Procedia Computer Science

Volume 80, 2016, Pages 108–118

ICCS 2016. The International Conference on Computational Science

High-Performance Tensor Contractions for GPUs

A. Abdelfattah[1], M. Baboulin[2], V. Dobrev[3], J. Dongarra[1,4], C. Earl[3], J. Falcou[2], A. Haidar[1], I. Karlin[3], Tz. Kolev[3], I. Masliah[2], and S. Tomov[1]

[1] Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
[2] University of Paris-Sud, France
[3] Lawrence Livermore National Laboratory, Livermore, CA, USA
[4] University of Manchester, Manchester, UK

Abstract
We present a computational framework for high-performance tensor contractions on GPUs. High-performance is difficult to obtain using existing libraries, especially for many independent contractions where each contraction is very small, e.g., sub-vector/warp in size. However, using our framework to batch contractions plus application-specifics, we demonstrate close to peak performance results. In particular, to accelerate large scale tensor-formulated high-order finite element method (FEM) simulations, which is the main focus and motivation for this work, we represent contractions as tensor index reordering plus matrix-matrix multiplications (GEMMs). This is a key factor to achieve algorithmically many-fold acceleration (*vs.* not using it) due to possible reuse of data loaded in fast memory. In addition to using this context knowledge, we design tensor data-structures, tensor algebra interfaces, and new tensor contraction algorithms and implementations to achieve 90+% of a theoretically derived peak on GPUs. On a K40c GPU for contractions resulting in GEMMs on square matrices of size 8 for example, we are 2.8× faster than CUBLAS, and 8.5× faster than MKL on 16 cores of Intel Xeon E5-2670 (Sandy Bridge) 2.60GHz CPUs. Finally, we apply autotuning and code generation techniques to simplify tuning and provide an architecture-aware, user-friendly interface.

Keywords: Tensor contractions, Tensor HPC, GPU, Batched linear algebra, FEM, Applications

1 Introduction

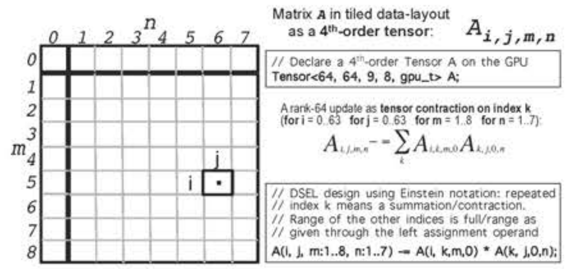
The development of high-performance tensor algebra is important due to tensors' frequent use in physics and engineering, where tensors provide a foundational mathematical tool for brief, yet comprehensive, formulations and solutions of problems in areas such as elasticity, fluid mechanics, multi-physics, quantum chemistry, general relativity, and many others [10]. Advances in microprocessors and storage technologies have made it feasible to target higher dimension and accuracy computational approaches that model mutilinear relations, e.g., in recent areas of high interest such as various data analysis applications and machine learning; that can also be formulated through tensors. At the same time, to enable these applications to efficiently

doi:10.1016/j.procs.2016.05.302

use tensor computations on current hardware, and in particular GPUs, a number of research challenges must be addressed, including advances in the development of scalable, tensor-based algorithms, autotuning, and code generation techniques, that are targets of this paper, towards setting the foundations for a high-performance tensor algebra library for accelerators [3].

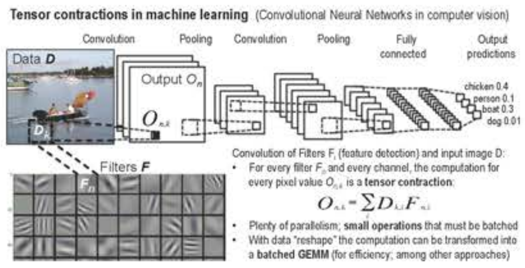
Tensors are multi-dimensional arrays that can be used to describe physical properties featuring multilinear relations. Well known mathematical objects like scalars, vectors, and matrices can be generalized to tensors that are of order zero, one, and two, respectively. Also, tensor transformations like flattening of a tensor to matrices or reshaping of matrices into tensors, can be used to link tensor computations to the developments in high-performance numerical linear algebra (LA). Therefore, similar to many applications, tensor computations can also significantly benefit from representing their computations in terms of BLAS, as well as from other dense LA algorithms and techniques for multicore and GPU architectures. While a comprehensive LAPACK-style initiative to tensors may be still far away [1], this work concentrates on the development of tensor contractions – a building block for tensor computations – through leveraging the current LA developments for GPU and multicore architectures in libraries like BLAS and MAGMA [20], and more specifically the MAGMA Batched computational framework [6, 8, 9].

