FASTMAT: A C++ LIBRARY FOR MULTI-INDEX ARRAY COMPUTATIONS

Rodrigo R. Paz[a], Mario A. Storti[a], Lisandro D. Dalcín[a], Hugo G. Castro[a,b] and Pablo A. Kler[c]

[a]*Centro Internacional de Métodos Computacionales en Ingeniería (CIMEC), INTEC(CONICET-UNL), Santa Fe, Argentina {mario.storti,dalcinl,rodrigo.r.paz}@gmail.com, sergio@cimne.upc.edu,*
`http://www.cimec.org.ar/mstorti`

[b]*Grupo de Investigación en Mecánica de Fluidos, Universidad Tecnológica Nacional, Facultad Regional Resistencia, Chaco, Argentina.*

[c]*Central Division of Analytical Chemistry (ZCH). Forschungszentrum Jülich GmbH, Germany.*

Keywords: Multi-index Array Library; Finite Element Method; Finite Volume Method; Parallel Computing.

Abstract. In this paper we introduce and describe an efficient thread-safe matrix library for computing element/cell residuals and Jacobians in Finite Elements and Finite Volumes-like codes. The library provides a wide range of multi-index tensor operations that are normally used in scientific numerical computations. The library implements an algorithm for choosing the optimal computation order when a product of several tensors is performed (i.e., the so-called ‘multi-product’ operation). Another key-point of the *FastMat* approach is that some computations (for instance the optimal order in the multi-product operation mentioned before) are computed in the first iteration of the loop body and stored in a *cache* object, so that in the second and subsequent executions these computations are retrieved from the cache, and then not recomputed. The library is open source and freely available within the multi-physics parallel FEM code PETSc-FEM **`http://www.cimec.org.ar/petscfem`** and it can be exploited on distributed and shared memory architectures as well as in hybrid approaches. We studied the performance of the library in the context of typical FEM tensor contractions.

1 INTRODUCTION

A variety of engineering applications and scientific problems related to Computational Mechanics (CM) area, and particularly in the Computational Fluid Dynamics (CFD) field, demand high computational resources (4; 8; 9; 17). A great effort has been made over the years to get high quality solutions (15) to large-scale problems in realistic time (12; 5) using different computing architectures (e.g., vector processors, distributed and shared memories, graphic process units or GPGPU's).

Most of the known physical phenomena are described mathematically in terms of tensor operations. Mathematical equations can involve simple expressions such as vector (tensors of rank one) summations to complex partial differential equations (PDE's). Furthermore, when modeling PDE problems using FEM and FVM it is expected to perform several tensor contractions repeatedly to compute element (or cell) contributions to global matrices (see section§3). As these computations are made at the elements loop it is mandatory to compute the contribution terms with high efficiency. This is a typical situation where FEM (or FVM) applications require the efficiency of a compiled code for processing huge amounts of data in deeply-nested loops. Fortran (especially Fortran 90 and above) and C++ are languages for efficiently implementing lengthy computations involving multi-dimensional tensor operations.

To the best of our knowledge, there are four projects related to tensor computing. The Multidimensional Tensor Library (TL) (11) is a tensor API designed to be used in the context of perturbation methods for solving Dynamic Stochastic General Equilibrium (DSGE) models. The aim of this library is not to provide a general interface to multidimensional linear algebra but providing the definition and creation of dense and sparse multidimensional tensor for specialized formulas in DSGE area. This tensor library is a part of Dynare code (11). Another project is the FTensor (13) package, which is a high performance tensor library written in C++. It provides a set of classes allowing abstraction in tensor algebra and delivering uncompromising efficiency. It uses template expressions to provide expressiveness and speed. The third project is a C++ class library for general scientific computing called Blitz++ (23). Blitz++ provides performance by exploiting class templates, optimizations such as loop fusion, unrolling, tiling, and algorithm specialization that can be performed automatically at compile time. The current versions provide dense arrays and vectors, random number generators, and small vectors and matrices. The fourth project is Eigen, a C++ template library for linear algebra (6) wich provides containers for matrices and vectors, a set of numerical solvers, and a variety of tensor operations. Fixed-size matrices are fully optimized in Eigen avoiding dynamic memory allocation and in the case of large matrices cache-friendliness capability is exploited.

There are many differences among these libraries and the approach proposed here, i.e. the FastMat library. In this sense the former are suited for sparse or dense tensors independently of the dimension of each tensor index (or index range), while the latter is specialized for generally dense FEM-like tensors/arrays defined at the element/cell level. Furthermore, the FastMat approach has been designed with the aim to perform matrix computations efficiently at the element level. One of the main feature is its thread-safety compliance so that it can be used in an Symmetric Multi-Processing (SMP) environment within OpenMP parallel blocks. Regarding the efficiency there is an operation caching mechanism which will be described in next sections. For FEM/FVM computations it is assumed that the code has an outer loop (usually the loop over elements) that is executed many times (as big as the number of elements/cell in the mesh/grid), and at each loop execution a set of operations or tensor products (depending on the PDE and discretization at hand) are performed with a reduced number of local (or element) vectors and matrices. So that, the goal of the FastMat approach is that in loop-repetitive computations the subsequent operations of involved tensors are cached making it an efficient tool.

Contrary to the multi-dimensional arrays approach of Fortran 90 compilers, writing dimension-independent (2D, 3D) and element-independent (triangles, quadrilaterals, tetrahedra, hexahedra, prisms) routines directly in C++ is generally a hard task. In this regard, as shown in sections §2.5 and §3, the FastMat approach for contraction of several tensors with different dimensions can be made in a single line of code, providing simplicity, clarity and elegance to the element-level routines.

The remainder of this article is organized as follows. Section §2 introduces and describes the FastMat general tensor algebra library for computing element residuals and Jacobians in the context of multi-threaded finite element codes. Also, the performance of FastMat is compared against other libraries such as those listed above. Section §3 presents a test based on computing typical element or edge-wise stabilization terms for general advection-diffusion systems of equations by means of FE/FV methods. The performance of the library is studied and the order in which multi-tensor products (or contractions) must be computed to reduce operation counts is discussed using a heuristic and exhaustive (optimal) algorithms (2). Concluding remarks are given in section §4.

2 THE *FASTMAT* MATRIX CLASS

2.1 Preliminaries

Finite element codes usually have two levels of programming. In the outer level a large vector describes the "*state*" of the physical system. Usually the `size` of this vector is the number of nodes times the number of fields minus the number of constraints (e.g. Dirichlet boundary conditions). So that the state vector size is $N_{\rm nod} \cdot n_{\rm dof} - n_{\rm constr}$. This vector can be computed at once by assembling the right hand side (RHS) and the stiffness matrix in a linear problem, iterated in a non-linear problem or updated at each time step through solution of a linear or non-linear system. The point is that at this outer level all global assemble operations, that build the residual vector and matrices, are performed. At the inner level, one performs a loop over all the elements in the mesh, compute the RHS vector and matrix contributions of each element and assemble them in the global vector/matrix. From one application to another, the strategy at the outer level (linear/non-linear, steady/temporal dependent, etc) and the physics of the problem that defines the FEM matrices and vectors may vary.

The FastMat matrix class has been designed in order to perform matrix computations efficiently at the element level. One of the key points in the design of the matrix library is the characteristic of being thread-safe so that it can be used in an SMP environment within OpenMP parallel blocks. In view of efficiency there is an operation caching mechanism which will be described later. Caching is also thread-safe provided that independent cache contexts are used in each thread.

It is assumed that the code has an outer loop (usually the loop over elements) that is executed many times, and at each execution of the loop a series of operations are performed with a rather reduced set of local (or element) vectors and matrices.

In many cases, FEM-like algorithms need to operate on *sub-matrices* i.e., columns, rows or sets of them. In general, performance is degraded for such operations because there is a certain amount of work needed to extract or set the sub-matrix. Otherwise, a copy of the row or column in an intermediate object can be made, but some overhead is expected due to the copy operations.

The particularity of FastMat is that at the first execution of the loop many quantities used in the operation are stored in an internal *cache* object, so that in the second and subsequent executions of the loop these quantities are retrieved from the cache. This C++ library is in public domain and can be accessed from (18).

