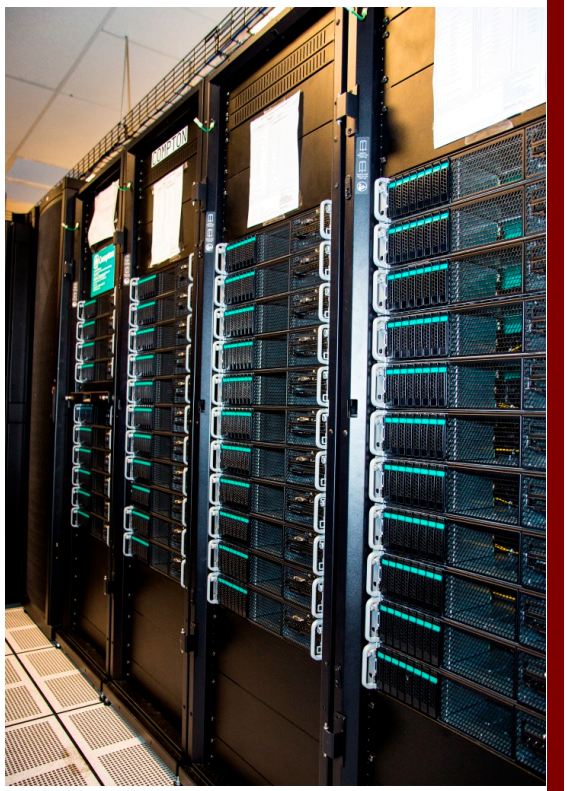
CCR
Center for Computing Research

Tensor Decompositions on Emerging Manycore Hardware with Genten and Kokkos

Eric Phipps (etphipp@sandia.gov)
and Tammy Kolda,
Sandia National Laboratories
Albuquerque New Mexico and Livermore California, USA

SIAM Conference on Computational Science and Engineering

February 25 - March 1, 2019



Sandia National Laboratories

Exceptional service in the national interest

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. SAND2018-1464 C

Genten Project

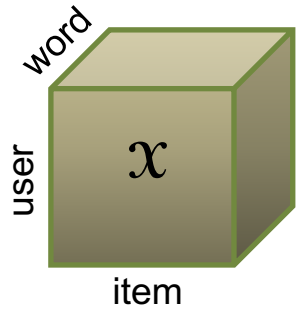
Sandia National Laboratories

- Work funded by LDRD program at Sandia National Labs
  - SNL: T. Kolda (PI), K. Devine, E. Phipps, D. Dunlavy, J. Duersch, C. Anderson-Bergman
  - Georgia Tech: Rich Vuduc, Jeff Young, Srinivas Eswar, Jiajia Li
  - Wake Forest: G. Ballard

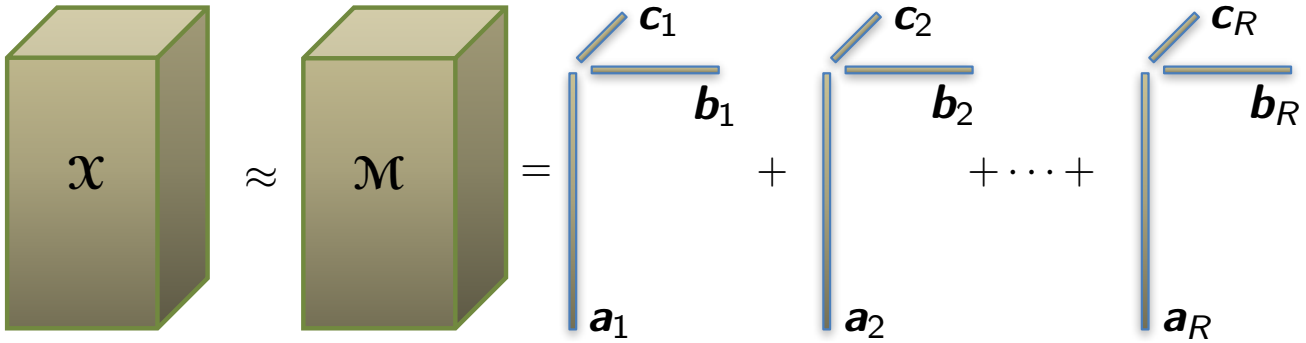
- Goals of the project:
  - Develop sparse tensor decomposition methods for more general tensor data types (Tammy's talk earlier)
  - Develop distributed memory parallel implementations (Karen's talk)
  - Develop shared memory parallel implementations for next-generation computing architectures (this talk)

Tensors

- N-way array used to represent multi-relationship data
  - E.g., word frequencies in Amazon product rankings:



- Canonical Polyadic (CP) tensor decomposition
  - Approximate tensor as a sum of rank-1 tensors
  - Discovers dominant relationships in data



$$\mathcal{X} \approx \mathcal{M} = \boldsymbol{a}_1 \circ \boldsymbol{b}_1 \circ \boldsymbol{c}_1 + \boldsymbol{a}_2 \circ \boldsymbol{b}_2 \circ \boldsymbol{c}_2 \cdots + \boldsymbol{a}_R \circ \boldsymbol{b}_R \circ \boldsymbol{c}_R$$

$$x(i,j,k) \approx m(i,j,k) = \sum_{l=1}^{R} a(i,l)b(j,l)c(k,l)$$

CP via Alternating Least-Squares (ALS)

- CP minimization problem
  - This assumes *Gaussian model* for tensor, resulting in least-squares

$$\min_{\mathcal{M}} \ \|\mathcal{X} - \mathcal{M}\|_F^2 \ \text{ s.t. } \ \mathcal{M} = \boldsymbol{a}_1 \circ \boldsymbol{b}_1 \circ \boldsymbol{c}_1 + \cdots + \boldsymbol{a}_R \circ \boldsymbol{b}_R \circ \boldsymbol{c}_R$$

- Solve via alternating linear least squares
  - Fix all but one term, solve linear least squares problem, iterate

Using…

$$\mathcal{M}_{(1)} = \boldsymbol{A}(\boldsymbol{C} \odot \boldsymbol{B})^T$$
$$\mathcal{M}_{(2)} = \boldsymbol{B}(\boldsymbol{C} \odot \boldsymbol{A})^T$$
$$\mathcal{M}_{(3)} = \boldsymbol{C}(\boldsymbol{B} \odot \boldsymbol{A})^T$$

Repeat until convergence…

$$\min_{\boldsymbol{A}} \left\|\mathcal{X}_{(1)} - \boldsymbol{A}(\boldsymbol{C} \odot \boldsymbol{B})^T\right\|_F^2$$
$$\min_{\boldsymbol{B}} \left\|\mathcal{X}_{(2)} - \boldsymbol{B}(\boldsymbol{C} \odot \boldsymbol{A})^T\right\|_F^2$$
$$\min_{\boldsymbol{C}} \left\|\mathcal{X}_{(3)} - \boldsymbol{C}(\boldsymbol{B} \odot \boldsymbol{A})^T\right\|_F^2$$

Repeat until convergence…

$$\boldsymbol{A} = \mathcal{X}_{(1)}(\boldsymbol{C} \odot \boldsymbol{B})(\boldsymbol{C}^T\boldsymbol{C} * \boldsymbol{B}^T\boldsymbol{B})^\dagger$$
$$\boldsymbol{B} = \mathcal{X}_{(2)}(\boldsymbol{C} \odot \boldsymbol{A})(\boldsymbol{C}^T\boldsymbol{C} * \boldsymbol{A}^T\boldsymbol{A})^\dagger$$
$$\boldsymbol{C} = \underbrace{\mathcal{X}_{(3)}(\boldsymbol{B} \odot \boldsymbol{A})}_{\text{MTTKRP}}(\boldsymbol{B}^T\boldsymbol{B} * \boldsymbol{A}^T\boldsymbol{A})^\dagger$$

