BLAS-on-flash

An Efficient Alternative for Scaling ML training and inference with NVMs

Suhas JS, Harsha Simhadri, Srajan Garg, Anil Kag, Venkatesh B

Microsoft Research India

Scope | Memory-Intensive non-DL workloads

scikit learn ML.NET mlpack + OpenBLAS INTEL® MKL

Typical use-cases

- Classification / Regression
- Topic Modeling
- Matrix Factorizations
- Clustering

Distributed ML | Current Landscape

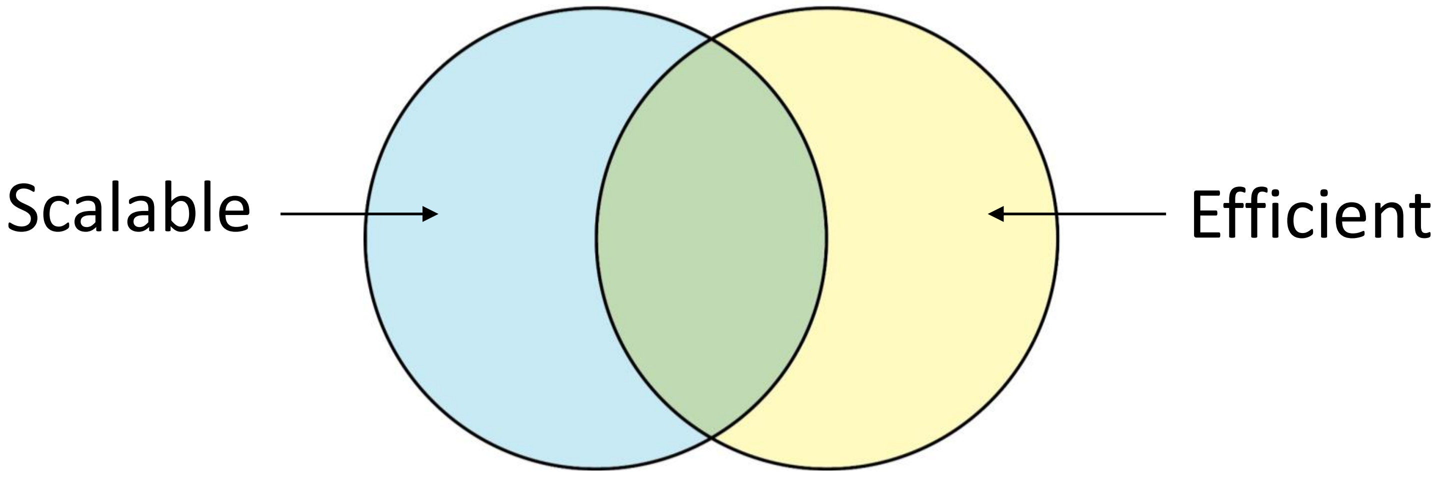
Pros

- Terabyte-scale machine learning
- Decent speedups on large clusters
- Widely used in production

Cons

- High setup + maintenance cost
- Code rewrite using specific abstractions
- Platform and programming inefficiencies

“Scalability” | Compact systems



- **GraphCHI** [Kyrola et al., OSDI’12]
- **Scalability! But at what COST?** [McSherry, Isard, Murray, HotOS’2016]
  - Are big ML platforms really scalable or more useful than single node platforms?
- **Ligra** [Shun, Blelloch, PPoPP’13], **Ligra+ ..**
  - Web scale graph processing on a single shared memory machine

BLAS-on-flash | Overview

Observations

- Legacy code = multi-threaded code + math library calls
- High locality in BLAS-3 operations $\Longrightarrow$ PCIe-SSDs' bandwidth sufficient

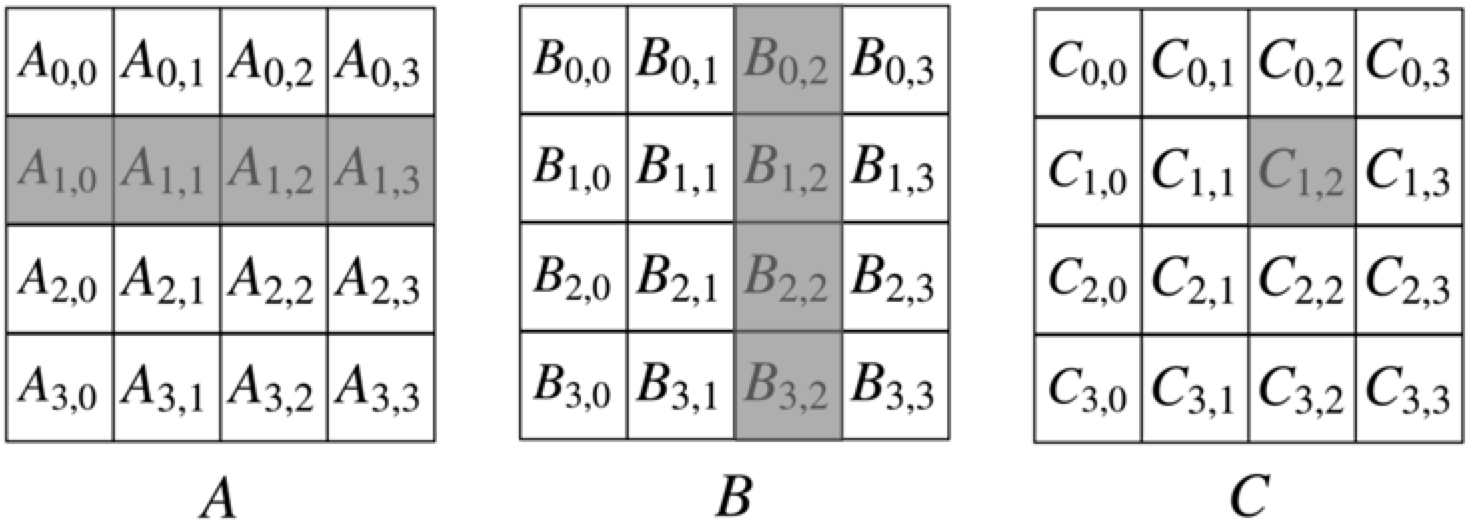
Contributions

- A library of matrix operations for large SSD-resident matrices (GBs – TBs)
- Link to legacy code via the standard BLAS and sparseBLAS API
- DAG definition + online scheduling to execute data-dependent computation

API | In-memory ⟶ BLAS-on-flash

In-memory	BLAS-on-flash
`- float *A;`	`+ flash_ptr<float> A;`
`- float* mat =` `(float*)malloc(len);`	`+ flash ptr<float> mat =` `flash::malloc<float>(len);`
`- sgemm(args, A, B, C);`	`+ flash::sgemm(args, A, B, C);`
`- legacy_fn(A);`	`+ float* mmap_A = A.ptr;` `+ legacy_fn(mmap_A);` // correct, but <u>possibly slow</u>

gemm | Task View



$$C_{1,2} = \sum_{j=0}^{j=3} A_{1,j} \cdot B_{j,2}$$

$$\mathrm{GEMM}(\mathbf{X}, \mathbf{Y}) := \mathbf{X} \cdot \mathbf{Y}$$

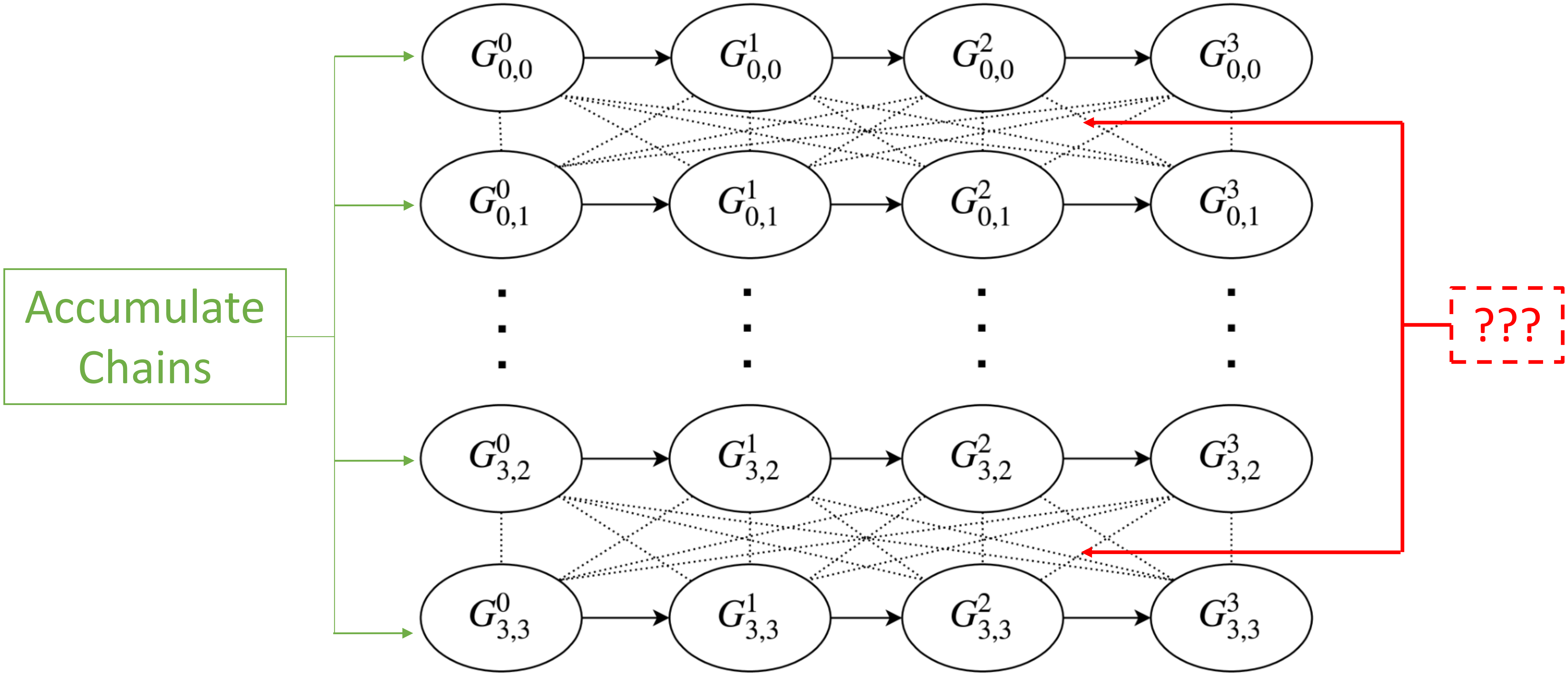
$$\begin{aligned} C_{1,2} = {} & \mathrm{GEMM}(A_{1,0}, B_{0,2}) + \mathrm{GEMM}(A_{1,1}, B_{1,2}) \\ & + \mathrm{GEMM}(A_{1,2}, B_{2,2}) + \mathrm{GEMM}(A_{1,3}, B_{3,2}) \end{aligned}$$