2.1.1 An introductory example.

Consider the following simple example: A given 2D finite element mesh composed by triangles, i.e. an array xnod of $2 \times N_{\rm nod}$ doubles with the node coordinates and an array icone with $3 \times n_{\rm elem}$ elements with node connectivities. For each element $0 \leq j < n_{\rm elem}$ its nodes are stored at icone[3*j+k] for $0 \leq k < 2$. For instance, it is required to compute the maximum and minimum value of the area of the triangles. This is a computation which is quite similar to those found in FEM analysis. For each element in the mesh two basic operations are needed: i) loading the node coordinates in local vectors $\mathbf{x}_1$, $\mathbf{x}_2$ and $\mathbf{x}_3$, ii) computing the vectors along the sides of the elements $\mathbf{a} = \mathbf{x}_2 - \mathbf{x}_1$ and $\mathbf{b} = \mathbf{x}_3 - \mathbf{x}_1$. The area of the element is, then, the determinant of the 2×2 matrix $\mathbf{J}$ formed by putting $\mathbf{a}$ and $\mathbf{b}$ as rows.

The FastMat code for the proposed computations is shown in Listing (1)

```
1   FastMat::CacheCtx ctx;
2   FastMat::CacheCtx::Branch b1;
3   FastMat x(&ctx,2,3,2),a(&ctx,1,2),b(&ctx,1,2),J(&ctx,2,2,2);
4   double starttime = MPI_Wtime(); // timing
5   for (int ie=0; ie<nelem; ie++) { // loop over elements
6     ctx.jump(b1);
7     for (int k=1; k<=3; k++) {
8       int node = icone[3*ie+(k-1)];
9       x.ir(1,k).set(&xnod[2*(node-1)]).rs();
10    }
11    x.rs();
12    a.set(x.ir(1,2));
13    a.minus(x.ir(1,1));
14
15    b.set(x.ir(1,3));
16    b.minus(x.ir(1,1));
17
18    J.ir(1,1).set(a);
19    J.ir(1,2).set(b);
20
21    double area = J.rs().det()/2.;
22    total_area += area;
23    if (ie==0) {
24      minarea = area;
25      maxarea = area;
26    }
27    if (area>maxarea) maxarea=area;
28    if (area<minarea) minarea=area;
29  }
30  printf("total_area %g, min area %g,max area %g, ratio: %g\n",total_area,
        minarea,maxarea,maxarea/minarea);
31  double elapsed = MPI_Wtime()-starttime; // elapsed time
```

Listing 1: Simple FEM-like code

Calls to the FastMat::CacheCtx ctx object are related to the caching manipulation and will be discussed later. Matrices are dimensioned in line 3, the first argument is the matrix (*rank*), and then, follow the dimensions for each index rank or *shape*. For instance FastMat x(2,3,2) defines a matrix of rank 2 and *shape* (3,2), i.e., with 2 indices ranging from 1 to 3, and 1 to 2 respectively. The rows of this matrix will store the coordinates of the local nodes to the element. FastMat matrices may have any number of indices or rank. Also they can have zero rank, which stands for scalars.

2.1.2 Current matrix views (the so-called 'masks').

In lines 7 to 10 of code Listing (1) the coordinates of the nodes are loaded in matrix `x`. The underlying philosophy in FastMat is that "*views*" (or "*masks*") of the matrix can be made without making any copies of the underlying values. For instance the operation `x.ir(1,k)` (for "*index restriction*") sets a view of `x` so that index 1 is restricted to take the value k reducing in one the rank of the matrix. As `x` has two indices, the operation `x.ir(1,k)` gives a matrix of rank one consisting of the k-th row of x. A call without arguments like in `x.ir()` cancels the restriction. Also, the function `rs()` (for "*reset*") cancels the actual view. Please, refer to the appendix section §5.1 for a synopsis of methods/operations available in the FastMat class.

2.1.3 Set operations.

The operation `a.set(x.ir(1,2))` copies the contents of the argument `x.ir(1,2)` in `a`. Also, `x.set(xp)` can be used, being `xp` an array of doubles (`double *xp`).

2.1.4 Dimension matching.

The `x.set(y)` operation, where `y` is another FastMat object, requires that `x` and `y` have the same "masked" dimensions. As the `.ir(1,2)` operation restricts index to the value of 2, `x.ir(1,2)` is seen as a row vector of size 2 and then can be copied to a. If the "masked" dimensions do not fit then an error is issued.

2.1.5 Automatic dimensioning.

In the example, `a` has been dimensioned at line 3, but most operations perform the dimensioning if the matrix has not been already dimensioned. For instance, if at line 3 a `FastMat a` is declared without specifying dimensions, then at line 12, the matrix is dimensioned taking the dimensions from the argument. But this does not apply to `set(double *)` since in this last case the argument (`double *`) does not give information about his dimensions. Other operations that define dimensions are products and contraction operations.

2.1.6 Concatenation of operations.

Many operations return a reference to the matrix (return value `FastMat &`) so that operations may be concatenated as in `A.ir(1,k).ir(2,j).`

2.2 Underlying implementation with BLAS/LAPACK

Some functions are implemented at the low level using BLAS(7)/LAPACK(14). Notably `prod()` uses BLAS3's *General Matrix Multiply* `dgemm` for large n ($n > n_{\max}$, with $n_{\max}$ configurable, usually $n_{\max} = 10$) and an FMGEMM function for $n < n_{\max}$. FMGEMM is a special completely unrolled version of `dgemm`, to be explained in detail later (see section §2.4).

So that the amortized cost of the prod() call is the same as for the underlying version of dgemm() plus an overhead which is not significative, except for $n < n_{\max}$.

As a matter of fact, a profiling study of FastMat efficiency in a typical FEM code has determined that the largest CPU consumption in the residual/Jacobian computation stage corresponds to prod() calls. Another notable case is eig() that uses LAPACK dgeev. The eig() method is not commonly used, but if it does, its cost may be significant so that a fast implementation as proposed here with dgeev is mandatory.

2.3 The FastMat operation cache concept

The idea with caches is that they are objects (class FastMatCache) that store information that can be computed in advance for the current operation. In the first pass through the body of the loop (i.e., ie=0 in the example of Listing (1)) a cache object is created for each of the operations, and stored in a list. This list is basically a vector (vector< >) of cache objects. When the body of the loop is executed the second time (i.e., ie>1 in the example) and the following, these values are not needed to be recomputed but they are read from the cache object instead. The use of the cache object is rather automatic and requires little intervention by the user but in some cases the position in the cache-list can get out of synchronization with respect to the execution of the operations and severe runtime errors may occur. This cache structure is similar to the *visualization pipeline* used in graphics libraries like *VTK* (see (16)).

The basic use of caching is to create the cache structure FastMat::CacheCtx ctx and keep the position in the cache structure *synchronized* with the position of the code. The process is very simple, when the code consists in a linear sequence of FastMat operations that are executed always in the same order. In this case the CacheCtx object stores a list of the cache objects (one for each FastMat operation). As the operations are executed the internal FastMat code is in charge of advancing the cache position in the cache list automatically. A linear sequence of cache operations that are executed *always* in the same order is called a *branch*.

Looking at the previous code (Listing (1)), it has one branch starting at the x.ir(1,k).set(...) line, through the J.rs().det() line. This sequence is repeated many times (one for each element) so that it is interesting to *reuse the cache list*. For this, a *branch* object b1 (class FastMat::CacheCtx::Branch) and a *jump* to this branch are created each time a loop is executed. In the first loop iteration the cache list is created and stored in the first position of the cache structure. In the next and subsequent executions of the loop, the cache is reused avoiding recomputing many administrative work related with the matrices.

The problem is when the sequence of operations is not always the same. In that case several jump() commands must be issued, each one to the start of a sequence of FastMat operations. Consider for instance the following code,

```
FastMat::CacheCtx ctx;
FastMat::CacheCtx::Branch b1, b2;
FastMat x(&ctx,1,3);
ctx.use_cache = 1;
int N=10000, in=0, out=0;

for (int j=0; j<N; j++) {
  ctx.jump(b1);
  x.fun(rnd);
  double len = x.norm_p_all();
  if (len<1.0) {
    in++;
    ctx.jump(b2);
    x.scale(1.0/len);
  }
}
printf("total %d, in %d (%f%%)\n",N,in,double(in)/N);
```

A vector `x` of size 3 is randomly generated in a loop (the line `x.fun(rnd);`). Then its length is computed, and if it is shorter than 1.0 it is scaled by `1.0/len`, so that its final length is one. In this case two branches are defined and two jumps are executed,

- branch `b1`: operations `x.fun()` and `x.norm_p_all()`,
- branch `b2`: operation `x.scale()`.

2.3.1 Thread-Safety and reentrancy.

If caching is not enabled, FastMat is reentrant and then thread-safe. If caching is enabled, then it is reentrant in the following sense, a context `ctx` must be created for each thread, and the matrices used in each thread must be associated with the context of that thread.

If creating the cache structures each time is too bad for efficiency, then the context and the matrices may be used in a parallel region, stored in variables, and reused in a subsequent parallel region.

2.3.2 Caching loop repetitive computations

If *caching* is not used the performance of the library is poor while the cached version is very fast, in the sense that almost all the CPU time is spent in performing multiplications and additions, and negligible CPU time is spent in auxiliary operations.

