BLAS-on-flash : An Efficient Alternative for Large Scale ML Training and Inference?

Suhas Jayaram Subramanya
Microsoft Research India
t-sujs@microsoft.com

Harsha Vardhan Simhadri
Microsoft Research India
harshasi@microsoft.com

Srajan Garg
IIT Bombay
srajan.garg@gmail.com

Anil Kag
Microsoft Research India
t-anik@microsoft.com

Venkatesh Balasubramanian
Microsoft Research India
t-venkb@microsoft.com

Abstract

Many large scale machine learning training and inference tasks are memory-bound rather than compute-bound. That is, on large data sets, the working set of these algorithms does not fit in memory for jobs that could run overnight on a few multi-core processors. This often forces an expensive redesign of the algorithm for distributed platforms such as parameter servers and Spark.

We propose an inexpensive and efficient alternative based on the observation that many ML tasks admit algorithms that can be programmed with linear algebra subroutines. A library that supports BLAS and sparse-BLAS interface on large SSD-resident matrices can enable multi-threaded code to scale to industrial scale datasets on a single workstation.

We demonstrate that not only can such a library provide near in-memory performance for BLAS, but can also be used to write implementations of complex algorithms such as eigensolvers that outperform in-memory (ARPACK) and distributed (Spark) counterparts.

Existing multi-threaded in-memory code can link to our library with minor changes and scale to hundreds of gigabytes of training or inference data at near in-memory processing speeds. We demonstrate this with two industrial scale use cases arising in ranking and relevance pipelines: training large scale topic models and inference for extreme multi-label learning.

This suggests that our approach could be an efficient alternative to expensive distributed *big-data* systems for scaling up structurally complex machine learning tasks.

1 Introduction

Data analysis pipelines in scientific computing as well as ranking and relevance often work on datasets that are hundreds of gigabytes to a few terabytes in size. Many algorithms in these pipelines, such as topic modeling [6], matrix factorizations [33], spectral clustering [32], extreme multi-label learning [47], are memory limited as opposed to being limited by compute. That is, on large datasets, a training algorithm that requires a few hours of compute on a multi-core workstation would run out of DRAM for its working set.

This forces users to move the algorithm to distributed big-data platforms such as Apache Spark [63, 64] or systems based on Parameter Servers [18, 37, 60], which incurs three costs: (1) the cost of rewriting code in a distributed framework, (2) the cost of a cluster of nodes or non-availability in production environments, and (3) inefficiencies of the platform in using the hardware. Training on these platforms can require dozens of nodes for moderate speedups over single threaded code for non-trivial algorithms [22, 39]. This could be due to platform overheads as well as mismatch between the structure of the algorithm and the platform's programming model [9, 17, 58], resulting in low processor utilization.

Several light-weight frameworks for single node workstations demonstrate that this inefficiency is unnecessary for many classes of algorithms that admit multi-threaded implementations that are orders of magnitude more efficient [16, 34, 52, 53]. It is also widely observed that many machine learning problems admit algorithms that are essentially compositions of linear algebra operations on sparse and dense matrices. High performance implementations of these algorithms typically invoke linear-algebra operations through standard APIs such as BLAS [10] and sparseBLAS [20]. High performance implementations for these standard APIs are provided by hardware vendors [26, 27, 43, 44].

Linear algebra kernels offer plenty of locality, so much so that the bandwidth required to run them on high-end multiprocessors can be provided by a non-volatile memory over PCIe or SATA bus [5, 13, 56]. Non-volatile memory is already widely deployed in cloud and developments in hardware and software eco-system position non-volatile memory as an inexpensive alternative to DRAM [4, 19, 49, 50]. Hardware technology and

interfaces for non-volatile memories have increasingly lower end-to-end latency (few μs) [25] and higher bandwidth: from 8 GT/s in PCIe3.0 to 16GT/s in PCIe4.0 [45] and 32GT/s in PCIe5.0. Hardware manufactures are also packaging non-volatile memory with processing units, e.g. Radeon PRO SSG [2] to increase available memory.

These observations point to a cost-effective solution for scaling linear algebra based algorithms to large datasets in many scenarios – use inexpensive PCIe-connected SSDs to store large matrices corresponding to the data and the model, and exploit the locality of linear algebra to develop a library of routines that can operate on these matrices with a limited amount of DRAM. By conforming to the standard APIs, such a library could be a replacement for code that would have linked to BLAS libraries such as Intel MKL or OpenBLAS [59].

We present empirical evidence that this approach can be practical, easy, and fast, by developing a library which provides near in-memory speeds on NVM-resident data for subroutines on dense matrices and sparse matrices.

Performance of our *BLAS-on-flash* library is comparable to that of in-memory Intel MKL implementations for level-3 BLAS and sparseBLAS kernels such as `gemm` (dense-dense matrix multiplication) and `csrmm` (sparse-dense matrix multiplication) on multiprocessor machines with SSDs. The key to this performance is using the knowledge of data-access patterns arising in linear algebra kernels to effectively pipeline IO with computation. Using these kernels, we can implement algorithms such as k-means clustering that run at near in-memory speeds.

To illustrate that this approach is not limited to simple kernels, we consider one of the most structurally complex numerical algorithms – eigensolvers. Using the *BLAS-on-flash* library, we built a general purpose symmetric eigensolver, which is critical to dimensionality reduction (e.g. PCA) and spectral methods. Specifically, we adapted the restarted block Krylov-Schur [67] algorithm to compute thousands of eigenvectors on SSD-resident data faster than standard in-memory solvers based on the IRAM algorithm [54] (e.g., Spectra [48], ARPACK [35]). On large bag of words text datasets running into hundreds of gigabytes, our implementation running on one multi-core workstation with under 50GB DRAM outperforms Spark MLlib's `computeSVD` [40] deployed on hundreds of executors, representing an order of magnitude efficiency gain in hardware utilization. Further, our solver can compute thousands of eigenvalues, while `computeSVD` is limited to 500 or fewer.

We present two use cases of the library for algorithms used in ranking and relevance pipelines that process hundreds of gigabytes of data: training topic models, and inference in Extreme Multi-Label learning.

Topic modeling [11] summarizes a corpus of documents, where each document is a collection of words from a fixed vocabulary, as a set of *topics* that are probability distributions over the vocabulary. Although most large scale algorithms are based on approximating and scaling an intractable probabilistic model on parameter servers [14, 61, 62], recent research [6] has shown that linear algebra based approaches can be just as good qualitatively. We take a highly optimized version of the algorithm in [6] that already outperforms prior art on single node workstations, and link to the eigensolvers and clustering algorithms written using our framework. This allows the algorithm to train a 2000 topic model on a 60 billion token corpus (500GB on disk) in under 4 hours.

Extreme Multi-Label Learning (XML) is the problem of learning to automatically annotate a data point with the most relevant subset of labels from an extremely large label set (often many millions of labels). This is an important task with many applications in tagging, ranking, and recommendation [8]. Models in extreme multi-label learning tasks are often ensembles of deep trees with small classifier(s) at each node. e.g. PfastreXML [47], Parabel [46]. In production, models that exceed DRAM in size need to score (i.e. infer) several hundreds of millions sparse data points from a space with million+ dimensions every week on a platform that provides machines with moderate sized DRAM. As datasets grow in size, XML algorithms need to scale 10x along multiple axes: model size, number of points scored and dimensionality of the data.

In this work, we start with PfastreXML and Parabel models and a dataset that needed 440 and 900 compute hours respectively on a VM with large RAM. We optimized this code to reduce in-memory run time by a factor of six. When the optimized code is linked to our library, it runs at about 90% of in-memory speed with a much smaller memory footprint.

These results suggest that, for complex numerical algorithms, our approach is capable of running at near in-memory speeds on large datasets while providing significant benefits in hardware utilization as compared to general-purpose big-data systems. Further, we envision our library being useful in the following scenarios: (1) Environments without multi-node support for MPI, Spark etc., (2) Laptops and workstations or VMs in cloud with limited RAM but large non-volatile memories, (3) Batch mode periodic retraining and inference of large scale models in production data analysis pipelines, (4) Extending the capabilities of legacy single-node ML training code.

Roadmap. Sections 2, 3 and 4 provide an overview of the interface, design, and the architecture of the library. Section 5 presents an evaluation of the performance of our library and algorithms written using the library.

Source code for our library has been released at `github.com/Microsoft/BLAS-on-flash`.

2 *BLAS-on-flash* : Overview and Interface

The *BLAS-on-flash* library provides an easy way to write external memory parallel algorithms, especially numerical algorithms processing large matrices, that run at near in-memory speed on SSD-resident data. At its core, it pipelines calls to an existing math library (like Intel MKL or OpenBLAS) on in-memory data blocks. Coupled with prefetching and intelligent scheduling, *BLAS-on-flash* allows the programmer to define computation on inputs that are limited only by the size of storage.

Our library is intended for programmers who already write multi-threaded code in C++ using shared memory pointers. *BLAS-on-flash* provides a rich interface utilizing C++ templates and inheritance to allow easy integrations with existing code with minimal modifications.

