FlashR: Parallelize and Scale R for Machine Learning using SSDs

Da Zheng*
Amazon

Disa Mhembere
Dept. of Computer Science,
Johns Hopkins University

Joshua T. Vogelstein
Institute for Computational Medicine,
Dept. of Biomedical Engineering,
Johns Hopkins University

Carey E. Priebe
Dept. of Applied Math and Statistics,
Johns Hopkins University

Randal Burns
Dept. of Computer Science,
Johns Hopkins University

Abstract

R is one of the most popular programming languages for statistics and machine learning, but it is slow and unable to scale to large datasets. The general approach for having an efficient algorithm in R is to implement it in C or FORTRAN and provide an R wrapper. FlashR accelerates and scales existing R code by parallelizing a large number of matrix functions in the R *base* package and scaling them beyond memory capacity with solid-state drives (SSDs). FlashR performs memory hierarchy aware execution to speed up parallelized R code by *(i)* evaluating matrix operations lazily, *(ii)* performing all 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 various machine learning and statistics algorithms on inputs of up to four billion data points. Despite the huge performance gap between SSDs and RAM, FlashR on SSDs closely tracks the performance of FlashR in memory for many algorithms. The R implementations in FlashR outperforms H_2O and Spark MLlib by a factor of $3 - 20$.

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

*The work is done when the author was at Johns Hopkins University

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178501

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. These 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, 9, 12, 19, 35, 36], recent works [21, 22, 38, 40] demonstrate a single-machine solution can process large datasets 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, 38, 40] 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 for machine learning.

To provide a simple programming framework for efficient and scalable machine learning, we present FlashR, an interactive R programming framework that executes R code in parallel and out-of-core automatically. For generality and simplicity, FlashR implements a set of generalized operations (GenOps) and uses them to override many R functions in the R *base* package to perform parallel computation on large matrices stored on SSDs. As such, FlashR parallelizes and scales existing R code with little/no modification. Our evaluation shows that we solve billion row, Internet-scale problems on

a single thick machine, which prevents the complexity, expense, and power consumption of distributed systems when they are not strictly necessary [22].

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" [34] 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 memory architecture (NUMA), which are prevalent in modern multiprocessor machines.

FlashR implements a new runtime system that executes a sequence of matrix operations in a memory hierarchy aware fashion and optimizes data placement and movement in the memory hierarchy without users' awareness. To achieve this, FlashR evaluates expressions lazily and fuses operations aggressively in a single parallel execution job. FlashR builds a directed acyclic graph (DAG) to represent a sequence of matrix operations. To increase the ratio of computation to I/O, FlashR requires only one pass over the input matrices to perform all operations in a DAG. It 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. FlashR by default keeps only the output matrices (leaf nodes) of the DAG in memory to have a small memory footprint.

We implement multiple machine learning algorithms 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 in the cloud. Furthermore, FlashR outperforms the same algorithms in H_2O [14] and Spark MLlib [36] 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. We argue 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 [20] package with little modification and outperforms the execution of the same functions in Revolution R Open [27] 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 key contributions include:

- We develop an R programming framework that parallelizes and scales native R code 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

Recent works on out-of-core linear algebra [26, 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 state-of-the-art in-memory performance, it is essential to move data efficiently throughout the memory hierarchy.

Many distributed frameworks have beeen developed for large-scale data analysis and machine learning. MapReduce [9] is used for parallelizing machine learning algorithms [7]. However, MapReduce is inefficient for matrix operations because its I/O streaming primitives do not match matrix data access patterns. Spark [36] provides more primitives for efficient computation and are used for distributed machine learning (MLlib [23]). SystemML [3, 12] develops an R-like scripting language for machine learning on MapReduce and Spark, and deploys optimizations, such as data compression [10] and hybrid parallelization [4]. 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 [19] formulates machine learning algorithms as graph computation; Petuum [35] is designed for machine learning algorithms with certain properties such as error tolerance; TensorFlow [1] trains deep neural networks with stochastic gradient descent and its variants.