2.3.3 Branching is not always needed.

However, branching is needed *only* if the instruction sequence changes during the same execution of the code. For instance, if a code like follows is considered

```
FastMat::CacheCtx ctx;
ctx.use_cache=1;
for (int j=0; j<N; j++) {
  ctx.jump(b1);
  // Some FastMat code...
  if (method==1) {
     // Some FastMat code for method 1 ...
  } else if (method==2) {
     // Some FastMat code for method 2 ...
  }
  // More FastMat code...
}
```

the `method` flag is determined at the moment of reading the data and then is left unchanged for the whole execution of the code, then it is not necessary to “jump()” since the instruction sequence will be always the same.

2.3.4 Cache mismatch.

The caching process may fail if a *cache mismatch* is produced. For instance, consider the following variation of the previous code

```
1  FastMat::CacheCtx ctx;
2  FastMat::CacheCtx::Branch b1, b2, b3;
3  //...
4  for (int j=0; j<N; j++) {
5    ctx.jump(b1);
6    x.fun(rnd);
7    double len = x.norm_p_all();
8    if (len<1.0) {
9      in++;
10     ctx.jump(b2);
11     x.scale(1.0/len);
12   } else if (len>1.1) {
13     ctx.jump(b3);
14     x.set(0.0);
15   }
16 }
```

There is an additional block in the conditional, if the length of the vector is greater than 1.1, then the vector is set to the null vector.

Every time that a branch is opened in a program block a `ctx.jump()` must be called using different arguments for the branches (i.e., `b1`, `b2`, etc). In the previous code there are three branches. The code shown is correct, but assume that the user forgets the `jump()` calls at lines 10 and 13 (sentences `ctx.jump(b2)` and `ctx.jump(b3)`), then when reaching the `x.set(0.0)`, the operation in line 14, the corresponding cache object would be that one corresponding to the `x.scale()` operation (line 11), and an incorrect computation will occur.

Each time that the retrieved cache does not match with the operation that will be computed or even when it does not exist a *cache mismatch* exception is produced.

2.3.5 Branch arrays.

In some computing cases, when branching implies a certain number of branches, a branch object is needed for each branch (doing it by hand could be cumbersome). So that, it can either created defining a plain STL (1) `vector<>` of branches or using the `FastMat2::CacheCtx2::Branchv` class as shown in Listing (2),

```
FastMat2::CacheCtx2 ctx;
FastMat2::CacheCtx2::Branch b1;
FastMat2::CacheCtx2::Branchv b2v(5);
FastMat2 x(&ctx,2,5,3);
int N = 10000;
ctx.use_cache = 1;
double sum=0;
for (int j=0; j<N; j++) {
  ctx.jump(b1);
  x.fun(rnd);
  int k = rand()%5;
  ctx.jump(b2v(k));
  x.ir(1,k+1);
  sum += x.norm_p_all();
  x.rs();
}
ctx.use_cache = 0;
```

Listing 2: Branch Arrays example.

At each iteration of the loop a matrix (`x` of 5x3) is randomly generated. Then, a row of it randomly picked and its norm is computed and accumulated on variable `sum`. If the line `ctx.jump(b2v(k))` is omitted in the code, then there will be only the main `b1` branch, and when the `norm_p_all()` instruction is executed it can be reached with a different value of `k`. It will not give an error but it will compute with the `k` stored in the cache object.

As mentioned above one can either define a plain STL `vector<>` of branches or use the `FastMat2::CacheCtx2::Branchv` class, as shown in previous code (2). The branch array can be created passing its dimension either, at the constructor as in code (2) or in the `init` method,

```
FastMat2::CacheCtx2::Branchv b2v;
// ... later...
b2v.init(5);
```

Listing 3: Branch Arrays.

Multi dimensional arrays can be created giving extra integer arguments.

2.3.6 Causes for a cache mismatch error.

In the first iteration of the computing loop, the cache is built with information obtained from the matrices and operations involved in the computation. At subsequent iterations the information of the current objects must coincide with those stored in the cache, that is

- The FastMat matrices involved must be the same, (i.e. their pointers to the matrices must be the same).
- The indices passed in the operation must coincide (for instance for the `prod()`, `ctr`, `sum()` operations).
- The masks (see §2.1.2) applied to each of the matrix arguments must be the same.

2.4 Efficiency

This benchmark computes the matrix product $C_{ij} = A_{ik}B_{kj}$ in a loop for a large number N of distinct square matrices A, B, C of varying size n, starting at $n = 2$. As mentioned before the amortized cost is the cost of the underlying `dgemm()` call plus an overhead due to the FastMat layer. The processing rate in Gflops is computed on the base of an operation count per matrix product, i.e. $2n^3$

$$\text{rate [Gflops]} = 10^{-9}\frac{N \cdot 2n^3}{\text{(elapsed time [secs])}} \tag{1}$$

The FastMat overhead is composed of a large overhead t_0 that is done once while building the cache structure for subsequent operations and a small overhead t_1 for each matrix product. Normally it is reduced to a few function calls, so the total computing time is

$$T = t_0 + N(t_1 + t_{\text{dgemm}}), \tag{2}$$

where t_{dgemm} is the computing time of the underlying `dgemm()` call. The first overhead (t_0) is amortized when the loop is executed a large number of times N, as it is usual in large CFD computations. As a reference the number of times for reaching a 50% amortization ($N_{1/2}$) is in the order of 15 to 30 (of course it could depend on many factors), that means that typicallly for $N = 300$, the overhead is 10% or less.

Related to the underlying implementation of `dgemm()`, several options were tested on an Intel Core i7–950 @ 3.07GHz, namely

- The BLAS implementation included in *Intel's Math Kernel Library* (version 137-12.0-2).
- The ATLAS implementation of BLAS (see (3; 24), version 3.8.3, Fedora RPM binary package). It was compared also with ATLAS self-configured and compiled from the sources, but the computing times were almost identical to those from the RPM version, so that we report only those for the RPM version.
- The *FMGEMM* library, which are fully unrolled versions of the product matrices. The code is generated with a Perl script (see the Listing 4 below for the case of a $3 \times 3 \times 3$ product. Note that if $n_{\max} = 10$ for instance, we generate $n_{\max} \times n_{\max} \times n_{\max} \times 4$ functions whose names are `p_N_M_P_XY()` where the integers `N`, `M`, `P`, are the matrix sizes and the two letters `XY` denote if the matrix is transposed or not, for instance `p_3_4_5_nt` corresponds to the product $\mathbf{C} = \mathbf{A}\mathbf{B}^t$ with $\mathbf{A}$, and $\mathbf{B}$ of sizes 3×4 and 5×4 respectively. For $n_{\max} = 10$ the source code so generated is approximately 2MB in length. Note also that in FMGEMM approach the matrices are constrained to have constant stride, otherwise we should generate code for each stride bearing more *code bloating*.

```
void p_3_3_3_nn(double*__restrict__ a, double*__restrict__ b,
                double*__restrict__ c) {
  c[0] = a[0]*b[0]+a[1]*b[3]+a[2]*b[6];
  c[1] = a[0]*b[1]+a[1]*b[4]+a[2]*b[7];
  c[2] = a[0]*b[2]+a[1]*b[5]+a[2]*b[8];
  c[3] = a[3]*b[0]+a[4]*b[3]+a[5]*b[6];
  c[4] = a[3]*b[1]+a[4]*b[4]+a[5]*b[7];
  c[5] = a[3]*b[2]+a[4]*b[5]+a[5]*b[8];
  c[6] = a[6]*b[0]+a[7]*b[3]+a[8]*b[6];
  c[7] = a[6]*b[1]+a[7]*b[4]+a[8]*b[7];
  c[8] = a[6]*b[2]+a[7]*b[5]+a[8]*b[8];
}
```

Listing 4: *FMGEMM* unrolled code

Figure (1) shows the computing rates using equation (1) for n small ($n \leq 20$). It can be seen that for MKL, ATLAS, Eigen, and Blitz libraries a large degradation of performance for low n is obtained, going all four under 0.25 Gflops for $n = 2$, being Blitz the most performant. The FMGEMM implementation is notably more efficient, reaching 1.6 Gflops, when using directly the functions like the `p_3_3_3_nn()` shown above. When used inside FastMat, the overhead t_1 degrades the performance significantly to 0.41 Gflops. That means that the overhead from FastMat is almost 75%, however, even so the FastMat `prod` function member is twice faster than the Blitz implementation. In fact, the overhead would be much larger if we would produce the dispatch to the appropriate FMGEMM function `p_N_M_P_XY()` in runtime for each execution of the loop. Of course the pointer to the appropriate function is computed outside the loop and stored in the cache object. This is another example that shows the benefits of the cache strategy used in FastMat.

