Performance portable parallel sparse CP-APR tensor decompositions

Sparse Days 2018, 28/09/2018

Presented by

Keita Teranishi

Christopher Forster (NVIDIA), Richard Barrett, Daniel Dunlavy, and Tamara Kolda
Develop production quality library software to perform CP factorization for Poisson Regression Problems for HPC platforms

Tensor Tool Box (http://www.tensortoolbox.org)
- Matlab only!

Support several HPC platforms
- Node parallelism (Multicore, Manycore and GPUs)

Major Questions
- Software Design
- Performance Tuning

This talk
- We are interested in two major variants
  - Multiplicative Updates
  - Projected Damped Newton for Row-subproblems
CP Tensor Decomposition

CANDECOMP/PARAFAC (CP) Model
Express the important feature of data using a small number of vector outer products

Model: \[ \mathcal{M} = \sum_r \lambda_r \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r \]

\[ x_{ijk} \approx m_{ijk} = \sum_r \lambda_r a_{ir} b_{jr} c_{kr} \]

Key references: Hitchcock (1927), Harshman (1970), Carroll and Chang (1970)
Poisson for Sparse Count Data

**Gaussian (typical)**

The random variable \( x \) is a continuous real-valued number.

\[
x \sim N(m, \sigma^2)
\]

\[
P(X = x) = \frac{\exp\left(-\frac{(x-m)^2}{2\sigma^2}\right)}{\sqrt{2\pi\sigma^2}}
\]

**Poisson**

The random variable \( x \) is a discrete nonnegative integer.

\[
x \sim \text{Poisson}(m)
\]

\[
P(X = x) = \frac{\exp(-m)m^x}{x!}
\]
Gaussian (typical)

The random variable $x$ is a continuous real-valued number.

$$x \sim N(m, \sigma^2)$$

$$P(X = x) = \frac{\exp\left(-\frac{(x-m)^2}{2\sigma^2}\right)}{\sqrt{2\pi\sigma^2}}$$

Poisson

The random variable $x$ is a discrete nonnegative integer.

$$x \sim \text{Poisson}(m)$$

$$P(X = x) = \frac{\exp(-m)m^x}{x!}$$
Sparse Poisson Tensor Factorization

Model: Poisson distribution (nonnegative factorization)

\[ x_{ijk} \sim \text{Poisson}(m_{ijk}) \text{ where } m_{ijk} = \sum_r \lambda_r a_{ir} b_{jr} c_{kr} \]

- Nonconvex problem!
  - Assume R is given
- Minimization problem with constraint
  - The decomposed vectors must be non-negative
- Alternating Poisson Regression (Chi and Kolda, 2011)
  - Assume (d-1) factor matrices are known and solve for the remaining one
Alternating Poisson Regression (CP-APR)

Repeat until converged...

1. \( \tilde{A} \leftarrow \arg \min_{A \geq 0} \sum_{ijk} m_{ijk} - x_{ijk} \log m_{ijk} \text{ s.t. } \mathcal{M} = \sum_r \tilde{a}_r \circ b_r \circ c_r \)

2. \( \lambda \leftarrow \mathbf{e}^T \tilde{A}; \ A \leftarrow \tilde{A} \cdot \text{diag}(1/\lambda) \)

3. \( \tilde{B} \leftarrow \arg \min_{B \geq 0} \sum_{ijk} m_{ijk} - x_{ijk} \log m_{ijk} \text{ s.t. } \mathcal{M} = \sum_r a_r \circ \tilde{b}_r \circ c_r \)

4. \( \lambda \leftarrow \mathbf{e}^T \tilde{B}; \ B \leftarrow \tilde{B} \cdot \text{diag}(1/\lambda) \)

5. \( \tilde{C} \leftarrow \arg \min_{C \geq 0} \sum_{ijk} m_{ijk} - x_{ijk} \log m_{ijk} \text{ s.t. } \mathcal{M} = \sum_r a_r \circ b_r \circ \tilde{c}_r \)

6. \( \lambda \leftarrow \mathbf{e}^T \tilde{C}; \ C \leftarrow \tilde{C} \cdot \text{diag}(1/\lambda) \)

Theorem: The CP-APR algorithm will converge to a constrained stationary point if the subproblems are strictly convex and solved exactly at each iteration. (Chi and Kolda, 2011)
Algorithm 1: CPAPR, Alternating Block Framework