Typically, programmers writing high-performance native code track data objects with *pointers* and manipulate these objects by passing their pointers to functions or linked libraries that perform operations such as matrix multiplication.

The *BLAS-on-flash* library provides a custom pointer type, `flash_ptr<T>`, to track large SSD-resident objects, and replaces the standard `T*` pointer type. A programmer can either invoke *BLAS-on-flash* library functions operating on `flash_ptr<T>` types or define new functions that operate on `flash_ptr<T>` types by specializing the `Task` class. The `Task` class allows a programmer to define inputs, outputs, and a compute function mapping inputs to outputs. A directed acyclic graph (DAG) of tasks defines a higher-level *kernel* (e.g. block matrix multiplication). In this section, we show how to use each of these functionalities.

2.1 The `flash_ptr<T>` type

The `flash_ptr<T>` is a replacement for standard `T*` pointers that allows programmers to handle large blocks of SSD-resident data. An object of type `flash_ptr<T>` can be created by one of two methods.
Allocation - Using an allocator provided by the library to allocate a large block on the disk. Akin to
`int *mat=(int *)malloc(len);`
the library allows creation of a scratch space on SSD:
`flash_ptr<int> mat=flash_malloc<int>(len);`
Mapping - Using a mapper provided by the library, one can create a `flash_ptr<T>` backed by an existing file. For example, `flash_ptr<float> mat_fptr = map_file<float>(matfile, READWRITE);` allows read/write access to the `float` matrix in `matfile`.
Using `flash_ptr<T>`, programmers can read and write to the backing file through our library calls. For example, one can write N elements to the file mapped to `mat_fptr` from an in-memory `mat_ptr` as follows:
`flash::write_sync(mat_fptr, mat_ptr, N);`

The `flash_ptr<T>` type supports pointer arithmetic and can be cast and used as a normal pointer through memory mapping for functionality not supported by the library (albeit with worse performance).

```
float* mmap_mat_ptr = mat_fptr.ptr;
```

2.2 Library Kernels

BLAS-on-flash kernels are functions that operate on `flash_ptr<T>` types, designed to be drop-in replacements for in-memory calls operating on `T*` types. Kernels we have implemented include:

- `gemm`: Takes two input matrices `A, B` of type `flash_ptr<float|double>` and outputs `C :=` $\alpha \cdot$ `op(A) *` `op(B)` $+ \beta \cdot$ `C`, where α and β are scalars, and `op(X)=X` or X^T. The library allows striding and all layout choices a standard BLAS `gemm` call would offer.
- `csrmm` : Performs same computation as `gemm`, but on a sparse `A` in Compressed Sparse Row (CSR) format and allows for `op(·)` only on `B`. In addition to the version where all matrices are of type `flash_ptr<float>`, we also provide a variant where `B` and `C` are in memory pointers. The CSR format stores three arrays: the non-zeros values ordered first by row and then columns, the column index of each non-zero value, and the offsets into the two previous arrays where each row starts.
- `csrgemv` : Takes a sparse matrix `A` on disk and computes `c := op(A)* b`, where `b` and `c` are in-memory vectors and `op(X)=X` or X^T.
- `csrcsc` : Converts a sparse matrix in CSR form into its Compressed Sparse Column (CSC) form with both inputs and outputs as `flash_ptr<T>` types. This is equivalent to transposing the input matrix.

In addition to basic kernels, we also implemented some higher-level algorithms like:

- `kmeans` : Given seed centers and input data points, all as `flash_ptr<float>` types, the kernel runs a specified number of Lloyd's iterations and overwrites the seeds with final cluster centroids.
- `sort` : Parallel sample sort on a `flash_ptr<T>` array using a user-defined comparator.

Using *BLAS-on-flash* kernels, programmers can eliminate memory limitations of their in-memory variants. For example, using `csrmm` and `csrgemv`, one could implement an eigensolver for flash-resident matrices. In a later section, we describe complex algorithms using these and other custom kernels to process large amounts of flash-resident data.

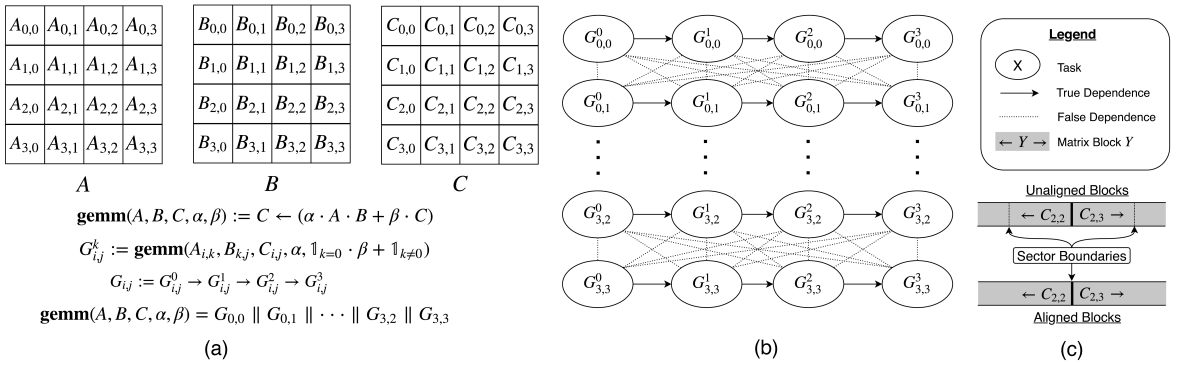

Figure 1: The gemm kernel, its DAG using the *Task* interface, and sector-sharing among adjacent output blocks in C.

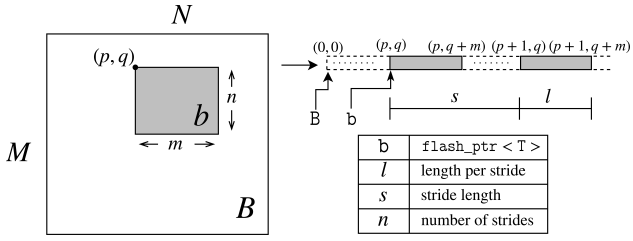

Figure 2: $\langle \mathtt{b}, \{l, s, n\} \rangle$ is an access specifier for block b of a flash-resident matrix B stored in Row-Major layout.

2.3 Tasks and Computation Graphs

A *BLAS-on-flash* kernel operating on large inputs is composed of smaller units of computation called tasks. New tasks are defined using the `Task` interface of the library. The `Task` interface allows users to define in-memory computations on smaller portions of the input. It also provides a mechanism to compose a computation graph by allowing parent-child relationships between tasks to encode dependencies.

Task inputs and outputs are uniquely described using an *access specifier*: ⟨`flash_ptr<T>, StrideInfo`⟩. Here, `flash_ptr<T>` points to the start of the data and `StrideInfo` describes an access pattern starting at `flash_ptr<T>`. An access pattern could be a:

- Strided access to retrieve a matrix block that touches a small *strip* – i.e. a subset – of each row/column of a dense matrix. This is specified using 3 parameters - number of strides, access length per stride (strip size) and the stride length before next access. For the matrix block b in Figure 2, these are n, l, and s respectively.
- Single contiguous access to a chunk of data, equivalent to a strided access with only one strip.

In addition to specifying the inputs and outputs, the user must implement the `execute` function that computes outputs using the inputs. The *BLAS-on-flash* runtime maps a `flash_ptr<T>` to an in-memory `T*` and makes this mapping available in `execute`. With inputs and outputs available as `T*` types, the programmer must detail operations on inputs using only in-memory function calls to produce outputs.

Figure 1a illustrates a task $G^k_{i,j}$, its inputs $(A_{i,k}, B_{k,j}, C_{i,j})$ and the computation in its `execute` as a block-matrix multiplication on its inputs using an in-memory `gemm` call.

A user can create a new kernel by specifying a directed acyclic graph (DAG) with a task at each node and directed edges from parent tasks to their child tasks. Once a task's parents are specified, the user injects it through the *BLAS-on-flash* Scheduler interface. By allowing tasks to be injected into the scheduler at runtime, the user can specify data-dependent computation graphs required for certain algorithms like eigensolvers.

Figures 1a and 1b illustrate the `gemm` kernel and the DAG associated with its implementation using the Block Matrix Multiplication algorithm. For inputs A, B, and C, shown with 16 blocks for each matrix, an output block $C_{i,j}$ is given by $C_{i,j} := \beta \cdot C_{i,j} + \alpha \cdot \sum_{k=0}^{k=3} A_{i,k} \cdot B_{k,j}$. The inner summation is converted into an *accumulate* chain by using a task $G^k_{i,j}$ in Figure 1a, for each k. $G_{i,j}$ depicts the dependence between successive tasks in the accumulate chain using arrows from a parent task to its child task. Figure 1a illustrates the composition of the `gemm` kernel using accumulate chains and Figure 1b gives the complete DAG for A, B, and C as the inputs and C as the output. The parallel composition operator $X||Y$ allows both X and Y to execute in parallel while the serial composition operator $X \rightarrow Y$ allows Y to execute only after X.

