S7255: CUTT: A HIGH-PERFORMANCE TENSOR TRANSPOSE LIBRARY FOR GPUs

Antti-Pekka Hynninen, 5/10/2017, GTC2017, San Jose CA
Tensor contractions are the most computationally intensive part of quantum many-body methods used in NWChem, DIRAC, LS-DALTON, and ACES-IV.

\[ D(a, b, i) += L(a, c, i, k) R(k, b, c) \]

Sum over repeated indices \( c \) and \( k \)

Evaluating tensor contractions **directly** requires implementing a lot of hard-to-write custom code.

**Indirect** approach transposes tensors and uses efficient linear algebra libraries (such as cuBLAS) to perform matrix multiply.
TENSOR CONTRACTIONS

Indirect approach

Reduction over a pair of indices shared by two tensors, e.g.

\[ D(a, b, i) + L(a, c, i, k) R(k, b, c) \]

This can be evaluated as

\[ L(a, c, i, k) \rightarrow L(a, i, k, c) \quad \# \text{ tensor transpose} \]

\[ R(k, b, c) \rightarrow R(k, c, b) \quad \# \text{ tensor transpose} \]

\[ D(a, i, b) + L(a, i, k, c) R(k, c, b) \quad \# \text{ matrix multiply} \]

\[ D(a, i, b) \rightarrow D(a, b, i) \quad \# \text{ tensor transpose} \]

Able to take advantage of the high-performance matrix multiply routines provided by cuBLAS
PREVIOUS WORK

No runtime high-performance tensor transpose library exists for GPUs

Previous implementation by my co-author [1] was sub-optimal on GPU platforms

![Graph showing bandwidth performance for Nvidia Tesla K20X with random permutations]

Work in [2] relies on compiler to build custom kernels e.g. not runtime


TENSOR TRANSPOSE ALGORITHMS
MATRIX TRANSPOSE: TILED ALGORITHM

Step 1:
Read 32x32 tile from global memory to shared memory

Step 2:
Read shared memory in transposed order and write to global memory

Tiled Algorithm

Constant shared memory usage (~32x32)

Performs well when \(d1\) and \(d5\) are fairly large (~32)

Poor performance for small (2-8) dimensions

Would it be possible to pack multiple small dimensions into shared memory?
PACKED ALGORITHM

No longer uses 32x32 shared memory tile

Loads entire dimensions into shared memory (not tiled)

As much shared memory is allocated as it takes to store the elements

Must choose which dimensions to pack

New problem: What if e.g. d5 is very large?
PACKED-SPLIT ALGORITHM

Split largest dimension

Number of splits is determined by the shared memory size

Must choose which dimensions to pack, and number of splits
MEMORY POSITION CALCULATION
GLOBAL MEMORY POSITION CALCULATION

H = Number of elements in shared memory
M = Number of elements in loop volume

Need to convert scalar positions s and p to global memory positions:

glRead = Global memory read
glWrite = Global memory write

Global memory position is split into:

\[ \text{glRead} = \text{glMinorRead}(s) + \text{glMajorRead}(p) \]
\[ \text{glWrite} = \text{glMinorWrite}(s) + \text{glMajorWrite}(p) \]
MAJOR POSITION CALCULATION

// int p = 0, ..., M-1
// int c[n] = {1, d3, d3*d4}
// int d[n] = {d3, d4, d6}
// int t[n] = {d1*d2, d1*d2*d3, d1*d2*d3*d4*d5}
int glMajorRead = 0;
for (int i=0; i < n; i++) {
    glMajorRead += ((p / c[i]) % d[i]) * t[i];
}

O(n)

Observation: p is constant within thread block (and therefore warp)
WARP-PARALLEL POSITION CALCULATION

// int p = 0, ..., M-1
// int c = {1, d3, d3*d4, 1, ..., 1}
// int d = {d3, d4, d6 , 1, ..., 1}
// int t = {d1*d2, d1*d2*d3, d1*d2*d3*d4*d5,...}
int glMajorRead = ((p / c) % d) * t;
for (int i=16;i >= 1;i/=2) {
    glMajorRead += __shfl_xor(glMajorRead, i);
}

