required to support the warp computations. Until all the requested data are cached in L2, the warp cannot be scheduled for computation.

Fig. 5 gives the hardware specifications of the NUMA machine and the NVIDIA Tesla K80 GPU used in this paper. While the number of cores and threads is much larger for the GPU, the numbers for the NUMA machine are quite high compared to previous CPU generations, e.g., 56 independent threads can run concurrently on a single machine. Although the amount of memory available on the CPU is 20X larger than on the GPU, the L2 cache on the GPU is 6X larger. This reflects the throughput emphasis of the GPU memory hierarchy as opposed to the latency optimization for CPU.

III. STOCHASTIC GRADIENT DESCENT

SGD is an iterative optimization algorithm for machine learning model training defined by an objective function of the form $\Lambda(\vec{w}) = min_{w \in \mathbb{R}^d} \sum_{i=1}^{N} f(\vec{w}; \vec{x_i}, y_i)$ in which a d-dimensional vector $\vec{w}$, $d \geq 1$, i.e., the model, has to be found such that the objective function is minimized. The constants $\vec{x_i}$ and y_i, $1 \leq i \leq N$, correspond to the feature vector of the i$^{\text{th}}$ data example and its scalar label, while f is the loss. SGD starts from an arbitrary model which is updated iteratively based on a batch of B random training examples $\vec{X}_k$ and their corresponding scalar labels $\vec{Y}_k$. The updated model is computed by moving along the opposite direction of the loss function gradient $\overrightarrow{\nabla f}$. The gradient is a d-dimensional vector consisting of entries given by the partial derivative with respect to each dimension, i.e., $\overrightarrow{\nabla f}(\vec{w}) = \left[\frac{\partial f(\vec{w})}{\partial w_1}, \ldots, \frac{\partial f(\vec{w})}{\partial w_d}\right]$. The step size α, the batch size B, and the number of epochs t are

Algorithm 1 Stochastic Gradient Descent (SGD)

Require:
Training examples $\vec{\vec{X}} \in \mathbb{R}^{N \times d}$ and their labels $\vec{Y} \in \mathbb{R}^N$
Loss function f and its gradient $\overrightarrow{\nabla f}$
Initial model $\vec{w} \in \mathbb{R}^d$ and step size $\alpha \in \mathbb{R}$
Number of epochs t and batch size B

1. **for** $k = 1$ **to** t **do**
   OPTIMIZATION EPOCH
2. Select a random subset of B examples $\vec{\vec{X}}_k = \{\vec{x}_{i_1}, \ldots, \vec{x}_{i_B}\}$ and their labels $\vec{Y}_k = \{y_{i_1}, \ldots, y_{i_B}\}$
3. Compute gradient estimate: $\vec{g} \leftarrow \sum_{\vec{\vec{X}}_k, \vec{Y}_k} \overrightarrow{\nabla f}(\vec{w})$
4. Update model: $\vec{w} \leftarrow \vec{w} - \alpha\vec{g}$
5. **end for**
6. **return** $\vec{w}$

Algorithm 2 Batch SGD Optimization Epoch

1. Compute gradient:
   for $i = 1$ **to** N **do** $\vec{g} \leftarrow \vec{g} + \overrightarrow{\nabla f}(\vec{w}; \vec{x_i}, y_i)$
2. Update model: $\vec{w} \leftarrow \vec{w} - \alpha\vec{g}$

Algorithm 3 Incremental SGD Optimization Epoch

1. **for** $i = 1$ **to** N **do**
2. Compute gradient estimate: $\vec{g} \leftarrow \overrightarrow{\nabla f}(\vec{w}; \vec{x_i}, y_i)$
3. Update model: $\vec{w} \leftarrow \vec{w} - \alpha\vec{g}$
4. **end for**

parameters specific to SGD, known as hyper-parameters of the model training problem.

Depending on the value of B, SGD can be classified into incremental ($B = 1$), mini-batch ($1 < B < N$), and batch ($B = N$). Incremental and batch SGD allow for sequential access to the training examples which is orders of magnitude more efficient. Thus, the optimization epoch becomes a linear scan over the training dataset in which the gradient is incrementally computed (batch SGD) and the model updated (incremental SGD). The differences between the two algorithms are in how the gradient is computed (estimated) and how many times the model is updated. In batch SGD, the gradient is computed exactly using all the N examples in the training dataset and the model is updated only once per epoch. Incremental SGD is at the other extreme. The gradient is approximated using a single example and the model is also updated for every example—N times per epoch. While identical from a computational perspective, the two algorithms are significantly different in terms of convergence. It is a well-known fact that incremental SGD has a convergence rate as much as N times faster than batch SGD for large N, when far from the minimum [3].

A. Synchronous Parallel SGD

The implementation of synchronous SGD consists of a sequence of primitive linear algebra function invocations for gradient computation and model update in `Batch SGD Optimization Epoch`. Each of these functions is blocking. This introduces a clear boundary between gradient computation and model update, essentially synchronizing access to the model. Parallelism is confined exclusively to intra-function processing. This allows for a variety of implementations, as long as the function API is preserved. ML frameworks, e.g., TensorFlow and BIDMach, capitalize on this abstraction and implement the linear algebra primitives with a unified API for both CPUs and GPUs. The benefit of this approach is that switching between architectures does not require any code modification. Our synchronous SGD implementation follows the common API approach. We use the ViennaCL library[8] which implements a full range of linear algebra primitives, has support for multi-thread CPU and GPU, and for dense and sparse data. Since the ViennaCL implementations use all the specific architectural optimizations and are among the fastest available, we do not have to apply further intra-primitive optimizations. For a specific configuration, e.g., CPU or GPU, all the primitives are executed on that device.

