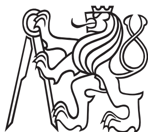
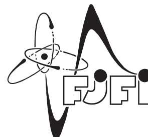
# Template numerical library for modern parallel architectures

**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




IMG 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, Allocator >`
- `DeviceType` says where the array resides
  - `TNL::Devices::Host` for CPU
  - `TNL::Devices::Cuda` for GPU
- `Allocator` performs the memory allocation
  - common CPU and GPU memory allocators
  - page-locked memory allocator
  - CUDA Unified Memory allocator
- I/O operations, elements manipulation ...

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

# Vectors

Vectors add algebraic operations to arrays:

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

# Parallel reduction

Parallel reduction is an 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

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

# Parallel reduction on GPU = 150 lines of code

```
1   template <unsigned int blockSize>
2   __global__ void cudaBlockReduction6(int size, int* dInput, int* dOutput, )
3   {
4      extern __shared__ volatile int sdata[];
5      unsigned int tid = threadIdx.x;
6      unsigned int gid = blockIdx.x*blockSize*2 + threadIdx.x;
7      unsigned int gridSize = blockSize*2*gridDim.x;
8      sdata[tid] = 0;
9      while (gid < size)
10     {
11        sdata[tid] += dInput[gid] + dInput[gid+blockSize];
12        gid += gridSize;
13     }
14     __syncthreads();
15
16     if (blockSize == 1024) {
17        if( tid<512 ) { sdata[ tid ] += sdata[ tid+512 ]; } __syncthreads(); }
18     if (blockSize >= 512) {
19        if( tid<256 ) { sdata[ tid ] += sdata[ tid+256 ]; } __syncthreads(); }
20     if (blockSize >= 256) {
21        if( tid<128 ) { sdata[ tid ] += sdata[ tid+128 ]; } __syncthreads(); }
22     if (blockSize >= 128) {
23        if( tid< 64 ) { sdata[ tid ] += sdata[ tid+ 64 ]; } __syncthreads(); }
24     if (tid < 32) {
25        if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
26        if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
27        if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
28        if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
29        if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
30        if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
31     }
32
33     if (tid == 0) dOutput[blockIdx.x] = sdata[0];
34  }
35
36  bool cudaGridReduction6( const int size,
37                           const int* deviceData,
38                           const int blockSize,
39                           const int desiredGridSize,
40                           int& result )
41  {
42     const int elementsInBlock = 2 * blockSize;
43     int gridSize = size / elementsInBlock + ( size % elementsInBlock != 0 );
44     gridSize = gridSize < desiredGridSize ? gridSize : desiredGridSize;
45     int auxArraySize = gridSize * sizeof( int );
46     int *deviceAux1, *deviceAux2;
47     if( cudaMalloc( ( void** ) &deviceAux1, auxArraySize ) != cudaSuccess ||
48         cudaMalloc( ( void** ) &deviceAux2, auxArraySize ) != cudaSuccess )
49     {
50        cerr << "Not enough memory on device." << endl;
51        return false;
52     }
53
54     int currentSize = size;
55     const int* reductionInput = deviceData;
56     while( currentSize > 1 )
57     {
58        dim3 cudaBlockSize( 0 ), cudaGridSize( 0 );
59        cudaBlockSize. x = blockSize;
60        gridSize = currentSize / elementsInBlock +
61                   ( currentSize % elementsInBlock != 0 );
62        cudaGridSize. x = gridSize < desiredGridSize ? gridSize : desiredGridSize;
63
64        size_t sharedMemory = blockSize * sizeof( int );
65        switch( blockSize )
66        {
67           case 1024:
68              cudaBlockReduction6< 1024 >
69                 <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
70                 ( currentSize, reductionInput, deviceAux1 );
71              break;
72           case 512:
73              cudaBlockReduction6< 512 >
74                 <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
75                 ( currentSize, reductionInput, deviceAux1 );
76              break;
77           case 256:
78              cudaBlockReduction6< 256 >
79                 <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
80                 ( currentSize, reductionInput, deviceAux1 );
81              break;
82           case 128:
83              cudaBlockReduction6< 128 >
84                 <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
85                 ( currentSize, reductionInput, deviceAux1 );
86              break;
87           case 64:
88              cudaBlockReduction6< 64 >
89                 <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
90                 ( currentSize, reductionInput, deviceAux1 );
91              break;
92           case 32:
93              cudaBlockReduction6< 32 >
94                 <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
95                 ( currentSize, reductionInput, deviceAux1 );
96              break;
97           case 16:
98              cudaBlockReduction6< 16 >
99                 <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
100                ( currentSize, reductionInput, deviceAux1 );
101             break;
102          case 8:
103             cudaBlockReduction6< 8 >
104                <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
105                ( currentSize, reductionInput, deviceAux1 );
106             break;
107          case 4:
108             cudaBlockReduction6< 4 >
109                <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
110                ( currentSize, reductionInput, deviceAux1 );
111             break;
112          case 2:
113             cudaBlockReduction6< 2 >
114                <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
115                ( currentSize, reductionInput, deviceAux1 );
116             break;
117          case 1:
118             cudaBlockReduction6< 1 >
119                <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
120                ( currentSize, reductionInput, deviceAux1 );
121             break;
122
123       }
124       cudaError error = cudaGetLastError();
125       if( error != cudaSuccess )
126       {
127          cerr << "The CUDA kernel ended with error code " << error << "."
128               << endl;
129          return false;
130       }
131       currentSize = cudaGridSize. x;
132
133       reductionInput = deviceAux1;
134       int* tmp = deviceAux1;
135       deviceAux1 = deviceAux2;
136       deviceAux2 = tmp;
137    }
138    if( cudaMemcpy( &result,
139                    reductionInput,
140                    sizeof( int ),
141                    cudaMemcpyDeviceToHost ) != cudaSuccess )
142    {
143       cerr << "An error occured while transfering data from device to host."
144            << endl;
145       return false;
146    }
147    cudaFree( deviceAux1 );
148    cudaFree( deviceAux2 );
149    return true;
150 }
```

