SC15

libSkylark: A Framework for High-Performance Matrix Sketching for Statistical Computing - http://xdata-skylark.github.io/libskylark/

G. Kollias[1], Y. Ineichen[2], H. Avron[2], V. Sindhwani[4], K. Klarkson[3], C. Bekas[2], A. Curioni[2]
IBM Research – {[1]Watson, [2]Zurich, [3]Almaden}, [4]Google

Introduction

The libSkylark library provides sketching-based matrix computations for machine learning suitable for general statistical data analysis and optimization applications.
Many tasks in machine learning and statistics ultimately end up being problems involving matrices: whether you're finding the key players in the bitcoin market, or inferring where tweets came from, or figuring out what's in sewage, you'll want to have a toolkit for least-squares and robust regression, eigenvector analysis, non-negative matrix factorization, and other matrix computations.

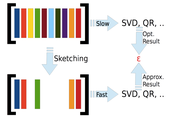


Figure 1: Sketching is a way to compress matrices that preserves key matrix properties; it can be used to speed up many matrix computations.

- For a good sketch B of input A, if solving the problem for B will yield a good solution for A.
  - Typically, up to some relative error ϵ, perhaps $\epsilon = 0.1$
- Sketches can also be used to get faster ways to find high-precision solutions to the original problem.
- Sketches can be used to summarize the data by identifying the most important rows or columns.
  - I.e. just sampling rows/columns of the matrix with probability proportional to their norms
- At its core, requires a robust, efficient and scalable mechanism for distributed generation of samples and scalable high performance matrix-matrix multiplication.

Matrix Completion	Nonlinear RLS, SVM, PCA	Robust Regression	Other Applications
Python: Python-based data analytics scripting layer			
PythonBinding: C++ to Python bindings			
NLA: Numerical Linear Algebra primitives (Least squares regression, low-rank approximation, randomized estimators)			
Sketch: Sketching kernels JL, FJL, Gaussian, Sign, Sparse Embedding			
Third-Party Libraries: MPI, Elemental, BLAS, CombBLAS, FFTW, Boost			

An Example Sketch: Johnson-Lindenstrauss Transform (JLT)

We identified the common structures in all sketching algorithms as implemented in libSkylark. This enables us to express the sketching algorithms in a generic, yet inclusive form. Subsequently we present a number of different streams (realized as vectors of random or deterministic quantities) defining the property of any sketch type.

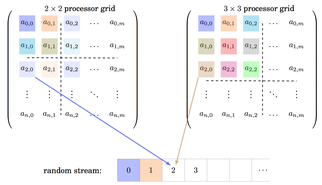
- *The mix_set* of indices of rows that will "mix" (linearly combine /accumulate) towards a row in A_hat.
- *The mix_factor[k][i]* is the scaling of the i-th row in A to accumulate into l-th row of A_hat (k = select_stream[l]).
- *The post_mix[k](x)* is a function that further process kth row x by independently processing each of its entries x[j] (immediately after the mix stage).
- *The value_stream* is a stream of real random samples drawn from a distribution. The distribution and the number of entries drawn from it (size) follow.
- *The select_stream* is a stream of sketch_size integers either random samples drawn from a distribution or a list of consecutive ones (select_stream[l] = l).
- *The hash_stream* is a stream of n integer random samples drawn from a distribution or None (used in the definition of mix_set).
- *The parameter_stream* is a stream of sketch_size real random samples drawn from a distribution or None (used in the definition of function post_mix[k](x)).

For example we can define JLT in terms of the following properties:

stream	value
mix_set	$[0, n)$
post_mix	value_stream[$k \times$ sketch_size $+ i$]
value_stream	normal distribution of size sketch_size $\times\, n$
select_stream	[0, sketch_size)
hash_stream	–
parameter_stream	–

Randomness

Our requirements are (i) random access to (ii) multiple potentially non-overlapping (iii) efficiently serializable streams of pseudo random numbers.



Unfortunately most conventional pseudorandom number generators (PRNGs) are inherently sequential: a random sample in their state space is a function of its immediate predecessor. This is a severe performance penalty, dramatically shrinking the envelope for scalability of the approach.

