BIDMach: Large-scale Learning with Zero Memory Allocation

John Canny and Huasha Zhao
Computer Science Division
UC Berkeley
Berkeley, CA 94720
{jfc,hzhao}@cs.berkeley.edu

Abstract

This paper describes recent work on the BIDMach toolkit for large-scale machine learning. BIDMach has demonstrated single-node performance that exceeds that of published cluster systems for many common machine-learning task. BIDMach makes full use of both CPU and GPU acceleration (through a sister library BIDMat), and requires only modest hardware (commodity GPUs). One of the challenges of reaching this level of performance is the *allocation barrier*. While it is simple and expedient to allocate and recycle matrix (or graph) objects in expressions, this approach is too slow to match the arithmetic throughput possible on either GPUs or CPUs. In this paper we describe a caching approach that allows code with complex matrix (graph) expressions to run at massive scale, i.e. multi-terabyte data, with *zero memory allocation* after initial start-up. We present a number of new benchmarks that leverage this approach.

1 Introduction

Modern hardware is capable of high-speed execution of the core operations needed for machine learning. Table 1 below shows the throughput of several operations on standard CPU and GPU hardware, The table also includes the throughput for a pure Java implementation, and the performance gain of a GPU implementation (last column) over Java. The machine used has dual 8-core Intel Sandy Bridge E5-2650 processors, and dual GTX-690 GPUs. We chose this combination to approximately balance the cost of CPU and GPUs ($1000 for each CPU or GPU).

Table 1: Performance of ML Primitives

Task	Java	CPU	GPU	GPU/Java
SGEMM(gflops)	2	550	3500	2×10^3
SPMV(gflops)	1	20	10	10
SPMM(gflops)	1	12	120	120
sort(gitems)	0.01	0.2	1	100
rand(gevals)	0.08	1.6	90	10^3
exp,ln(gevals)	0.02	0.6	200	10^4

SGEMM is a dense matrix/matrix multiply, SPMV and SPMM are products of a sparse matrix with respectively a dense vector or a dense matrix. The next row shows sort performance (integer or float keys) in billions of items/sec. The fifth row shows throughput in billions of samples/sec for basic random number generation, and the last row shows throughput in billion evaluations per second for transcendental function evaluation. The sparse matrix measurements, which can be sensitive to matrix structure, were evaluated on typical (power law distributed) representing term counts in text.

The CPU column shows the speed for a high-performance CPU library, either Intel MKL or Intel IPP. The GPU column includes both CUDA and BIDMat routines, whichever is faster.

SPMV is the dominant step in regression and SVM on sparse data (like text, web, clickthrough etc), and performance typically scales with the choice of SPMV primitive used. SPMM is the bottleneck primitive for factor models such as LDA (Latent Dirichlet Allocation) NMF with KL-divergence loss and a fast version of ALS (Alternating Least Squares), together with a related primitive called SDDMM [1]. As was shown in [1], SDDMM can be implemented with similar performance to SPMM. Thus one can expect about a 100-fold improvement in performance for factor models over naive Java or C implementations. Random number generation is a bottleneck for carefully-written MCMC algorithms (one must first remove non-memory-coherent linked data structures). Transcendental function evaluation is a bottleneck for very sparse problem using transcendentals (e.g. Logistic regression) and for Lp distance evaluation (e.g for computational genomics). SGEMM will often dominate performance on dense problems, and also is the bottleneck for the L2-loss form of NMF. Many graph algorithms have a matrix interpretation using the adjacency matrix of the graph, e.g. Pagerank, triangle counting etc. and can similarly leverage a high-performance matrix layer. Sorting is the bottleneck for many sparse algorithms, including Gibbs samplers for LDA etc. While one can implement machine learning algorithms without using high-performance matrix libraries, this is likely to be slow. Most machine learning algorithms bottleneck on a calculation that is equivalent to one of the above. High-performing systems such as BIDMach, Weka, SciPy etc layer machine learning algorithms on a matrix layer. Then the benefits of high-performance matrix kernels will carry over to the learning layer.