`gemm` | Chain View

$$C_{1,2} = \sum_{j=0}^{j=3} A_{1,j} \cdot B_{j,2}$$

$$G_{1,2} := \begin{array}{c}\mathrm{GEMM}(A_{1,0}, B_{0,2})\\ [G^0_{1,2}]\end{array} \rightarrow \begin{array}{c}\mathrm{GEMM}(A_{1,1}, B_{1,2})\\ [G^1_{1,2}]\end{array} \rightarrow \begin{array}{c}\mathrm{GEMM}(A_{1,2}, B_{2,2})\\ [G^2_{1,2}]\end{array} \rightarrow \begin{array}{c}\mathrm{GEMM}(A_{1,3}, B_{3,2})\\ [G^3_{1,2}]\end{array}$$

gemm | DAG view



gemm | Kernel – Task Creation

```
gemm(flash_ptr<float> A,
     flash_ptr<float> B,
     flash_ptr<float> C, args...){
  GemmTask tasks[4][4][4];

  // create tasks
  for(i : {0, 1, 2, 3}){
    for(k : {0, 1, 2, 3}){
      for(j : {0, 1, 2, 3}){
        // C_ik += A_ij * B_jk
        tasks[i][k][j] = GemmTask(A_ij, B_jk, C_ik, args...);
      }
    }
  }
```

gemm | Kernel – DAG Creation

```
  // create accumulate chains
  for(i : {0, 1, 2, 3}){
    for(k : {0, 1, 2, 3}){
      // accumulate chain for C_ik
      for(j : {0, 1, 2}){
        tasks[i][k][j].add_parent(tasks[i][k][j+1]);
      }
    }
  }
```

gemm | Kernel – DAG Submission

```
  // submit tasks to Scheduler
  for(i : {0, 1, 2, 3}){
    for(k : {0, 1, 2, 3}){
      for(j : {0, 1, 2, 3}){
        Scheduler.submit_task(tasks[i][k][j]);
      }
    }
  }
```

gemm | Kernel – Poll Completion

```
  // poll completion
  for(i : {0, 1, 2, 3}){
    for(k : {0, 1, 2, 3}){
      for(j : {0, 1, 2, 3}){
        while(!tasks[i][k][j].is_complete()){
          usleep(1000);
        }
      }
    }
  }
}
```

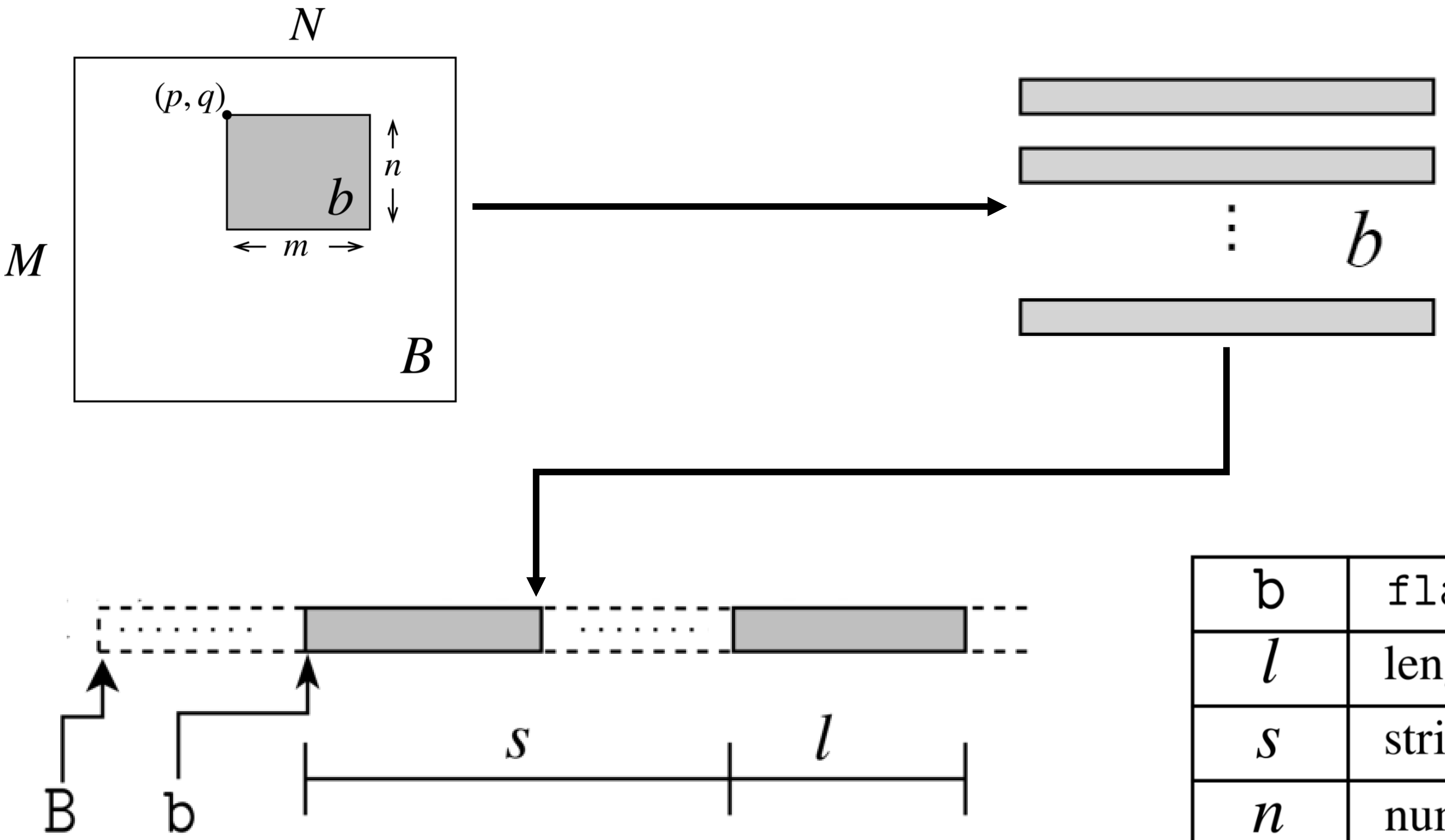
gemm | Task – Input/Output

```
class GemmTask : public Task{
  GemmTask(flash_ptr<float> a,
           flash_ptr<float> b,
           flash_ptr<float> c, args...){
    // declare read-only inputs
    this->add_read(a);
    this->add_read(b);

    // declare read-write inputs
    this->add_read(c);
    this->add_write(c);
  }
```

gemm | Task – Computation

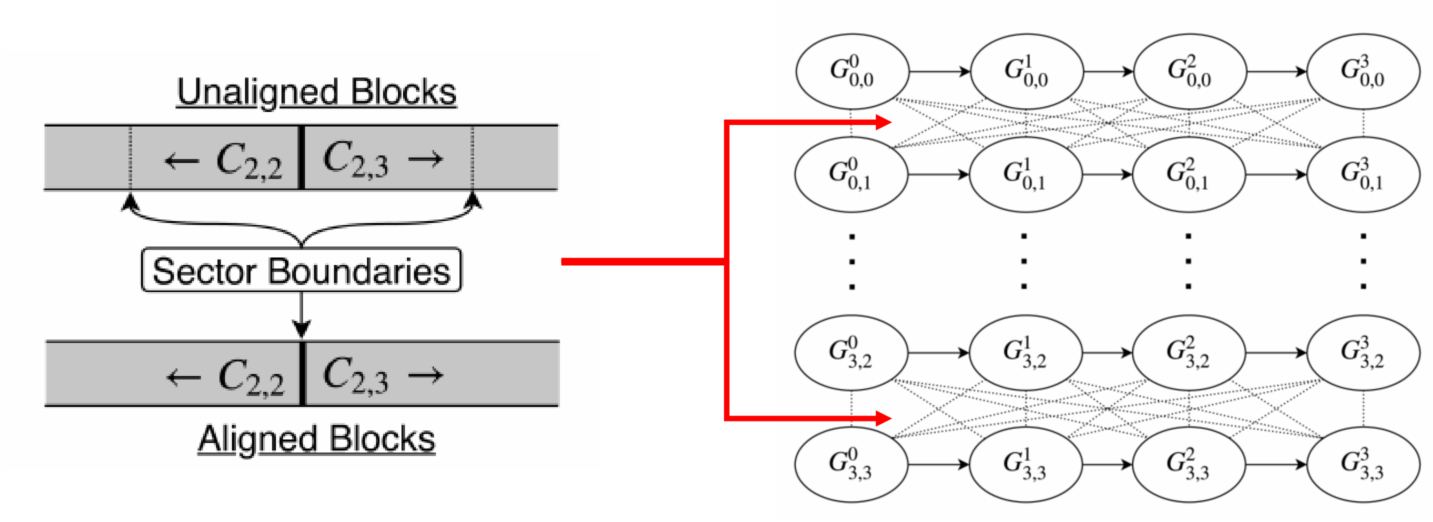
```
  void execute(){
    // get in-memory bufferes
    float* a_ptr = this->buffers[a];
    float* b_ptr = this->buffers[b];
    float* c_ptr = this->buffers[c];

    // execute in-memory computation
    mkl_sgemm(a_ptr, b_ptr, c_ptr, args...);
  }
}
```

