Machine Learning at the Limit

John Canny*†, Huasha Zhao*
* UC Berkeley
Berkeley,
CA 94720, USA
{canny,hzhao}@berkeley.edu

Ye Chen
Dianping.com
Changning District Anhua Rd 492 C-2,
Shanghai, China 200050
ye.chen@dianping.com

Bobby Jaros†
† Yahoo Research
701 First Ave
Sunnyvale, CA, 94089, USA
bjaros@yahoo-inc.com

Jiangchang Mao
Microsoft
1020 Enterprise Way
Sunnyvale, CA 94089, USA
jmao@microsoft.com

Abstract— **Many systems have been developed for machine learning at scale. Performance has steadily improved, but there has been relatively little work on explicitly defining or approaching the limits of performance. In this paper we describe the application of *roofline design*, an approach borrowed from computer architecture, to large-scale machine learning. In roofline design, one exposes ALU, memory, and network limits, and the constraints they imply for algorithms. Using roofline design, we have developed a system called BIDMach which has demonstrated the highest performance to date for many ML problems. On one GPU-accelerated node, it generally outperforms other single-machine toolkits and cluster toolkits running on 100s of nodes. This performance level is enabled by a relatively small number of rooflined matrix primitives. Such performance implies a dramatic reduction in the energy used to perform these calculations. Beyond matrix kernels, roofline design can be applied to the end-to-end design of machine learning algorithms which minimize memory usage to optimize speed. This approach offers a further 2x to 3x gain in performance.**

Roofline design can also be applied to network primitives. We describe recent work on a sparse allreduce primitive called Kylix. We have shown that Kylix approaches the practical network throughput limit for allreduce, a basic primitive for distributed machine learning. Using Kylix, we describe an efficient transformation from model-parallel to data-parallel calculations. This transformation uses a secondary storage roofline, with similar parameters to the network. Finally, we describe several deployments of these techniques on real-world problems in two large internet companies. Once again, single node rooflined design demonstrated substantial gains over alternatives on either single nodes or clusters.

Keywords: Scalable Machine Learning, Big Data, Distributed Systems

I. INTRODUCTION

A great deal of research effort has been devoted to big data analysis, with many successes. However, while there have been many improvements in algorithms, the question of the *practical performance limits* of large-scale machine learning remains largely open. Defining and approaching those limits is the goal of the proposed work. We argue that a systematic approach to defining the performance limits of data analysis leads to *dramatic* improvements. It has already produced a prototype system, BIDMach, which on a single node outperforms all other systems, included distributed toolkits running on clusters (of a hundred nodes or more) for many of the most common learning tasks. BIDMach also includes a distributed computing primitive called Kylix which holds the record for distributed pagerank and is a widely-applicable to other distributed ML tasks.

What kind of performance limits do we mean? The limits (and foci of our work) are:

- Single-machine performance, which includes running time and scalability.
- Cluster performance, for larger datasets and/or distributed models.

In the list above we included scalability for single machines, whereas scalability is often equated (incorrectly) with implementation on a cluster. Scalability consists of two parts:

- Dataset scalability
- Model scalability

Dataset scalability means the ability to process very large large datasets. BIDMach supports dataset scalability by streaming training data from secondary or network storage. Model scalability on single nodes is more challenging, but we present a general solution with modest performance later in the paper.

The approach we take is based on *roofline design*, borrowed from computer architecture [1].. Roofline design begins with quantification of hardware limits (e.g. ALU throughput, memory speed, network speed, I/O speed etc). The goals are similar to efficient algorithm design which aims to match known lower bounds. But asymptotic analysis is not enough for large-data tasks – most scalable algorithms

are O(n) for n input instances and are therefore undifferentiated, while *constant factors make or break a large-scale calculation.* In the post-Moore's-Law world, the constants associated with common operations (retrieving an array element or doing an addition) vary *over five orders of magnitude*. Single-word main memory access take tens of nanoseconds while register operations on a GPU (factoring in parallelism) equate to tenths of a picosecond. This ratio is significantly more than the speedup that is practically feasible by parallelizing on a cluster, and single-machine optimization is clearly more economical in energy, space and capital and running costs. Roofline design leads to a focus on *single-machine optimization first, and cluster computing second*. Our benchmarks on CPUs have shown that most other systems are far from their roofline limits. This suggests that without explicit roofline design, performance tends to be well below the limits. Roofline design applies in very similar fashion to single machines (based on ALU, memory and I/O performance), and to clusters (based on network throughput).

Roofline design applies to any hardware platform, but we have found the largest gains with GPUs. This is not too surprising – while they have shown great promise in image and scientific computing tasks, their potential for machine learning (with the significant exception of deep learning) has not been thoroughly explored. Secondly, the standard matrix toolkit for sparse operations on GPUs, e.g. NVIDIA's Cusparse library, is far from the roofline limit on typical (power law, sparse) machine learning data. This again appears to be a consequence of priority for scientific data in the development of those kernels. By taking a roofline approach and optimizing for power law data, we have written our own sparse matrix kernels, and optimized them close to their theoretical (roofline) limits. This leads to significant speedups across a broad range of ML algorithms.

A. *Contributions*

An early overview of the BIDMach toolkit was presented in [2]. The Kylix system was described in [3]. The approach underlying these systems, i.e. roofline design which we describe here, has not previously been described or applied to scalable machine learning. This paper includes a much more thorough and representative set of benchmarks, and several industrial case studies which validate the performance of the toolkit in real-world settings. We also describe a general approach for model scalability on single machines which is new.

II. Roofline Design

The graphic in Fig. 1 depicts a performance roofline for typical processor hardware today. This graphic focuses on ALU and memory which are the primary determinants of performance for single-machine algorithms.