There is a large literature for deploying lazy evaluation and operation fusion in a programming framework to improve performance. There are a few attempts in the APL literature for deferred operations. For example, Guibas et al. [13] defers operations for streaming data among operations and reordering operations; Ching et al. [6] compiles APL code to fuse operations for better parallelization. Riposte [30] uses *tracing* to collect operations for vectorization and vector fusion with JIT to speed up operations on vectors in R.

Delite [5] is a system designed to parallelize domain-specific languages (DSL), such as OptiML [29] for machine learning, in a heterogeneous computation environment in a single machine. This system defers operation execution to allow both data and task parallelism. DESOLA [28] is a linear algebra library that defers matrix operations and deploys runtime code generation to fuse operations and arry constraction. All of the works above rely on compilation to achieve optimizations such as operation fusion or operation reordering. It is difficult to compile a dynamic programming language such as APL and R. The compilation is inefficent or requires some constraints in the language, while runtime compilation has large overhead. FlashR adopts and enhances these techniques with a focus on large-scale data analysis. Unlike most of these works, FlashR applies lazy evaluation and operation fusion at runtime without compilation and focuses on reducing data movement in the memory hierarchy.

Sequoia [11] is a programming language designed to facilitate memory hierarchy aware parallel programming on large arrays. It exposes memory hierarchy to the programming model and performs static analysis at compile time. In contrast, FlashR enhances an existing popular programming language and hides memory hierarchy from R users and optimize data movement at runtime.

TileDB's [24] designs an efficient strategy to support array modification. It manages data moficiation as "fragments". This strategy can be adopted by FlashR for large modifications on matrices.

3 Design

FlashR parallelizes and scales matrix operations in R for machine learning and statistics 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 [37], 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 [39] that performs sparse matrix multiplication in semi-external memory.

3.1 Programming interface

FlashR provides a matrix-oriented functional programming interface built on a small set of GenOps (Table 1). GenOps take matrices and some functions as input and output new matrices that represent computation results. Input functions

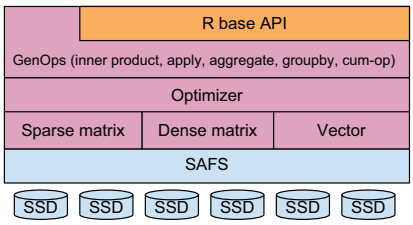


Figure 1. The architecture of FlashR.

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 $B_j = k, \forall j$
$C = inner.prod(A, B, f1, f2)$	$t = f1(A_{i,k}, B_{k,j})$, $C_{i,j} = f2(t, C_{i,j}), \forall k$
$C = cum.row(A, f)$	$C_{i,j} = f(A_{i,j}, C_{i,j-1})$
$C = cum.col(A, f)$	$C_{i,j} = f(A_{i,j}, C_{i-1,j})$

define computation on individual elements in matrices, and all of these functions for GenOps in the current implementation are predefined. All GenOps are lazily evaluated for better performance (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.

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

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*.

FlashR overrides a large number of matrix functions in the R *base* package with GenOps to scale and parallelize existing R code with little/no modification. Table 2 shows a small

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

Function	Implementation with GenOps
$C = A + B$	$C = mapply(A, B, "+")$
$C = pmin(A, B)$	$C = mapply(A, B, "pmin")$
$C = sqrt(A)$	$C = sapply(A, "sqrt")$
$c = sum(A)$	$c = agg(A, "+")$
$C = rowSums(A)$	$C = agg.row(A, "+")$
$c = any(A)$	$c = agg(A, "\|")$
$C = unique(A)$	$C = groupby(A, "uniq")$
$C = table(A)$	$C = groupby(A, "count")$
$C = A\%*\%B$	integers: $C = inner.prod(A, B, "*", "+")$ floating-points: BLAS sparse matrices: SpMM [39]

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

Function	Description
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.
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
[]	Get rows/columns/elements from a matrix
[] ←	Set rows/columns/elements from a matrix
materialize	Materialize a *virtual matrix*
set.cache	Set to cache materialized data
as.vector	Convert to an R vector
as.matrix	Convert to an R matrix

subset of R matrix operations overridden by FlashR and their implementations with GenOps.