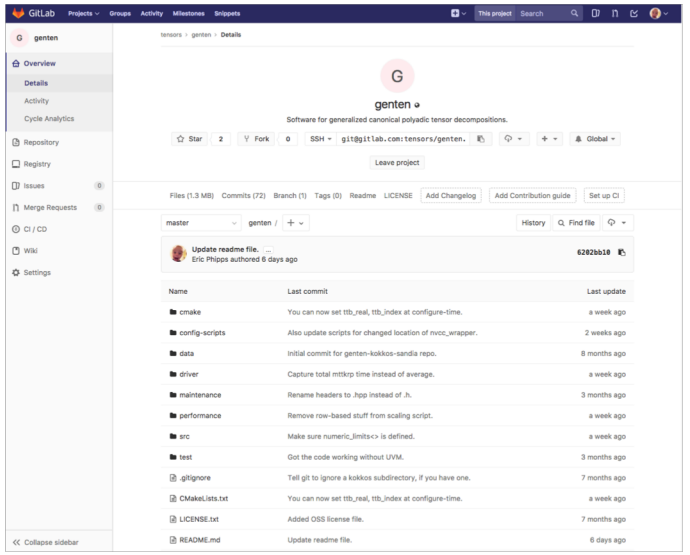
- Matricized Tensor Times Khatri-Rao Product (MTTKRP):
  - Most expensive part of CP-ALS

$$\boldsymbol{V} = \mathcal{X}_{(3)}(\boldsymbol{A} \odot \boldsymbol{B}) \implies v(i,l) = \sum_{(i,j,k)\in\mathcal{N}(\mathcal{X})} x(i,j,k)a(j,l)b(k,l), \quad l = 1,\ldots,R$$

Genten : Software for Generalized Canonical Polyadic Tensor Decompositions

Sandia National Laboratories

- New software package Genten developed at SNL
  - E. Phipps, T. Kolda, D. Dunlavy, G. Ballard, T. Plantenga
  - Based on C++ port of Matlab Tensor Toolbox
  - Publicly available at https://gitlab.com/tensors/genten
  - Implements full CP-ALS algorithm for sparse tensors
- Incorporates shared memory parallelism for emerging manycore hardware using Kokkos
  - Multicore CPUs via OpenMP, pThreads
  - GPUs via Nvidia Cuda and AMD ROCm
  - Intel Xeon Phi (a.k.a. KNC/KNL) via OpenMP
- Implements parallelism for all performance-critical operations
  - MTTKRP, tensor inner product, norms, ...
  - Can use optimized third-party libraries (MKL, cuBLAS, ...)
  - Natively handles data transfers between CPU, GPU memory



What is Kokkos?

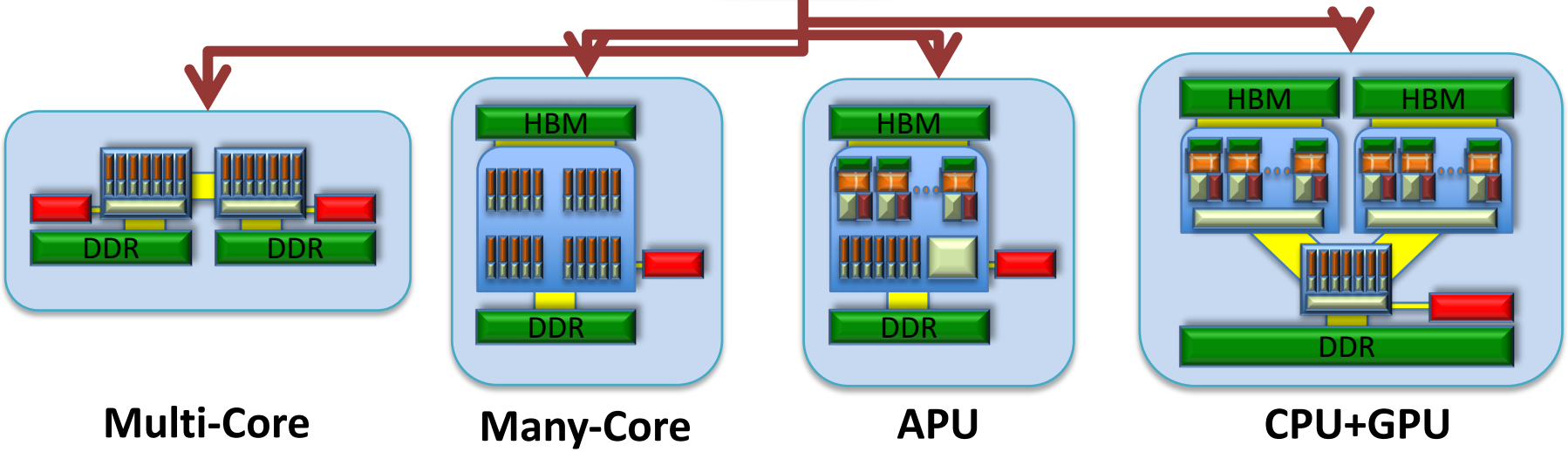
Sandia National Laboratories

LAMMPS
Albany
Drekar
EMPIRE
SPARC
SIERRA
Trilinos
etc...

Applications & Libraries

Kokkos

performance portability for C++ applications



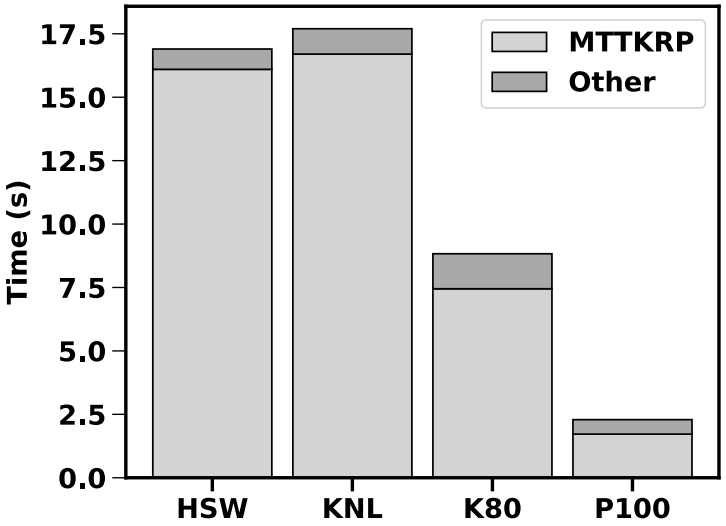
Multi-Core **Many-Core** **APU** **CPU+GPU**

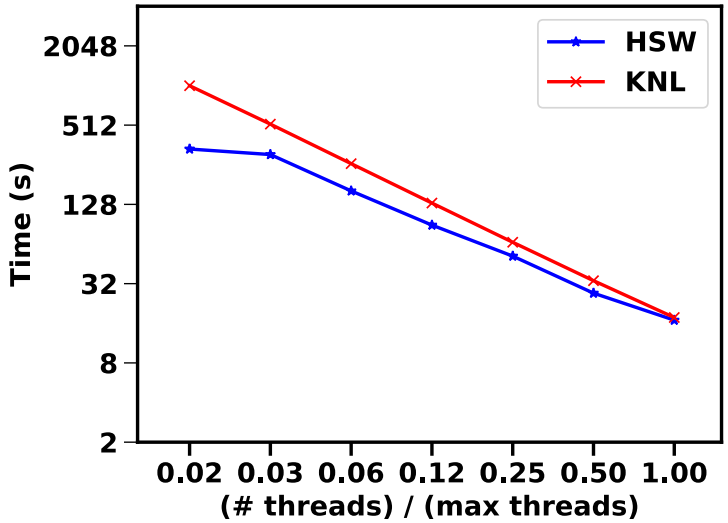
H.C. Edwards, C. Trott, *et al.*, https://github.com/kokkos/kokkos

Genten CP-ALS Performance*

- Synthetic data tensor (double precision)
  - 30K x 40K x 50K, 10M non-zeros, R = 128 (perf_CpAlsRandomKtensor performance test in Genten)
  - 10 CP-ALS iterations

