Blocking Optimization Strategies for Sparse Tensor Computation

Jee Choi¹, Xing Liu¹, Shaden Smith², and Tyler Simon³
¹IBM T. J. Watson Research, ²University of Minnesota, ³University of Maryland Baltimore County

SIAM Annual Meeting
July 12th, 2017
Tensors are multi-dimensional arrays

• CANDECOMP/Parafac (CP) decomposition creates a set of factor matrices
The take-away from this presentation

• There is lack of clear understanding about performance bottlenecks in sparse tensor decomposition
• Using various blocking techniques mitigate these bottlenecks
• Our optimizations demonstrate significant speedup on synthetic and real-world data for both shared-memory and distributed implementations
Fix every other factor matrix and solve for the remaining one

procedure CP-ALS (X, R)
    repeat
        C = \(X_{(3)}(B \otimes A)(B^TB \ast A^TA)^X\)
        normalize columns of C to length 1
        B = \(X_{(2)}(C \otimes A)(C^TC \ast A^TA)^X\)
        normalize columns of B to length 1
        A = \(X_{(1)}(C \otimes B)(C^TC \ast B^TB)^X\)
        store column norms of A in \(\lambda\) and normalize to 1
    until max iteration reached or error less than \(\epsilon\)
end procedure
Calculating MTTKRP is the primary bottleneck

procedure CP-ALS $(X, R)$
repeat
C = $X_{(3)}(B \odot A)(B^TB \ast A^TA)^X$
normalize columns of C to length 1
B = $X_{(2)}(C \odot A)(C^TC \ast A^TA)^X$
normalize columns of B to length 1
A = $X_{(1)}(C \odot B)(C^TC \ast B^TB)^X$
store column norms of A in $\lambda$ and normalize to 1
until max iteration reached or error less than $\epsilon$
end procedure
Calculating MTTKRP is the primary bottleneck

```plaintext
procedure CP-ALS (X, R)
    repeat
        C = X(3) (B ⊗ A) (B^T B * A^T A)^X
        normalize columns of C to length 1
        B = X(2) (C ⊗ A) (C^T C * A^T A)^X
        normalize columns of B to length 1
        A = X(1) (C ⊗ B) (C^T C * B^T B)^X
        store column norms of A in \( \lambda \) and normalize to 1
    until max iteration reached or error less than \( \epsilon \)
end procedure
```
Calculating MTTKRP is the primary bottleneck

procedure CP-ALS \((X, R)\)
repeat
\[ C = X_{(3)}(B \odot A)(B^T B \ast A^T A)^X \]
normalize columns of \( C \) to length 1
\[ B = X_{(2)}(C \odot A)(C^T C \ast A^T A)^X \]
normalize columns of \( B \) to length 1
\[ A = X_{(1)}(C \odot B)(C^T C \ast B^T B)^X \]
store column norms of \( A \) in \( \lambda \) and normalize to 1
until max iteration reached or error less than \( \epsilon \)
end procedure
Calculating MTTKRP is the primary bottleneck

procedure CP-ALS \((X, R)\)

repeat

\[ C = X_{(3)}(B \odot A)(B^T B * A^T A)^X \]

normalize columns of \(C\) to length 1

\[ B = X_{(2)}(C \odot A)(C^T C * A^T A)^X \]

normalize columns of \(B\) to length 1

\[ A = X_{(1)}(C \odot B)(C^T C * B^T B)^X \]

store column norms of \(A\) in \(\lambda\) and normalize to 1

until max iteration reached or error less than \(\varepsilon\)

end procedure
Calculating MTTKRP is the primary bottleneck

\begin{algorithm}
\textbf{procedure} CP-ALS \((X, R)\)
\begin{algorithmic}
\REPEAT
\STATE \(C = X_{(3)}(B \odot A)(B^T B \ast A^T A)^X\)
\STATE normalize columns of \(C\) to length 1
\STATE \(B = X_{(2)}(C \odot A)(C^T C \ast A^T A)^X\)
\STATE normalize columns of \(B\) to length 1
\STATE \(A = X_{(1)}(C \odot B)(C^T C \ast B^T B)^X\)
\STATE store column norms of \(A\) in \(\lambda\) and normalize to 1
\UNTIL max iteration reached or error less than \(\epsilon\)
\ENDREPEAT
\end{algorithmic}
\textbf{end procedure}
\end{algorithm}
Problem is formulated as matrix operations

procedure CP-ALS (X, R)
    repeat
        C = X(3) (B \odot A) (B^T B * A^T A)^X
        normalize columns of C to length 1
        B = X(2) (C \odot A) (C^T C * A^T A)^X
        normalize columns of B to length 1
        A = X(1) (C \odot B) (C^T C * B^T B)^X
        store column norms of A in \lambda and normalize to 1
    until max iteration reached or error less than \epsilon
end procedure
Directly computing MTTKRP is very expensive