B. Asynchronous Parallel SGD

Asynchronous SGD consists of a single function that implements the `Incremental SGD Optimization Epoch`. For each training example, this function first computes the gradient and immediately applies it to a model update. Parallelism is achieved by executing several instances of the function, i.e., threads, concurrently over partitions of the examples. Thus, multiple model updates are executed concurrently—without synchronization primitives, e.g., mutexes or locks. Moreover, access to the model in gradient computation can also interfere with the update—or a stale model is used. Hogwild [11], [26], [27], [31], [33], [40] is the most representative algorithm in this category. It is the exact `Incremental SGD Optimization Epoch` with the iterations of the loop executed in parallel. This makes the parallel implementation of Hogwild very simple—a single directive has to be added in OpenMP[9]. Hogwild prioritizes hardware over statistical efficiency and achieves better time to convergence in many scenarios. For this to be the case, though, a naive implementation is not sufficient [33]. The architectural optimizations of the hardware platform have to be carefully considered. In DimmWitted [40], Zhang and Re give a Hogwild implementation optimized for NUMA CPU architectures. We adopt this implementation in our work and extend it to GPU architectures by considering the interaction between data access path, and model and data replication. The GPU optimizations we introduce are quite different because

[8] http://viennacl.sourceforge.net/
[9] http://www.openmp.org/

dataset	#examples	#features	#nnz/exp (avg)	size (s/d)	LR & SVM sparsity (%)	MLP sparsity (%)	MLP architecture
covtype	581,012	54	54 to 54 (54)	– / 485MB	100	100	54-10-5-2
w8a	64,700	300	0 to 114 (12)	4.4MB / 155MB	3.88	3.88	300-10-5-2
real-sim	72,309	20,958	1 to 3,484 (51)	87MB / 12.1GB	0.25	42.64	50-10-5-2
rcv1	677,399	47,236	4 to 1,224 (73)	1.2GB / 256GB	0.16	64.38	50-10-5-2
news	19,996	1,355,191	1 to 16,423 (455)	134MB / 217GB	0.03	22.50	300-10-5-2

TABLE I: Experimental datasets. "avg" means average number of non-zero (nnz) features per example. "s" is for sparse. "d" is for dense. Sparsity is computed as avg/#features and is given as a percentage. MLP architecture gives the structure of the deep net in terms of number of layers and number of units per layer, e.g., 50 units in the input layer for real-sim.

of the layered parallelism consisting of blocks and threads, the SIMD execution within a warp, and the distinct memory hierarchy optimized for throughput rather than latency. Due to lack of space, the details are included in the extended version of the paper [25].

IV. EXPERIMENTAL EVALUATION

We perform an extended empirical study across the exploratory axes defined in Fig. 1 with respect to the performance axes introduced in Fig. 2. The goal is to fully characterize the relationship between the considered configurations and understand the relevance of each performance measure. We validate our results by comparing against two representative analytics frameworks that have support both for CPU and GPU—TensorFlow and BIDMach. We emphasize that the main objective of the comparison is to add reference points on the performance axes beyond our implementation, while the direct comparison between frameworks and gradient descent algorithms are secondary. Due to space limitation, we include only a subset of the results here, while the complete discussion is available in the extended report [25]. Specifically, our experiments target the following questions:

- What is role of the computing architecture, i.e., CPU or GPU, on the performance of synchronous SGD?
- What is role of the computing architecture, i.e., CPU or GPU, on the performance of asynchronous SGD?
- How do synchronous and asynchronous SGD compare on CPU and GPU separately, and across architectures?
- Are our implementations efficient with respect to TensorFlow and BIDMach?

A. Setup

Implementation. We implement all the 8 configurations in Fig. 1 following the best practices for Intel multi-core CPUs and NVIDIA GPUs, respectively. We use OpenMP for multi-thread programming on the CPU and CUDA 9.1 on the GPU. Synchronous SGD is implemented using the ViennaCL (1.7.1) linear algebra library which provides optimized primitives with the same API for CPU and GPU. This allows us to have identical implementations—only compiled with different flags. We implement our own CPU functions and GPU kernels for asynchronous SGD. We have separate implementations for dense and sparse data that use optimized data structures. All the code is written in `C++`. For the implementations in TensorFlow (0.12.0) and BIDMach (2.0.1), we write only the driver programs which define the objective function corresponding to the analytic model. We then invoke the synchronous SGD optimizer which calls the linear algebra kernels necessary in the gradient computation. While the driver is written in `python` for TensorFlow and `scala` for BIDMach, the linear algebra kernels are coded in `C++/CUDA` and are highly-optimized.

System. The properties of the computing architectures used in the experiments are presented in Fig. 5. They are mounted in the same physical machine running Ubuntu 16.04 SMP with Linux kernel 4.4.0-77 and CUDA 9.1. Out of the two cards inside the Tesla K80 GPU, only one is used in the experiments. The two cards are seen as two independent GPUs by the operating system and have to be programmed independently. This is the default setting both in TensorFlow and BIDMach which require the programmer to specify the GPU on which the SGD optimizer executes.

