Acrotensor: Flexible Tensor Contractions on the GPU

Aaron Fisher, Tzario Kolev, Johann Dahm

Feb 2019

Aaron Fisher
Computational Scientist
Why should we care about tensor algebra?

- All the usual dense matrix operations can be represented with tensor algebra.
- Tensor algebra extends naturally to enable batching.
- Higher rank tensor algebra has many applications including:
  - Finite elements
  - Machine Learning
  - Quantum simulation
- Growth opportunity for linear algebra packages

\[ T_i = \sum_i A_{ii} \quad d = \sum_i a_i b_i \quad A_{ij} = a_i b_j \]

\[ C_{ik} = \sum_j A_{ij} B_{jk} \]

\[ C_{eik} = \sum_j A_{eij} B_{ejk} \]
Variety of general tensor algebra packages

- **CPU Only**
  - Ftensor
  - Taco
  - libtensor
  - Numpy::einsum
  - ...

- **GPU**
  - Magma
  - TensorFlow::einsum
  - TensorComprehensions
  - ...

  Reshape to large batches of GEMMS which leaves performance on the table

  Evolutionary search tuned GPU implementations
Acrotensor

- Open source high performance C++/CUDA tensor contraction library.
- Supports general batched tensor contractions with an easy to use interface.
- Utilizes CUDA JIT capability to exploit the structure of the tensor contractions and the dimensions of the tensors.
- Supports fusing of multiple tensor contraction into a single CUDA kernel.
- Well tested with unit tests and test integrations with finite element software.
- Available on Github:  
  — https://github.com/LLNL/acrotensor
Tensor objects in Acrotensor

- Tensors are stored as flat arrays with dimension and stride information.
- Index arithmetic used to allow natural tensor indexing into the flat arrays.
- CUDA memory mapping allows for transfers between CPU and GPU.

Tensor T(3,2,2);
Dims = [3,2,2]
Strides = [4,2,1]
Data = [0,0,0,0,0,0,0,0,0,0,0,0]

\[
\text{ind} = i_0 s_0 + i_1 s_1 + i_2 s_2;
\]
\[
T(i,j,k) = Data[\text{ind}];
\]

T.MoveToGPU()
T.MoveFromGPU();

T(3,2,2)
0,0,0
0,0,1
0,1,0
0,1,1
1,0,0
1,0,1
1,1,0
1,1,1
2,0,0
2,0,1
2,1,0
2,1,1
Tensor contractions are defined by “kernel” strings passed into a “TensorExecutor” object.

Any index not on the LHS of the equation is considered a sum contraction.

Tensor objects are passed in as well and their dimensions are used to set the loop dimensions for the contraction.

Different TensorExecutor objects can be instantiated to handle contractions on different devices (CPU, GPU).

\[
\text{for } i = 0..N_i-1 \\
\text{for } j = 0..N_j-1 \\
B(i) += A(i,j)\times B(j)
\]
Putting it all together for user friendly tensor algebra

```cpp
TensorEngine TE("Cuda");
Tensor M(mdims, element_matrices->GetData(0));
Tensor B(ir1d->Size(), p+1
Tensor C(ir1d->Size(), p+1, p+1);
Tensor D(ddims);
Tensor W(wdims);
Tensor T(num_elements, irfull->Size());
for (int k = 0; k < B.GetDim(0); ++k)
{
    const IntegrationPoint &ip = ir1d->IntPoint(k);
    el->CalcShape1D(ip, shape);
    for (int i = 0; i < B.GetDim(1); ++i)
        B(k,i) = shape[i];
}
...
TE["D_e_k1_k2 = W_k1_k2 T_e_k1_k2" ,D, W, T];
TE["E_e_i2_j2_k1 = C_k2_i2_j2 D_e_k1_k2" ,E, C, D];
TE["M_e_i1_i2_j1_j2 = C_k1_i1_j1 E_e_i2_j2_k1" ,M, C, E];
M.MoveFromGPU();
```
Approach for generating contraction CUDA kernels

\[ Z_{m,e,k1,k2,k3} = U_{n,e,k1,k2,k3} D_{e,m,n,k1,k2,k3} \]

- Contraction operation is considered a series of loops that will map to GPU hardware.

for \( m = 0..Nm-1 \)
  for \( e = 0..Ne-1 \)
    for \( k1 = 0..Nk1-1 \)
      for \( k2 = 0..Nk2-1 \)
        for \( k3 = 0..Nk3-1 \)
          for \( n = 0..Nn-1 \)
Approach for generating contraction CUDA kernels

\[ Z_{m,e,k1,k2,k3} = U_{n,e,k1,k2,k3} \cdot D_{e,m,n,k1,k2,k3} \quad \text{m=n=3, e=10000, k1=k2=k3=4} \]