Microprocessor and storage technology advances have significantly influenced the design of high-performance numerical algorithms and libraries over the years. While humans perceive well algorithms expressed in terms of scalar computations (tensors of order zero), advances towards vector machines in the 70's lead to the development of the LINPACK library to use vector operations (or 1^{st}-order tensors), which was redesigned for performance into LAPACK in the 80's to better use cache-based machines through matrix-matrix operations (2^{nd}-order tensors). Operations on higher-dimensional data were added in the 90's for distributed-memory systems, where ScaLAPACK was designed for 2D-block cyclic matrix distributions. For the emerging in the 00's multicore architectures, the PLASMA library introduced tiled algorithms and tiled data layouts (4^{th}-order tensors; see Figure 1). In the 2010's, the MAGMA libraries were designed for heterogeneous architectures, including current investigations on new data layouts, functionalities, batched computations, and possibly generalizations to tensors, in order to provide applications new functionalities to deal efficiently with multi-dimensional data.


n
0 1 2 3 4 5 6 7
m
0 1 2 3 4 5 6 7 8
i
j
Matrix A in tiled data-layout as a 4th-order tensor: $A_{i,j,m,n}$
A rank-64 update as tensor contraction on index k (for i = 0..63 for j = 0..63 for m = 1..8 for n = 1..7):
$A_{i,j,m,n} -= \sum_k A_{i,k,m,0} A_{k,j,0,n}$

```
// Declare a 4th-order Tensor A on the GPU
Tensor<64, 64, 9, 8, gpu_t> A;
```

```
// DSEL design using Einstein notation: repeated
// index k means a summation/contraction.
// Range of the other indices is full/range as
// given through the left assignment operand
A(i, j, m:1..8, n:1..7) -= A(i, k,m,0) * A(k, j,0,n);
```



Convolution of Filters F_i (feature detection) and input image D:
- For every filter F_n and every channel, the computation for every pixel value $O_{n,k}$ is a **tensor contraction**:

$$O_{n,k} = \sum_i D_{k,i} F_{n,i}$$

- Plenty of parallelism; **small operations** that must be batched
- With data "reshape" the computation can be transformed into a **batched GEMM** (for efficiency; among other approaches)

Figure 1: **Left**: Example of a 4^{th}-order tensor resulting from tile matrix layout used in dense LA, a tensor contraction, and a possible tensor contractions design using Einstein summation notation and a Domain Specific Embedded Language (or *DSEL*) . **Right**: Illustration of tensor contractions needed and viable approaches to solve them in machine learning.

Figure 1 Left illustrates the notion of tensor and tensor contraction in DLA, as well as one of our tensor contraction design using a *DSEL* . Figure 1 Right illustrates the need of tensor contractions in machine learning. The computational characteristics in this case are common

to many applications: the operations of interest are in general small, they must be batched for efficiency, and various transformations may have to be explored to transform the batched small computations to regular and therefore efficient to implement operations, e.g., GEMM.

Plans to modernize numerical libraries for up-coming supercomputers must consider the projected changes in system architectures that will define the landscape on which HPC software will execute. For example, one of the next generation of supercomputers, exceeding 150 petaflop peak performance, is aimed to be deployed by 2018 and to include hybrid nodes, coupling powerful latency-optimized processors (IBM Power9) with highly parallel throughput-optimized accelerators (NVIDIA Volta GPUs) through NVIDIA NVLink interconnect. The platforms will also benefit from 3D-stacked memories. The challenges are severe to prepare libraries for systems like this. The efforts must be multidisciplinary, incorporating LA, languages, code generations and optimizations, domain science and application-specific numerical algorithms. Of particular interest is BLAST, a high-order FEM hydrodynamics research code currently developed at LLNL. Compared to low-order methods, BLAST improves the accuracy of Arbitrary Lagrangian-Eulerian (ALE) hydrodynamic simulations and provides a viable path to extreme parallel computing and exascale architectures. A core requirement to enable this path however is the development of advanced methods and software for tensor contractions, as introduced, on GPU and multicore architectures.

2 Tensors in numerical libraries

The use of tensors in physics and engineering applications has motivated an extensive research in both algorithms for tensor algebra and computational aspects for tensor-based simulations. See the survey [10], and the references cited there, for an overview of the linear algebra aspects of tensors research, including higher-order tensor decompositions, their applications, and available software. A general overview with current and future direction in tensor research is presented in the *Future directions in tensor-based computation and modeling* workshop report [1].