Methodology. We perform all the experiments at least 10 times and report the average value as the result. Each task is run for at least 10 iterations and the hardware efficiency is measured as the average execution time over the total number of iterations. The time to evaluate the loss is not included in the iteration time. All configurations/systems are initialized with the same model which gives the same initial loss. The SGD step size is chosen by griding its range in powers of 10, e.g., $\{10^{-6}, 10^{-5}, \ldots, 10^{2}\}$, and selecting the value that generates the fastest time to convergence. For end-to-end performance, we measure the wall-clock time it takes for each configuration to converge to a loss that is within 10%, 5%, 2%, and 1% of the optimal loss. Following prior work [40], we obtain the optimal loss by running all configurations for a full day and choosing the lowest. The time to load the data and output the result is not included. Moreover, in the case of GPU, the time to transfer the data and the model to/from the GPU global memory is also not included—we measure only the kernel execution time.

Datasets and tasks. We consider five real datasets (Table I) that exhibit large variety in size, dimensionality, and sparsity. The number of dimensions varies from tens to more than 1 million, while the number of non-zero entries per example is as small as 1 for most of the datasets. In terms of physical size, the sparse representation is as small as 4.4 MB for `w8a` – it can be cached (almost) completely both on CPU and GPU – and as large as 1.2 GB for `rcv1`. In dense format, only `covtype` and `w8a` fit on the GPU, while `rcv1` and `news` cannot be processed even on the CPU. These datasets have been used previously to evaluate the performance of

task	dataset	time-to-convergence (sec)			time-per-iteration (msec)			epochs	speedup	
		gpu	cpu-seq	cpu-par	gpu	cpu-seq	cpu-par		cpu-seq/cpu-par	cpu-par/gpu
LR	covtype	1.05	145.11	1.29	15	2,073	18.42	70	112.54	1.23
	w8a	0.37	148.88	0.46	4.87	1,959	6.05	76	323.80	1.24
	real-sim	3.10	1,537.90	7.67	4.43	2,197	10.96	700	200.46	2.47
	rcv1	31.69	2,227.05	48.06	44.82	3,150	67.98	707	46.34	1.52
	news	0.65	240.21	3.68	6.37	2,355	36.08	102	65.27	5.66
SVM	covtype	10.22	1,344.65	13.50	14.27	1,878	18.85	716	99.63	1.32
	w8a	0.78	342.85	0.80	4.13	1,814	4.23	189	428.84	1.02
	real-sim	0.23	75.59	0.46	6.22	2,043	12.43	37	164.36	2.00
	rcv1	1.13	111.61	2.61	29.74	2,937	68.69	38	42.76	2.31
	news	0.30	98.42	1.69	6.67	2,187	37.56	45	58.23	5.63
MLP	covtype	1,498	19,398	10,009	919	11,908	6,145	1,629	1.94	6.68
	w8a	83.57	909	388	107	1,161	495	783	2.34	4.64
	real-sim	21.99	229	93.98	130	1,365	556	168	2.46	4.26
	rcv1	48.91	1,146	241	1,193	16,960	5,880	41	2.89	4.93
	news	4.03	35.04	16.08	40.23	357	164	98	2.17	4.08

TABLE II: Synchronous SGD performance to 1% convergence error. The best values for each dataset are underlined.

parallel SGD on NUMA CPU [40] and GPU [33]—more details can be found therein. `covtype` is the representative dense dataset throughout the paper since it is complete, while `news` has the highest sparsity ratio of 3×10^{-4}. To prevent the experiments with deep nets, i.e., fully connected multi-layer perceptron (MLP), from not fitting in the GPU memory, we define the number of input neurons as 50 for `real-sim` and `rcv`, and 300 for `w8a` and `news`. The features are grouped and reorganized by averaging the values of hundreds of consecutive features to match the input layer size of the MLP architecture. As a result, most of the data sparsities increase on the transformed datasets. They are listed together with the MLP architectures in Table I. We use a dense format to represent all the transformed sparse datasets when executing MLP in TensorFlow. Thus, synchronous SGD becomes batch gradient descent since it uses the complete dataset to compute the exact gradient. With five datasets and four points on the exploratory axes, we obtain 20 configurations per model. Since we consider three tasks – LR, SVM, and MLP – we have 60 configurations overall. We do not include any regularization in the objective function in order to measure only the time spent in the actual computation.

B. Results

We project the results on a subset of dimensions in the exploratory axes to facilitate a direct comparison between configurations. For synchronous and asynchronous updates taken separately, we compare the CPU and GPU implementations. For each computing architecture, we compare synchronous and asynchronous SGD, and the synchronous solutions in TensorFlow and BIDMach, respectively. To better explore the synchronous and asynchronous SGD between configurations, we include the results for regression tasks based on LR and classification tasks based on SVM and MLP.

