Parallel Tensor Computations in Python or C++ Using Cyclops

Edgar Solomonik

Department of Computer Science
University of Illinois at Urbana-Champaign

PASC 2018

July 3, 2018

L·P·N·A @ CS@Illinois
A library for parallel tensor computations

Cyclops Tensor Framework (github.com/cyclops-community/ctf)
- distributed-memory symmetric/sparse/dense tensor objects

Matrix<int> A(n, n, AS|SP, World(MPI_COMM_WORLD));
Tensor<float> T(order, is_sparse, dims, syms, ring, world);
T.read(...); T.write(...); T.slice(...); T.permute(...);

- parallel contraction/summation of tensors

C["ij"] = A["ik"]*B["kj"];   // matmul
C["ijl"] += A["ikl"]*B["kj"];// batched matmul
Z["abij"] += V["ijab"];      // tensor transpose
T["wxyz"] += U["uw"]*T["uxyz"];// TTM
T["abij"] = T["abij"]*D["abij"]; // Hadamard product
S["ii"] = v["i"];           // S = diag(v)
v["i"] += S["ii"];          // v += diag(S)
M["ij"] += Function<>([](double x){ return 1/x; })(v["j"]);

- ~2000 commits since 2011, open source since 2013
Electronic structure calculations with cyclops

Extracted from Aquarius (lead by Devin Matthews)
https://github.com/devinamatthews/aquarius

\[
\begin{align*}
\text{FMI}["mi"] & \quad += \quad 0.5 \times \text{WMNEF}["mnef"] \times T2["efin"]; \\
\text{WMNIJ}["mnij"] & \quad += \quad 0.5 \times \text{WMNEF}["mnef"] \times T2["efij"]; \\
\text{FAE}["ae"] & \quad -= \quad 0.5 \times \text{WMNEF}["mnef"] \times T2["afmn"]; \\
\text{WAMEI}["amei"] & \quad -= \quad 0.5 \times \text{WMNEF}["mnef"] \times T2["afin"]; \\
\end{align*}
\]

\[
\begin{align*}
\text{Z2}["abij"] & \quad = \quad \text{WMNEF}["ijab"]; \\
\text{Z2}["abij"] & \quad += \quad \text{FAE}["af"] \times T2["fbij"]; \\
\text{Z2}["abij"] & \quad -= \quad \text{FMI}["ni"] \times T2["abnj"]; \\
\text{Z2}["abij"] & \quad += \quad 0.5 \times \text{WABEF}["abef"] \times T2["efij"]; \\
\text{Z2}["abij"] & \quad += \quad 0.5 \times \text{WMNIJ}["mnij"] \times T2["abmn"]; \\
\text{Z2}["abij"] & \quad -= \quad \text{WAMEI}["amei"] \times T2["ebmj"]; \\
\end{align*}
\]

- CTF has been integrated with QChem, VASP (CC4S), and PySCF
- Is also being used for other applications, e.g. by IBM+LLNL collaboration to perform 49-qubit quantum circuit simulation
Electronic structure calculations with Cyclops

**CCSD** up to 55 (50) water molecules with cc-pVDZ

**CCSDT** up to 10 water molecules with cc-pVDZ

compares well to **NWChem** (up to 10x speed-ups for CCSDT)
Tensor<> Ea, Ei, Fab, Fij, Vabij, Vijab, Vabcd, Vijkl, Vaibj;
... // compute above 1-e an 2-e integrals

Tensor<> T(4, Vabij.lens, *Vabij.wrld);
T["abij"] = Vabij["abij"];

divide_EaEi(Ea, Ei, T);

Tensor<> Z(4, Vabij.lens, *Vabij.wrld);
Z["abij"] = Vijab["ijab"];  
Z["abij"] += Fab["af"]*T["fbij"];  
Z["abij"] -= Fij["ni"]*T["abnj"];  
Z["abij"] += 0.5*Vabcd["abef"]*T["efij"];  
Z["abij"] += 0.5*Vijkl["mnij"]*T["abmn"];  
Z["abij"] += Vaibj["amei"]*T["ebmj"];  

divide_EaEi(Ea, Ei, Z);

double MP3_energy = Z["abij"]*Vabij["abij"];
Sparse MP3 code

Strong and weak scaling of sparse MP3 code, with
(1) dense $V$ and $T$ (2) sparse $V$ and dense $T$ (3) sparse $V$ and $T$

