BLAS-on-flash : An 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

B. Venkatesh
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 [34], spectral clustering [33], extreme multi-label learning [45], 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 [61] or systems based on Parameter Servers [37, 19, 58], 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 [23, 38]. This could be due to platform overheads as well as mismatch between the structure of the algorithm and the platform's programming model [56, 9, 18], 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 [35, 49, 50, 17]. 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 code for these algorithms is typically written as a main thread consisting of glue code that invokes linear algebra calls through standard APIs such as BLAS [10] and sparseBLAS [21]. High performance implementations for these standard APIs are provided by hardware vendors [27, 28, 41, 42].

Linear algebra kernels offer plenty of locality, so much so that the bandwidth required for running them on high-end multiprocessors can be provided by a non-volatile memory over PCIe or SATA bus [54, 5, 13]. 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 [3, 47, 20, 16]. Hardware technology and interfaces for non-volatile memories have increasingly lower end-to-end latency (few μs) [26] and higher bandwidth: from 4-8 GT/s in PCIe3.0 to 16GT/s in PCIe4.0 [43] 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 [57].

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.

The 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 eigen-solver which is critical to dimensionality reduction (e.g. PCA) and spectral methods. Specifically, we wrote an implementation of the restarted block Krylov-Schur [63] algorithm that can compute hundreds or thousands of eigen-vectors on SSD-resident data faster than standard in-memory solvers based on the IRAM algorithm [52] (e.g., Spectra [46], ARPACK [36]). On large bag of words text data sets running into hundreds of gigabytes, our implementation running on one multi-core workstation with under 50GB DRAM outperforms Spark MLlib's `computeSVD` [39] 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] involves summarizing 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 [59, 14, 60], 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 (500GB on disk) corpus 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 that has many applications in tagging, ranking and recommendation [8]. Models in extreme multi-label learning tasks are often ensembles of deep trees with smaller classifier(s) at each node. e.g. PfastreXML [45], Parabel [44]. 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 the dimensionality of data.

In this work, we start with PfastreXML and Parabel models and a dataset that needed 440 and 900 compute hours repsectively on a VM with large RAM. We optimized this code and reduced in-memory compute 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 fooprint.

These results suggest that for complicated 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 inferencing 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.

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 (Intel MKL) on in-memory data blocks with prefetching via a standard Linux asynchronous IO calls (`io_submit`) that use NVMe block drivers to access the SSD. The in-memory math and the IO calls can easily be replaced with other libraries. The size of matrices that the library can handle is limited by the size of the SSD.

The *BLAS-on-flash* library is intended for programmers who already write multi-threaded code in C++ using shared memory pointers. The interface of the library is C++ based and designed to make it easy to integrate with such code with a few modifications.

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

In similar spirit, the *BLAS-on-flash* library interface provides a `flash_ptr<T>` type to refer to large, possibly SSD-resident, objects in place of pointer type `T*`. The programmer can invoke functions provided by the library that operate on objects of type `flash_ptr<T>`. The library allows users to define new functions that operate on `flash_ptr<T>` by specializing the `Task` class. Programmers can define programs by stitching together a directed acyclic graph (DAG) of tasks using existing code with library-defined and user-defined tasks. 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 in-memory pointers of type `T` that allows the programmers to handle large blocks of SSD-resident data. There are two ways of creating a new `flash_ptr<T>`. The first is by allocating a large block on the disk using the allocator provided by the library. For example, using

```
flash_ptr<int> mat=flash::malloc<int>(len);
```

in place of `int *mat=(int *)malloc(len);` allows the user to create a large scratch space on SSD.

The second way to create a `flash_ptr<T>` is to map it to existing files. For example, using

```
flash_ptr<float> mat_fptr =
map_file<float>(mat_file, READWRITE);
```

allows read and write accesses to the floating point matrix in the file named `mat_file`. This allows the users to read and write to the file through the library. For example, the following writes N elements to the file mapped to `mat_fptr` from an in-memory buffer `mat_ptr`.

```
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.

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

2.2 Library Kernels

BLAS-on-flash kernels are functions that operate on `flash_ptr<T>` types, and are designed to be drop-in replacements for in-memory calls that operate 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)` $\star$ `op(B)` $+ \beta \cdot$ `C`, where α and β are floating point scalars, and `op(·)` is either the matrix `X` or its tranpose. The library allows striding and layout choices that 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(X)` 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` for in-memory vectors `b` and `c`, where `op(X)`=`X` or $\texttt{X}^\texttt{T}$.
- `csrcsc` : Converts a sparse matrix in CSR form into its Compressed Sparse Column (CSC) form with both inputs and outputs as `flash_ptr<float>`. This is equivalent to computing transpose of the input matrix.
- `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` : Sorts an array of type `flash_ptr<T>` using a user-defined comparator using the parallel SampleSort algorithm.

Using these kernels, one could overcome the memory limitations faced by 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 data.