The y-axis of Fig. 1 shows the potential throughput in arithmetic operations/second. The x-axis is "operational intensity" which is the number of operations applied to each data value (in units of operations per byte). The intensity is much lower for sparse operations – elements referenced by sparse matrix indices in a matrix multiply may be used only once, while dense matrix multiply typically uses each datum many times. The horizontal lines reflect the maximum ALU throughput for each type of processor (the graph is drawn for Intel i7 and NVIDIA GTX-680 processors). GPUs have much higher ALU throughput since the GPU chip area is almost entirely ALU vs. about 10% ALU for a modern CPU. Thus for dense matrix multiply, GPUs are potentially 10x faster and this is roughly the difference observed in practice.

The diagonal lines reflect memory bandwidth. Since bandwidth is flow in bytes/second it defines a linear relationship between the x-axis (flops/byte) and the y-axis (in flops/sec). On a log-log graph, this relationship is always linear with unit slope. From the graph we see that have much higher main-memory bandwidth. This leads to potential order-of-magnitude advantage for sparse as well as dense operations. This is very significant for machine learning on typical data (text, social networks, web data, server logs,..).

Roofline design provides a systematic way to evaluate the potential throughput of an algorithm on the basis of the resources (especially the types of memory) that it uses. The roofline limit then serves as a target for the final performance of the algorithm. While rooflining does not provide guidance on *how* to design an algorithm to approach the limit, knowledge of the limit provides a target to work toward, and it may suggest strategies for improvement.

This simple picture becomes more complex as we consider memory capacity. A GPU has several forms of memory of varying speeds and capacities. A CPU also has other memory types but they are provisioned only as caches, not directly accessible to the programmer, and enticing the

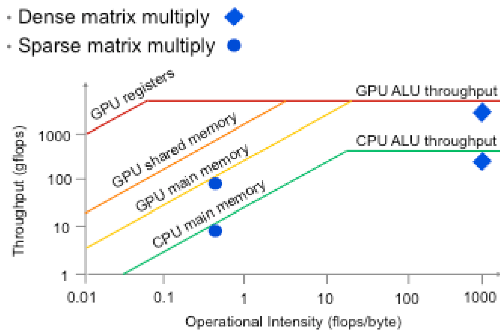

Figure 1. Rooflines for a typical CPU and GPU

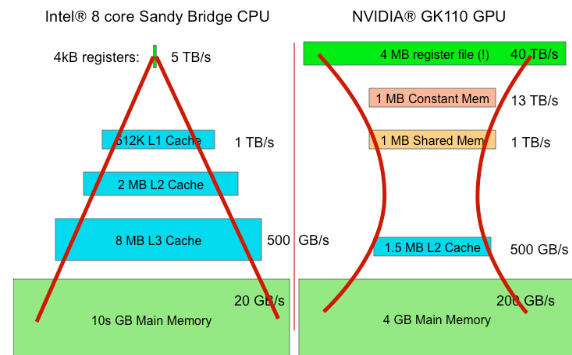

Figure 2. Memory capacities and throughput

CPU to use them can be challenging. The other memory types in a GPU *are* exposed to the programmer, and this makes them applicable to a wider variety of situations.

Figure 2. shows the memory capacities for representative CPUs and GPUs. The CPU memories decrease in capacity sharply as we move up the hierarchy. By contrast, GPU memories exhibit an "hourglass" shape, and there is actually more full-speed register memory than L2 cache. Between these devices, there is a 3-order-of-magnitude gap in register capacity. For problems that allow full use of GPU register memory (example later) performance is extremely high.

TABLE I. ROOFLINES AND OBSERVED PERFORMANCE FOR SPARSE MATRIX PRIMITIVES

Operation	Roofline Limit	Observed Performance
A * S	72 gflops	45 gflops
A * S^T	72 gflops	31 gflops
S ° (A * B)	72 gflops	63 gflops

A simple example of the application of rooflining is given in Table 1. The three most common bottleneck operations in BIDMach are SPMM (Sparse-Dense Matrix Multiply), SPMM with transpose and sparsely-filtered dense-dense multiply, denoted SDDMM(S,A,B) for sparse S and dense A,B. SDDMM(S,A,B) is equivalent to (A * B) ° S where ° is the Hadamard (element-wise) product. The first two arise in algorithms whose loss function has the form

$$L = f(AS)$$

for input data S and a model matrix A. Columns of S are (sparse vector) input instances, and S typically has several columns comprising a minibatch of instances. This formula covers logistic and linear regression and SVM, and causal estimators derived from them. The gradient of the loss has the form

$$dL/dA = f^{'}(AS)\, S^T$$

and to evaluate it we need a dense-sparse multiply for AS and a dense-sparse-transposed multiply for $f'(.)\, S^T$. Factor models such as LDA (Latent Dirichlet Allocation) [LDA], NMF (Non-negative Matrix Factorization) [NMF] and SVD++ (a collaborative filtering algorithm) [SVD++] use a loss that can be written as:

$$L = g_S(A^T B) \quad (1)$$

The matrix factors A and B are dense here, and the loss typically depends only on the values of the product $A^T B$ at the non-zeros in the sparse matrix S. Its derivatives *wrt* the factors A, B are:

$$dL/dA = B\, g_S'(A^T B)^T \quad \text{and} \quad dL/dB = A\, g_S'(A^T B) \quad (2)$$

and evaluation of these gradients requires the three primitives described so far: (i) SDMM(S,A,B), for $g(..)$ and $g'(..)$, (ii) A * S for $A\, g'(...)$ and (iii) A * S^T for $B\, g'(...)^T$.
A naïve, main-memory roofline is 72 gflops for all these operations, based on the following argument: The dense input matrix elements must be read and written once for sufficiently sparse S, and two operations (multiply-add) performed on them, or one operation per word. The coefficients of S are read but used many times and do not affect the roofline. The memory throughput for the K40 device on which these measurements were made is 288 GB/s or 72 Gwords/sec, yielding 72 Gflops.