PASC 2018
Custom tensor element types

Cyclops permits arbitrary element types and custom functions

- CombBLAS/GraphBLAS-like functionality
- See examples for SSSP, APSP, betweenness centrality, MIS, MIS-2
- Functionality to handle serialization of pointers within user-defined types is under development
- Block-sparsity via sparse tensor (local) of dense tensors (parallel)

```cpp
// Define Monoid tmon to perform matrix summation as addition ...

Matrix< Matrix<> > C(nblk, nblk, SP, self_world, tmon);

C["ij"] = Function< Matrix<> >( // Define the operation
    [](Matrix<> mA, Matrix<> mB){
        Matrix<> mC(mA.nrow, mB.ncol);
        mC["ij"] += mA["ik"]*mB["kj"];
        return mC;
    }
)(A["ik"], B["kj"]);```

PASC 2018 Parallel Tensor Computations Using Cyclops 7/15
for sparse tensors, a cyclic layout provides a load-balanced distribution
Parallel contraction in Cyclops

Cyclops uses nested parallel matrix multiplication variants

- **1D variants**
  - perform a different *matrix-vector product* on each processor
  - perform a different *outer product* on each processor

- **2D variants**
  - perform a different *inner product* on each processor
  - *scale a vector* on each processor then sum

- **3D variants**
  - perform a different *scalar product* on each processor then sum
  - can be achieved by *nesting* 1D+1D+1D or 2D+1D or 1D+2D

- All variants are *blocked* in practice, naturally generalized to sparse matrix products
Tensor blocking/virtualization

Preserving symmetric-packed layout using cyclic distribution constrains possible tensor blockings

Subdivision into more blocks than there are processors (virtualization)
Data mapping and redistribution

Transitions between contractions require redistribution and refolding

- 1D/2D/3D variants naturally map to 1D/2D/3D processor grids
- Initial tensor distribution is oblivious of contraction
  - by default each tensor distributed over all processors
  - user can specify any processor grid mapping
- Global redistribution done by one of three methods
  - reassign tensor blocks to processors (easy+fast)
  - reorder and reshuffle data to satisfy new blocking (fast)
  - treat tensors as sparse and sort globally by function of index
- Matricization/transposition is then done locally
  - dense tensor transpose done using HPTT (by Paul Springer)
  - sparse tensor converted to CSR sparse matrix format
For contractions, local summation and contraction is done via BLAS, including batched GEMM.

Threading is used via OpenMP and threaded BLAS.

GPU offloading is available but not yet fully robust.

For sparse matrices, MKL provides fast sparse matrix routines.

To support general (mixed-type, user-defined) elementwise functions, manual implementations are available.

User can specify blocked implementation of their function to improve performance.
Performance modeling and intelligent mapping

- **Performance models** used to select best contraction algorithm
- Based on *linear cost model for each kernel*

\[ T \approx \alpha S + \beta W + \nu Q + \gamma F \]

- *Scaling of* \( S, W, Q, F \) is a function of parameters of each kernel
- *Coefficients* for all kernels depend on compiler/architecture
- Linear regression with Tykhonov regularization used to select coefficients \( x^* \)
- Model training done by benchmark suite that executes various end-functionality for growing problem sizes, collecting observations of parameters in rows of \( A \) and execution timing in \( t \)

\[ x^* = \text{argmin}_{x} (||Ax - t||_2 + \lambda ||x||_2) \]
Using Cython, we have provided a Python interface for Cyclops

Follows numpy.ndarray conventions, plus sparsity and MPI execution

\[
Z[\text{"abij"}] += V[\text{"ijab"}]; \quad \text{\texttt{\# C++}}
\]

\[
Z.\text{i}(\text{"abij"}) \ll V.\text{i}(\text{"ijab"}) \quad \text{\texttt{\# Python}}
\]

\[
W[\text{"mnij"}] += 0.5*W[\text{"mnef"}]*T[\text{"efij"}]; \quad \text{\texttt{\# C++}}
\]

\[
W.\text{i}(\text{"mnij"}) \ll 0.5*W.\text{i}(\text{"mnef"})*T.\text{i}(\text{"efij"}) \quad \text{\texttt{\# Python}}
\]

\[
einsum(\text{"mnef,efij->mnij"},W,T) \quad \text{\texttt{\# numpy-style Python}}
\]

Python interface is under active development, but is functional and available (DEMO)
Future directions and acknowledgements

Future/ongoing directions in Cyclops development

- General abstractions for tensor decompositions
- Concurrent scheduling of multiple contractions
- Fourier transforms along tensor modes
- Improvements to functionality and performance for linear algebra

Acknowledgements

- Devin Matthews (UT Austin), Jeff Hammond (Intel Corp.), James Demmel (UC Berkeley), Torsten Hoefler (ETH Zurich), Zecheng Zhang, Eric Song, Eduardo Yap, Linjian Ma (UIUC)
- Computational resources at NERSC, CSCS, ALCF, NCSA, TACC