FlashR provides a set of functions for matrix creation, element access and execution tuning (Table 3). Like GenOps, FlashR avoids data movement in most of these matrix operations. For example, transpose of a matrix only needs to change data access from the original matrix in the subsequent matrix operations [13]; reading columns from a tall matrix outputs a new matrix that indicates the columns to be accessed from the original matrix; writing to a matrix outputs a *virtual matrix* (see Section 3.4) that constructs the modified matrix on the fly. FlashR provides functions for tuning the execution of lazily evaluated operations. *materialize* forces FlashR to perform actual computation. *set.cache* informs FlashR to save the computation results of a matrix during computation. *as.vector* and *as.matrix* convert FlashR vectors and matrices to R vectors and matrices, which potentially force FlashR to perform computation.

3.1.1 Examples

We showcase some classic algorithms to illustrate the programming interface of FlashR.

```
# `X' is the data matrix, whose rows are data points.
# `y' stores the labels of data points.
logistic.regression <- function(X,y) {
  grad <- function(X,y,w)
    (t(X)%*%(1/(1+exp(-X%*%t(w)))-y))/length(y)
  cost <- function(X,y,w)
    sum(y*(-X%*%t(w))+log(1+exp(X%*%t(w))))/length(y)
  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)
    # Convert it to an R value for the while loop.
    l2 <- as.vector(cost(X, y, theta+eta*(-g)))
    while (l2 < as.vector(l)+delta*eta)
      eta <- eta * 0.2
    theta <- theta + (-g) * eta
  }
}
```

Figure 2. A simplified implementation of logistic regression using gradient descent with line search.

```
# X is the data matrix. C is cluster centers.
kmeans <- function(X,C) {
  I <- NULL
  num.moves > nrow(X)
  while (num.moves > 0) {
    D <- inner.prod(X, t(C), "euclidean", "+")
    old.I <- I
    I <- agg.row(D, "which.min")
    # Inform FlashR to save data during computation.
    I <- set.cache(I, TRUE)
    CNT <- groupby.row(rep.int(1, nrow(I)), I, "+")
    C <- sweep(groupby.row(X, I, "+"), 2, CNT, "/")
    if (!is.null(old.I))
      num.moves <- as.vector(sum(old.I != I))
  }
}
```

Figure 3. A simplified implementation of k-means.

Logistic regression is a commonly used classification algorithm. We implement this algorithm for binary-class problems and use gradient descent with line search to minimize the *cost* function. This implementation solely uses the R *base* functions overridden by FlashR (Figure 2) and can be executed in the existing R framework.

Figure 3 implements k-means, a popular clustering algorithm [18], with GenOps. It uses *inner.prod* to compute the Euclidean distance between data points and cluster centers and outputs a matrix whose rows represent the distances to centers. It uses *agg.row* to find the closest cluster for each data point. It then uses *groupby.row* to count the number of data points in each cluster and compute cluster centers.

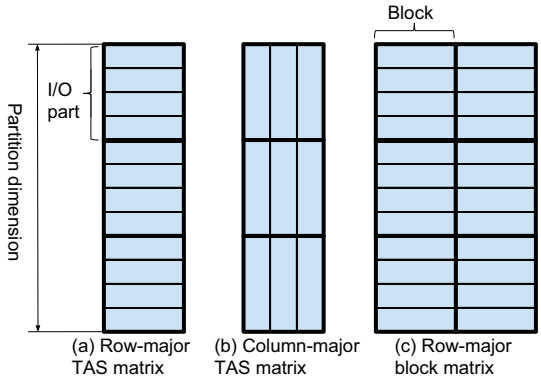


Figure 4. The format of a tall dense matrix.

3.2 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. Dense matrices are optimized for all types of storage, including NUMA memory and SSDs.

3.2.1 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 these two types of matrices. FlashR supports both row-major and column-major layouts (Figure 4(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 4). We refer to the dimension that is partitioned as the *partition dimension*. All elements in an I/O-partition are stored contiguously regardless of the data layout. All I/O-partitions have the same number of rows regardless of the number of columns. The number of rows in an I/O-partition is 2^i, where $i \in \mathbb{N}$. This produces column-major TAS matrices whose data are well aligned in memory to encourage CPU vectorization.

FlashR stores the I/O partitions of an in-memory matrix 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 and reduce memory allocation overhead.