The third operation is particularly close to its roofline, while the first is about 60% of it. The third operation only reads A and B, and so reads dominate the running time. The other two operations include reads and writes in roughly equal proportion. This likely accounts for the difference in their throughputs.

There is a significant difference between A*S and A*S^T. Although these routines use very similar code, the coefficient ordering in A*S allows output values for a given column to be accumulated in registers (BIDMach uses column-major order). For A*S^T on the other hand, this is not possible (S^T is not ordered by columns) and an atomic main-memory add-write must be used instead. This operation is more expensive than a normal write and accounts for the difference. Roofline design cannot account for all these nuances, but it can reliably determine when a kernel is far from its theoretical limit. Knowing this, the code designer can strive to bring it close, and typically (almost always in our experience) eventually succeed. These operations are the most important

TABLE II PERFORMANCE COMPARISON ON REPRESENTATIVE DATASETS

Systems A/B	Algorithm	Dataset (Size)	Dim	Time (s)	Cost ($)	Energy (KJ)
Spark-72 BIDMach	Logistic Regression	RCV1 (0.5GB)	103	30 14	0.07 0.002	120 3
Spark-128 BIDMach	Logistic Regression	Criteo (12 GB)	1	400 81	1.00 0.01	2500 6
Spark-384 BIDMach	K-Means	MNIST (24GB)	4096	1100 735	9.00 0.12	22000 140
GraphLab-576 BIDMach	Matrix Factorization	Netflix (4 GB)	100	376 90	16 0.015	10,000 20
Yahoo-1000 BIDMach	LDA (Gibbs)	News (100 GB)	1024	220k 300k	40k 100	4E7 6E4

for common ML algorithms, but other routines may cause bottlenecks if their implementation is poor. All the basic matrix operations in BIDMat/BIDMach have been rooflined. In the table below we include some recent benchmarks of BIDMach against some state-of-the-art toolkits. BIDMach's benchmark page includes many single-machine benchmarks, but we focus on cluster benchmarks because the performance gap is less intuitive.

Criteo is a large-scale prediction task from Kaggle. Spark-XX is a Spark cluster with XX cores, and similarly for GraphLab-XXX. Yahoo-1000 is a 1000-node cluster with an unspecified number of cores. BIDMach ran on a single EC2 g2.xlarge instance. The scripts for these benchmarks are all available as part of the BIDMach distribution. We tuned both algorithms in each row for best accuracy. Both algorithms achieved comparable final accuracy with the exception of the CRITEO dataset. We were unable to achieve a ROC AUC higher than 0.62 with Spark, while BIDMach achieved 0.72. This is likely due to Spark's Map-reduce design. Model updates are only done at the end of a full pass over the dataset, whereas BIDMach performed several hundred thousand updates in each pass over the dataset. Costs are based on the cost/time for the recommended Amazon EC2 instances (m3.xlarge for Spark, g2.xlarge for BIDMach and c3.8xlarge for Graphlab).

We note that the running times for BIDMach are very competitive, and generally lower than these mid-to-large sized clusters. Cost is generally two orders of magnitude lower for BIDMach, and the energy consumption ratio is somewhat higher than the cost ratio. For large clusters (last two lines of the table), these gaps widen to approximately three orders of magnitude. The table demonstrates several things from the point of view of roofline design: The other ML toolkits are an order of magnitude roughly from their CPU roofline. BIDMach operates at GPU roofline speeds which is an order of magnitude higher than CPU rooflines, but the remainder of the two-order-of-magnitude gap is attributable to non-rooflined design. The second trend from the table is the decreasing efficiency of cluster systems with number of nodes. The cluster communication primitives used by these systems have not been rooflined and in fact become quite inefficient on large clusters. We take up that point shortly.

A. *End-to-End Rooflined Algorithms: Word2Vec*

The current architecture of BIDMach is based on matrix primitives. Each matrix primitive is individually rooflined, but in the course of an algorithm, there may be several excursions to the same main memory (i.e. model memory) which limit the algorithm's ultimate performance. We recently began exploring designs which are memory-rooflined end-to-end. i.e. we examine the memory needs of an algorithm in its course of execution, and try to approach the theoretical limits implied by them. Multiple memory accesses to the same locations are merged into one. From Fig. 2 earlier we can see that there is an abundance (4MB) of register memory available on the sample GPU which can serve as temporary storage for this purpose. Columns of model memory corresponding to input instance features are cached in register memory for the duration of processing of those instances.

TABLE III: END-TO-END WORD2VEC PERFORMANCE

Processors	Dim	Gflops	Word/s
Titan-x	300	205	8.5M
Titan-x(4)	300	500	20M
Titan-x	600	220	4.5M
Titan-x(4)	600	540	11M

We applied this approach to the Word2Vec algorithm with negative sampling. Word2Vec [4] is a widely-used semantic embedding scheme. The Word2Vec implementation in [4] uses an approximate softmax operation over the entire vocabulary. Two approximations are used, one of them: negative sampling, is generally more accurate and we focused on it for our implementation. Negative sampling involves retrieving model vectors for a large-number of randomly-chosen samples. The memory access times for those vectors sets the performance limit for the algorithm.

Word2Vec updates the model using stochastic gradient updates of the form:

$$\delta v_s = f(u_i\, v_s^T)\, u_i \qquad \text{and} \qquad \delta u_i = f(u_i\, v_s^T)\, v_s \qquad (3)$$

for model vectors u_i and v_s corresponding to an input word i and a random sample word s. From these updates we see that the model vectors are accessed three times: in the inner product $u_i^T\, v_s$, in a scalar/vector multiplication, and finally as the target of an addition operation (when the deltas are added to the base vector values). When implemented using matrix primitives, each of these stages would be separated out as a matrix operation on a minibatch of words (for efficiency). The use of minibatches allows the use of matrix primitives on minibatches which are typically much faster than vector operations. But this also prevents caching of the model vectors for the next operation.