There has been a number of software packages developed for tensors. Some are designed to be used through numerical computing mathematical environments like the Tensor Toolbox[1] for Matlab, GRTensor II[2] for Maple, or Ricci[3] for Mathematica, and therefore do not target accelerators, and high-performance computing in general. A few are specialized for particular applications, most notably for quantum chemical computations. For example, [12] presents early work on using accelerators to accelerate tensor contractions on GPUs. The approach uses code generation techniques and is incorporated in the NW Chem computational chemistry suite. More recent work [16] also uses code generation techniques and autotuning (with a system called Baracuda, based on CUDA-CHiLL and a high-level module Optimizing Compiler with Tensor OPeration Intelligence (OCTOPI)) to report significant acceleration compared to NW Chem on particular tensor contractions.

While we use code generation and autotuning, our approach also relies on context knowledge, and in particular that tensor reshaping techniques can often transform tensor contractions from many applications into many GEMMs, similar to the example on Figure 1, Right from machine learning. This transformation can not easily be detect automatically, and moreover, even if the transformation is somehow indicated, it is well known that general GEMM optimizations using only compiler techniques can not match in performance best-tailored solutions.

[1]http://www.sandia.gov/∼tgkolda/TensorToolbox/
[2]http://grtensor.phy.queensu.ca/
[3]http://www.math.washington.edu/∼lee/Ricci/

Similar to our approach but for CPU systems only, [18] executes contractions via GEMM on a properly ordered and structured tensor. As in the other approaches in quantum chemistry, large tensor contractions are targeted. In contrast, we target many but small contractions, that are often sub-warp/vector in size (see next section), resulting in large-scale computations. Tensor reshuffle operations are done to cast the contractions to GEMMs, when possible, and a batched approach with custom-built BLAS kernels is used to efficiently execute them.

Additional closely related efforts include [13, 17] and [19].

3 Tensor formulation for high-order FEM

We derive tensor formulations for the high-order FE kernels of the Lagrangian phase of BLAST, which solves the Euler equations of compressible hydrodynamics in a moving Lagrangian frame [5, 7]. On a semi-discrete level, the conservation laws of Lagrangian hydrodynamics can be written as:

$$\begin{aligned}
\textbf{Momentum Conservation:}\quad \mathbf{M}_{\mathcal{V}}\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} &= -\mathbf{F}\cdot\mathbf{1}, && (1)\\
\textbf{Energy Conservation:}\quad \frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t} &= \mathbf{M}_{\mathcal{E}}^{-1}\mathbf{F}^{T}\cdot\mathbf{v}\,, && (2)\\
\textbf{Equation of Motion:}\quad \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} &= \mathbf{v}, \quad \text{where} && (3)
\end{aligned}$$

$\mathbf{v}$, $\mathbf{e}$, and $\mathbf{x}$ are the unknown velocity, specific internal energy, and grid position, respectively; $\mathbf{M}_{\mathcal{V}}$ and $\mathbf{M}_{\mathcal{E}}$ 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.

To illustrate the tensor formulation of this problem, consider the finite element (FE) mass matrix for an element (zone) E with a weight ρ, as a 2-dimensional tensor:

$$(M_E)_{ij} = \sum_{k=1}^{nq} \alpha_k\, \rho(q_k)\, \varphi_i(q_k)\, \varphi_j(q_k)\, |J_E(q_k)|\,, \qquad i,j = 1,\ldots,nd, \quad \text{where}$$

nd is the number of 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, and $\{q_k\}_{k=1}^{nq}$ and $\{\alpha_k\}_{k=1}^{nq}$ are the points and weights of the quadrature rule.

Tensors can be introduced by taking the $nq \times nd$ matrix B, $B_{ki} = \varphi_i(q_k)$, and the $nq \times nq$ diagonal matrix D_E, $(D_E)_{kk} = \alpha_k\, \rho(q_k)\, |J_E(q_k)|$. Then, $(M_E)_{ij} = \sum_{k=1}^{nq} B_{ki}(D_E)_{kk}B_{kj}$, i.e.,

$$M = B^T D B \ \ (\text{omitting the element index } E)\,.$$

Using FEs of order p with a quadrature rule of order p, we have $nd = O(p^d)$ and $nq = O(p^d)$, so B is a dense $O(p^d) \times O(p^d)$ matrix. If the FE basis and the quadrature rule have tensor product structure, then in 2D,

