FlashR: Parallelize and Scale R for Machine Learning

Anonymous Author(s)

Abstract

R is one of the most popular programming languages for statistics and machine learning, but the R framework is relatively slow and unable to scale to large datasets. The general approach for speeding up an implementation in R is to implement the algorithms in C or FORTRAN and provide an R wrapper. FlashR takes a different approach: it parallelizes a large number of matrix functions in the R *base* package and scales them beyond memory capacity by utilizing solid-state drives (SSDs) automatically. As such, FlashR parallelizes and scales existing R code with little/no modification.

FlashR accelerates parallelized R code by deploying techniques that reduce data movement between CPU and SSDs: *(i)* evaluating all matrix operations lazily, *(ii)* performing all matrix operations in a DAG in a single execution and with only one pass over data to increase the ratio of computation to I/O, *(iii)* performing two levels of matrix partitioning and reordering computation on matrix partitions to reduce data movement in the memory hierarchy. We evaluate FlashR on a variety of machine learning and statistics algorithms on inputs of up to four billion data points. Despite the huge performance gap between SSDs and RAM, FlashR running on SSDs closely tracks the performance of FlashR in memory for many machine learning algorithms. The R code of these machine learning algorithms executed in FlashR outperforms the in-memory execution of H_2O and Spark MLlib by a factor of $3 - 20$.

Keywords R, parallel, machine learning, solid-state drives

1 Introduction

The explosion of data and the increasing complexity of data analysis generate a growing demand for parallel, scalable statistical analysis and machine learning tools that are simple and efficient. Simple tools need to be programmable, interactive, and extensible, allowing scientists to encode and deploy complex algorithms. Successful examples include R, SciPy, and Matlab. Efficiency dictates that tools should leverage modern computer architectures, including scalable parallelism, high-speed networking, and fast I/O from memory and storage. The current approach for utilizing the full capacity of modern parallel systems often uses a low-level programming language such as C and parallelizes computation with MPI or OpenMP. This approach is time-consuming and error-prone, and requires machine learning researchers to develop expertise in parallel programming models.

While conventional wisdom addresses large-scale data analysis and machine learning with clusters [1, 10, 12, 18, 37, 38], recent works [20, 21, 40, 42] demonstrate a single-machine solution can deal with large-scale data analysis efficiently in a multicore machine. The advance of solid-state drives (SSDs) allows us to tackle data analysis in a single machine efficiently at a larger scale and more economically than possible before. Previous SSD-based graph analysis frameworks [17, 40, 42] have demonstrated the comparable efficiency to state-of-the-art in-memory graph analysis, while scaling to arbitrarily large datasets. This work extends these findings to matrix operations using SSDs for machine learning and data analysis.

To provide a simple programming environment for efficient and scalable machine learning, we present FlashR, an interactive R-based programming framework that executes R code in parallel and out-of-core automatically. FlashR stores large vectors and matrices on SSDs and overrides many R functions in the R *base* package to perform computation on these external-memory vectors and matrices. As such, FlashR executes existing R code with little/no modification. FlashR focuses on optimizations in a single machine (with multiple CPUs and many cores) and scales matrix operations beyond memory capacity by utilizing SSDs. Our evaluation shows that we can solve billion row, Internet-scale problems on a single thick node, which can prevent the complexity, expense, and power consumption of distributed systems when they are not strictly necessary [21].

To utilize the full capacity of a large parallel machine, we overcome many technical challenges to move data from SSDs to CPU efficiently for matrix computations. Notably, there exist large performance disparities between CPU and memory and between memory and SSDs, at least an order of magnitude between every two layers. The "memory gap" [36] continues to grow, with the difference between CPU and DRAM performance increasing exponentially. There are also performance differences between local and remote memory in a non-uniform

PPoPP'18, Feb 24–28, 2018, Vienna, Austria
2018. ACM ISBN . . . $15.00
https://doi.org/

memory architecture (NUMA), which are prevalent in modern multiprocessor machines.

FlashR evaluates expressions lazily and fuses operations aggressively in a single parallel execution job to minimize data movement. FlashR builds a directed acyclic graph (DAG) to represent a sequence of matrix operations. To increase the ratio of computation to I/O, materialization of any matrix operation in a DAG triggers materialization of all operations in the DAG. FlashR requires only one pass over the input matrices of the DAG to perform all operations. FlashR by default keeps only the output matrices (leaf nodes) of the DAG in memory to have a small memory footprint. When materializing a DAG, FlashR assigns the same partitions from different matrices to the same NUMA node to reduce remote memory access, performs two levels of matrix partitioning and reorders computation on matrix partitions to reduce data movement in the memory hierarchy.

We implement multiple machine learning algorithms, including principal component analysis, logistic regression and k-means, in FlashR. We demonstrate that with today's fast commodity storage technology, the out-of-core execution of FlashR achieves performance comparable to their in-memory execution, both on a large parallel machine and Amazon cloud. Furthermore, FlashR outperforms the same algorithms in H_2O [13] and Spark MLlib [38] by a factor of $3 - 20$ in a large parallel machine with 48 CPU cores. In the Amazon cloud, FlashR using only one fourth of the resources still matches or even outperforms H_2O and Spark MLlib. This suggests that FlashR is a much more cost-effective solution for large-scale data analysis in the cloud. FlashR effortlessly scales to datasets with billions of data points and its out-of-core execution uses a negligible amount of memory compared with the dataset size. In addition, FlashR executes the R functions in the R MASS [19] package with little modification and outperforms the execution of the same functions in Revolution R Open [28] by more than an order of magnitude.

Given its high-level array-oriented programming interface and superior performance, we argue that FlashR significantly lowers the requirements for writing parallel and scalable implementations of machine learning algorithms. It also offers new design possibilities for data analysis clusters, replacing memory with larger and cheaper SSDs and processing bigger problems on fewer nodes. FlashR is released as an open-source project at http://flashx.io.

Our contributions include:

- We develop an R-based programming framework that executes native R code in parallel and out-of-core automatically.
- We design multiple techniques in our framework to move data from I/O storage to the CPU cache efficiently and demonstrate that with today's I/O technology, our SSD-based solution delivers performance approaching that of in-memory solutions for many machine learning algorithms.
- We demonstrate that with sufficient system-level optimizations, R code can easily scale to terabytes of data in a single machine and significantly outperform optimized parallel machine learning libraries.

2 Related Work

Basic Linear Algebra Subprograms (BLAS) defines a small set of vector and matrix operations. There exist a few highly-optimized BLAS implementations, such as MKL [23] and ATLAS [35]. Distributed libraries [2, 14, 26] build on BLAS and distribute computation with MPI. BLAS provides a limited set of matrix operations and requires users to manually parallelize the remaining matrix operations.