By writing an implementation that processes model vectors end-to-end in GPU code, we can cache u_i and v_s in register memory and reduce the memory bandwidth by a factor of three. We also use sample-sharing technique to reduce memory bandwidth, described in a forthcoming paper. The result is presented in the table III. We see that the throughput on a single NVIDIA Titan-X processor, at 205 gflops, is roughly 3x the throughput of the optimized matrix kernels we described earlier. The highest performance we were able to obtain with a CPU system was 2.2M wd/s or 60 gflops on dual-processor, 18-core Haswell E5-2666. The GPU Word2Vec performance is significantly higher than any other system at this time, and a factor of 5x faster than the original Google implementation (which is well-optimized) on the fastest CPU hardware we were able to test.

Because of the gap between GPU memory speeds and network bandwidth (300GB/s vs. 1 GB/s for 10 Gb Ethernet), and because of its heavy memory requirements, it is difficult to accelerate word2vec on a cluster. The only

implementation we know of (in Spark) uses data parallelism only and achieves 0.2 M word/sec on a 10-node cluster.

The end-to-end implementation of Word2Vec in BIDMach uses custom C (CUDA) code. But note the similarity between (3) and (2). With small modification, the code for (3) can be used for the factor models with structure (2). Primarily, the changes are to the loss function *f()* or *g()*. We are moving to a template code design, with data movement separated from the loss function. The overwhelming majority of work in the design of one of these kernels is in data movement. This work can be done once for a broad family of algorithms. It is likely too that the loss could be written in Scala to simplify code design, using an already-developed Java-to-CUDA compiler. For factor/clustering models, this change reduces memory accesses by 3x, and is likely to produce a 3x improvement in performance. For simpler models following (1), there is a 2x reduction in memory accesses and a likely 2x improvement in overall speed. This should further widen the performance gap evident in table II.

B. *Data-Parallelism for Power-Law data*

We have seen that the throughput of GPU main memory is around 300GB/s while 10 Gbit Ethernet commonly used in production clusters is only about 1 GB/sec. It is very challenging to distribute the algorithm across a cluster. On the other hand, multiple GPUs on a single node communicate over a PCI bus, and can achieve speeds in the 10-30 GB/s range. This is still more than an order of magnitude slower than main memory, so care must be taken to minimize communication. Here we can take advantage of the structure of power-law data. In power-law data, the frequency of occurrence of the feature of rank r (the rank is the position of the feature when sort in descending order of frequency) is proportional to a power of the rank:

$$F \;\alpha\; r^{-p}$$

and for many dataset, the exponent p is very close to 1. An efficient communication pattern for this data should broadcast updates to features in proportion to the frequency of updates. Or put another way, the number of local updates between communication of those updates should be approximately constant. A communication pattern that fits these requirements is shown in Fig. 4.

Fig. 4 shows the size of the blocks of data communicated in successive model updates rounds. The blocks are always powers-of-two multiples of the base block size. A block of size b contains all features of ranks 1...b, i.e. the b most frequent features. As the block size doubles, the update rate for blocks of that size halves. It follows that the update rate for a feature of rank r is roughly (within a factor of 2) inversely proportional to its rank. We implemented this update scheme for the Word2Vec algorithm and ran it on a 4-GPU node. The results are shown in table III. The base block size had to be small – 1000 features, to produce the best speedup. A simple star topology was used for communication with the CPU as the hub.

This pattern is suitable for synchronizing dense power-law data on a small network (i.e. a rack) or on a single node. For more general data, or large networks, there is a scalability problem with simple communication patterns which we take up next.

C. *Rooflined Cluster Allreduce*

Not every calculation can be done in reasonable time on a single node, and models may be too large to fit in the memory of a single machine. For these cases, cluster computing is a cost-effective option. We distinguish cluster computing from the generally more-expensive scientific computing approach. Scientific clusters usually use high-speed non-commodity networking, use fixed group allocation (a specified number of machines for a specified time), and cover the overhead of high or perfect availability (failed nodes are quickly replaced and repaired). Cluster computers are available at much lower, but the application programmer must deal with one or more of (i) dynamic allocation: a variable number of machines in use at any given time (ii) node failures (iii) slower and less predictable network speeds, and unknown (although usually inferable) network topology.

GraphLab/Powergraph, Hadoop, Spark etc., all use direct all-to-all communication (a "shuffle" operation) for distributed model updates (aka an Allreduce operation). That is, every feature has a "home node" and all updates to that feature are forwarded to the home node. The updates are accumulated and then sent to all the nodes who request the new value of that feature. Home nodes are distributed in balanced fashion across the network. Unfortunately this approach is not scalable: as the number of nodes N increases, the packet size for each point-to-point message decreases as (1/N) – assuming fixed data per node. If the total dataset size is fixed, then message size decreases as ($1/N^2$).

Eventually, the time to send each message hits a floor value determined by overhead in the TCP stack and switch latencies. For Amazon EC2 we measured an effective minimum packet size of around 2 MB. Powergraph is particularly susceptible to this floor effect, since it reduces message size by skipping inactive vertex updates [5]. The floor effect was already hampering performance on typical datasets (the Twitter follower graph [5]) on clusters of 64 nodes.

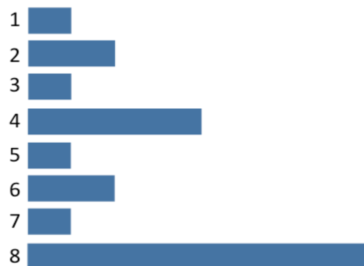

Figure 4. Communication block sizes in successive (numbered) rounds.