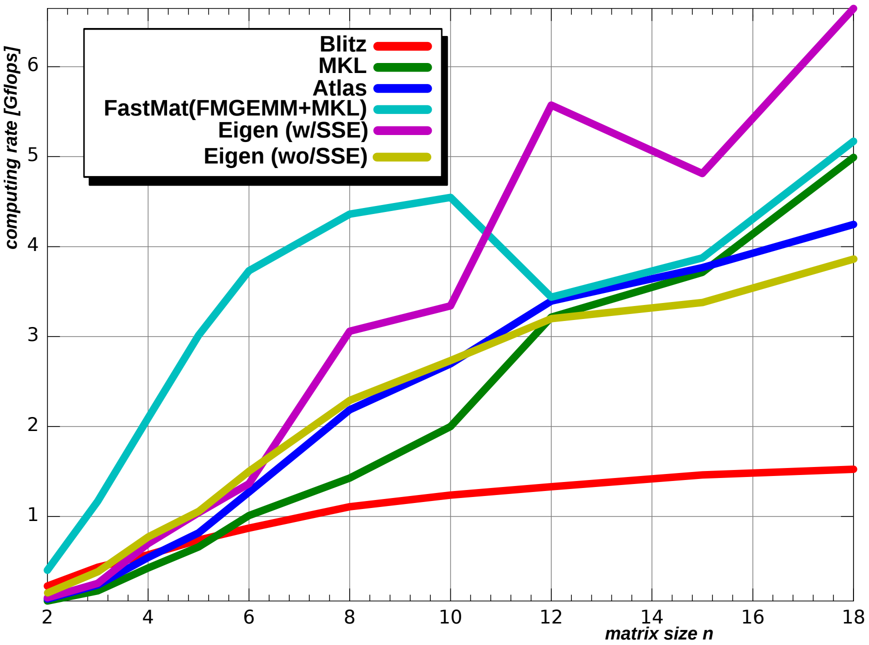

Figure 1: Efficiency comparatives for the matrix product $\mathbf{C} = \mathbf{AB}$ on an Intel Core i7–950 @ 3.07GHz, for square matrices of small size $n \leq 20$. Vertical axis is computing speed as given per equation (1).

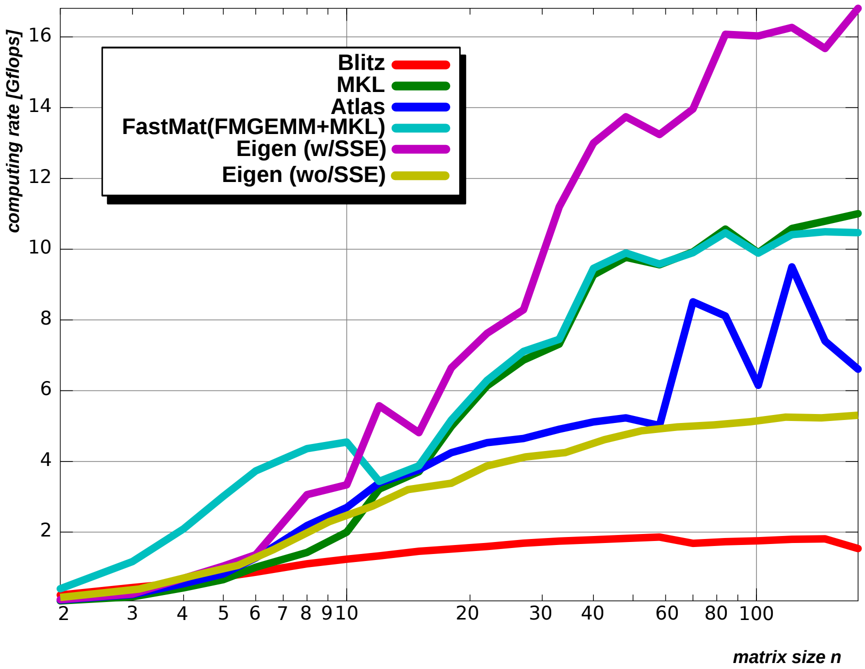

Figure 2: Efficiency comparatives for the matrix product $\mathbf{C} = \mathbf{AB}$ on an Intel Core–i7 950 @ 3.07GHz, for square matrices size $1 \leq n \leq 177$. Vertical axis is computing speed as given per equation (1).

The FMGEMM functions tend to saturate their performance near 3 Gflops for $n \leq 7$, so FastMat switches to the underlying BLAS implementation for matrices of size $n > n_{\max}$ with $n_{\max}$ a parameter configurable at compile time (the default is set with $n_{\max} = 10$).

In Figure (2) the performance of the combination FMGEMM+MKL is shown. As a reference, the curves for plain `dgemm()` from Atlas and MKL libraries, Blitz and Eigen are also included. Atlas is more performant than MKL for small $n \leq 12$, and the converse is true for $n > 12$. As FMGEMM is already used (since it is more performant) for small n, the better choice is FMGEMM and MKL. However, the choice for the underlying BLAS library can be made by the user during installation.

Regarding the Eigen test results, we notice that by default this library employs SSE instructions for the computations. As seen in figures, the Eigen library (using SSE instructions, noted as w/SSE in the figure) is almost 50% more efficient than MKL for large n, i.e. $n > 30$, however it is much less efficient than FastMat(FMGEMM) for small n, which is typically the case for FEM computations. For large matrices, the SSE-enabled code is 3x faster than the code not using SSE instructions (noted as wo/SSE in the figure).

It must be remarked that all these tests have been done on one core of the quad-core Intel i7–950 processor, i.e. no shared memory parallelism of the processor has been exploited. In fact, the user can configure to exploit parallelism at this level (i.e., inside `dgemm()`) by choosing the MKL library and setting the number of threads within the BLAS library functions via the `MKL_NUM_THREADS` environment variable, but then, the parallelism can not be exploited at the FastMat level, as discussed in §2.3.1. We note that normally, the parallelism at FastMat level is more efficient (since it is at a higher granularity). MKL can reach 40 Gflops for large matrices ($n > 100$) but this is not the most representative case in a FEM production code.

2.5 FastMat "Multi-product" operation

A common operation in FEM codes (see equation 14 and reference (10)) and many other applications is the product of several matrices or tensors. In addition, this kind of operation usually consume the largest part of the CPU time in a typical FEM computation of residuals and Jacobians. The number of operations (and consequently the CPU time) can be largely reduced by choosing the order in which the products are performed.

For instance consider the following operation

$$A_{ij} = B_{ik} C_{kl} D_{lj}, \tag{3}$$

(Einstein's convention on repeated indices is assumed). The same operation using matricial notation is

$$\mathbf{A} = \mathbf{BCD}, \tag{4}$$

where $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ are rectangular (rank 2) matrices of shape (m_1, m_4), (m_1, m_2), (m_2, m_3), (m_3, m_4), respectively. As the matrix product is associative the order in which the computations are performed can be chosen at will, so that it can be performed in the following two ways

$$\begin{aligned} \mathbf{A} &= (\mathbf{BC})\mathbf{D}, &\qquad &\text{(computation tree } \mathbf{CT1}\text{)}, \\ \mathbf{A} &= \mathbf{B}(\mathbf{CD}), &\qquad &\text{(computation tree } \mathbf{CT2}\text{)}. \end{aligned} \tag{5}$$

The order in which the computations are performed can be represented by a Complete Binary Tree. In this paper the order will be described using parentheses. The number of operations (op. count) for the different trees, and in consequence the CPU time, can be very different. The cost of performing the first product $\mathbf{BC}$ in the first row of (5) (and a product of two rectangular matrices in general) is

$$\text{op. count} = 2m_1m_2m_3, \tag{6}$$

which can be put as,

$$\begin{aligned} \text{op. count} &= 2\,(m_1m_3)(m_2), \\ &= 2\,(\text{prod. of dims for } \mathbf{B} \text{ and } \mathbf{C} \text{ free indices}) \\ &\times (\text{prod of dims for contracted indices}) \end{aligned} \tag{7}$$

or alternatively,

$$\begin{aligned} \text{op. count} &= 2\,\frac{(m_1m_2)(m_2m_3)}{m_2} \\ &= 2\,\frac{(\text{prod of } \mathbf{B} \text{ dims}) \times (\text{prod of } \mathbf{C} \text{ dims})}{(\text{prod of dims for contracted indices})} \end{aligned} \tag{8}$$

and for the second product is $2m_1m_3m_4$ so that the total cost for the first *computation tree* **CT1** is $2m_1m_3(m_2 + m_4)$. If the second computation tree **CT2** is used, then the number of operations is $2m_2m_4(m_1 + m_3)$. This numbers may be very different, for instance when $\mathbf{B}$ and $\mathbf{C}$ are square matrices and $\mathbf{D}$ is a vector, i.e. $m_1 = m_2 = m_3 = m > 1$, $m_4 = 1$. In this case the operation count is $2m^2(m+1) = O(m^3)$ for **CT1** and $4m^2 = O(m^2)$ for **CT2**, so that **CT2** is much more convenient.