Recent works on out-of-core linear algebra [27, 32] redesign algorithms to achieve efficient I/O access and reduce I/O complexity. These works are orthogonal to our work and can be adopted. Optimizing I/O alone is insufficient. To achieve performance comparable to state-of-the-art in-memory implementations, it is essential to move data efficiently in the entire memory hierarchy.

MapReduce [10] has been used for parallelizing machine learning algorithms [8]. Even though MapReduce simplifies parallel programming, it still requires low-level programming. As such, frameworks, such as Pig Latin [24] and FlumeJava [6], are built on top of MapReduce to reduce programming complexity. MapReduce is inefficient for matrix operations because its I/O streaming primitives do not match matrix data access patterns.

Spark [38] is a distributed, in-memory framework that provides more primitives for efficient computation and provides a distributed machine learning library (MLlib [22]). Although Spark is efficient, it usually requires a large amount of RAM to process large datasets.

SystemML [4, 12] develops an R-like scripting language for machine learning on top of MapReduce and Spark. It deploys many optimizations, such as data compression [11] and hybrid parallelization [5]. These optimizations are orthogonal with the ones in FlashR and can be adopted.

Distributed machine learning frameworks have been developed to train machine learning models on large datasets. For example, GraphLab [18] formulates machine learning algorithms as graph computation; Petuum [37] is designed for machine learning algorithms with certain properties such as error tolerance; TensorFlow [1] trains machine learning models, especially deep neural

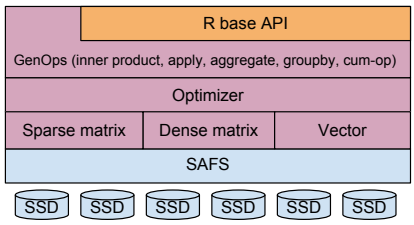


Figure 1. The architecture of FlashR.

networks, with optimization algorithms such as stochastic gradient descent.

Efforts to parallelize array programming include Revolution R [28] and Matlab's parallel computing toolbox, which offer multicore parallelism and explicit distributed parallelism using MPI and MapReduce. Other works focus on implicit parallelization. Presto [33] extends R to sparse matrix operations in distributed memory for graph analysis. Ching et. al [7] parallelize APL code by compiling it to C. Accelerator [31] compiles data-parallel operations on the fly to execute programs on a GPU.

OptiML [29] is a domain-specific language for machine learning in a heterogeneous computation environment such as multi-core processors and GPU. It designs a new programming language and relies on a compiler to generate code for the heterogenous environment.

3 Design

FlashR provides parallelized matrix operations in R for machine learning and statistics. It scales matrix operations beyond memory capacity by utilizing fast I/O devices, such as solid-state drives (SSDs), in a non-uniform memory access (NUMA) machine. Figure 1 shows the architecture of FlashR. FlashR supports a small number of classes of generalized operations (GenOps) and uses GenOps to implement many matrix operations in the R base package to provide users a familiar programming interface. The GenOps simplify the implementation and improve expressiveness of the framework. The optimizer aggressively merges operations to reduce data movement in the memory hierarchy. FlashR stores matrices on SSDs through SAFS [39], a user-space filesystem for SSD arrays, to fully utilize high I/O throughput of SSDs.

FlashR supports both sparse matrices and dense matrices. For large sparse matrices, FlashR integrates with the work [41] that stores sparse matrices in a compact format on SSDs and performs sparse matrix multiplication in semi-external memory, i.e., keeping the sparse matrix on SSDs and the dense matrix or part of the dense matrix in memory.

3.1 Dense matrices

FlashR optimizes for dense matrices that are rectangular—with a longer and shorter dimension—because of their frequent occurrence in machine learning and statistics.

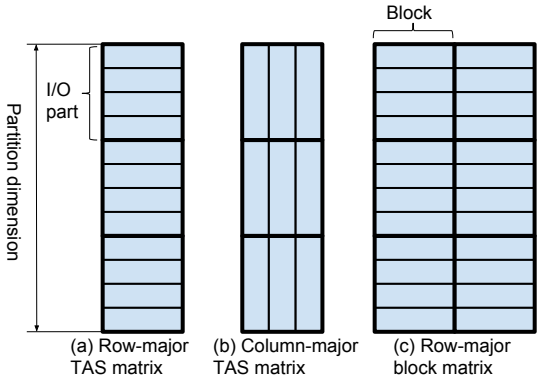


Figure 2. The format of a tall dense matrix.

Dense matrices can be stored physically in memory or on SSDs or represented virtually by a sequence of computations.

3.1.1 Tall-and-skinny (TAS) matrices

A data matrix may contain a large number of samples with a few features (tall-and-skinny), or a large number of features with a few samples (wide-and-short). We use similar strategies to optimize wide-and-short matrices. FlashR supports both row-major and column-major layouts (Figure 2(a) and (b)), which allows FlashR to transpose matrices without a copy. We store vectors as a one-column TAS matrix.

A TAS matrix is partitioned physically into I/O-partitions (Figure 2). We refer to the dimension that is partitioned as *partition dimension*. All elements in an I/O-partition are stored contiguously regardless of the data layout in the matrix. All I/O-partitions have the same number of rows regardless of the number of columns in a TAS matrix. The number of rows in an I/O-partition is always 2^i, where $i \in \mathbb{N}$. This produces column-major TAS matrices whose data are well aligned in memory to encourage CPU vectorization.

When a TAS matrix is stored in memory, FlashR stores its I/O partitions in fixed-size memory chunks (e.g., 64MB) across NUMA nodes. I/O partitions from different matrices may have different sizes. By storing I/O partitions in fixed-size memory chunks shared among all in-memory matrices, FlashR can easily recycle memory chunks to reduce memory allocation overhead.

When a TAS matrix is stored on SSDs, it is stored as a SAFS file [39]. For such a matrix, an I/O partition is accessed in a single I/O request. We rely on SAFS to map the data of a matrix evenly across SSDs. SAFS allows applications to specify the data mapping strategies. By default, FlashR uses a hash function to map data to SSDs to fully utilize the bandwidth of all SSDs even if we access only a subset of columns from a TAS matrix.

Table 1. Generalized operations (GenOps) in FlashR. A, B and C are matrices, and c is a scalar. f is a user-defined function that operates on elements of matrices. $A_{i,j}$ indicates the element in row i and column j of matrix A.