1.1 The Memory Allocation Barrier

In [1] several end-to-end machine learning algorithms were described which leveraged the GPU primitives above and which on a single node exceeded the performance of any published *cluster* implementation. Obtaining those results was challenging. While the matrix primitives above are fast, a complex machine learning algorithm generates many intermediate results which are themselves large matrices. A language like Java is a benefit here, because these objects can be automatically allocated and garbage collected without the matrix library having to manage reference counts. But it becomes a liability as the problem scales up. SPMV and SPMM are memory-limited functions. When results have to be repeatedly allocated and garbage-collected, the memory overhead for each datum is multiplied (since it must be at least cleared by the GC, and often copied somewhere else in memory). The overall performance of the algorithm often drops by a factor of two or three. On small machines with a few GB of memory we have seen steady-state performance (i.e. throughput over hours) drop by a *factor of five* due to garbage collection. We call this the *allocation barrier*.

For GPUs, the problem is much worse. There is no garbage collector in the standard GPU libraries. And importantly, memory allocation on a GPU is much slower (1 GB/sec) relative to GPU memory access (150GB/s). So even if a GPU garbage collector were available it would have *an overhead of more than 100x*. Instead, we have implemented a caching scheme in BIDMat which supports re-use of intermediate matrices, and which achieves zero-memory allocation after a start-up period. This approach complements *mini-batch* processing of a large dataset. i.e. where data are read from a massive dataset and processed in blocks. Many ML algorithms have a mini-batch formulation, e.g those that use stochastic gradient optimization, "online" or "streaming" implementations [2], [3], and most MCMC implementations.

2 Architecture of a Scalable Learner

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[1]. DataSources are currently implemented either as: (i) in-memory datasets, (ii) disk-backed matrices, coarsely striped across many disks for high read and writes (iii) HDFS sources, which stream blocks directly from the datanode in an HDFS cluster. While individual

[1] Actually a DataSource in general returns several matrices for each block, e.g. for regression it may return separate matrices of features and targets.

HDFS nodes typically have quite low read throughput (20MB/s), the throughput from a 100-node cluster is well-matched to a BIDMach learner process.

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 threads can run only on a CPU). See figure 1. BIDMach avoids "canned" machine learning routines in favor of a modular "GLM" approach where a basic model can be mixed with different regularizers, different optimizers (Conjugate gradient, L-BFGS, ADAGRAD etc.), and with various mixins which add losses for things like component dependence.

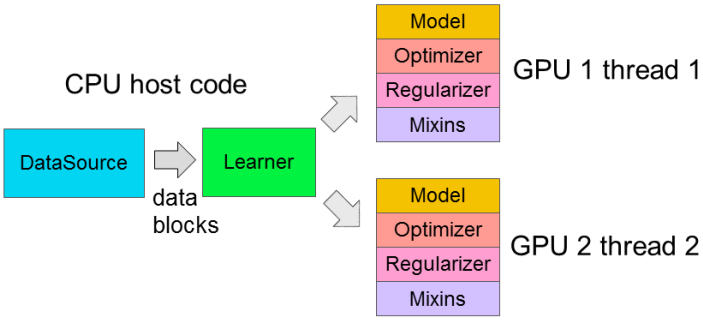

Figure 1: Learner architecture

2.1 Caching Learner Code

The source code for matrix-based variational LDA E- or mean-step is given below.

```
def eStep(sdata:Mat, user:Mat):Unit = {
   for (i <- 0 until opts.uiter) {
     val preds = SDDMM(mm, user, sdata)
     val unew = user o (mm * (sdata / preds)) + opts.alpha
     user <-- exppsi(unew)
   }
}
```

We include the code here to aid in the description of our caching scheme. The inputs to the E-step routine are two matrices: `sdata` which is a block of sparse input samples, and `user` which is a dense matrix of topic weights for those samples, and also the word $\times$ topic matrix `mm` which is inherited from a containing lexical scope.

The algorithm generates both explicit (preds, unew) and implicit temporary matrices (`sdata/preds` etc.), one for each operation whose result is not saved to an explicit target. These matrices constitute a substantial amount of storage, since each turns out to have the same size as one of the input matrices `sdata` or `user`, and the intermediate results are being computed in a loop which repeats `opts.uiter` times (typically 10-20). Thus the temp storage which needs to be allocated and freed for this routine is typically 20-60 times the size of the input. For 256-dimensional LDA of a 1TB input DataSource, the total of all `user` matrices is 10 TB, requiring around 0.5 PB of temp storage allocation. At typical GPU allocation speeds (1 GB/s) this would slow the algorithm down by more than 100x (even leaving aside the problem that there is no GPU garbage collector to reclaim the temps). Clearly we should avoid this if possible.