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 and provides a mechanism to compose a computation graph by allowing parent-child relationships between tasks. Our scheduler guarantees that child tasks will not be executed until its parent tasks are complete.

Task inputs and outputs are uniquely described using an *access specifier*: ⟨`flash_ptr<T>, StrideInfo`⟩. Here `flash_ptr<T>` describes the start of an access 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 length to stride 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 input and output, the user must supply the `execute` function to complete the definition of a new task. Inputs specified in the task interface are made available as in-memory buffers for the `execute` function to acceess. Once the execution is complete, the buffers marked as outputs through the task interface are written back to their corresponding location on file. Figure 1 (a) 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 task to child task. 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 implemention using the Block Matrix Multiplication algorithm. For inputs A, B, and C, shown with 16 blocks for each matrix, the 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. $G_{i,j}$ indicates 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 the 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 packaged into a single module representing the kernel. This helps the user adapt in-memory code to use our library by modifying one computational kernel at a time. We demonstrate this by adapting the memory-intensive kernels in the ISLE topic modeling algorithm.

3 Library Design

In this section, we'll enumerate the technical challenges our library solves to achieve near in-memory performance for matrix operations with a small amount of DRAM. To motivate these challenges, we use the example of a `gemm` kernel on single precision (32 bit) floating point matrices A, B, and C of sizes 32768×32768 each, blocked as shown in Figure 1. Assume that the matrix block size is 8192×8192. Consider its execution 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.

3.1 Pipelining

Since our library focuses on batch compute, utilizing disk bandwidth, rather than minimizing disk access latency, is critical for performance. Since PCIe-based SSDs have limited bandwidth, where possible, the library must overlap prefetching inputs and writing back outputs with actual computation. To maximize throughput from the system, we saturate all available cores with compute and use hyper-threading to perform IO. To illustrate the balance between computation and IO, we present an analysis of the `gemm` kernel.

Each task in the `gemm` kernel performs 1TFLOP of compute on 768MB of input to produce 256MB output. On our `test` system, each such task requires 0.75s of IO time for 1s of compute (using all 32 threads per 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, instead of 64s if executed completely in memory. However, if we prefetch and issue write-backs for next and previous tasks respectively during any task's compute time, we can in principle complete the kernel in 64.75s accounting for prefetch of first task and write-back of last task. In reality, however, we noticed that mixing reads and writes results in less than peak bandwidths for just reading and just writing (mea-

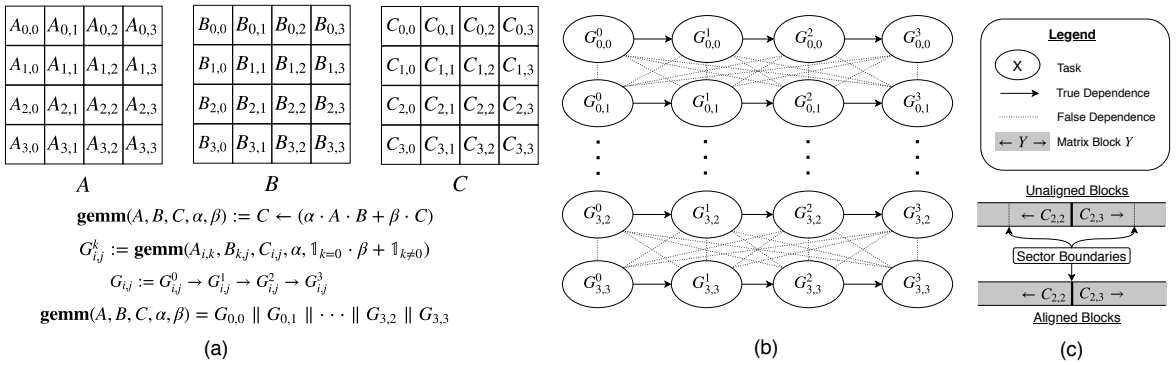

Figure 1: The `gemm` kernel, its DAG using the *Task* interface, and sector-sharing among adjacent output blocks in `C`.

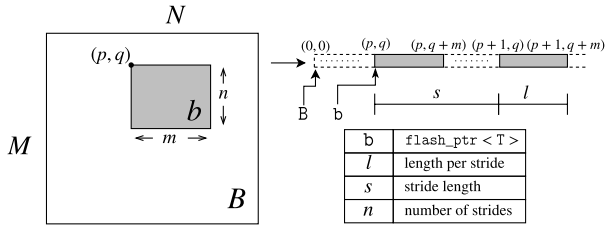

Figure 2: $\langle \mathtt{b}, \{l, s, n\}\rangle$ is an access specifier for block b of a flash-resident matrix B stored in Row-Major layout.

sured with the `fio` tool[4]). As a result, the kernel runs slower than would be expected with perfect pipelining. Getting the kernel to run at in-memory speed requires additional measures to save bandwidth.