GenOp	Description
$C = sapply(A, f)$	$C_{i,j} = f(A_{i,j})$
$C = mapply(A, B, f)$	$C_{i,j} = f(A_{i,j}, B_{i,j})$
$c = agg(A, f)$	$c = f(A_{i,j}, c), \forall i, j$
$C = agg.row(A, f)$	$C_i = f(A_{i,j}, C_i), \forall j$
$C = agg.col(A, f)$	$C_j = f(A_{i,j}, C_j), \forall i$
$C = groupby(A, f)$	$C_k = f(A_{i,j}, C_k)$, where $A_{i,j} = k, \forall i, j$
$C = groupby.row(A, B, f)$	$C_{k,j} = f(A_{i,j}, C_{k,j})$, where $B_i = k, \forall i$
$C = groupby.col(A, B, f)$	$C_{i,k} = f(A_{i,j}, C_{i,k})$, where $B_j = k, \forall j$
$C = inner.prod(A, B, f1, f2)$	$t = f1(A_{i,k}, B_{k,j})$, $C_{i,j} = f2(t, C_{i,j}), \forall k$
$C = cum.row(A, f)$	$C_{i,j} = f(A_{i,j}, C_{i,j-1})$
$C = cum.col(A, f)$	$C_{i,j} = f(A_{i,j}, C_{i-1,j})$

3.1.2 Block matrices

FlashR stores a tall matrix as a *block matrix* (Figure 2(c)) comprised of TAS blocks with 32 columns each, except the last block. Each block is stored as a separate TAS matrix. We decompose a matrix operation on a block matrix into operations on individual TAS matrices to take advantage of the optimizations on TAS matrices and reduce data movement. Coupled with the I/O partitioning on TAS matrices, this strategy enables 2D-partitioning on a dense matrix and each partition fits in main memory.

3.2 Programming interface

FlashR provides a matrix-oriented functional programming interface built on Generalized Operations (GenOps). Instead of parallelizing each R matrix function individually, FlashR simplifies the implementation by providing a small set of GenOps and using the GenOps to parallelize R matrix functions. GenOps (Table 1) take matrices and some functions as input and output new matrices that represent computation results. The input function defines computation on individual elements in input matrices, and, in the current implementation, all of these functions for GenOps are predefined. GenOps provide a flexible and concise programming interface and, thus, we focus on optimizing the small set of matrix operations. All of the GenOps are lazily evaluated to gain efficiency (Section 3.4).

GenOps are classified into four categories that describe different data access patterns.

Element-wise operations: *sapply* is an element-wise unary operation; *mapply* is an element-wise binary operation.

Table 2. Some of the R matrix functions implemented with GenOps.

Function	Implementation with GenOps
$C = A + B$	$C = mapply(A, B, "+")$
$C = A - B$	$C = mapply(A, B, "-")$
$C = pmin(A, B)$	$C = mapply(A, B, "pmin")$
$C = pmax(A, B)$	$C = mapply(A, B, "pmax")$
$C = sqrt(A)$	$C = sapply(A, "sqrt")$
$C = abs(A)$	$C = sapply(A, "abs")$
$c = sum(A)$	$c = agg(A, "+")$
$C = rowSums(A)$	$C = agg.row(A, "+")$
$C = colSums(A)$	$C = agg.col(A, "+")$
$c = any(A)$	$c = agg(A, "\|")$
$c = all(A)$	$c = agg(A, "\&")$
$C = unique(A)$	$C = groupby(A, "uniq")$
$C = table(A)$	$C = groupby(A, "count")$
$C = A\%*\%B$	integers: $C = inner.prod(A, B, "*", "+")$ floating-points: BLAS sparse matrices: SpMM [41]

Aggregation: *agg* computes aggregation over all elements in a matrix and outputs a scalar value; *agg.row* computes over all elements in every row and outputs a vector; *agg.col* computes over all elements in every column and outputs a vector.

Groupby: *groupby* splits the elements of a matrix into groups, applies *agg* to each group and outputs a vector; *groupby.row* splits rows into groups and applies *agg.col* to each group; *groupby.col* splits columns into groups and applies *agg.row* to each group. Both *groupby.row* and *groupby.col* output a matrix.

Inner product is a generalized matrix multiplication that replaces multiplication and addition with two functions.

Cumulative operation performs computation cumulatively on the elements in rows or columns and outputs matrices with the same shape as the input matrices. Special cases in R are *cumsum* and *cumprod*.

With the GenOps, we reimplement a large number of matrix functions in the R *base* package. By overriding existing R matrix functions, FlashR scales and parallelizes existing R code with little/no modification. Table 2 shows a small subset of R matrix operations overridden by FlashR and their implementations with GenOps.

In addition to computation operations, FlashR provides other matrix functions, such as matrix reshaping and element access (Table 3). Like GenOps, FlashR tries to avoid data movement in most of these matrix operations. For example, transpose of a matrix in FlashR does not physically move elements in the matrix and, instead, FlashR just accesses data in the transposed matrix differently in the subsequent matrix operations. Reading columns from a tall matrix outputs a new matrix that indicates the columns to be accessed from the original matrix, instead of creating a new matrix that contains

Table 3. Some of the miscellaneous functions in FlashR for matrix creation, matrix access and etc.

Function	Description
rep.int	Create a vector of a repeated value
seq.int	Create a vector of sequence numbers
runif.matrix	Create a uniformly random matrix
rnorm.matrix	Create a matrix under a normal distribution
load.dense	Read a dense matrix from text files.
load.sparse	Read a sparse matrix from text file.
dim	Get the dimension information of a matrix
length	Get the number of elements in a matrix
t	Matrix transpose
rbind	Concatenate matrices by rows
cbind	Concatenate matrices by columns
[]	Get rows/columns/elements from a matrix
[] ←	Set rows/columns/elements from a matrix
set.cache	Set to cache materialized data
materialize	Materialize a *virtual matrix*

the specified columns. Writing data to a matrix is lazily evaluated and outputs a *virtual matrix* (see Section 3.4) that constructs the modified matrix on the fly.

3.3 Parallelize matrix operations

FlashR parallelizes matrix operations to achieve performance for both a single matrix operation and a sequence of matrix operations. When parallelizing a matrix operation on a large parallel NUMA machine with a deep memory hierarchy, FlashR aims at achieving good I/O performance and load balancing as well as reducing remote memory access in the NUMA architecture. All matrix operations are evaluated with a single pass over input data.

For good load balancing and I/O performance, FlashR uses a global task scheduler to assign I/O-partitions to threads dynamically. Initially, the scheduler assigns multiple contiguous I/O-partitions to a thread. The thread reads these in a single large I/O asynchronously. The number of contiguous I/O-partitions assigned to a thread is determined by the block size of SAFS. As the computation nears an end, the scheduler dispatches single I/O-partitions. The scheduler dispatches I/O-partitions sequentially to maximize contiguity on SSD. When FlashR writes an output matrix to SSDs, contiguity makes it easier for the file system to merge writes from multiple threads, which helps to sustain write throughput and reduces write amplification [30].

Parallelization strategies in FlashR vary based on the matrix operations and matrix shape because matrix operations have various data dependencies (Figure 3). Currently, FlashR parallelizes matrix operations with 1D partitioning, because most of machine learning datasets have many more samples than features.

