Programming GPU using TNL

Tomáš Oberhuber Jakub Klinkovský Radek Fučík
Vítězslav Žabka Aleš Wodecki

Department of Mathematics,
Faculty of Nuclear Sciences and Physical Engineering,
Czech Technical University in Prague

HPCSE 2019

Why GPU?

	Nvidia V100	Intel Xeon E5-4660
Cores	5120 @ 1.3GHz	16 @ 3.0GHz
Peak perf.	15.7/7.8 TFlops	0.4 / 0.2 TFlops
Max. RAM	32 GB	1.5 TB
Memory bw.	900 GB/s	68 GB/s
TDP	300 W	120 W

$\approx 8,000$ \$

Difficulties in programming GPUs?

Unfortunately,

- the programmer must have good knowledge of the hardware
- porting a code to GPUs often means rewriting the code from scratch
- lack of support in older numerical libraries

Numerical libraries which makes GPUs easily accessible are being developed.

Template Numerical Library

TNL = Template Numerical Library

- is written in C++ and profits from meta-programming
- provides unified interface to multi-core CPUs and GPUs (via CUDA)
- wants to be user friendly
- www.tnl-project.org
- $\approx$ 300k lines of templated code
- MIT license

Arrays

Arrays are basic structures for memory management

- `TNL::Array< ElementType, DeviceType, IndexType >`
- `DeviceType` says where the array resides
  - `TNL::Devices::Host` for CPU
  - `TNL::Devices::Cuda` for GPU
- memory allocation, I/O operations, elements manipulation ...

```
Array< float, Devices::Cuda, int > a( 100 );
a.evaluate( [] __cuda_callable__ ( int i ) { return i%5; } );
```

Vectors

Vectors add algebraic operations to arrays:

- `TNL::Vector< RealType, DeviceType, IndexType >`
- addition, multiplication, scalar product, l_p norms ...

Vector and Array View

- arrays and vectors supports data sharing
- both are relatively complex structures
- TNL uses also lightweight counterparts `ArrayView`, `VectorView`
- both can be passed efficiently on GPU for example
- neither perform dynamic memory allocation/deallocation or deep copies

```
Vector< float, Devices::Cuda, int > v( 100 );
VectorView< float, Devices::Cuda, int > view( v );
```

Parallel reduction

Parallel reduction is operation taking all array/vector elements as input and returns one value as output:

- array comparison
- scalar product
- l_p norm
- minimal/maximal value
- sum of all elements

```
float sum( 0.0 )
for( int i = 0; i < size; i++ )
    sum += a[ i ];
```