Counter-based random access pseudo-random streams are our solution infrastructure:

- Generally functions of the form: $x_n = b_k(n)$. Here n is a counter, k is a key, both integers, and $b_k()$ is a member of a family of bijections.
- CBRNGs are inherently parallelizable.
- Random123 [5] provides a high performance implementation of CBRNGs.

From random access pseudo-random streams to matrices of random samples in parallel
Given a `counter` and a `key` input argument, a single invocation of the CBRNG yields a pair of 64-bit numbers that we can regard as a uniformly random 128-bit unsigned integer. With this we can turn to the question of how to generate an `R[i,j]` entry in an $s \times n$ sketching matrix given an `counter` and `key`.

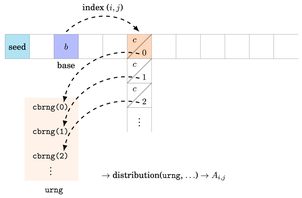


Figure 3: Behind the scenes: Random123 stream provides two `uint64_t` values r_0 and r_1 given a base b and some index i.

Summarizing, in order to compute an arbitrary `R[i,j]` entry of the $s \times n$ sketching matrix:

- (i, j) coordinates are converted to the column-major traversal index: `index = j * s + i`
- The corresponding location in the URNG-provided stream is assigned to the counter: `counter = base + index`
- A URNG object is thus constructed: `urng = URNG(counter, key)` and is passed to the distribution constructor: `distribution = Distribution(urng)`.
- The distribution object is called: `R[i,j] = distribution()`

Some notes on the implementation in libSkylark:

- The sketching matrix or parts of it can be lazily evaluated without any communication.
- We use the substream approach, where each matrix gets its slice in the global stream that is uniquely identified by the seed of our libSkylark context.
- We introduce the "one stream for all processes" approach: this is a random access stream where every single process is granted the capability to compute any matrix entry. This is a departure from the classical approaches for parallelization.

High-Performance Sketching

Sketching a distributed **dense matrix** typically amounts to multiplying it by a matrix of random samples.

- For dense sketches we use Elemental's distributed dense matrices [4] and this implies the use of the distribution schemes this library provides.
- We follow a similar strategy as discussed in [7] to minimize the communication costs:

- **cover-gemm**: Enforce the **index coverage** requirement directly. This would entail communication cost for redistributions. After this the distributed GEMM would be realized just by GEMM-ing local matrix parts.
- **align-gemm-collect**: Enforce the **alignment** requirement only: GEMM local parts to get $\hat{c}_{ij}$ as dot products of stride-indexed A rows and B columns and finally collect and accumulate missing partial sums in the dot products as already computed at other processes to form c_{ij}.

Table 3: Implementation cases for distributed GEMM and their underlying generic patterns; "stationary" means that no communication happens over the matrix object to follow.

Implementation case	Generic pattern	Comment
`outer_panel`	*cover-gemm*	stationary C
`matrix_panel`	*align-gemm-collect*	stationary A
`panel_matrix`	*align-gemm-collect*	stationary B
`inner_panel`	*align-gemm-collect*	

For sketches of distributed **sparse matrix**, the sketching matrix can be stored compactly.

- Currently all distributed sparse matrices are stored in the doubly compressed sparse column (DCSC) format [2] (CombBLAS).
- Sketches are never realized but exist as two random number streams (values/hashes).
- Communication schemes make use of one-sided primitives.
- For sparse sketches the savings of not realizing and communicating the sketching matrix is marginal.

Scalability the Sketching Layer

We experiment our parallel framework through the computation of sketches on BlueGene/Q^{TM}. We benchmark a columnwise JLT application on a tall-and-thin dense matrix. Least square problems are typical applications where this sketch transforms are used in practice.

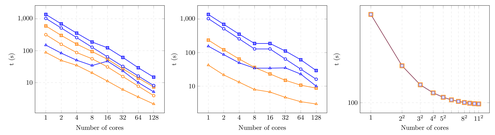


Figure 5: Timings strong scaling runs for elemental (blue) and libSkylark (orange) on BG/Q using 4 OpenMP threads per core. Squares denote the total runtime, circles the random sample generation time and triangles the distributed gemm. LEFT: JLT on tall-and-thin dense matrices, MIDDLE: FJLT on tall-and-thin dense matrices, RIGHT: Sparse sketch on sparse matrix vs. CombBLAS (blue).