FlashR stores an SSD-based matrix as a SAFS file [37]. An I/O partition is accessed asynchronously with direct I/O to bypass the Linux page cache for better I/O performance. We rely on SAFS to map the data of a matrix evenly across SSDs. By default, we use a hash function to map data to fully utilize the bandwidth of all SSDs even if we access only a subset of columns from a TAS matrix.

3.2.2 Block matrices

FlashR stores a tall matrix as a *block matrix* (Figure 4(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.

3.3 Parallelize matrix operations

When executing matrix operations in parallel, FlashR aims at achieving good I/O performance and load balancing as well as reducing remote memory access in a NUMA machine. FlashR evaluates a matrix operation with a single pass over input data.

For good load balancing and I/O performance, FlashR uses a global task scheduler to dispatch I/O-partitions to threads sequentially and dynamically. Initially, the scheduler assigns multiple contiguous I/O-partitions to a thread. The thread reads them in a single 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 increase contiguity on SSD. When FlashR writes data 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 [31].

Parallelization strategies in FlashR are based on the matrix operations and matrix shape because matrix operations have various data dependencies (Figure 5).

Operations (a, b, c, d): a partition i of the output matrix solely depends on partitions i of the input matrices. This simplifies parallelization. FlashR assigns partitions i of all matrices to the same thread to avoid remote memory access. There is no data sharing among threads.

Operations (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 B is read-only, computation does not require synchronization. B is generally small and FlashR keeps it in memory.

Operations (g, h, i): the output matrix contains the aggregation over all partitions of the input matrices. To parallelize these operations, each thread maintains a local buffer for partial aggregation results. FlashR combines all partial results at the end of the computation.

Cumulative operations (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

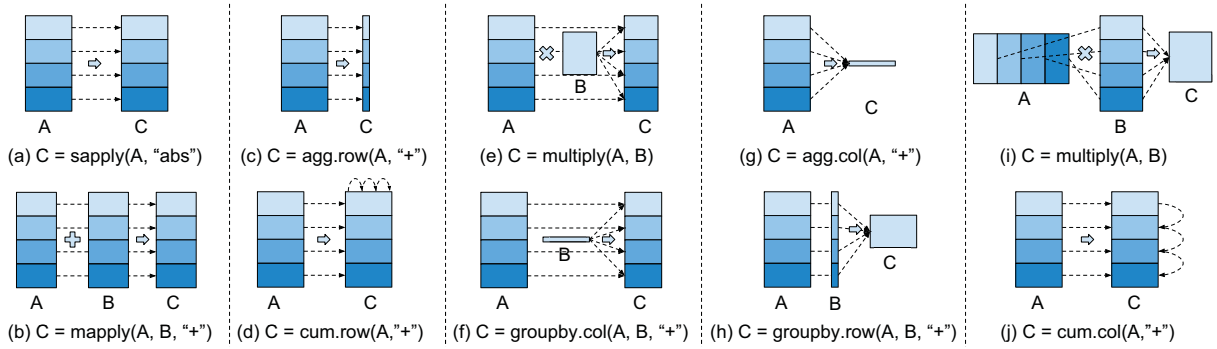


Figure 5. Data flow for the GenOps in Table 1 on tall matrices with 1D partitioning.

[15]. To reduce I/O and fuse this operation with others (Section 3.5), FlashR performs this operation with a single scan over input by taking advantage of sequential task dispatching and asynchronous I/O. FlashR maintains and shares a current global accumulated result and a small set of local accumulated results among all threads. If the data that a partition i depends on is ready, a thread computes this partition. Otherwise, a thread moves to the next partition $i + 1$. If the number of pending partitions reaches a threshold, a thread sleeps 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 evaluates matrix operations, such as GenOps, 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 and 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 the ones that load data from external sources, such as *load.dense*, and the ones that output matrices with the size depending on data of the input matrices, such as *unique* and *table*. An 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, store results in memory.

Figure 6 (a) shows an example of DAG that represents the k-means computation in a single iteration (Figure 3). 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.

FlashR stops constructing a DAG and starts to materialize the computation in the DAG when encountering the following functions: *(i) materialize* to materialize a virtual matrix; *(ii) as.vector* and *as.matrix* to convert to R objects; *(iii)* access to individual elements of a sink matrix; *(iv) unique* and *table*, whose output size depends on input data. The first two cases give users the opportunity to control DAG materialization for better speed, while the last two cases are implicit DAG materialization to simplify programming.

