Sparse Tensor Factorization on Many-Core Processors with High-Bandwidth Memory

Shaden Smith\textsuperscript{1*}, Jongsoo Park\textsuperscript{2}, and George Karypis\textsuperscript{1}

\textsuperscript{1}Department of Computer Science & Engineering, University of Minnesota
\textsuperscript{2}Parallel Computing Lab, Intel Corporation
*shaden@cs.umn.edu
Outline

Introduction

Tensor Decomposition

Algorithmic Optimizations for Many-Core Processors

Experiments

Conclusions
Table of Contents

Introduction

Tensor Decomposition

Algorithmic Optimizations for Many-Core Processors

Experiments

Conclusions
Many applications today are *data intensive*:
- Irregular memory accesses.
- Non-uniform work distributions.
- High memory footprints.
- High memory bandwidth demands.

HPC systems are turning to *many-core processors*:
- Hundreds of concurrent threads.
- Emerging architectures feature high-bandwidth memory:
  - Intel Xeon Phi (Knights Landing – KNL)
  - NVIDIA Pascal
  - AMD Fiji
Many-core challenges

We must re-evaluate our algorithms for emerging architectures:

▶ Require $\approx 10 \times$ more parallelism without $10 \times$ more memory.
▶ Fine-grained synchronization is often expensive.
▶ Vectorization is essential for performance.
▶ How to best utilize the high-bandwidth memory?

We evaluate some of the above design directions via:

▶ KNL is our target many-core processor.
▶ Tensor decomposition is our data intensive application.
Intel Knights Landing (KNL)

- Up to 288 concurrent threads (72 cores \(\times\) 4-way SMT)
- 16GB of on-package MCDRAM
  - \(\approx\) 480GB/s memory bandwidth
  - Either managed explicitly or treated as an LLC.

Knights Landing Overview

Chip: 36 Tiles interconnected by 2D Mesh
Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: MCDRAM: 16 GB on-package; High BW DDR4: 6 channels @ 2400 up to 384GB
IO: 36 lanes PCIe Gen3, 4 lanes of DMI for chipset
Node: 1-Socket only
Fabric: Omni-Path on-package (not shown)

Vector Peak Perf: 3+TF DP and 6+TF SP Flops
Scalar Perf: \(~3\times\) over Knights Corner
Streams Triad (GB/s): MCDRAM: 400+; DDR: 90+
Table of Contents

Introduction

Tensor Decomposition

Algorithmic Optimizations for Many-Core Processors

Experiments

Conclusions
Tensors

- Tensors are the generalization of matrices to higher dimensions.
- Allow us to represent and analyze multi-dimensional data.
- Applications in precision healthcare, cybersecurity, recommender systems, ...
The CPD models a tensor as the summation of rank-1 tensors.

- A rank-1 tensor is the outer product of $m$ vectors.

\[
\mathbf{X}(i,j,k) \approx \sum_{f=1}^{F} \mathbf{A}(i,f) \times \mathbf{B}(j,f) \times \mathbf{C}(k,f)
\]

**Notation**

- $\mathbf{A}, \mathbf{B}, \mathbf{C}$, each with $F$ columns, will be used to denote the factor matrices for a 3D tensor.
Alternating least squares (ALS)

ALS cyclically updates one factor matrix at a time while holding all others constant.

**Algorithm 1** CPD-ALS

1: while not converged do
2: \( A^T \leftarrow (C^T C \times B^T B)^{-1} (X_1 (C \circ B))^T \)
3: \( B^T \leftarrow (C^T C \times A^T A)^{-1} (X_2 (C \circ A))^T \)
4: \( C^T \leftarrow (B^T B \times A^T A)^{-1} (X_3 (B \circ A))^T \)

\( \star \) denotes the Hadamard (elementwise) product.