Synchronous SGD. Table II contains the time to convergence, and hardware and statistical efficiency results for synchronous SGD implemented with the ViennaCL library. Since the parallel implementations always achieve convergence faster, it is clear that parallelism helps. When comparing the multi-core CPU and GPU solutions, there is a clear trend—*GPU is always faster than parallel CPU in time to convergence*. Since the statistical efficiency is identical in synchronous SGD, this also translates into faster GPU time per iteration, i.e., better hardware efficiency. Given that ViennaCL defines its internal representation for dense and sparse data and implements optimized kernels for CPU and GPU independently, the difference is due exclusively to the computational power of the two architectures—the GPU has more FLOPS than the CPU. The gap between the two architectures increases with the sparsity of the data – to more than 5X on `news` – and with the complexity of the task—MLP always takes longer than LR and SVM. Parallelizing linear algebra operations on sparse data is known to be a difficult task because of the irregular memory access [38]. This turns out to be more acute for the CPU memory hierarchy. We find that ViennaCL takes advantage of the programmability of the GPU memory and exploits it to optimize coalesced access to sparse data. The high variance in statistical efficiency across dataset/task combinations confirms that convergence rate is a property of both the task and the dataset and is independent of sparsity. When comparing the time per iteration, however, we observe similar results on GPU and CPU although their gradients are quite different—matrix batch processing and vectorization hide the higher latency of element-wise operations.

Given that parallel CPU uses 56 threads, we expect a speedup in the range of 56 over sequential CPU. This is the case for `news`. The speedup for `rcv1` is slightly below 56—due to the large size of the dataset which does not allow for efficient caching even in L3. We obtain super-linear speedup on `covtype`, `w8a`, and `real-sim`—on `w8a` the speedup is more than 400X for SVM. The reason for this is the improved cache behavior when all the cores are in use. `w8a` can be entirely cached in L1 due to its small size, while `real-sim` and `covtype` are cached in L2 and L3, respectively. None of these datasets can be cached on a single core for sequential execution. GPU further improves over parallel CPU by 1-5X.

The speedup of parallel over sequential CPU for MLP is around 2X on all of the datasets. This is completely unexpected

task	dataset	time-to-convergence (sec)			time-per-iteration (msec)			epochs			speedup	
		gpu	cpu-seq	cpu-par	gpu	cpu-seq	cpu-par	gpu	cpu-seq	cpu-par	cpu-seq/cpu-par	gpu/cpu-par
LR	covtype	1.97	0.60	1.51	15	150	251	135	4	6	0.60	0.06
	w8a	0.22	0.27	0.18	2.8	15	5.9	80	18	27	2.54	0.47
	real-sim	2.48	1.35	0.52	27	25	8.1	92	54	61	3.09	3.33
	rcv1	18.29	20.37	4.64	226	345	71	81	59	65	4.86	3.18
	news	∞	5.47	∞	65	53	8.7	∞	103	∞	6.09	7.47
SVM	covtype	0.96	0.16	0.35	15	53	77	63	3	4	0.69	0.19
	w8a	∞	0.54	1.89	2.6	2.2	5.6	∞	239	333	0.39	1.18
	real-sim	3.46	1.82	1.28	14	11	7.6	247	164	166	1.45	1.84
	rcv1	10.25	22.71	7.57	94	216	68	109	105	111	3.18	1.38
	news	∞	20.01	1.79	50	47	8.4	∞	425	211	5.60	5.95
MLP	covtype	2,106	6,365	288	6,056	19,058	814	344	334	354	23.42	7.44
	w8a	495	1,284	986	635	1,668	92.61	776	770	10,635	18.01	6.85
	real-sim	140	317	11.14	715	1,925	107	196	165	108	18.04	6.70
	rcv1	352	724	34.47	8,326	17,234	858	42	42	40	20.08	9.70
	news	18.25	47.35	1.12	234	512	34.04	78	91	32	15.06	6.87

TABLE III: Asynchronous SGD performance to 1% convergence error. The best values for each dataset are underlined. ∞ stands for lack of convergence in 300 seconds for LR and SVM, and in 5 hours for MLP, and an unknown number of iterations, e.g., par on `news` does not converge to 1% error within 300 seconds for LR.

because matrix-matrix multiplication is supposed to benefit more from parallelism. At a close inspection, we found that ViennaCL parallelizes matrix product based on the size of the result matrix, which is at most 300x10 for our MLP architectures. Since ViennaCL requires a minimum size that is larger than 5000, there is no parallelism applied to matrix multiplication. As such, the speedup is generated by the other linear algebra operations in the gradient computation. To verify this claim, we plot the speedup for several MLP configurations over `real-sim` in Fig. 6. We observe that, as we increase the size of the deep net, the speedup increases to as much as 26X for a very large net. The reason this is still smaller than 56X is because the input layer cannot be parallelized. Moreover, the GPU speedup over parallel CPU is almost constant—the largest configuration does not fit in the GPU memory.

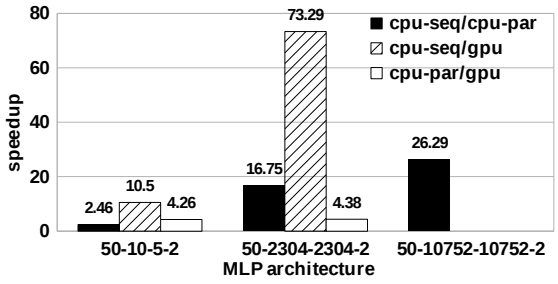

Fig. 6: Speedup on real-sim for different MLP architectures.