Access Specifier | Block Definition



b	`flash_ptr<T>`
l	length per stride
s	stride length
n	number of strides

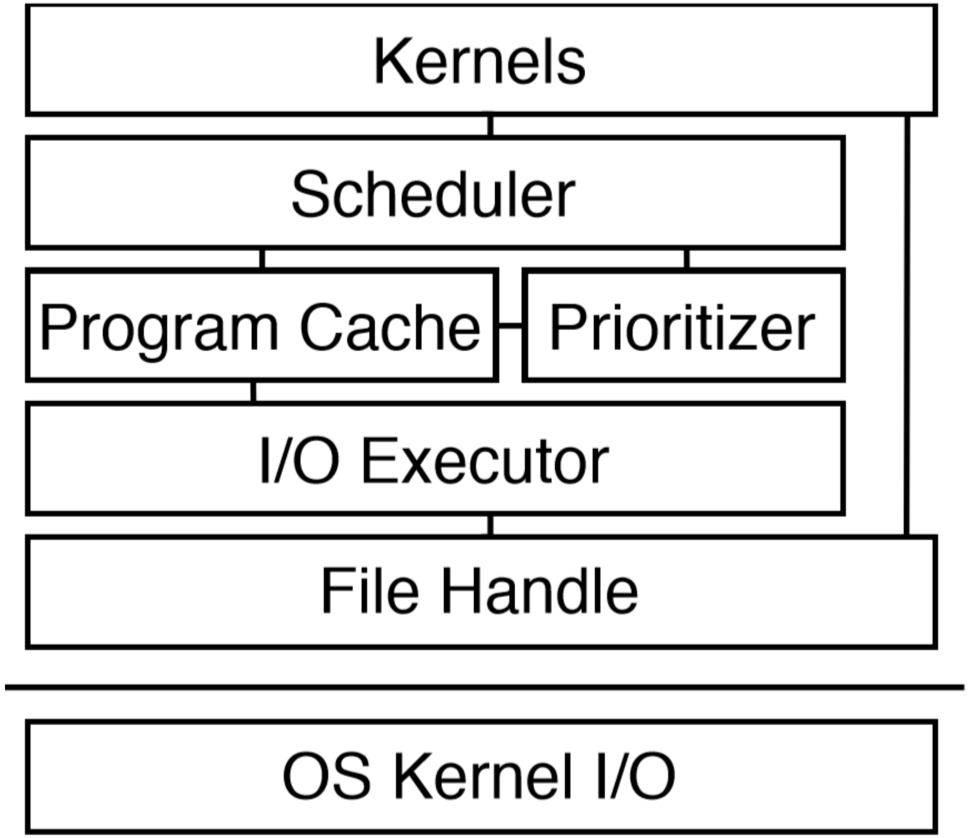
Sector Alignment | Correct vs fast



- Sector-level sharing between adjacent unaligned blocks
- *Conflicting* writes detected and ordered automatically
- Aligned operations extract highest performance

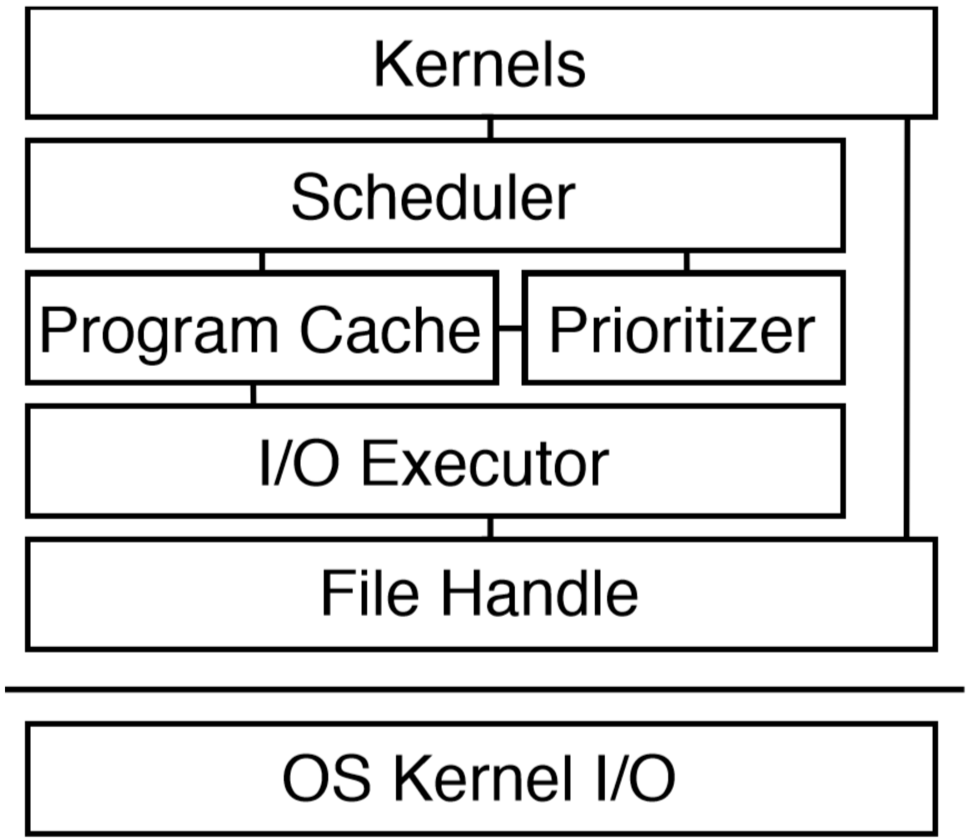
Software Stack | Architecture

- Kernel
- <u>Scheduler</u>
  - Schedule I/O + compute
  - Tunable inter-task parallelism
  - DAG state management



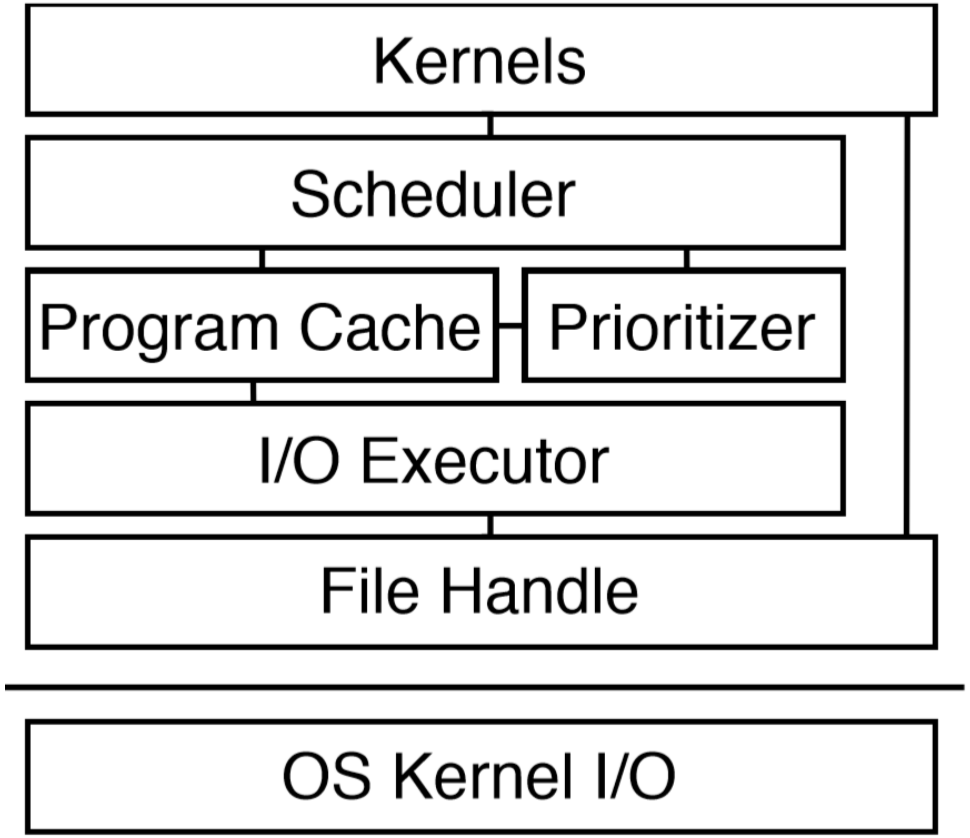
Software Stack | Architecture

- Kernel
- Scheduler
- <u>Prioritizer</u>
  - Prioritize data reuse
  - **<u>Heuristic</u>:** min # of bytes to prefetch