3.5 DAG materialization

When computation is triggered, we evaluate all operations in a DAG to increase the ratio of computation to I/O. FlashR fuses all operations in a DAG into a single parallel execution to reduce data movement in the memory hierarchy. This data-driven, operation fusion allows out-of-core problems to approach in-memory speed.

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 cache data in memory or on SSDs during computation, similar to caching a resilient distributed dataset (RDD) in Spark [36]. Figure 3 shows an example that we cache I, a vector of partition Ids assigned to each data point, for the next iteration.

3.5.1 Reduce data movement

FlashR performs memory hierarchy aware execution, when evaluating operations in a DAG, to reduce data movement

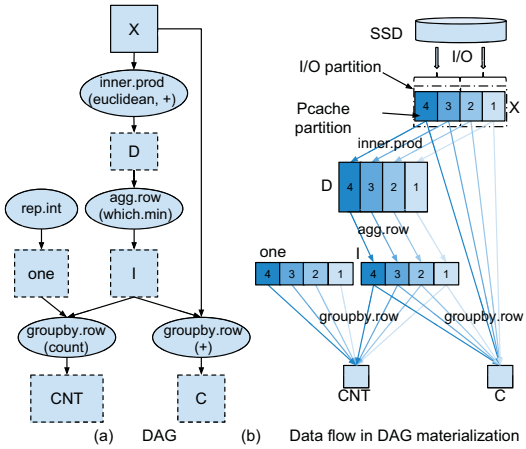


Figure 6. (a) Matrix operations are lazily evaluated to form a directed-acyclic graph (DAG); (b) The data flow in DAG materialization with two levels of partitioning: matrix X on SSDs is first partitioned and is read to memory in I/O partitions; an I/O partition is further split into processor cache (Pcache) partitions; once a Pcache partition is materialized, it is passed to the next GenOp to reduce CPU cache misses.

between SSDs and RAM, between NUMA nodes as well as between RAM and CPU cache.

FlashR materializes matrix partitions separately in a DAG 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 6 (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 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 6 (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 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 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 [14] and Spark MLlib [23]. We use FlashR to accelerate existing R functions in the MASS package and compare with Revolution R Open [27].

We conduct experiments on our local server and Amazon cloud. The local server has four Intel Xeon E7-4860 2.6 GHz processors, each of which has 12 cores, and 1TB of DDR3-1600 memory. It 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 run FlashR on an EC2 i3.16xlarge instance with 64 virtual CPUs, 488GB of RAM and 8 NVMe SSDs. The NVMe SSDs together provide 15.2TB of space and 16GB/s of sequential I/O throughput. We run Ubuntu 16.04 and use ATLAS 3.10.2 as the default BLAS library.

4.1 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. Like the algorithms shown in Section 3.1.1, we implement these algorithms completely with the R code and rely on FlashR to execute them in parallel and out-of-core. We implement these algorithms identically to our competitors (SparkML and H2O).

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

Principal Component Analysis (PCA) is commonly used for dimension reduction in many data analysis tasks. We implement PCA by computing eigenvalues on the Gramian matrix A^TA of the input matrix A.

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 for classification. We use the LBFGS algorithm [16] for optimization. In

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 dimensions 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	$O(n \times p^2)$	$O(n \times p)$

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 into k clusters. 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 [2] 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 a multivariate normal distribution. We use the implementation in the MASS package.

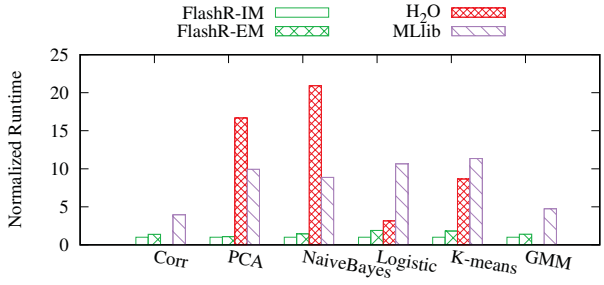
Linear discriminant analysis (LDA) is a linear classifier that assumes the normal distribution with a different mean for each class but sharing the same covariance matrix among classes. We use the implementation in the MASS package with some trivial modifications.