The task injection and logic required for creating a DAG corresponding to a kernel are then packaged into a single function call. This method of packaging allows programmers to replace in-memory calls with *BLAS-on-flash* variants with minimal modifications to existing pipelines. We demonstrate this by replacing memory-intensive kernels in the ISLE topic modeling algorithm, with a *BLAS-on-flash* variant, one kernel at a time.

3 Library Design

BLAS-on-flash supports online scheduling of tasks from a user-defined dynamic graph using a limited DRAM budget with the aim of executing it at near in-memory performance. This requires addressing two resource management problems: (1) effective utilization of the limited DRAM budget by avoiding redundant copies of data shared between tasks, and (2) realizing effective pipelining of computation and IO by better utilization of the limited disk bandwidth offered by PCIe-based SSDs. The library addresses these problems by improving *buffer reuse*, and determining a task schedule likely to minimize disk reads and writes.

We use the `gemm` kernel operating on single precision floating point matrices as an example. The following calculation illustrates the gap between the running times of an in-memory and an SSD-based version on a machine, `test`, with 32 cores capable of 1TFLOPs, and an NVMe SSD with sustained read and write bandwidths of 3GB/s and 0.5GB/s, respectively. Assume that the input and output matrices are of size 32768×32768 each, blocked as in Figure 1. Assume that the matrix block size is 8192×8192. Each task in the `gemm` kernel requires 1TFLOP of compute on 768MB of input to produce 256MB output. On the `test` system, each such task requires 0.75s of IO time for 1s of compute, when using all 32 threads for one task. Since every task has the same IO and compute requirements, a `gemm` kernel with 64 tasks would take 112s to execute out of memory without pipelining, instead of 64s if executed completely in memory. It is to be noted that, in reality, mixing reads and writes results in reduced read throughput [3]. We do not address this issue here. We instead focus on solving the two problems stated above within the constraints set by the hardware and OS. We are specifically interested in buffer management policies that optimize performance for DAGs arising from linear algebra kernels and algorithms involving matrix operations.

3.1 Buffer Reuse

A task scheduler executing the DAG in Figure 1b might execute tasks $G^1_{0,0}$ and $G^0_{1,0}$ concurrently. If the scheduler is naive, it might prefetch block $B_{0,0}$ twice, thus replicating it in memory. In addition to wasting limited DRAM, this would waste the limited disk bandwidth. Redundant reads can be eliminated, where possible, by enforcing uniqueness of data in memory. The *BLAS-on-flash* runtime ensures such uniqueness by using *reference counters* for in-memory buffers (described Section 4), allowing data reuse, where possible.

3.2 Prioritized Scheduling

Although Buffer Reuse reduces disk reads, the programmer still needs to carefully manage the order of task injection to maximize data reuse between tasks active in memory. To avoid the programmer this burden, and allow the scheduler to takeover this task, we propose a heuristic to select a task for prefetching, based on data currently buffered into memory and the IO requirements of the tasks in the ready list. Our heuristic selects the task that requires the minimum number of bytes to be prefetched given the current contents of the memory buffer. For kernels like `gemm` and `csrmm`, this heuristic minimizes the number of input matrix blocks read by scheduling tasks with high input and output locality. If all matrix blocks are uniform in size, this also reduces the number of write-back operations.

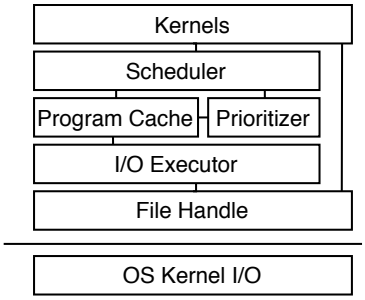

Figure 3: The *BLAS-on-flash* software stack.

Suppose that, at some point, in an execution of the `gemm` DAG in Figure 1, $M = \{A_{0,0}, A_{1,0}, A_{1,1}, B_{0,0}, B_{1,0}, B_{1,1}, C_{0,0}, C_{1,0}, C_{1,1}\}$ is the set of blocks in memory, and the following tasks are executing concurrently.

$$G^1_{0,0} := \mathbf{gemm}(A_{0,1}, B_{1,0}, C_{0,0}, \alpha, 1)$$
$$G^0_{1,0} := \mathbf{gemm}(A_{1,0}, B_{0,0}, C_{1,0}, \alpha, \beta)$$
$$G^1_{1,1} := \mathbf{gemm}(A_{1,1}, B_{1,1}, C_{1,1}, \alpha, 1)$$
$$G^1_{1,0} := \mathbf{gemm}(A_{1,1}, B_{1,0}, C_{1,0}, \alpha, 1)$$

If $G^1_{0,0}, G^0_{1,0}$, and $G^1_{1,1}$ are the latest 3 tasks to complete execution, $G^0_{1,0}$'s child task, $G^1_{1,0}$, is now ready for execution. By scheduling $G^1_{1,0}$ instead of the next-in-queue task, $G^1_{1,0}$ can immediately start execution without requiring any IO. Since outputs from the accumulate chains $G_{0,0}$, $G_{0,1}, G_{1,0}$, and $G_{1,1}$ exhibit high locality, our heuristic schedules tasks from such *nearby* accumulate chains to reduce disk operations.

4 Architecture

The *BLAS-on-flash* library implementation consists of the software stack in Figure 3. We describe the role of each of the 5 layers:

File Handle provides a read-write interface using access specifiers for all library calls. Implementations can be specialized for hardware interfaces (e.g. NVMe, SATA, or network) as required. We implement this interface for SSDs using the Linux kernel asynchronous

IO syscall interface – `io_submit` to submit IO jobs and `io_getevents` to reap job completions. Compared to user-space NVMe drivers like `SPDK` [24] and `unvme` [41], the `io_submit` syscall interface provides a simpler interface for sector-level unbuffered asynchronous IO with minimal performance penalties.

IO Executor maintains a thread-pool to service IO requests generated by *Program Cache*. To exploit parallelism and ensure correctness, *IO Executor* executes only *non-overlapping* requests in parallel. A pair of requests are *overlapping* if they modify at least one common sector on the disk. For example, consider Figure 1c. When the leading dimension of a matrix block is aligned to the device sector size, $C_{2,2}$ and $C_{2,3}$ can be operated on concurrently. Otherwise, writes to common sectors must be ordered to avoid data corruption. To detect overlaps between requests, each write request is advertised to other threads. A request is added to a thread-local backlog queue if it overlaps with an advertised request. Each thread in the thread-pool services its backlog queue with a higher priority in its next cycle.

Program Cache is the memory subsystem for *BLAS-on-flash* . It manages allocation, deallocation, prefetch, and eviction of in-memory buffers. *Program Cache* allows for buffer re-use by mapping *access specifiers* to reference-counted in-memory buffers. Each map entry is in one of four states - Active(A), Prefetch(P), Write-Back(W), or Zero-Reference(Z). An entry in state A indicates an active reference, i.e, at least one task has a reference to the buffer. An entry in P is a prefetch in progress, W is a write-back in progress, and Z is an entry with zero active references. Entries are one of 3 types - `R-only`, `W-only` and `RW`, corresponding to read-only, write-only and read-write entries. It uses this information to serve four types of requests.

- COMMIT - *commits* a task to memory by ensuring all inputs and outputs are mapped to in-memory buffers. If some inputs/outputs are not already mapped, it evicts some in-memory buffers to free up memory, allocates memory, and queues up prefetches to *IO Executor*. It also increases reference counts for mapped buffers. State is unchanged if the request fails because no entries were eligible for eviction.
- RELEASE - *returns* a task's inputs and outputs; also decreases reference counts for returned buffers.
- UPDATE - Checks and updates status of pending IO operations.
- Batch HIT/MISS - Typical HIT/MISS queries on a cache to aid prioritization during scheduling.

Program Cache entries transition states according to Figure 4. `R-Only`, `W-only` and `RW` are transitions corresponding to read-only, write-only, and read-write entries, respectively. If a COMMIT request is successful

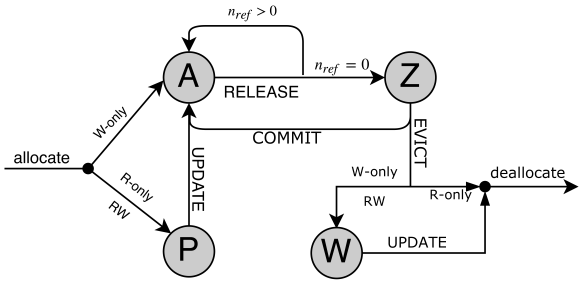

Figure 4: State transition diagram for *Program Cache* entries.

and a new entry is created, memory is allocated using `malloc`. If the entry requires data on disk to be read (`R-only`, `RW`), a prefetch is queued. Since entries in Z already contain prefetched data, COMMIT requests transition them directly to A, avoiding a redundant read. Entries enter state A with exactly one active reference. Additional COMMIT requests for entries in A only increase reference counts, and RELEASE requests decrease the same. Entries with zero active references in A transition to Z, making them available for eviction. Evicting a *dirty* entry (`RW`, `W-only`) queues a write-back and transitions the entry from Z to W. Entries in P transition into A, and those in W get de-allocated once their IO operations are complete

Prioritizer uses Batch HIT/MISS queries on *Program Cache* to rank the list of ready tasks in increasing order of their prefetch sizes given the current cache state.