To go further we should represent the packet size constraint using a roofline diagram (Fig. 5): The horizontal line represents full network bandwidth, the diagonal line is the throughput roll-off because of packet latency, here around 10 msecs. An all-to-all shuffle network is bandwidth optimal so long as we are on the upper horizontal line, each node sends its data at its full NIC speed and then receives it at the same rate. Our goal is to stay near this roofline in any novel network.

In order to stay near the roofline we developed a *hierarchical butterfly network primitive* called Kylix (Fig. 6). In the first layer of this network, nodes are placed in groups of k_1 nodes, where k_1 is chosen so that we are at the blue point in the figure above. Packet size is inversely proportional the number of nodes k_1 in the group. So this is the largest degree that keeps the network throughput at the roofline. A smaller degree group will require more layers in the network and higher latency. The same principle applies to the next layer which has groups of k_2 nodes etc., and the third. Things become more complicated with sparse data (in a good way), however, and the optimal degrees form a non-increasing sequence.

Kylix uses a *nested, heterogeneous-degree butterfly*. That is, it is a network with m layers, and where layer i is partitioned into groups of size k_i. Data at layer i is partitioned into packets of size proportional to $1/k_i$, and k_i can be chosen to keep the packet size above the floor. Communication within each group uses a round-robin (many-to-many) protocol. This network works particularly well with sparse, power-law data. Since it reduces the data in several layers, collisions between sparse features reduce the total number of features and the total communication needed at each layer. This implies that the optimal layer degree decreases as one moves deeper into the network, and that the butterfly is therefore heterogenous in degree.

An example network is shown in Fig 6. The first (upper) layer degree is 3, the next layer has degree 2. Each column (indexed as Pi) represents a single node, and time is increasing downwards. Outbound values are updates emitted by each node, inbound values are the features requested by each node for the next operation, and are in general different. This approach approves on the best previously-reported

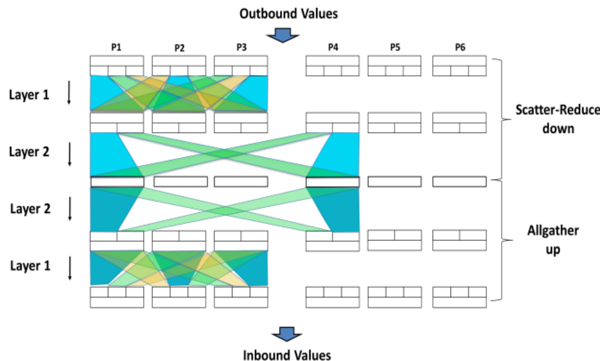

Figure 6. Multi-layer communication in Kylix

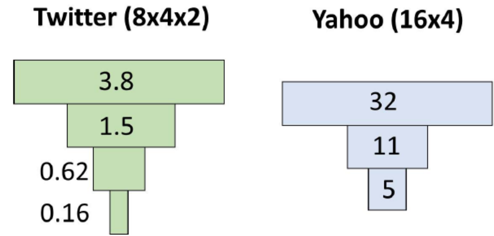

Figure 7. Network communication volume for the two datasets in GB.

performance on Pagerank (by Powergraph) by 5x [6].

Because of the collisions of sparse terms, the amount of communication and the running time decreases down the network layers. It has the discrete "Kylix" shape that gave the approach its name, as illustrated in Fig. 7.

Kylix achieves an aggregate network throughput of 3 Gb/s with 10 Gb/s Amazon EC2 nodes. While this is somewhat lower than the theoretical limit, it was achieved with a standard Java TCP/IP stack (this is close to the maximum speed we measured for back-and-forth TCP/IP messages through the Java stack). In any case, Kylix should scale with the network performance. The most important point is that Kylix uses this bandwidth efficiently. From figure 7, we see that most communication is in the first layer. The first layer comprises first-level shuffles and so represents the lower bound on communication. The additional levels run at the same Gb/s speed but comprise a little more than 50% of the first layer data volume. Thus Kylix performs an allreduce on a large network with near-optimal (about 30% below the bound) throughput. This is better than any competing system and should scale to much larger networks. Performance on the two datasets is illustrated in Fig. 8.

An important aspect of practical commodity cluster systems is fault-tolerance. We implemented a simple fault-tolerance scheme by replicating all the nodes in the network. This roughly doubles the total storage needed and computation time for a fixed number of nodes, but provides a good degree of fault tolerance (both nodes in a replica pair must fail, which should not happen until there are $O(\sqrt{(N)})$

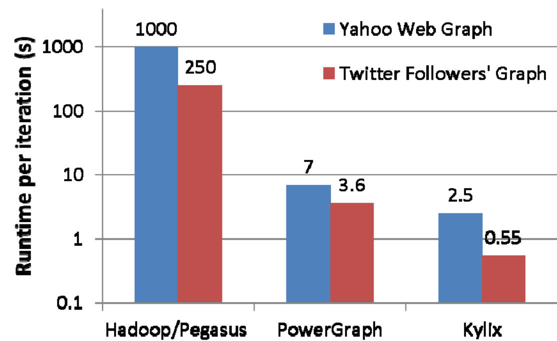

Figure 8. Kylix Pagerank performance

failures). We are currently working on a fault-tolerance scheme where replication only occurs on nodes in layers 2 and below. This approach requires no additional storage, and should have a time overhead much less than 100% (we estimate around 25% for a typical network). Faults will remove one layer-1 group's data from the calculation, but the system will continue with the rest of the data. There should be only a modest performance penalty for this, and in dynamic allocation systems the supervisor process should soon provide a replacement for the failed node.

