Blocking Optimizations for Sparse MTTKRP

Jee Choi
CIS, University of Oregon

Workshop on Compiler Techniques for Sparse Tensor Algebra
Cambridge, MA, January 26th, 2019
MTTKRP operation is expensive

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
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
How data is accessed for each non-zero in the tensor

Reduce computation by processing fibers (CSF)

Mode-2 fiber
\( x = 10, z = 30 \)

\( X_{(1)} \)
Reduce computation by processing fibers (CSF)
Roofline model applied to CSF MTTKRP

• Let’s calculate the # of flops and # of bytes and compare
  • Flops: $W = 2R(m + P)$
  • Data: $Q = 2m \text{(value + mode-2 index)} + 2P \text{ (mode-3 index + mode-3 pointer)}$
    + $(1-\alpha)Rm \text{ (mode-2 factor)} + (1-\alpha)RP \text{ (mode-3 factor)}$

• Arithmetic Intensity
  • Ratio of work to communication $I = \frac{W}{Q}$
  • $I = \frac{W}{(Q \ast 8 \text{ Bytes})} = \frac{R}{8 + 4R(1-\alpha)}$

$m = \# \text{ of nonzeros}$
$P = \# \text{ of non-empty fibers}$
$R = \text{rank}$
$\alpha = \text{cache hit rate}$
Arithmetic intensity vs. system balance (on the latest CPU)

Arithmetic Intensity

Perfect cache hit

Cache hit = 0.99

Cache hit = 0.95
Arithmetic intensity vs. system balance (on the latest CPU)

Arithmetic Intensity

1000

100

10

1

0.1

16  32  64  128  256  512  1024  2048

Rank

System balance – 22-core CPU

Perfect cache hit

Cache hit = 0.99

Cache hit = 0.95
Arithmetic intensity vs. system balance (on the latest CPU)

- Perfect cache hit → memory-bound on lower ranks

- System balance – 22-core CPU
  - Cache hit = 0.99
  - Cache hit = 0.95

Graph showing arithmetic intensity vs. system balance with different cache hit ratios and system balance metrics.
Arithmetic intensity vs. system balance (on the latest CPU)

- Perfect cache hit → memory-bound on lower ranks
- Less than perfect cache hit → memory bound for any rank

Perfect cache hit

System balance – 22-core CPU

Cache hit = 0.99

Cache hit = 0.95
Reduce computation by processing fibers (CSF)
A pressure point analysis reveals the bottleneck

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

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

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

Removing per-fiber access to matrix \( C \) has a bigger impact than increasing flops.
Memory access to B is the primary bottleneck

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

Eliminating our suspect has a huge impact.
Completely removing it give us an extra 6% - why?

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

Unexplained 6% decrease in exec. time
Conclusions from our empirical analysis

• Flops aren’t the issue
• Bottlenecks
  1. Data access to factor matrix B (and not the tensor, e.g., SpMV)
  2. Load instructions (why previous attempt at cache blocking was not successful)
Cache/register blocking should help alleviate these bottlenecks

- Flops aren’t the issue
- Bottlenecks
  1. Data access to factor matrix B → cache blocking
  2. Load instructions → register blocking
We use n-D blocking and rank blocking

• Multi-dimensional blocking
  – 3D blocking – maximize re-use of both matrix B and C
  – Multiple access to the factor matrices

Make sure this fits in the LLC
We use n-D blocking and rank blocking

• Multi-dimensional blocking
  – 3D blocking – maximize re-use of both matrix B and C
  – Multiple access to the factor matrices

• Rank blocking
  – Agnostic to tensor sparsity
  – Similar to register blocking
  – Tensor replication
We can combine n-D blocking with rank blocking

• Multi-dimensional + rank blocking
  – Partial replication
  – “Best of both worlds” re-use
  – Even more repeated accesses to tensor/factor
## Performance summary for single node