3.2 Buffer Sharing

To increase pipelining efficiency, we use knowledge of task inputs and outputs to increase reuse of data prefetched into memory. Certain kernels like `gemm` perform $O(n^3)$ computation on inputs of size $O(n^2)$. This gap between computation and IO implies that prefetched data can be reused. A naive task scheduler executing the `gemm` kernel using the DAG in Figure 1b can perform $\frac{n^3}{b}$ reads resulting in multiple copies of the same data being present in memory. In Figure 1b, if $G^1_{0,0}$ and $G^0_{1,0}$ execute in parallel, a naive scheduler would prefetch $B_{0,0}$ twice. Data duplication and redundant IO adversely affect pipeline efficiency, and can also reduce effective available memory which could stall prefetching. Redundant IO results in more mixing of read and write accesses to disk than necessary, resulting in worse performance than expected.

We reduce such redundancy by mapping access specifiers to unique buffers. This allows the *BLAS-on-flash* scheduler to issue only one read to fetch $B_{0,0}$ and the same buffer is provided to both $G^1_{0,0}$ and $G^0_{1,0}$ as read-only buffers.

3.3 Prioritized Scheduling

How does one build a real-time online scheduler that can maximize buffer sharing among tasks in a dynamic DAG with little context? In a First-In-First-Out(FIFO) task scheduler, any occurrence of buffer sharing is purely accidental. We propose a heuristic to select a task to prefetch based on the data currently buffered in memory and the input requirements of the tasks in the DAG that are ready at the moment.

Our heuristic selects the task that requires the fewest number of bytes to be prefetched given the current content of the memory buffer. This forces buffer sharing and maximizes overlap with in-memory buffers while reading the minimum number of bytes from disk. For kernels like `gemm`, shown in Figure 1b, our heuristic keeps an output matrix block, $C_{i,j}$, in memory and executes the accumulate chain $G_{i,j}$ on it. $C_{i,j}$ is written back to disk only once, at the end of the chain $G_{i,j}$. Furthermore, our heuristic schedules accumulate chains with high input and output locality i.e. chains that use adjacent blocks in both input and output matrices. For sparse kernels, like `csrmm`, our heuristic achieves optimal buffer sharing by scheduling tasks with a common input block.

Consider the following example of the `gemm` kernel (Figure 1). Let $M = \{A_{0,0}, A_{1,0}, A_{1,1}, B_{0,0}, B_{1,0}, B_{1,1}, C_{0,0}, C_{1,0}, C_{1,1}\}$ be the set of all matrix blocks in memory and the following be four tasks part of the `gemm` kernel.

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

Let $G^1_{0,0}, G^0_{1,0}$, and $G^1_{1,1}$ be the latest 3 tasks to complete execution. Since $G^0_{1,0}$ has completed, its child, $G^1_{1,0}$, is now ready for execution. By scheduling $G^1_{1,0}$ instead of a 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

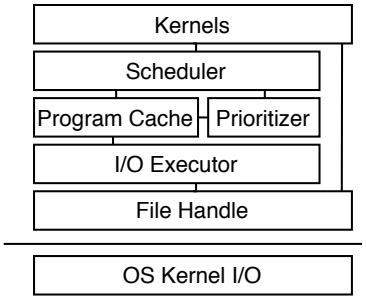

Figure 3: The *BLAS-on-flash* software stack

locality, such *nearby* accumulate chains execute concurrently and share more buffers than non-nearby chains.

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. Specialized implementations can be provided for different hardware IO interfaces (e.g. SATA or network). We implement the IO interface for SSDs using the Linux NVMe driver. This implementation uses the Linux kernel asynchronous IO syscall interface, `io_submit`, to submit IO jobs and `io_getevents` to reap completions of operations on flash-resident files. The `io_submit` syscall interface provides a simple interface to bypass kernel buffers and execute asynchronous IO at sector-level. We chose this over user-space NVMe drivers like `SPDK` [25] and `unvme` [40] because of its simplicity. Each access specifier for large matrix blocks translates into a list of contiguous accesses that can listed using an array of `iocb` structs that can be submitted at once. The large number of simultaneous accesses saturates available bandwidth.

IO Executor consists of a threadpool that accepts IO requests generated by the Program Cache and executes non-overlapping requests concurrently. Overlap check is necessary for ensuring correctness since the IO layer does not attempt to serialize access to sectors. Consider Figure 1c. When matrix block sizes are aligned to a multiple of the flash device's sector size, $C_{2,2}$ and $C_{2,3}$ can be read from and/or written into concurrently. However, if $C_{2,2}$ and $C_{2,3}$ happen to share sectors on the disk i.e. there exist some sectors on the device containing portions of both $C_{2,2}$ and $C_{2,3}$, writes to the common sectors must be ordered to avoid data corruption. Every write access is advertised and followed up with an overlap check on other threads' writes. If a conflict is detected, the thread adds the access to a local backlog and requests a different access to execute. Backlog accesses are revisited in the next cycle and executed if no overlap is detected.