- For operations illustrated in Figure 3 (a, b, c, d), a partition i of the output matrix solely depends on partitions i of the input matrices. This significantly simplifies parallelization. Partitions i of all matrices are assigned to the same thread to avoid remote memory access. There is no data sharing among threads.
- For operations in Figure 3 (e, f), a partition i of the output matrix still solely depends on a partition i of the input matrix A, but the input matrix B is shared by all threads. Because matrix B is read-only, threads do not need synchronization to perform computation. Matrix B is generally small, so FlashR always keeps it in memory.
- For operations in Figure 3 (g, h, i), the output matrix contains the aggregation over all partitions of the input matrices. To parallelize these operations, FlashR maintains a local buffer for the partial aggregation result in each thread and combines all partial results at the end of the computation. The local buffers may grow large in *groupby.row* because the size of partial aggregation results in each thread is determined by the number of different keys in the input data. As such, during computation, *groupby.row* merges partial aggregation in a local buffer to a global buffer when the size of a local buffer reaches a threshold.
- For cumulative operations on a tall matrix in Figure 3 (j), a partition i of the output matrix depends on a partition i of the input matrix as well as a partition $i-1$ of the output matrix. Executing this operation in parallel typically requires two passes over the input data [15]: the first pass builds a tree of aggregation bottom-up and the second pass traverses the tree top-down to compute the final result. To reduce I/O and evaluate this operation with other operations together (Section 3.5), FlashR performs computation in a cumulative operation with a single scan over input data. To perform cumulative operations efficiently in parallel, FlashR takes advantage of sequential task dispatching and asynchronous data access. FlashR maintains a current global accumulated result and a small set of local accumulated results from some partitions, and shares them among all threads. If the data that a partition i depends on is ready, a thread computes the partition and passes it to the subsequent matrix operations. Otherwise, a thread moves to the next partition $i+1$ and gets its dependency data ready. If the number of pending partitions reaches a threshold, a thread puts itself to sleep and waits for all dependency data to become available.

3.4 Lazy evaluation

In practice, FlashR almost never evaluates a single matrix operation alone. Instead, it always evaluates matrix

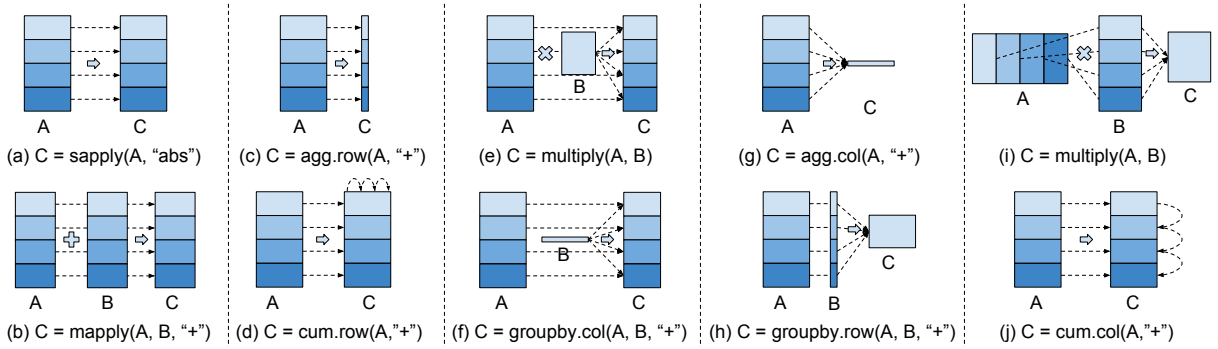


Figure 3. Data flow for the GenOps in Table 1 on tall matrices with 1D partitioning.

operations in Section 3.2 lazily and constructs directed acyclic graphs (DAG) to represent computation. Lazy evaluation is essential to achieve substantial performance for a sequence of matrix operations in a deep memory hierarchy. FlashR grows each DAG as large as possible so that later on it evaluates all matrix operations inside a DAG in a single parallel execution to increase the ratio of computation to I/O.

With lazily evaluation, matrix operations output *virtual matrices* that represent the computation result, instead of storing data physically. In the current implementation, the only operations that are not lazily evaluated are operations that load data from external sources, such as *load.dense* and *load.sparse*, and the operations that output matrices with the size depending on data of the input matrices, such as *unique* and *table*. *Virtual matrices* materialize data on the fly during computation and transfer output as input to the subsequent matrix operation. A matrix operation on a *block matrix* may output a block *virtual matrix*.

Some of the matrix operations output matrices with a different *partition dimension* size than the input matrices and, in general, forms the edge nodes of a DAG. We denote these matrices as *sink matrices*. Operations, such as aggregation and groupby, output *sink matrices*. *Sink matrices* tend to be small and, once materialized, their materialized results are stored in memory.

Figure 4 (a) shows an example of DAG that represents the computation of the k-means algorithm. A DAG comprises a set of matrix nodes (rectangles) and computation nodes (ellipses). The majority of matrix nodes are virtual matrices (dashed line rectangles). In this example, only the input matrix X has materialized data. A computation node references a GenOp and input matrices and may contain some immutable computation state, such as scalar variables and small matrices.

3.5 DAG materialization

FlashR evaluates all computation in a DAG in a single parallel execution and fuses matrix operations to reduce data movement in the memory hierarchy to achieve efficiency. This data-driven, operation fusion allows out-of-core problems to approach in-memory speed.

FlashR allows both explicit and implicit DAG materialization to simplify programming while providing the opportunity to tune the code for better speed. For explicit materialization, a user can invoke the *materialize* function in Table 3 to materialize a virtual matrix. Implicit materialization includes access to individual elements of a sink matrix and invoking matrix operations whose output size depends on input data, such as *unique* and *table*. Materialization on a virtual matrix triggers materialization on the DAG where the virtual matrix connects.

By default, FlashR saves the computation results of all sink matrices of the DAG in memory and discards the data of non-sink matrices on the fly. Because sink matrices tend to be small, this rule leads to small memory consumption. In exceptional cases, especially for iterative algorithms, it is helpful to save some non-sink matrices to avoid redundant computation and I/O across iterations. We allow users to set a flag on any virtual matrix with *set.cache* to materialize and cache data in memory or on SSDs during computation, similar to caching a resilient distributed dataset (RDD) in Spark [38]. Internally, FlashR uses this mechanism to lazily evaluate *runif.matrix*: FlashR generates random numbers when the matrix is used for the first time and the generated random numbers are cached so that the matrix returns the same data when the matrix is accessed again.

