Portability and Scalability of Sparse Tensor Decompositions on CPU/MIC/GPU Architectures

Keita Teranishi, Christopher Forster, Daniel Dunlavy and Tamara Kolda

SAND2017-7812 PE
HPDA Tensor Project

- Develop production quality library software to perform CP factorization for *Poisson Regression Problems* for HPC platforms
- 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

\[
\mathbf{X} \approx \left( \begin{array}{ccc}
\lambda_1 & \mathbf{c}_1 & \mathbf{b}_1 \\
\mathbf{a}_1 & \lambda_2 & \mathbf{b}_2 \\
& \lambda_R & \mathbf{b}_R \\
\end{array} \right) + \cdots
\]

Model: \( \mathcal{M} = \sum_r \lambda_r \mathbf{a}_r \odot \mathbf{b}_r \odot \mathbf{c}_r \)

\[
x_{ijk} \approx m_{ijk} = \sum_r \lambda_r \mathbf{a}_{ir} \mathbf{b}_{jr} \mathbf{c}_{kr}
\]

- Express the important feature of data using a small number of vector outer products

Key references: Hitchcock (1927), Harshman (1970), Carroll and Chang (1970)
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
New Method: 
Alternating Poisson Regression (CP-APR)

Repeat until converged...

1. $\tilde{A} \leftarrow \arg \min_{A \geq 0} \sum_{ijk} m_{ijk} - x_{ijk} \log m_{ijk}$ s.t. $M = \sum_{r} \tilde{a}_{r} \circ b_{r} \circ c_{r}$

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

3. $\tilde{B} \leftarrow \arg \min_{B \geq 0} \sum_{ijk} m_{ijk} - x_{ijk} \log m_{ijk}$ s.t. $M = \sum_{r} a_{r} \circ \tilde{b}_{r} \circ c_{r}$

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

5. $\tilde{C} \leftarrow \arg \min_{C \geq 0} \sum_{ijk} m_{ijk} - x_{ijk} \log m_{ijk}$ s.t. $M = \sum_{r} a_{r} \circ b_{r} \circ \tilde{c}_{r}$

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

**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)
Minimization problem is expressed as:

\[
\min_{\tilde{A}^{(n)}>0} f(\tilde{A}^{(n)}) = e^T [\tilde{A}^{(n)} \Pi^{(n)} - X^{(n)} \ast \log(\tilde{A}^{(n)} \Pi^{(n)})] e
\]
2 major approaches
- **Multiplicative Updates** like Lee & Seung (2000) for matrices, but extended by Chi and Kolda (2011) for tensors
- **Newton and Quasi-Newton method for Row-subproblems** by Hansen, Plantenga and Kolda (2014)
Key Elements of MU and PDNR methods

Multiplicative Update (MU)

- Key computations
  - Khatri-Rao Product $\prod^{(n)}$
  - 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 $\prod^{(n)}$
  - 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.         $B \leftarrow (A^{(n)} + S)\Lambda$ ($S$ is used to remove inadmissible zeros)

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

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

8.             $\Phi^{(n)} \leftarrow (X^{(n)} \circ \max(B\Pi^{(n)}, \epsilon))(\Pi^{(n)})^T$

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

10.     end

11.     $\lambda = e^T B$

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. **CPAPR_PDNR** $(\mathcal{X}, \mathcal{M})$;
   - **Input**: Sparse $N$-mode Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times \ldots \times I_N$ 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. **Let** $\Pi^{(n)} = (A^{(N)} \odot \ldots \odot A^{(n+1)} \odot A^{(n-1)} \odot \ldots A^{(1)})^T$

6. **for** $i = 1, \ldots, I_n$ **do**

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

8. **end**

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

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

11. **end**

12. **until** all mode subproblems converged;

---

Key Computations
PARALLEL CP-APR ALGORITHMS
Parallelizing CP-APR

- 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
    - Quad and Half float if needed.

- Kokkos::View() accommodates performance-aware multidimensional array data objects
  - Light-weight C++ class to

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

- Extensive support of atomics
Parallel Programming with Kokkos

- Provide parallel loop operations using C++ language features
- Conceptually, the usage is no more difficult than OpenMP. The annotations just go in different places.
Why Kokkos?

- Comply C++ language standard!
- Support multiple back-ends
  - Pthread, OpenMP, CUDA, Intel TBB and Qthread
- Support multiple data layout options
  - Column vs Row Major
  - Device/CPU memory
- Support different parallelism
  - Nesting support
  - Vector, threads, Warp, etc.
  - Task parallelism (under development)
Array Access by Kokkos

