Towards a High-Performance Tensor Algebra Package for Accelerators

M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov

Abstract

Numerous important applications, e.g., high-order FEM simulations, can be expressed through tensors. Examples are computation of FE matrices and SpMV products expressed as generalized tensor contractions. Contractions by the first index can often be represented as tensor index reordering plus gemm, which is a key factor to achieve high-performance. We present ongoing work on the design of a high-performance package in MAGMA for Tensor algebra that includes techniques to organize tensor contractions, data storage, and parametrization related to batched execution of large number of small tensor contractions. We apply auto-tuning and code generation techniques to provide an architecture-aware, user-friendly interface.

Motivation

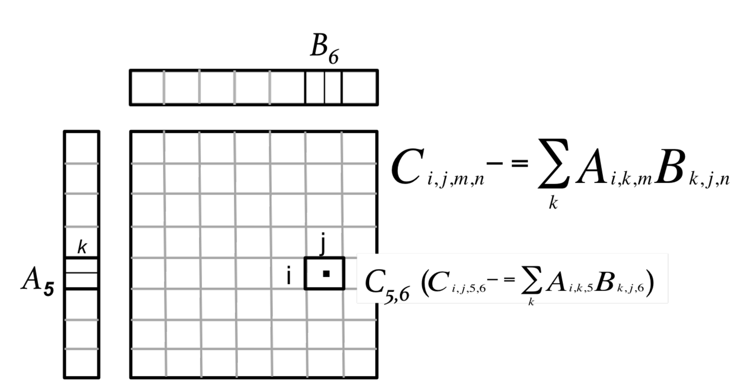
Numerous important applications can be expressed through tensors:

- High-order FEM simulations
- Signal Processing
- Numerical Linear Algebra
- Numerical Analysis
- Data Mining
- Deep Learning
- Graph Analysis
- Neuroscience and more

The goal is to design a:

- High-performance package for Tensor algebra
- Built-in architecture-awareness (GPU, Xeon Phi, multicore)
- User-friendly interface

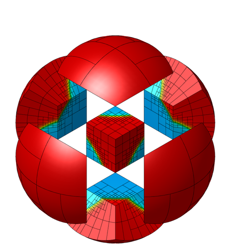
Example cases



Numerical linear algebra:

- A 4-dimensional tensor contraction
- rank-k update on matrices in tile format (k can be small, e.g., sub-vector/warp size)
- Must determine (in software) if possible to do it through batched GEMM kernels

Lagrangian Hydrodynamics in the BLAST code[1]



On semi-discrete level our method can be written as

$$\textbf{Momentum Conservation:} \quad \frac{d\mathbf{v}}{dt} = -\mathbf{M}_\mathbf{v}^{-1}\mathbf{F}\cdot\mathbf{1}$$

$$\textbf{Energy Conservation:} \quad \frac{d\mathbf{e}}{dt} = \mathbf{M}_\mathbf{e}^{-1}\mathbf{F}^\mathbf{T}\cdot\mathbf{v}$$

$$\textbf{Equation of Motion:} \quad \frac{d\mathbf{x}}{dt} = \mathbf{v}$$

where $\mathbf{v}$, $\mathbf{e}$, and $\mathbf{x}$ are the unknown velocity, specific internal energy, and grid position, respectively; $\mathbf{M}_\mathbf{v}$ and $\mathbf{M}_\mathbf{e}$ are independent of time velocity and energy mass matrices; and $\mathbf{F}$ is the generalized corner force matrix depending on $(\mathbf{v},\mathbf{e},\mathbf{x})$ that needs to be evaluated at every time step.

[1] V. Dobrev, T.Kolev, R.Rieben. *High order curvilinear finite element methods for Lagrangian hydrodynamics*. SIAM J.Sci.Comp.34(5), B606–B641. (36 pages)

Tensor operations in high-order FEM

Consider the **FE** mass matrix **ME** for an element/zone **E** with weight ρ, as a 2-dimensional tensor:

$$(M_E)_{ij} = \sum_{k=1}^{nq} \alpha_k\, \rho(q_k)\, \varphi_i(q_k)\, \varphi_j(q_k)\, |J_E(q_k)|\,,$$