Program Cache tracks *BLAS-on-flash* memory usage, maps access specifiers to buffers and stores the mapping as a unique entry. Each entry is given one of four tags - Active(A), Prefetch(P), Write-Back(W), or Zero-Reference(Z). Tag A against an entry indicates an active reference, i.e, at least one task has a reference to the buffer. Tag P indicates that a prefetch associated with the access specifier is in progress, W indicates a write-back in progress, and Z otherwise. It uses this information to serve five types of requests.

- Batch HIT/MISS requests ask if buffers corresponding to a batch of access specifiers are in memory. A response is returned for each unique access specifier.
- A COMMIT request asks the cache to *commit* a task to memory by issuing prefetch requests to the IO Executor. A task is said to be committed if all its inputs and outputs can be allocated by either provisioning unused memory or by evicting unused buffers (i.e. its entry has a Z tag). If the request is successful, some buffers with Z tag are evicted to free up memory. This causes some write-backs and prefetch requests are enqueued to a backlog. Program Cache state is unchanged if the request is unsuccessful.
- A RELEASE request returns a task's inputs and outputs to the cache. Reference counts are decreased for all returned buffers, and entries tagged A with zero references are tagged with Z and made available for eviction in future.
- A SERVICE request checks entries tagged W for completion of write-backs and frees the associated buffer if the operation is complete. If memory is available, backlog prefetch requests are issued to the IO executor and are tagged with P.
- A GET request queries the status of entries tagged P that have issued prefetch requests. If an entry tagged P has finished prefetching, it is tagged with A and the associated buffer is returned as response; a NULL buffer is returned otherwise.
- A FLUSH request evicts all buffers tagged Z and issues write-backs if required. A, P and W tagged entries are not affected.

Prioritizer implements scheduling heuristics by maintaining a list of ready tasks and issuing HIT/MISS requests to the Program Cache. Other task selection heuristics can be implemented in place of Prioritizer.

Scheduler provides an interface for users to inject tasks at runtime and track their progress through the 4 stages of the pipeline — Wait, Prefetch, Compute and Complete. It also provides an interface to enable/disable the Prioritizer, enable/disable overlap checks in IO executor and exposes the FLUSH request to the Program Cache to the user.

To select the next task to prefetch, the Scheduler queries the Prioritizer and issues a ALLOCATE request to the Program Cache. A task is said to be Complete if it has finished all 4 stages. It is then removed from the DAG and a RELEASE request is issued for the task to the Program Cache.

5 Algorithms and Evaluation

We now discuss the implementation and performance of the kernels provided by the library as well as algorithms built using the library — eigensolvers, an SVD-based algorithm for topic modeling, and inference in extreme multi-label learning. We will compare the running time and memory requirements of in-memory and SSD-based versions of these algorithms. We use Intel MKL 2018 and Ubuntu 16.04LTS in all our experiments.

5.1 Machines

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 for the strided accesses created by our library. `z840` is chosen to represent a typical bare-metal workstation machine configured with two Samsung 960EVO SSDs in RAID0 configuration. This provides sustained read speed of about 3GB/s and write speed of about 2.2GB/s. `L32s VM` is a virtual machine configured for heavy IO. We believe that the underlying hardware supports very high bandwidths; however, the hypervisor throttles it to 1.6GB/s sustained reads and writes or 160K IO ops/second. `M64-32ms VM` is a virtual machine with 1.7TB RAM which we use for running in-memory experiments that require large amounts of memory. The Apache Spark instances used for comparison run MLLib 2.1 on `DS14v2 VM` clusters.

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 [31, 29, 5]. We also use the fact that BLAS and sparseBLAS computations can be tiled so that they write the output to disk just once [13, 12], 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 4 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. For example, a 1024x1000 `float` matrix in row-major layout that would require 4096 bytes for each row is 512-aligned.
- **unaligned**. A matrix is said to be *unaligned* if it is **not** 512-aligned. For example, a 500x500 matrix with 32-bit `floats` in row-major form would require 2000 bytes, which is not a multiple of 512, and is said to be unaligned.

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. To flush an unaligned matrix block, we need to read in the start and end sectors of each row in the block, overwrite them with new values and then issue a write to disk. In the case of aligned matrix blocks, we need only one write to flush it to the disk.

We define the *reduction dimension* to be the dimension along which summation is carried out during matrix multiplication. Using notation from Figure 1, if all three matrices, A, B, and C, are stored in row-major form, reduction dimension is the number of columns in A, or equivalently the number of rows in B. For a given block size, increasing reduction dimension increases the length of the accumulate chain, which in turn translates to fewer disk writes per FLOP. Since write bandwidth is low, and writes affect read bandwidth as well, our scheduler should use a smaller ratio of writes to FLOPs to improve performance for larger reduction dimension. Figure 4 demonstrates that this is indeed the case. In fact, because of careful pipelining, our library outperforms in-memory MKL calls in many instances.

Further, as expected, *BLAS-on-flash* performs better on 512-aligned instances than in the unaligned instances. We attribute this to the increased number of requests issued to the disk in the unaligned case.