- Architectures
  - HSW (OpenMP): Intel Xeon E5-2698v3 CPU, 2.3 GHz, 2 sockets, 16 cores/socket, 2 threads/core
  - KNL (OpenMP): Intel Xeon Phi 7250, 68 cores (using up to 64), 4 threads/core, 16 GB HBM in cache mode
  - K80 (Cuda): Nvidia Kepler K80 GPU, 12 GB GDDR5
  - P100 (Cuda): Nvidia Pascal P100 GPU, 16 GB HBM





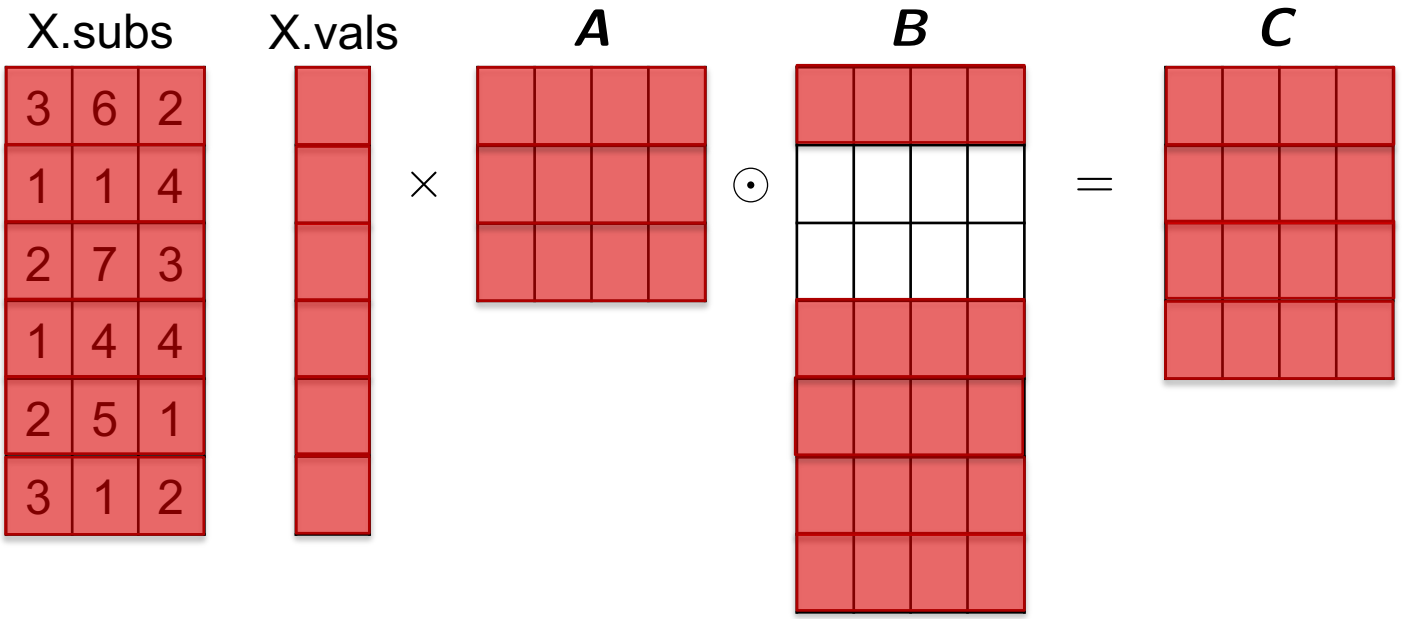
*E. Phipps and T. Kolda, *Software for Sparse Tensor Decomposition on Emerging Computing Architectures,* submitted to SIAM SISC, arXiv.org:1809.09175.

MTTKRP

- Sparse tensors stored in coordinate (COO) format
  - List of nonzero values and coordinates (subscripts)

$$\mathcal{X} \in \mathbb{R}^{3\times 7\times 4}, \quad R = 4$$

$$\boldsymbol{C} = \mathcal{X}_{(3)}(\boldsymbol{A} \odot \boldsymbol{B}) \implies c(k,l) = \sum_{(i,j,k)\in\mathcal{N}(\mathcal{X})} x(i,j,k)a(i,l)b(j,l), \quad l = 1,\ldots,R$$



Serial MTTKRP Algorithm

```
Sptensor X;
Ktensor u;
FacMatrix v;

const int nnz = X.nnz();
for (int i=0; i<nnz; ++i)
{
  double *tmp = new double[R];

  const size_t k = X.subscript(i,n);
  const double x_val = X.value(i);
  for (int j = 0; j < R; ++j)
    tmp[j] = x_val;

  for (int m = 0; m < N; m++) {
    if (m != n) {
      const double *row = u[m].rowptr(X.subscript(i,m));
      for (int j = 0; j < R; ++j)
        tmp[j] *= row[j];
    }
  }

  for (int j = 0; j < R; ++j)
    v.entry(k,j) += tmp[j];

  delete [] tmp;
}
```

Matricized Tensor Times Khatri-Rao Product (MTTKRP)

- Factor matrices stored row-wise (Kokkos::LayoutRight)

$$\boldsymbol{V} = \boldsymbol{\mathcal{X}}_{(n)} \left(\boldsymbol{U}^N \odot \cdots \odot \boldsymbol{U}^{n+1} \odot \boldsymbol{U}^{n-1} \odot \cdots \odot \boldsymbol{U}^1 \right)$$

$$v(i_n, :) = \sum_{i \in \mathcal{N}(\boldsymbol{\mathcal{X}})} x_i u^1(i_1, :) \dots u^{n-1}(i_{n-1}, :) u^{n+1}(i_{n+1}, :) \dots u^N(i_N, :)$$

Flat Parallelism Over Tensor Non-zeros

```
typedef Kokkos::RangePolicy<ExecSpace> Policy;
const int nnz = X.nnz();
Policy policy(0,nnz);
Kokkos::parallel_for( policy,
                      KOKKOS_LAMBDA(const int i)
{
  double *tmp = new double[R];

  const size_t k = X.subscript(i,n);
  const double x_val = X.value(i);

  for (int j = 0; j < R; ++j)
    tmp[j] = x_val;

  for (int m = 0; m < N; m++) {
    if (m != n) {
      const double *row = u[m].rowptr(X.subscript(i,m));
      for (int j = 0; j < R; ++j)
        tmp[j] *= row[j];
    }
  }

  for (int j = 0; j < R; ++j)
    Kokkos::atomic_add(&v.entry(k,j), tmp[j]);

  delete [] tmp;
});
```

Parallelize MTTKRP over tensor non-zeros

Multiple threads writing to same row of *V* requires atomic update (not the only way to do it)