$$M_{i_1,i_2,j_1,j_2} = \sum_{k_1,k_2} (B^{1d}_{k_1,i_1} B^{1d}_{k_1,j_1})(B^{1d}_{k_2,i_2} B^{1d}_{k_2,j_2}) D_{k_1,k_2}\,, \tag{4}$$

where B^{1d} is a dense $O(p) \times O(p)$ matrix and D is a dense $O(p) \times O(p)$ matrix. In 3D,

$$M_{i_1,i_2,i_3,j_1,j_2,j_3} = \sum_{k_1,k_2,k_3} (B^{1d}_{k_1,i_1} B^{1d}_{k_1,j_1})(B^{1d}_{k_2,i_2} B^{1d}_{k_2,j_2})(B^{1d}_{k_3,i_3} B^{1d}_{k_3,j_3}) D_{k_1,k_2,k_3}, \quad \text{where} \tag{5}$$

D is a dense $O(p) \times O(p) \times O(p)$ tensor, and $i = (i_1, \cdots, i_d)$, $j = (j_1, \cdots, j_d)$, $k = (k_1, \cdots, k_d)$ are the decompositions of the dof and quadrature point indices into the tensor components along logical coordinate axes. Note that if we store the tensors as column-wise 1D arrays, then

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

i.e. we can reinterpret M as a $nd \times nd$ matrix, or a fourth order tensor of dimensions $nd_1 \times nd_2 \times nd_1 \times nd_2$, or a vector of dimension nd^2, without changing the underlying storage.

The action of the operator, $U = MV$, can be written in the tensor product case as

$$U_{i_1,i_2} = \sum_{k_1,k_2,j_1,j_2} B^{1d}_{k_1,i_1} B^{1d}_{k_2,i_2} D_{k_1,k_2} B^{1d}_{k_1,j_1} B^{1d}_{k_2,j_2} V_{j_1,j_2}, \text{ and} \tag{6}$$

$$U_{i_1,i_2,i_3} = \sum_{k_1,k_2,k_3,j_1,j_2,j_3} B^{1d}_{k_1,i_1} B^{1d}_{k_2,i_2} B^{1d}_{k_3,i_3} D_{k_1,k_2,k_3} B^{1d}_{k_1,j_1} B^{1d}_{k_2,j_2} B^{1d}_{k_3\ ,j_3} V_{j_1,j_2,j_3}. \tag{7}$$

Given B^{1d}, D, and V, tensors M and U must be computed based on the generalized contractions (4)–(7). The matrix sizes are relatively small, e.g., with $p = 2..8$ and $d = 2$ or 3.

4 Tensor contraction interfaces and code generation

To develop high-quality HPC software for tensor contractions, we impose the following three main requirements on our interface design:

Convenience of use: Interfaces of HPC libraries are extremely important for the libraries' adoption by the community. Interfaces must provide convenience and practicality of use, including ease of interoperability between libraries. Ideally, a tensor data type standard must be defined. The standard for a dense matrix is a pointer, sizes, leading dimension, and assumption for column-major data layout, e.g., as in BLAS and LAPACK. For tensors, motivated by reviewing interfaces in available libraries and the success of BLAS, we propose to represent a tensor by its dimensions and a data layout abstraction. The abstraction is to provide a choice of predefined layouts, and ways to switch it, or to easily add user-specified layouts, without changing the underlying algorithms. In general, a specific layout provides the formula of mapping the multi-dimensional tensor to the memory, e.g., a second order tensor can be stored as a column-major matrix A with leading dimension lda, in which case the abstraction maps $A_{i,j}$ to $A[i + j * lda]$ (see Listing 1).

Readability: Numerical software must be understandable, which is needed for ease of maintenance, as well as code optimizations, and testing. While we can easily implement any interface, e.g., even expressing the interface and tensor APIs in a DSEL if needed (plus code generation techniques to translate the DSEL to a standard language; see the example in Einstein tensor notations on Figure 1 Left), we determined that it is better to provide implementations relying on a standard language and the code generation features provided within the language. The C++14 standard for example is expressive enough to allow us to implement readable and easy to use interfaces.

Performance: While we expect to obtain high performance mostly through algorithmic design and autotuning (see Section 5), we do not want to compromise on optimization opportunities like removing parameters checking and loop unrolling to eliminate jumps and loop count decrements. These optimizations are essential especially for the small computations that we target. Therefore, in our design we consider the use of compiler features related to code generation (e.g., templates, etc.), as further discussed below.