FlashR partitions matrices in a DAG and materializes partitions separately in most cases. This is possible because all matrices in a DAG except sink matrices share the same *partition dimension* and the same I/O partition size. As illustrated in Figure 4 (b), a partition i of a virtual matrix requires data only from partitions i of the parent matrices. All DAG operations in a partition are processed by the same thread so that all data required by the computations are stored and accessed in the memory

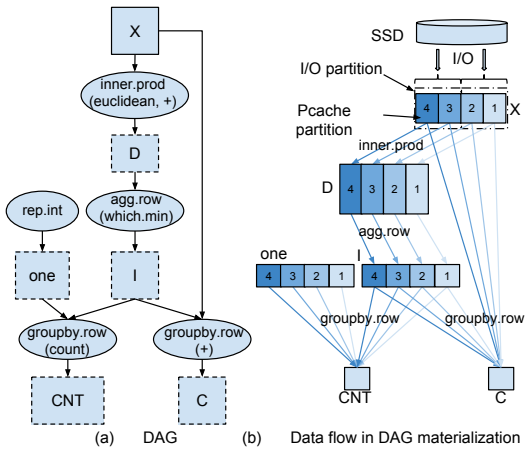


Figure 4. (a) Matrix operations are lazily evaluated to form a directed-acyclic graph (DAG); (b) The data flow in DAG materialization with two levels of partitioning: matrix X on SSDs is first partitioned and is read to memory in I/O partitions; an I/O partition is further split into processor cache (Pcache) partitions; once a Pcache partition is materialized, it is passed to the next GenOp to reduce CPU cache misses.

close to the processor to increase the memory bandwidth in a NUMA machine.

FlashR uses two-level partitioning on dense matrices to reduce data movement between SSDs and CPU (Figure 4 (b)). It reads data on SSDs in I/O partitions and assigns these partitions to a thread as a parallel task. It further splits I/O-partitions into processor cache (Pcache) partitions at runtime. Each thread materializes one Pcache-partition at a time from a matrix. Regular tall matrices are divided into TAS matrices and matrix operations are converted to running on these TAS matrices instead. As such, a Pcache-partition is sufficiently small to fit in the CPU L1/L2 cache.

To reduce CPU cache pollution and reduce data movement between CPU and memory, a thread performs depth-first traversal in a DAG and evaluates matrix operations in the order that they are traversed. Each time, a thread performs a matrix operation on a Pcache partition of a matrix and passes the Pcache partition to the subsequent matrix operation, instead of materializing the next Pcache partition. This ensures that a Pcache partition resides in the CPU cache when the next matrix operation consumes it. In each thread, all intermediate matrices have only one Pcache partition materialized at any time.

To further reduce CPU cache pollution, FlashR recycles memory buffers used by Pcache partitions in the CPU cache. FlashR maintains a counter on each Pcache partition. When the counter indicates the partition has

```
# X is the data matrix. C is cluster centers.
kmeans.iter <- function(X,C) {
  # Compute pair-wise distance.
  D<-inner.prod(X,t(C), "euclidean","+")
  # Find the closest center.
  I<-agg.row(D,"which.min")
  # Count the number of data points in each cluster.
  CNT<-groupby.row(rep.int(1,nrow(I)),I,"+")
  # Compute the new centers.
  C<-sweep(groupby.row(X,I,"+"),2,CNT,"/")
  list(C=C,I=I)
}
```

Figure 5. The FlashR implementation for an iteration of k-means.

been used by all subsequent matrix operations, the memory buffer of the partition is recycled and used to store the output of the next matrix operation. As such, the next matrix operation writes its output data in the memory that is already in CPU cache.

4 Machine learning algorithms

To illustrate the programming interface of FlashR, we showcase some classic algorithms written in FlashR. One is implemented with R *base* functions and can run in the existing R framework without any modification, and the other is written with GenOps.

K-means is a commonly used clustering algorithm that partitions data points into k clusters and minimizes the mean distance between the data points and the cluster center. We use this algorithm to illustrate programming with GenOps (Figure 5). It uses *inner.prod* to compute the Euclidean distance between every data point and every cluster center and outputs a matrix with each row representing the distances to centers. It uses *agg.row* to find the closest cluster for each data point. The output matrix assigns data points to clusters. It then uses *groupby.row* to count the number of data points in each cluster and compute the mean of each cluster.

Logistic regression is a commonly used classification algorithm. We implement this algorithm solely with the R *base* functions overridden by FlashR. Figure 6 implements logistic regression for problems with binary-class labels. It uses steepest gradient descent with line search to minimize the *cost* function. This example does not have a regularization term in the cost function.

5 Experimental evaluation

We evaluate the efficiency of FlashR on statistics and machine learning algorithms both in memory and on SSDs. We compare the R implementations of these algorithms with the ones in two optimized parallel machine learning libraries H_2O [13] and Spark MLlib [22]. We further use FlashR to accelerate existing R functions

```
# 'X' is the data matrix, whose rows represent
# data points and columns represent features.
# 'y' stores the labels of data points. 'w' is
# the weight vector.
logistic.regression <- function(X,y) {
  Xtw <- X%*%t(w)
  grad <- function(X,y,w)
    (t(X)%*%(1/(1+exp(-Xtw))-y))/length(y)
  cost <- function(X,y,w)
    sum(y*(-Xtw)+log(1+exp(Xtw)))/length(y)
  # Gradient descent with line search.
  theta <- matrix(rep(0, num.features), nrow=1)
  for (i in 1:max.iters) {
    g <- grad(X, y, theta)
    l <- cost(X, y, theta)
    eta <- 1
    delta <- 0.5 * (-g) %*% t(g)
    while (cost(X, y, theta+eta*(-g)) < l+delta*eta)
      eta <- eta * 0.2
    theta <- theta + (-g) * eta
  }
  theta
}
```

Figure 6. A simplified implementation of logistic regression using steepest gradient descent in FlashR.

in the MASS package and compare with Revolution R Open [28].

We conduct experiments on our local server and Amazon cloud. The local server is a large NUMA machine with four Intel Xeon E7-4860 2.6 GHz processors, each of which has 12 cores, and 1TB of DDR3-1600 memory. The machine is equipped with 24 OCZ Intrepid 3000 SSDs, which together are capable of 12 GB/s for read and 10 GB/s for write. We also run FlashR on an EC2 i3.16xlarge instance, which has 64 virtual CPUs, 488GB of RAM and 8 NVMe SSDs. The NVMe SSDs together provide 15.2TB of space and up to 16GB/s of sequential I/O throughput. We run Ubuntu 16.04 and use ATLAS 3.10.2 as the default BLAS library on both machines.

5.1 Benchmark algorithms

We benchmark FlashR with some commonly used machine learning algorithms. These algorithms have various ratios of computation and I/O complexity (Table 4) to thoroughly evaluate performance of FlashR on SSDs. Like the algorithms shown in Section 4, we implement these algorithms completely with the R code and rely on FlashR to execute them in parallel and out-of-core.

Correlation computes pair-wise Pearson's correlation [25] and is commonly used in statistics.

Principal Component Analysis (PCA) computes uncorrelated variables from a large dataset. PCA is commonly used for dimension reduction in many data analysis tasks. We compute PCA by computing eigenvalues on the Gramian matrix A^TA of the input matrix A.