<table>
<thead>
<tr>
<th>Data set</th>
<th>Dimensions</th>
<th>nnz</th>
<th>Sparsity</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poisson2</td>
<td>$2K \times 16K \times 2K$</td>
<td>121M</td>
<td>1.9e-3</td>
<td>2.0×</td>
</tr>
<tr>
<td>Poisson3</td>
<td>$30K \times 30K \times 30K$</td>
<td>135M</td>
<td>5.0e-6</td>
<td>1.7×</td>
</tr>
<tr>
<td>Netflix</td>
<td>$480K \times 18K \times 80$</td>
<td>80M</td>
<td>1.2e-4</td>
<td>3.1×</td>
</tr>
<tr>
<td>NELL-2</td>
<td>$12K \times 9K \times 29K$</td>
<td>77M</td>
<td>2.4e-5</td>
<td>2.2×</td>
</tr>
<tr>
<td>Reddit</td>
<td>$1.2M \times 23K \times 1.3M$</td>
<td>924M</td>
<td>2.8e-8</td>
<td>2.1×</td>
</tr>
<tr>
<td>Amazon</td>
<td>$4.8M \times 1.8M \times 1.8M$</td>
<td>1.7B</td>
<td>2.5e-8</td>
<td>3.5×</td>
</tr>
</tbody>
</table>
For small tensors, blocking becomes more effective at higher rank sizes

- With small dimension sizes, there is already good cache re-use without explicit blocking
- Only when rank size is large enough, do we see significant benefit from blocking
For large tensors, blocking becomes less effective at higher ranks

- With large dimension sizes and large ranks, data sets are so big, a large number of blocks are required, and the overhead of blocking outweighs the benefit.
More potential benefit from blocking with real data sets

- Real data sets have clustering patterns which lead to higher speedups from blocking
- Combining rank blocking with n-D blocking yields the highest speedup
Rank blocking on distributed systems

- Strong scalability problems with traditional partitioning
  - Fewer non-zero per node -> lower efficiency & higher comm. cost -> poor scalability
Rank blocking on distributed systems

- **Scalability problems**
  - Fewer non-zero per node -> lower efficiency & higher comm. cost -> poor scalability

- **Rank blocking**
  - No comm. between proc. sets
  - Tensor replication
## Rank blocking on distributed systems

<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>
<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>
<td>3.025</td>
<td>2x1x1</td>
<td>1.554</td>
<td>1x1x1x2</td>
<td>1.447</td>
</tr>
<tr>
<td>2</td>
<td>0.540</td>
<td>1x1x4</td>
<td>0.367</td>
<td>1x1x1x4</td>
<td>0.423</td>
<td>1.158</td>
<td>4x1x1</td>
<td>0.727</td>
<td>1x1x1x4</td>
<td>0.720</td>
</tr>
<tr>
<td>4</td>
<td>0.286</td>
<td>2x1x4</td>
<td>0.208</td>
<td>1x1x1x8</td>
<td>0.217</td>
<td>0.519</td>
<td>8x1x1</td>
<td>0.403</td>
<td>1x1x1x8</td>
<td>0.401</td>
</tr>
<tr>
<td>8</td>
<td>0.138</td>
<td>2x2x4</td>
<td>0.107</td>
<td>1x1x1x16</td>
<td>0.124</td>
<td>0.256</td>
<td>16x1x1</td>
<td>0.194</td>
<td>1x1x1x16</td>
<td>0.190</td>
</tr>
<tr>
<td>16</td>
<td>0.087</td>
<td>2x2x8</td>
<td>0.058</td>
<td>1x1x2x16</td>
<td>0.065</td>
<td>0.113</td>
<td>32x1x1</td>
<td>0.103</td>
<td>2x1x1x16</td>
<td>0.100</td>
</tr>
<tr>
<td>32</td>
<td>0.056</td>
<td>4x2x8</td>
<td>0.043</td>
<td>1x1x4x16</td>
<td>0.034</td>
<td>0.083</td>
<td>32x2x1</td>
<td>0.056</td>
<td>4x1x1x16</td>
<td>0.055</td>
</tr>
<tr>
<td>64</td>
<td>0.030</td>
<td>4x4x8</td>
<td>0.028</td>
<td>2x1x4x16</td>
<td>0.022</td>
<td>0.048</td>
<td>64x2x1</td>
<td>0.037</td>
<td>8x1x1x16</td>
<td>0.030</td>
</tr>
</tbody>
</table>