Notation

\( \star \) denotes the Hadamard (elementwise) product.
MTTKRP – elementwise

Elementwise formulation:

$$A(i,:) \leftarrow A(i,:) + X(i,j,k) [B(j,:) \ast C(k,:)]$$

Disclaimer
This is a simplification of how MTTKRP is implemented.
MTTKRP – Parallelism

- Each *slice* of non-zeros can be processed independently.
  - Great, if we store three copies of our tensor ordered by slice.
  - Otherwise we must synchronize on $A(i,:)$, $B(j,:)$, etc.

\[
\begin{align*}
A & \quad \text{i} \\
B & \quad \text{j} \\
C & \quad \text{k}
\end{align*}
\]
When we cannot afford additional tensor representations:

- For $p$ threads, do a $p$-way tiling of each tensor mode.
- Distributing the tiles allows us to eliminate the need for mutexes.
Table of Contents

Introduction

Tensor Decomposition

Algorithmic Optimizations for Many-Core Processors

Experiments

Conclusions
Scalability challenges

Sparse tensors inherit the scalability challenges of sparse matrices:

- Unstructured sparsity patterns.
  - Fine-grained synchronizations and atomics.
- Non-uniform work distributions.
  - *Hub slices* prevent load balanced coarse-grained parallelism.

Tensors also bring unique challenges:

- Mode-centric computations.
  - We cannot always afford to optimize data structures for every mode.
  - *p*-way tiling for higher-order tensors is not practical.
- Mode lengths are highly variable.
  - We may have 1M users but only 5 purchase contexts.
Decomposing Hub Slices

Skewed non-zero distribution can result in load imbalance.

Example:

- A tensor of Amazon product reviews contains a slice with 6.5% of the total non-zeros.
- A 1D decomposition cannot be load balanced with more than 16 threads.
Decomposing Hub Slices

Skewed non-zero distribution can result in load imbalance.

Example:
- A tensor of Amazon product reviews contains a slice with 6.5% of the total non-zeros.
- A 1D decomposition cannot be load balanced with more than 16 threads.

Solution:
- Extract the hub slices and use coarse-grained parallelism for the remaining ones.
- Fine-grained (i.e., non-zero based) parallelism used for hub slices.
Partial tiling

Constructing $p^N$ tiles is not practical for high thread counts ($p$) or tensor dimensionality ($N$).

- Tile a few modes and selectively use mutexes for the remaining ones.
- Writes to $A$ must be synchronized.
- Blocks of $B$ and $C$ are distributed among threads.
Privatization

Short modes will suffer from high lock contention.

- Give each thread a private factor matrix and aggregate at the end.
Table of Contents

Introduction

Tensor Decomposition

Algorithmic Optimizations for Many-Core Processors

Experiments

Conclusions
## Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>NNZ</th>
<th>Dimensions</th>
<th>Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outpatient</td>
<td>87M</td>
<td>1.6M, 6K, 13K, 6K, 1K, 192K</td>
<td>4.1</td>
</tr>
<tr>
<td>Netflix</td>
<td>100M</td>
<td>480K, 18K, 2K</td>
<td>1.6</td>
</tr>
<tr>
<td>Delicious</td>
<td>140M</td>
<td>532K, 17M, 3M</td>
<td>2.7</td>
</tr>
<tr>
<td>NELL</td>
<td>143M</td>
<td>3M, 2M, 25M</td>
<td>2.4</td>
</tr>
<tr>
<td>Yahoo</td>
<td>262M</td>
<td>1M, 624K, 133</td>
<td>4.3</td>
</tr>
<tr>
<td>Reddit</td>
<td>924M</td>
<td>1.2M, 23K, 1.3M</td>
<td>15.0</td>
</tr>
<tr>
<td>Amazon</td>
<td>1.7B</td>
<td>5M, 18M, 2M</td>
<td>36.4</td>
</tr>
</tbody>
</table>
Experimental Setup

Software:
- Modified SPLATT library v1.1.1
- Open source C++ code with OpenMP parallelism
- Modified to use AVX-2 and AVX-512 intrinsics for vectorization.