- For a $1000 \times 1000 \times 1000$ tensor with rank 100...
  - $X_{(3)}$ is a $1,000 \times 1,000,000$ matrix, and
  - $(B \bigodot A)$ is a $1,000,000 \times 100$ matrix
  - Direct computation is expensive
But not necessary

- For a $1000 \times 1000 \times 1000$ tensor with rank 100...
  - $X_{(3)}$ is a $1,000 \times 1,000,000$ matrix, and
  - $(B \odot A)$ is a $1,000,000 \times 100$ matrix
  - Direct computation is expensive
- Not necessary for sparse tensors.
MTTKRP expressed as matrix operations

Matricized tensor

1,000,000

X

A'

1,000

1,000,000

100

Khatri-Rao Product

Same size as the factor matrix you want to calculate
MTTKRP simplified

Khatri-Rao Product
MTTKRP simplified

Load 1 row each from B and C
MTTKRP simplified

Hadamard product

Load 1 row each from B and C
Load 1 row each from B and C

Hadamard product

Scale by non-zero

MTTKRP simplified
MTTKRP simplified

- Load 1 row each from B and C
- Scale by non-zero Hadamard product
- Accumulate to A
Can it be done more efficiently?

- Load 1 row each from B and C
- Hadamard product
- Scale by non-zero
- Accumulate to A
In 3D space...

Reduce computing by processing at fiber granularity

**Fiber** – column vectors with all but one mode fixed.
Reduce computing by processing at fiber granularity

**Fiber** – column vectors with all but one mode fixed

3. Buffer

1,000,000

1,000

X

A'

B

C
Reduce computing by processing at fiber granularity

**Fiber** – column vectors with all but one mode fixed

**Mode-2 fiber**
\[ x = 10, \ z = 30 \]

1,000,000

3. Buffer

1,000

10,20,30

10,40,30

1,000,000

100

A'

B

C
First load rows from B (mode-2 matrix)

Load 2 rows of B

row 20
row 40

10,20,30
10,40,30
100
1,000,000
1,000,000
X
1,000
A'

B
C
Scale the rows by non-zero values

Load 2 rows of B
row 20
row 40

Scale
10,20,30
1,000,000

Scale
10,40,30
1,000,000

X
1,000

A'

1,000,000
Accumulate them to a temporary buffer
Load the “common” row from C (mode-3 matrix)
Hadamard product with buffer

Load 2 rows of B

Row 20

Row 40

Row 30

Scale

10,20,30

Scale

10,40,30

Accum to buffer

Hadamard product

A'
Accumulate to destination matrix \((A')\)

- Scale
- Load 2 rows of B
- Accumulate
- Hadamard product
- Accum to buffer
This is called compressed sparse fiber (CSF)

Claimed Savings by others

• Naïve COO kernel
  • Regular: 3 * m * R flops (2mR for initial product + scale, mR for accumulation)

• CSF
  • 2R(m + P) flops, P is # of non-empty fibers
  • typically p <<< m

• DFacTo
  • Formulates MTTKRP as SpMV
  • Each column is computed independently via 2 SpMV
  • 2R(m + P) flops

• GigaTensor
  • MapReduce
  • Increased parallelism vs. more flops
  • 5mR flops

\[ m = \# \text{ of non-zeros} \]
\[ P = \# \text{ of non-empty fibers} \]
\[ R = \text{rank} \]
Does this make sense?

• Sparse computations are memory bandwidth-bound
• SPLATT tries cache blocking through expensive hypergraph partitioning (without much success)
Roofline model visualized (for an old Intel Nehalem CPU)
Commonly used scientific kernels

- DGEMM
- SpMV
- FFT
Roofline model applied to MTTKRP

• Sparse computations are memory bandwidth-bound
• Let’s calculate the # of flops and # of bytes and compare
  • Flops: \( W = 2R(m + P) \)
  • Bytes: \( Q = 2m \) (value + mode-2 index) + \( 2P \) (mode-3 index + mode-3 pointer)
    + \( (1-\alpha)Rm \) (mode-2 factor) + \( (1-\alpha)RP \) (mode-3 factor)