$i, j = 1,\ldots, nd$, where

- nd is the number of FE degrees of freedom (dofs)
- nq is the number of quadrature points
- $\{\varphi_i\}_{i=1}^{nd}$ are the FE basis functions on the reference element
- $|J_E|$ is the determinant of the element transformation
- $\{q_k\}_{k=1}^{nq}$ and $\{\alpha_k\}_{k=1}^{nq}$ are the points and weights of the quadrature rule

Take the nq x nd matrix $B_{ki} = \varphi_i(q_k)$, and $(D_E)_{kk} = \alpha_k\,\rho(q_k)\,|J_E(q_k)|$.
Then, $(M_E)_{ij} = \sum_{k=1}^{nq} B_{ki}(D_E)_{kk}B_{kj}$, or omitting the E subscript $M = B^T D B$.

Using **FE** of order p, we have $nd = O(p^d)$ and $nq = O(p^d)$, so **B** is dense $O(p^d)$ x $O(p^d)$ matrix.

If the **FE** basis and the quadrature rule have tensor product structure, we can decompose dofs and quadrature point indices in logical coordinate axes

$$\mathbf{i} = (\mathbf{i}_1, \ldots, \mathbf{i}_d), \quad \mathbf{j} = (\mathbf{j}_1, \ldots, \mathbf{j}_d), \quad \mathbf{k} = (\mathbf{k}_1, \ldots, \mathbf{k}_d)$$

so $\mathbf{M}_{ij}$ can be viewed as 2d-dimensional tensor $\mathbf{M}_{i1,\ldots,id,\,j1,\ldots,jd}$.

Summary of kernels needed:

- Assembly of M, referred as equations (1) & (2) below
- Evaluations of M times V, referred as equations (3) & (4) below

TENSOR KERNELS FOR ASSEMBLY/EVALUATION 3

stored components	FLOPs for assembly	amount of storage	FLOPs for matvec	numerical kernels
full assembly				
M	$O(p^{3d})$	$O(p^{2d})$	$O(p^{2d})$	$B, D \mapsto B^T DB,\ x \mapsto Mx$
decomposed evaluation				
B, D	$O(p^{2d})$	$O(p^{2d})$	$O(p^{2d})$	$x \mapsto Bx,\ x \mapsto B^T x,\ x \mapsto Dx$
near-optimal assembly – equations (1) and (2)				
$M_{i_1,\cdots,j_d}$	$O(p^{2d+1})$	$O(p^{2d})$	$O(p^{2d})$	$A_{i_1,k_2,j_1} = \sum_{k_1} B^{1d}_{k_1,i_1} B^{1d}_{k_1,j_1} D_{k_1,k_2}$ (1a)
				$A_{i_1,i_2,j_1,j_2} = \sum_{k_2} B^{1d}_{k_2,i_2} B^{1d}_{k_2,j_2} C_{i_1,k_2,j_1}$ (1b)
				$A_{i_1,k_2,k_3,j_1} = \sum_{k_1} B^{1d}_{k_1,i_1} B^{1d}_{k_1,j_1} D_{k_1,k_2,k_3}$ (2a)
				$A_{i_1,i_2,k_3,j_1,j_2} = \sum_{k_2} B^{1d}_{k_2,i_2} B^{1d}_{k_2,j_2} C_{i_1,k_2,k_3,j_1}$ (2b)
				$A_{i_1,i_2,i_3,j_1,j_2,j_3} = \sum_{k_3} B^{1d}_{k_3,i_3} B^{1d}_{k_3,j_3} C_{i_1,i_2,k_3,j_1,j_2}$ (2c)
near-optimal evaluation (partial assembly) – equations (3) and (4)				
B^{1d}, D	$O(p^d)$	$O(p^d)$	$O(p^{d+1})$	$A_{j_1,k_2} = \sum_{j_2} B^{1d}_{k_2,j_2} V_{j_1,j_2}$ (3a)
				$A_{k_1,k_2} = \sum_{j_1} B^{1d}_{k_1,j_1} C_{j_1,k_2}$ (3b)
				$A_{k_1,i_2} = \sum_{k_2} B^{1d}_{k_2,i_2} C_{k_1,k_2}$ (3c)
				$A_{i_1,i_2} = \sum_{k_1} B^{1d}_{k_1,i_1} C_{k_1,i_2}$ (3d)
				$A_{j_1,j_2,k_3} = \sum_{j_3} B^{1d}_{k_3,j_3} V_{j_1,j_2,j_3}$ (4a)
				$A_{j_1,k_2,k_3} = \sum_{j_2} B^{1d}_{k_2,j_2} C_{j_1,j_2,k_3}$ (4b)
				$A_{k_1,k_2,k_3} = \sum_{j_1} B^{1d}_{k_1,j_1} C_{j_1,k_2,k_3}$ (4c)
				$A_{k_1,k_2,i_3} = \sum_{k_3} B^{1d}_{k_3,i_3} C_{k_1,k_2,k_3}$ (4d)
				$A_{k_1,i_2,i_3} = \sum_{k_2} B^{1d}_{k_2,i_2} C_{k_1,k_2,i_3}$ (4e)
				$A_{i_1,i_2,i_3} = \sum_{k_1} B^{1d}_{k_1,i_1} C_{k_1,i_2,i_3}$ (4f)
matrix-free evaluation				
none	none	none	$O(p^{d+1})$	evaluating entries of B^{1d}, D, (3a)–(4f) sums