Table 4. Computation and I/O complexity of the benchmark algorithms. For iterative algorithms, the complexity is per iteration. n is the number of data points, p is the number of the features in a point, and k is the number of clusters. We assume $n > p$.

Algorithm	Computation	I/O
Correlation	$O(n \times p^2)$	$O(n \times p)$
PCA	$O(n \times p^2)$	$O(n \times p)$
Naive Bayes	$O(n \times p)$	$O(n \times p)$
Logistic regression	$O(n \times p)$	$O(n \times p)$
K-means	$O(n \times p \times k)$	$O(n \times p)$
GMM	$O(n \times p^2 \times k)$	$O(n \times p + n \times k)$
mvrnorm	$O(n \times p^2)$	$O(n \times p)$
LDA	$O(n \times p^2)$	$O(n \times p)$

Naive Bayes is a classifier that applies Bayes' theorem with the "naive" assumption of independence between every pair of features. Our implementation assumes data follows the normal distribution.

Logistic regression is a linear regression model with categorical dependent variables. We use the LBFGS algorithm [16] to optimize logistic regression. In the experiments, it converges when $logloss_{i-1} - logloss_i < 1e-6$, where $logloss_i$ is the logarithmic loss at iteration i.

K-means is an iterative clustering algorithm that partitions data points to k clusters. Its R implementation is illustrated in Figure 5. In the experiments, we run k-means to split a dataset into 10 clusters by default. It converges when no data points move.

Gaussian mixture models (GMM) assumes data follows a mixture of Gaussian distribution and learns parameters of Gaussian mixture models from data. It typically uses the expectation-maximization (EM) algorithm [3] to fit the models, similar to k-means. In the experiments, it converges when $loglike_{i-1} - loglike_i < 1e-2$, where $loglike_i$ is the mean of log likelihood over all data points at iteration i.

Multivariate Normal Distribution (mvrnorm) generates samples from the specified multivariate normal distribution. We use the implementation in the MASS package.

Linear discriminant analysis (LDA) is a linear classifier that assumes the normal distribution with a different mean for each class but sharing the same covariance matrix among classes. We use the implementation in the MASS package with some trivial modifications.

5.2 Datasets

We use two real-world datasets with billions of data points (Table 5) to benchmark the algorithms above. The Criteo dataset has over four billion data points with binary labels (click vs. no-click), used for advertisement click prediction [9]. PageGraph-32ev are 32

Table 5. Datasets. All datasets are stored as dense matrices.

Data Matrix	#rows	#cols
PageGraph-32ev [34]	3.5B	32
Criteo [9]	4.3B	40
PageGraph-32ev-sub [34]	336M	32
Criteo-sub [9]	325M	40

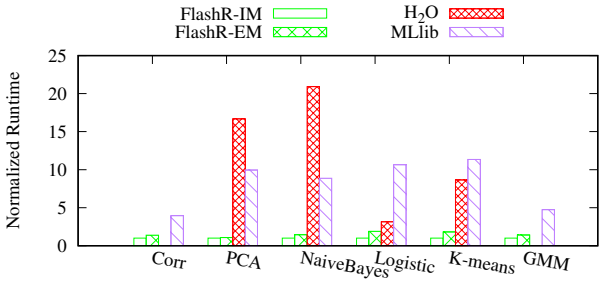
singular vectors that we computed on the largest connected component of a Page graph, which has 3.5 billion vertices and 129 billion edges [34]. Because Spark MLlib and H_2O cannot process the entire datasets in a single machine, we take part of the Criteo and PageGraph-32ev datasets to create smaller datasets. PageGraph-32ev-sub is the first 336 million data points of the PageGraph-32ev dataset. Criteo-sub contains the data points collected on the first two days, which is about one tenth of the whole dataset.

5.3 Comparative performance

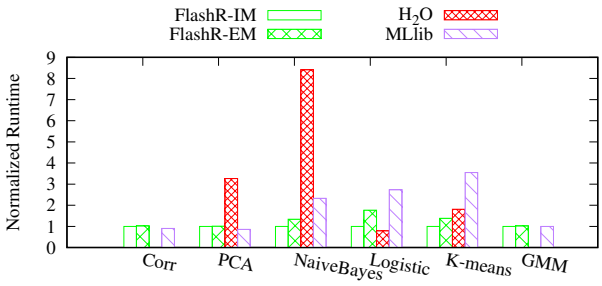
We evaluate FlashR against H_2O [13] and Spark MLlib [22] as well as Revolution R Open [28] in our local server and in the Amazon cloud. Before running the algorithms in H_2O and MLlib, we ensure that all data are loaded and cached in memory. When running in the 48 CPU core local server, all frameworks use 48 threads. When running in the cloud, we compare FlashR running in one of the largest I/O-optimized instance (i3.16xlarge) with MLlib and H_2O in a cluster with four of the largest general-purpose instances (m4.16xlarge), which in total has 256 CPU cores, 1TB RAM and 20Gbps network. We also use FlashR to parallelize functions (mvrnorm and LDA) in the R MASS package and compare their performance with Revolution R Open. We use Spark v2.0.1, H_2O v3.14.2 and Revolution R Open v3.3.2.

FlashR running on SSDs (FlashR-EM) achieves at least half the performance of running in memory (FlashR-IM), while outperforming H_2O and Spark MLlib significantly on all algorithms (Figure 7a) in the large parallel machine with 48 CPU cores. When all frameworks run on the same hardware, FlashR-IM achieves 4 to 10 times performance gain when compared with MLlib, and 3 to 20 times performance gain when compared with H_2O. All implementations rely on BLAS for matrix multiplication, and H_2O and MLlib implement non-BLAS operations with Java and Scala, respectively. Spark materializes operations such as aggregation separately. In contrast, FlashR fuses matrix operations and performs two-level partitioning to minimize data movement in the memory hierarchy and keeps data in CPU cache to achieve high memory bandwidth.

We further evaluate the performance of FlashR on Amazon EC2 cloud and compare it with Spark MLlib and H_2O on an EC2 cluster (Figure 7b). H_2O recommends



(a) In a large parallel machine with 48 CPU cores.



(b) In the Amazon cloud. FlashR-IM and FlashR-EM run on one EC2 i3.16xlarge instance (64 CPU cores) and Spark MLlib runs on a cluster of four EC2 m4.16xlarge instances (256 CPU cores).

Figure 7. The normalized runtime of FlashR in memory (FlashR-IM) and on SSDs (FlashR-EM) compared with H_2O and Spark MLlib. Correlation and GMM are not available in H_2O. We run k-means and GMM on the PageGraph-32ev-sub dataset and all other algorithms on the Criteo-sub dataset.

allocating a total of four times the memory of the input data. As such, we use 4 m4.16xlarge instances that provide sufficient memory and computation power for Spark MLlib and H_2O to process the datasets (PageGraph-32ev-sub and Criteo-sub). Even though Spark MLlib and H_2O have four times as much computation power as FlashR, FlashR still outperforms both distributed machine learning libraries in most algorithms. Because the NVMes in i3.16xlarge provide higher I/O throughput than the SSDs in our local server, the performance gap between FlashR-IM and FlashR-EM gets smaller.