Single divide, modulo, and multiply
O(1) i.e. performance independent of tensor rank
Works up to n=32
MINOR POSITION CALCULATION

For Tiled algorithm this is trivial

For Packed and Packed-Split, pre-compute positions and store into registers

Number of registers per thread:

\[
\text{numReg} = \frac{H - 1}{\text{blockDim.x}} + 1
\]

int glMinorRead[numReg]

int shRead[numReg]

int glMinorWrite[numReg]

Template kernel with numReg
ALGORITHM & PARAMETER CHOICE
CHOOSING THE BEST ALGORITHM

Algorithm choice: Tiled, Packed, Packed-Split

Tiled: no free parameters

Packed: input and output ranks

Packed-Split: input and output ranks, number of splits

Large performance differences between different algorithm and parameter choices
Measure - plans perform all possible tensor transposes and choose the best performing plan. LARGE overhead

Heuristic - plans choose best plan by estimating the transpose runtime based on analytical GPU performance model. SMALL overhead

Heuristic plans must be used in QM calculations

Getting the heuristic planning to work accurately was a major hurdle

Better approach is needed for choosing the heuristic plans (Machine Learning?)
BENCHMARK 1

Tensor ranks 2 to 7

Ratio between largest and smallest tensor dimensions 1:1, 5:1, and 15:1

Tensor volume normally distributed with average 200M elements and standard deviation of 20M elements

500 random permutations for each tensor rank and ratio

9000 tensor transposes in total
**TESLA K20X**

*maximum bandwidth measured using GPU-STREAM:*

TESLA P100
Tensor ranks 8 and 12

Rank 8: \((5, 3, 2, 4, 35, 33, 37, 40)\) 200M elements

Rank 12: \((2, 3, 4, 3, 2, 2, 3, 2, 20, 18, 22, 24)\) 328M elements

500 random permutations for both tensor ranks

Simulates realistic workload in Quantum Chemistry calculations
TESLA M40

Bar chart showing bandwidth (GiBs) and percentage of maximum bandwidth for different ranks and measures.
PERFORMANCE DISTRIBUTION
BENCHMARK 3

Set of 57 tensor transposes from (TTC):


Somewhat “easy” benchmark due to small number of permutations
TESLA K40M

TTC average 140 GiB/s

cuTT average 144 GiB/s

Real world tensor contractions performed on TAL-SH (Tensor Algebra Library for Shared Memory Computers)

Dmitry I. Lyakh at Oak Ridge National Laboratory

9306 random permutations on tensors up to rank 8

Matrix multiply performed using cuBLAS
**TESLA K20X**

GPU

(a) Single precision

(b) Double precision

\[ D = D + L \cdot R \]

**Arithmetic Intensity**

\[
\text{Arithmetic Intensity} = \frac{2\sqrt{\text{vol}(D)\text{vol}(L)\text{vol}(R)}}{\text{vol}(D) + \text{vol}(L) + \text{vol}(R)}
\]
TESLA M40

Single precision
TESLA P100

(a) Single precision

(b) Double precision
CONCLUSIONS & ACKNOWLEDGEMENTS
CONCLUSIONS

Fully runtime library for high-performance tensor transposing on NVIDIA GPUs

Extensive benchmarking

Achieves median of 70-80% of the maximum achievable memory bandwidth

Performance equals or exceeds the performance of compiler-based approach (TTC)

Enables close to peak FLOP tensor contractions on P100

Integrated as part of TAL-SH (https://github.com/DmitryLyakh/TAL_SH)

Work underway to be used in NWChem, DIRAC, LS-DALTON, and ACES-IV

Source code available at: https://github.com/ap-hynninen/cutt

Manuscript available at: https://arxiv.org/abs/1705.01598
ACKNOWLEDGEMENTS

Dmitry I. Lyakh at Oak Ridge Leadership Computing Facility at ORNL

ORNL where 80% of the work was done