`csrmm`. The `csrmm` kernel requires $O(n^3 * s)$ floating point operations on an input matrices of $n \times n$ di-

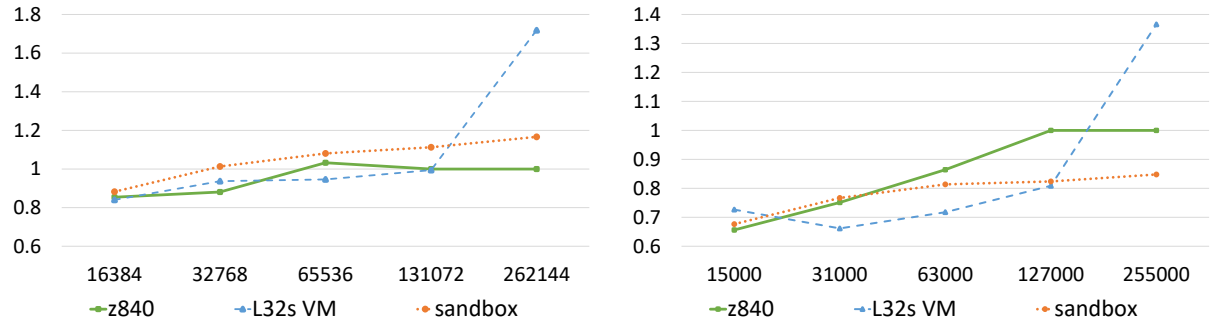

Figure 4: Ratio of in-memory MKL `gemm` to *BLAS-on-flash* `gemm` running times for 512-aligned (left) and unaligned (right) instances for various values of reduction dimension (d). The matrix dimensions are $2^{15} \times d \times 2^{15}$ and $31000 \times d \times 31000$ for the aligned and unaligned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

mensions with sparsity s (this represents an input size of $O(n^2(1+s))$ and output size of n^2). 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 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 tasks in `csrmm` loaded into memory 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 5 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 $s \in \{10^{-2}, 10^{-3}, 10^{-4}\}$. It is evident that the efficiency of the `csrmm` kernel decreases with sparisty.

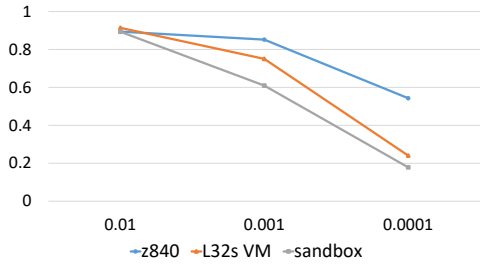

Figure 5: Ratio of in-memory MKL `csrmm` to *BLAS-on-flash* `csrmm` running times for $(2^{20} \times 2^{17} \times 2^{12})$ sized instances and various values of sparsity. *BLAS-on-flash* library has a 8GB Program Cache. `csrmm` tasks in *BLAS-on-flash* library use 4 threads each and the number of simultaneous tasks is detemined by Program Cache.

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 matrix and output matrices are in a row-major format. Note that this means that the matrices are 512-aligned. A performance drop is expected in the unaligned case just as in `gemm`.

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 implemented 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 sparsiy 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 over $2\times$ on Small and Medium datasets. We attribute this to poor multi-threading in MKL's implementation. We conclude that the *BLAS-on-flash* `csrmm` kernel is reasonably efficient even on large and extremely-sparse inputs.

`csrcsc`. To support the compressed sparse column format (CSC), we implemented a kernel `csrcsc` to convert a large sparse matrix between the CSR and CSC formats. This kernel is relatively fast and takes about 100 seconds for a 100GB matrix. Intel MKL's `csrcsc` call fails on

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 projects 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.

Medium and Large data sets.

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

$$\begin{aligned} \mathbf{A}\mathbf{v}_i &= \lambda_i \mathbf{v}_i \qquad \forall i \\ \mathbf{v}_i^T \mathbf{v}_j = 0, \|\mathbf{v}_i\|_2 &= 1 \qquad \forall i \neq j \\ |\lambda_1| \geq &|\lambda_2| \ldots \geq |\lambda_k| \end{aligned}$$

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 following equations describe SVD of a matrix $\mathbf{M}$, and how it may be formulated as a symmetric eigen-decomposition problem.