Flat Parallelism Over Tensor Non-zeros

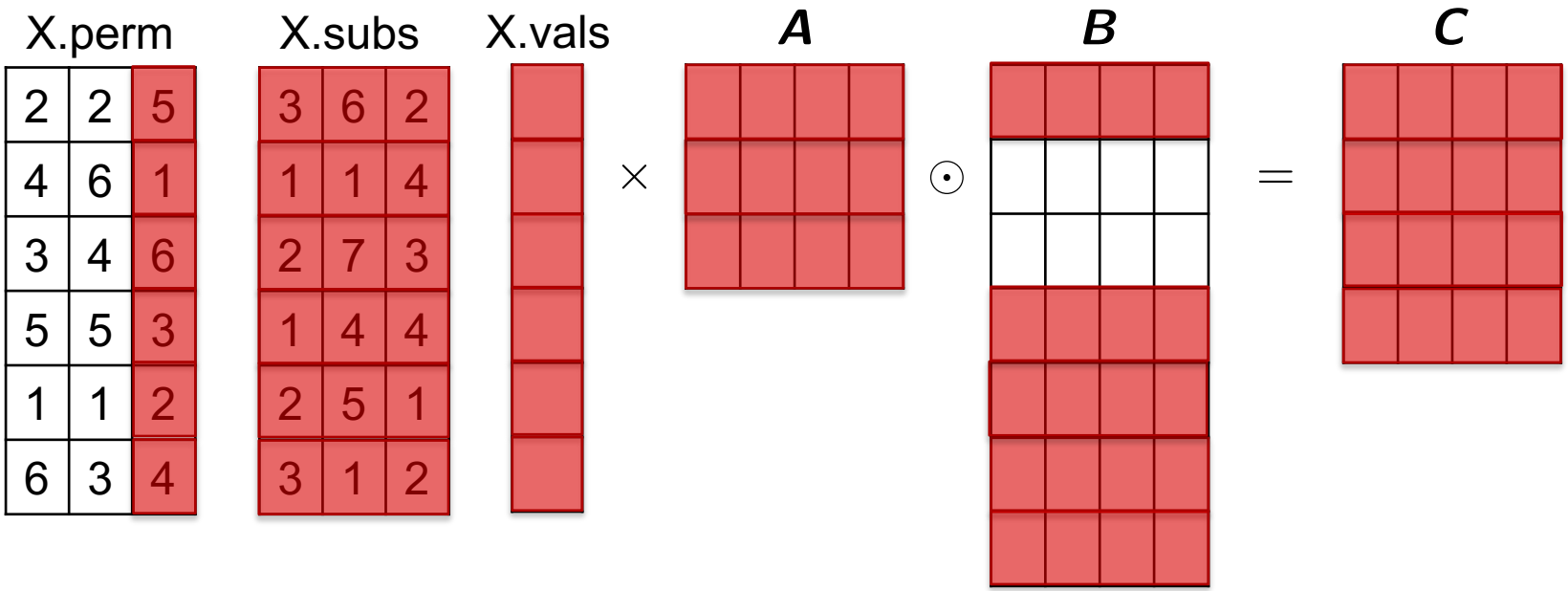
```
typedef Kokkos::RangePolicy<ExecSpace> Policy;
const int nnz = X.nnz();
Policy policy(0,nnz);
Kokkos::parallel_for( policy,
                      KOKKOS_LAMBDA(const int i)
{
  double *tmp = new double[R];

  const size_t k = X.subscript(i,n);
  const double x_val = X.value(i);

  for (int j = 0; j < R; ++j)
    tmp[j] = x_val;

  for (int m = 0; m < N; m++) {
    if (m != n) {
      const double *row = u[m].rowptr(X.subscript(i,m));
      for (int j = 0; j < R; ++j)
        tmp[j] *= row[j];
    }
  }

  for (int j = 0; j < R; ++j)
    Kokkos::atomic_add(&v.entry(k,j), tmp[j]);

  delete [] tmp;
});
```

Improvements in MTTKRP implementation:

- Thread teams to process in parallel:
  - Tensor non-zeros
  - Factor matrix columns
- Thread-scalable mechanism for tmp buffer
  - Kokkos scratch-pad memory (MTTKRP-A)
  - Thread-local, compile-time polymorphic arrays (MTTKRP-B)
- Blocking on R for better cache performance, occupancy
- Experimenting with alternatives to atomics
  - Thread privatization buffers

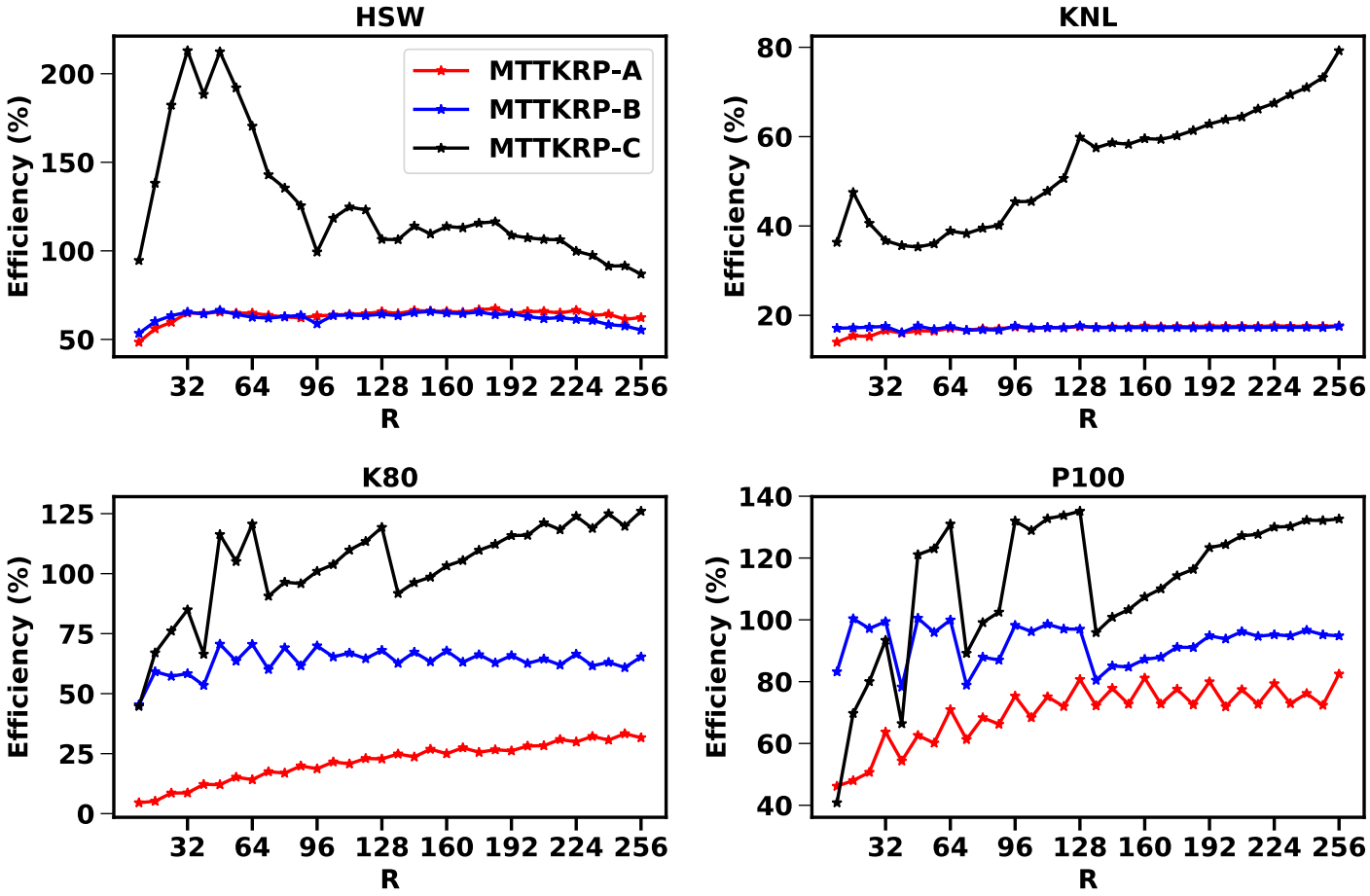
Permuted MTTKRP Inspired by COO Mat-Vec

- Poorly structured tensors result in high atomic contention
  - COO sparse matrix-vector product leverages sorting by increasing row-index
- Apply this to coordinate-based sparse tensor format
  - Don't want multiple copies of tensor with different sortings
  - Instead, compute permutation array p so that X.subscript(p(:,n),n) is increasing
  - Each thread only writes when row index X.subscript(p(:,n),n) changes
  - Substantially reduces number of atomic updates and improves their locality



MTTKRP Percentage of Peak Bandwidth

$$\text{GB/s} = \Big((NR + 3) \times sizeof(\text{Real}) + N \times sizeof(\text{Ordinal}) \Big) \times \frac{NNZ}{t}$$



- MTTKRP-A: Kokkos scratch-pad arrays
- MTTKRP-B: Compile-time polymorphic arrays
- MTTKRP-C: Permutation-based MTTKRP

