Parallelizing and Scaling Tensor Computations*

Muthu Baskaran, Benoit Meister, Richard Lethin

Reservoir Labs

* Patent Pending Technology
Introduction

ENSIGN (Exascale Non-Stationary Graph Notation)

- **Goal**
  - Optimized tensor toolbox for dynamic graph analytics
    - Produce/provide optimized tensor computations for large-scale parallel systems

- **Features**
  - High-performance implementation of different variants of tensor decomposition methods
    - Scalable optimizations for tensor computations
  - Automatic parallelization and data locality optimizations
    - Inter-operate with Reservoir Labs' auto-parallelizing compiler R-Stream
Presentation Roadmap

About ENSIGN Tensor Toolbox

Optimization and Parallelization Techniques

Performance Evaluation

Summary & Forward Work
ENSIGN Tensor Toolbox

Available in two versions

- **ETTB++v3.5.1**
  - Accelerated C++ version of tensor toolbox built on top of Sandia National Laboratories C++ Tensor Toolbox v1.0.2

- **ETTB v1.0**
  - Accelerated C version of tensor toolbox

Toolbox of core algorithms

- CP decomposition variants
  - CP-ALS, CP-APR\(^1\), CP-APR-PDNR\(^2\), INDSCAL

- Tucker decomposition variants
  - HOOI, memory-efficient HOOI\(^3\)

- Methods for “low-rank updates”\(^4\)

- Coupled (Joint) tensor decomposition\(^5\)

- Standardized tensor decomposition\(^6\)

---

\(^1\)[Chi, Kolda 2012], \(^2\)[Hansen et al, 2014], \(^3\)[Kolda et al, 2008], \(^4\)[Ohara 2010], \(^5\)[MATLAB CMTF Toolbox, Acar], \(^6\)[Brown et al, 2014]
Presentation Roadmap

About ENSIGN Tensor Toolbox

Optimization and Parallelization Techniques

Performance Evaluation

Summary & Forward Work
New Data Structures for Scaling the Computations

\[ S, b, \ldots, k, \ldots, 1 \]

\[ D[0][M_{d1}]...[M_{da}] \]

\[ D[nnz-1][M_{d1}]...[M_{da}] \]

\[ S : matrix storing the coordinates of sparse modes in each distinct bucket \]
\[ D[nnz][M_{d1}]...[M_{da}] : dense sub-tensors \]

**Mode-generic sparse tensor storage**

\((a) Mode generic format\)

**Mode-specific sparse tensor storage**

\((b) Mode specific format\)

Optimizing Dense Computations

R-Stream optimizations

- State-of-the-art polyhedral compiler algorithms
- Optimize dense computations automatically
- Advanced compiler techniques using "polyhedral model" for
  - Parallelization, Locality, Contiguity, Vectorization, ...
  - "Affine" transformations of arbitrary loop nests

Affine Schedule $\theta$ maps iterations to multi-dimensional space-time

Initial Iteration space of a statement $S(i,j)$

Transformed Iteration space of $S(i,j)$
Optimizing Sparse Computations

Challenge: Optimizing "sparse" computations

• A "data-driven" scheduling problem
• Need to efficiently handle irregular memory accesses
• Current parallelization efforts (including R-Stream) have scope for improvement
  – Parallelism, synchronization, data locality, etc.

Goals of our improvisation techniques

– Uncover more concurrency
– Reduce synchronization
– Improve data locality
– Achieve load balance
– Reduce scheduling overhead
Mixed Static and Dynamic Runtime Scheduling

- Static scheduling – poor load balance, low scheduling overhead
- Dynamic scheduling – good load balance, high scheduling overhead
- Our Approach – Achieves the pros of both schemes
  - One dynamic scheduling iteration to get a load balanced pattern
  - Static scheduling using the pattern for later iterations
  - Good load balance, low scheduling overhead

Improved Data Locality

• Memory-hierarchy aware approach
  – Task distribution across processor cores in the dynamic scheduling iteration governed by
    • Data touched by them
    • Memory in which data resides
  – Over-loaded cores “steal” tasks from “topologically” closer neighbors that are under-loaded
    • NUMA topology in shared memory systems
  – Facilitate data sharing across cores

Optimizing Tucker Decomposition

Data Reuse Optimization

Tucker decomposition algorithm (HOOI method)

repeat
  for $n = 1 \ldots N$ do
    $y = \mathbf{X} \times_1 \mathbf{A}^{(1)T} \ldots \times_{n-1} \mathbf{A}^{(n-1)T} \times_{n+1} \mathbf{A}^{(n+1)T} \ldots \times_N \mathbf{A}^{(N)T}$
    $\mathbf{A}_n = \mathbf{J}_n$ leading left singular vectors of $\mathbf{Y}_n$
  end for
  $\mathbf{g} = y \times_N \mathbf{A}^{(N)T}$
until convergence

No. of reuses: $\frac{N^2}{2} - \frac{3N}{2} + 1$

Optimizing Tucker Decomposition
Data Reuse Optimization

Tucker decomposition algorithm
(HOOI method)

repeat
for $n = 1 \ldots \left[ \frac{N}{2} \right]$ do
  $y = X \times \mathbf{A}^{(N)^T} \times_{n+1} \mathbf{A}^{(n+1)^T} \times_{n-1} \mathbf{A}^{(n-1)^T} \times_1 \mathbf{A}^{(1)^T}$
  $\mathbf{A}_n = J_n$ leading left singular vectors of $Y_n$
end for
for $n = \left[ \frac{N}{2} \right] + 1 \ldots N$ do
  $y = X_1 \mathbf{A}^{(1)^T} \times_{n-1} \mathbf{A}^{(n-1)^T} \times_{n+1} \mathbf{A}^{(n+1)^T} \times_{N} \mathbf{A}^{(N)^T}$
  $\mathbf{A}_n = J_n$ leading left singular vectors of $Y_n$
end for
$y = y \times \mathbf{A}^{(N)^T}$
until convergence