D. Efficiently Simulating a Cluster on a Single Node

We argued in our discussion of word2vec that it's extremely difficult to approach the performance of a GPU-memory based algorithm with a cluster solution on a commodity network. On the other hand secondary storage speeds are often comparable to, and may be significantly higher than, network speeds. A single hard drive at 100 MB/s is close to the speed of a 1Gb/s network, while a single SSD drive at 500 GB/s is half the maximum speed of a 10 Gb/s network. RAID arrays can easily outperform the attached network.

The common communication networks (mapreduce, MPI or Kylix's allreduce) used for cluster computing are layered. Communication occurs only between successive layers. Thus there is a straightforward simulation of these networks using a single node by simulating successively the nodes in each layer, and storing messages on disk. Thus in round R, we do the following:

for $R = 1,\ldots,N_{rounds}$
  for $i = 1,\ldots,N_{nodes}$
    Load node i state from disk
    Receive all messages m_{ji}^{R} directed to node i.
    Process messages
    Send all outgoing messages m_{ik}^{R+1} for next round
    Save node i state to disk.

Where m_{ji}^{R} denotes the message from node j to node i in round R and N_{rounds} and N_{nodes} are the number of rounds and nodes respectively. This approach was first described in the GraphChi system [7]. But GraphChi did not address the design of the "communication" network for on-disk communication, which is critical for good performance. We take this up next.

The main challenge with using disk storage for communication is latency. Once again, this implies a roofline constraint on throughput, shown in Fig. 9. Disk latencies are high, in the millisecond range and these imposes a penalty on small messages to disk. Fig. 9 shows the roofline for a small RAID array with 200 MB/s throughput. With these values we see that the roofline corner is around 1 MB, at which size latency equals transit time (and throughput is about half of the maximum). Packet size should be larger than this value for good performance. Using an (80%) efficient packet size of 5 MB gives virtually identical parameters to the Kylix design task for EC2 nodes presented in section II.C. Thus the optimal "network" design on 64 virtual nodes for the Twitter dataset analyzed there is 8x4x2 layered network while the optimal design for the Yahoo dataset is 16x4 as per figure 7.

E. Model-Parallel to Data-Parallel Transformation

Beyond single-node simulation, the disk messaging approach also supports parallelization where we run the simulation of many nodes with the same model (on disk) and different data on several physical nodes. This approach effectively transforms a communication-intensive model-parallel computation into a data-parallel one with potentially much lower communication. The communication between physical nodes can use the block size schedule from Fig. 4 with power-law data to improve throughput. In our Word2Vec implementation, between-GPU communication volume was less than 1% of the memory bandwidth consumed on each node. This approach may have significant performance advantages over running a normal parallel implementation over the physical network. The prerequisites for this approach to be efficient are:

1. Secondary storage throughput should be as fast as, and preferably faster than network throughput.
2. The overhead of loading and saving node data from disk must be small compared to communication cost.
3. Secondary storage must be large enough to hold the full model on each node.

Item 1 is likely to be true for older clusters using 1 Gb/s networking and magnetic disks. It's also true for newer systems with SSD storage (500 MB/s typical) and 10 Gb/s networking, since the 10Gb/s limit is rarely achieved with typical network stacks. Also the fat tree network architecture of most data centers implies that the single-node share of aggregate bandwidth across a cluster that spans racks and external switches is much less than 10 Gb/s.

Item 2 is true for most graph algorithms on large sparse graphs since model and data are both based on edges and vertices. For other algorithms, e.g. word2vec, it is only true for very large minibatches of data.

Item 3 is problem-specific. Typical ratios for secondary storage to memory capacity on data nodes are 10-100. This is a useful range for which it may be more efficient to use disk rather than the network as the primary communication medium.

An implementation of this approach is the subject of future work.

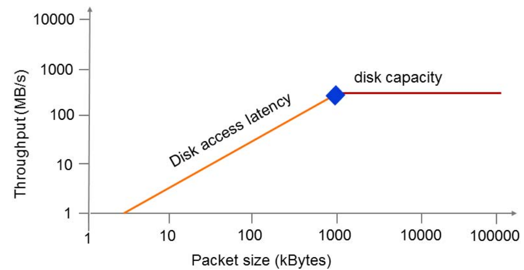

Figure 9: Roofline for disk throughput

III. BIDMACH'S ARCHITECTURE

We briefly review BIDMach's architecture here. While a primary goal is performance, the system was also designed to support ease-of-use and rapid prototyping of ML algorithms. BIDMach uses the Scala language which provides an interactive environment (a REPL), a state-of-the-art programming language combining functional, object-oriented and distributed programming (Actor) abstractions and good performance. Scala is based on the Java virtual machine, and so provides access to the large existing codebase of Java programs, including Hadoop, Spark, Storm and Yarn. While BIDMach can support arbitrary inference methods, it is designed for mini-batch algorithms such as SGD (Stochastic Gradient Descent). This design allows it to process large datasets or stream data of arbitrary size.

The toolkit comprises two major modules: BIDMat, a matrix/distributed computing layer, and BIDMach, the machine learning layer. BIDMat is similar to other functional/interactive matrix toolkits like Matlab and Numpy/Scipy. But BIDMat has several novel features:

- Tight integration of GPU computation. All matrix types (dense, sparse, single precision, double, integer matrices) are implemented on CPU or GPU. Using a generic matrix type, the programmer can ignore the implementation type. A caching scheme is included to manage memory for minibatch ML algorithms.
- The addition of machine-learning-oriented operators. Many operations that are useful for machine learning, e.g. sparsely-sampled matrix products, max-sum products, a variety of sorts and joins, are included.
- Simple APIs to rooflined network primitives (as distributed matrices) to simplify the process of distributing an algorithm.
- Rooflined primitives for CPU and GPU computation.

BIDMach's main class structure is illustrated in Fig. 10. The DataSource, Learner, Optimizer and Mixin classes are all shared across different learning algorithms. The code for a new model comprises model update code (e.g. model gradient) and an evaluation method for online cross-validated loss estimation.