1. CPAPR \((\mathcal{X}, \mathcal{M})\);
   **Input**: Sparse \(N\)-mode Tensor \(\mathcal{X}\) of size \(I_1 \times I_2 \times \ldots I_N\) and the number of components \(R\)
   **Output**: Kruskal Tensor \(\mathcal{M} = [\lambda; A^{(1)} \ldots A^{(N)}]\)

2. Initialize
3. repeat
4. \hspace{1em} for \(n = 1, \ldots, N\) do
5. \hspace{2em} Let \(\Pi^{(n)} = (A^{(N)} \odot \ldots \odot A^{(n+1)} \odot A^{(n-1)} \odot \ldots A^{(1)})^T\)
6. \hspace{2em} Compute \(\bar{A}^{(n)}\) that minimize \(f(\bar{A}^{(n)})\) s.t. \(\bar{A}^{(n)} \geq 0\)
7. \hspace{2em} \(A^{(n)} \leftarrow \bar{A}^{(n)}\)
8. end
9. until all mode subproblems converged;

Minimization problem is expressed as:

\[
\min_{\bar{A}^{(n)}>0} f(\bar{A}^{(n)}) = e^T [\bar{A}^{(n)} \Pi^{(n)} - X_{(n)} \ast \log(\bar{A}^{(n)} \Pi^{(n)})] e
\]
Minimization problem is expressed as:
\[
\min_{\tilde{A}^{(n)}>0} f(\tilde{A}^{(n)}) = e^T [\tilde{A}^{(n)} \Pi^{(n)} - X^{(n)} \ast \log(\tilde{A}^{(n)} \Pi^{(n)})] e
\]
Minimization problem is expressed as:

\[
\min_{A^{(n)} > 0} f(A^{(n)}) = e^T [A^{(n)} \Pi^{(n)} - X^{(n)} * \log(A^{(n)} \Pi^{(n)})] e
\]
Minimization problem is expressed as:

- 2 major approaches


ϕ is called Khatori-Rao product
(Column wise Kronecker product)
\[
C = [C_1 | C_2 | C_3] \\
D = [D_1 | D_2 | D_3]
\]
\[
C \circ D = [C_1 \otimes D_1 | C_2 \otimes D_2 | C_3 \otimes D_3]
\]

Π is expressed in sparse matrix (indices and values).
Key Elements of MU and PDNR methods

**Multiplicative Update (MU)**

- **Key computations**
  - Khatri-Rao Product $\Pi^{(n)}$
  - Multiplicative Update Modifier (10+ iterations)

- **Key features**
  - Factor matrix is updated all at once
  - Exploits the convexity of row subproblems for global convergence

**Projected Damped Newton for Row-subproblems (PDNR)**

- **Key computations**
  - Khatri-Rao Product $\Pi^{(n)}$
  - Constrained Non-linear Newton-based optimization for each row

- **Key features**
  - Factor matrix can be updated by rows
  - Exploits the convexity of row-subproblems
Algorithm 1: CP-APR-MU, Multiplicative Update

1 CP-APR-MU ($\mathcal{X}, \mathcal{M}$);

Input: Sparse $N$-mode Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times \ldots I_N$ and the number of components $R$

Output: Kruskal Tensor $\mathcal{M} = [\lambda; A^{(1)} \ldots A^{(N)}]$

2 Initialize

3 repeat

4 for $n = 1, \ldots, N$ do

5 \quad $B \leftarrow (A^{(n)} + S)\Lambda$ (S is used to remove inadmissible zeros)

6 \quad Let $\Pi^{(n)} = (A^{(N)} \odot \ldots \odot A^{(n+1)} \odot A^{(n-1)} \odot \ldots A^{(1)})^T$

7 \quad for $i = 1, \ldots, 10$ do

8 \quad \quad $\Phi^{(n)} \leftarrow (X_{(n)} \odot \max(B\Pi^{(n)}, \epsilon))(\Pi^{(n)})^T$

9 \quad \quad $B \leftarrow B \ast \Phi^{(n)}$

10 \quad end

11 $\lambda = \epsilon^T B$

12 $A^{(n)} \leftarrow B\Lambda^{-1}$, where $\Lambda = \text{diag}(\lambda)$

13 end

14 until all mode subproblems converged;

Key Computations
Algorithm 1: CPAPR-PDNR algorithm

1 \textbf{CPAPR\_PDNR} ($\mathcal{X}, \mathcal{M}$);

\textbf{Input} : Sparse $N$-mode Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times \ldots I_N$ and the number of components $R$

\textbf{Output}: Kruskal Tensor $\mathcal{M} = [\lambda; A^{(1)} \ldots A^{(N)}]$

