High-Performance Dense Tucker Decomposition on a GPU Cluster

Jee Whan Choi, Xing Liu*, and Venkatesan Chakaravarthy

IBM Research
SC’18, November 2018

*Xing Liu is currently at Intel Research
Tucker is a multi-linear decomposition method

\[ X \approx GU \]

\[ U^{(1)} \]

\[ U^{(2)} \]

\[ U^{(3)} \]
Many operations are done after converting a tensor into a matrix (*matricization*)
Vectors inside the tensor are called **fibers**
Lay out the fibers as column vectors of a matrix

Mode-1 matricization
Lay out the fibers as column vectors of a matrix

\[ \prod_{i \neq n} I_i \]

Mode-\( n \) matricization
Dense Tucker decomposition time is dominated by dense matrix operations

1. **Data:** $X, \{R_n\}$
2. **Result:** $\mathcal{G}, \{U^{(n)}\}$
3. $y \leftarrow X$
4. **for** $n = 1, \ldots, N$ **do**
   5. $S \leftarrow Y^{(n)}(n)Y^{(n)T}$;  // Gram matrix construction
   6. $U^{(n)} \leftarrow R_n$ leading eigenvectors of $S$;  // Eigensolve
   7. $y \leftarrow y \times_n U^{(n)T}$;  // TTM
5. **end**
6. $\mathcal{G} \leftarrow y$;

**Sequentially Truncated Higher-Order Singular Value Decomposition (STHOSVD)**
Dense Tucker decomposition time is dominated by dense matrix operations.

Sequentially Truncated Higher-Order Singular Value Decomposition (STHOSVD)

1. **Data:** $X$, $\{R_n\}$
2. **Result:** $G$, $\{U^{(n)}\}$
3. $y \leftarrow X$
4. **for** $n = 1, \ldots, N$ **do**
5. $S \leftarrow Y^{(n)}Y^{T}_{(n)}$; // Gram matrix construction
6. $U^{(n)} \leftarrow R_n$ leading eigenvectors of $S$; // Eigensolve
7. $y \leftarrow y \times_n U^{(n)}T$; // TTM
8. $G \leftarrow y$;
Other algorithms use similar operations – also dominated by dense matrix operations

\begin{itemize}
  \item Data: \( \mathcal{X}, \{R_n\} \)
  \item Result: \( \mathcal{G}, \{U^{(n)}\} \)
  \item initialize \( \{U^{(n)}\} \) using HOSVD;
  \item repeat
    \begin{itemize}
      \item for \( n = 1, \ldots, N \) do
        \begin{itemize}
          \item \( y \leftarrow \mathcal{X} \times_1 U^{(1)T} \times_{n-1} U^{(n-1)T} \times_{n+1} U^{(n+1)T} \times_n U^{(N)T} \) // TTM sequence
          \item \( U^{(n)} \leftarrow R_n \) leading eigenvectors of \( Y^{(n)} \)
          \item // Eigensolve
        \end{itemize}
    \end{itemize}
  \item until fit ceases to improve;
  \item \( \mathcal{G} \leftarrow \mathcal{X} \times_1 U^{(1)T} \times_N U^{(N)T}; \)
\end{itemize}

Higher-Order Orthogonal Iteration (HOOI)
Just offload it to GPUs, get $1000\times$ speedup and be done.

- Offload it to GPUs
- Get $1000\times$ speedup
- Live happily ever after
Thank you

Q & A

jwchoi@us.ibm.com
Life is not quite that simple...

- Offload it to GPUs
- Get 1000× speedup
- Live happily ever after
There are three key issues with using GPUs for Tucker

- **Traditional partitioning method is ill-suited for GPUs**
  - Multi-dimensional (N-D) partitioning leads to lower GPU utilization & performance

- **Redundant communication (kind of)**
  - Fixing the above issue changes the when and where communication occurs, leading to opportunities for optimization

- **Eigendecomposition does not scale**
  - Strong scaling issues
Traditional tensor partitioning method is ill-suited for GPUs

N-D partitioning
N-D partitioning has two key advantages for sparse tensors and CP decomposition

• Strikes a balance between minimizing synchronization (coarse-grained) and load balancing (fine-grained)
• Save memory by partitioning factor matrices across processors (good for CP decomposition).