Parallel reduction on GPU = 150 lines of code

```
template <unsigned int blockSize>
__global__ void cudaBlockReduction6(int size, int* dInput, int* dOutput, )
{
   extern __shared__ volatile int sdata[];
   unsigned int tid = threadIdx.x;
   unsigned int gid = blockIdx.x*blockSize*2 + threadIdx.x;
   unsigned int gridSize = blockSize*2*gridDim.x;
   sdata[tid] = 0;
   while (gid < size)
   {
      sdata[tid] += dInput[gid] + dInput[gid+blockSize];
      gid += gridSize;
   }
   __syncthreads();

   if (blockSize == 1024) {
      if( tid<512 ) { sdata[ tid ] += sdata[ tid+512 ]; } __syncthreads(); }
   if (blockSize >= 512) {
      if( tid<256 ) { sdata[ tid ] += sdata[ tid+256 ]; } __syncthreads(); }
   if (blockSize >= 256) {
      if( tid<128 ) { sdata[ tid ] += sdata[ tid+128 ]; } __syncthreads(); }
   if (blockSize >= 128) {
      if( tid< 64 ) { sdata[ tid ] += sdata[ tid+ 64 ]; } __syncthreads(); }
   if (tid < 32) {
      if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
      if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
      if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
      if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
      if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
      if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
   }

   if (tid == 0) dOutput[blockIdx.x] = sdata[0];
}

bool cudaGridReduction6( const int size,
                         const int* deviceData,
                         const int blockSize,
                         const int desiredGridSize,
                         int& result )
{
   const int elementsInBlock = 2 * blockSize;
   int gridSize = size / elementsInBlock + ( size % elementsInBlock != 0 );
   gridSize = gridSize < desiredGridSize ? gridSize : desiredGridSize;
   int auxArraySize = gridSize * sizeof( int );
   int *deviceAux1, *deviceAux2;
   if( cudaMalloc( ( void** ) &deviceAux1, auxArraySize ) != cudaSuccess ||
       cudaMalloc( ( void** ) &deviceAux2, auxArraySize ) != cudaSuccess )
   {
      cerr << "Not␣enough␣memory␣on␣device." << endl;
      return false;
   }

   int currentSize = size;
   const int* reductionInput = deviceData;
   while( currentSize > 1 )
   {
      dim3 cudaBlockSize( 0 ), cudaGridSize( 0 );
      cudaBlockSize. x = blockSize;
      gridSize = currentSize / elementsInBlock +
                 ( currentSize % elementsInBlock != 0 );
      cudaGridSize. x = gridSize < desiredGridSize ? gridSize : desiredGridSize;

      size_t sharedMemory = blockSize * sizeof( int );
      switch( blockSize )
      {
         case 1024:
            cudaBlockReduction6< 1024 >
               <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
               ( currentSize, reductionInput, deviceAux1 );
            break;
         case 512:
            cudaBlockReduction6< 512 >
               <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
               ( currentSize, reductionInput, deviceAux1 );
            break;
         case 256:
            cudaBlockReduction6< 256 >
               <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
               ( currentSize, reductionInput, deviceAux1 );
            break;
         case 128:
            cudaBlockReduction6< 128 >
               <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
               ( currentSize, reductionInput, deviceAux1 );
            break;
         case 64:
            cudaBlockReduction6< 64 >
               <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
               ( currentSize, reductionInput, deviceAux1 );
            break;
         case 32:
            cudaBlockReduction6< 32 >
               <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
               ( currentSize, reductionInput, deviceAux1 );
            break;
         case 16:
            cudaBlockReduction6< 16 >
               <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
               ( currentSize, reductionInput, deviceAux1 );
            break;
         case 8:
            cudaBlockReduction6< 8 >
               <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
               ( currentSize, reductionInput, deviceAux1 );
            break;
         case 4:
            cudaBlockReduction6< 4 >
               <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
               ( currentSize, reductionInput, deviceAux1 );
            break;
         case 2:
            cudaBlockReduction6< 2 >
               <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
               ( currentSize, reductionInput, deviceAux1 );
            break;
         case 1:
            cudaBlockReduction6< 1 >
               <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
               ( currentSize, reductionInput, deviceAux1 );
            break;

      }
      cudaError error = cudaGetLastError();
      if( error != cudaSuccess )
      {
         cerr << "The␣CUDA␣kernel␣ended␣with␣error␣code␣" << error << "."
              << endl;
         return false;
      }
      currentSize = cudaGridSize. x;

      reductionInput = deviceAux1;
      int* tmp = deviceAux1;
      deviceAux1 = deviceAux2;
      deviceAux2 = tmp;
   }
   if( cudaMemcpy( &result,
                   reductionInput,
                   sizeof( int ),
                   cudaMemcpyDeviceToHost ) != cudaSuccess )
   {
      cerr << "An␣error␣occured␣while␣transfering␣data␣from␣device␣to␣host."
           << endl;
      return false;
   }
   cudaFree( deviceAux1 );
   cudaFree( deviceAux2 );
   return true;
}
```

Parallel reduction in TNL

Take a look at scalar product:

```
float result( 0.0 );
for( int i = 0; i < size; i++ )
    result += a[ i ] * b[ i ];
```

Let us rewrite it using C++ lambda functions as:

```
float a[ size ], b[size ];

...

auto fetch = [=] (int i)->float { return a[i]*b[i];};
auto reduce = [] (float& x, const float& y) { x += y;};

float result( 0.0 );
for( int i = 0; i < size; i++ )
    reduce( result, fetch( i ) );
```

Parallel reduction in TNL

Another example - l_p-norm:

```
const float p = 2.0;
float a[ size ];

auto fetch = [=] (int i)->float { return pow( fabs( a[i] ), p );};
auto reduce = [] (float& x, const float& y) { x += y;};

float result( 0.0 );
for( int i = 0; i < size; i++ )
    reduce( result, fetch( i ) );
```

Parallel reduction in TNL

Another example - arrays comparison:

```
bool zero = true;
const float p = 2.0;
float a[size], b[size];
...
auto fetch = [=] (int i)->bool { return ( a[i] == b[i] ); };
auto reduce = [] (float& x, const float& y) { x = x && y; };

float result( zero );
for( int i = 0; i < size; i++ )
    reduce( result, fetch( i ) );
```