Asynchronous SGD. Table III depicts the time to convergence to 1% error, and hardware and statistical efficiency for asynchronous SGD. When comparing the CPU and GPU solutions, there is a clear trend— *(parallel) CPU is (always) faster than GPU in time to convergence*. Specifically, on dense and low-dimensional data, the sequential CPU solution is faster, while on sparse data, parallel CPU dominates. The reason behind this Hogwild behavior on CPU is well-known [33], [40]—concurrent updates to the same features of the model generate cache-coherency conflicts that slow down execution and convergence. Essentially, parallelism is beneficial only on sparse data and models. Since there is no cache coherency mechanism on the GPU, one may expect the GPU solution to be considerably faster due to the higher degree of parallelism. However, the GPU bottleneck turns out to be vectorized execution inside a warp which generates a significant number of model update conflicts. While the warp shuffling optimization reduces the number of conflicts inside a warp, the number of concurrent warps is a lower bound that cannot be overcome. In the case of sparse data, however, update conflicts are not an issue. The problem is the irregular access to the model across the examples inside a warp. First, there is a high variance in the number of non-zero entries—several orders of magnitude. This forces threads to stall while longer examples finish. Second, all accessed model indexes have to be cached before a vectorized instruction can be executed. This incurs a large number of slow memory transactions per instruction. In the best case, parallel CPU achieves a speedup of 6X over sequential CPU on `news` which is consistent with results published in the literature [27], [33]. The best speedup of GPU over parallel CPU is at most 16X on `covtype`—however, CPU still converges faster.

Asynchronous SGD for MLP is executed as mini-batch SGD on a single CPU thread, and as Hogbatch on multiple CPU threads and GPU. We fix the batch size to 512 for all the datasets. Parallel CPU always achieves the highest hardware efficiency as the idle CPU threads are used to process the data batches on small nets concurrently. The speedup of parallel over sequential CPU shows the improvement to be as much as 23X. It is less than 56 because there are threads assigned to parallel element-wise computations in ViennaCL on different data batches. For GPU, although we have 56 CPU threads to set-up the GPU kernels for the data batches, there is only one kernel performing on the GPU at any given time instant. The execution of GPU kernels does not follow the computation order of mini-batch SGD, therefore the GPU implementation

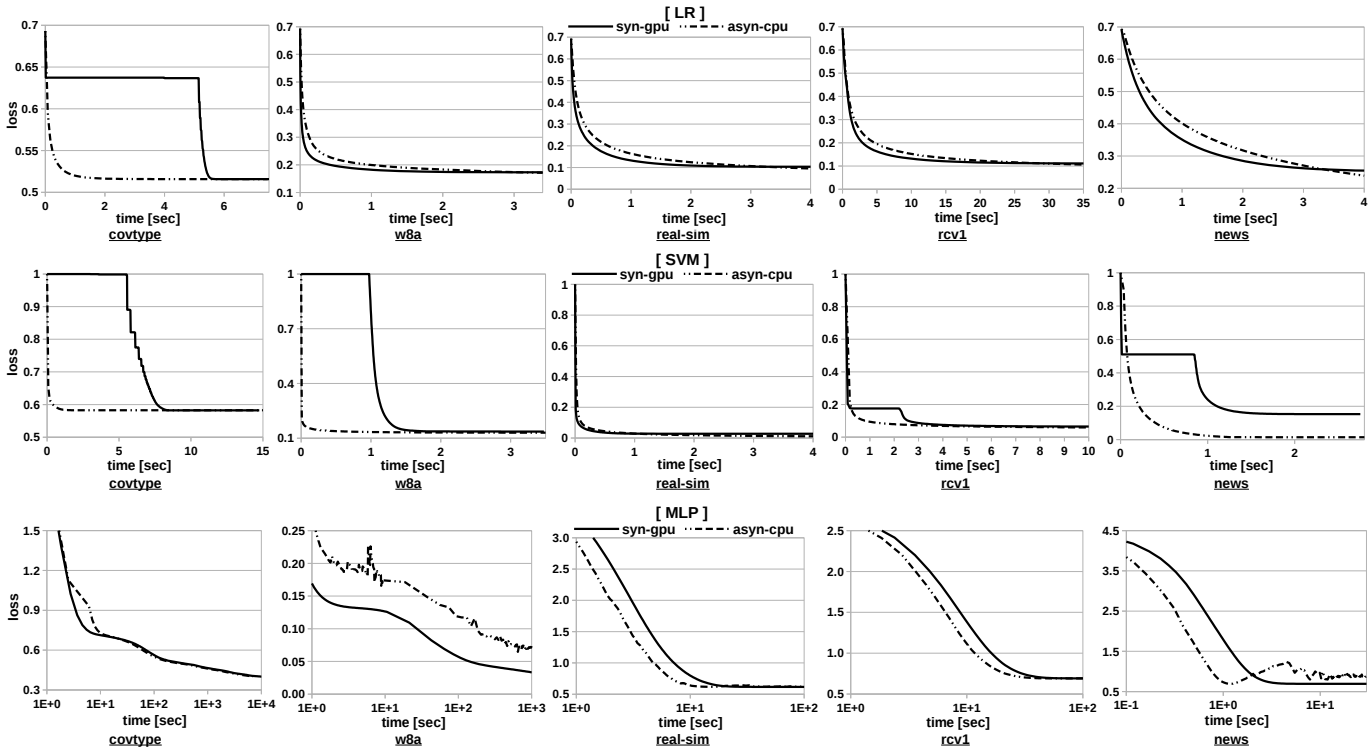

Fig. 7: Time to convergence comparison between synchronous GPU and asynchronous CPU.