2 \textbf{Initialize}

3 \textbf{repeat}

4 \textbf{for} $n = 1, \ldots, N$ \textbf{do}

5 \hspace{1em} Let $\Pi^{(n)} = (A^{(N)} \odot \cdots \odot A^{(n+1)} \odot A^{(n-1)} \odot \cdots A^{(1)})^T$

6 \hspace{1em} \textbf{for} $i = 1, \ldots, I_n$ \textbf{do}

7 \hspace{2em} Find $b_i^{(n)}$ s.t. $\min_{b_i^{(n)} \geq 0} f_{\text{row}}(b_i^{(n)}, x_i^{(n)}, \Pi^{(n)})$

8 \hspace{1em} \textbf{end}

9 \hspace{1em} $\lambda = e^T B^{(n)}$ where $B^{(n)} = [b_1^{(n)} \ldots b_{I_n}^{(n)}]^T$

10 \hspace{1em} $A^{(n)} \leftarrow B^{(n)} \Lambda^{-1}$, where $\Lambda = \text{diag}(\lambda)$

11 \hspace{1em} \textbf{end}

12 \textbf{until} all mode subproblems converged;

Key Computations
Focus on on-node parallelism for multiple architectures

- Multiple choices for programming
  - OpenMP, OpenACC, CUDA, Pthread …
  - Manage different low-level hardware features (cache, device memory, NUMA…)

- Our Solution: Use Kokkos for productivity and performance portability
  - Abstraction of parallel loops
  - Abstraction Data layout (row-major, column major, programmable memory)
  - Same code to support multiple architectures
What is Kokkos?

Templated C++ Library by Sandia National Labs (Edwards, et al)
- Serve as substrate layer of sparse matrix and vector kernels
- Support any machine precisions
  - Float, Double, etc

Kokkos::View() accommodates performance-aware multidimensional array data objects
- Light-weight C++ class to accommodate abstractions for platform specific features (host, device, GPU’s shared memory, data access pattern, etc.)

Parallelizing loops using C++ language standard
- Lambda
- Functors

Extensive support of atomics
Provide parallel loop operations using C++ language features

Conceptually, the usage is no more difficult than OpenMP. The annotations just go in different places.

Support for task parallel computing is ongoing (Task Parallel Kokkos and UINTHA)
Algorithm 1: CP-APR-MU in source

1. \textbf{CP-APR-MU} $X, M, R$;
   
   \textbf{Input}: Sparse $N$-mode Tensor $X$ of size $I_1 \times I_2 \times \ldots I_N$ and the number of components $R$
   
   \textbf{Output}: Kruskal Tensor $\mathcal{M} = [\lambda; A^{(1)} \ldots A^{(N)}]$

2. initializeBuffer($X, R$)
3. $\mathcal{E} \leftarrow$ computeIndexMap($X$)
4. \textbf{repeat}
5. \hspace{1em} \textbf{for} $n = 1, \ldots, N$ \textbf{do}
6. \hspace{2em} $M \leftarrow$ offset($M, n$) (Remove inadmissible zeros)
7. \hspace{2em} $M \leftarrow$ distribute($M, n$) (Scale the elements of $A^n$ by $\lambda$)
8. \hspace{2em} $\Pi^{(n)} \leftarrow$ computePi($M, \mathcal{E}^{(n)}$)
9. \hspace{2em} \textbf{for} $i = 1, \ldots, 10$ \textbf{do}
10. \hspace{3em} $\Phi_i^{(n)} \leftarrow$ computePhi($A_i^{(n)}, \Pi^{(n)}, \mathcal{E}^{(n)}$)
11. \hspace{3em} $A_{i+1}^{(n)} \leftarrow A_i^{(n)} \Phi_i^{(n)}$
12. \hspace{2em} \textbf{end}
13. \hspace{2em} $M \leftarrow$ normalize($M, A, n$)
14. \hspace{1em} \textbf{end}
15. \textbf{until} all mode subproblems converged;

\textbf{Data Parallel}
Algorithm 1: CP-APR-PDNR in source

1 \textbf{CP-APR-PDNR} $X, M, R$;
\hspace{1em} \textbf{Input}: Sparse $N$-mode Tensor $X$ of size $I_1 \times I_2 \times \ldots I_N$ and the number of components $R$
\hspace{1em} \textbf{Output}: Kruskal Tensor $\mathcal{M} = [\lambda; A^{(1)} \ldots A^{(N)}]$