• Arithmetic Intensity
  • Ratio of work to communication \( I = \frac{W}{Q} \)
  • \( I = \frac{W}{(Q \times 8 \text{ Bytes})} = \frac{R}{(8 + 4R(1-\alpha))} \)
Arithmetic intensity of MTTKRP with rank = 32

- Cache hit = 0.5
- Cache hit = 1.0 (perfect)
- Cache hit = 0.9
Arithmetic intensity vs. rank for various cache hit rates

Arithmetic
Intensity

Rank
16 32 64 128 256 512 1024 2048

Perfect cache hit
Cache hit = 0.99
Cache hit = 0.95
Cache hit = 0.8
Cache hit = 0.5
Arithmetic intensity vs. system balance (on the latest CPU)

Arithmetic Intensity

System balance – 22-core CPU

Perfect cache hit

Cache hit = 0.99

Cache hit = 0.95

Rank

16  32  64  128  256  512  1024  2048
Our initial conclusion from a theoretical point of view

• On recent systems, MTTKRP is **likely** memory-bound
  • Even with a perfect cache hit rate, MTTKRP should be memory-bound on lower ranks
  • If we fail to get good cache re-use, MTTKRP will most likely be memory bound for any rank
A pressure point analysis reveals the bottleneck

• Pressure point analysis
  • Probe potential bottlenecks by creating and eliminating instructions/data access
  • If we suspect that # of registers is the bottleneck, try increasing/decreasing their usage to see if the exec. time changes.
  • Inline assembly to prevent dead code elimination (DCE)

Kenneth Czechowski, Performance Analysis Using the Pressure Point Analysis, PhD dissertation
A pressure point analysis reveals the bottleneck

<table>
<thead>
<tr>
<th>Time</th>
<th>Pressure point</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Baseline (2R(m + P) flops)</td>
</tr>
</tbody>
</table>
Using COO instead of CSF only increases exec. time by < 2%

<table>
<thead>
<tr>
<th>Time</th>
<th>Pressure point</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Baseline (2R(m + P) flops)</td>
</tr>
<tr>
<td>2.64</td>
<td>Move flops to inner loop (3 * m * R flops)</td>
</tr>
</tbody>
</table>

Increasing flops only changes time by < 2%
Removing access to C (accessed once per fiber): exec. time down by 7%  

<table>
<thead>
<tr>
<th>Time</th>
<th>Pressure point</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Baseline ((2R(m + P) \text{ flops}))</td>
</tr>
<tr>
<td>2.64</td>
<td>Move flops to inner loop ((3 \times m \times R \text{ flops}))</td>
</tr>
<tr>
<td>2.43</td>
<td>Access to C removed</td>
</tr>
</tbody>
</table>

Removing per-fiber access to matrix C has a bigger impact than increasing flops
Suspicion confirmed: Memory access to B is the bottleneck

<table>
<thead>
<tr>
<th>Time</th>
<th>Pressure point</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Baseline (2R(m + P) flops)</td>
</tr>
<tr>
<td>2.64</td>
<td>Move flops to inner loop (3 * m * R flops)</td>
</tr>
<tr>
<td>2.43</td>
<td>Access to C removed</td>
</tr>
<tr>
<td>1.81</td>
<td>Access to B limited to L1 cache</td>
</tr>
</tbody>
</table>

Limiting our suspect has a huge impact
<table>
<thead>
<tr>
<th>Time</th>
<th>Pressure point</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Baseline ((2R(m + P) \text{ flops}))</td>
</tr>
<tr>
<td>2.64</td>
<td>Move flops to inner loop ((3 \times m \times R \text{ flops}))</td>
</tr>
<tr>
<td>2.43</td>
<td>Access to C removed</td>
</tr>
<tr>
<td>1.81</td>
<td>Access to B limited to L1 cache</td>
</tr>
<tr>
<td>1.63</td>
<td>Access to B removed completely</td>
</tr>
</tbody>
</table>

Completely removing it gives us an extra 6% - why?