2.5.1 Algorithms for the determination of the computation tree.

There is a *simple algorithm* that exploits this heuristic rule in a general case (2). If the *multi-product* is

$$\mathbf{R} = \mathbf{A}_1\mathbf{A}_2 \cdots \mathbf{A}_n \tag{9}$$

with $\mathbf{A}_k$ of shape (m_km_{k+1}), then the operation count c_k for each of the possible products $\mathbf{A}_k\mathbf{A}_{k+1}$, is computed, namely $c_k = 2m_km_{k+1}m_{k+2}$ for $k = 1$ to $n-1$. Let c_{k^*} be the minimum operation count, then the corresponding product $\mathbf{A}_{k^*}\mathbf{A}_{k^*+1}$ is performed and the pair $\mathbf{A}_q, \mathbf{A}_{q+1}$ is replaced by this product. Then, the list of matrices in the multi-product is shortened by one. The algorithm proceeds recursively until the number of matrices is reduced to only one. The cost of this algorithm is $O(n^2)$ (please note that this refers to the number of operations needed to determine the *computation tree*, not in actually computing the matrix product).

For a small number of matrices the optimal tree may be found by performing an exhaustive search over all possible orders. The cost is in this case $O(n!)$. In Figure (3) the computing times of the exhaustive optimal and the heuristic algorithms is shown for a number of matrices up to 8. Of course it is prohibitive for a large number of matrices, but it can be afforded for up to 6 or 7 matrices, which is by far the most common case. The situation is basically the same but more complex in the full tensorial case, as it is implemented in the FastMat library. First consider a product of two tensors like this

$$A_{ijk} = B_{klj}C_{il}, \tag{10}$$

where tensors A, B, C have shape (m_1, m_2, m_3), (m_3, m_4, m_2), (m_1, m_4) respectively. i, j, k are *free* indices while l is a *contracted* index. The cost of this product is (equations (7) and (8))

$$\text{op. count} = 2m_3m_2m_4m_1 \tag{11}$$

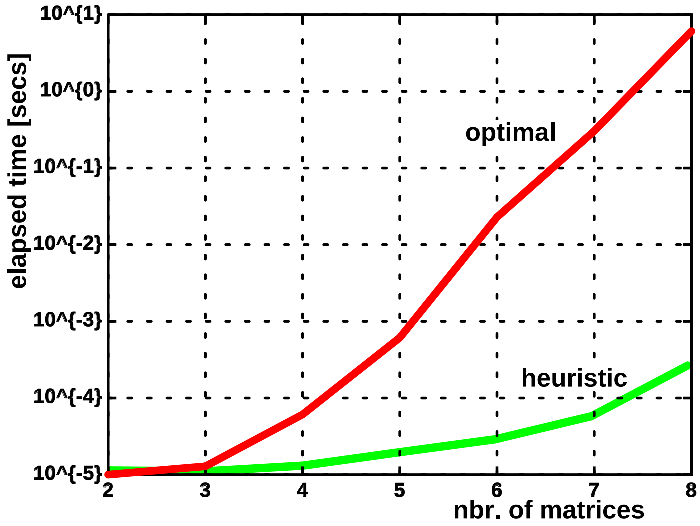

Figure 3: Cost of determination of the optimal order for computing the product of matrices with the heuristic and exhaustive (optimal) algorithms.

On one hand, the modification with respect to the case of rectangular (rank 2) matrices is that every matrix can be contracted with any other in the list. So that, the heuristic algorithm must check now all the pair of distinct matrices which is $n'(n'-1)/2$ where $1 \leq n' \leq n$ is the number of actual matrices. This must be added over n' so that the algorithm is $O(n^3)$. On the other hand, regarding the optimal order, it turns out to be that the complexity for its computation is

$$\prod_{n'=2}^{n} \frac{n'(n'-1)}{2} = O\left(\frac{(n!)^2}{2^n}\right). \tag{12}$$

In the FastMat library the strategy is to use the exhaustive approach for $n \leq n_{\text{ct,max}}$ and the heuristic one otherwise. The value of $n_{\text{ct,max}}$ is 5 by default but dynamically configurable by the user.

3 TEST EXAMPLES: COMPUTATION OF SUPG STABILIZATION TERM. ELEMENT RESIDUAL AND JACOBIAN

As an example, consider the computation of the stabilization term (see reference (10)) for general advective-diffusive systems, i.e.:

$$\frac{\partial \mathcal{H}(\mathbf{U})}{\partial t} + \frac{\partial \mathcal{F}_{c,j}(\mathbf{U})}{\partial x_j} = \frac{\partial \mathcal{F}_{d,j}(\mathbf{U}, \nabla \mathbf{U})}{\partial x_j} + \mathbf{G}. \tag{13}$$

Here $\mathbf{U} \in \mathbb{R}^n$ is the state vector, t is time, $\mathcal{F}_{c,j}, \mathcal{F}_{d,j}$ are the advective and diffusive fluxes respectively, $\mathbf{G}$ is a source term including, for instance, gravity acceleration or external heat sources, and x_j are the spatial coordinates.

The notation is standard, except perhaps for the *'generic enthalpy function'* $\mathcal{H}(\mathbf{U})$. The inclusion of the enthalpy function allows the inclusion of conservative equations in terms of non-conservative variables.

The discrete variational formulation of (13) with added SUPG stabilizing term (19; 20; 21) is written as follows: find find $\mathbf{U}^h \in \mathcal{S}^h$ such that, for every $\mathbf{W}^h \in \mathcal{V}^h$,

$$\begin{aligned}&\int_\Omega \mathbf{W}^h \cdot \left(\frac{\partial \mathcal{H}(\mathbf{U}^h)}{\partial t} + \frac{\partial \mathcal{F}_{c,j}}{\partial x_j} - \mathbf{G}\right) \mathrm{d}\Omega + \int_\Omega \frac{\partial \mathbf{W}^h}{\partial x_j} \mathcal{F}_{d,j} \, \mathrm{d}\Omega - \int_{\Gamma_h} \mathbf{W}^h \cdot \mathbf{H}^h \, \mathrm{d}\Gamma \\ &+ \underbrace{\sum_{e=1}^{n_{elem}} \int_\Omega \tau_e \, \mathbf{A}_k^T \frac{\partial \mathbf{W}^h}{\partial x_k} \cdot \left(\frac{\partial \mathcal{H}(\mathbf{U})}{\partial t} + \frac{\partial \mathcal{F}_{c,j}(\mathbf{U})}{\partial x_j} - \frac{\partial \mathcal{F}_{d,j}(\mathbf{U}, \nabla \mathbf{U})}{\partial x_j} - \mathbf{G}\right) \mathrm{d}\Omega}_{\text{Residual of SUPG term}} \\ &= 0,\end{aligned} \tag{14}$$

where

$$\begin{aligned}\mathcal{S}^h &= \left\{\mathbf{U}^h | \mathbf{U}^h \in [\mathbf{H}^{1h}(\Omega)]^m, \, \mathbf{U}^h\big|_{\Omega^e} \in [P^1(\Omega^e)]^m, \;\; \mathbf{U}^h = \mathbf{g} \text{ at } \Gamma_g\right\} \\ \mathcal{V}^h &= \left\{\mathbf{W}^h | \mathbf{W}^h \in [\mathbf{H}^{1h}(\Omega)]^m, \, \mathbf{U}^h\big|_{\Omega^e} \in [P^1(\Omega^e)]^m, \;\; \mathbf{U}^h = 0 \text{ at } \Gamma_g\right\}\end{aligned} \tag{15}$$

are the space of interpolation and weight function respectively, τ_e are stabilization parameters (a.k.a. '*intrinsic times*'), Γ_g is the Dirichlet part of the boundary where $\mathbf{U} = \mathbf{g}$ is imposed, and Γ_h is the Neumann part of the boundary where $\mathcal{F}_{d,j} n_j = \mathbf{H}$ is imposed.

According to (20) the stabilization parameter τ^{supg} is computed as

$$\tau^{\text{supg}} = \frac{h_{\text{supg}}}{2||a||} \tag{16}$$

with

$$h_{\text{supg}} = 2\left(\sum_{i=1}^{n_{en}} |\mathbf{s} \cdot \nabla \mathbf{W}_i^h|\right)^{-1}, \tag{17}$$

where s is a unit vector pointing in the streamline direction.