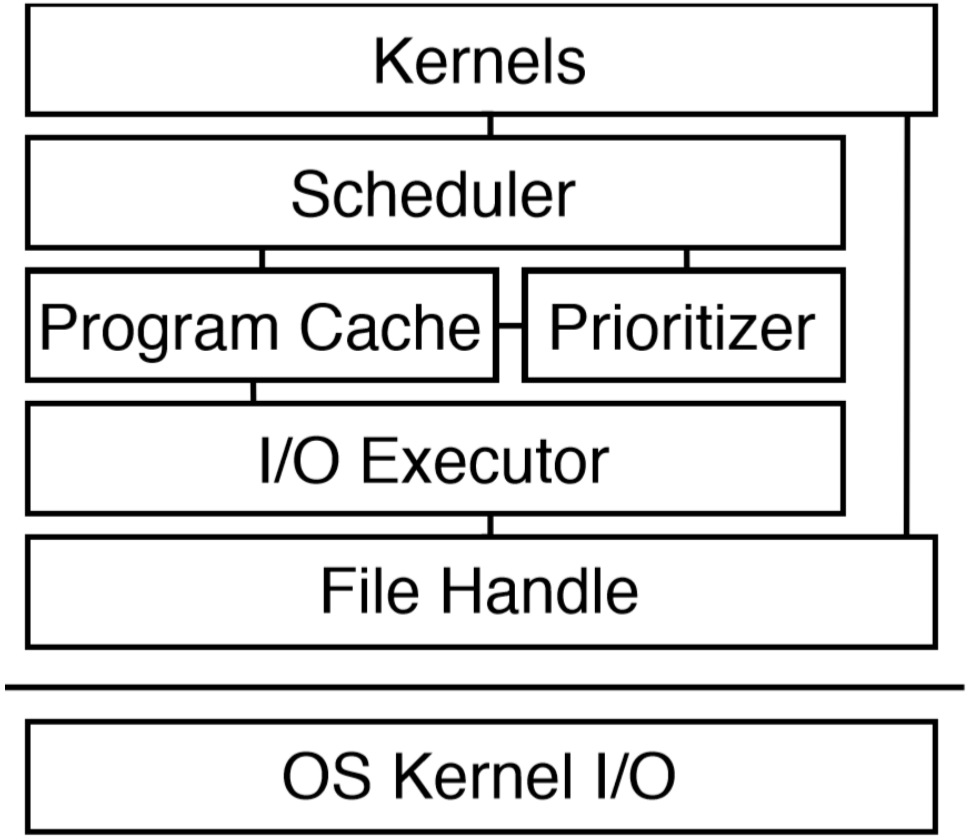
Software Stack | Architecture II

- <u>Program Cache</u>
  - `(flash_ptr<T>,AS)` $\longrightarrow$ `T*`
  - Uniqueness in DRAM contents
  - Data-reuse
  - Hit/miss queries



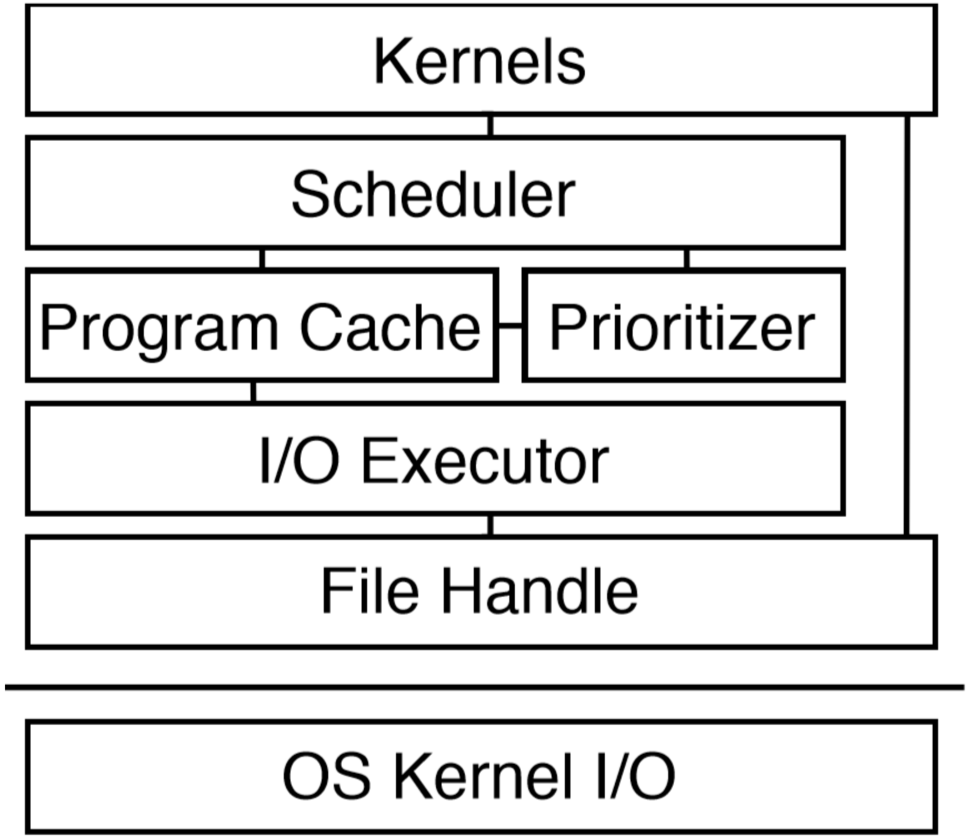
Software Stack | Architecture II

- Program Cache
- <u>I/O Executor</u>
  - Thread-pool + blocking I/O
  - Order *conflicting* writes



Software Stack | Architecture II

- Program Cache
- I/O Executor
- <u>File Handle</u>
  - Concurrent strided I/O requests
  - Linux kernel AIO + `libaio`



Evaluation | Hardware Specifications

Class	Name	Processor(s)	Cores	RAM	Disk	Read BW	Write BW
Workstation	Z840	E5-2620v4 x2	16	32GB	2x 960EVO 1TB	3GB/s	2.2GB/s
Virtual Machines (VM)	M64	E7-8890v3 x2	32	1792GB	SATA SSD	250MB/s	250MB/s
	L32s	E5-2698Bv3 x2	32	256GB	6TB vSSD	1.4GB/s	1.4GB/s
Bare-Metal Server	Sandbox	Gold 6140 x2	36	512GB	3.2TB PM1725a	4GB/s	1GB/s
Spark Cluster [x40]	DS14v2	E5-2673v3 x2	16	112GB	SATA SSD	250MB/s	250MB/s

Evaluation | Datasets

- Sparse Matrices from bag-of-words representation
  - Rows ⇔ Words
  - Columns ⇔ Documents
  - Value ⇔ Frequency
- Datasets used:

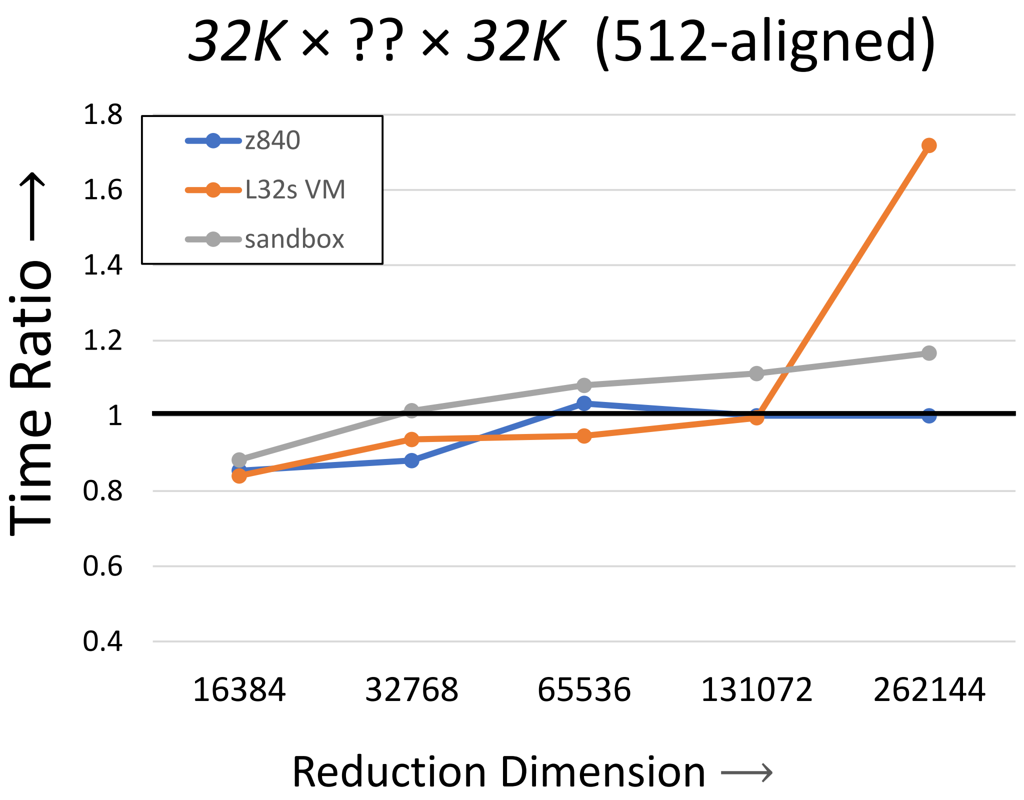
Name	# cols	# rows	NNZs	Tokens	File size (CSR)
Small (Pubmed)	8.15M	140K	428M	650M	10.3GB
Medium (Bing)	22M	1.56M	6.3B	15B	151GB
Large (Bing)	81.7M	2.27M	22.2B	65B	533GB