$$\begin{aligned} \mathbf{M}\mathbf{u}_i = \sigma_i \mathbf{v}_i,\ \|\mathbf{v}_i\|_2 = \|\mathbf{u}_i\|_2 = 1 \forall i \\ \mathbf{u}_i^T \mathbf{u}_j = 0,\ \mathbf{v}_i^T \mathbf{v}_j = 0 \, \forall i \neq j \\ |\sigma_1| \geq |\sigma_2| \ldots \geq |\sigma_k| \\ \mathbf{M}\mathbf{M}^T \mathbf{u}_i = \sigma_i^2 \mathbf{u}_i,\ \mathbf{M}^T \mathbf{M} \mathbf{v}_i = \sigma_i^2 \mathbf{v}_i \\ \texttt{svd}(\mathbf{M}) = \texttt{syevd}(\mathbf{M}\mathbf{M}^T) = \texttt{syevd}(\mathbf{M}^T\mathbf{M}) \end{aligned}$$

To showcase the versatility of our library, we implement a symmetric eigensolver and time it on large matrices (in CSR format) obtained from text corpora in Table 2. Eigensolvers are typically iterative and involve many subroutines. Among the many flavors of eigensolvers available, we picked the Kyrlov-subspace class of algorithms because they have been shown to be stable for a wide-variety of matrices. This class of algorithms uses iterated Sparse Matrix-Vector (csrgemv) products to converge on the solution pairs.

csrgemv is bandwidth bound and not suitable for an eigensolver operating on SSD-resident matrices. To overcome this limitation, we implemented the Restarted Block Krylov-Schur (Block KS) [63] algorithm. The Block KS algorithm can potentially use fewer matrix accesses to achieve the same tolerance by using a csrmm kernel in place of csrgemv by expanding the Krylov basis several columns at a time. Although the Block KS algorithm performs extra computation compared to its non-block variants, this extra work is highly parallel and the IO saved makes up for the extra compute.

The running time of an eigensolver depends not only on the input size and the number of eigenpairs desired, but also the distribution of the eigenvalues of the matrix. The further apart eigenvalues are spaced, the faster the convergence. The largest singular values of the sparse matrices we use are well separated – this helps Block KS converge faster without many restarts.

Evaluation. We benchmark both our in-memory and flash-based single node implmentations of the Block KS algorithm against a single node and distributed implementation of the Implicitly Restarted Arnoldi Method (IRAM) algorithm. The single node version is provided by $Spectra$ [46], a C++ header-only implementation of $ARPACK$ [36], while the distributed version is computeSVD in Apache Spark MLLib library (v2.1). The Spark job was deployed on both a shared and a dedicated Hadoop cluster through YARN [53] to workers with 1 core and 8GB memory each and a driver node with 96GB memory. The shared cluster uses Xeon E5-2450L processors and 10Gb Ethernet, while the dedicated cluster uses DS14v2 VM nodes.

Table 4 compares the time taken to solve for the top singular values of sparse matrices in Table 2 to a tolerance precision of 10^{-4} (this is sufficient for the SVD-based topic modeling algorithm described in the next subsection). It must be noted that the computeSVD uses double precision floating point numbers while our algorithm uses single precision. We solve for only 200 singular values on the large data set and 500 on the Medium data set because the Spark solver would not solve for more. On the other hand, our implementation easily scales to thousands of singular values.

The flash version of Block KS works almost as fast as the in-memory version. Further, both Block KS implementations easily outperform both Spectra and Spark jobs in time to convergece. Spark does not see any benefit from more workers beyond a point; in fact it becomes slower. These results demonstrate that our flash-based eigensolver utilizes hardware order(s) of manginutes 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

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 clsuter. On standalone cluster with 64 works, the Large dataset needed 10GB memory per worker.

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 $P_{\cdot j}$ represents the probability of words in the document j, and (3) sampling the observed documents $A_{\cdot j}$ using a multinomial distribution based on the p.d.f. $P_{\cdot j}$. 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. It is based on linear-algebraic techniques (unlike other LDA based algorithms which are based on MCMC techniques) that can provably recover the underlying topic matrix under reasonable assumptions on the observed data. Empirically, it has been shown to yield qualitatively better topics in real world data. The open source implementation is faster than other single node implementations of any topic modeling algorithms [51]. 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 on 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 linked ISLE to the *BLAS-on-flash* framework to leverage the flash-based Block KS eigensolver and the clustering algorithms built in our framework.

Evaluation. Table 5 compares the running times of the in-memory version and the version that uses 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 budget for in-memory buffers. We note that the number of tokens in this data set (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		L32s VM	
		in-mem	flash	in-mem	flash
Small(1000)	1.0	15	27	18	37
Medium(1000)	0.1	46	66	63	72
Medium(2000)	0.1	119	144	158	212
Large(1000)	0.1	–	149	163*	172
Large(2000)	0.1	–	228	285*	279
Large(5000)	0.1	–	522	980*	664
Large(2000)	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 data set 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 while requiring under 128GB of main memory. For the Large dataset, the flash version running on sandbox is able to perform better than the in-memory version on M64-32ms VM . We think this is because sandbox has newer hardware, and because the eigensolver and kmeans kernels written using *BLAS-on-flash* achieve near in-memory performance.

5.5 Extreme Multi-Label Learning

Extreme multi-label (XML) learning 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. For instance, one might wish to build an extreme multi-label classifier that recommends a subset of millions of Amazon items that a user might wish to buy or view next after buying or viewing a given item. Many popular XML algorithms use tree based methods due to low training and prediction times. Here, we present experiments with two such algorithms which use ensembles of trees: PfastreXML [30] and Parabel [44].