Performance on Real Tensors

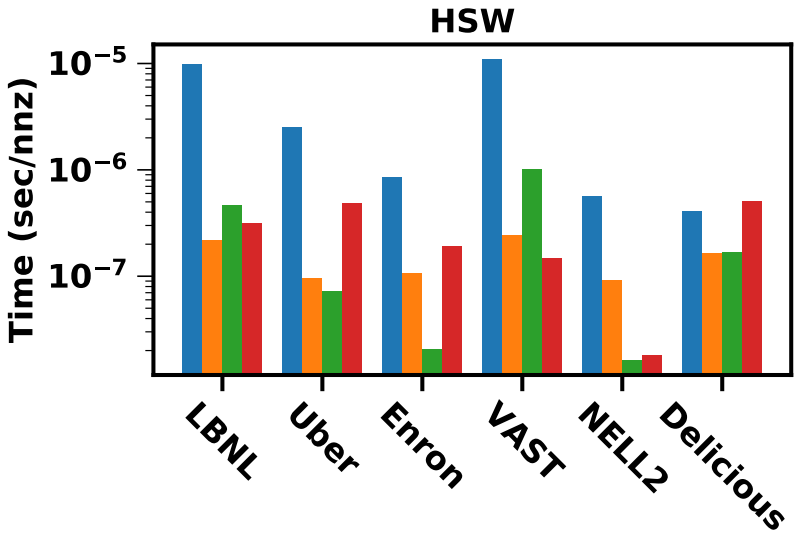
Sandia National Laboratories

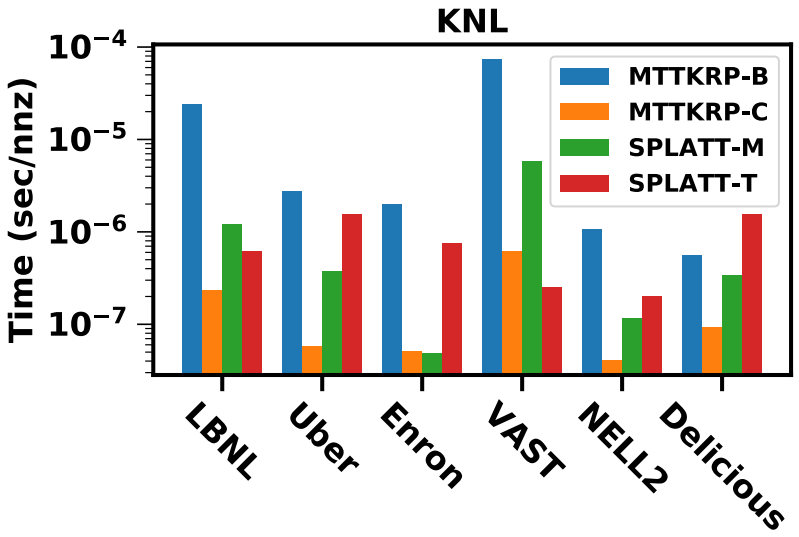
- Look at Genten MTTKRP performance on several tensors from the FROSTT collection
  - Shaden Smith et al, https://frost.io
  - Fixed 10 iterations of CP-ALS
  - R =16 for all tests

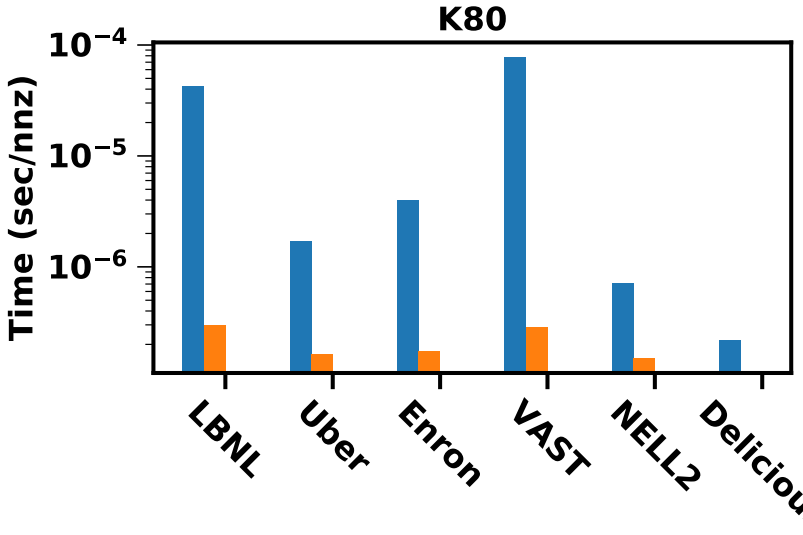
Name	Order	Dimensions	Nonzeros
LBNL	5	$1.6\text{K} \times 4.2\text{K} \times 1.6\text{K} \times 4.2\text{K} \times 868\text{K}$	1.7M
Uber	4	$183 \times 24 \times 1.1\text{K} \times 1.7\text{K}$	13M
Enron	4	$6.0\text{K} \times 5.7\text{K} \times 244\text{K} \times 1.2\text{K}$	54M
VAST	5	$165\text{K} \times 11\text{K} \times 2 \times 100 \times 89$	26M
NELL2	3	$12\text{K} \times 9.1\text{K} \times 29\text{K}$	77M
Delicious	4	$532\text{K} \times 17\text{M} \times 2.5\text{M} \times 1.4\text{K}$	140M

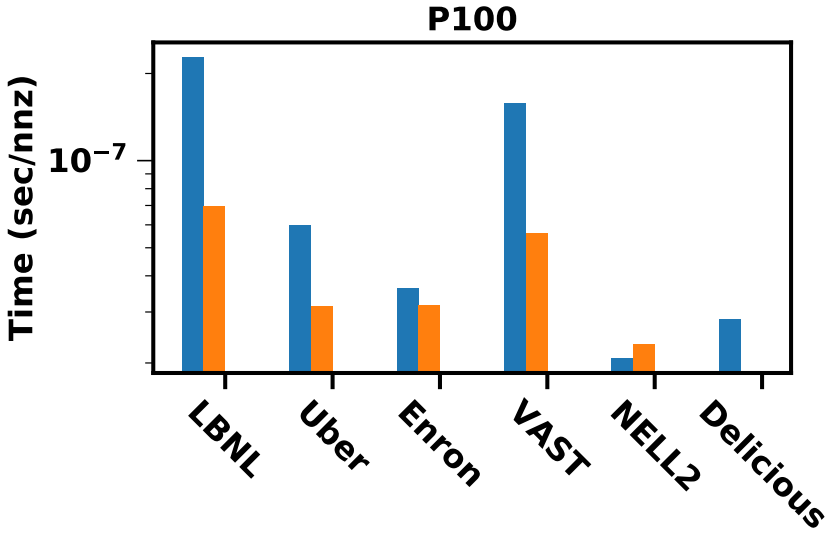
- Compare MTTKRP-B/C to SPLATT on OpenMP architectures
  - Shaden Smith et al, http://glaros.dtc.umn.edu/gkhome/splatt/overview
  - Compressed Sparse Fiber (CSF) storage format (Smith and Karypis, 2015)
  - Use default 2 CSF modes, with and without tiling

Total MTTKRP Time per Nonzero









Summary & Conclusions

Sandia National Laboratories

- New software package Genten
  - Publicly available at https://gitlab.com/tensors/genten
  - POC: Eric Phipps, etphipp@sandia.gov
- Built on Kokkos for shared-memory parallelism and high performance, portable to emerging parallel architectures
- Simple modification of standard MTTKRP algorithm to store and iterate over permutation arrays improves performance
- Future Plans
  - Continue to improve performance (MTTKRP, sorting, ...)
  - Kokkos implementation of state-of-the-art formats such as CSF and HiCOO
  - Incorporate generalized tensor decomposition R&D by Kolda et al for general tensor data types
  - Incorporate distributed memory parallelism (K. Devine)

Acknowledgments

- Thanks to T. Kolda and others for many helpful comments!