Parallel reduction in TNL

To perform the same on GPU in TNL just add `__cuda_callable__` to lambdas...

```
auto fetch = [=] __cuda_callable__ (int i)->bool { return ( a[i] == b[i] ); };
auto reduce = [] __cuda_callable__ (float& x, const float& y) { x = x && y; };
```

... and for certain reasons, deliver `volatile` version of `reduce`:

```
auto volatileReduce = [] __cuda_callable__ (volatile float& x,
                                             volatile const float& y)
        { x = x && y; };
```

This could be avoided when CUDA compiler supports C++17 better. Now call

```
Reduction< Devices::Cuda >::reduce( size, reduce, volatileReduce, fetch, zero );
```

Expression Templates in TNL

Expression templates are efficient tool for (vector) algebraic operations.
Expression

$$\vec{x} = \vec{a} + 2\vec{b} + 3\vec{c}$$

can be evaluated in C as follows:

```
for( int i = 0; i < size; i++ )
        x[ i ]= a[ i ] + 2 * b[ i ] + 3 * c[ i ];
```

It is:

- efficient
- relatively simple
- works only on CPU - sequentially

Expression Templates in TNL

We can use operators overloading in C++:

```
x = a + 2 * b + 3 *c;
```

- it is very simple and easy to read
- can be performed in parallel on multicore CPUs or GPUs
- it is inefficient

Expression Templates in TNL

We can use BLAS/cuBLAS:

```
cublasHandle_t handle;
cublasSaxpy( handle, size, 1.0, y, 1, x, 1 );
cublasSaxpy( handle, size, 2.0, b, 1, x, 1 );
cublasSaxpy( handle, size, 3.0, c, 1, x, 1 );
```

- it is pretty hard to read
- works only for single precision
- more efficient than C++ version but still less efficient than C version

Expression Templates in TNL

Expression templates take simple formula...

```
x = a + 2 * b + 3 *c;
```

... parse it and evaluate the same way as C.

In TNL, `a`, `b` and `c` are `VectorViews`

```
VectorView< Real, Device, Index > a, b, c;
```

- it is simple and easy to read
- works for any type `Real` (`float`/`double`) and any Device (CPU/GPU)
- it is very efficient

Expression Templates & Parallel Reduction in TNL

Example:

```
using Vector = Vector< float, Devices::Cuda, int >;
using View = VectorView< float, Devices::Cuda, int >;
Vector av( 100 ), bv( 100 ), cv( 100 ), dv( 100 );
View a( av ), b( bv ), c( cv ), d( dv );
...
float scalarProduct = ( a, b + 3 * c );
d = a + b * c + sin( d );
a = min( b, c );
float min_a = min( a );
float total_min = min( min( a, b ) );
```

Performance comparison

Performance was tested on:

- GPU Nvidia P100
  - 16 GB HBM2 @ 732 GB/s
  - 3584 CUDA cores, 4.7 TFlops in double precision
- CPU
  - AMD Ryzen 5 2600, 8MB L3 cache

Expression Templates in TNL

Scalar product: `r = ( x, y )`.

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	20.8	2.7	0.1	49.3	69.9	1.41
200k	18.5	12.2	0.6	90.1	108.6	1.20
400k	18.4	13.3	0.7	142.2	159.1	1.11
800k	11.9	13.2	1.1	207.4	233.4	1.12
1.6M	13.6	15.6	1.1	313.6	333.3	1.06
3.2M	14.9	17.9	1.2	381.0	403.7	1.05
6.4M	16.7	17.0	1.0	417.1	431.8	1.03

Expression Templates in TNL

Vector addition: `x += a`.

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	46.0	11.6	0.2	152.2	174.8	1.14
200k	42.3	7.0	0.1	196.6	216.1	1.09
400k	16.9	29.3	1.7	277.6	294.4	1.06
800k	16.8	23.9	1.4	326.2	333.6	1.02
1.6M	17.3	25.1	1.4	362.5	374.2	1.03
3.2M	17.5	25.3	1.4	422.4	436.8	1.03
6.4M	17.4	25.7	1.4	456.6	469.8	1.02

Expression Templates in TNL

Vector addition: `x += a + b`.

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	30.3	29.3	0.9	188.3	190.9	1.01
200k	30.5	31.8	1.0	218.0	230.7	1.05
400k	13.7	32.8	2.3	243.1	305.9	1.25
800k	11.6	23.0	1.9	263.8	353.0	1.33
1.6M	11.7	24.4	2.0	285.9	389.4	1.36
3.2M	11.7	24.8	2.1	312.9	442.8	1.41
6.4M	11.7	25.7	2.1	327.3	471.9	1.44

Expression Templates in TNL

Vector addition: `x += a + b + c`.