No. of reuses: $\geq \frac{N^2}{2} - \frac{3N}{2} + \frac{(N-2)^2}{4} + 1 + 1$

Optimizing Tucker Decomposition
Memory-efficient Scalable Optimization

Memory blowup problem in Tucker decomposition

• Intermediate tensors in computation
  – Storage vs computation trade-off
• Uses mode-generic sparse formats for intermediate tensors in computation
  – State-of-the-art approach uses dense formats for intermediate tensors
• Optimally categorizes modes as elementwise and standard based on available memory (similar to Kolda et al. 2008)
  – Optimal order of n-Mode products in a sequence that reduces total computation cost and total memory consumption
• Uses data reuse optimization

Presentation Roadmap

About ENSIGN Tensor Toolbox

Optimization and Parallelization Techniques

Performance Evaluation

Summary & Forward Work
Performance Evaluation

Benchmarked different methods on different sized datasets

- Intel Xeon E5-4620 2.2 GHz (Quad socket 8-core)

Reservoir Labs Tensorstation™
Performance Evaluation
CP-ALS evaluation

<table>
<thead>
<tr>
<th>Tensor</th>
<th>Size</th>
<th>Non-zeros</th>
<th>#iterations timed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook</td>
<td>63891 x 63891 x 1591</td>
<td>737934</td>
<td>50</td>
</tr>
</tbody>
</table>

![Bar chart showing time (s) vs number of cores for Baseline and ENSIGN](attachment:chart.png)
Performance Evaluation

CP-ALS evaluation

<table>
<thead>
<tr>
<th>Tensor</th>
<th>Size</th>
<th>Non-zeros</th>
<th>#iterations timed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyber</td>
<td>565872 x 795 x 13868 x 21862</td>
<td>4865458</td>
<td>50</td>
</tr>
</tbody>
</table>

The table shows the performance evaluation of CP-ALS for a tensor named Cyber. The table includes columns for Tensor, Size, Non-zeros, and #iterations timed.

The chart below illustrates the time (in seconds) taken by Baseline and ENSIGN for different numbers of cores. The x-axis represents the number of cores, ranging from 1 to 32, and the y-axis represents the time in seconds, ranging from 0 to 300. The chart shows a comparison between Baseline (blue bars) and ENSIGN (orange bars) for each number of cores.
Performance Evaluation
CP-APR evaluation

<table>
<thead>
<tr>
<th>Tensor</th>
<th>Size</th>
<th>Non-zeros</th>
<th>#iterations timed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook</td>
<td>63891 x 63891 x 1591</td>
<td>737934</td>
<td>190</td>
</tr>
</tbody>
</table>

![Graph showing performance evaluation of CP-APR with different numbers of cores and runtime.](image)
Performance Evaluation
CP-APR evaluation

<table>
<thead>
<tr>
<th>Tensor</th>
<th>Size</th>
<th>Non-zeros</th>
<th>#iterations timed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyber</td>
<td>14811811 x1899 x1899 x 3 x 6067</td>
<td>2085108</td>
<td>50</td>
</tr>
</tbody>
</table>
Performance Evaluation

Tucker evaluation

- Timed \textit{all but one} sequence of tensor matrix products
- Data set
  - Number of modes: 4
  - Dimensionality of input tensor: 1000 x 1000 x 1100 x 200
  - Number of non-zeros = 5.5M

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>175.17</td>
</tr>
<tr>
<td>Kolda et al. Approach</td>
<td>21.79</td>
</tr>
<tr>
<td>Our Approach (partial data reuse)</td>
<td>9.29</td>
</tr>
<tr>
<td>Our Approach (optimal data reuse)</td>
<td>7.12</td>
</tr>
</tbody>
</table>

Our sequential version: 3x over existing approach

Time for one iteration; typically 75-100 iterations
Performance Evaluation
Tucker evaluation

- Timed parallel code (with optimal data reuse)

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.12</td>
</tr>
<tr>
<td>2</td>
<td>6.25</td>
</tr>
<tr>
<td>4</td>
<td>3.80</td>
</tr>
<tr>
<td>8</td>
<td>2.57</td>
</tr>
</tbody>
</table>

Our parallel version: 8.5x over existing approach's sequential version
Presentation Roadmap

About ENSIGN Tensor Toolbox

Optimization and Parallelization Techniques

Performance Evaluation

Summary & Forward Work
Summary of ENSIGN Techniques

Performance:

• Developed techniques to effectively parallelize and scale large sparse and dense tensor computations
  – New efficient sparse formats
  – Extract maximal parallelism
  – Extract data locality & data reuse
    • Reduce data movement
  – Reduce/Avoid unnecessary computations

Capability:

• Software released to customers
• Demonstrated on real-world problems
Ongoing and Forward Work

More focus on applying ENSIGN on real-world problems

- Genomics
- Cyber security

Enhancing usability of the tool

- Graphical User Interface
- Visualization of decompositions

Distributed-memory versions of tensor methods