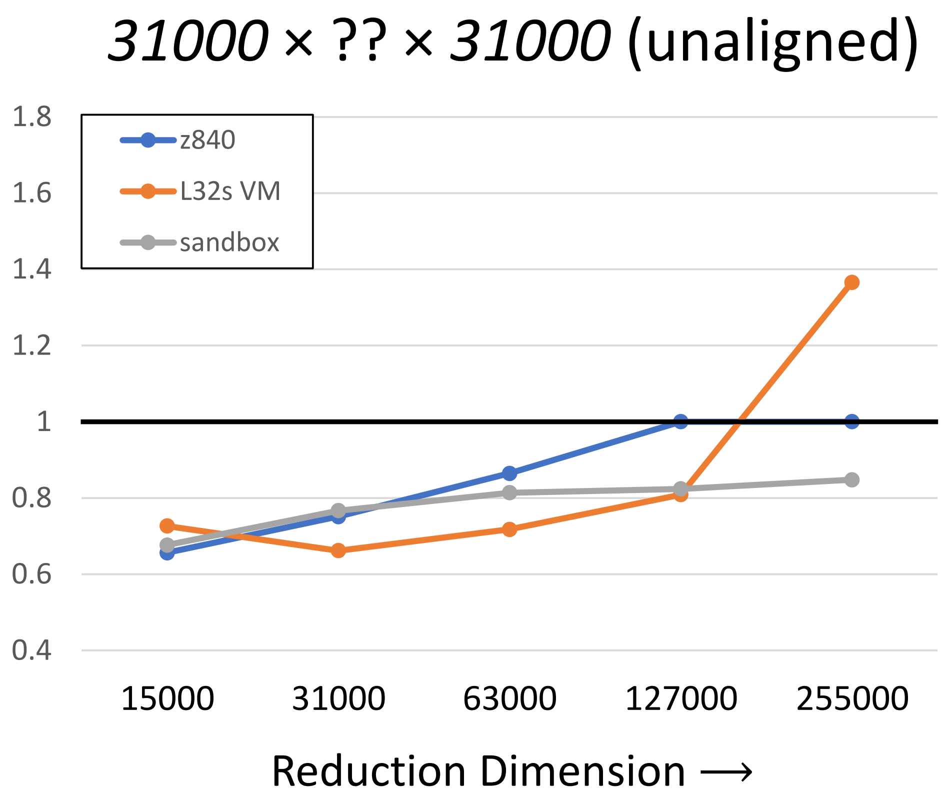
- Context: Parameter servers with dozens of nodes process 100--200B tokens

Evaluation | Metrics

- <u>Time</u> – Absolute time to completion
- <u>Memory</u> – Maximum DRAM usage
- <u>Time ratio</u>
  - In-memory : Flash
  - 0.25 ⇒ Flash version is <u>0.25x as fast</u> as In-memory
- <u>Memory ratio</u>
  - Flash : In-memory
  - 0.5 ⇒ Flash version needs <u>0.5x</u> as much as In-memory's DRAM

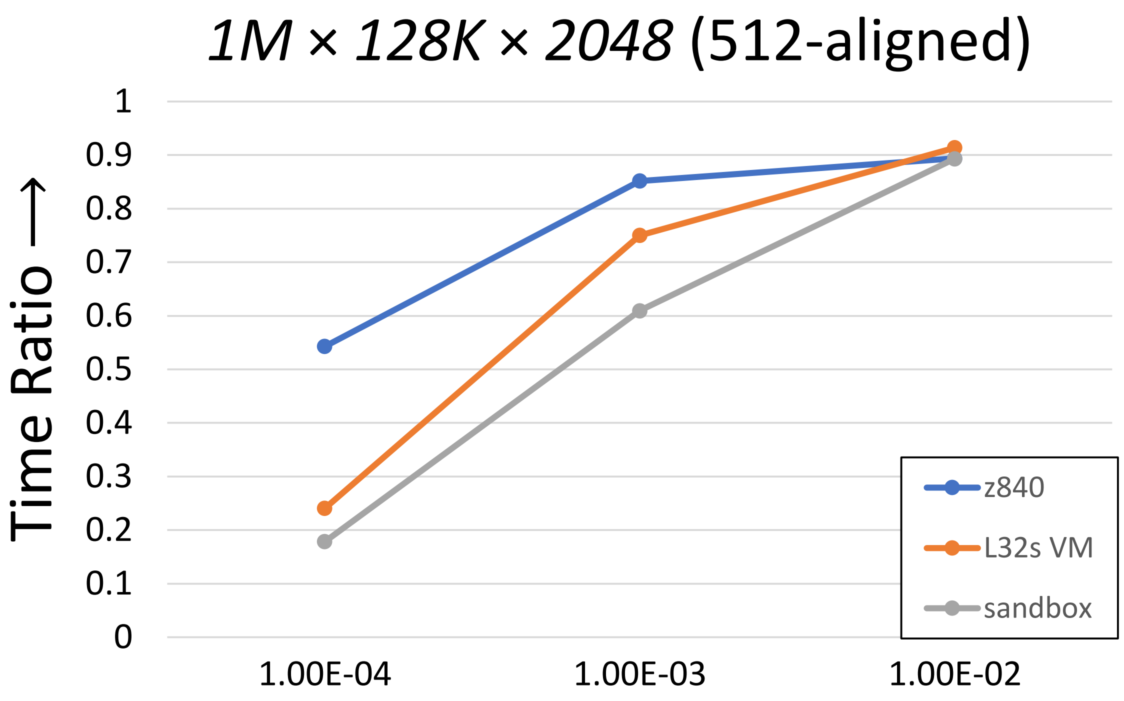
gemm | 8GB RAM is all you need ?





- Larger inner dimension ⟶ Longer accumulate chains ⟶ Lower disk pressure

`csrmm` | Sparsity ruins the party

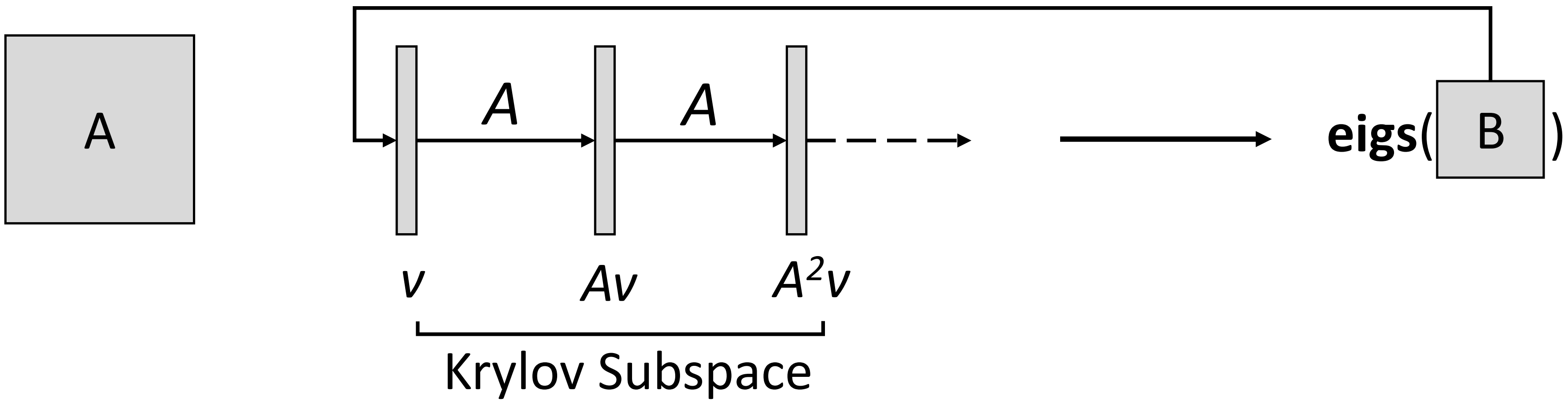


- Dimensionality reduction, projection operations (e.g. PCA)
- <u>No reuse</u>

- Compute:Communication ∽ Sparsity
- Max out disk bandwidth (read + write)

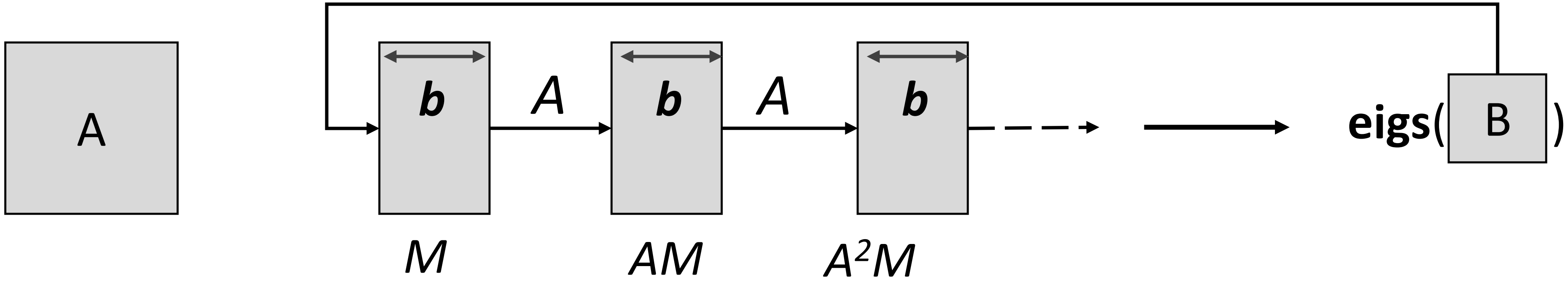
SVD | Choosing the right algorithm

- SVD using symmetric *eigensolvers*
- Lanczos Algorithm
  - ARPACK, Spark MLLib
  - $\approx 2k$ matrix-vector (`gemv`) calls for k eigenvalues
  - Streaming matrix from SSD $\Longrightarrow$ bad performance
  - DRAM bandwidth $\approx$ 30x Flash bandwidth