Related to performance, a lost optimization opportunity is if static checking and compile time information is not provided as part of the scientific libraries used. As an example, LAPACK routines start by checking entry parameters dynamically, which results in extra time for small size computations. The type of algorithms that we intend to use, e.g., as the MAGMA Batched algorithms, also require specific tuning [8] as performance will greatly vary depending on numerous parameters. To avoid these performance drawbacks, and also benefit from optimization techniques like compiler loop unrolling for static dimension problems, we use features of the new C++14 standard. In particular, the `constexpr` specifier enables to evaluate the value of a function or variable at compile time. We use this feature with integral constants and template specialization to design our tensor dimensions layout:

```
// Template specialization
constexpr auto layout = of_size<5,3>();
// Using Integral constant
constexpr auto layout1 = of_size(5_c,3_c);
// Using dynamic dimensions
constexpr auto layout2 = of_size(5,3);
// Access Dimensions at compile time
constexpr auto dim1 = layout(1);
```

Listing 1: Dimensions for Tensors

```
// Create a rank 2 tensor of size 5,3 on GPU
constexpr tensor<float,gpu_> d_ts(of_size<5,3>());
// Create a rank 2 tensor of size 5,3 on CPU
constexpr tensor<float> ts(of_size<5,3>());
// Use thrust to fill d_ts with 9
thrust::fill(d_ts.begin() , d_ts.end() , 9);
// Copy d_ts from GPU to ts on CPU
copy( d_ts , ts ) ;
```

Listing 2: Create Tensor and copy

As seen in Listing 1, we propose 3 ways to specify dimensions using the function *of_size* which returns a layout containing the static or dynamic dimension. Each operator inside the layout uses the `constexpr` keyword which enables us to return sizes at compile time if possible.

We can then freely extract the dimensions (Listing 1) and use them to specify our CPU and GPU kernels at compile time. Our tensor model is based on our layout, the data type of the tensor and its locality. The memory buffer is based on vector from the Standard Template Library for CPU and thrust [4] for GPU. To generate a tensor, we need to pass a data type and locality as template parameter and the size to the constructor (Listing 2).

Transfers between CPU and GPU tensors can be expressed through the function copy (Listing 2). This function will use the stream 0 by default but a stream can be passed for asynchronous copy. We have designed two models for batched computing (Listing 3). The first one is based on allocating a single memory block for all tensors to improve data transfers and locality while the other is a group of same size tensors.

```
// Create a batch that will contain 15 tensors of size 5,3,6
constexpr auto batch<float , gpu_> b = make_batch(of_size(5_c,3_c,6_c) , 15);
// Accessing a tensor from the batch returns a view on it
constexpr auto view_b = b(0);
// Create a grouping of tensors of same size tensors
constexpr auto group<float,gpu_> g(of_size(5_c,3_c));
// Add a tensor to the group
constexpr auto tensor<float,gpu_> d_ts( of_size(5_c,3_c) );
g.push_back(d_ts);
```

Listing 3: Batched tensors

Once we have defined these functions we can call the kernel to compute a batched dgemm on tensors of dimension 2.

```
constexpr auto batch<float , gpu_> b =  make_batch(of_size(5_c,3_c) , 15);
constexpr auto batch<float , gpu_> b1 = make_batch(of_size(5_c,3_c) , 15);
// Product of two tensor batched of dimension 2 for matrix product using C++ operator
constexpr auto c = b * b1;
// Product using the batch dgemm function that can be specialized depending on parameters
constexpr auto c = batch_gemm(b , b1 );
```

Listing 4: Tensor Operations

5 Algorithms design for performance and portability

The generalized tensor contractions (4)–(7) can be summarized to a few kernels [3]. One natural way to implement them is as a sequence of pairwise contractions, i.e. by evaluating the sums one at a time. While this is an efficient approach, especially when coupled with a batched approach as in MAGMA [8, 9], there is enough complexity here that maybe one can come up with something better. Indeed, a number of the kernels needed can be reduced further to a tensor index reordering (generalized transpose) plus the dgemm $A, B \mapsto A^T B$. This is because contraction by the first index, for example

$$C_{i_1,i_2,i_3} = \sum_{k_1} A_{k_1,i_1} B_{k_1,i_2,i_3},$$

can be written as $Reshape(C)^{nd_1 \times (nd_2 nd_3)} = A^T \ Reshape(B)^{nq_1 \times (nd_2 nd_3)}$. If the contraction is not by the first index, reducing it to $A^T B$ requires data reordering. However, we note that there is a way not to do this explicitly since the matrices are very small; we organize our algorithms (next) to have a phase of reading A and B to shared memory, followed by a computational phase. Thus, the reading can be from consecutive or not data, and in either case to benefit from data reuse (of the small A and B in shared memory) using the same computational phase.