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	25.4	7.7	0.30	194.7	236.5	1.21
200k	23.7	16.1	0.67	228.3	277.6	1.21
400k	13.0	31.0	2.38	218.3	330.9	1.51
800k	10.0	24.5	2.45	233.3	370.6	1.58
1.6M	9.9	23.6	2.38	249.6	403.4	1.61
3.2M	9.8	25.2	2.57	266.6	444.8	1.66
6.4M	9.8	25.9	2.64	276.6	471.3	1.70

Matrix formats

TNL supports the following matrix formats (on both CPU and GPU):

- dense matrix format
- tridiagonal and multidiagonal matrix format
- Ellpack format
- CSR format
- SlicedEllpack format
- ChunkedEllpack format

Oberhuber T., Suzuki A., Vacata J., *New Row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA*, Acta Technica, 2011, vol. 56, no. 4, pp. 447-466.

Heller M., Oberhuber T., *Improved Row-grouped CSR Format for Storing of Sparse Matrices on GPU*, Proceedings of Algoritmy 2012, 2012, Handlovičová A., Minarechová Z. and Ševčovič D. (ed.), pages 282-290.

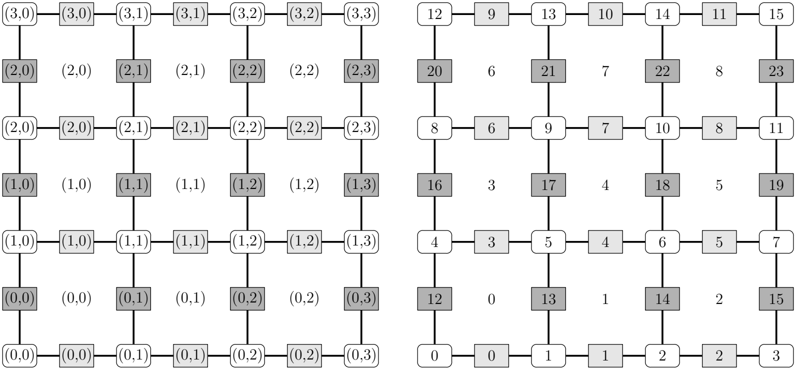
Numerical meshes

TNL supports

- structured orthogonal **grids** – 1D, 2D, 3D
    - mesh entities are generated on the fly
- unstructured **meshes** – nD
    - mesh entities are stored in memory

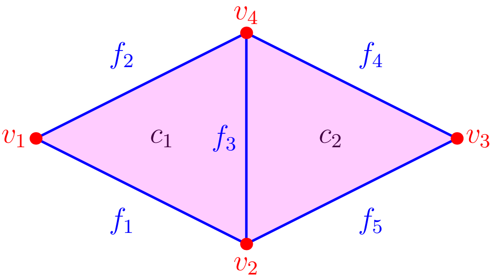
Structured grids

`TNL::Meshes::Grid< Dimensions,Real,Device,Index >`



Grid provides mapping between coordinates and global indexes.

Unstructured meshes



$I_{0,1} =$

	f_1	f_2	f_3	f_4	f_5
v_1	1	1			
v_2	1		1		1
v_3				1	1
v_4		1	1	1	

$I_{0,2} =$

	c_1	c_2
v_1	1	
v_2	1	1
v_3		1
v_4	1	1

Unstructured meshes

`TNL::Meshes::Mesh< MeshConfig, Device >`

- can have arbitrary dimension
- `MeshConfig` controls what mesh entities and links between them are stored
- it is done in the compile-time thanks to C++ templates

Based on `MeshConfig`, the mesh is fine-tuned for specific numerical method in compile-time.

Solvers

ODEs solvers

- Euler, Runge-Kutta-Merson

Linear systems solvers

- Krylov subspace methods (CG, BiCGSTab, GMRES, TFQMR)
- highly parallel CWYGMRES method

Klinkovský J., Oberhuber T., Fučík R., *Performance evaluation of distributed MGSR- and CWY- based GMRES variants of MHFEM*, submitted to International Journal of High Performance Computing Applications.

Oberhuber T., Suzuki A., Žabka V., *The CUDA implementation of the method of lines for the curvature dependent flows*, Kybernetika, 2011, vol. 47, num. 2, pp. 251–272.