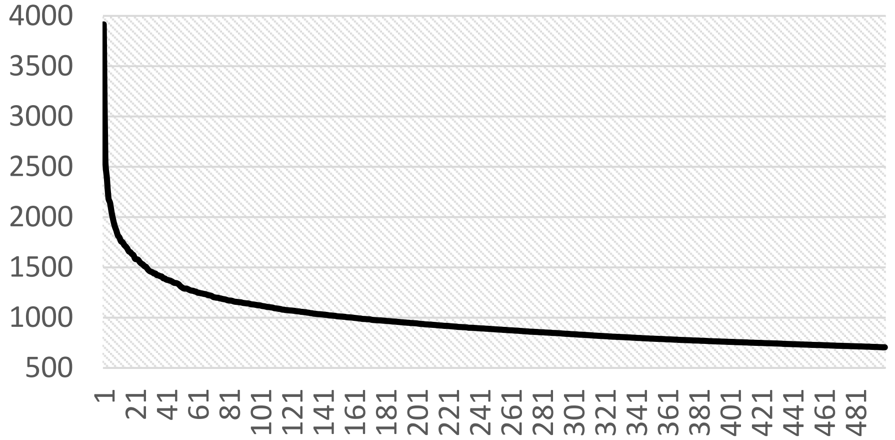


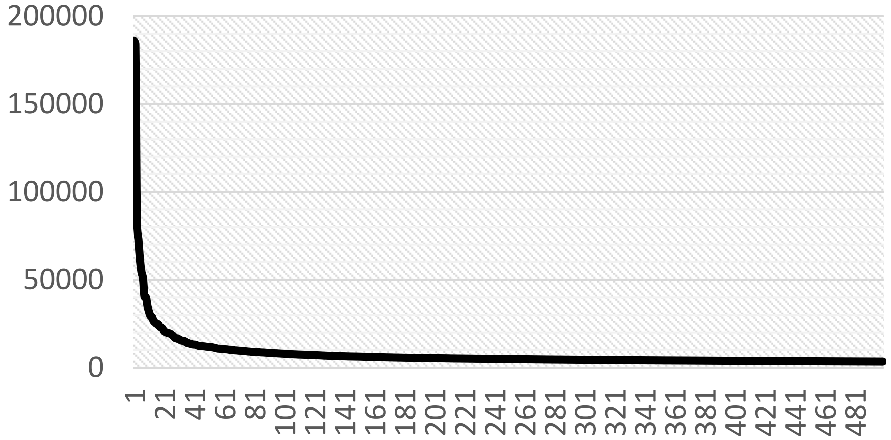
SVD | Choosing the right algorithm

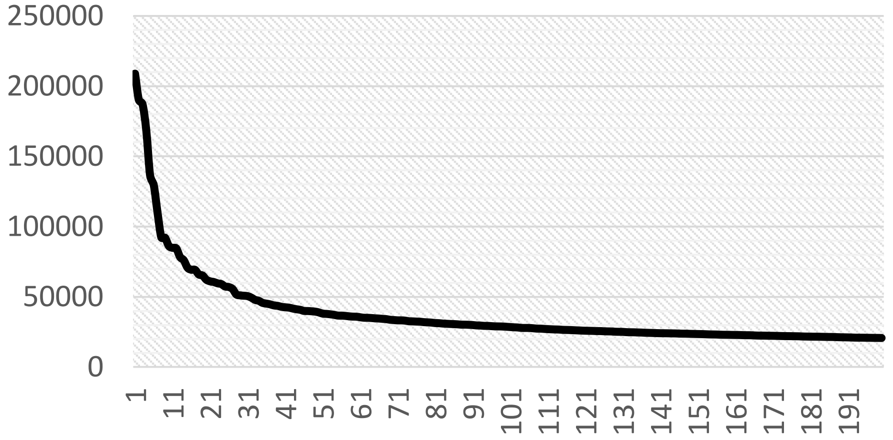
- SVD using symmetric *eigensolvers*
- Lanczos Algorithm
- Block Krylov-Schur Algorithm [Zhou, Saad, 2008]
  - Use $\approx \frac{2k}{b}$ matrix-matrix (`gemm`) calls for k eigenvalues
  - b-fold reduction in number of matrix access
  - Eigenvalues need to be well separated to get speedups



Eigenvalues| Text datasets



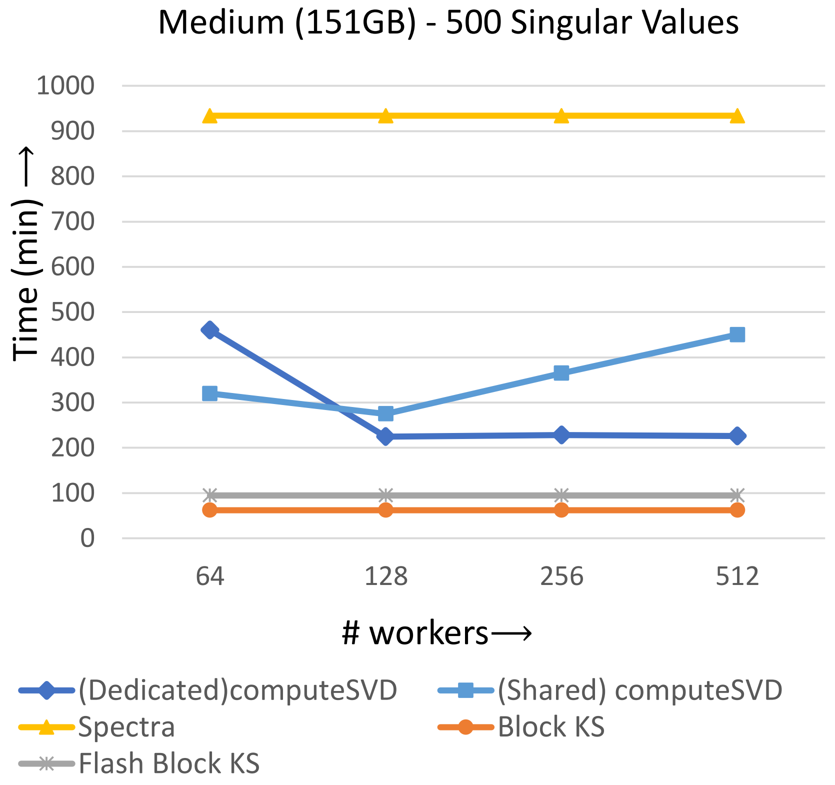




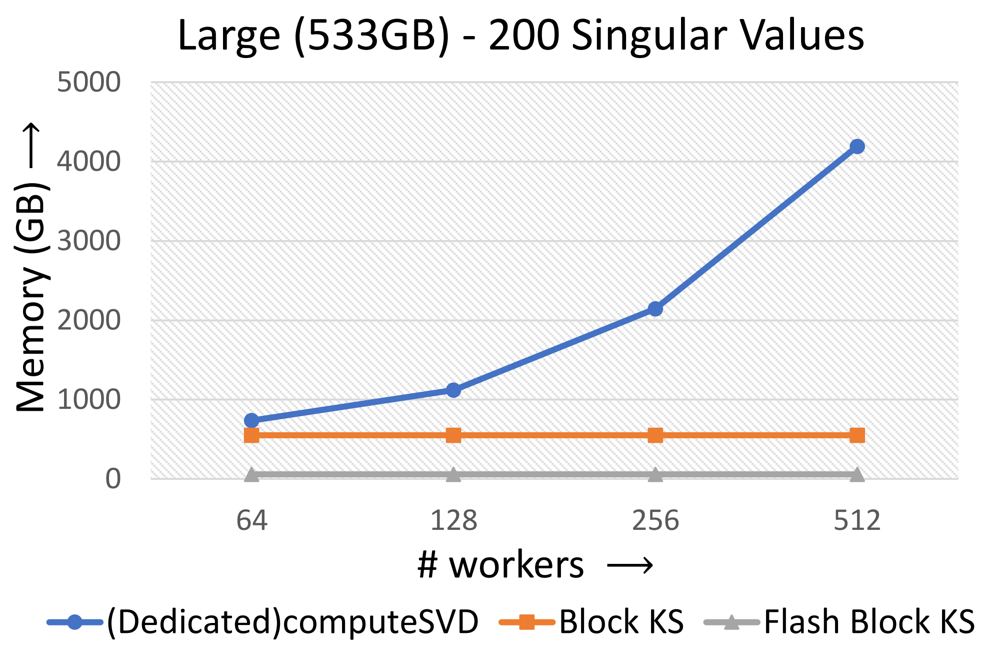
- Spectrum for text data tapers off
  - $a_i \approx \frac{1}{i^\gamma}$ for some $\gamma > 1$
- Gap between successive eigenvalues large enough for block methods