4.2 Datasets

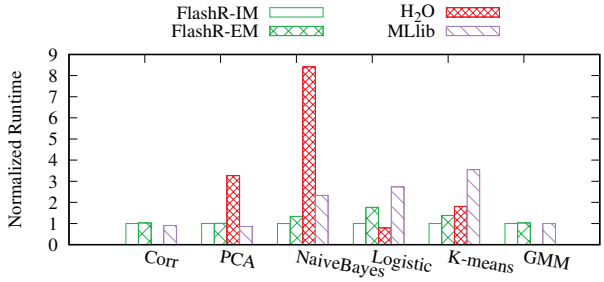
We use two real-world datasets with billions of data points (Table 5) to benchmark the algorithms. The Criteo dataset has over four billion data points with binary labels (click vs. no-click), used for advertisement click prediction [8]. PageGraph-32ev are 32 singular vectors that we computed on the largest connected component of a Page graph, which has 3.5 billion vertices and 129 billion edges [33]. Because Spark MLlib and H_2O cannot process the entire datasets in a single machine, we take part of these two datasets to create smaller ones. 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.

Table 5. Benchmark datasets.

Data Matrix	#rows	#cols
PageGraph-32ev [33]	3.5B	32
Criteo [8]	4.3B	40
PageGraph-32ev-sub [33]	336M	32
Criteo-sub [8]	325M	40



(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.

4.3 Comparative performance

We evaluate FlashR against H_2O [14], Spark MLlib [23] and Revolution R Open [27] 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. All frameworks use 48 threads in the local server. In the cloud, we run FlashR in one i3.16xlarge instance and MLlib and H_2O in a cluster with four m4.16xlarge instances, which in total has 256 CPU cores, 1TB RAM and 20Gbps network. All iterative algorithms take the same number of iterations and generate similar accuracy[1]. 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.

[1]The only exception is the logistic regression in Spark because we cannot control its number of iterations

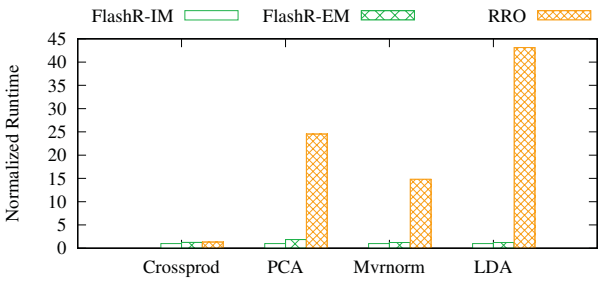


Figure 8. The normalized runtime of FlashR-IM and FlashR-EM compared with Revolution R Open on a data matrix with one million rows and one thousand columns on the 48 CPU core server.

FlashR on SSDs (FlashR-EM) achieves at least half the performance of running in memory (FlashR-IM), while outperforming H_2O[2] and Spark MLlib significantly on all algorithms (Figure 7a) in the 48 CPU core server. In the same hardware, FlashR-IM achieves 4 to 10 times performance gain compared with MLlib, and 3 to 20 times performance gain compared with H_2O. All implementations rely on BLAS for matrix multiplication, but H_2O and MLlib implement non-BLAS operations with Java and Scala. 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.

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

FlashR 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 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.

[2] H_2O develops machine learning algorithms individually and adding a new algorithm in H_2O requires writing it from scratch, which is a non-trivial task. H_2O does not provide implementations for correlation and GMM, so we do not provide results for these two algorithms.

Table 6. The runtime and memory consumption of FlashR on the billion-scale datasets on the 48 CPU core machine. We measure the runtime of iterative algorithms when they converge. We run k-means and GMM 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

4.4 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 4.1).

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 $10 - 20$ iterations to converge. Even GMM, a computation-intensive algorithm, 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. Two factors contributes to the small memory consumption: FlashR only saves materialized results of sink matrices; FlashR uses direct I/O to access data from SSDs and does not cache data internally.