Scheduler provides an interface to inject tasks at runtime. Once injected, tasks are executed using a 5-stage pipeline — Wait, Ready, Prefetch, Compute, and Complete. All tasks start out in **Wait** stage, and advance to **Ready** stage when all its parents have finished Compute stage. In each scheduling round, *Scheduler* tries a COMMIT request to *Program Cache* with the highest priority task obtained from *Prioritizer*. If successful, this task advances to **Prefetch** stage. When all its inputs and outputs are mapped to in-memory buffers, the task moves to **Compute** stage. Tasks in Compute stage are executed using a thread-pool maintained by *Scheduler*. Tasks finishing Compute stage are recorded as **Complete**, and *Scheduler* issues a RELEASE request to *Program Cache* with these completed tasks. Once all tasks in a kernel are complete, *Scheduler* allows the programmer to flush any outputs in *Program Cache* to persist results to disk.

5 Algorithms and Evaluation

We now discuss the implementation of the kernels provided by the library and complex algorithms built using these kernels, and compare the running times and memory requirements of in-memory and SSD-based versions. We implemented an eigensolver, an SVD-based

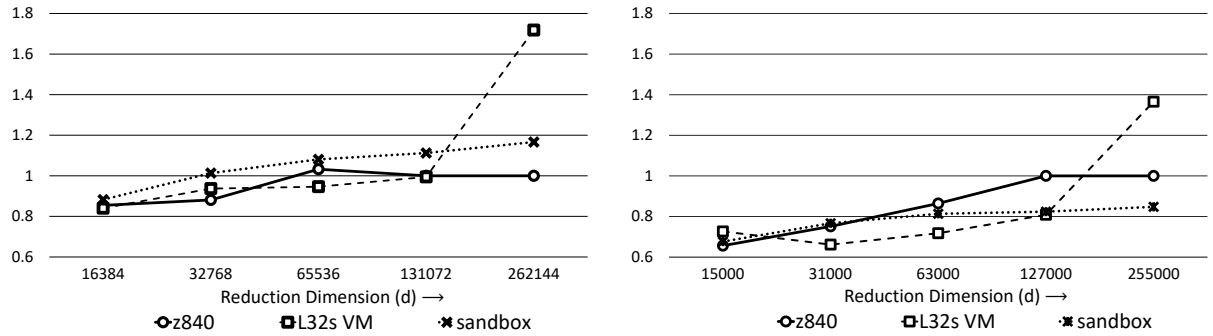

Figure 5: Ratio of in-memory MKL `gemm` to *BLAS-on-flash* `gemm` running times for 512-aligned (left) and unaligned (right) instances for various values of reduction dimension (d). The matrix dimensions are $2^{15} \times d \times 2^{15}$ and $31000 \times d \times 31000$ for the aligned and unaligned plots. *BLAS-on-flash* library has a 8GB Program Cache. `gemm` tasks in *BLAS-on-flash* library use 4 threads each. Program Cache budget determines the number of simultaneous tasks.

algorithm for topic modeling, and two inference algorithms for XML models. This choice of algorithms represents the state-of-the-art for a subset of non-deep learning problems used in ranking and relevance pipelines. Where available, we compare our implementations of these algorithms with prior implementations.

5.1 Experimental setup

The library allows the user to control the number of threads per task (T) and the maximum number of tasks that can be simultaneously executed (K). On a machine with N cores, one would typically choose $T \times K = N$. Within this constraint, the optimal values of T and K are determined by the compute-communication ratio of the task and the parallelism within the task. For the pipeline to execute K tasks in parallel in steady state, the *Scheduler* needs to hold $3K$ tasks in memory to account for K tasks each in Prefetch, Compute and Complete stages of the pipeline. Therefore, in the case of `gemm` and `csrmm` kernels, setting T=1 and K=N increases pressure on disk and *Program Cache*. On the other hand, when K=1 with T=N, MKL does not realize T-fold parallelism with small block sizes. We find T=4, K=$N/4$ to be a good tradeoff, empirically.

Table 1 lists the configurations of machines used to evaluate our library. `sandbox` is a high-end bare-metal server with enterprise class Samsung PM1725a SSD capable of sustained read speeds of up to 4GB/s and write speeds of up to 1GB/s. `z840` is chosen to represent a typical bare-metal workstation machine configured with two Samsung 960EVO SSDs in RAID0 configuration, providing sustained read speed of about 3GB/s and write speed of about 2.2GB/s. `L32s VM` is a virtual machine on Azure configured for heavy IO with I/O throttled to a sustained 1.6GB/s or 160K IO ops/second. `M64-32ms VM` is a virtual machine on Azure with 1.7TB RAM that we'll use for running experiments with large memory requirements. We use Intel MKL 2018 and Ubuntu 16.04LTS on all the machines listed above. Apache Spark instances run Apache Spark MLlib 2.1 on a cluster of Azure `DS14v2 VM` instances.

Name	Processor	Cores	RAM	SSD
`sandbox`	Gold 6140	36	512GB	3.2TB
`z840`	E5-2620v4	16	32GB	2TB
`L32s VM`	E5-2698Bv3	32	256GB	6TB
`M64-32ms VM`	E7-8890v3	32	1.7TB	–
`DS14v2 VM`	E5-2673v3	16	112GB	–

Table 1: Intel Xeon-based machines used in experiments.

5.2 Matrix kernels

General Matrix Multiply (`gemm`) and Sparse (CSR) Matrix Multiply (`csrmm`) are perhaps the most used kernels in math libraries. Therefore, it is important to optimize their performance with careful selection of tiling patterns and prefetch and execution orders in order to minimize IO. For this, we build on well-established results on exploiting locality in matrix multiplications [5, 28, 30]. We also use the fact that BLAS and sparseBLAS computations can be tiled so that they write the output to disk just once [12, 13], thus saving on write bandwidth.

`gemm`. The block matrix multiplication algorithm in Figure 1 requires $O(n^3)$ floating point operations for $n \times n$ matrices. With block size b, it reads $O(n^3/b)$ bytes from disk and writes $O(n^2)$ bytes back. It is ideal for the library to increase the block size b as much as its in-memory buffer allows so as to decrease the amount of IO required. Figure 5 presents the ratio of running times of the in-memory MKL `gemm` call to that of our library for various *reduction dimension* sizes in two cases:

- **512-aligned**. A matrix is *512-aligned* if the size of its leading dimension is a multiple of 512. e.g., a 1000x1024 `float` matrix in row-major layout, that would require 4096 bytes for each row.

- **unaligned**. A matrix is *unaligned* if it is **not** 512-aligned, e.g., a 500x500 `float` matrix in row-major form, that would require 2000 bytes per row.

The distinction between 512-aligned and unaligned matrices is important as the two cases generate a different number of disk access when a block of the matrix is to be fetched or written to. Flushing an unaligned matrix block to disk requires two reads and one write per row – read the start and end sectors of each row in the block, and write-back the overwritten values. A 512-aligned block requires only one write per row.

We define the *reduction dimension* (RD) to be the dimension along which summation happens during matrix multiplication. Using notation from Figure 1, if $A, B,$ and C are all stored in row-major form, the RD is the number of columns in A. Given a block size, increasing the RD increases the length of the accumulate chain, resulting in fewer disk writes per chain. Pipelining efficiency increases with longer accumulate chains, due to lower write-back operations per chain, as demonstrated by Figure 5. In fact, due to careful pipelining, our library outperforms in-memory MKL calls in many instances.

We also evaluated the performance of `gemm` when DRAM overflow is serviced by OS paging mechanisms. We timed a problem of dimension $49K \times 49K \times 49K$ (30GB size) on the `z840` machine with 32GB and 16GB of RAM. For runs with 16GB RAM, we pin a 128GB swap partition to the SSD. The OS-paged version with 16GB RAM ran $1.6\times$ slower than the in-memory version with 32GB RAM. On a larger problem size ($64K \times 64K \times 64K$, 48GB size) and 16GB RAM, OS paging results in a more substantial slowdown – about 13x slower than what in-memory version would have taken.

`csrmm`. The `csrmm` kernel performs $O(n^3 s)$ floating point operations on $n \times n$ size input matrices with sparsity s, representing inputs of size $O(n^2(1+s))$ and output of n^2 size. For a matrix whose sparsity is uniform across rows and columns, with a block size of b, the compute to IO ratio is only $O(bs)$ as opposed to $O(b)$ for `gemm`. For sparse matrices such as those in Table 2 arising from text data (e.g. bag-of-words representation), sparsity can be as low as $s = 10^{-4}$. Therefore, although the execution of in-memory `csrmm` tasks is slower (sparse operations are $10 - 100\times$ slower than dense operations), the low locality (bs as opposed to b) makes it hard to always obtain near in-memory performance. Figure 6 demonstrates the effect of sparsity on `csrmm` by fixing the problem dimensions at $2^{20} \times 2^{17} \times 2^{12}$ and measuring the ratio of in-memory to *BLAS-on-flash* running times for $s \in \{10^{-4}, 10^{-3}, 10^{-2}\}$. It is evident that the efficiency of the `csrmm` kernel decreases with sparsity.