- This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

Sandia National Laboratories

Backup Slides

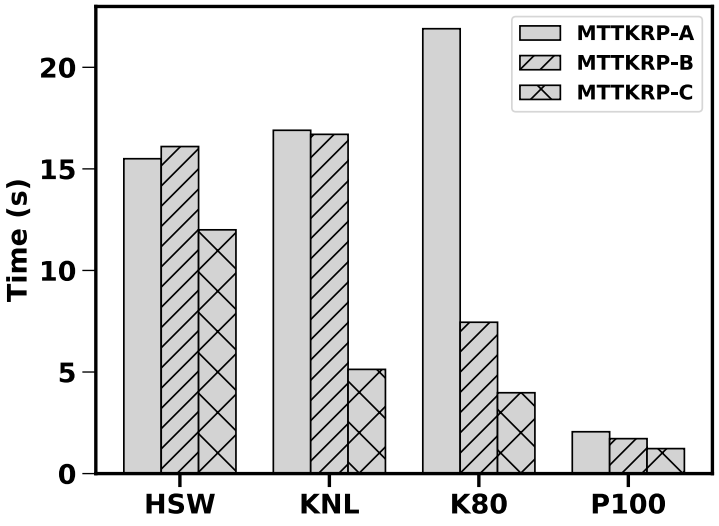
Improving MTTKRP Performance

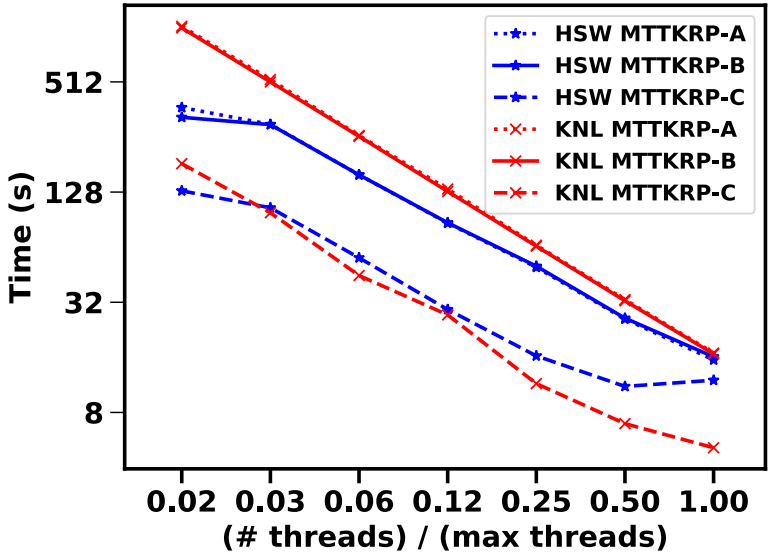
Sandia National Laboratories

- Problems with current algorithm
  - High atomic contention if tensor is poorly structured
  - Poor-locality in writes to rows of result factor matrix
- Take inspiration from coordinate-based (COO) sparse mat-vec
  - Nonzero indices are sorted by increasing row index
  - Allows marching down the rows, improving locality
  - Reduce partial sums across non-zeros with the same row index to reduce atomic writes
- Apply this to coordinate-based sparse tensor format
  - For each mode n, compute permutation array p so that X.subscript(p(:,n),n) is increasing
    - Requires N sorting operations at setup (cost of a few MTTKRP)
    - Store 2-D permutation array same size as coordinates (~2x increased memory usage)
  - MTTKRP algorithm:
    - Each thread team loops over blocks of non-zeros in p(:) order
    - Accumulate result until change in row index
    - Atomic write if first or last row in block, otherwise normal write
- *Substantially reduces number of atomic updates and improves their locality*

Permuted MTTKRP Performance

- MTTKRP-A: Team-based MTTKRP algorithm w/Kokkos scratch-pad arrays
  - Store tmp buffer in shared memory on GPU
- MTTKRP-B: Team-based MTTKRP algorithm w/compile-time polymorphic arrays
  - Store tmp buffer in registers on GPU
- MTTKRP-C: Team-based MTTKRP w/permutation arrays and compile-time polymorphic arrays for tmp buffers





Total MTTKRP Time

Sandia National Laboratories

Name	Order	Dimensions	Nonzeros
LBNL	5	1.6K × 4.2K × 1.6K × 4.2K × 868K	1.7M
Uber	4	183 × 24 × 1.1K × 1.7K	13M
Enron	4	6.0K × 5.7K × 244K × 1.2K	54M
VAST	5	165K × 11K × 2 × 100 × 89	26M
NELL2	3	12K × 9.1K × 29K	77M
Delicious	4	532K × 17M × 2.5M × 1.4K	140M

- MTTKRP-B: MTTKRP w/compile-time polymorphic arrays
- MTTKRP-C: Permutation-based MTTKRP

Arch.	Method	LBNL	Uber	Enron	VAST	NELL2	Delicious
HSW	MTTKRP-B	16.6	8.3	45.7	284.0	42.8	57.4
	MTTKRP-C	0.4	0.3	5.7	6.3	7.0	22.7
	SPLATT-M	0.8	0.2	1.1	26.1	1.3	23.4
	SPLATT-T	0.5	1.6	10.4	3.8	1.4	71.2
KNL	MTTKRP-B	41.2	9.2	108.0	1910.0	82.4	79.1
	MTTKRP-C	0.4	0.2	2.8	16.1	3.2	12.9
	SPLATT-M	2.0	1.2	2.6	150.5	9.0	47.7
	SPLATT-T	1.0	5.1	41.3	6.5	15.6	219.1
K80	MTTKRP-B	72.4	5.6	218.0	2010.0	55.2	30.3
	MTTKRP-C	0.5	0.5	9.4	7.4	11.6	–
P100	MTTKRP-B	0.4	0.2	2.0	4.1	1.6	4.0
	MTTKRP-C	0.1	0.1	1.7	1.5	1.8	–

Tensor Sorting Cost

- Permutation method requires *N* sorting operations
  - Sort tensor coordinates for each mode
  - Small, but not inconsequential cost

- Sorting methods used
  - OpenMP: Parallel stable sort from Intel
  - Cuda: Thrust stable sort

Sorting time divided by avg. (permuted) CP-ALS iteration time

Architecture	LBNL	Uber	Enron	VAST	NELL2	Delicious
HSW	0.6	4.3	9.2	5.3	6.2	4.0
KNL	1.0	5.5	8.6	1.3	7.6	4.9
K80	1.1	3.0	3.3	3.3	2.6	–
P100	0.5	1.3	4.2	3.6	4.6	–

CP via Alternating Least-Squares (ALS)

Sandia National Laboratories

For least-squares error model (Gaussian approximation), solve minimization by fixing all but one term, then iterate

Input: $\mathcal{X}$, R, $MaxIts$, tol

1: Initialize $\boldsymbol{A}^{(1)}, \ldots, \boldsymbol{A}^{(N)}$
2: $i \leftarrow 0$
3: **repeat**
4: **for** $n = 1, \ldots, N$ **do**
5: $\boldsymbol{V}^{(n)} \leftarrow \left(\boldsymbol{A}^{(1)}\right)^T \boldsymbol{A}^{(1)} * \cdots * \left(\boldsymbol{A}^{(n-1)}\right)^T \boldsymbol{A}^{(n-1)} * \left(\boldsymbol{A}^{(n+1)}\right)^T \boldsymbol{A}^{(n+1)} * \cdots * \left(\boldsymbol{A}^{(N)}\right)^T \boldsymbol{A}^{(N)}$ // Gramian
6: $\boldsymbol{B}^{(n)} \leftarrow \mathcal{X}_{(n)} \left(\boldsymbol{A}^{(N)} \odot \cdots \odot \boldsymbol{A}^{(n+1)} \odot \boldsymbol{A}^{(n-1)} \odot \cdots \odot \boldsymbol{A}^{(1)}\right)$ // MTTKRP
7: $\boldsymbol{A}^{(n)} \leftarrow \boldsymbol{B}^{(n)} \left(\boldsymbol{V}^{(n)}\right)^{\dagger}$ // Solve
8: $\lambda_k \leftarrow \|\boldsymbol{a}_k^{(n)}\|, \; k = 1, \ldots, R$ // ColumnNorm
9: $\boldsymbol{A}^{(n)} \leftarrow \boldsymbol{A}^{(n)} \boldsymbol{\Lambda}^{-1}$ // ColumnScale
10: **end for**
11: $i \leftarrow i + 1$
12: **until** $i \geq MaxIts$ **or** change in $\left\|\mathcal{X} - [\![\boldsymbol{\lambda}; \boldsymbol{A}^{(1)}, \ldots, \boldsymbol{A}^{(N)}]\!]\right\| < tol$