APPROACH AND RESULTS

User-friendly interface

To provide various interfaces, including one using C++11. Top level design to provide features similar to the `mshadow` library. https://github.com/dmlc/mshadow

```
// Our current interface :
// create a 2 x 5 x 2 float tensor , default locality is cpu using std::vector as default backend for data
Tensor<2,5,2> ts;
// create a 2 x 5 x 2 tensor on the gpu using thrust as the default backend for data
Tensor<2,5,5,gpu_> d_ts;
// Call a thrust function to set values to 9
thrust::fill(d_ts.begin() , d_ts.end() , 9);
// Send back values to the cpu tensor
ts = d_ts ;
// Reorder the 2 x 5 x 2 tensor to a matrix 2 x 10 using views
view<2,10> mat = ts ;
```

Index reordering/reshape

If we store tensors as column-wise 1D arrays,

$$M^{nd_1\times nd_2\times nd_1\times nd_2}_{i_1,i_2,j_1,j_2} = M^{nd\times nd}_{i,j} = M^{nd^2}_{i+nd\,j} = M^{nd^2}_{i_1+nd_1 i_2+nd(j_1+nd_1 j_2)}$$

, *i.e.*, M can be interpreted as a 4th order tensor, a nd x nd matrix, or a vector of size nd^2, without changing the storage. We can define

$$Reshape(T)^{m_1\times\cdots\times m_q}_{j_1,\cdots,j_q} = T^{n_1\times\cdots\times n_r}_{i_1,\cdots,i_r}$$

as long as $n_1 \ldots n_r = m_1 \ldots m_q$ and for every $i_{1..r}, j_{1..q}$ $i_1 + n_1 i_2 + \ldots + n_1 n_2 \ldots n_{r-1} i_r = j_1 + m_1 j_2 + \ldots + m_1 m_2 \ldots m_{q-1} j_q$.

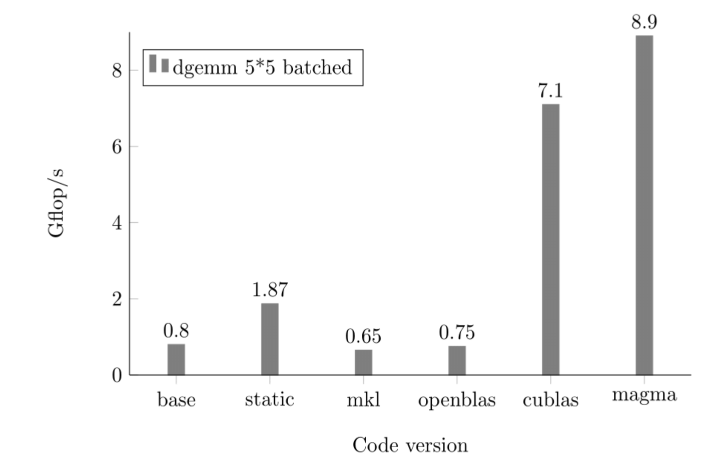
Contractions can be implemented as a sequence of pairwise contractions. There is enough complexity here to search for something better: code generation, index reordering, and autotuning will be used, *e.g.*, contractions (3a) - (4f) can be implemented as tensor index-reordering plus gemm $A, B \to A^T B$.