Oberhuber T., Suzuki A., Vacata J., Žabka V., *Image segmentation using CUDA implementations of the Runge-Kutta-Merson and GMRES methods*, Journal of Math-for-Industry, 2011, vol. 3, pp. 73–79.

Multiphase flow in porous media

We consider the following system of n partial differential equations in a general coefficient form

$$\sum_{j=1}^{n} N_{i,j} \frac{\partial Z_j}{\partial t} + \sum_{j=1}^{n} \mathbf{u}_{i,j} \cdot \nabla Z_j + \nabla \cdot \left[m_i \left(-\sum_{j=1}^{n} D_{i,j} \nabla Z_j + \mathbf{w}_i \right) + \sum_{j=1}^{n} Z_j \mathbf{a}_{i,j} \right] + \sum_{j=1}^{n} r_{i,j} Z_j = f_i$$

for $i = 1, ..., n$, where the unknown vector function $\vec{Z} = (Z_1, ..., Z_n)^T$ depends on position vector $\vec{x} \in \Omega \subset \mathbb{R}^d$ and time $t \in [0, T]$, $d = 1, 2, 3$.

Multiphase flow in porous media

Initial condition:

$$Z_j(\vec{x},0) = Z_j^{ini}(\vec{x}), \quad \forall \vec{x} \in \Omega,\ j = 1, \dots, n,$$

Boundary conditions:

$$\begin{aligned} Z_j(\vec{x},t) &= Z_j^{\mathcal{D}}(\vec{x},t), \quad \forall \vec{x} \in \Gamma_j^{\mathcal{D}} \subset \partial\Omega,\ \ j = 1, ..., n, \\ \vec{v}_i(\vec{x},t) \cdot \vec{n}_{\partial\Omega}(\vec{x}) &= v_i^{\mathcal{N}}(\vec{x},t), \quad \forall \vec{x} \in \Gamma_i^{\mathcal{N}} \subset \partial\Omega,\ \ i = 1, ..., n, \end{aligned}$$

where $\vec{v}_i$ denotes the conservative velocity term

$$\vec{v}_i = -\sum_{j=1}^{n} \mathbf{D}_{i,j} \nabla Z_j + \mathbf{w}_i.$$

Numerical method

- Based on the mixed-hybrid finite element method (MHFEM)
  - one global large sparse linear system for traces of $(Z_1, , \ldots, Z_n)$ (on faces) per time step
- Semi-implicit time discretization
- General spatial dimension (1D, 2D, 3D)
- Structured and unstructured meshes

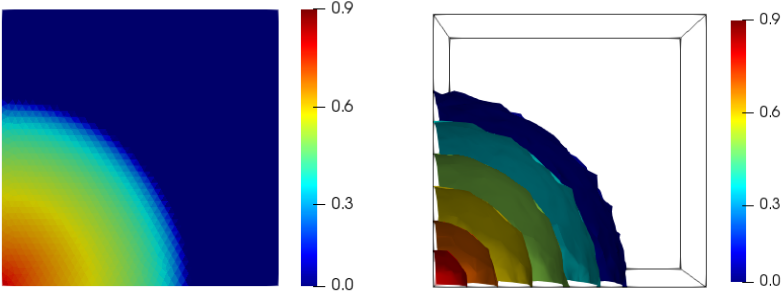
R. Fučík, J.Klinkovský, T. Oberhuber, J. Mikyška, *Multidimensional Mixed–Hybrid Finite Element Method for Compositional Two–Phase Flow in Heterogeneous Porous Media and its Parallel Implementation on GPU*, submitted to Computer Physics Communications.

McWhorter–Sunada problem

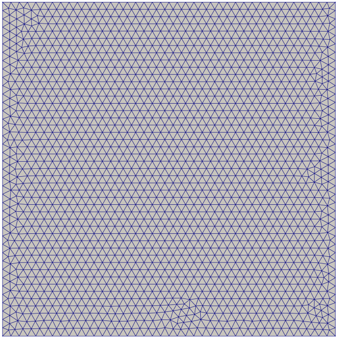
Benchmark problem – generalization of the McWhorter–Sunada problem