can be regarded as Hogbatch with very low concurrency. This is reflected by the almost identical statistical efficiency and only a 2X speedup compared to sequential CPU. `w8a` is the only dataset on which GPU outperforms CPU in time to convergence, even though it has larger time per iteration. This is due to the extensive number of update conflicts over the dense MLP model triggered by a large number of concurrent threads. This phenomenon does not occur on GPU because the updates are sequential. Nonetheless, parallel CPU always outperforms GPU in time per iteration—by 6X or more.

Synchronous vs. asynchronous. We perform a direct comparison in time to convergence only between synchronous GPU and asynchronous CPU—the optimal configurations identified for each model update strategy. We measure the loss as a function of time for exactly the same hyper-parameters and the same initialization conditions. This allows us to isolate the effect of the update strategy while using the optimal computing architecture. The results are depicted in Fig. 7. Synchronous GPU achieves better convergence for certain dataset/task pairs, while asynchronous CPU is better for others. Given that this is essentially a comparison between batch gradient descent – which corresponds to synchronous GPU – and stochastic gradient descent – which corresponds to asynchronous CPU – we do not expect a single winner all the time. As shown previously in the literature [19], the best optimization strategy is particular to the task and the dataset. Our results confirm this finding for parallel optimizers with different model update strategies.

CPU vs. GPU. We compare our SGD implementations on CPU and GPU with the solutions in TensorFlow and BIDMach which support both architectures through simple code settings. In each case, we measure the speedup generated by the GPU implementation over parallel CPU. Since TensorFlow is optimized for dense linear algebra operations specific to deep nets, we provide results only for the MLP models. Similarly, since BIDMach is optimized for generalized linear models on dense and sparse data, we provide results only for LR and SVM. The main point of this experiment is to validate that our parallel implementations are efficient. Moreover, since both TensorFlow and BIDMach support only synchronous SGD, we indirectly compare ViennaCL with the linear algebra kernels in these other frameworks. From Fig. 8, we observe that our implementations provide similar or better speedup than BIDMach for LR and SVM on sparse data. This implies that the ViennaCL GPU kernels for sparse data are superior to those in BIDMach—optimized for dense data. We observe the same trend for the MLP models implemented with TensorFlow (Fig. 9). In this case, we always obtain a superior GPU speedup. Overall, these results prove that our parallel implementations on CPU and GPU have the same relative performance as BIDMach and TensorFlow. The asynchronous SGD speedup is more nuanced and cannot be compared with other solutions because none are available.

C. Summary

We are in the position to provide answers to the questions identified at the beginning of the section:

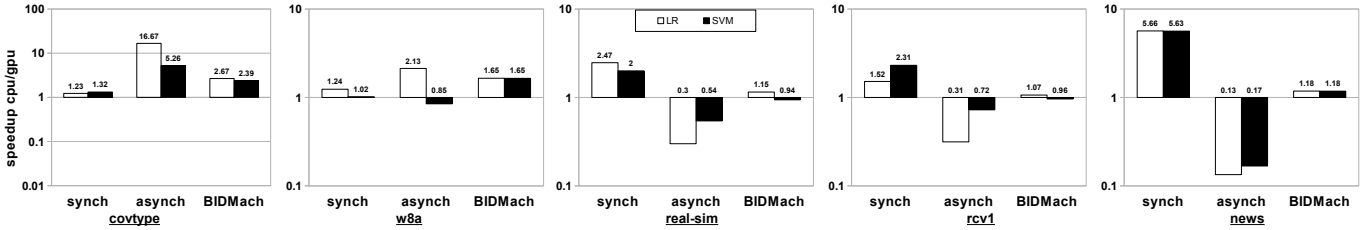

Fig. 8: Speedup in hardware efficiency of GPU over parallel CPU for LR and SVM performed with our synchronous and asynchronous implementations as well as with BIDMach.

- On synchronous SGD, GPU always outperforms parallel CPU in hardware efficiency and, consequently, in time to convergence. For LR and SVM, the difference is minimal for small low-dimensional datasets and increases with dimensionality and sparsity—for a maximum speedup of 5.66X. For MLP, the speedup is at least 4X in all the cases. However, this speedup is nowhere close to the degree of parallelism gap between CPU and GPU.
- On asynchronous SGD, CPU is undoubtedly the optimal solution, outperforming GPU in time to convergence even when the GPU has a speedup of 10X or more. The main reason is the complex interaction between hardware and statistical efficiency under asynchronous parallelism. For MLP, the speedup of parallel CPU over GPU is always 6X or larger—with a maximum of 9.7X on the `rcv1` dataset.
- While GPU is the optimal architecture for synchronous SGD and CPU is optimal for asynchronous SGD, choosing the better of synchronous GPU and asynchronous CPU is task- and dataset-dependent. The choice between these two mirrors the comparison between BGD and SGD.
- Our synchronous SGD implementations provide similar or better speedup than TensorFlow and BIDMach when executed on GPU and parallel CPU. This confirms that parallel CPU should be considered as a competitive alternative for training machine learning models with SGD.