We also benchmark the `csrmm` call required to project the sparse bag-of-words datasets listed in Table 2 into a 1024-dimensional space (say, obtained from Principal Component Analysis). The dense input and output matrices are 512-aligned and in row-major format. A performance drop is expected in the unaligned case.

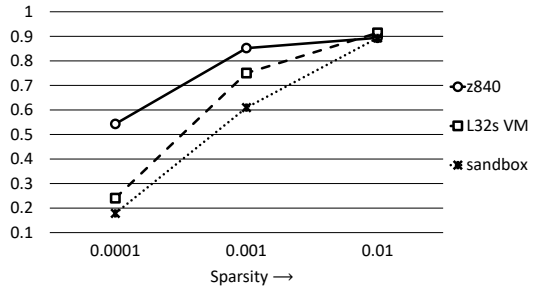

Figure 6: Ratio of in-memory MKL `csrmm` to *BLAS-on-flash* `csrmm` running times for $(2^{20} \times 2^{17} \times 2^{12})$ sized instances and various values of sparsity. *BLAS-on-flash* uses a 8GB Program Cache. Each `csrmm` task uses 4 threads and the number of simultaneous tasks is determined by number of cores in the system.

Dataset	#Cols	#Rows	NNZs	Tokens	Size
Small	8.15M	140K	428M	650M	10.3GB
Medium	22M	1.56M	6.3B	15.6B	151GB
Large	81.7M	2.27M	22.2B	65B	533GB

Table 2: Sparse matrices bag-of-words text data sets. Columns and rows of the matrix represent the documents and words in the vocabulary of a text corpora. The (i, j)-th entry of the matrix represents the number of times the j-th word in the vocabulary occurs in the i-th document.

Table 3 compares the performance of the `csrmm` in *BLAS-on-flash* to the in-memory version provided by MKL on `z840` , `L32s VM` and `sandbox` machines. `z840` is too small to run the in-memory version for all three data sets because it has only 32GB RAM. Since projecting the Large dataset into 1024 dimensions requires 559GB of RAM, both `L32s VM` and `sandbox` are unable to do it in memory. As an approximation to the speed of an in-memory call on `L32s VM` we ran it on `M64-32ms VM` which has 1.7TB RAM.

Despite a sparsity of 2×10^{-4}, the `csrmm` in *BLAS-on-flash* is about 50% as fast as its in-memory counterpart on the Medium dataset (when the dense matrices are in row-major layout). We picked row-major order for dense matrices because our library was able to outperform MKL's `csrmm` implementation for column-major order by a factor of $> 2\times$ on Small and Medium datasets. We attribute this to poor multi-threading in MKL's implementation.

5.3 Eigensolver

Eigen-decomposition is widely used in data analytics, e.g., dimensionality reduction. Given a symmetric matrix $\mathbf{A}$, a symmetric *eigensolver* attempts to find k

Dataset (#eigenvalues)	Block Krylov-Schur				Spectra	computeSVD (shared) Number of Spark Executors				computeSVD (dedicated) Number of Spark Executors			
	L32s VM		sandbox										
	in-mem	flash	in-mem	flash		64	128	256	512	64	128	256	512
Medium(500)	76	182	63	95	934	320	275	365	450	460	225	228	226
Large (200)	154*	429	–	153	–	–	–	169	230	236	126	104	164

Table 4: Time, in minutes, to compute eigenvalues. For both Medium and Large datasets, Block KS is run with block=25. For Medium, nev=500 and ncv=2500 and for Large, nev=200 and ncv=1500. We run Block KS in-memory on M64-32ms VM as an approximation to L32s VM . Spark MLlib's computeSVD was timed with 64, 128, 256, 512 workers with 8GB memory on both a shared and a dedicated cluster. The Large dataset needs at least 256 workers to run on the shared cluster. On stand alone cluster with 64 works, the Large dataset needed 10GB memory per worker.

Dataset	z840	L32s VM		sandbox	
	flash	in-mem	flash	in-mem	flash
Small	34.7	8.2	24.5	6.9	35.2
Medium	135.75	58.5	101.3	49.5	98.0
Large	636.2	512.3*	390.6	–	354.9

Table 3: Running times in seconds for csrmm operations that project datasets in Table 2 into 1024-dimensions. *BLAS-on-flash* has a 16GB Program Cache. *This is run on M64-32ms VM as an approximation to L32s VM .

eigenvalue-eigenvector pairs $(\lambda_i, \mathbf{v}_i)$ such that

$$\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i \quad \forall i$$

$$\mathbf{v}_i^T \mathbf{v}_j = 0, \|\mathbf{v}_i\|_2 = 1 \quad \forall i \neq j, \quad |\lambda_1| \geq |\lambda_2| \ldots \geq |\lambda_k|$$

Popular dimensionality reduction techniques like Principal Component Analysis (PCA) and Singular Value Decomposition (SVD) use the symmetric eigenvalue decomposition (syevd) to compute the projection matrices required for dimensionality reduction. The SVD of a matrix $\mathbf{M}$ can be formulated as a symmetric eigendecomposition problem as follows:

$$\mathbf{M}\mathbf{u}_i = \sigma_i \mathbf{v}_i, \ \|\mathbf{v}_i\|_2 = \|\mathbf{u}_i\|_2 = 1 \forall i$$

$$\mathbf{u}_i^T \mathbf{u}_j = 0, \ \mathbf{v}_i^T \mathbf{v}_j = 0 \, \forall i \neq j, |\sigma_1| \geq |\sigma_2| \geq \ldots \geq |\sigma_k|$$

$$\mathbf{M}\mathbf{M}^T \mathbf{u}_i = \sigma_i^2 \mathbf{u}_i, \ \mathbf{M}^T \mathbf{M} \mathbf{v}_i = \sigma_i^2 \mathbf{v}_i$$

$$\texttt{svd}(\mathbf{M}) = \texttt{syevd}(\mathbf{M}\mathbf{M}^T) = \texttt{syevd}(\mathbf{M}^T\mathbf{M})$$

To showcase the versatility of our library, we implement a symmetric eigensolver and time it on large sparse matrices (in CSR format) obtained from text corpora in Table 2. Among the many flavors of eigensolvers, we picked the Krylov-subspace class of algorithms as they have been shown to be stable for a wide variety of matrices. These algorithms use iterated Sparse Matrix-Vector (csrgemv) products to converge on eigen-pairs.

Since csrgemv is bandwidth-bound, it is not suitable for an eigensolver operating on SSD-resident matrices. To overcome this limitation, we implement the Restarted Block Krylov-Schur (Block KS) algorithm [67]. The Block KS algorithm can potentially use fewer matrix accesses to achieve the same tolerance by using a csrmm kernel in place of csrgemv. Although the Block KS algorithm performs extra computation compared to its non-block variants, this extra work is highly parallel and the IO savings offset the extra compute.

Analysis of eigenvalues of our sparse matrices reveals a large gap between successive eigenvalues. Since time to convergence is inversely correlated with this gap, the Block KS algorithm converges quickly, to the desired tolerance, on our test datasets.

Evaluation. We benchmark both our in-memory and SSD-based single node implementations of the Block KS algorithm against single node and distributed implementations of the Implicitly Restarted Arnoldi Method (IRAM) algorithm. The single node version is provided by *Spectra* [48], a C++ header-only implementation of *ARPACK* [35], while the distributed version (computeSVD) is provided by Apache Spark MLlib library v2.1. The Spark job was deployed on both a shared and a dedicated Hadoop cluster through YARN [55] to workers with 1 core and 8GB memory each and a driver node with 96GB memory. The shared cluster runs Xeon E5-2450L processors with 10Gb Ethernet, while the dedicated cluster uses DS14v2 VM nodes. Other distributed SVD solvers, such as those provided by ScaLAPACK and Spark KeystoneML, do not adequately support sparse matrices, and are omitted from this comparison.

Table 4 compares the time taken to solve for the top singular values of sparse matrices in Table 2 to a tolerance of 10^{-4} (this is sufficient for the SVD-based topic modeling algorithm described in Section 5.4). It must be noted that computeSVD uses double precision floating point numbers while our algorithm uses single precision. We solve for 200 singular values on the large data set and 500 on the Medium data set because the Spark solver was unable to solve for more. Our implementation, on the other hand, easily scales to thousands of singular values on a single node.

The flash version of Block KS runs almost as fast as the in-memory version on datasets with sparsity up to 10^{-3}; the gap widens as sparsity decreases below 10^{-4}. Further, both Block KS implementations outper-

form Spectra and Spark jobs in time to convergence. Spark does not see any benefit from adding more workers beyond a point; in fact it becomes slower. These results demonstrate that our flash-based eigensolver utilizes hardware order(s) of magnitudes more efficiently than distributed methods.