- Loop sized are fixed to hard values using the input tensor sizes.

for m = 0..3-1
  for e = 0..10000-1
    for k1 = 0..4-1
      for k2 = 0..4-1
        for k3 = 0..4-1
          for n = 0..3-1
Approach for generating contraction CUDA kernels

\[ Z_{m,e,k1,k2,k3} = U_{n,e,k1,k2,k3} D_{e,m,n,k1,k2,k3} \quad \text{m=n=3, e=10000, k1=k2=k3=4} \]

- Heuristics used to classify indices as outer, middle, and contraction.

for \( m = 0..3-1 \)
  for \( e = 0..10000-1 \)
    for \( k1 = 0..4-1 \)
      for \( k2 = 0..4-1 \)
        for \( k3 = 0..4-1 \)
          for \( n = 0..3-1 \)
Approach for generating contraction CUDA kernels

\[ Z_{m,e,k1,k2,k3} = U_{n,e,k1,k2,k3} \ D_{e,m,n,k1,k2,k3} \]

- Loops are reordered and mapped to GPU structures.

\[
\text{for } e = 0..10000-1 \\
\text{for } m = 0..3-1 \\
\text{for } k1 = 0..4-1 \\
\text{for } k2 = 0..4-1 \\
\text{for } k3 = 0..4-1 \\
\text{for } n = 0..3-1
\]

Each block corresponds to these indices.

The thread ID on a given block corresponds to these indices.

Each thread loops over these indices.
CUDA Code Generated

```c
void Kernel(double * const T0, double const * const T1, double const * const T2)
{
    double sum;
    const unsigned int outidx = blockIdx.x;
    __syncthreads();
    unsigned int I0 = outidx;    // e

    int I4;
    int I1;
    int I2;
    int I3;
    I4 = ((threadIdx.x + 0) / 64);    // m
    I1 = ((threadIdx.x + 0) / 16) % 4;    // k1
    I2 = ((threadIdx.x + 0) / 4) % 4;    // k2
    I3 = ((threadIdx.x + 0) / 1) % 4;    // k3
    if (threadIdx.x < 192)
    {
        sum = 0.0;
        for (unsigned int I5 = 0; I5 < 3; ++I5) {    // n
            sum += T1[I5*5120000 + I0*64 + I1*16 + I2*4 + I3*1]*T2[I0*576 + I4*192 + I5*64 + I1*16 + I2*4 + I3*1];
        }
        T0[I4.x*5120000 + I0*64 + I1*16 + I2*4 + I3*1] = sum;
        __syncthreads();
    }
}
```
Optimizations applied to CUDA kernels

- Fixed size contraction loops are unrolled.
- Middle loops are reordered to maximize coalesced memory reads and memory coherence.
- Multiple can be fused together into a single kernel sharing the same outer loop.
- Small tensors that are used more than once in a single kernel are cached in fast shared memory.
- Outputs of one contraction that will be used in another contraction are cached in shared memory.
- CUDA block size is to avoid idling threads while maintaining high occupancy and maximizing shared memory use.
Matrix-free Finite Element 3D diffusion operator applied to a vector 200 times.

10,000 to 80,000 3\textsuperscript{rd} and 5\textsuperscript{th} order elements in the meshes.

Tested TensorFlow and AcroTensor implementations.

T2 = \texttt{tf.einsum}('sc,eabc->eabs', B, X)
T1 = \texttt{tf.einsum}('rb,eabs->eabs', B, T2)
U1 = \texttt{tf.einsum}('qa,eabs->eabs', G, T1)
T1 = \texttt{tf.einsum}('rb,eabs->eabc', G, T2)
U2 = \texttt{tf.einsum}('qa,eabs->eabs', B, T1)
T2 = \texttt{tf.einsum}('sc,eabc->eabs', B, X)
T1 = \texttt{tf.einsum}('rb,eabs->eabs', B, T2)
U3 = \texttt{tf.einsum}('qa,eabc->eabc', B, T1)
Z = \texttt{tf.einsum}('neqrs,emmqrq->meqrs', U, D)
T1 = \texttt{tf.einsum}('sc,eqrs->ecqr', B, Z1)
T2 = \texttt{tf.einsum}('rb,ecqr->ebcq', B, T1)
Y += \texttt{tf.einsum}('qa,ebcq->eabc', G, T2)
T1 = \texttt{tf.einsum}('sc,eqrs->ecqr', B, Z2)
T2 = \texttt{tf.einsum}('rb,ecqr->ebcq', G, T1)
Y += \texttt{tf.einsum}('qa,ebcq->eabc', B, T2)
T1 = \texttt{tf.einsum}('sc,eqrs->ecqr', G, Z3)
T2 = \texttt{tf.einsum}('rb,ecqr->ebcq', B, T1)
Y += \texttt{tf.einsum}('qa,ebcq->eabc', B, T2)