• For dense data sets, load balancing can be achieved with a simpler (e.g., 1-D) partitioning strategies
• Core tensor takes up more space
N-D partitioning leads to bad performance if # of processors along a mode is large...ish

\[ P_1 \times \ldots \times P_N \text{ processors} \]

\[ (2 \times 2 \times 3 \text{ in this example}) \]

\[ \hat{P}_n = \prod_{i \neq n} P_i \]

\[ \hat{I}_n = \prod_{i \neq n} I_i \]

\[ P_n \text{ matrix multiplications of size} \]

\[ \frac{I_n}{P_n} \times \hat{I}_n \]

\[ \text{all-reduce} \]
Our slice-block partitioning method partitions at the granularity of slices

Each processor has $n_s$ tensor slices $(I_n \times I_{n+1})$

Slice-block partitioning

$n_s = \frac{\prod_{i \neq n, n+1} I_i}{P}$

1 matrix multiplication of size $I_n \times \frac{I_n}{P}$
Our partitioning strategy changes *when* and *where* communication occurs

- **N-D partitioning**
  - Column-wise communication during multiplication
  - All-reduce after multiplication
  - No communication required for re-matricization (only local copies)

- **Slice-block partitioning**
  - No communication during multiplication
  - All-reduce after multiplication
  - Communication required for re-matricization
Slice-block partitioning changes *where* communication occurs

- **N-D – Gram matrix and TTM**
  
  \[(P_n - 1)(\alpha + \beta \frac{I}{P})\]
  
  - Each processor calculates 1 column block of \(S_c\) which requires sending & receiving \(I/P\) elements from \(P_n - 1\) processors (with same j and k indices).
  - All-reduce is ignored, since the gram matrix S is *much* smaller.

- **Slice-block – Matricization**

  \[(P - 1)\alpha + \beta \frac{I}{P}\]

  - Each processor sends & receives \(I/P\) elements across \(P\) processors
  - All-reduce is ignored, since the gram matrix S is *much* smaller.
Typical mode-n tensor matricization “swaps” the $1^{st}$ and the $n^{th}$ mode
  - Mode ordering: $n, 2, ..., (n-1), 1, (n+1), ..., N$

What if we “rotate” the modes instead?
  - Mode ordering: $n, (n+1), ..., N, 1, 2, ..., (n-1)$

Added benefit of slice-block partitioning - we can reduce the communication even further
Rotation allows transpose to replace matricization

Explicit transpose is unnecessary if we use the “transpose matrix” option found in most libraries

Communication is now only required for every other mode
We see good speedup with GPUs. Are we done?

- After the all-reduce to calculate the Gram matrix, eigendecomposition was done (redundantly) on every node
- Why?
  - With a CPU cluster, eigendecomposition time was negligible
  - Distributed eigendecomposition on small matrices (Gram) scales poorly
Distributed eigendecomposition (*pdsyevx*) does not scale on small matrices
Use a randomized SVD algorithm instead

1. **Data:** \( A \in \mathbb{R}^{m \times n} \); number of left singular vectors required, \( r \)
2. **Result:** left singular vectors \( \Upsilon \)
3. \( O \leftarrow \text{Gaussian random matrix} \in \mathbb{R}^{n \times 2r} \)
4. \( T \leftarrow AO \)
5. \([Q \ R] \leftarrow QR(T); \quad \text{// Find orthonormal basis for range of } T,\)
6. \( B \leftarrow Q^T A; \quad \quad \quad \text{\quad \quad \quad \quad \quad // such that } A \approx QQ^T A = QB\)
7. \( G \leftarrow BB^T; \quad \quad \quad \text{\quad \quad \quad \quad \quad // Gram matrix construction}\)
8. \([\hat{\Upsilon} \ \hat{\Sigma} \ \hat{\Psi}] \leftarrow \text{SVD}(G);\)
9. \( \Upsilon \leftarrow Q \hat{\Upsilon}; \)
Test platform – 64 node POWER8 system /w 4 P100 GPUs
Our evaluation was done on **five input tensors** and **four implementation variations**.