FlashR running both in memory and on SSDs outperforms Revolution R Open by more than an order of magnitude even on a small dataset ($n = 1,000,000$ and $p = 1000$) (Figure 8). Revolution R Open uses Intel MKL to parallelize matrix multiplication. As such, we only compare the two frameworks with computations that use matrix multiplication heavily. Both FlashR and Revolution R Open run the mvrnorm and LDA implementations from the MASS package. For simple matrix operations such as crossprod, FlashR slightly outperforms Revolution R Open. For more complex computations, the

Table 6. The runtime and memory consumption of FlashR on the billion-scale datasets on the 48 CPU core machine. The runtime of iterative algorithms is measured when the algorithms converge. We run k-means on PageGraph-32ev and the remaining algorithms on Criteo.

	Runtime (min)	Peak memory (GB)
Correlation	1.5	1.5
PCA	2.3	1.5
NaiveBayes	1.3	3
LDA	38	8
Logistic regression	29.8	26
k-means	18.5	28
GMM	350.6	18

performance gap between FlashR and Revolution R increases. Even though matrix multiplication is the most computation-intensive operation in an algorithm, it is insufficient to only parallelize matrix multiplication to achieve high efficiency.

5.4 Scalability

We show the scalability of FlashR on the billion-scale datasets in Table 5. In these experiments, we run the iterative algorithms on the datasets until they converge (see their convergence condition in Section 5.1).

Even though we process the billion-scale datasets in a single machine, none of the algorithms are prohibitively expensive. Simple algorithms, such as Naive Bayes and PCA, require one or two passes over the datasets and take only one or two minutes to complete. Iterative algorithms in this experiment take about $10 - 20$ iterations to converge and finish computation quickly. Even though GMM is a computation-intensive algorithm, it does not take a prohibitively long time to complete.

FlashR scales to datasets with billions of data points easily when running out-of-core. All of the algorithms have negligible memory consumption. The scalability of FlashR is mainly bound by the capacity of SSDs. The functional programming interface generates a new matrix in each matrix operation, which potentially leads to high

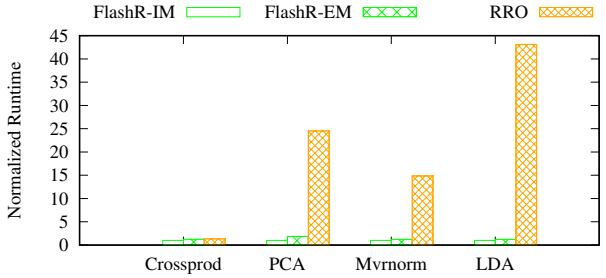


Figure 8. In-memory (FlashR-IM) and out-of-fore (FlashR-EM) FlashR compared with Revolution R Open on a data matrix with one million rows and one thousand columns when running on the parallel machine with 48 CPU cores.

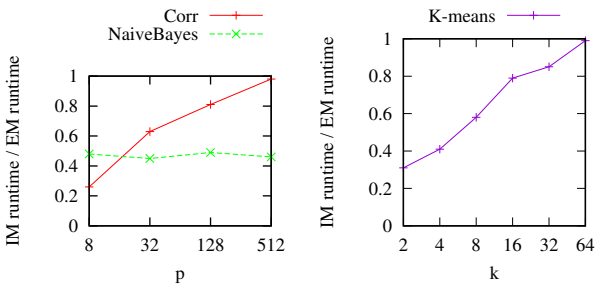


Figure 9. The relative runtime of FlashR in memory versus on SSDs on a dataset with $n = 100M$ while varying p (the number of features) on the left and varying k (the number of clusters) on the right.

memory consumption. Due to lazy evaluation and virtual matrices, FlashR only needs to materialize the small matrices to effectively reduce memory consumption.

5.5 Computation complexity versus I/O complexity

We further compare the performance of FlashR in memory and in external memory for algorithms with different computation and I/O complexities. We pick three algorithms from Table 4: *(i)* Naive Bayes, whose computation and I/O complexity are the same, *(ii)* correlation, whose computation complexity grows quadratically with p while its I/O complexity grows linearly with p, *(iii)* k-means, whose computation complexity grows linearly with k while its I/O complexity is independent from k. We run the first two algorithms on datasets with $n = 100M$ and p varying from 8 to 512. We run k-means on a dataset with $p = 100M$ and $p = 32$ and vary the number of clusters from 2 to 64.

As the number of features or clusters increases, the performance gap between in-memory and external-memory execution narrows and the external-memory performance approaches in-memory performance for correlation and k-means but not Naive Bayes (Figure 9). This observation conforms with the computation and I/O complexity of the algorithms in Table 4. For correlation and k-means, the number of clusters or features causes computation to grow more quickly than I/O, driving performance toward a computation bound. The computation bound is realized on few features or clusters for an I/O throughput of 10GB/s. Because most of the machine learning algorithms in Table 4 have computation complexities that grow quadratically with p, we expect FlashR on SSDs to achieve the same performance as in memory on datasets with a higher dimension size.

5.6 Effectiveness of optimizations

We illustrate the effectiveness of our memory optimizations in FlashR for DAG materialization. We focus on two main optimizations: matrix operation fusion in main

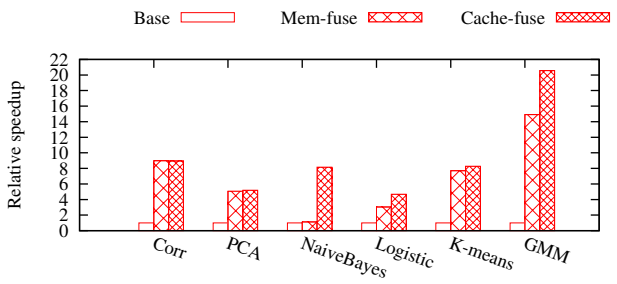


Figure 10. The speedup by the optimizations on different algorithms running on SSDs. The optimizations are applied to FlashR incrementally.

memory to reduce data movement between SSDs and main memory (mem-fuse), and matrix operation fusion in CPU cache to reduce data movement between main memory and CPU cache (cache-fuse). Due to the limit of space, we only illustrate their effectiveness when FlashR runs on SSDs.

Both optimizations have significant performance improvement on all algorithms (Figure 10). Operation fusion in main memory (mem-fuse) achieves substantial performance improvement in most algorithms, even in GMM, which has the highest asymptotic computation complexity. This indicates that materializing every matrix operation separately causes SSDs to be the main bottleneck in the system and fusing matrix operations in memory significantly reduces I/O. Operation fusion in the CPU cache (cache-fuse) has significant impact on the algorithms that are more complex and less bottlenecked by I/O. This suggests that memory bandwidth is a limiting performance factor once I/O has been optimized.