BIDMach is designed to process large datasets in small batches to support mini-batch, streaming and online algorithms. It relies on an abstraction called a "DataSource" to do this. A DataSource is like an iterator over matrix blocks and includes a next() method which returns a matrix containing a block of samples. DataSources are currently implemented either as: (i) in-memory datasets, (ii) local disk-backed matrices, coarsely striped across many disks for high read and writes, or (iii) HDFS sources, which stream blocks directly from the datanode in an HDFS cluster. The throughput from a 40-100-node HDFS cluster is well-matched to a BIDMach learner process.

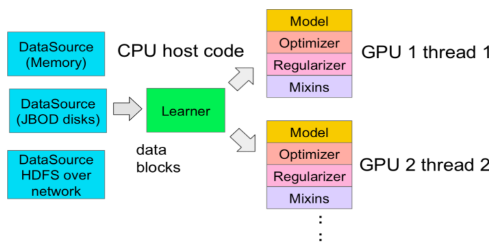

Figure 10: BIDMach's architecture

From the DataSource, the blocks of data are distributed by a Learner class to various instances of algorithm-specific code running in separate threads (normally on different GPUs, although one or more threads can run on a CPU). The framework separates the functions of model, which produces updates (usually gradient steps) from minibatch inputs, from regularizers, from different optimizers (Conjugate gradient, L-BFGS, ADAGRAD etc.) and includes support for a (possibly empty) list of mixins. Model updates for distributed learning can be either synchronous (as though there were only one model instance), or asynchronous, similar to the approach used in the Hogwild! System.

IV. RELATED WORK

There has been a great deal of work recently on tools for Big Data. Much of this work emphasizes scale-up on clusters [5, 8-12] without applying single-node acceleration (CPU- and GPU-specific acceleration libraries). An exception is Vowpal Wabbit (VW) [13] from Yahoo and now Microsoft, which uses a variety of techniques, including Intel CPU intrinsic instructions, to achieve high throughput. However, even VW does not match the performance on BIDMach on CPU hardware for many problems – rooflining is a systematic process of identifying and improving *every* kernel to its limit, and therefore more systematically eliminates bottlenecks in a large calculation. There are many special-purpose libraries for particular tasks that have been heavily optimized, e.g. libsvm, liblinear, libFM. These libraries have very good performance on CPU hardware but BIDMach is substantially faster on a GPU-equipped system. We maintain benchmarks for many of these systems on the BIDMach wiki:

https://github.com/BIDData/BIDMach/wiki/Benchmarks

V. CASE STUDIES

Here we describe several case studies of the application of BIDMach in two major internet companies.

A. *Offer/Query Categorization*

In offer/query categorization, the goal is to train a classification model for predicting product categories for incoming offers/retail queries. The training dataset has a few million examples and 500k features, and there are 2000 categories in the product taxonomy. A summer intern ran BIDMach's multi-label logistic regression classifier on the 2000 targets. On a single PC with GPU, training was completed in 2 min for one set of tuning parameters. This is 500x faster than the original in-house implementation of logistic regression with batch SGD in C++. Thanks to the dramatic improvement in speed, the intern was able to explore many parameter combinations and achieved a 7% improvement in accuracy over the original model.

B. *Auction Simulation*

Sponsored search auctions allow advertisers to bid in real-time on each search query. They include $N_{auction}$ parameters per auction that can be tuned to improve advertiser and publisher-directed metrics. These parameters are tuned using simulation based on recent live auction logs. The logs contain histories of search queries and bids, and the simulator simulates each of query auctions counterfactually many times using N_{params} different set of parameters. The parameters are further refined by query cluster of which there are $N_{clusters}$.

A large number of parameters need to be explored: The full parameter set is

$N_{auction} \times N_{params} \times N_{clusters}$

And approximately 1GB in size. The auction simulation is run over several weeks of data, producing a large array of basic performance indicators. From this array of basic indicators, optimal parameters choices for desired KPIs (Key Performance Indicators) are derived using a simple Lagrange multiplier step. This latter step takes a few minutes, and can be done online.

Auction simulation is quite complex so we developed first a Scala reference implementation for debugging. Then we wrote two GPU implementations. The first relied on shared memory for working storage, and achieved an 70x speedup over the Scala reference implementation. The second used more aggressive coding, moving almost all the state of each auction to register memory, and e.g. sorting bids in registers. This gave a full 400x speedup over the Scala implementation. The single-node GPU implementation was faster than the previous production simulation running on a medium-sized cluster, and uses per-cluster parameter search which the previous system could not.

This example hints at the (largely unexplored) potential speedups on graphics processors by using register implementations. Whereas GPUs have roughly 10x the computing power and perhaps 10x the memory speed vs. standard CPUs, they have 1000x the register storage. This storage supports single-cycle access with no contention, and for suitable algorithms, opens the door to multiple order-of-magnitude speedups.

C. *SVM Classification*

A colleague was performing SVM classification of feature vectors extracted from images using Berkeley's Caffe toolkit. The image dataset included approximately 0.5 million images with labels from 500 classes. The feature vector datasets were obtained from a "vanilla" caffe classifier trained on ImageNet data, and on a custom classifier trained on the labeled data. Using libsvm, a popular SVM package, training on one of the feature datasets took one week. The classifier for the other dataset had not converged after one week. Training a multilabel classifier with BIDMach took 90 seconds on both datasets. This is perhaps a best case for BIDMach – the dataset featured multiple labels, which libsvm does not support. The feature set was dense and quite large, which leverages the fast matrix multiply primitives in BIDMach. The end-to-end throughput was around 200 gflops.