In summary, all of the (6)–(7) operations can be implemented through reshaping and the kernels $A_{i,j,k,l} = \sum_s B_{i,s,j} C_{k,s,l}$ and $A_{i,k,l,j} = \sum_s B_{i,s,j} C_{k,s,l}$. The matrix assembly contractions (4)–(5) can also be reduced to a common kernel (plus reshaping): $A_{k,i,l,j} = \sum_s B_{s,i} B_{s,j} C_{k,s,l}$.

5.1 Performance model

To evaluate the efficiency of our algorithms we derive theoretical bounds for the maximum achievable performance $P_{max} = F/T_{min}$, where F is the flops needed and T_{min} is the fastest time to solution. For simplicity, consider $C = \alpha AB + \beta C$ on square matrices of size n. We have $F \approx 2n^3$ and $T_{min} = min_T(T_{Read(A,B,C)} + T_{Compute(C)} + T_{Write(C)})$. Note that we have to read/write $4n^2$ elements, or $32n^2$ Bytes for double precision (DP) calculations. Thus, if the maximum achievable bandwidth is B (in Bytes/second), we take $T_{min} = 4n^2/B$ in DP. Note that this time is theoretically achievable if the computation totally overlaps the data transfers and does not disrupt the maximum rate B of read/write to the GPU memory. Thus,

$$P_{max} = \frac{2n^3 B}{32n^2} = \frac{nB}{16} \text{ in DP.}$$

The achievable bandwidth can be obtained by benchmarks, e.g., using NVIDIA's `bandwidthTest`. For example, on a K40 GPU with ECC on the peak is 180 GB/s, so in that case P_{max} is 11.25 n GFlop/s (denoted by the dashed line on Figure 3). Thus, when $n = 16$ for example, we expect a theoretical maximum performance of 180 GFlop/s in DP.

5.2 Algorithms for tensor contractions through batched GEMMs

As described, we transform the tensor contractions to batched GEMM. To achieve HP on batched GEMM for small matrices that are sub-warp in size, we apply the following techniques:

- Using templates and `constexpr` specifiers we manage to avoid parameters checking and get compiler-unrolled loops in our kernels;

- We avoid passing arrays of pointers to the batched matrices, which are quite large, by passing them through formula/function that is part of the tensor's structure definition;

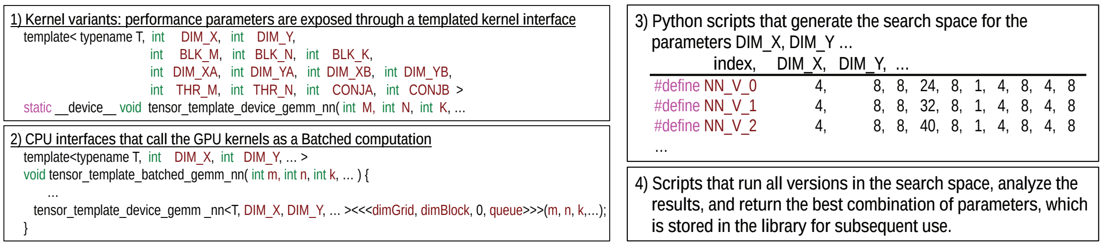
- The kernels are organized as follows: (1) Read A and B into shared memory; (2) Compute AB in registers; (3) Update C. Reading A, B, and C is through functions in the tensor's structure definition, which allows us to work with matrices that are not stored in the standard matrix format. Thus, we do not need explicit data reordering for some contractions, in order to efficiently benefit from the GEMM computation (step 2 above);

- We developed versions based on: how A and B are read; prefetching or not C by intermixing reading C with the computation in (2); number of matrices handled by a thread block, combined with prefetching variations for A and B; number of threads per thread block; and combinations of the above.

In all cases, to maximize data reuse, a single GEMM is done on a single thread block. Batching of the computation is done as in [9]. A GPU GEMMs technique, used for large matrices since the Fermi architecture [15], is to apply hierarchical communications/blocking on all memory levels, including a new register blocking. Register blocking can be added by making a single thread compute more than one element of a resulting matrix (in the same row or column, so that when an element from A or B is loaded into a register, it gets used more than once). Current results show that register blocking may not be needed in this case.