Knights Landing:
- KNL 7250 at TACC (Stampede).
- 68 cores (up to 272 threads)
- 16GB MCDRAM
- 94GB DDR4

Xeon:
- 2× 22-core Intel Xeon E5- 2699v4 Broadwell
- 2× 55MB last-level cache
- 128GB DDR4
Synchronization primitives

Synchronization overheads on Outpatient.

64B CAS simulates having CAS as wide as an AVX-512 vector.
MCDRAM

Non-zeros use $O(1)$ storage but spawn $O(F)$ accesses to the factors.

- Focus on placing the factors in MCDRAM.
Non-zeros use $O(1)$ storage but spawn $O(F)$ accesses to the factors.

- Focus on placing the factors in MCDRAM.

- Stacked bars encode read-BW (bottom) and write-BW (top).
- KNL’s maximum read-BW out of MCDRAM is 380 GB/s.
Comparison against Broadwell (rank 16)

- Up to 25% speedup over a 44-core Intel Xeon system.
- Managing MCDRAM gives us 30% speedup when dataset is larger than 16GB.
Scaling rank on Yahoo

Tensors with short modes fit inside the large cache of BDW systems.
- KNL is $2 \times$ faster as we scale the CPD rank.

![Chart showing time per MTTKRP operation for BDW and KNL with varying CPD ranks.](chart.png)
Table of Contents

Introduction

Tensor Decomposition

Algorithmic Optimizations for Many-Core Processors

Experiments

Conclusions
Wrapping up

▶ Many-core processors with high-bandwidth memory can accelerate data-intensive applications.
▶ We have to revisit some algorithms to expose parallelism, reduce synchronization, and improve load balance.
▶ Managing MCDRAM can be helpful for large problem sizes.

All of our work is open source:

http://cs.umn.edu/~splatt/
https://github.com/ShadenSmith/splatt-ipdps17

Datasets:

http://frostt.io/
Backup Slides
Compressed sparse fiber (CSF)

- Modes are recursively compressed.
- Paths from roots to leaves encode non-zeros.
- The tree structure encodes opportunities for savings.
MTTKRP with CSF

/* foreach outer slice */
for(int i=0; i < I; ++i) {
    /* foreach fiber in slice */
    for(int s = s_ptr[i]; s < s_ptr[i+1]; ++s) {
        accum[0:r] = 0;

        /* foreach nnz in fiber */
        for(int nnz = f_ptr[s]; nnz < f_ptr[s+1]; ++nnz) {
            int k = f_ids[nnz];
            accum[0:r] += vals[nnz] * C[k][0:r];
        }

        int j = s_ids[s];
        A[i][0:r] += accum[0:r] * B[s][0:r];
    }
}

/* foreach outer slice */
Hub Slices

Load imbalance on the Amazon dataset.

<table>
<thead>
<tr>
<th>Mode</th>
<th>BDW slice</th>
<th>BDW hub</th>
<th>KNL slice</th>
<th>KNL hub</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.72</td>
<td>0.04</td>
<td>0.84</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>0.13</td>
<td>0.04</td>
<td>0.05</td>
<td>0.03</td>
</tr>
<tr>
<td>3</td>
<td>0.07</td>
<td>0.03</td>
<td>0.24</td>
<td>0.18</td>
</tr>
</tbody>
</table>

$\text{imbalance} = \frac{t_{\text{max}} - t_{\text{avg}}}{t_{\text{max}}}.$
Figure: Speedup over untiled MTTKRP while tiling the longest (Tile-1), two longest (Tile-2), and three longest modes (Tile-3).
Figure: Speedup over untiled MTTKRP using one, two, and three tiled modes with privatization for synchronization. Privatized modes were selected with $\gamma=0.2$. 
Figure: Effects of the number of CSF representations on MTTKRP runtime, using 1, 2, and $M$ representations. Amazon is omitted due to memory constraints.