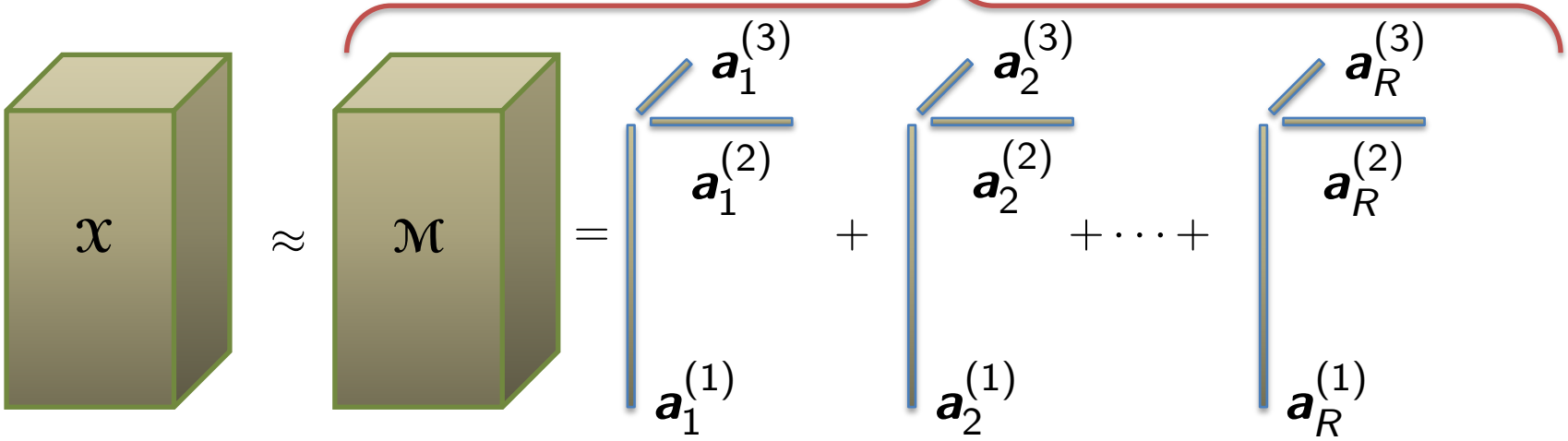
Output: $\boldsymbol{\lambda}$, $\boldsymbol{A}^{(1)}, \ldots, \boldsymbol{A}^{(N)}$

Canonical Polyadic (CP) Decomposition

Approximate tensor as a sum of rank-1 tensors to find dominant relationships

Data

CP Model



$$\mathcal{X}_{i_1,i_2,\ldots,i_N} \approx \mathcal{M}_{i_1,i_2,\ldots,i_N} = \sum_{j=1}^{R} \lambda_j \boldsymbol{a}_{i_1,j}^{(1)} \boldsymbol{a}_{i_2,j}^{(2)} \cdots \boldsymbol{a}_{i_N,j}^{(N)}$$

$$\min_{\mathcal{M}} \quad \|\mathcal{X} - \mathcal{M}\|$$

$$\text{s.t.} \quad \mathcal{M} = [\![\boldsymbol{\lambda}; \boldsymbol{A}^{(1)}, \boldsymbol{A}^{(2)}, \ldots, \boldsymbol{A}^{(N)}]\!] \quad , \quad \boldsymbol{A}^{(n)} = [\boldsymbol{a}_1^{(n)}, \ldots, \boldsymbol{a}_R^{(n)}]$$

What is Kokkos?

Sandia National Laboratories

- **ΚΌΚΚΟΣ** (Greek, not an acronym)
  - Translation: "granule" or "grain" ; *like grains of sand on a beach*
  - H.C. Edwards, C. Trott*, et al*
- Performance Portable Thread-Parallel Programming Model
  - E.g., "X" in "MPI+X" ; not a distributed-memory programming model
  - Application identifies its parallelizable grains of computations *and* data
  - Kokkos maps those computations onto cores *and* that data onto memory
- Fully Performance Portable C++11 Library Implementation
  - *Not* a language extension (e.g., OpenMP, OpenACC, OpenCL, …)
  - Production – open source at https://github.com/kokkos/kokkos
    - ✓ Compilers: GNU, LLVM, Intel, NVIDIA, IBM XL, Cray
    - ✓ Multicore CPU - including NUMA architectural concerns
    - ✓ Intel Xeon Phi (KNC/KNL)
    - ✓ NVIDIA GPU (Kepler, Pascal, Volta)
    - ✓ IBM Power 8/9
    - ○ AMD Fusion

Advantages of Kokkos (my view)

Sandia National Laboratories

- Tools to build portable, thread-parallel algorithms with good performance
  - Tools to manage memory allocations and multiple memory spaces
  - Ability to introduce data layout polymorphism
  - Simple, portable interfaces to advanced architecture-specific capabilities
  - Light-weight abstractions that don't impede performance
- Future-proofing
  - Substantial buy-in from DOE and vendors
  - Expected to be part of C++-2x standard
- Flexibility to write directly to the underlying programming model
  - Can always write specializations specific to an architecture if necessary/desired

Kokkos Abstractions

Sandia National Laboratories

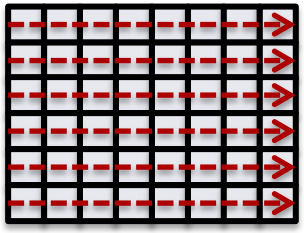
- Parallel Pattern of user's computations
  - parallel_for, parallel_reduce, parallel_scan, task-graph, ... *(extensible)*
- Execution Policy tells *how* user computation will be executed
  - Static scheduling, dynamic scheduling, thread-teams, ... *(extensible)*
- Execution Space tells *where* user computations will execute
  - Which cores, numa region, GPU, ... *(extensible)*
- Memory Space tells *where* user data resides
  - Host memory, GPU memory, high bandwidth memory, ... *(extensible)*
- Layout (policy) tells *how* user data is laid out in memory
  - Row-major, column-major, array-of-struct, struct-of-array … *(extensible)*
- Differentiating: Layout and Memory Space
  - Versus other programming models (OpenMP, OpenACC, …)
  - Critical for performance portability …

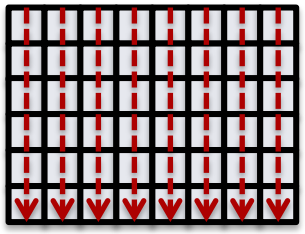
Layout Abstraction: Multidimensional Array

Sandia National Laboratories

- **Classical (50 years!) data pattern for science & engineering codes**
  - Computer languages hard-wire multidimensional array layout mapping
  - Problem: different architectures *require* different layouts for performance
  - ➢ **Leads to architecture-specific versions of code to obtain performance**
  - E.g., "Array of Structure" ↔ "Structure of Array" redesigns

e.g., "row-major" CPU caching





e.g., "column-major" GPU coalescing

- **Kokkos *separates* layout from user's computational code**
  - *Choose* layout for architecture-specific memory access pattern
    - ➢ **Without modifying user's computational code**
  - **Polymorphic** layout via C++ template meta-programming *(extensible)*
    - ➢ **e.g., Hierarchical Tiling layout** (array of structure of array)
- **Bonus: easy/transparent use of special data access hardware**
  - Atomic operations, GPU texture cache, ... *(extensible)*

Performance Portability & Future Proofing

Sandia National Laboratories

Integrated mapping of users' parallel computations *and* data through abstractions of patterns, policies, spaces, *and* layout.