Code Generation

C++11 features will be used as much as possible. Additional needs will be handled by defining a domain specific embedded language (DSEL). This technique is used in C++ to take advantage of DSL features while using the optimizations provided by a standard compiler. It will handle the generation of versions (index reordering, next) to be empirically evaluated and be part of the autotuning framework.

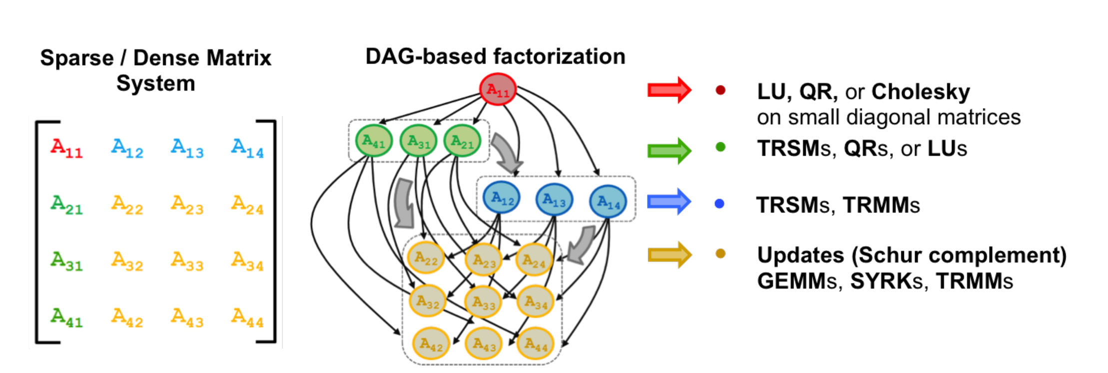
Autotuning

We are developing fixed-size gemm kernels for GPUs, Xeon Phi, and multicore (see Figure on Right for a single core intel Xeon E5-2620 and K40) through an autotuning framework. A number of generic versions are developed and parametrized for performance. The parameters are autotuned (empirically) to find "best" kernels for specific size.



Batched LA

Tensor contractions are transformed through reshapes to batched LA operations, many of which available in MAGMA[2] http://icl.cs.utk.edu/magma/ (including LU, QR, Cholesky, GEMM, GEMV, TRSM, SYRK).



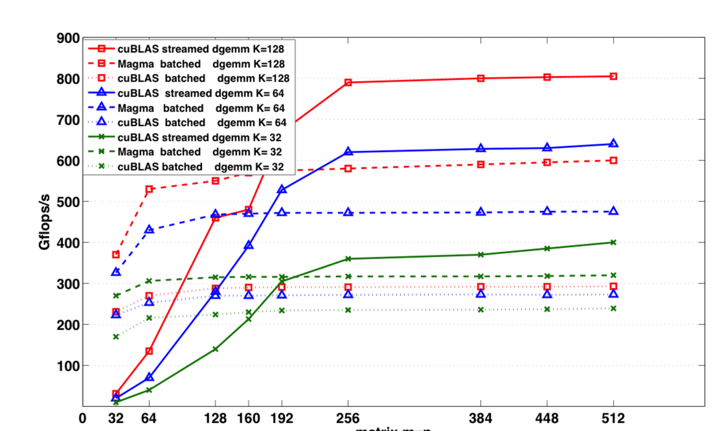


Figure:
Batched dgemms on K40 GPU.
Batch count is 2,000.

MAGMA exceeds in performance CUBLAS for "small" sizes, currently tuned for above 32. Current work is concentrated on kernels for fixed smaller (sub-warp) sizes.

[2] A.Haidar, T.Dong, S.Tomov, P.Luszczek, and J.Dongarra. *A framework for batched and GPU-resident factorization algorithms applied to block Householder transformations*. ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015.

Conclusions and Future directions

- High-performance package on Tensor Algebra has the potential for high-impact on a number of important applications
- Multidisciplinary effort
- Current results show promising performance, where various components will be leveraged from autotuning MAGMA Batched linear algebra kernels, and BLAST from LLNL
- This is an ongoing work

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.
LLNL release number LLNL-POST-676632

Smoky Mountains Computational Sciences and Engineering Conference
Gatlinburg, Tennessee, Aug 31- Sept 2, 2015
http://computing.ornl.gov/workshops/SMC15/

ICL's work on this material was supported by the National Science Foundation under Grant ACI-1339822, the Department of Energy, and NVIDIA.