<table>
<thead>
<tr>
<th>Name</th>
<th>Modes</th>
<th>Dimension</th>
<th>Rank</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D3000-R300</td>
<td>3</td>
<td>3000</td>
<td>300</td>
<td>216 GB</td>
</tr>
<tr>
<td>3D2000-R1000</td>
<td>3</td>
<td>2000</td>
<td>1000</td>
<td>64 GB</td>
</tr>
<tr>
<td>4D200-R20</td>
<td>4</td>
<td>200</td>
<td>20</td>
<td>13 GB</td>
</tr>
<tr>
<td>4D400-R64</td>
<td>4</td>
<td>400</td>
<td>64</td>
<td>205 GB</td>
</tr>
<tr>
<td>5D128-R16</td>
<td>5</td>
<td>128</td>
<td>16</td>
<td>275 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Re-use</th>
<th>Eigenvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NoReuse + DSYEVX</td>
<td>No</td>
<td>DSYEVX</td>
</tr>
<tr>
<td>2</td>
<td>Reuse + DSYEVX</td>
<td>Yes</td>
<td>DSYEVX</td>
</tr>
<tr>
<td>3</td>
<td>Reuse + PDSYEVX</td>
<td>Yes</td>
<td>PDSYEVX</td>
</tr>
<tr>
<td>4</td>
<td>Reuse + RndSVD</td>
<td>Yes</td>
<td>Randomized SVD</td>
</tr>
</tbody>
</table>
On tensors with large dimension size, we saw good scaling across all four GPUs, but not on small dimension size.

<table>
<thead>
<tr>
<th>Function</th>
<th>Performance (GFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1× GPU</td>
</tr>
<tr>
<td></td>
<td>3D3000-R300</td>
</tr>
<tr>
<td>Gram_0</td>
<td>5008</td>
</tr>
<tr>
<td>TTM_0</td>
<td>3914</td>
</tr>
<tr>
<td></td>
<td>4D400-R64</td>
</tr>
<tr>
<td>Gram_0</td>
<td>2685</td>
</tr>
<tr>
<td>TTM_0</td>
<td>895</td>
</tr>
</tbody>
</table>
We see up to 12.5\times speedup on four GPUs over 2 POWER8 CPUs on a 3000 \times 3000 \times 3000 tensor (rank = 300)
Speedup stops at ~4\times on a smaller 400\times400 \times 400 \times 400 tensor (rank = 64)

<table>
<thead>
<tr>
<th>Reuse + DSYEVD</th>
<th>GPU x 1</th>
<th>GPU x 2</th>
<th>GPU x 3</th>
<th>GPU x 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time (s)</td>
<td>2.30</td>
<td>3.89</td>
<td>3.98</td>
<td>3.59</td>
</tr>
<tr>
<td>Speedup over CPU</td>
<td>2.30</td>
<td>1.63</td>
<td>1.61</td>
<td>1.19</td>
</tr>
</tbody>
</table>
GPU shows similar accuracy to CPU when decomposition rank is changed.

<table>
<thead>
<tr>
<th>Calculated rank $R_n$</th>
<th>CPU: Reuse + DSYEVX</th>
<th>GPU: Reuse + RndSVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>110</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>210</td>
<td>210</td>
<td>210</td>
</tr>
<tr>
<td>310</td>
<td>310</td>
<td>310</td>
</tr>
<tr>
<td>410</td>
<td>410</td>
<td>410</td>
</tr>
<tr>
<td>510</td>
<td>510</td>
<td>510</td>
</tr>
<tr>
<td>610</td>
<td>610</td>
<td>610</td>
</tr>
<tr>
<td>710</td>
<td>710</td>
<td>710</td>
</tr>
<tr>
<td>810</td>
<td>810</td>
<td>810</td>
</tr>
<tr>
<td>910</td>
<td>910</td>
<td>910</td>
</tr>
<tr>
<td>1010</td>
<td>1010</td>
<td>1010</td>
</tr>
</tbody>
</table>

Data set: 2D2000-R1000
Randomized SVD shows potential performance vs. accuracy trade-off (3D3000-R300)
Randomized SVD shows potential performance vs. accuracy trade-off (4D200-R20)
Parallel efficiency on 64 nodes is better on CPUs

<table>
<thead>
<tr>
<th>Input</th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Parallel Eff.</td>
<td>Speedup</td>
</tr>
<tr>
<td>3D3000-R300</td>
<td>0.17</td>
<td>10.59</td>
</tr>
<tr>
<td>4D200-R20</td>
<td>0.15</td>
<td>9.79</td>
</tr>
<tr>
<td>4D400-R64</td>
<td>0.44</td>
<td>28.08</td>
</tr>
<tr>
<td>5D128-R16</td>
<td>0.72</td>
<td>45.93</td>
</tr>
</tbody>
</table>
Comparison against TuckerMPI (3D3000-R300)

Execution time (s)

Number of nodes

Ours: Reuse + RndSVD
Comparison against TuckerMPI (4D400-R64)

Ours: Reuse + RndSVD
Comparison against TuckerMPI (5D128-R16)

Ours: Reuse + RndSVD
Comparison against TuckerMPI (4D200-R20)

Ours: Reuse + RndSVD