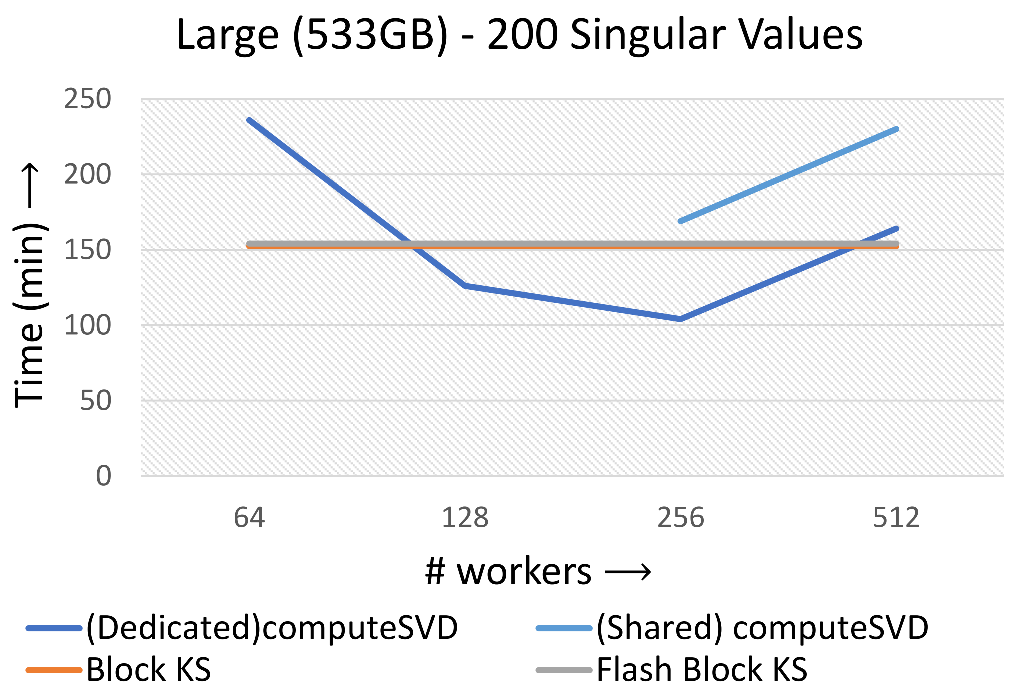
Eigensolvers | Comparison

- Solve for top-K largest eigenvalues
- Spectra (Lanczos, Eigen + MKL)
- Spark MLlib `computeSVD`
  - Shared + dedicated mode
  - 500 singular values hardcoded limit
  - OOM on driver node (>200 singular values)
- Block Krylov-Schur (Block KS)
  - 5000 singular values on Large dataset with 64GB DRAM in under a day



Eigensolvers | Cost-effectiveness



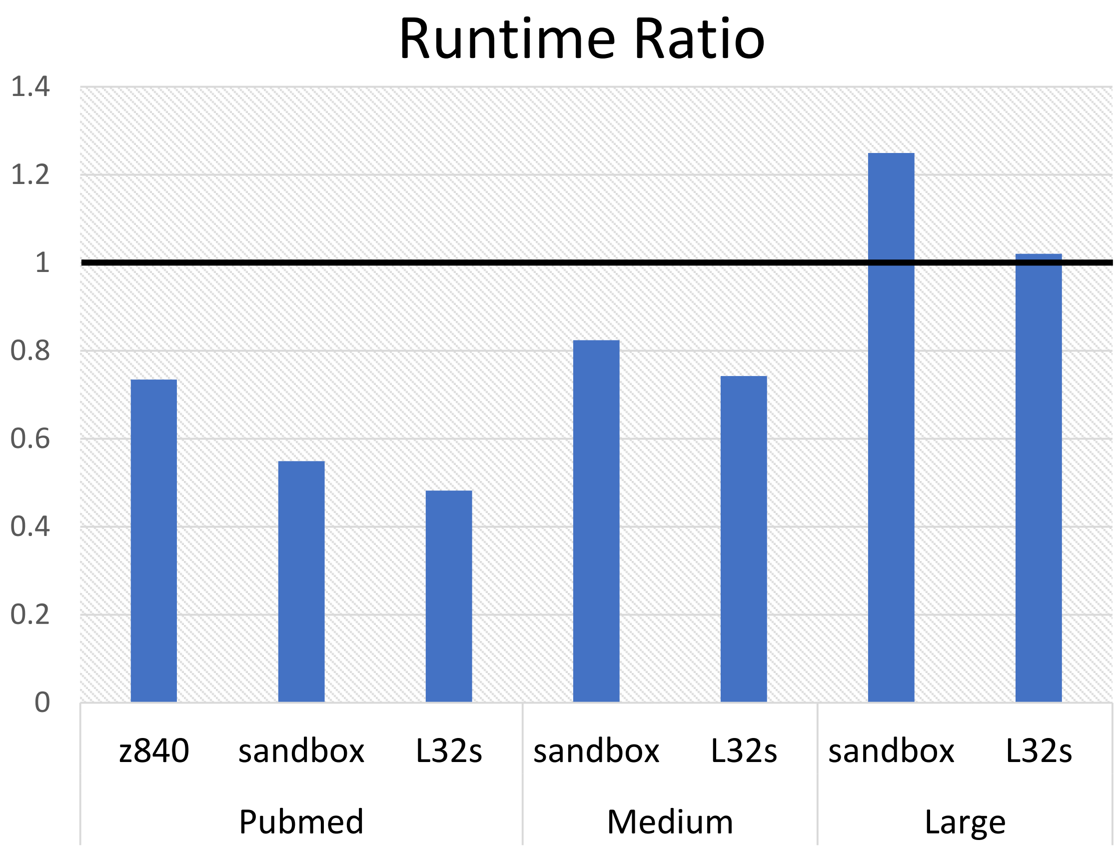


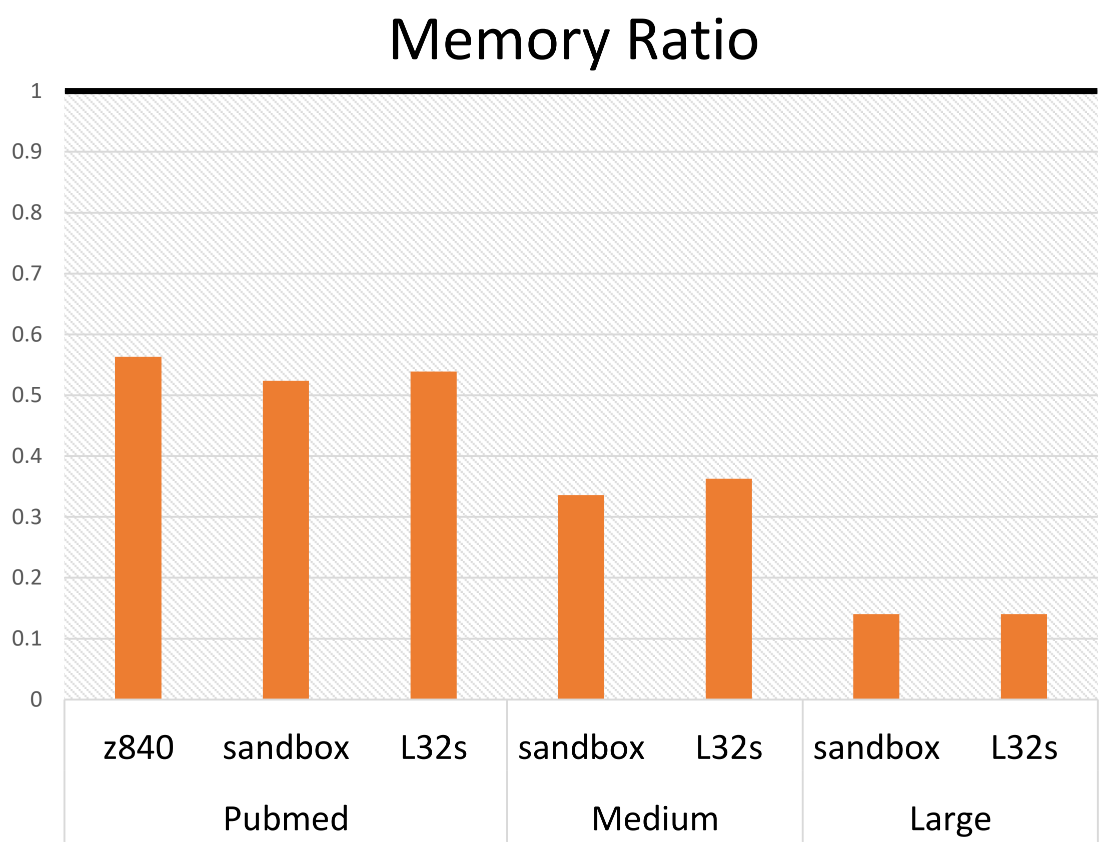
- Single node vs Distributed solvers
  - **16x fewer processing cores, >73x reduction in DRAM requirement**
  - ≈70% performance of best distributed solver runtime
  - Orders of magnitude better hardware utilization, orders of magnitude cheaper

ISLE | Web-Scale Topic Modeling

- Current
  - 2000-topic model, 533GB input → >1TB DRAM
- Goal
  - 5000+-topic model, >533GB input
  - 128GB RAM machines in production
- Expensive steps – SVD, Clustering
- ISLE + BLAS-on-Flash
  - Flash Block KS for SVD
  - Flash k-means for clustering
  - Custom kernels for other operations

ISLE | Larger models, lower costs ?