5.3 Autotuning

The complexity of tuning our algorithms is handled through autotuning [11, 2]. We note that although tuning is important, the algorithmic design (as in Section 5.2) is the more critical part for obtaining high-performance, as the overall success is limited by the quality of the algorithms used. With that in mind, there are generally two kinds of approaches for doing autotuning – model-driven optimization and empirical optimization. We use a combination. The model-driven part comprises compiler code generation and optimizations (as in Section 4). For the empirical optimization, a large number of parametrized code variants are generated and run on a given platform to discover the one that gives the best performance. The effectiveness of empirical optimization depends on the chosen parameters to optimize, and the search heuristic used. The algorithm designs and implementations using templates are very convenient in setting up our autotuning framework. The process is illustrated on Figure 2.



1) Kernel variants: performance parameters are exposed through a templated kernel interface

```
template< typename T, int  DIM_X, int  DIM_Y,
                      int  BLK_M, int BLK_N, int  BLK_K,
                      int DIM_XA, int DIM_YA, int DIM_XB, int DIM_YB,
                      int  THR_M, int THR_N, int  CONJA, int  CONJB >
static __device__ void tensor_template_device_gemm_nn( int M, int N, int K, ...
```

2) CPU interfaces that call the GPU kernels as a Batched computation

```
template<typename T, int  DIM_X, int  DIM_Y, ... >
void tensor_template_batched_gemm_nn( int m, int n, int k, ... ) {
     ...
  tensor_template_device_gemm _nn<T, DIM_X, DIM_Y, ... ><<<dimGrid, dimBlock, 0, queue>>>(m, n, k,...);
}
```

3) Python scripts that generate the search space for the parameters DIM_X, DIM_Y ...

```
                index,   DIM_X,  DIM_Y,  ...
#define NN_V_0      4,        8,  8,  24,  8,  1,  4,  8,  4,  8
#define NN_V_1      4,        8,  8,  32,  8,  1,  4,  8,  4,  8
#define NN_V_2      4,        8,  8,  40,  8,  1,  4,  8,  4,  8
...
```

4) Scripts that run all versions in the search space, analyze the results, and return the best combination of parameters, which is stored in the library for subsequent use.

Figure 2: Autotuning framework for high-performance tensor contractions in MAGMA.

Further details on the autotuning framework can be found in [2], while details on the kernels, including their extension to multicore CPUs are available in our technical report [14].

6 Experiments on tensor computations

Figure 3 shows the DP performance of four of our autotuned kernels from Section 5.2. Each of the kernels gives best results for certain dimensions. Shown is also the performance on 16

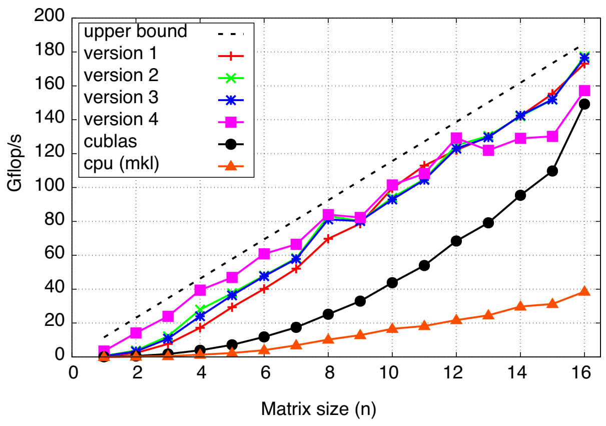


Figure 3: Performance comparison of tensor contraction versions using batched $C = \alpha AB + \beta C$ on 100,000 square matrices of size n (x-axis) on a K40c GPU and 16 cores of Intel Xeon E5-2670 (Sandy Bridge) 2.60 GHz CPUs.

cores Intel Sandy Bridge CPUs. The implementation uses OpenMP to run in parallel a GEMM per thread, where the GEMMs call MKL. For $n = 16$ the speedup is about $4\times$ and grows for smaller sizes. Similar trend is observed in a comparison to the batched DGEMM in CUBLAS. Our performance is within 90% of the theoretical maximum, as derived in Section 5.1, and denoted by a dashed line. Slight improvement may be possible through register blocking and tuning.