When operators are sufficiently local (e.g. $A + B$ or a small convolution kernel), one can compile a series of operations into a single-pass operator that is mapped across an input matrix [4]. However, global operators (e.g. SDDMM, matrix-* which both occur in the LDA routine) are common in machine learning code, and do not support this approach. Instead we use a caching scheme which is invisible to the programmer, which allows us to re-use matrices from one mini-batch call to the next.

2.2 Matrix Caching

In a functional language, objects are immutable. Expressions like $A \otimes B$ always have the same value within a scope where A and B are fixed, and the results can be safely cached and re-used. BIDMat does not currently include immutable matrix types, although most operations in Learner code would work with them. However all matrices *have immutable dimensions*. Furthermore, knowing the dimensions of two input matrices and the type of operator and certain additional arguments *determines the result matrix*. Rather than caching values, we can cache containers that can hold the values of future expressions involving the same input matrices, and any other input arguments that determine the output dimensions.

To be more concrete, each matrix has a GUID which is set when the matrix is created. A basic operation like $C = A + B$, rather than allocating new storage for C, first checks the matrix cache using a key

```
(A.guid, B.guid, hash("+"))
```

If A and B are immutable, the output is fully determined by their values and the operator $+$. It is therefore safe to use a *container* for C within any code segment where A and B are fixed. A and B are actually mutable which is important for us to be able to re-use their storage. But once set, we can derive arbitrary expression DAGs from them with recycled storage.

Some operations return matrices whose dimensions depend on other matrices. For instance, an expression $A(I, J)$ with matrix A and integer matrices I and J returns a matrix whose size is the product of the number of elements in I times the number of elements in J. Since these matrices have immutable dimensions, the following hash key will find a correctly-sized container if one has been saved:

```
(A.guid, I.guid, J.guid, hash("apply"))
```

Scalars (floating point or integer) can be mixed with matrices in expressions and have the usual mathematical interpretation. e.g. $A+1$, $B*2$ etc. Since scalars are values rather than references (as for matrices) there are no GUIDs associated with scalars, and caching needs to be done carefully. To avoid aliasing the scalar values must be incorporated into keys. e.g. $A + 1$ is cached with key

```
(A.guid, 1, hash("+"))
```

But these can lead to a failure of caching in common functional expressions like $a * A$ where a is a scalar which is changing from one iteration to the next. Each evaluation leads to a separate cached value even though the symbol a is the same, and its value is assumed fixed in that scope. The simplest solution is to use 1x1 matrices for scalars. Setting a's contents and keeping it fixed inside model methods will generate only one cached value.

Going back to example 2.1, every operator and function for the LDA E-step represents a functional operation and uses a cached output. The exception is the assignment statement at the end of the for loop. The `<--` operator copies a result into the LHS, modifying the matrix `user`. This pattern mimics the updates to loop variables in functional for loops in Scala and other functional languages. Without this operation, the expression DAG depth (and the temporary storage) for this calculation would be proportional to `opt.uiter`. By placing the assignment operator at the end of the function, we ensure that each pass through the for loop uses only immutable values, and avoids aliasing. We note that use of the assignment operator is important even if matrix caching is not used. Without it, a number of temps will be created and recyled proportional to the number of iterations.

While it is a general approach, matrix caching strongly complements the Learner pattern described earlier. Since the input to the E-step is a mini-batch, matrices `sdata` and `user` have the same size on each call. The Learner is designed to use *exactly the same matrices* to hold these values, and pass them to the model update function. This minimizes allocation, but also ensures that any temps created inside the update will be re-used when caching is enabled. Thus there are no new temps allocated at any step in the algorithm, and yet the code has a simple functional form.

All models in BIDMach use matrix caching, and have zero memory allocation after the first call. This eliminates memory allocation overhead and allows the models to be learned at the full speed of the hardware. An updated set of benchmarks using these routines are included in an appendix to the paper.

References

[1] John Canny and Huasha Zhao. Big data analytics with small footprint: Squaring the cloud. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2013.

[2] Naiyang Guan, Dacheng Tao, Zhigang Luo, and Bo Yuan. Online nonnegative matrix factorization with robust stochastic approximation. *Neural Networks and Learning Systems, IEEE Transactions on*, 23(7):1087–1099, 2012.

[3] Matthew Hoffman, Francis R Bach, and David M Blei. Online learning for latent dirichlet allocation. In *advances in neural information processing systems*, pages 856–864, 2010.

[4] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frdo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013.

[5] Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010.

[6] Huasha Zhao and John Canny. Sparse allreduce: Efficient scalable communication for power-law data. In *submitted*, 2014.