Eliminating it completely gives us an extra 6% boost
Conclusions from our empirical analysis

• Flops aren’t the issue

• Bottlenecks
  1. Data access to B
  2. Load instructions
Cache/register blocking should help alleviate these bottlenecks

• Flops aren’t the issue
• Bottlenecks
  1. Data access to B → cache blocking
  2. Load instructions → register blocking
procedure mttkrp (X ∈ ℝ^{I×J×K}, R)
1: for i ← 0 to I do // for each row
2:   for j ← i_ptr[i] to i_ptr[i+1] do // for each fiber
3:     for k ← p_ptr[j] to p_ptr[j+1] do // for each nz in fiber
4:       for r ← 0 to R do // go through entire rank
5:         buffer[r] += vals[k] * B[j_index[k]][r] // buffer
6:         for r ← 0 to R do
7:           A[i][r] += buffer[r] * C[k_index[j]][r] // accumulate
end procedure
procedure mttkrp (X ∈ \(R^{I \times J \times K}\), R)
1: for i ← 0 to I do // for each row
2: for j ← i_ptr[i] to i_ptr[i+1] do // for each fiber
3: for k ← p_ptr[j] to p_ptr[j+1] do // for each nz in fiber
4: for r ← 0 to R do // go through entire rank
5: buffer[r] += vals[k] * B[j_index[k]][r] // buffer
6: for r ← 0 to R do
7: A[i][r] += buffer[r] * C[k_index[j]][r] // accumulate
end procedure

3 LD instructions
procedure mttkrp (X ∈ R^{I×J×K}, R)
1:    for i ← 0 to I do // for each row
2:        for j ← i_ptr[i] to i_ptr[i+1] do // for each fiber
3:            for r ← 0 to R do in 16 increments
4:                for k ← p_ptr[j] to p_ptr[j+1] do // for each nz in fiber
5:                    registers += vals[k] * B[j_index[k]][r] // buffer
6:                A[i][r] += registers * C[k_index[j]][r] // accumulate
end procedure
procedure mttkrp \( (X \in \mathbb{R}^{I \times J \times K}, R) \)
1: for \( i \leftarrow 0 \) to \( I \) do \hspace{1cm} // for each row
2: for \( j \leftarrow i_{ptr}[i] \) to \( i_{ptr}[i+1] \) do \hspace{1cm} // for each fiber
3: for \( r \leftarrow 0 \) to \( R \) do \hspace{1cm} 16 increments
4: for \( k \leftarrow p_{ptr}[j] \) to \( p_{ptr}[j+1] \) do \hspace{1cm} // for each nz in fiber
5: \hspace{1cm} \text{registers} += \text{vals}[k] \times B[j_{index}[k]][r] \hspace{1cm} // buffer
6: \hspace{1cm} A[i][r] += \text{registers} \times C[k_{index}[j]][r] \hspace{1cm} // accumulate
end procedure

2 LD instructions
We use n-D blocking (intuitive) and rank blocking (less intuitive)

- Multi-dimensional blocking
- Rank blocking
We use n-D blocking (intuitive) and rank blocking (less intuitive)

- Multi-dimensional blocking
  - 3D blocking – maximize re-use of both matrix B and C
- Rank blocking
We use n-D blocking (intuitive) and rank blocking (less intuitive)

- Multi-dimensional blocking
  - 3D blocking – maximize re-use of both matrix B and C
- Rank blocking

Make sure this fits in the LLC
We use n-D blocking (intuitive) and rank blocking (less intuitive)

- Multi-dimensional blocking
  - 3D blocking – maximize re-use of both matrix B and C
- Rank blocking
  - Agnostic to tensor sparsity
  - Very little change to the code required

Make sure this fits in the LLC

Increase the chance of finding rows in cache
Rank blocking visualized...

Load 2 rows of B
row 20
row 40
row 30

10,20,30
Scale

10,40,30
Scale

1,000,000

Accum to buffer
Hadamard product

1,000

A'
Accumulate

B
C
Rank blocking visualized...

Load 2 rows of B
row 20
row 40
row 30
row 10
Accumulate

Scale

1,000,000
1,000
1,000,000
10,40,30
10,20,30
Hadamard product
Accum to buffer

Scale
## Performance Summary

<table>
<thead>
<tr>
<th>Data set</th>
<th>Dimensions</th>
<th>nnz</th>
<th>Sparsity</th>
<th># fibers</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poisson1</td>
<td>$256 \times 256 \times 256$</td>
<td>1.5M</td>
<td>$8.8 \times 10^{-2}$</td>
<td>54K</td>
<td>3.1x</td>
</tr>
<tr>
<td>Poisson2</td>
<td>$2K \times 16K \times 2K$</td>
<td>121M</td>
<td>$1.9 \times 10^{-3}$</td>
<td>2.5M</td>
<td>2.5x</td>
</tr>
<tr>
<td>Poisson3</td>
<td>$2K \times 16K \times 2K$</td>
<td>6.4M</td>
<td>$1.0 \times 10^{-4}$</td>
<td>830K</td>
<td>2.0x</td>
</tr>
<tr>
<td>Netflix</td>
<td>$480K \times 18K \times 80$</td>
<td>80M</td>
<td>$1.2 \times 10^{-4}$</td>
<td>5M</td>
<td>2.1x</td>
</tr>
<tr>
<td>NELL-2</td>
<td>$12K \times 9K \times 29K$</td>
<td>77M</td>
<td>$2.4 \times 10^{-5}$</td>
<td>21M</td>
<td>2.2x</td>
</tr>
</tbody>
</table>
Register blocking yields large speedups for small data sets.