2 initializeBuffer($X, R$)
3 $\mathcal{E} \leftarrow \text{computeIndexMap}(X)$
4 repeat
5 \hspace{1em} for $n = 1, \ldots, N$ do
6 \hspace{2em} $M \leftarrow \text{distribute}(M, n)$ (Scale the elements of $A^n$ by $\lambda$ )
7 \hspace{2em} $\Pi^{(n)} \leftarrow \text{computePi}(A, \mathcal{E}^{(n)})$
8 \hspace{2em} \textbf{parallel for} $i = 1, \ldots, I_n$ do
9 \hspace{3em} $a^n_i \leftarrow \text{rowSolvePDNR}(a^n_i, X^n, \Pi^n, \mathcal{E}^{(n)}_i)$
10 \hspace{2em} end
11 \hspace{1em} end
12 $M \leftarrow \text{normalize}(M, A, n)$
13 until all mode subproblems converged;
Use Kokkos::View for all data structures

Sparse Tensor
- Similar to the Coordinate (COO) Format in Sparse Matrix representation

Atomics
- Expensive for CPUs and Manycore
  - Data Rows of Factor Matrices
  - Efficient for the latest GPUs

CP-APR-PDNR
- Nested Parallelism
  - Top Level: Individual Newton Solve
  - Second Level: Vectorized operations
Modifier Computation is the major part of CP-APR-MU.

- **Two ways** to parallelize, which affects the way to access the output factor matrices

1. **Partition with respect to the mode**
   - **No atomics** to access the output vectors by partition
   - **Extra indexing** is required to access nonzero entries by partition (*reordering*)

2. **Partition COO sparse tensor storage format**
   - **No extra indexing** is required
   - Need efficient hardware supported **atomics**
   - The output vector elements are updated by multiple threads concurrently
   - Large outermost loop irrespective of the mode sizes

- Recent work by Smith and Karypis, and Li and Vuduc suggest more efficient data format than COO
Performance Test

Strong Scalability
- Problem size is fixed

Random Tensor
- $3K \times 4K \times 5K$, 10M nonzero entries
- 100 outer iterations

Realistic Problems
- Count Data (Non-negative)
- Available at http://frostt.io/
- 10 outer iterations

<table>
<thead>
<tr>
<th>Data</th>
<th>Dimensions</th>
<th>Nonzeros</th>
<th>Rank (*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBNL</td>
<td>2K x 4K x 2K x 4K x 866K</td>
<td>1.7M</td>
<td>10</td>
</tr>
<tr>
<td>NELL-2</td>
<td>12K x 9K x 29K</td>
<td>77M</td>
<td>10</td>
</tr>
<tr>
<td>NELL-1</td>
<td>3M x 2M x 25M</td>
<td>144M</td>
<td>10</td>
</tr>
<tr>
<td>Delicious</td>
<td>500K x 17M x 3M x 1K</td>
<td>140M</td>
<td>10</td>
</tr>
</tbody>
</table>

(*) if not indicated.
Scalability of CPAPR-MU on CPU (Random)

CP-APR-MU method, 100 outer-iterations, (3000 x 4000 x 5000, 10M nonzero entries), R=100, 2 Haswell (14 core) CPUs per node, HyperThreading disabled
CP-APR-MU: Performance on GPUs (10 inner, 10 outer iterations, 10 components)

<table>
<thead>
<tr>
<th>Data</th>
<th>Haswell CPU 1-core</th>
<th>2 Haswell CPUs 14-cores</th>
<th>2 Haswell CPUs 28-cores</th>
<th>Intel KNL (Cache Mode) 68-core CPU</th>
<th>NVIDIA P100 GPU</th>
<th>NVIDIA V100 GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(s)</td>
<td>Speedup</td>
<td>Time(s)</td>
<td>Speedup</td>
<td>Time(s)</td>
<td>Speedup</td>
</tr>
<tr>
<td>Random</td>
<td>185</td>
<td>1</td>
<td>22</td>
<td>8.4</td>
<td>13</td>
<td>14.11</td>
</tr>
<tr>
<td>LBNL</td>
<td>39</td>
<td>1</td>
<td>19</td>
<td>2.05</td>
<td>13</td>
<td>3.0</td>
</tr>
<tr>
<td>NELL-2</td>
<td>1157</td>
<td>1</td>
<td>137</td>
<td>8.44</td>
<td>87</td>
<td>13.29</td>
</tr>
<tr>
<td>NELL-1</td>
<td>3365</td>
<td>1</td>
<td>397</td>
<td>16.62</td>
<td>258</td>
<td>20.9</td>
</tr>
<tr>
<td>Delicious</td>
<td>4170</td>
<td>1</td>
<td>2183</td>
<td>1.91</td>
<td>1872</td>
<td>2.23</td>
</tr>
</tbody>
</table>
Performance Comparison: Atomic vs Non-Atomic