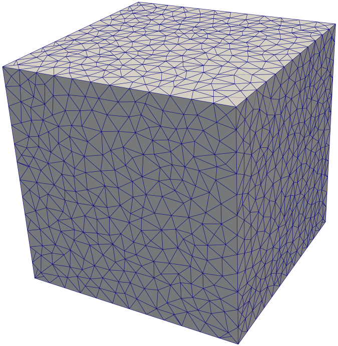
- Two phase flow in porous media
- General dimension (1D, 2D, 3D)
- Radial symmetry
- Point injection in the origin
- Incompressible phases and neglected gravity
- Semi-analytical solution by McWhorter and Sunada (1990) and Fučík *et al.* (2016)

McWhorter–Sunada problem



McWhorter–Sunada problem





McWhorter–Sunada problem

Numerical simulations were performed on:

- 6-core CPU Intel i7-5820K at 3.3 GHz with 15 MB cache
- GPU Tesla K40 with 2880 CUDA cores at 0.745 GHz

McWhorter–Sunada problem 2D

	GPU	CPU										
		1 core		2 cores			4 cores			6 cores		
DOFs	*CT*	*CT*	*GSp*	*CT*	*Eff*	*GSp*	*CT*	*Eff*	*GSp*	*CT*	*Eff*	*GSp*
Orthogonal grids												
960	1,5	0,7	**0,45**	0,4	0,79	**0,28**	0,3	0,52	**0,22**	0,3	0,41	**0,18**
3 720	11,0	13,2	**1,20**	7,6	0,87	**0,69**	4,8	0,68	**0,44**	4,0	0,55	**0,37**
14 640	46,3	197,0	**4,25**	107,5	0,92	**2,32**	65,7	0,75	**1,42**	52,6	0,62	**1,14**
58 080	380,0	4 325,7	**11,38**	2 360,6	0,92	**6,21**	1 448,1	0,75	**3,81**	1 195,8	0,60	**3,15**
231 360	4 449,9	91 166,3	**20,49**	49 004,3	0,93	**11,01**	29 182,1	0,78	**6,56**	24 684,0	0,62	**5,55**
Unstructured meshes												
766	1,5	0,4	**0,27**	0,3	0,60	**0,22**	0,2	0,45	**0,15**	0,2	0,32	**0,14**
2 912	8,9	6,2	**0,70**	3,7	0,84	**0,42**	2,3	0,66	**0,26**	2,0	0,52	**0,23**
11 302	51,1	122,0	**2,39**	67,7	0,90	**1,32**	40,3	0,76	**0,79**	32,5	0,63	**0,64**
44 684	396,1	2 695,6	**6,80**	1 480,7	0,91	**3,74**	855,2	0,79	**2,16**	671,7	0,67	**1,70**
178 648	4 008,3	57 404,2	**14,32**	32 100,5	0,89	**8,01**	18 814,1	0,76	**4,69**	16 414,0	0,58	**4,09**

McWhorter–Sunada problem 3D

	GPU	CPU										
		1 core		2 cores			4 cores			6 cores		
DOFs	*CT*	*CT*	*GSp*	*CT*	*Eff*	*GSp*	*CT*	*Eff*	*GSp*	*CT*	*Eff*	*GSp*
Orthogonal grids												
21 600	2,1	15,2	**7,30**	8,0	0,96	**3,82**	4,4	0,86	**2,13**	3,4	0,75	**1,62**
167 400	30,8	564,3	**18,33**	319,5	0,88	**10,38**	186,7	0,76	**6,07**	150,3	0,63	**4,88**
1 317 600	828,0	20 569,5	**24,84**	12 406,1	0,83	**14,98**	7 092,6	0,73	**8,57**	5 533,7	0,62	**6,68**
10 454 400	31 805,6	(not computed on 1, 2 and 4 cores)								234 066,0		7,36
Unstructured meshes												
5 874	1,4	2,0	**1,48**	1,2	0,85	**0,88**	0,7	0,68	**0,54**	0,6	0,54	**0,46**
15 546	2,6	8,7	**3,30**	4,9	0,89	**1,85**	2,9	0,75	**1,10**	2,3	0,64	**0,86**
121 678	23,9	330,9	**13,87**	184,8	0,90	**7,75**	107,9	0,77	**4,53**	93,4	0,59	**3,92**
973 750	566,2	12 069,5	**21,32**	6 506,3	0,93	**11,49**	3 771,0	0,80	**6,66**	3 306,2	0,61	**5,84**
7 807 218	37 695,3	(not computed on CPU)										

Conclusion

Currently we are working on:

- MPI
- nd-arrays ($\Rightarrow$ nd-grids)
- adaptive grids
- documentation

More about TNL ...

TNL is available at

`www.tnl-project.org`

under MIT license.