- Better performance can mainly be attribute to the much lower cost of generating random samples and avoiding to communication involved with the matrix of random samples.
- On the right side we show a substantial performance improvement of libSkylark compared to the baseline case (FJLT).
- In addition to the savings in communication for random samples, we require less random samples in the FJLT as in the JLT case.
- As expected sparse sketches are only marginally better than a straight forward sparse matrix matrix multiplication.

Numerical Linear Algebra and Machine Learning

On top of the sketching layer, a suite of NLA and ML primitives are implemented:

- Least-squares solver (based on the Blendenpik approach [1]).
- Randomized SVD (based on Halko, Martinsson and Tropp [3])
- Kernel-based regression and classification [6]

Additional primitives are under active development.

Application: Word Embeddings for Natural Language Processing

- Word embeddings map words to vectors so that the similarity between words and word combinations is captured by correlation between vectors. **It is a very useful tool for Natural Language Processing**.
- We used `gensim.corpora.WikiCorpus` for parsing the Wikipedia dump and hyperwords for converting to Positive Pointwise Mutual Information (PPMI) matrix.
- For word w and context c, $PMI(w, c)$ is defined as the log ratio between w and c's joint probability and the product of their marginal probabilities.
- PPMI is obtained by zeroing negative entries in the PMI matrix.
- Randomized SVD is then used to compute the word embedding.

Table 4: Compared the performance of libSkylark's randsvd approach to GloVe (`http://www-nlp.stanford.edu/data/glove.6B.100d.txt.gz`). We conduct similarity performance comparison for GloVe and our randsvd generated embeddings using similar methodology and suite of test datasets as in related papers from Omer Levy (the hyperwords package developer).

Benchmark	GloVe (Spearman rho)	randsvd (Spearman rho)
RADINSKY_MTURK (287/287 pairs)	0.599986	0.634235
WS353 (353/353 pairs)	0.492414	0.571149
WS353.RELATEDNESS (252/252 pairs)	0.489905	0.494797
LUONG_RARE (1782/2034 pairs)	0.331318	0.368968
WS353.SIMILARITY (203/203 pairs)	0.543087	0.651343
MEN (3000/3000 pairs)	0.664732	0.672004

Application: Phone Classification for Speech Recognition

- **Phone classification** is used as part of a speech recognition pipeline.
- We use libSkylark kernel-based classifier to train a classifier using the TIMIT dataset.
- The solver exhibits good scaling on BG/Q and a commodity cluster, and state-of-the art classification quality (comparable to Deep Neural Nets).

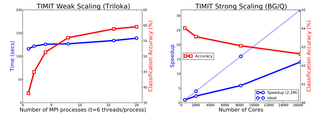


Figure 6: Strong and weak scaling on TIMIT.

Acknowledgements and References

The authors acknowledge the support from XDATA program of the Defense Advanced Research Projects Agency (DARPA), administered through Air Force Research Laboratory contract FA8750-12-C-0323. We also thank Jack Poulson for his helpful insights and discussions on Elemental.

[1] Haim Avron, Petar Maymounkov, and Sivan Toledo. Blendenpik: Supercharging lapack's least-squares solver. *SIAM Journal on Scientific Computing*, 32(3):1217–1236, 2010.

[2] Aydin Buluc and John R. Gilbert. On the representation and multiplication of hypersparse matrices. In *Proc. 22nd IEEE International Symposium on Parallel and Distributed Processing (22nd IPDPS'08)*, pages 1–11, Miami, Florida, USA, April 2008. IEEE Computer Society (Los Alamitos, CA).

[3] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[4] Jack Poulson, Bryan Marker, Robert A van de Geijn, Jeff R Hammond, and Nichols A Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software (TOMS)*, 39(2):13, 2013.

[5] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 16:1–16:12, New York, NY, USA, 2011. ACM.

[6] V. Sindhwani and H. Avron. High-performance kernel machines with implicit distributed optimization and randomization. In *JSM Proceedings, Tradeoffs in Big Data Modeling - Section on Statistical Computing*, 2014. To appear in Technometrics.

[7] R. A. Van De Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.