So that, the following product must be computed

$$R_{p\mu}^{\text{SUPG},e} = W_{p,k} A_{k\mu\nu} \tau_{\nu\alpha} R_\alpha^{\text{SUPG},gp}, \tag{18}$$

where

- $R^{\text{SUPG},e}$ (shape $(n_{\text{el}}, n_{\text{dof}})$, identifier `res`) is the SUPG residual contribution from the element e,
- $W_{p,k}$ (shape $(n_{\text{dim}}, n_{\text{el}})$, identifier `gN`) are the spatial gradients of the weight functions $\mathbf{W}_p$,
- $A_{k\mu\nu} = (\partial \mathcal{F}_{c,j\mu} / \partial U_\nu)$ (shape $(n_{\text{el}}, n_{\text{dof}}, n_{\text{dof}})$, identifier `A`) are the Jacobians of the advective fluxes $\mathcal{F}_{c,j\mu}$ with respect to the state variables U_ν,
- $\tau_{\nu\alpha}$ (shape $(n_{\text{dof}}, n_{\text{dof}})$, identifier `tau`) is the matrix of *intrinsic times*, and
- $R_\alpha^{\text{SUPG},gp}$ (shape (n_{dof}), identifier `R`) is the vector of residuals per field at the Gauss point.

This tensor products arise, for instance, in the context of the FEM-Galerkin SUPG stabilizing methods (see References (10; 20)). This multi-product is just an example of typical computations that are performed in a FEM based CFD code. This operation can be computed in a FastMat call like this

```
    res.prod(gN,A,tau,R,1,-1,-1,2,-2,-2,-3,-3);
```

where if the j-th integer argument is positive it represents the position of the index in the resulting matrix, otherwise if the j-th argument is -1 then a contraction operation is performed over all these indices (see appendix §5.1.3).

The `FastMat::prod()` method then implements several possibilities for the computing tree

- natural, left to right (L2R): The products are performed in the order the user has entered them, i.e. $(((A_1 A_2) A_3) A_4) \cdots$
- heuristic: Uses the heuristic algorithm described in section §2.5.1.
- optimal: An exhaustive brute-force approach is applied in order to determine the computation tree with the lowest operation count.

In table 1 the operation counts for these three strategies are reported. The first three columns show the relevant dimension parameters. n_{dim} may be 1,2,3 and n_{el} may be 2 (segments), 3 (triangles), 4 (quads in 2D, tetras in 3D) and 8 (hexahedra). The values explored for n_{dof} are

- $n_{\text{dof}} = 1$: scalar advection-diffusion,
- $n_{\text{dof}} = n_{\text{dim}} + 2$: compressible flow,
- $n_{\text{dof}} = 10$: advection-diffusion for 10 species.

3.1 Operation counts for the computation of element residuals.

The costs of the tensor operation defined in equation (18) (where the involved tensors are as described above) are evaluated in terms of the gains (%) (see Table 1). The gains are related to the cost of products performed in the natural order, so that

- **CT1=L2R** is: `((gN*R)*tau)*A`
- **CT2** is: `(gN*(tau*R))*A`
- **CT3** is: `gN*((A*tau)*R)`
- **CT4=R2L** is: `gN*(A*(tau*R))`

Note that **CT1** corresponds to computing the products in natural order from left to right (**L2R**) and **CT4** from right to left (**R2L**).

3.2 Operation counts for the computation of element Jacobians.

This product is similar as the one described above, but now the Jacobian of the residual term is computed so that the last tensor **is not a vector**, but rather a rank 3 tensor,

$$J^{\text{SUPG},e}_{p\mu q\nu} = W_{p,k} A_{k\mu\beta} \tau_{\beta\alpha} J^{\text{SUPG},gp}_{\alpha q\nu}, \tag{19}$$

```
J.prod(gN,A,tau,JR,1,-1,-1,2,-2,-2,-3,-3,3,4);
```

where

Table 1: Operation count for the stabilization term in the SUPG formulation of advection-diffusion of $n_{\rm dof}$ number of fields. Other relevant dimensions are the space dimension $n_{\rm dim}$ and the number of nodes per element $n_{\rm el}$.

$n_{\rm dim}$	$n_{\rm el}$	$n_{\rm dof}$	#ops(nat)	heur.	#ops	gain (%)	opt.	#ops	gain (%)
1	2	1	12	**CT3**	8	33.33%	**CT3**	8	33.33%
1	2	3	180	**CT1**	84	53.33%	**CT4**	48	73.33%
1	2	10	4800	**CT1**	840	82.50%	**CT4**	440	90.83%
2	3	1	24	**CT4**	18	25.00%	**CT4**	18	25.00%
2	3	4	672	**CT2**	272	59.52%	**CT4**	144	78.57%
2	3	10	7800	**CT1**	2520	67.69%	**CT4**	720	90.77%
2	4	1	32	**CT4**	22	31.25%	**CT4**	22	31.25%
2	4	4	896	**CT2**	352	60.71%	**CT4**	160	82.14%
2	4	10	10400	**CT1**	3660	67.69%	**CT4**	760	92.69%
3	4	1	40	**CT4**	32	20.00%	**CT4**	32	20.00%
3	4	5	1800	**CT2**	770	57.22%	**CT4**	320	82.22%
3	4	10	11200	**CT2**	2840	74.64%	**CT4**	1040	90.71%
3	8	1	80	**CT4**	56	30.00%	**CT4**	56	30.00%
3	8	5	3200	**CT4**	440	87.78%	**CT4**	440	87.78%
3	8	10	22400	**CT2**	5480	75.54%	**CT4**	1280	94.29%

Table 2: Operation count for the Jacobian of the SUPG term in the formulation of advection diffusion for $n_{\rm dof}$ number of fields. Other relevant dimensions are the space dimension $n_{\rm dim}$ and the number of nodes per element $n_{\rm el}$.

$n_{\rm dim}$	$n_{\rm el}$	$n_{\rm dof}$	#ops(nat)	heur.	#ops	gain (%)	opt.	#ops	gain (%)
1	2	1	16	**CT6**	14	12.50%	**CT6**	14	12.50%
1	2	3	360	**CT5**	360	0.00%	**CT7**	234	35.00%
1	2	10	12400	**CT5**	12400	0.00%	**CT7**	6800	45.16%
2	3	1	32	**CT6**	34	5.56%	**CT6**	34	5.56%
2	3	4	1728	**CT5**	1728	0.00%	**CT6**	1600	7.41%
2	3	10	25200	**CT5**	25200	0.00%	**CT7**	19600	22.22%
2	4	1	56	**CT6**	52	7.14%	**CT6**	26	7.14%
2	4	4	2816	**CT5**	2816	0.00%	**CT7**	2304	18.18%
2	4	10	41600	**CT5**	41600	0.00%	**CT7**	26400	36.54%
3	4	1	64	**CT6**	62	3.12%	**CT6**	62	3.12%
3	4	5	5600	**CT5**	5600	0.00%	**CT6**	5350	4.46%
3	4	10	42400	**CT5**	42400	0.00%	**CT7**	39600	6.60%
3	8	1	192	**CT6**	182	5.21%	**CT6**	182	5.21%
3	8	5	19200	**CT6**	17950	6.51%	**CT7**	16350	14.84%
3	8	10	148800	**CT5**	148800	0.00%	**CT7**	92400	37.90%

- $J^{\mathrm{SUPG},gp}_{\alpha q\beta}$ (shape $(n_{\mathrm{dof}}, n_{\mathrm{el}}, n_{\mathrm{dof}})$, identifier `JR`) is the Jacobian of residuals per field at the Gauss point.

Possible orders (or computation trees):

- **CT5=L2R**: `((gN*A)*tau)*JR`
- **CT6**: `(gN*(A*tau))*JR`
- **CT7**: `gN*((A*tau)*JR)`

Note that **CT5** corresponds to computing the products from left to right.

Discussion of the influence of computation tree. From the previous examples it is noticed that:

- In many cases the use of the **CT** determined with the heuristic or optimal orders yields a significant gain in operation count. The gain may be 90% or even higher in a realistic case like the computations for equations (18) and (19).
- In the presented cases, the heuristic algorithm yielded always a reduction in operation count, though in general this is not guaranteed. In some cases the heuristic approach yielded the optimal **CT**. In others it gave a gain, but far from the optimal one.
- It is very interesting that neither the heuristic nor optimal computation trees are the same for all combinations of parameters (n_{dim},n_{el},n_{dof}). For instance in the computation of the Jacobian the optimal order is `(gN*(A*tau))*JR` in some cases and `gN*((A*tau)*JR)` in others (designated as **CT6** and **CT7** in the tables). This means that it would be impossible for the user to choose an optimal order for all cases. This must be computed automatically at run-time as proposed here in FastMat library.
- In some cases the optimal or heuristically determined orders involve contractions that are not in the natural order, for instance the **CT** `(gN*(tau*R))*A` (designated as **CT2**) is obtained with the heuristic algorithm for the computation of the element residual for some set of parameters. Note that the second scheduled contraction involves the `gN` and `tau*R` tensors, even if they do not share any contracted indices.