Intel CPUs: Software-based atomic operations
NVIDIA GPUs: Hardware-based atomic operations
Performance of CPU-APR-MU with respect to different rank size

CP-APR-MU (Random tensor 3Kx4Kx5K, 100 outer iterations)
Performance of CP-APR-MU (LBNL-Network) with respect to different rank sizes

CP-APR-MU (LBNL-NETWORK, 10 outer iterations)
CPAPR-PDNR method, 100 outer-iterations, 1831221 inner iterations total, (3000 x 4000 x 5000, 10M nonzero entries), R=10, 2 Haswell (14 core) CPUs per node, HyperThreading disabled
## Results: CPAPR-PDNR Scalability

<table>
<thead>
<tr>
<th>Data</th>
<th>Haswell CPU 1 core</th>
<th>2 Haswell CPUs 14 cores</th>
<th>2 Haswell CPUs 28 cores</th>
<th>KNL (Cache Mode) 68-core CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(s)</td>
<td>Speedup</td>
<td>Time(s)</td>
<td>Speedup</td>
</tr>
<tr>
<td>Random</td>
<td>238</td>
<td>1</td>
<td>23.7</td>
<td>10.03</td>
</tr>
<tr>
<td>LBNL</td>
<td>1049</td>
<td>1</td>
<td>187</td>
<td>2.35</td>
</tr>
<tr>
<td>NELL-2</td>
<td>2154</td>
<td>1</td>
<td>326</td>
<td>6.63</td>
</tr>
<tr>
<td>NELL-1</td>
<td>17212</td>
<td>1</td>
<td>4241</td>
<td>4.05</td>
</tr>
<tr>
<td>Delicious</td>
<td>28053</td>
<td>1</td>
<td>3684</td>
<td>5.15</td>
</tr>
</tbody>
</table>
Our implementation exhibits very good scalability with the random tensor.

- Similar mode sizes
- Regular distribution of nonzero entries
  - Some cache effects
  - Kokkos is NUMA-aware for contiguous memory access (first-touch)

Some scalability issues with the realistic tensor problems.

- Irregular nonzero distribution and disparity in mode sizes
- PDNR needs some improvement to handle row subproblems
- Preprocessing could improve the locality
  - Explicit Data partitioning (Smith and Karypis)
Conclusion

Development of Portable on-node Parallel CP-APR Solvers
- Data parallelism for MU method
- Mixed Data/Task parallelism for PDNR method
- Multiple Architecture Support using Kokkos
- Performance on CPU, Manycore and GPUs

Scalable Performance for random sparse tensor

Future Work
- Projected Quasi-Newton for Row-subproblems (PQNR)
- GPU and Manycore support for PDNR and PQNR
- Performance tuning to handle irregular nonzero distributions and disparity in mode sizes
Difference between P100 and V100 (300W)

**GPU PERFORMANCE COMPARISON**

<table>
<thead>
<tr>
<th></th>
<th>P100</th>
<th>V100</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL Training</td>
<td>10 TFLOPS</td>
<td>120 TFLOPS</td>
<td>12x</td>
</tr>
<tr>
<td>DL Inferencing</td>
<td>21 TFLOPS</td>
<td>120 TFLOPS</td>
<td>6x</td>
</tr>
<tr>
<td>FP64/FP32</td>
<td>5/10 TFLOPS</td>
<td>7.5/15 TFLOPS</td>
<td>1.5x</td>
</tr>
<tr>
<td>HBM2 Bandwidth</td>
<td>720 GB/s</td>
<td>900 GB/s</td>
<td>1.2x</td>
</tr>
<tr>
<td>STREAM Triad Perf</td>
<td>557 GB/s</td>
<td>855 GB/s</td>
<td>1.5x</td>
</tr>
<tr>
<td>NVLink Bandwidth</td>
<td>160 GB/s</td>
<td>300 GB/s</td>
<td>1.9x</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>4 MB</td>
<td>6 MB</td>
<td>1.5x</td>
</tr>
<tr>
<td>L1 Caches</td>
<td>1.3 MB</td>
<td>10 MB</td>
<td>7.7x</td>
</tr>
</tbody>
</table>

Courtesy: NVIDIA