5.4 SVD-based Topic Modeling

Topic modeling involves the recovery of underlying *topics* from a text corpus where each document is represented by the frequency of words that occur in it. Mathematically, the problem posits the existence of a topic matrix M whose columns $M_{.l}$ are probability distributions over the vocabulary of the corpus. The observed data is assumed to be generated by (1) picking a matrix W, whose columns sum to one and represent linear combinations of topic columns in M, (2) calculating $P = MW$, where the j-th column $P_{.j}$ represents the probability of words in the document j, and (3) sampling the observed documents $A_{.j}$ using a multinomial distribution based on the p.d.f. $P_{.j}$. The computational problem is to recover the underlying topic matrix M, given the observations A.

ISLE, or Importance Sampling for Learning Edge topics, is a direct adaption of the TSVD algorithm [6] for recovering topic models [42]. Unlike the LDA class of algorithms (based on MCMC techniques), ISLE uses linear-algebraic techniques to provably recover the underlying topic matrix under reasonable assumptions on the observed data. Empirically, it has been shown to yield qualitatively better topics on real world data. The open source implementation [42] is faster than other single node implementations of any topic modeling algorithms. It takes as input bag-of-words representation for documents in CSR or CSC format, and does the following steps: (1) threshold to *denoise* the data, (2) use SVD to compute a lower dimensional space to project the documents into, (3) cluster documents using k-means++ initialization and the k-means algorithm in the projected space, (4) use the resultant clusters to seed clustering in the original space using the k-means algorithm, and finally (5) construct the topic model. For large datasets, sampling techniques can be used to pick a subset of data for the expensive steps (2), (3), and (4). We adapt ISLE to use the *BLAS-on-flash* framework by leveraging our flash-based Block KS eigensolver and the clustering algorithms built using our framework.

Evaluation. Table 5 compares the running times of the in-memory version and a flash-based version using the *BLAS-on-flash* library. Using this redesigned pipeline, we were able to train a 5000-topic model with a DRAM requirement of 1.5TB on both `L32s VM` and `sandbox` machines with only 32GB allocated to *Program Cache*. We note that the number of tokens in this dataset (about 65 billion) is in the same ballpark as the number of tokens processed by LDA-based topic modeling algorithms in Parameter Server based systems that use multiple nodes [37].

Dataset (# Topics)	Sample Rate	`sandbox` in-mem	`sandbox` flash	`L32s VM` in-mem	`L32s VM` flash
Small(1K)	1.0	15	27	18	37
Medium(1K)	0.1	46	66	63	72
Medium(2K)	0.1	119	144	158	212
Large(1K)	0.1	–	149	163*	172
Large(2K)	0.1	–	228	285*	279
Large(5K)	0.1	–	522	980*	664
Large(2K)	0.4	–	532	684*	869

Table 5: Running time of the ISLE algorithm in minutes. *We use `M64-32ms VM` as an approximation to `L32s VM` for the Large dataset.

On the Medium dataset, where it is possible to run an in-memory version, notice that the code linked to *BLAS-on-flash* achieved about $65-80\%$ in-memory performance using less than 128GB RAM. On the Large dataset, the flash version run on `sandbox` is faster than the in-memory version on `M64-32ms VM`. We attribute this to newer hardware on `sandbox` , and near in-memory performance of eigensolver and kmeans kernels written with *BLAS-on-flash* .

5.5 Extreme Multi-Label Learning

Extreme multi-label learning (XML) addresses the problem of automatically annotating a data point with the most relevant subset of labels from an extremely large label set. It has many applications in tagging, ranking and recommendation. Many popular XML algorithms use tree based methods due to their low training and prediction complexity. In this subsection, we present experiments with two such algorithms that use ensembles of trees: PfastreXML [29] and Parabel [46].

In a current deployment, both algorithms train an ensemble of trees (50 trees for PfastreXML, 3 for Parabel) using 40 million data points, each of which is a sparse vector in 4.5M dimensions. Once trained, each tree in the ensemble predicts label probabilities for 250M test data points. Both training and inference are difficult to scale – training requires weeks on a machine with few terabytes of RAM, and inference currently requires dozens of machines. As XML algorithms are applied to larger problems (e.g. web search), they need to scale to datasets with billions of points and hundreds of millions of labels, and train trees that are hundreds of gigabytes in size.

Because of the memory limitations of the platforms on which these algorithms are deployed, orchestrating data and models out of SSDs becomes critical. We demonstrate the capabilities of our library in such cases. We focus on inference since it is run more frequently than

Algorithm 1 PfastreXML Inference

function CLASSIFY(N, v)
  if N is *leaf* **then**
    return $N.prob$
  else
    if $\langle N.w, v\rangle + N.b > 0$ **then**
      return CLASSIFY($N.right$, v)
    else
      return CLASSIFY($N.left$, v)

Algorithm 2 Parabel Inference

function SCORE(T, v, α, k)
  $L \leftarrow [(T, 0.0)]$
  for each level in T from root to leaves **do**
    $L' \leftarrow [\,]$
    for (N, s) in L **do**
      $s_l \leftarrow \langle N.w_l, v\rangle + N.b_l$
      $s_r \leftarrow \langle N.w_r, v\rangle + N.b_r$
      $s_l \leftarrow \alpha \cdot s - max(0, 1 - s_l)^2$
      $s_r \leftarrow \alpha \cdot s - max(0, 1 - s_r)^2$
      Append $[(N.left, s_l), (N.right, s_r)]$ to L'
    $L \leftarrow top_k(L', k)$
  return L

training. Similar techniques can be applied for training.

PfastreXML: During training, trees are grown by recursively partitioning nodes starting at the root until each tree is *fully grown*. A node N is split by learning a hyperplane $N.w$ and bias $N.b$ to partition training points between its left and right children, $N.left$ and $N.right$. Node partitioning terminates when a node contains fewer points than a threshold. Leaf nodes contain a probability distribution over the label set ($N.prob$). During inference, a tree with root R assigns the probability vector over labels for a point v dictated by CLASSIFY(R, v).

Parabel: During training, a tree T is grown by recursively partitioning its nodes to distribute the labels. Labels assigned to a node N are partitioned in equal numbers to its two children, $N.left$ and $N.right$. A node N containing fewer labels than a threshold is split into multiple leaf nodes with one label per leaf node. Each tree node N contains two probabilistic linear classifiers, with weights and biases $(N.w_l, N.b_l)$ and $(N.w_r, N.b_r)$, that decide whether the data point has relevant labels in its left and right subtrees. These classifiers are trained to maximize the a-posteriori probability distribution over the training data. The Parabel inference algorithm is described in Algorithm 2. α is a discount factor and k is the beam width for beam search on tree T. $top_k(L, k)$ returns the top k entries in list L, ordered by their scores in descending order. Given a point, v, and the root node, R, likely labels and their associated scores for v are contained in the return value of SCORE(R, v, α, k).

The inference code downloaded for both algorithms from the XML repository [8] is single-threaded and takes about 440 hours and 900 hours for PfastreXML and Parabel inference, respectively, on Azure D14v2 nodes with 112GB RAM and 16 cores. The orchestration required to complete the inference in under two days is complex and increases the likelihood of failures.

PfastreXML inference involves a depth-first traversal of a non-balanced binary tree while Parabel inference requires breadth-first beam search on a balanced binary tree. In both cases, we noticed that the baseline code was inefficient and modified the code to take a batch of test data points (about 2-4 million per batch) and perform a level-by-level, or breadth-first, traversal of the tree. With this transformation, the new inference code was about $6\times$ faster on nodes with a large amount of RAM. We think this is close to the limit of how fast this inference can run with DDR3 memory.

We use the *BLAS-on-flash* library to orchestrate the level-by-level traversal of each tree for a batch of points. For both algorithms, we construct one task for each `(level, batch)` pair. For PfastreXML inference, the DAG is data-dependent, while for Parabel, it is dependent only on the tree height. Since inference is data parallel, *BLAS-on-flash* can run tasks corresponding to multiple batches concurrently. It also orders the prefetches of tree levels and data to maximize re-use.

Evaluation. We compare the in-memory and *BLAS-on-flash* variants of the inference code on models in two regimes – Medium and Large. The Medium-sized models consist of 20GB trees containing about 25 million nodes each, while the Large Parabel model consists of 122GB trees. The Medium-sized models fit in the memory of the largest machines used in the inference platform, while the Large-sized model does not fit in the memory of any machine in the platform. We use a total of 50 trees for PfastreXML and 3 for Parabel inference. Our test data consists of 250 million points, each a sparse vector in 4.3M dimensions and taking up 500GB of storage when stored in a compressed sparse format.