![Graph showing speedup comparison between Baseline and Register for different rank sizes (16, 32, 64, 128, 256, 512, 1024). The graph illustrates that Register generally outperforms Baseline, with notable speedup increases as rank size decreases.]
Poisson 2 – sparsity = 1.9e-3
Poisson 3 – sparsity = 1.0e-4
Netflix – sparsity = 1.2e-4
NELL – sparsity = $2.4 \times 10^{-5}$
Distributed rank blocking shows better scalability

<table>
<thead>
<tr>
<th>Nodes</th>
<th>SPLATT</th>
<th>3D grid</th>
<th>3D time</th>
<th>4D grid</th>
<th>4D time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.028</td>
<td>1x1x2</td>
<td>0.718</td>
<td>1x1x1x2</td>
<td>0.826</td>
</tr>
<tr>
<td>2</td>
<td>0.54</td>
<td>1x1x4</td>
<td>0.367</td>
<td>1x1x1x4</td>
<td>0.423</td>
</tr>
<tr>
<td>4</td>
<td>0.286</td>
<td>2x1x4</td>
<td>0.208</td>
<td>1x1x1x8</td>
<td>0.217</td>
</tr>
<tr>
<td>8</td>
<td>0.138</td>
<td>2x2x4</td>
<td>0.107</td>
<td>1x1x1x16</td>
<td>0.124</td>
</tr>
<tr>
<td>16</td>
<td>0.087</td>
<td>2x2x8</td>
<td>0.058</td>
<td>1x1x2x16</td>
<td>0.065</td>
</tr>
<tr>
<td>32</td>
<td>0.056</td>
<td>4x2x8</td>
<td>0.043</td>
<td>1x1x4x16</td>
<td>0.034</td>
</tr>
<tr>
<td>64</td>
<td>0.03</td>
<td>4x4x8</td>
<td>0.028</td>
<td>2x1x4x16</td>
<td>0.022</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPLATT</th>
<th>3D grid</th>
<th>3D time</th>
<th>4D grid</th>
<th>4D time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.025</td>
<td>2x1x1</td>
<td>1.554</td>
<td>1x1x1x2</td>
<td>1.447</td>
</tr>
<tr>
<td>1.158</td>
<td>4x1x1</td>
<td>0.727</td>
<td>1x1x1x4</td>
<td>0.720</td>
</tr>
<tr>
<td>0.519</td>
<td>8x1x4</td>
<td>0.403</td>
<td>1x1x1x8</td>
<td>0.401</td>
</tr>
<tr>
<td>0.256</td>
<td>16x1x1</td>
<td>0.194</td>
<td>1x1x1x16</td>
<td>0.190</td>
</tr>
<tr>
<td>0.113</td>
<td>32x1x1</td>
<td>0.103</td>
<td>1x1x2x16</td>
<td>0.100</td>
</tr>
<tr>
<td>0.083</td>
<td>31x2x1</td>
<td>0.056</td>
<td>1x1x4x16</td>
<td>0.055</td>
</tr>
<tr>
<td>0.048</td>
<td>64x2x1</td>
<td>0.037</td>
<td>2x1x4x16</td>
<td>0.030</td>
</tr>
</tbody>
</table>
The take-away from the section

• There was a lack of clear understanding about performance bottlenecks in tensor decomposition
  • We show that the key computation is LD and memory-bound
• Using various blocking techniques mitigate these bottlenecks
• Our optimizations demonstrate significant speedup over synthetic and real-world data for both shared-memory and distributed implementations
  • We use 3D and rank blocking strategies to achieve up to 3.2x speedup on real world-data and 2.0x on synthetic
Future Work

- Extending this work to do performance modeling
  - Correlate tiling/blocking size to cache hit rate
  - Take advantage of block structures
  - Fiber/slice/cube/etc. permutation – new storage formats for tensors (a la SpMV)
Q & A

I am currently on the academic job market! Please email me at jee@gatech.edu or visit http://jeewhanchoi.com for my application materials.