We also generated Random Forest classifiers for the two datasets. The datasets were too large for in-memory RF algorithms, but BIDMach includes out-of-memory Forest algorithm, which took less than half an hour to generate a 16-tree, depth-20 forest. Classification accuracy was quite high, and comparable to Caffe's own ImageNet performance.

The above case studies are a reasonable representative sample of real-world problems to which BIDMach can be applied. We did not select them for suitability for BIDMach, and we are in the course of evaluating many other applications at this time.

VI. PERSPECTIVE

To summarize: roofline design as realized in the BIDMach toolkit offers orders of magnitude gains for large-scale inference problems. The gains are perhaps surprising in some cases, since they are larger than is implied by the CPU/GPU performance gap. It follows that many other tools are running well below the roofline limits for the hardware they are running on. This reinforces the importance of roofline design, since without it the risks of wasted performance are high.

The BIDMach toolkit was designed for both high-performance and rapid prototyping and customization of machine learning algorithms. By using a matrix layer, ML algorithms can be written quickly in a high-level language similar to Matlab or SciPy. This was the "first generation" design of the toolkit. We are now moving to a high-performance design with templated data movement code and a small amount of custom code (typically the loss function and its derivatives) for each model. While there is some loss of flexibility with this approach, it may actually simplify development for new algorithms that use a previous data movement pattern.

We described techniques for local parallelization of power-law data on a multi-GPU machine, which exploit the structure of the data. We argued that scalability does not require implementation on a large cluster. Our results suggest that hardware should scale conservatively from single machines, to single racks, and only then to larger networks. The performance of our basic algorithms is in the 10's of gigaflops and is competitive with small to medium-sized cluster. Word2vec achieves 500 gflops on a multi-GPU machine. This level of performance is probably beyond the reach of a model-parallel cluster implementation on any number of nodes. The gap in throughput between GPU memory and commodity networking (300 GB/s vs < 1 GB/s) is simply too large, and linear scaling is almost never seen for tightly-coupled calculations such as these. As we move more algorithms to the end-to-end pattern, this will likely be true for those algorithms as well.

In cases where a cluster implementation is essential, e.g. for graph algorithms on large datasets, we described a rooflined design called Kylix. While Kylix's end-to-end throughput at 3 Gb/s was well below the theoretical roofline

(10 Gb/s) for its network, we are not aware of other systems reaching higher speeds in practice. Among other challenges: full node bandwidth is only available on lightly-loaded networks. The outgoing bandwidth for a typical rack is much lower (3-10x) than the aggregate of the nodes in it, and later switches further constrain bandwidth. And a large machine learning calculation creates its own load since it is bottlenecked by the network. Thus it is virtually impossible to scale a large calculation linearly. In fact there will be several discrete steps where per-node throughput decreases. Kylix can be tailored to handle these topologies. Since it does progressive reductions, the layers of Kylix can be localized within rack or within the scope of a given switch. At each layer, Kylix reduces the bandwidth needed by doing reductions on the local subset of nodes.

We next explored the rooflined design of “distributed”-model algorithms on single nodes using secondary storage. It is even possible to parallelize such an implementation by replicating the model on each node. Under the right conditions (graph algorithms being a good match), this approach should exceed the performance of a direct model-parallel networked implementation.

In many cases, the speed advantage from rooflined ML algorithms can be used to gain accuracy by more thorough exploration of hyper-parameters (as in the first case study). On single-node GPU machines, there is no job queuing or contention for machines, so iteration is extremely fast. Alternatively, more complex and accurate models can be run in the same time as simple models with other toolkits. e.g. the random forest training time in the last example is lower than the time needed to train a libsvm SVM model, or a logistic regression model in Scikit-Learn on that dataset.

REFERENCES

[1] S. Williams, A. Waterman, and D. Patterson, “Roofline: An Insightful Visual Performance Model for Multicore Architectures,” *Communications of the ACM,* vol. 52, no. 4, 2009.

[2] J. Canny, and H. Zhao, “Big Data Analytics with Small Footprint: Squaring the Cloud,” in Proc. ACM SIGKDD Conf. on Knowledge Discovery and Data Mining, Chicago, 2013.

[3] H. Zhao, and J. Canny, “Kylix: A Sparse Allreduce for Commodity Clusters,” in Proc. Int. Conference on Parallel Processing (ICPP), 2014.

[4] T. Mikolov, I. Sutskever, K. Chen *et al.*, “Distributed representations of words and phrases and their compositionality,” in NIPS, 2013.

[5] J. E. Gonzalez, Y. Low, H. Gu *et al.*, "PowerGraph: Distributed graph-parallel computation on natural graphs," *OSDI*, 2012.

[6] H. Zhao, and J. Canny, *Sparse Allreduce: Efficient Scalable Communication for Power-Law Data*, Cornell, 2013.

[7] A. Kyrola, G. Blelloch, and C. Guestrin, “GraphChi: Large-Scale Graph Computation on Just a PC,” in Operating Systems Design and Implementation, 2012.

[8] “Mahout,” *http://mahout.apache.org/*.

[9] M. Zaharia, M. Chowdhury, T. Das *et al.*, “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,” in NSDI, 2012.

[10] Y. Low, J. Gonzalez, A. Kyrola *et al.*, “Graphlab: A new framework for parallel machine learning,” *arXiv preprint arXiv:1006.4990*, 2010.

[11] M. Li, D. G. Andersen, J. W. Park *et al.*, “Scaling Distributed Machine Learning with the Parameter Server,” in Operating Systems Design and Implementation, 2014.

[12] J. Shin, S. Wu, F. Wang *et al.*, “Incremental Knowledge Base Construction Using DeepDive,” in Proceedings of the VLDB Endowment, 2015.

[13] A. Agarwal, O. Chapelle, M. Dud´ık *et al.*, "A Reliable Effective Terascale Linear Learning System."