We benchmark both inference algorithms on `z840` , `L32s VM` , and `sandbox` and use 2^{21} points/batch for `z840` and 2^{22} points/batch for `L32s VM` and `sandbox` . The size of *Program Cache* for *BLAS-on-flash* is set at 20GB for `z840` and 40GB for `L32s VM` and `sandbox` . We use 32 compute threads on `z840` and `L32s VM` and 64 threads on `sandbox` .

Table 6 presents the running times and memory requirements of our code on the Medium and Large-sized models. Inference code written with *BLAS-on-flash* runs at over 90% of in-memory speed using only a third of the required memory. The memory requirement can be further reduced by decreasing the test batch size or by splitting each `(level, batch)` task into multiple tasks in

	PfastreXML (50 trees)		Parabel (3 trees)	
	in-mem	flash	in-mem	flash
sandbox	45 (155)	51.0 (42)	27.3 (125)	25.3 (47.6)
L32s VM	69.2 (149)	67.0 (42)	44.3 (123)	45.8 (48)
z840	–	118 (26.2)	–	71.5 (30.5)

	Time (hours)		RAM (GB)	
	in-mem	flash	in-mem	flash
sandbox	51.7	57.0	241.3	80.1
L32s VM	108.4	118.2	235.5	80.9

Table 6: Running time in hours and peak DRAM usage in GB (inside parenthesis) for XML inference on 250×10^6 data points using an ensemble of medium-sized trees (left) and large Parabel trees (right). We used 64 threads on sandbox and 32 threads on L32s VM and z840 . Inference with large Parabel tree uses 70GB Program Cache.

an accumulate chain. This reduction in working set, with practically no impact on performance, critically enables us to execute inference on larger models, that can provide greater accuracy for ranking and relevance tasks.

6 Conclusion

We have demonstrated that (a) dense and sparse linear algebra kernels can be designed to run at near in-memory speeds on large SSD-resident datasets, (b) memory-intensive algorithms built using the library can match in-memory implementations, and (c) for complex numerical algorithms like eigensolvers, careful co-design of algorithm and software stack can offer large gains in hardware utilization and keep the costs of data analytics pipelines low.

Our results suggest that operating on data stored in fast non-volatile memory on a single node could provide an efficient alternative to distributed big-data systems for training and inference of industrial scale machine learning models for algorithms with large memory requirements. We do not make such claims about computationally intensive workloads such as training CNNs using GPUs. Further, our library provides a higher value proposition for the large quantity of NVM storage already deployed as storage in data centers. Our library can also be adapted to support GPU and other PCIe storage devices like Optane with minor changes.

7 Other Related Work

Recent work [7, 12, 13] has studied parallel and sequential external memory algorithms in the setting where writes to non-volatile memories are much more expensive than reads. They conclude that for kernels like sorting and FFTs, decreasing writes to non-volatile external memory is possible at the price of more reads. However, this is not the case in the case of linear algebra. Simple reordering of the matrix tiles on which the in-memory computation is performed can achieve asymptotic reduction in the amount of writes for gemm and csrmm calls without increase in reads. We use this observation extensively in our work.

FlashEigen [65] implements the Block KS eigensolver for large-scale graph analysis using a custom filesystem on an array of SSDs. While FlashEigen supports only a limited set of matrix operations, our library allows execution of user-defined computation graphs on user-defined data structures. Our library uses separate IO threads to effectively pipeline IO with computation resulting in a narrow-gap with in-memory performance, while FlashEigen worker threads perform IO and then computation on matrix blocks assigned to them.

Partitioned Global Address Space systems such as FaRM [19] and UPC [15, 21, 31, 66] that present an unified view of the entire memory available in a distributed system present an alternative for programs considered here to scale to larger data and model sizes. However, the network bandwidth available presents a barrier to the scalability of sparse kernels just as in the case of Spark. Further, with careful co-design, we feel that a large range of workloads (of up to a few terabytes in size) can be processed on a single node without the cost overhead of a cluster of RDMA-enabled nodes. Scaling our library to such systems remains future work.

The problem of smart buffer-cache management for SSDs and other non-volatile memories has been studied in the database community. For example, Ma et al. [38] evaluate design choices such as paging policies that arise when one tries to extend in-memory database to hard-drives, SSDs, 3D XPoint, etc. LeanStore [36] proposes a new storage management system to extend in-memory databases to SSDs with little overhead. In contrast, our library relies on a task scheduler designed to better utilize the buffer-cache for access patterns that typically arise in linear algebra.

While our system uses existing processing and memory hardware, new hardware and accelerators that move computation to the memory have been proposed. For example, [1] proposes how expensive access patterns such as shuffle, transpose, pack/unpack might be performed in accelerator co-located with DRAM, and analyzes potential energy gains for math kernels from such accelerators. Further, systems that proposes moving entire workloads to memory systems have been proposed [23, 51, 57].

8 Acknowledgments

The authors would like to thank Anirudh Badam, Ravi Kannan, Muthian Sivathanu, and Manik Varma for their useful comments and advice.

References

[1] AKIN, B., FRANCHETTI, F., AND HOE, J. C. Data reorganization in memory using 3D-stacked DRAM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), ISCA '15, ACM, pp. 131–143.

[2] AMD. Radeon™ Pro SSG. https://pro.radeon.com/en/product/pro-series/radeon-pro-ssg/, 2018.

[3] ANANDTECH. Mixed Random Read/Write Performance - Samsung 960 EVO (1TB) Review. https://www.anandtech.com/show/10833/the-samsung-960-evo-1tb-review/8, 2016.

[4] ARULRAJ, J., AND PAVLO, A. How to Build a Non-Volatile Memory Database Management System. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), SIGMOD '17, ACM, pp. 1753–1758.

[5] BALLARD, G., DEMMEL, J., HOLTZ, O., AND SCHWARTZ, O. Minimizing Communication in Linear Algebra. *SIAM Journal on Matrix Analysis and Applications 32*, 3 (2011), 866–901.

[6] BANSAL, T., BHATTACHARYYA, C., AND KANNAN, R. A provable SVD-based algorithm for learning topics in dominant admixture corpus. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (Cambridge, MA, USA, 2014), NIPS'14, MIT Press, pp. 1997–2005.

[7] BEN-DAVID, N., BLELLOCH, G. E., FINEMAN, J. T., GIBBONS, P. B., GU, Y., MCGUFFEY, C., AND SHUN, J. Parallel Algorithms for Asymmetric Read-Write Costs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* (2016), SPAA '16, ACM, pp. 145–156.

[8] BHATIA, K., DAHIYA, K., JAIN, H., PRABHU, Y., AND VARMA, M. The extreme classification repository: Multi-label datasets and code. http://manikvarma.org/downloads/XC/XMLRepository.html.

[9] BILENKO, M., FINLEY, T., KATZENBERGER, S., KOCHMAN, S., MAHAJAN, D., NARAYANAMURTHY, S., WANG, J., WANG, S., AND WEIMER, M. Salmon: Towards Production-Grade, Platform-Independent Distributed ML. In *The ML Systems Workshop at ICML* (2016).

[10] BLACKFORD, L. S., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., HEROUX, M., KAUFMAN, L., LUMSDAINE, A., PETIET, A., POZO, R., REMINGTON, K., AND WHALEY, R. C. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw. 28*, 2 (June 2002), 135–151.

[11] BLEI, D. M., NG, A. Y., AND JORDAN, M. I. Latent Dirichlet Allocation. *J. Mach. Learn. Res. 3* (Mar. 2003), 993–1022.

[12] BLELLOCH, G. E., FINEMAN, J. T., GIBBONS, P. B., GU, Y., AND SHUN, J. Sorting with Asymmetric Read and Write Costs. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (2015), SPAA '15, ACM, pp. 1–12.

[13] CARSON, E., DEMMEL, J., GRIGORI, L., KNIGHT, N., KOANANTAKOOL, P., SCHWARTZ, O., AND SIMHADRI, H. V. Write-Avoiding Algorithms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2016), pp. 648–658.

[14] CHEN, J., LI, K., ZHU, J., AND CHEN, W. WarpLDA: a Cache Efficient O(1) Algorithm for Latent Dirichlet Allocation. *Proc. VLDB Endow. 9*, 10 (June 2016), 744–755.

[15] CHEN, W.-Y., BONACHEA, D., DUELL, J., HUSBANDS, P., IANCU, C., AND YELICK, K. A Performance Analysis of the Berkeley UPC Compiler. In *Proceedings of the 17th Annual International Conference on Supercomputing* (2003), ICS '03, ACM, pp. 63–73.

[16] DHULIPALA, L., BLELLOCH, G., AND SHUN, J. Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (2017), SPAA '17, ACM, pp. 293–304.

[17] DINH, D., SIMHADRI, H. V., AND TANG, Y. Extending the Nested Parallel Model to the Nested Dataflow Model with Provably Efficient Schedulers. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* (2016), SPAA '16, ACM, pp. 49–60.

[18] DMTK. Multiverso: Parameter Server for Distributed Machine Learning. https://github.com/Microsoft/Multiverso, 2015.

[19] DRAGOJEVIC, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)* (April 2014).

[20] DUFF, I. S., HEROUX, M. A., AND POZO, R. An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum. *ACM Trans. Math. Softw. 28*, 2 (June 2002), 239–267.