In a current deployment, both these algorithms train an ensemble of trees (50 trees for PfastreXML, and 3 for Parabel) from 40 million data points, each of which is a

sparse vector in a 4.5 million-dimensional space. Once trained, each tree in the ensemble predicts label probabilities/ranks for 250 million test data points. The test dataset takes 500GB of space when stored in a sparse format. Both training and inference are difficult to scale – training requires weeks on a machine with multiple terabytes of RAM, and inference currently consumes dozens of machines. As XML algorithms are applied to larger search problems and advertisement domains, it is projected that they need to scale to much larger datasets with billions of points and hundreds of millions of labels (e.g. web search), and train trees that are hundreds of gigabytes in size.

In such scenarios, 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 here since it is run at a much higher frequency than training — typically once every week in ad recommendation applications. Similar techniques can be applied for training.

PfastreXML: For training, trees are grown by recursively partitioning nodes starting at the root until each tree is *fully grown*. Nodes are split by learning a hyperplane which partitions training points between left and right children. Node partitioning terminates when a node contains less than a certain number of points. Leaf nodes contain a probability distribution over label. During inference, predictions are made in logarithmic time by passing a test point down from the root to a leaf in each of the trees in ensemble. At each internal node, the test point is sent to the left or right child based on its position with respect to the hyperplane at that node.

Parabel: For training, trees are grown by recursively partitioning the nodes by distributing the labels assigned to the node in equal amounts to each of its two children. Nodes containing less than a user specified number of labels are split into multiple leaf nodes with each label allotted to a separate leaf. Each tree node contains a probabilistic linear classifier which decides whether a data point has some relevant labels in its left or right subtree. The node classifiers are trained so as to maximize the aposteriori probability distribution over the training data. For inference, predictions are made in logarithmic time by passing a test data point down a fixed number of paths, at most 10 in our case, in each of the balanced trees of the ensemble. Along each such path from root to a leaf, the probabilities estimated by classifiers are multiplied to give the corresponding path probability. Finally, the labels are ranked in the decreasing order of the average probabilities where average is over paths that lead to the label in different trees.

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 D14 v2 SKUs with 112GB RAM and 16 cores. The orchestration required to complete the inference in under two days is complex and increases the likelihood of disk, node or network failures.

In the code we started from, PfastreXML inference involves a depth-first traversal of a non-balanced binary tree while Parabel inference requires breadth-first beam search of a balanced binary tree. In both cases, the noticed that the baseline code was inefficient. We improved the code to take a batch of test data points (about 2-4 million per batch) and traverse the tree in a Breadth-First order, i.e., level by level. With this transformation, the in-memory running times of the inference code improved by about $6\times$ on nodes with a large amount of RAM. We feel this is close to the limit of how fast this code can run based on DDR3 bandwidth.

We use the *BLAS-on-flash* library to orchestrate level-by-level traversal of the trees for a batch of points with the *Task* interface. We construct one task for a (level, batch) pair and dynamically inject new tasks into the scheduler. Since inference is data parallel for a given (level, batch) pair, we execute multiple batches in parallel and rely on the *BLAS-on-flash* library to prefetch levels and points.

Evaluation. We compare the in-memory and *BLAS-on-flash* -based versions 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, 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 dataset does not fit in the memory of any machine in the platform. We use a total of 50 trees for PfastreXML and 3 trees for Parabel inference. Our test data consists of 250 million points drawn from a 4.3 million dimensional space and is about 500GB in compressed sparse format.

We benchmark both inference algorithms on `z840` , `L32s VM` , and `sandbox` and use 2^{21} points per batch for `z840` and 2^{22} points for `L32s VM` and `sandbox` . The size of Program Cache for the *BLAS-on-flash* library is 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` .

Tables 6 and 7 present the running time and memory requirement of our code on the medium- and large-sized models. Using the *BLAS-on-flash* library, the inference code 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 data batch size or splitting each level of the tree into multiple tasks. The reduction in working set with practically no impact on

	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)

Table 6: Running time in hours and peak DRAM usage in GB (inside paranthesis) for XML inference on an ensemble of medium-sized trees for 250×10^6 data points. We used 64 threads on sandbox and 32 threads on L32s VM and z840 .

	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 7: Running time in hours and peak DRAM usage (in GB) for inference on the large Parabel model for 250 million data points. We used 64 threads on sandbox and 32 threads on L32s VM and allocate 70GB as Program Cache budget.

performance critically enables us to execute inference on larger models (for ranking and relevance tasks) that do not fit in DRAM, for greater accuracy.

6 Other Related Work

Recent work [12, 7, 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. Fortunately, in the case of linear algebra, this is not the case. 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.

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, tranpose, 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 [48, 24, 55]. Some of the techniques in our work are complementary to this line of work.

Partitioned Global Address Space systems such as FaRM [20] and UPC [22, 15, 62, 32] 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.