7 Conclusions and future directions

We presented an approach based on tensors for high-order FEM and we described it as a multidisciplinary effort that includes a tensor formulation of the numerical algorithms of a domain science problem, the definition of tensor interfaces and code generation, and the design of algorithms design and tuning for performance and portability. We developed a proof-of-concept software for it, showing that it has the potential to get optimal performance. We currently achieve within 90% of a theoretically derived peak on GPUs, using on-the-fly tensor reshaping to cast tensor contractions to small but many GEMMs, executed using a batched computing approach, custom-built GEMM kernels for small matrices, and autotuning.

As future work, there are many options that need to be coded and tuned. The resulting HP tensor contractions package will be released as open source through the MAGMA library. It remains to integrate the developments in the BLAST code, and complete the autotuning in anticipation for easy portability to future supercomputers like Sierra, and new GPU architectures like the Volta with its 3D-stacked memory, expected to further favor tensor computations.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. CSR 1514286 and ACI-1339822, NVIDIA, the Department of Energy, and in part by the Russian Scientific Foundation, Agreement N14-11-00190. This work was further 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-CONF-681073-DRAFT.

References

[1] Future directions in tensor-based computation and modeling, May 2009. Workshop. Arlington, Virginia. Technical report published at http://www.cs.cornell.edu/CV/TenWork/FinalReport.pdf.

[2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Performance, Design, and Autotuning of Batched GEMM for GPUs. In *ISC High Performance (ISC 2016), to appear*, Frankfurt, Germany, 06-2016 2016.

[3] M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov. Towards a high-performance tensor algebra package for accelerators. http://computing.ornl.gov/workshops/SMC15/presentations/. Smoky Mountains Computational Sciences and Engineering Conference (SMC'15), Gatlinburg, TN, Sep 2015.

[4] N. Bell and J. Hoberock. Thrust: A 2 6. *GPU Computing Gems Jade Edition*, page 359, 2011.

[5] Veselin Dobrev, Tzanio V. Kolev, and Robert N. Rieben. High-order curvilinear finite element methods for lagrangian hydrodynamics. *SIAM J. Scientific Computing*, 34(5), 2012.

[6] T. Dong, A. Haidar, P. Luszczek, A. Harris, S. Tomov, and J. Dongarra. LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU. In *HPCC 2014*, August 2014.

[7] Tingxing Dong, Veselin Dobrev, Tzanio Kolev, Robert Rieben, Stanimire Tomov, and Jack Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.

[8] Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *IJHPCA*, doi:10.1177/1094342014567546, 02/2015.

[9] Azzam Haidar, Tingxing Dong, Stanimire Tomov, Piotr Luszczek, and Jack Dongarra. A Framework for Batched and GPU-Resident Factorization Algorithms Applied to Block Householder Transformations. In *High Performance Computing*, volume 9137 of *Lecture Notes in Computer Science*, pages 31–47. 2015.

[10] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 51(3):455–500, August 2009.

[11] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 2009 International Conference on Computational Science, ICCS'09*, Baton Roube, LA, May 25-27 2009. Springer.

[12] Wenjing Ma, S. Krishnamoorthy, O. Villay, and K. Kowalski. Acceleration of Streamed Tensor Contraction Expressions on GPGPU-Based Clusters. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 207–216, Sept 2010.

[13] Stefano Markidis, Jing Gong, Michael Schliephake, Erwin Laure, Alistair Hart, David Henty, Katherine Heisey, and Paul F. Fischer. Openacc acceleration of the nek5000 spectral element code. *IJHPCA*, 29(3):311–319, 2015.

[14] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and Jack Dongarra. High-performance matrix-matrix multiplications of very small matrices. Technical Report UT-EECS-16-740, 03-2016 2016.

[15] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An Improved Magma Gemm For Fermi Graphics Processing Units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, November 2010.

[16] Thomas Nelson, Axel Rivera, Prasanna Balaprakash, Mary W. Hall, Paul D. Hovland, Elizabeth R. Jessup, and Boyana Norris. Generating efficient tensor contractions for gpus. Technical report, 2015.

[17] Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, Paul F. Fischer, and Paul D. Hovland. Speeding up nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 253–262, New York, NY, USA, 2010. ACM.

[18] Edgar Solomonik, Devin Matthews, Jeff Hammond, John Stanton, and James Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. Technical Report UCB/EECS-2014-143, EECS Department, University of California, Berkeley, Aug 2014.

[19] Kevin Stock, Thomas Henretty, Iyyappa Murugandi, P. Sadayappan, and Robert J. Harrison. Model-driven simd code generation for a multi-resolution tensor kernel. In *IPDPS*, pages 1058–1067. IEEE, 2011.

[20] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010. DOI: 10.1016/j.parco.2009.12.005.