Note that, in order to perform the computation of an efficient **CT** for the multi-product (with either the heuristic or optimal algorithms) it is needed that the library implements the operation in functional form as in the FastMat library. Operator overloading is not friendly with the implementation of an algorithm like this, because there is no way to capture the whole set of matrices and the contraction indices. The computation of the optimal or heuristic order is done only once and stored in the cache object. In fact this is a very good example of the utility of using caches.

Using more elaborated estimations of computing time In the present work it is assumed that the computing time is directly proportional to the number of operations. This may not be true, but note that the computation tree could be determined with a more direct approach. For instance, by benchmarking the products and then determining the **CT** that results in the lowest computing time, not in operation count.

3.3 Comparison of FastMat2 with Blitz for a FEM multiprod

A comparison of efficiency is shown in Figure (4). The multiproduct described by equation (19) was computed using FastMat and Blitz libraries for the same combinations of tensor sizes $(n_{\rm dim}, n_{\rm el}, n_{\rm dof})$ in the examples shown in sections §3.1 and §3.2). In the figure, the computing rate (in multiproducts per second units, i.e. the number of times that the whole computation (19) can be performed per second) is reported for the Blitz and FastMat libraries. Blitz doesn't compute an optimal computation tree; it's up to the user to determine the order in which the contractions are performed so we compute Blitz times by performing the multiproduct en L2R and R2L order. In particular for the operation (19) the L2R ordering coincides withe the heuristic one for near one half of the possible size combinations (see Table 2) In the FastMat case it must be remembered that the optimal computation tree is computed only once at the start of the loop and stored in a FastMat cache. As it can be seen from the plots, FastMat is always faster than Blitz even if using a non optimal computation tree. FastMat with the optimal computation tree is at least 1.9x times faster than Blitz (using the best of L2R and R2L). Also, note that FastMat with the optimal computation tree is always faster than Fastmat with both L2R and R2L computation trees, being 2.3x faster in the most extreme case. This shows the advantage of using the optimal computation tree.

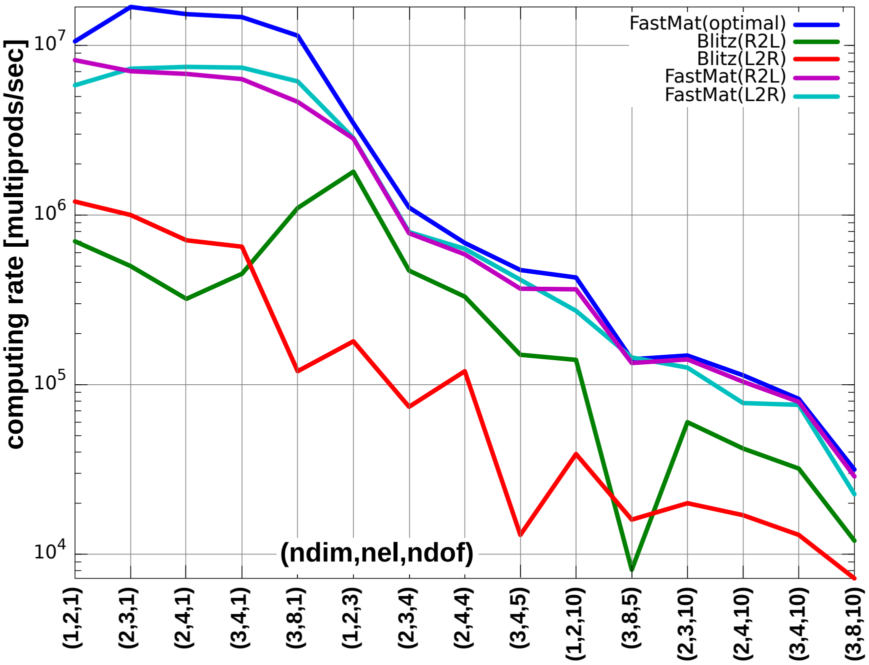

Figure 4: Efficiency comparatives for the multiproduct of equation (19) on an Intel Core i7–950 @ 3.07GHz, for different combinations of matrix dimensions.

3.4 Results for SUPG matrix using multiproduct in a shared memory architecture

In this section the Jacobian matrix of equation (19) was computed in a shared memory environment. The equipment used to perform this tests was a six-core Intel(R) Xeon(R) CPU W3690 @ 3.47GHz. The speedup obtained for the Jacobian matrix computation (for tensor sizes $(n_{\rm dim} = 3, n_{\rm el} = 8, n_{\rm dof} = 5)$) is illustrated in Figure (5). As shown in the figure, the Jacobian computation stage using the FastMat multiprod operation scales linearly with the number of cores/threads. The computing rates range from 1.55 Gflops using one core to 8.99

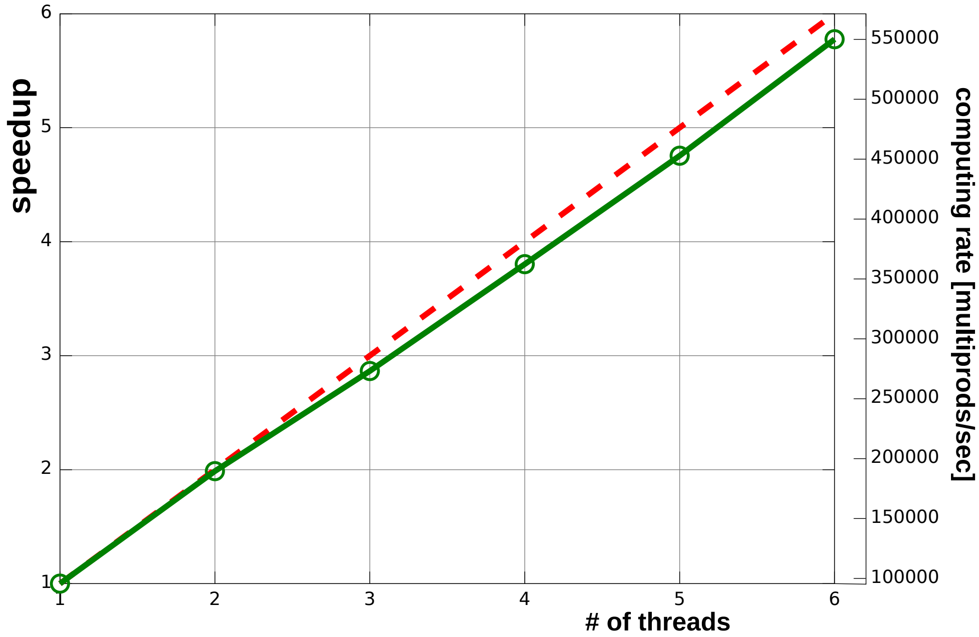

Figure 5: Speedup for the Jacobian matrix computation.

Gflops in 6 cores (95,000 to 550,000 multiprods/sec respectively) .

4 CONCLUSIONS

In this paper, we introduced and evaluated an efficient, thread-safe library for tensor computations. Having in mind that for large scale meshes all FEM computation inside the loop over elements must be evaluated millions of times, the process of caching all FEM operations at the element level (provides by FastMat library) is a key-point in the performance for the computing/assembling stages. The most important features of the FastMat tensor library can be summarized as follows:

- Repeated multi-index operations at the element level are stored in cache objects. Hence, element routines are very fast, in the sense that almost all the CPU time is spent in performing multiplications and additions, and negligible CPU time is spent in auxiliary operations. This feature is the key-point when dealing with large/fine FEM meshes.

- For the very common multi-product tensor operation, the order in which the successive tensor contractions are computed is always performed at the minimum operation count cost depending on the number of tensor/matrices in the FEM terms (exhaustive or heuristic approaches).

- The FastMat library implements index contractions in a general way. The number of contracted tensors and the range of their indices can be variable. Also, a complete set of common tensor operations are available in the library.

- The FastMat library is “thread-safe”, i.e., element residuals and Jacobians can be computed in the context of multi-threaded FEM/FVM codes.