4.5 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 the number of dimensions p while its I/O complexity grows linearly with p, *(iii)* k-means, whose computation complexity grows linearly with the number of clusters 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

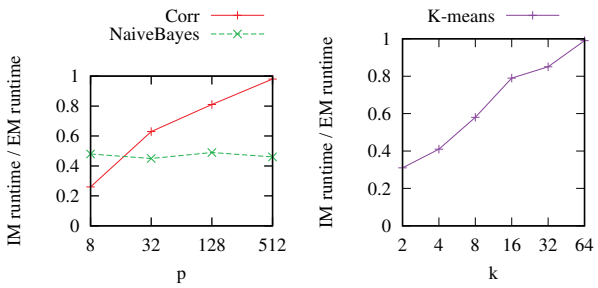


Figure 9. The relative runtime of FlashR in memory versus on SSDs on a dataset with $n = 100M$ while varying p (the number of dimensions) on the left and varying k (the number of clusters) on the right.

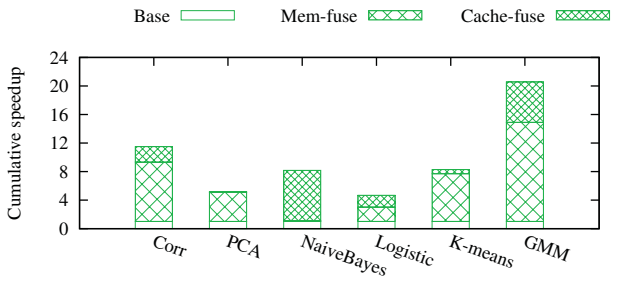


Figure 10. The relative speedup by applying the optimizations in FlashR incrementally over the base implementation running on SSDs. The base implementation does not have optimizations to fuse matrix operations.

k-means on a dataset with $n = 100M$ and $p = 32$ and vary the number of clusters from 2 to 64.

As the number of dimensions or the number of 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, increasing p or k causes computation to grow more quickly than I/O, driving performance toward a computation bound. The computation bound is realized on few dimensions 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.

4.6 Effectiveness of optimizations

We illustrate the effectiveness of memory hierarchy aware execution in FlashR. We focus on two main optimizations: operation fusion in memory to reduce data movement between SSDs and memory (mem-fuse), and operation fusion in CPU cache to reduce data movement between 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). 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. Cache-fuse has significant impact on the algorithms that are more complex and less bottlenecked by I/O. This demonstrates that memory bandwidth is a limiting performance factor once I/O is optimized.

5 Conclusions

We present FlashR, a matrix-oriented programming framework that executes 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 achieves high performance and scalability for many machine learning algorithms. R implementations executed in FlashR outperform H_2O and Spark MLlib on all algorithms by a factor of 3−20. 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. 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.

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.

6 Acknowledgements

We would like to thank the PPoPP reviewers for their insightful comments. This work is supported by NSF Grant # 1649880.

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] 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.

[3] 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.

[4] 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.

[5] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. 2011. A Domain-Specific Approach to Heterogeneous Parallelism. In *Proceedings of the 16th Annual Symposium on Principles and Practice of Parallel Programming*.

[6] 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.

[7] 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*.

[8] 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).

[9] 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.

[10] 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.

[11] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*.

[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] Leo J. Guibas and Douglas K. Wyatt. 1978. Compilation and Delayed Evaluation in APL. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*.

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

[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] S. Lloyd. 2006. Least Squares Quantization in PCM. *IEEE Trans. Inf. Theor.* 28, 2 (Sept. 2006).

[19] 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).

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

[21] 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*.

[22] 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)*.

[23] 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).

[24] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. 2016. The TileDB Array Data Storage Manager. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 349–360.

[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] 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.

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

[28] Francis P. Russell, Michael R. Mellor, Paul H. J. Kelly, and Olav Beckmann. 2011. DESOLA: An Active Linear Algebra Library Using Delayed Evaluation and Runtime Code Generation. *Sci. Comput. Program.* (2011).

[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] J. Talbot, Z. DeVito, and P. Hanrahan. 2012. Riposte: A trace-driven compiler and parallel VM for vector code in R. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[31] 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.

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

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

[35] 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.

[36] 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.

[37] 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*.

[38] 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)*.

[39] 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).

[40] 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)*.