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

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 $\implies$ 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

- 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 possibly slow
gemm | Task View

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

\[
\text{GEMM}(X, Y) := X \cdot Y
\]

\[
C_{1,2} = \text{GEMM}(A_{1,0}, B_{0,2}) + \text{GEMM}(A_{1,1}, B_{1,2}) + \text{GEMM}(A_{1,2}, B_{2,2}) + \text{GEMM}(A_{1,3}, B_{3,2})
\]
gemm | Chain View

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

\[ G_{1,2} := \text{GEMM}(A_{1,0}, B_{0,2}) \]
\[ G_{1,2}^{0} \]
\[ \rightarrow \text{GEMM}(A_{1,1}, B_{1,2}) \]
\[ G_{1,2}^{1} \]
\[ \rightarrow \text{GEMM}(A_{1,2}, B_{2,2}) \]
\[ G_{1,2}^{2} \]
\[ \rightarrow \text{GEMM}(A_{1,3}, B_{3,2}) \]
\[ G_{1,2}^{3} \]
gemm | DAG view

Accumulate Chains
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_ik * B_jk
                tasks[i][k][j] = GemmTask(A_ik, B_jk, C_ik, args...);
            }
        }
    }
}
// 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]);
        }
    }
}
/ **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]);
        }
    }
}
// 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);
            }
        }
    }
}

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);
  }
}
`void execute(){
   // get in-memory buffers
   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

<table>
<thead>
<tr>
<th>b</th>
<th>flash_ptr &lt; T &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>length per stride</td>
</tr>
<tr>
<td>s</td>
<td>stride length</td>
</tr>
<tr>
<td>n</td>
<td>number of strides</td>
</tr>
</tbody>
</table>
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**
- **Scheduler**
  - Schedule I/O + compute
  - Tunable inter-task parallelism
  - DAG state management
Software Stack | Architecture

• Kernel
• Scheduler
• Prioritizer
  • Prioritize data reuse
  • **Heuristic**: min # of bytes to prefetch
Software Stack | Architecture II

- **Program Cache**
  - \((\text{flash\_ptr}<T>, \text{AS}) \rightarrow T^*\)
  - Uniqueness in DRAM contents
  - Data-reuse
  - Hit/miss queries
Software Stack | Architecture II

- Program Cache
- I/O Executor
  - Thread-pool + blocking I/O
  - Order conflicting writes
Software Stack | Architecture II

- Program Cache
- I/O Executor
- File Handle
  - Concurrent strided I/O requests
  - Linux kernel AIO + libaio
## Evaluation | Hardware Specifications

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

• Sparse Matrices from bag-of-words representation
  • Rows ⇔ Words
  • Columns ⇔ Documents
  • Value ⇔ Frequency

• Datasets used:

<table>
<thead>
<tr>
<th>Name</th>
<th># cols</th>
<th># rows</th>
<th>NNZs</th>
<th>Tokens</th>
<th>File size (CSR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small (Pubmed)</td>
<td>8.15M</td>
<td>140K</td>
<td>428M</td>
<td>650M</td>
<td>10.3GB</td>
</tr>
<tr>
<td>Medium (Bing)</td>
<td>22M</td>
<td>1.56M</td>
<td>6.3B</td>
<td>15B</td>
<td>151GB</td>
</tr>
<tr>
<td>Large (Bing)</td>
<td>81.7M</td>
<td>2.27M</td>
<td>22.2B</td>
<td>65B</td>
<td>533GB</td>
</tr>
</tbody>
</table>

• Context: Parameter servers with dozens of nodes process 100--200B tokens
Evaluation | Metrics

- **Time** – Absolute time to completion
- **Memory** – Maximum DRAM usage

**Time ratio**
- In-memory : Flash
  - 0.25 → Flash version is 0.25x as fast as In-memory

**Memory ratio**
- Flash : In-memory
  - 0.5 → Flash version needs 0.5x as much as In-memory’s DRAM
gemm | 8GB RAM is all you need?

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

### Graphs

**32K × ?? × 32K (512-aligned)**
- Time Ratio vs. Reduction Dimension
- Lines for z840, L32s VM, and sandbox

**31000 × ?? × 31000 (unaligned)**
- Time Ratio vs. Reduction Dimension
- Lines for z840, L32s VM, and sandbox
Sparsity ruins the party

- Dimensionality reduction, projection operations (e.g. PCA)
- No reuse

- Compute: Communication \sim Sparsity
- Max out disk bandwidth (read + write)
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 \( \implies \text{bad performance} \)
  • DRAM bandwidth \( \approx 30x \) Flash bandwidth

\[
\begin{align*}
A &\quad A &\quad A &\quad \cdots &\quad \text{eigs} (B) \\
&\quad v &\quad Av &\quad A^2v &\quad \text{Krylov Subspace}
\end{align*}
\]
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

\[ A \rightarrow b \rightarrow A \rightarrow b \rightarrow eigs(B) \]
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
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 **subset of labels** to query point from pool of **millions of labels**
- 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**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>PfastreXML</th>
<th>Parabel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree Type</td>
<td>Unbalanced</td>
<td>Balanced</td>
</tr>
<tr>
<td></td>
<td>Binary Trees</td>
<td>Binary Trees</td>
</tr>
<tr>
<td>Traversal</td>
<td>Depth First</td>
<td>10-wide Beam</td>
</tr>
<tr>
<td></td>
<td>Search (DFS)</td>
<td>Search (BFS)</td>
</tr>
<tr>
<td># trees</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>Time</td>
<td>440 hours</td>
<td>900 hours</td>
</tr>
</tbody>
</table>

• 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 \(\Rightarrow\) 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 \(\implies\) larger inputs possible

• Relevant for Optane/NVDIMMs, GraphCore

[Cost comparison image]

github.com/Microsoft/BLAS-on-flash