7 Conclusion

Our results suggests that operating on data stored in fast non-volatile memories on a singlde node could provide an efficient alternative to distributed big-data systems for training and inference of industrial scale machine learning models for algorithms that are not computation intensive. On complicated numerical algorithms such as eigensolvers, we demonstrated that careful co-design of algorithm and software stack can offer large gains in hardware utilization and keep the costs of data analytics pipelines low. Further, our library provides a higher value proposition for the large quantity of NVM storage that has already been 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.

8 Acknowledgments

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

9 Availability

The topic modeling training and extreme multi-label learning inference code that we have adapted to use the *BLAS-on-flash* library can be downloaded from the two following sites:

github.com/Microsoft/ISLE

manikvarma.org/downloads/XC/XMLRepository.html

We intend to release our library, the eigensolvers, and the adapted code on github once the conditions of anonynomity are removed.

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* (New York, NY, USA, 2015), ISCA '15, ACM, pp. 131–143.

[2] AMD. Radeon™ Pro SSG, 2018.

[3] 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* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 1753–1758.

[4] AXBOE, J. Fio-flexible io tester. *URL http://freecode.com/projects/fio* (2014).

[5] BALLARD, G., DEMMEL, J., HOLTZ, O., AND SCHWARTZ, O. Minimizing communication in numerical 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* (New York, NY, USA, 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.

[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* (New York, NY, USA, 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* (New York, NY, USA, 2003), ICS '03, ACM, pp. 63–73.

[16] DASK DEVELOPMENT TEAM. *Dask: Library for dynamic task scheduling*, 2016.

[17] 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* (New York, NY, USA, 2017), SPAA '17, ACM, pp. 293–304.

[18] 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* (New York, NY, USA, 2016), SPAA '16, ACM, pp. 49–60.

[19] DMTK. Multiverso: Parameter server for distributed machine learning, 2015.

[20] 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).

[21] 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.

[22] EL-GHAZAWI, T., AND SMITH, L. Upc: Unified parallel c. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.

[23] 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 Factorization at Scale: a Comparison of Scientific Data Analytics in Spark and C+MPI Using Three Case Studies. *ArXiv e-prints* (July 2016).

[24] 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* (New York, NY, USA, 2011), MICRO-44, ACM, pp. 339–350.

[25] INTEL. Storage performance development kit (spdk), 2016.

[26] INTEL®. Optane™ memory, 2017.

[27] INTEL®. Math Kernel Library Sparse BLAS level 2 and 3 routines, 2018.

[28] INTEL®. Math Kernel Library Sparse BLAS level 2 and 3 routines, 2018.

[29] 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.

[30] 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).

[31] 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* (New York, NY, USA, 1981), STOC '81, ACM, pp. 326–333.

[32] 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* (New York, NY, USA, 2014), ARRAY'14, ACM, pp. 26:26–26:31.

[33] KANNAN, R., VEMPALA, S., AND VETTA, A. On clusterings: Good, bad and spectral. *J. ACM 51*, 3 (May 2004), 497–515.

[34] 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.

[35] 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.

[36] LEHOUCQ, R., MASCHHOFF, K., SORENSEN, D., AND YANG, C. ARPACK software, 2009.

[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] 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.

[39] 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.

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

[41] NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. Scalable parallel programming with cuda. *Queue 6*, 2 (Mar. 2008), 40–53.

[42] NVIDIA. cuSPARSE library, 2017.

[43] PCI-SIG. Pci express base specification revision 4.0, version 1.0, October 2017.

[44] 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).

[45] 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* (New York, NY, USA, 2014), KDD '14, ACM, pp. 263–272.

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

[47] SCALEMP™. vSMP Foundation Flash Expansion, 2018.

[48] 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.

[49] 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* (New York, NY, USA, 2013), PPoPP '13, ACM, pp. 135–146.

[50] SHUN, J., ROOSTA-KHORASANI, F., FOUNTOULAKIS, K., AND MAHONEY, M. W. Parallel local graph clustering. *Proc. VLDB Endow. 9*, 12 (Aug. 2016), 1041–1052.

[51] SIMHADRI, H. V. Svd and importance sampling-based algorithms for large scale topic modeling. `https://github.com/Microsoft/ISLE`, 2017.

[52] 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.

[53] 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.

[54] VITTER, J. S. External memory algorithms and data structures: Dealing with massive data. *ACM Comput. Surv. 33*, 2 (June 2001), 209–271.

[55] 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* (New York, NY, USA, 2016), CODES '16, ACM, pp. 14:1–14:3.

[56] 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.

[57] XIANYI, Z. Openblas, 2017.

[58] 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* (New York, NY, USA, 2015), KDD '15, ACM, pp. 1335–1344.

[59] 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.

[60] 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.

[61] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.

[62] 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.

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