V. RELATED WORK

SGD on CPU. Bismarck [11] and GLADE [29] present methods to implement SGD inside a database. DimmWitted [40] provides a study on how to implement parallel SGD on NUMA architectures. While similar exploratory axes and measure terminology are introduced, the focus on GPU is what distinguishes our paper from DimmWitted. Hogwild [26] performs model updates concurrently and asynchronously without locks. Due to this simplicity – and the near-linear speedup – Hogwild is widely used in many analytics tasks [9], [11], [16], [23], [32], [36]. Hogbatch [33] is an extension to Hogwild that is more scalable to cache-coherent architectures, while Cyclades [27] reduces model update conflicts using graph partitioning. Hogwild extensions to big models based on model partitioning are introduced in [28], [31]. Buckwild [7] is a low-precision variant of Hogwild that represents the data and model with fewer bits. Model averaging [41] is an alternative method to parallelize SGD that is adequate in distributed settings. A detailed experimental comparison of Hogwild and averaging is provided in [30]. The integration of relational join with gradient computation has been studied in [20], [21], [35]. These solutions work only for batch gradient descent (BGD), not SGD. A cost-based optimizer that selects between sequential BGD and SGD is proposed in [19].

SGD on GPU. SGD is supported by all the major deep learning frameworks, including Caffe, TensorFlow, MXNet, BIDMach, SINGA, Theano, and Torch. These frameworks implement optimized kernels for GPU processing. As far as we can tell, all these kernels are for synchronous SGD—there is no Hogwild GPU kernel. As pointed out in [13], since convolutions are the most expensive operation in deep learning, they are the main candidate for offloading on GPU. GeePS [6] implements a distributed parameter server for training across multiple GPUs. Omnivore [14] is an optimizer for deep learning on CPU and GPU that achieves better SGD performance because of careful data partitioning and placement. The asynchronous SGD supported in Omnivore is cross-device, not within the GPU—the case in our work. GPUs are effectively used for querying deep neural networks in NoScope [39]. The work outside deep learning is targeting low-rank matrix factorization for recommender systems. In [18], dynamic scheduling strategies for low-rank matrix factorization on GPU are explored. The problem is modeled as a graph and scheduling is executed for independent subgraphs which do not have update conflicts. cuMF_SGD [39] extends dynamic scheduling with optimized SGD kernels that leverage the GPU cache, warp-shuffle instructions, and low-precision arithmetic. This is the only Hogwild GPU kernel we found in the literature. However, the design space is not explored at all.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we perform a comprehensive experimental study of parallel SGD for training machine learning models. The main value is to map the overall solution space and provide a useful guide for applying parallel SGD in practice. We measure hardware efficiency, statistical efficiency, and time to convergence as a function of the objective function (LR, SVM, and MLP), computing architecture (NUMA CPU and GPU), model update strategy (synchronous and asynchronous), and data sparsity. We draw several interesting insights from our experiments on five real datasets. GPU is always faster than parallel CPU in time per iteration on synchronous SGD

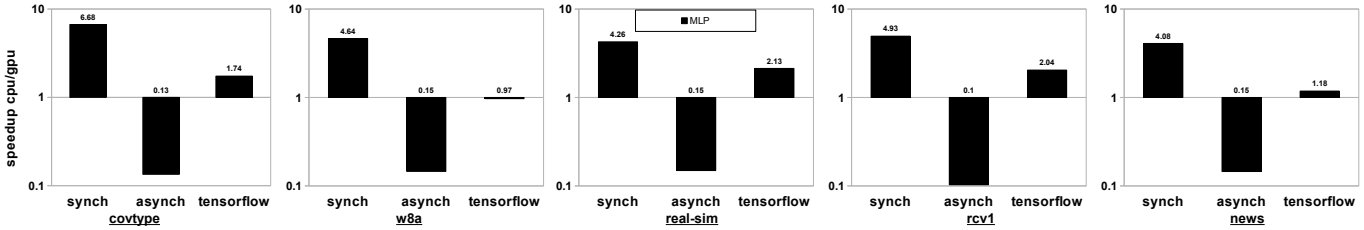

Fig. 9: Speedup in hardware efficiency of GPU over parallel CPU for MLP performed with our synchronous and asynchronous implementations as well as with TensorFlow.

and, thus, in time to convergence. While the gap between GPU and parallel CPU is larger for complex deep nets models (up to 6.7X), the speedup is nowhere close to the degree of parallelism difference between CPU and GPU. Asynchronous SGD on CPU always outperforms GPU in time to convergence, even when GPU has a speedup larger than 10X in hardware efficiency. While GPU is the optimal architecture for synchronous SGD and CPU is optimal for asynchronous SGD, choosing the better of synchronous GPU and asynchronous CPU is task- and dataset-dependent. Thus, CPU should not be easily discarded. In future work, we plan to consider low-precision formats in data representation and study heterogeneous solutions that integrate concurrent processing across CPU and GPU. Moreover, we plan to consider other machine learning models such as matrix factorization and other types of neural nets beyond the fully connected multi-layer perceptron.

REFERENCES

[1] A. Agarwal et al. A Reliable Effective Terascale Linear Learning System. *JMLR*, 15(1), 2014.

[2] A. Ghoting et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE 2011*.