[21] EL-GHAZAWI, T., AND SMITH, L. UPC: Unified Parallel C. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (2006), SC '06, ACM.

[22] GITTENS, A., DEVARAKONDA, A., RACAH, E., RINGENBURG, M., GERHARDT, L., KOTTALAM, J., LIU, J., MASCHHOFF, K., CANON, S., CHHUGANI, J., SHARMA, P., YANG, J., DEMMEL, J., HARRELL, J., KRISHNAMURTHY, V., MAHONEY, M. W., AND PRABHAT. Matrix Factorizations at Scale: a Comparison of Scientific Data Analytics in Spark and C+MPI Using Three Case Studies. *ArXiv e-prints* (July 2016).

[23] GUO, Q., GUO, X., BAI, Y., AND İPEK, E. A Resistive TCAM Accelerator for Data-Intensive Computing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), MICRO-44, ACM, pp. 339–350.

[24] INTEL. Storage Performance Development Kit (SPDK), 2016.

[25] INTEL®. Optane™ memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html, 2017.

[26] INTEL®. Math Kernel Library. https://software.intel.com/en-us/mkl, 2018.

[27] INTEL®. Math Kernel Library Sparse BLAS level 2 and 3 routines. https://software.intel.com/en-us/mkl-developer-reference-c-sparse-\blas-level-2-and-level-3-routines, 2018.

[28] IRONY, D., TOLEDO, S., AND TISKIN, A. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput. 64*, 9 (Sept. 2004), 1017–1026.

[29] JAIN, H., PRABHU, Y., AND VARMA, M. Extreme Multi-label Loss Functions for Recommendation, Tagging, Ranking and Other Missing Label Applications. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (August 2016).

[30] JIA-WEI, H., AND KUNG, H. T. I/O Complexity: The Red-Blue Pebble Game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (1981), STOC '81, ACM, pp. 326–333.

[31] KAMIL, A., ZHENG, Y., AND YELICK, K. A local-view array library for partitioned global address space C++ programs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (2014), ARRAY'14, ACM, pp. 26:26–26:31.

[32] KANNAN, R., VEMPALA, S., AND VETTA, A. On Clusterings: Good, Bad and Spectral. *J. ACM 51*, 3 (May 2004), 497–515.

[33] KUMAR, A., SINDHWANI, V., AND KAMBADUR, P. Fast Conical Hull Algorithms for Near-separable Non-negative Matrix Factorization. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28* (2013), ICML'13, JMLR.org, pp. I–231–I–239.

[34] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 31–46.

[35] LEHOUCQ, R., MASCHHOFF, K., SORENSEN, D., AND YANG, C. ARPACK Software. `http://www.caam.rice.edu/software/ARPACK/`, 2009.

[36] LEIS, V., HAUBENSCHILD, M., KEMPER, A., AND NEUMANN, T. LeanStore: In-Memory Data Management Beyond Main Memory. In *Proceedings of the 34th IEEE International Conference on Data Engineering* (2018).

[37] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 583–598.

[38] MA, L., ARULRAJ, J., ZHAO, S., PAVLO, A., DULLOOR, S. R., GIARDINO, M. J., PARKHURST, J., GARDNER, J. L., DOSHI, K., AND ZDONIK, S. Larger-than-Memory Data Management on Modern Storage Hardware for In-Memory OLTP Database Systems. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (2016), DaMoN '16, ACM, pp. 9:1–9:7.

[39] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! But at what COST? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2015), HOTOS'15, USENIX Association, pp. 14–14.

[40] MENG, X., BRADLEY, J., YAVUZ, B., SPARKS, E., VENKATARAMAN, S., LIU, D., FREEMAN, J., TSAI, D., AMDE, M., OWEN, S., XIN, D., XIN, R., FRANKLIN, M. J., ZADEH, R., ZAHARIA, M., AND TALWALKAR, A. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res. 17*, 1 (Jan. 2016), 1235–1241.

[41] MICRONSSD. UNVMe - A User Space NVMe Driver, 2016. `https://github.com/MicronSSD/unvme`.

[42] MICROSOFT. ISLE: Importance sampling-based algorithms for large scale topic modeling. `https://github.com/Microsoft/ISLE`, 2018.

[43] NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. Scalable Parallel Programming with CUDA. *Queue 6*, 2 (Mar. 2008), 40–53.

[44] NVIDIA. cuSPARSE library. `http://docs.nvidia.com/cuda/cusparse/index.html`, 2017.

[45] PCI-SIG. PCI Express Base Specification Revision 4.0, Version 1.0. `https://members.pcisig.com/wg/PCI-SIG/document/10912?downloadRevision=active`, October 2017.

[46] PRABHU, Y., KAG, A., HARSOLA, S., AGRAWAL, R., AND VARMA, M. Parabel: Partitioned Label Trees for Extreme Classification with Application to Dynamic Search Advertising. In *Proceedings of the International World Wide Web Conference* (April 2018).

[47] PRABHU, Y., AND VARMA, M. FastXML: A Fast, Accurate and Stable Tree-classifier for eXtreme Multi-label Learning. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2014), KDD '14, ACM, pp. 263–272.

[48] QIU, Y. Spectra - Sparse Eigenvalue Computation Toolkit as a Redesigned ARPACK, 2015. `https://spectralib.org`.

[49] ROCKLIN, M. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference* (2015), pp. 130–136.

[50] SCALEMP™. vSMP Foundation Flash Expansion. `http://www.scalemp.com/products/flx/`, 2018.

[51] SHAFIEE, A., NAG, A., MURALIMANOHAR, N., BALASUBRAMONIAN, R., STRACHAN, J. P., HU, M., WILLIAMS, R. S., AND SRIKUMAR, V. ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2016), ISCA '16, IEEE Press, pp. 14–26.

[52] SHUN, J., AND BLELLOCH, G. E. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2013), PPoPP '13, ACM, pp. 135–146.

[53] SHUN, J., ROOSTA-KHORASANI, F., FOUNTOULAKIS, K., AND MAHONEY, M. W. Parallel Local Graph Clustering. *Proc. VLDB Endow. 9*, 12 (Aug. 2016), 1041–1052.

[54] SORENSEN, D. C. Implicit Application of Polynomial Filters in a k-Step Arnoldi Method. *SIAM Journal on Matrix Analysis and Applications 13*, 1 (1992), 357–385.

[55] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., ET AL. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), ACM, p. 5.

[56] VITTER, J. S. External Memory Algorithms and Data Structures: Dealing with MASSIVE Data. *ACM Comput. Surv. 33*, 2 (June 2001), 209–271.

[57] WANG, K., ANGSTADT, K., BO, C., BRUNELLE, N., SADREDINI, E., TRACY, II, T., WADDEN, J., STAN, M., AND SKADRON, K. An Overview of Micron's Automata Processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (2016), CODES '16, ACM, pp. 14:1–14:3.

[58] WEIMER, M., CHEN, Y., CHUN, B.-G., CONDIE, T., CURINO, C., DOUGLAS, C., LEE, Y., MAJESTRO, T., MALKHI, D., MATUSEVYCH, S., ET AL. REEF: Retainable Evaluator Execution Framework. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 1343–1355.

[59] XIANYI, Z. OpenBLAS. `http://www.openblas.net/`, 2017.

[60] XING, E. P., HO, Q., DAI, W., KIM, J.-K., WEI, J., LEE, S., ZHENG, X., XIE, P., KUMAR, A., AND YU, Y. 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* (2015), KDD '15, ACM, pp. 1335–1344.

[61] YUAN, J., GAO, F., HO, Q., DAI, W., WEI, J., ZHENG, X., XING, E. P., LIU, T.-Y., AND MA, W.-Y. LightLDA: Big Topic Models on Modest Computer Clusters. In *Proceedings of the 24th International Conference on World Wide Web* (Republic and Canton of Geneva, Switzerland, 2015), WWW '15, International World Wide Web Conferences Steering Committee, pp. 1351–1361.

[62] YUT, L., ZHANG, C., SHAO, Y., AND CUI, B. LDA*: A robust and large-scale topic modeling system. *Proc. VLDB Endow. 10*, 11 (Aug. 2017), 1406–1417.

[63] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. 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)* (San Jose, CA, 2012), USENIX, pp. 15–28.

[64] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., GHODSI, A., GONZALEZ, J., SHENKER, S., AND STOICA, I. Apache spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.

[65] ZHENG, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. An SSD-based eigensolver for spectral analysis on billion-node graphs. *arXiv preprint arXiv:1602.01421* (2016).

[66] ZHENG, Y., KAMIL, A., DRISCOLL, M. B., SHAN, H., AND YELICK, K. UPC++: A PGAS extension for C++. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2014), IPDPS '14, IEEE Computer Society, pp. 1105–1114.

[67] ZHOU, Y., AND SAAD, Y. Block Krylov–Schur method for large symmetric eigenvalue problems. *Numerical Algorithms 47*, 4 (Apr 2008), 341–359.