\texttt{TE.BeginMultiKernelLaunch();}
\texttt{TE("T2_e_i1_i2_k3 = B_k3_i3 X_e_i1_i2_i3", T2, B, X);}
\texttt{TE("T1_e_i1_k2_k3 = B_k2_i2 T2_e_i1_i2_k3", T1, B, T2);}
\texttt{TE("U1_e_k1_k2_k3 = G_k1_i1 T1_e_i1_k2_k3", U1, G, T1);}
\texttt{TE("T1_e_i1_k2_k3 = G_k2_i2 T2_e_i1_i2_k3", T1, G, T2);}
\texttt{TE("U2_e_k1_k2_k3 = B_k1_i1 T1_e_i1_k2_k3", U2, B, T1);}
\texttt{TE("T2_e_i1_i2_k3 = G_k3_i3 X_e_i1_i2_i3", T2, G, X);}
\texttt{TE("T1_e_i1_k2_k3 = B_k2_i2 T2_e_i1_i2_k3", T1, B, T2);}
\texttt{TE("U3_e_k1_k2_k3 = B_k1_i1 T1_e_i1_k2_k3", U3, B, T1);}
\texttt{TE("Z_m_e_k1_k2_k3 = U_n_e_k1_k2_k3 D_e_m_n_k1_k2_k3", Z, U, D);}
\texttt{TE("T1_e_i3_k1_k2 = B_k3_i3 Z1_e_k1_k2_k3", T1, B, Z1);}
\texttt{TE("T2_e_i2_i3_k1 = B_k2_i2 T1_e_i3_k1_k2", T2, B, T1);}
\texttt{TE("Y_e_i1_i2_i3 = G_k1_i1 T2_e_i2_i3_k1", Y, G, T2);}
\texttt{TE("T1_e_i3_k1_k2 = B_k3_i3 Z2_e_k1_k2_k3", T1, B, Z2);}
\texttt{TE("T2_e_i2_i3_k1 = G_k2_i2 T1_e_i3_k1_k2", T2, G, T1);}
\texttt{TE("Y_e_i1_i2_i3 += B_k1_i1 T2_e_i2_i3_k1", Y, B, T2);}
\texttt{TE("T1_e_i3_k1_k2 = G_k3_i3 Z3_e_k1_k2_k3", T1, G, Z3);}
\texttt{TE("T2_e_i2_i3_k1 = B_k2_i2 T1_e_i3_k1_k2", T2, B, T1);}
\texttt{TE("Y_e_i1_i2_i3 += B_k1_i1 T2_e_i2_i3_k1", Y, B, T2);}
\texttt{TE.EndMultiKernelLaunch();}
# FEM diffusion performance timings

<table>
<thead>
<tr>
<th># elements</th>
<th>TensorFlow (s)</th>
<th>AcroTensor (s)</th>
<th>Improvement Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>0.25</td>
<td>0.05</td>
<td>5.0</td>
</tr>
<tr>
<td>20,000</td>
<td>0.44</td>
<td>0.06</td>
<td>7.3</td>
</tr>
<tr>
<td>40,000</td>
<td>0.78</td>
<td>0.12</td>
<td>6.5</td>
</tr>
<tr>
<td>80,000</td>
<td>1.50</td>
<td>0.23</td>
<td>6.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># elements</th>
<th>TensorFlow (s)</th>
<th>AcroTensor (s)</th>
<th>Improvement Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>0.75</td>
<td>0.12</td>
<td>6.3</td>
</tr>
<tr>
<td>20,000</td>
<td>1.54</td>
<td>0.23</td>
<td>6.7</td>
</tr>
<tr>
<td>40,000</td>
<td>3.03</td>
<td>0.45</td>
<td>6.7</td>
</tr>
<tr>
<td>80,000</td>
<td>6.37</td>
<td>0.90</td>
<td>7.1</td>
</tr>
</tbody>
</table>

Note: Acrotensor requires additional 0.2 s to JIT compile the kernel on the first pass.
Summary and Future Work

- Acrotensor is a C++ library for large scale tensor contractions on GPUs.
- JIT compilation is utilized to provide a user friendly dynamic interface without giving up on high performance.
- Acrotensor performs >5x faster than TensorFlow’s similar einsum operation.
- Future work
  - CPU JIT compilation for single- and multi-threaded execution.
  - Further GPU optimization.
  - Integration with interested application codes.

https://github.com/LLNL/acrotensor