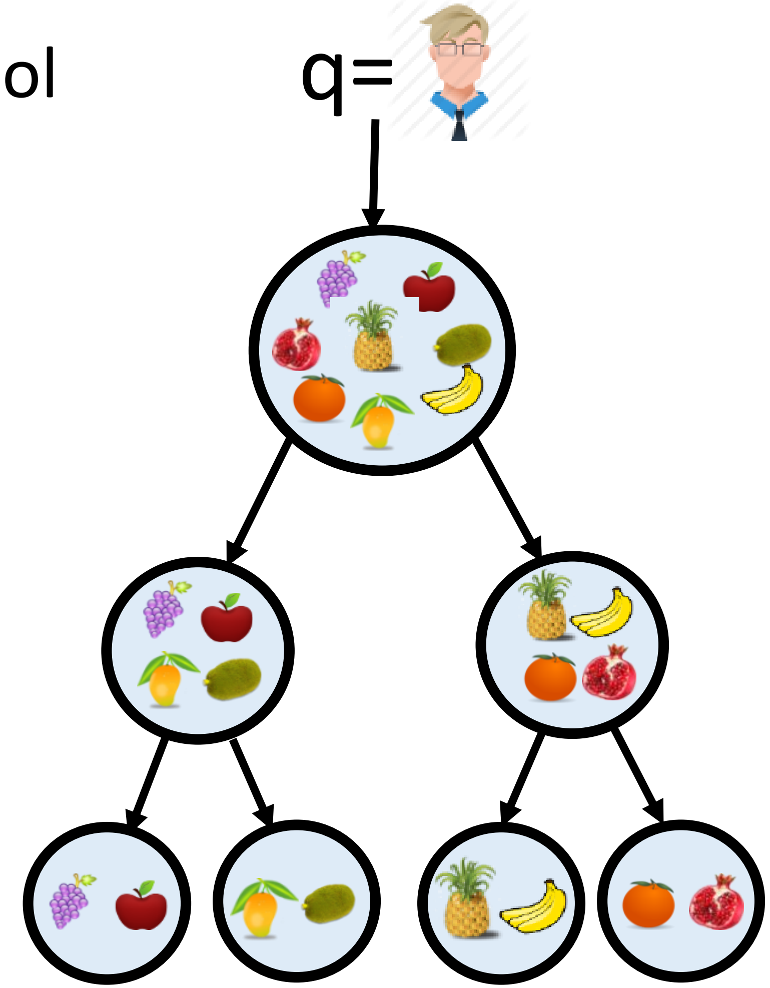




- In-memory baselines for Large dataset are run on M64 due to high DRAM requirement
- **>7x reduction in memory usage** with no overheads on large datasets

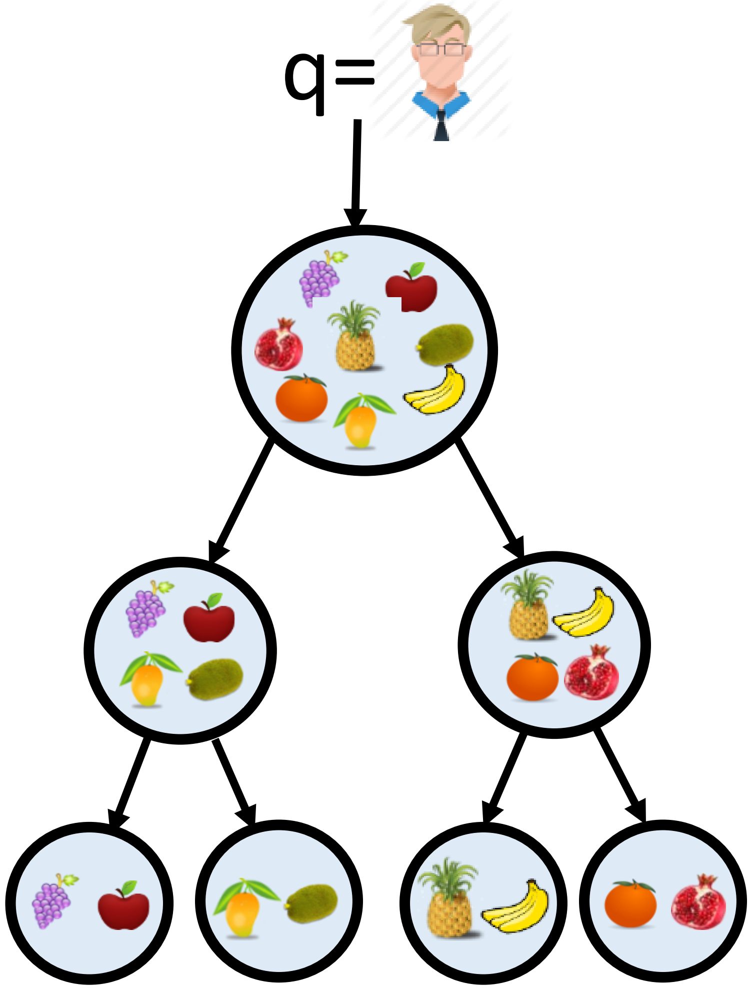
XML | Web-scale classification

- Assign a <u>subset of labels</u> to query point from pool of <u>millions of labels</u>
- Decision-tree like approaches for 100M+ labels
- Bing Related Search + Recommendations
- PfastreXML
  - Depth-First Search traversal
  - Large ensemble of fast and inaccurate trees
- Parabel
  - Breadth-First Beam-Search traversal
  - Small ensemble of slow and accurate trees



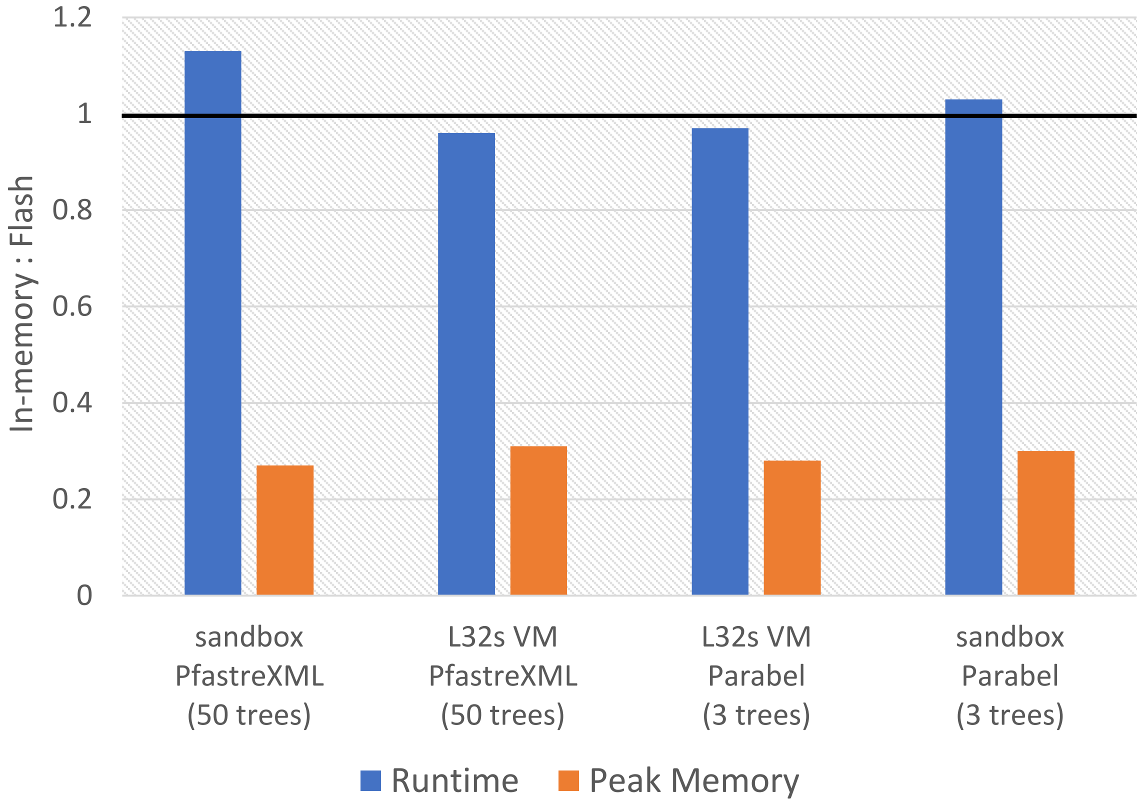
XML | Web-scale classification

- Weekly inference, infrequent training
  - 250M points inference (≈500GB) against 14GB trees
  - Runs on a cluster of DS14 (112GB) nodes
- Why BLAS-on-Flash?
  - 150GB models exist, unable to run on DS14
  - >250GB models foreseeable
- In-memory baseline
  - Improved existing multi-threaded in-memory code
  - 6x faster than current production code



XML | Algorithms + Evaluation

Algorithm	PfastreXML	Parabel
Tree Type	Unbalanced Binary Trees	Balanced Binary Trees
Traversal	Depth First Search (DFS)	10-wide Beam Search (BFS)
# trees	50	3
Time	440 hours	900 hours



- Inference running out of flash uses less DRAM without performance regressions
- Inference on larger models ⇒ Better quality predictions

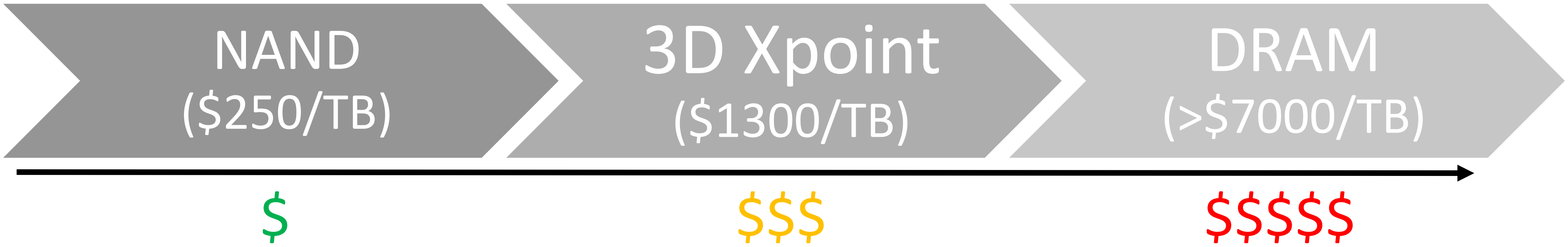
In the works

- Decision Trees training (LightGBM)
  - Train gradient-boosted decision trees on TBs of data
  - Out-of-core training for better models at low-cost
- k-Approximate Nearest Neighbor (k-ANN) Search
  - Serve queries on 100B+ points in few ms each
  - DRAM limitations $\Longrightarrow$ partition dataset, mirror + aggregate response
  - Use disk-resident indexes to increase points-per-node

Conclusion

We have developed set of math routines utilizing a DAG execution engine for large SSD-resident data

- Near in-memory performance
- Drastic reduction in memory usage ⟹ larger inputs possible
- Relevant for Optane/NVDIMMs, GraphCore



github.com/Microsoft/BLAS-on-flash