Kokkos::View<double **, Layout, Space>

View<double **, Right, Space>  View<double **, Left, Space>

Row-major

Thread 0 reads
Thread 1 reads

Column-major

Thread 0 reads
Thread 1 reads
Array Access by Kokkos

Kokkos::View<double **, Layout, Space>

View<double **, Right, Host>  View<double **, Left, CUDA>

Row-major
Thread 0 reads
Thread 1 reads

Column-major
Thread 0 reads
Thread 1 reads

Contiguous reads per thread
Coalesced reads within warp
Parallel CP-APR-MU

Algorithm 1: CP-APR-MU in source

1. CP-APR-MU $X, M, R$;
   - **Input**: Sparse $N$-mode Tensor $X$ of size $I_1 \times I_2 \times \ldots \times I_N$ and the number of components $R$
   - **Output**: Kruskal Tensor $\mathcal{M} = [\lambda; A^{(1)} \ldots A^{(N)}]$
2. initializeBuffer($X, R$)
3. $\mathcal{E} \leftarrow$ computeIndexMap($X$)
4. repeat
5.   for $n = 1, \ldots, N$ do
6.     $M \leftarrow$ offset($M, n$) (Remove inadmissible zeros)
7.     $M \leftarrow$ distribute($M, n$) (Scale the elements of $A^n$ by $\lambda$)
8.     $\Pi^{(n)} \leftarrow$ computePi($M, \mathcal{E}^{(n)}$)
9.     for $i = 1, \ldots, 10$ do
10.     $\Phi_i^{(n)} \leftarrow$ computePhi($A_i^{(n)}, \Pi^{(n)}, \mathcal{E}^{(n)}$)
11.     $A_i^{(n)} \leftarrow A_i^{(n)} \Phi_i^{(n)}$
12.     end
13.   $M \leftarrow$ normalize($M, A, n$)
14. end
15. until all mode subproblems converged;
Algorithm 1: CP-APR-PDNR in source

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

2. `initializeBuffer(X, R)`
3. `\mathcal{E} \leftarrow computeIndexMap(X)`
4. `repeat`
5. `for n = 1, \ldots, N do`
6. `M \leftarrow distribute(M, n)` (Scale the elements of $A^n$ by $\lambda$)
7. `\Pi^{(n)} \leftarrow computePi(A, \mathcal{E}^{(n)})`
8. `parallel for i = 1, \ldots, I_n do`
9. `\quad a_i^n \leftarrow rowSolvePDNR(a_i^n, X^n, \Pi^n, \mathcal{E}_i^{(n)})`
10. `end`
11. `M \leftarrow normalize(M, A, n)`
12. `end`
13. `until all mode subproblems converged;`
Notes on Data Structure

- Use Kokkos::View
- Sparse Tensor
  - Similar to the Coordinate (COO) Format in Sparse Matrix representation
- Kruskal Tensor & Khatri Rao Product
  - Provides options for row or column major
    - Kokkos::View provides an option to define the leading dimension.
    - Determined during compile or run time
- Avoid Atomics
  - Expensive in CPUs and Manycore
  - Use extra indexing data structure
- CP-APR-PDNR
  - Creates a pool of tasks
  - A dedicated buffer space (Kokkos::View) is assigned to individual task
PERFORMANCE
Performance Test

- **Strong Scalability**
  - Problem size is fixed
- **Random Tensor**
  - 3K x 4K x 5K, 10M nonzero entries
  - **100 outer iterations**
- **Realistic Problems**
  - Count Data (Non-negative)
  - Available at [http://frostt.io/](http://frostt.io/)
  - **10 outer iterations**
- **Double Precision**

<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>
CPAPR-MU on CPU (Random)

CP-APR-MU method, 100 outer-iterations, (3000 x 4000 x 5000, 10M nonzero entries), R=10, PC cluster, 2 Haswell (14 core) CPUs per node, MKL-11.3.3, HyperThreading disabled
Results: CPAPR-MU 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 68-core CPU</th>
<th>NVIDIA P100 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>
</tr>
<tr>
<td>Random</td>
<td>1715*</td>
<td>1</td>
<td>279</td>
<td>6.14</td>
<td>165</td>
</tr>
<tr>
<td>LBNL</td>
<td>131</td>
<td>1</td>
<td>32</td>
<td>4.09</td>
<td>32</td>
</tr>
<tr>
<td>NELL-2</td>
<td>1226</td>
<td>1</td>
<td>159</td>
<td>7.77</td>
<td>92</td>
</tr>
<tr>
<td>NELL-1</td>
<td>5410</td>
<td>1</td>
<td>569</td>
<td>9.51</td>
<td>349</td>
</tr>
<tr>
<td>Delicious</td>
<td>5761</td>
<td>1</td>
<td>2542</td>
<td>2.26</td>
<td>2524</td>
</tr>
</tbody>
</table>

100 outer iterations for the random problem
10 outer iterations for realistic problems
* Pre-Kokkos C++ code on 2 Haswell CPUs:
  1-core, 2136 sec
  14-cores, 762 sec
  28-cores, 538 sec
CPAPR-PDNR method, 100 outer-iterations, 1831221 inner iterations total, (3000 x 4000 x 5000, 10M nonzero entries), R=10, PC cluster, 2 Haswell (14 core) CPUs per node, MKL-11.3.3, 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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(s)</td>
<td>Speedup</td>
<td>Time(s)</td>
</tr>
<tr>
<td>Random</td>
<td>817*</td>
<td>1</td>
<td>73</td>
</tr>
<tr>
<td>LBNL</td>
<td>441</td>
<td>1</td>
<td>187</td>
</tr>
<tr>
<td>NELL-2</td>
<td>2162</td>
<td>1</td>
<td>326</td>
</tr>
<tr>
<td>NELL-1</td>
<td>17212</td>
<td>1</td>
<td>4241</td>
</tr>
<tr>
<td>Delicious</td>
<td>18992</td>
<td>1</td>
<td>3684</td>
</tr>
</tbody>
</table>

100 outer iterations for the random problem
10 outer iterations for realistic problems
* Pre-Kokkos C++ code spends 3270 sec on 1 core
Performance Issues

- 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
  - Task-parallel code may have some memory locality issues to access sparse tensor, Kruskal Tensor, and Khatori-Rao product
  - Preprocessing could improve the locality
    - Explicit Data partitioning (Smith and Karypis)
    - Possible to implement using Kokkos
Memory Bandwidth (Stream Benchmark)

- All cores deliver approximately 8x performance improvement from single thread.
- Hard to scale using all cores with memory-bound code.
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

- 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