6 Conclusions

We present FlashR, a matrix-oriented programming framework that executes R-programmed machine learning algorithms in parallel and out-of-core automatically. FlashR scales to large datasets by utilizing commodity SSDs.

Although R is considered slow and unable to scale to large datasets, we demonstrate that with sufficient system-level optimizations, FlashR powers the R programming interface to achieve high performance and scalability for developing many machine learning algorithms. R implementations executed in FlashR outperform H_2O and Spark MLlib on all algorithms by a factor of $3 - 20$, using the same shared memory hardware. FlashR scales to datasets with billions of data points easily with negligible amounts of memory and completes all algorithms within a reasonable amount of time.

Even though the current I/O technologies, such as solid-state drives (SSDs), are an order of magnitude slower than DRAM, the external-memory execution of many algorithms in FlashR achieves performance approaching their in-memory execution. As the number of features and other factors, such as the number of clusters in clustering algorithms, increase, we expect FlashR on SSDs to achieve the same performance as in memory. We demonstrate that an I/O throughput of 10 GB/s saturates the CPU for many algorithms, even in a large parallel NUMA machine.

FlashR simplifies the programming effort of writing parallel and out-of-core implementations for large-scale machine learning. With FlashR, machine learning researchers can prototype algorithms in a familiar programming environment, while still getting efficient and scalable implementations. We believe FlashR provides new opportunities for developing large-scale machine learning algorithms.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[2] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, and Hong Zhang. 2015. PETSc Web page. http://www.mcs.anl.gov/petsc. (2015). http://www.mcs.anl.gov/petsc

[3] Jeff Bilmes. 1998. *A Gentle Tutorial of the EM Algorithm and its Application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models*. Technical Report. International Computer Science Institute.

[4] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1425–1436.

[5] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas R. Burdick, and Shivakumar Vaithyanathan. 2014. Hybrid Parallelization Strategies for Large-scale Machine Learning in SystemML. *Proc. VLDB Endow.* 7, 7 (March 2014), 553–564.

[6] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: Easy, Efficient Data-parallel Pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA.

[7] Wai-Mee Ching and Da Zheng. 2012. Automatic Parallelization of Array-oriented Programs for a Multi-core Machine. *International Journal of Parallel Programming* 40, 5 (2012), 514–531.

[8] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. 2006.

Map-reduce for Machine Learning on Multicore. In *Proceedings of the 19th International Conference on Neural Information Processing Systems*.

[9] criteo Accessed 2/11/2017. Criteo's 1TB Click Prediction Dataset. https://blogs.technet.microsoft.com/machinelearning/2015/04/01/now-available-on-azure-ml-criteos-1tb-click-prediction-dataset/. (Accessed 2/11/2017).

[10] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA.

[11] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2016. Compressed Linear Algebra for Large-scale Machine Learning. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 960–971.

[12] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. 2011. SystemML: Declarative Machine Learning on MapReduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA.

[13] H2O Accessed 2/7/2017. H2O machine learning library. http://www.h2o.ai/. (Accessed 2/7/2017).

[14] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. 2005. An Overview of the Trilinos Project. *ACM Trans. Math. Softw.* (2005).

[15] Richard E. Ladner and Michael J. Fischer. 1980. Parallel Prefix Computation. *J. ACM* 27, 4 (Oct. 1980), 831–838.

[16] D. C. Liu and J. Nocedal. 1989. On the limited memory BFGS method for large scale optimization. *Mathematical Programming: Series A and B* (1989).

[17] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA.

[18] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (2012).

[19] mass Accessed 2/12/2017. Package MASS. https://cran.r-project.org/web/packages/MASS/index.html. (Accessed 2/12/2017).

[20] Alexander Matveev, Yaron Meirovitch, Hayk Saribekyan, Wiktor Jakubiuk, Tim Kaler, Gergely Odor, David Budden, Aleksandar Zlateski, and Nir Shavit. 2017. A Multicore Path to Connectomics-on-Demand. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[21] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[22] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2015. MLlib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research* 17, 1 (2015).

[23] MKL Accessed 1/24/2016. Intel Math Kernel Library. https://software.intel.com/en-us/intel-mkl. (Accessed 1/24/2016).

[24] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA.

[25] Karl Pearson. 1895. Notes on regression and inheritance in the case of two parents. In *Proceedings of the Royal Society of London*. 240–242.

[26] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. 2013. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. *ACM Trans. Math. Softw.* 39, 2 (Feb. 2013), 13:1–13:24.

[27] Gregorio Quintana-Ortí, Francisco D. Igual, Mercedes Marqués, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. 2012. A Runtime System for Programming Out-of-Core Matrix Algorithms-by-Tiles on Multithreaded Architectures. *ACM Trans. Math. Softw.* 38, 4 (Aug. 2012), 25:1–25:25.

[28] rro Accessed 2/12/2017. Microsoft R Open. https://mran.microsoft.com/open/. (Accessed 2/12/2017).

[29] Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. 2011. OptiML: an implicitly parallel domainspecific language for machine learning. In *in Proceedings of the 28th International Conference on Machine Learning*.

[30] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ: Advanced Photo Caching on Flash for Facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA.

[31] David Tarditi, Sidd Puri, and Jose Oglesby. 2006. Accelerator: Using Data Parallelism to Program GPUs for General-purpose Uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA.

[32] Sivan Toledo. 1999. External Memory Algorithms. Boston, MA, USA, Chapter A Survey of Out-of-core Algorithms in Numerical Linear Algebra, 161–179.

[33] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S. Schreiber. 2013. Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, New York, NY, USA.

[34] webgraph Accessed 4/18/2014. Web graph. http://webdatacommons.org/hyperlinkgraph/. (Accessed 4/18/2014).

[35] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2000. Automated Empirical Optimization of Software and the ATLAS Project. *PARALLEL COMPUTING* 27 (2000), 2001.

[36] Maurice V. Wilkes. 2001. The Memory Gap and the Future of High Performance Memories. *SIGARCH Comput. Archit. News* 29, 1 (March 2001), 2–7.

[37] Eric P. Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[38] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets:

A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 15–28.

[39] Da Zheng, Randal Burns, and Alexander S. Szalay. 2013. Toward Millions of File System IOPS on Low-Cost, Commodity Hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*.

[40] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*.

[41] Da Zheng, Disa Mhembere, Vince Lyzinski, Joshua Vogelstein, Carey E. Priebe, and Randal Burns. 2016. Semi-External Memory Sparse Matrix Multiplication on Billion-node Graphs. *IEEE Transactions on Parallel & Distributed Systems* (2016).

[42] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*.