- **Versus other thread parallel programming models (mechanisms)**
  - OpenMP, OpenACC, OpenCL, ... have parallel execution
  - OpenMP 4 finally has execution spaces; when memory spaces ??
  - ➢ **All of these neglect data layout mapping**
    - Requiring significant code refactoring to change data access patterns
    - Cannot provide *performance* portability
  - ➢ **All require language and compiler changes for extension**
- **Kokkos extensibility "future proofing" for evolving architectures**
  - Library extensions, not compiler extensions
  - E.g., Intel KNL high bandwidth memory ← just another memory space
- **Productivity versus other programming models?**

Flat Parallelism Over Tensor Non-zeros

Sandia National Laboratories

```
typedef Kokkos::RangePolicy<ExecSpace> Policy;
const int nnz = X.nnz();
Policy policy(0,nnz);
Kokkos::parallel_for( policy,
                      KOKKOS_LAMBDA(const int i)
{
  double *tmp = new double[R];

  const size_t k = X.subscript(i,n);
  const double x_val = X.value(i);

  for (int j = 0; j < R; ++j)
    tmp[j] = x_val;

  for (int m = 0; m < N; m++) {
    if (m != n) {
      const double *row = u[m].rowptr(X.subscript(i,m));
      for (int j = 0; j < R; ++j)
        tmp[j] *= row[j];
    }
  }

  for (int j = 0; j < R; ++j)
    Kokkos::atomic_add(&v.entry(k,j), tmp[j]);

  delete [] tmp;
});
```

Dynamic memory allocation (very bad on GPU)

Not exploiting parallelism over columns of factor matrices

Kokkos Hierarchical Parallelism

Sandia National Laboratories

- Kokkos maps a league of teams across a given index range
  - Each team consists of a given number of “threads”
  - Each “thread” can iterate over a “vector” range
- Team:
  - CPU/Phi: One or more cores
  - GPU: Cuda thread block
- Thread:
  - CPU/Phi: Hyperthread in a core
  - GPU: Warp in a thread block
- Vector:
  - CPU/Phi: Vector lane in a SIMD instruction
  - GPU: Thread within a warp

Adding Scratch Space and Vector Parallelism

Sandia National Laboratories

```
typedef Kokkos::TeamPolicy<ExecSpace> Policy;
const int nnz = X.nnz();
const int VectorSize = 8;
const int TeamSize = 1;
Policy policy(nnz,TeamSize,VectorSize);

typedef Kokkos::View< double*,
  ExecSpace::scratch_memory_space,
  Kokkos::MemoryUnmanaged > ScratchSpace;
const size_t bytes = ScratchSpace::shmem_size(R);

Kokkos::parallel_for(
  policy.set_scratch_size(0,Kokkos::PerTeam(bytes)),
  KOKKOS_LAMBDA(Policy::member_type team)
{
  const size_t i = team.league_rank();

  ScratchSpace tmp(team.team_scratch(0), R);

  const size_t k = X.subscript(i,n);
  const double x_val = X.value(i);
```

```
  Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                       [&] (const int r)
  {
    tmp[r] = x_val;
  });

  for (int m = 0; m < N; ++m) {
    if (m != n) {
      const double *row = u[m].rowptr(X.subscript(i,m));
      Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                           [&] (const int r)
      {
        tmp[r] *= row[r];
      });
    }
  }

  Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                       [&] (const int r)
  {
    Kokkos::atomic_add(&v.entry(k,r), tmp[r]);
  });
});
```

Vector parallelism across columns of factor matrices

Adding Scratch Space and Vector Parallelism

Sandia National Laboratories

```
typedef Kokkos::TeamPolicy<ExecSpace> Policy;
const int nnz = X.nnz();
const int VectorSize = 8;
const int TeamSize = 1;
Policy policy(nnz,TeamSize,VectorSize);

typedef Kokkos::View< double*,
  ExecSpace::scratch_memory_space,
  Kokkos::MemoryUnmanaged > ScratchSpace;
const size_t bytes = ScratchSpace::shmem_size(R);

Kokkos::parallel_for(
  policy.set_scratch_size(0,Kokkos::PerTeam(bytes)),
  KOKKOS_LAMBDA(Policy::member_type team)
{
  const size_t i = team.league_rank();

  ScratchSpace tmp(team.team_scratch(0), R);

  const size_t k = X.subscript(i,n);
  const double x_val = X.value(i);
```

```
  Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                       [&] (const int r)
  {
    tmp[r] = x_val;
  });

  for (int m = 0; m < N; ++m) {
    if (m != n) {
      const double *row = u[m].rowptr(X.subscript(i,m));
      Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                           [&] (const int r)
      {
        tmp[r] *= row[r];
      });
    }
  }

  Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                       [&] (const int r)
  {
    Kokkos::atomic_add(&v.entry(k,r), tmp[r]);
  });
});
```

Allocation from scratch memory space (shared memory on GPU)

Adding Scratch Space and Vector Parallelism

Sandia National Laboratories

```
typedef Kokkos::TeamPolicy<ExecSpace> Policy;
const int nnz = X.nnz();
const int VectorSize = 8;
const int TeamSize = 1;
Policy policy(nnz,TeamSize,VectorSize);

typedef Kokkos::View< double*,
  ExecSpace::scratch_memory_space,
  Kokkos::MemoryUnmanaged > ScratchSpace;
const size_t bytes = ScratchSpace::shmem_size(R);

Kokkos::parallel_for(
  policy.set_scratch_size(0,Kokkos::PerTeam(bytes)),
  KOKKOS_LAMBDA(Policy::member_type team)
{
  const size_t i = team.league_rank();

  ScratchSpace tmp(team.team_scratch(0), R);

  const size_t k = X.subscript(i,n);
  const double x_val = X.value(i);
```

```
  Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                       [&] (const int r)
  {
    tmp[r] = x_val;
  });

  for (int m = 0; m < N; ++m) {
    if (m != n) {
      const double *row = u[m].rowptr(X.subscript(i,m));
      Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                           [&] (const int r)
      {
        tmp[r] *= row[r];
      });
    }
  }

  Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                       [&] (const int r)
  {
    Kokkos::atomic_add(&v.entry(k,r), tmp[r]);
  });
});
```

- Final MTTKRP algorithm:
  - Incorporates team size > 1 on GPU for better occupancy (process blocks of non-zeros)
  - Blocking on R for better cache performance
    - Template algorithm on R block size for better vectorization
    - Choose block size to balance occupancy, tensor reads, minimize remainder term

Adding Team Parallelism

Sandia National Laboratories

```
typedef Kokkos::TeamPolicy<ExecSpace> Policy;
const bool is_cuda = std::is_same<ExecSpace,
  Kokkos::Cuda>::value;
const int nnz = X.nnz();
const int VectorSize = 8;
const int TeamSize = is_cuda ? 128/VectorSize : 1;
const int LeagueSize = (nnz+TeamSize-1)/TeamSize;
Policy policy(LeagueSize,TeamSize,VectorSize);

typedef Kokkos::View< double**, Kokkos::LayoutRight,
  ExecSpace::scratch_memory_space,
  Kokkos::MemoryUnmanaged > ScratchSpace;
const size_t bytes = ScratchSpace::shmem_size(TeamSize,R);

Kokkos::parallel_for(
  policy.set_scratch_size(0,Kokkos::PerTeam(bytes)),
  KOKKOS_LAMBDA(Policy::member_type team)
{
  const int team_index = team.team_rank();
  const size_t i = team.league_rank()*TeamSize+team_index;
  if (i >= nnz) return;

  ScratchSpace tmp(team.team_scratch(0), TeamSize, R);

  const size_t k = X.subscript(i,n);
  const double x_val = X.value(i);

  Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                       [&] (const int r)
  {
    tmp(team_index,r) = x_val;
  });

  for (int m = 0; m < n; ++m) {
    if (m != n) {
      const double *row = u[m].rowptr(X.subscript(i,m));
      Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                           [&] (const int r)
      {
        tmp(team_index,r) *= row[r];
      });
    }
  }

  Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                       [&] (const int r)
  {
    Kokkos::atomic_add(&v.entry(k,r), tmp(team_index,r));
  });
});
```