Appendix

We collect together a number of benchmarks for BIDMach, and include new benchmarks for each problem which improve on previously-published results. More details on the datasets used are given in [1]. We first report on recent development of high-speed network primitives for scalable data mining. In this paper, we addressed the *allocation barrier* to performance. A parallel challenge occurs in the network layer. Traditional network primitives achieve much lower throughput than hardware-accelerated matrix primitives. To bridge this gap, in [6], we introduced a new network primitive, a *Sparse Allreduce* that provides higher throughput, scalability and error tolerance, for power-law data. The performance of this primitive on graph data is shown in the last row (BIDMat+SA) in the table below. It can be seen that this approach provides a five-fold improvement over the closest comparable system, Powergraph.

Table 2: Pagerank Performance

System	Graph VxE	Time(s)	Gflops	nodes x CPU cores
Hadoop	?x1.1B	198	0.015	50x8
Spark	40Mx1.5B	97.4	0.03	50x2
Twister	50Mx1.4B	36	0.09	60x4
PowerGraph	40Mx1.4B	3.6	0.8	64x8
BIDMat	60Mx1.4B	6	0.5	1x8
BIDMat+disk	60Mx1.4B	24	0.16	1x8
BIDMat+SA	60Mx1.4B	0.6	4.8	64x8

The following benchmarks are for a mini-batch algorithm for Variational LDA [3]. We include an additional line in the table derived from the current Learner implementation which distributes across 4 GPUs (two dual GPU boards) on our test node. These modifications increase the speed ratio for our single-node implementation to 10x vs the closest comparable system, Powergraph, running on 64 nodes.

Table 3: LDA Performance

System	Docs/hr	Gflops	nodes x CPU cores x GPUs
Smola[5]	1.6M	0.5	100x8x0
PowerGraph	1.1M	0.3	64x16x0
BIDMach	3.6M	30	1x8x1
BIDMach	10M	85	1x8x4

In addition to the experiments above, we computed a 256-dimensional LDA factorization of a 1 TB dataset (50,000 word features), which comprises approximately 9 months of the Twitter "Gardenhose" (a random 10% sample of all tweets). The throughput was 80 gflops, and each iteration took approximately 3 hours. Convergence to a perplexity of 90 was achieved after two passes over the dataset, which corresponded to 200,000 mini-batch updates. To our knowledge this is significantly larger than any other completed LDA factorization, on a cluster or otherwise.

Now for ALS (Alternating Least Squares), we add a new experiment on quad-GPUs. This change produces similar improvements across scales. Note that as described in [1], rewriting ALS to follow a loop-interleaving pattern led to an asymptotic improvement. So the time differences below are due only in part to the faster GPU primitives we used. Results are from the "small Netflix" dataset developed by the GraphLab group.

Table 4: ALS Performance

System \ Dimension	20	100	1000	nodes x CPU cores x GPUs
Powergraph	1000s	**	**	64x8x0
BIDMach (CG)	150s	500s	4000s	1x8x1
BIDMach (CG)	50s	150s	1300s	1x8x4

Finally, we conducted some new experiments on multi-model regression. Regression involves relatively little computation and is typically an I/O bound calculation. Therefore there is little advantage from GPU acceleration for single-model regression. However, BIDMach has been written to support multi-model regression using shared SPMM operations. Each regression can be a distinct GLM model (i.e. one logistic, one linear), with different features, regularization, and targets. So one can regress many tuned regressors against multiple targets, which is a common scenario in industry (e.g. targeting many ad campaigns). The table below shows time in seconds on two datasets (unfortunately the dataset used for the Spark experiments is no longer available) for logistic regression. However, regression performance is normally a function only of dataset size and number of iterations.

Table 5: Regression Performance

System \ NumTargets	1	300	nodes x CPU cores x GPUs
Spark	46s	**	100x4
BIDMach	100s	500s	1x8x1

Note that single-node GPU regression with one target is 2x slower than a cluster implementation on 100 nodes. This is still a 50x advantage in throughput per node. However, with hundreds of targets, running time increases only slightly for a GPU implementation. Comparable benchmark data is not available for Spark. However, Spark uses naive sparse matrix algebra and transcendental functions, and these appear to be the bottleneck for the single-target implementation. Running time should scale linearly with number of targets, and the GPU multi-model implementation looks to have more than a 20x advantage over the 100-node cluster implementation for hundreds of targets.