ACKNOWLEDGMENTS

This work has received financial support from *European Research Council (ERC)* (Spain, Advanced Grant, Real Time Computational Mechanics Techniques for Multi-Fluid Problems, Reference: ERC-2009-AdG, Dir: Dr. Sergio Idelsohn), *Consejo Nacional de Investigaciones Científicas y Técnicas* (CONICET, Argentina, grant PIP 5271/05), *Universidad Nacional del Litoral* (UNL, Argentina, grants CAI+D 2009 65/334), *Agencia Nacional de Promoción Científica y Tecnológica* (ANPCyT, Argentina, grants PICT 01141/2007, PICT 2008-0270 "Jóvenes Investigadores", PICT-1506/2006), *Secretaría de Ciencia y Tecnología de la Universidad Tecnológica Nacional, Facultad Regional Resistencia, Chaco* (UTN FRRe, Argentina, grant PID 2012 25/L057) and *Universidad Tecnológica Nacional, Facultad Regional San Nicolás* (Buenos Aires).

5 APPENDIX

5.1 Synopsis of FastMat operations

5.1.1 One-to-one operations.

The operations are from one element of `A` to the corresponding element in `*this`.
The one-to-one operations implemented so far are

- `FastMat& set(const FastMat &A)`: Copy matrix.
- `FastMat& add(const FastMat &A)`: Add matrix.
- `FastMat& minus(const FastMat &A)`: Substract a matrix.
- `FastMat& mult(const FastMat &A)`: Multiply (element by element) (like Matlab `.*`).
- `FastMat& div(const FastMat &A)`: Divide matrix (element by element, like Matlab `./`).
- `FastMat& axpy(const FastMat &A, const double alpha)`: Axpy operation (element by element): `(*this) += alpha * A`

5.1.2 In-place operations.

These operations perform an action on all the elements of a matrix.

- `FastMat& set(const double val=0.)`: Sets all the element of a matrix to a constant value.
- `FastMat& scale(const double val)`: Scale by a constant value.
- `FastMat& add(const double val)`: Adds constant val.
- `FastMat& fun(double (*)(double) *f)`: Apply a function to all elements.
- `FastMat2& fun(func *function, void *user_args)`: Apply a function with optional arguments to all elements.

5.1.3 Generic "sum" operations (sum over indices).

These methods perform some associative reduction operation an all the indices of a given dimension resulting in a matrix which has a lower rank. It's a generalization of the `sum/max/min` operations in Matlab (22) that returns the specified operation per columns, resulting in a row vector result (one element per column). Here you specify a number of integer arguments, in such a way that

- if the j-th integer argument is positive it represents the position of the index in the resulting matrix, otherwise
- if the j-th argument is -1 then the specified operation (sum/max/min etc...) is performed over all this index.

For instance, if a `FastMat A(4,2,2,3,3)` is declared then
`B.sum(A,-1,2,1,-1)` means

$$B_{ij} = \sum_{k=1..2,l=1..3} A_{kjil}, \text{ for } i = 1..3,\ j = 1..2 \tag{20}$$

These operation can be extended to any binary associative operation. So far the following operations are implemented

- `FastMat& sum(const FastMat &A, const int m=0,..)`: Sum over all selected indices.
- `FastMat& sum_square(const FastMat &A, const int m=0,..)`: Sum of squares over all selected indices.
- `FastMat& sum_abs(const FastMat &A,+ const int m=0,..)`: Sum of absolute values all selected indices.
- `FastMat& min(const FastMat &A, const int m=0,..)`: Minimum over all selected indices.
- `FastMat& max(const FastMat &A, const int m=0,..)`: Maximum over all selected indices.
- `FastMat& min_abs(const FastMat &A, const int m=0,..)`: Min of absolute value over all selected indices.
- `FastMat& max_abs(const FastMat &A, const int m=0,..)`: Max of absolute value over all selected indices.

5.1.4 Sum operations over all indices.

When the sum is over all indices the resulting matrix has zero dimensions, so that it is a scalar. You can get this scalar by creating an auxiliary matrix (with zero dimensions) casting with operator `double()` as in

```
FastMat A(2,3,3),Z;
... // assign elements to A
double a = double(Z.sum(A,-1,-1));
```

or using the `get()` function

```
double a = Z.sum(A,-1,-1).get();
```

without arguments, which returns a double. In addition there is for each of the previous mentioned "generic sum" function a companion function that sums over all indices. The name of this function is obtained by appending `_all` to the generic function

```
double a = A.sum_square_all();
```

The list of these functions is

- `double sum_all() const`: Sum over all indices.
- `double sum_square_all() const`: Sum of squares over all indices.
- `double sum_abs_all() const`: Sum of absolute values over all indices.
- `double min_all() const`: Minimum over all indices.
- `double max_all() const`: Maximum over all indices.
- `double min_abs_all() const`: Minimum absolute value over all indices.
- `double max_abs_all() const`: Maximum absolute value over all indices.

5.1.5 Export/Import operations.

These routines allow to convert matrices from or to arrays of doubles

- `FastMat& set(const double *a)`: Copy from array of doubles.
- `FastMat& export(double *a)`: Export to a double vector.
- `const FastMat2& export(double *a)`: Constant export to a double vector.

REFERENCES

[1] The GNU C++ library: Standard Template Library Programmer's guide (2011). URL **http://gcc.gnu.org**

[2] Aho, A., Ullman, J., Hopcroft, J.: Data structures and algorithms. Addison Wesley (1983)

[3] ATLAS: Automatically tuned linear algebra software (2011). URL **http://acts.nersc.gov/atlas**

[4] Balay, S., Gropp, W., McInnes, L.C., Smith, B.: Efficient management of parallelism in object oriented numerical software libraries. Modern Software Tools in Scientific Computing. pp. 163–202 (1997)

[5] Behara, S., Mittal, S.: Parallel finite element computation of incompressible flows. Parallel Computing. **35**, 195–212 (2009)

[6] Benoit, J., Guennebaud, G.: Eigen 3 (2011). URL `http://eigen.tuxfamily.org`

[7] BLAS: Basic Linear Algebra Subprograms (2010). URL `http://www.netlib.org/blas`

[8] Dalcin, L.: PETSc for Python. (2010). URL `http://petsc4py.googlecode.com`

[9] Dalcin, L., Paz, R., Kler, P., Cosimo, A.: Parallel distributed computing using python. Advances in Water Resources **34**(9), 1124–1139 (2011)

[10] Donea, J., Huerta, A.: Finite element methods for flow problems. Wiley & Sons (2003)

[11] Kamenik, O.: TL – Multidimensional Tensor Library (2011). URL `http://www.dynare.org`

[12] Komatitsch, D., Erlebacher, G., Göddeke, D., Michéa, D.: High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. Journal of Computational Physics **229**, 7692–7714 (2010)

[13] Landry, W.: FTensor – a high performance tensor library (2004). URL `http://www.gps.caltech.edu/~walter/FTensor`

[14] LAPACK: Linear Algebra PACKage (2010). URL `http://www.netlib.org/lapack`

[15] Paz, R., Nigro, N., Storti, M.: On the efficiency and quality of numerical solutions in CFD problems using the interface strip preconditioner for domain decomposition methods. International Journal for Numerical Methods in Fluids. **51**(1), 89–118 (2006)

[16] Shroeder, W., Martin, K., Lorensen, B.: The visualization toolkit. Prentice Hall, Englewood Cliffs, NJ **810**, 811 (1996). URL `http://www.vtk.org`

[17] Sonzogni, V., Yommi, A., Nigro, N., Storti, M.: A parallel finite element program on a Beowulf cluster. Advances in Engineering Software **33**(7-10), 427–443 (2002)

[18] Storti, M., Nigro, N., Paz, R., Dalcin, L.: PETSc-FEM – A General Purpose, Parallel, Multi-Physics FEM Program (2010). URL `http://www.cimec.org.ar/petscfem`

[19] Tezduyar, T., Aliabadi, S., Behr, M., Johnson, A., Kalro, V., Litke, M.: Flow simulation and high performance computing. Computational Mechanics **18**(6), 397–412 (1996)

[20] Tezduyar, T., Osawa, Y.: Finite element stabilization parameters computed from element matrices and vectors. Computer Methods in Applied Mechanics and Engineering. **190**, 411–430 (2000)

[21] Tezduyar, T., Senga, M., Vicker, D.: Computation of inviscid supersonic flows around cylinders and spheres with the supg formulation and YZ β shock-capturing. Computational Mechanics **38**(4–5), 469–481 (2006)

[22] The MathWorks Inc.: MATLAB (2011). URL `http://www.mathworks.com/matlab`

[23] Veldhuizen, T.: Blitz++ – Object-Oriented Scientific Computing (2010). URL `http://www.oonumerics.org/blitz`

[24] Whaley, R., Petitet, A., Dongarra, J.: Practical experience in the numerical dangers of heterogeneous computing. Parallel Computing. **27**((1-2)), 3–35 (2001)