[3] D. P. Bertsekas. Incremental Gradient, Subgradient, and Proximal Methods for Convex Optimization: A Survey. MIT 2010.

[4] L. Bottou. *Neural Networks: Tricks of the Trade*. Springer, 2012.

[5] J. Chen, R. Monga, S. Bengio, and R. Józefowicz. Revisiting Distributed Synchronous SGD. *CoRR*, abs/1604.00981, 2016.

[6] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server. In *EuroSys 2016*.

[7] C. De Sa, M. Feldman, K. Olukotun, and C. Ré. Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent. In *ISCA 2017*.

[8] M. Dixon, D. Klabjan, and J. Bang. Classification-based Financial Markets Prediction using Deep Neural Networks. *CoRR*, abs/1603.08604, 2016.

[9] J. Duchi, M. I. Jordan, and B. McMahan. Estimation, Optimization, and Parallelism When Data Is Sparse. In *NIPS 2013*.

[10] E. Sparks et al. MLI: An API for Distributed Machine Learning. In *ICDM 2013*.

[11] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD 2012*.

[12] G. Hinton et al. Deep Neural Networks for Acoustic Modeling in Speech Recognition. *IEEE Signal Processing Magazine*, 29:82–97, 2012.

[13] S. Hadjis, F. Abuzaid, C. Zhang, and C. Ré. Caffe con Troll: Shallow Ideas to Speed Up Deep Learning. In *DanaC 2015*.

[14] S. Hadjis, C. Zhang, I. Mitliagkas, and C. Ré. Omnivore: An Optimizer for Multi-device Deep Learning on CPUs and GPUs. *CoRR*, abs/1606.04487, 2016.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *CVPR 2016*.

[16] J. Dean et al. Large Scale Distributed Deep Networks. In *NIPS 2012*.

[17] J. Hellerstein et al. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 5(12), 2012.

[18] R. Kaleem, S. Pai, and K. Pingali. Stochastic Gradient Descent on GPUs. In *GPGPU@PPoPP 2015*.

[19] Z. Kaoudi, J.-A. Quian-Ruiz, S. Thirumuruganathan, S. Chawla, and D. Agrawal. A Cost-based Optimizer for Gradient Descent Optimization. In *SIGMOD 2017*.

[20] A. Kumar, J. Naughton, and J. M. Patel. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD 2015*.

[21] A. Kumar, J. Naughton, J. M. Patel, and X. Zhu. To Join or Not to Join? Thinking Twice about Joins before Feature Selection. In *SIGMOD 2016*.

[22] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *ISCA 2010*.

[23] J. Liu, S. Wright, V. Bittorf, and S. Sridhar. An Asynchronous Parallel Stochastic Coordinate Descent Algorithm. In *ICML 2014*.

[24] M. Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI 2016*.

[25] Y. Ma, F. Rusu, and M. Torres. Stochastic Gradient Descent on Highly-Parallel Architectures. *CoRR*, abs/1802.08800, 2018.

[26] F. Niu, B. Recht, C. Ré, and S. J. Wright. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS 2011*.

[27] P. Xinghao, M. Lam et al. CYCLADES: Conflict-free Asynchronous Machine Learning. *CoRR*, abs/1605.09721, 2016.

[28] C. Qin and F. Rusu. Dot-Product Join: Scalable In-Database Linear Algebra for Big Model Analytics. In *SSDBM 2017*.

[29] C. Qin and F. Rusu. Scalable I/O-Bound Parallel Incremental Gradient Descent for Big Data Analytics in GLADE. In *DanaC 2013*.

[30] C. Qin and F. Rusu. Speculative Approximations for Terascale Distributed Gradient Descent Optimization. In *DanaC 2015*.

[31] C. Qin, M. Torres, and F. Rusu. Scalable Asynchronous Gradient Descent Optimization for Out-of-Core Models. *PVLDB*, 10(10):986–997, 2017.

[32] B. Recht, C. Ré, J. Tropp, and V. Bittorf. Factoring Non-Negative Matrices with Linear Programs. In *NIPS 2012*.

[33] S. Sallinen et al. High Performance Parallel Stochastic Gradient Descent in Shared Memory. In *IPDPS 2016*.

[34] S. Zheng et al. Asynchronous Stochastic Gradient Descent with Delay Compensation for Distributed Deep Learning. *CoRR*, abs/1609.08326, 2016.

[35] M. Schleich, D. Olteanu, and R. Ciucanu. Learning Linear Regression Models over Factorized Joins. In *SIGMOD 2016*.

[36] T. Chilimbi, Y. Suzue et al. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI 2014*.

[37] W. Wang, M. Zhang, G. Chen, H. V. Jagadish, B. C. Ooi, and K.-L. Tan. Database Meets Deep Learning: Challenges and Opportunities. *SIGMOD Record*, 45(2), 2016.

[38] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *SC 2007*.

[39] X. Xie, W. Tan, L. L. Fong, and Y. Liang. CuMF_SGD: Parallelized Stochastic Gradient Descent for Matrix Factorization on GPUs. In *HPDC 2017*.

[40] C. Zhang and C. Ré. DimmWitted: A Study of Main-Memory Statistical Analytics. *PVLDB*, 7(12), 2014.

[41] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized Stochastic Gradient Descent. In *NIPS 2010*.