# Flexible parallel reduction in TNL

Take a look at scalar product:

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

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

```
1  float a[ size ], b[size ];
2
3  ...
4
5  auto fetch = [=] (int i)->float { return a[i]*b[i];};
6  auto reduce = [] (float& x, const float& y) { x += y;};
7
8  float result( 0.0 );
9  for( int i = 0; i < size; i++ )
10     reduce( result, fetch( i ) );
```

# Flexible parallel reduction in TNL

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

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

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

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

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

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

# Expression Templates in TNL

Algebraic vector expressions are handled by **expression templates**:

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

- simple
- works for both CPU and GPU
- efficient
  - **one** loop on CPU
  - specialized **one** CUDA kernel for each expression on GPU

# Expression Templates & Parallel Reduction in TNL

Example:

```
1 using Vector = Vector< float, Devices::Cuda, int >;
2 Vector a( 100 ), b( 100 ), c( 100 ), d( 100 );
3 ...
4 float scalarProduct = ( a, b + 3 * c );
5 d = a + b * c + sin( d );
6 a = min( b, c );
7 float min_a = min( a );
8 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
  - Intel Core i7-5820K, 3.3GHz, 16MB 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 | 17.4 | 5.0 | **0.3** | 49.3 | 69.9 | **1.41** |
| 200k | 17.4 | 5.0 | **0.3** | 90.1 | 108.6 | **1.20** |
| 400k | 17.7 | 5.0 | **0.3** | 142.2 | 159.1 | **1.11** |
| 800k | 13.7 | 4.8 | **0.3** | 207.4 | 233.4 | **1.12** |
| 1.6M | 12.6 | 4.8 | **0.4** | 313.6 | 333.3 | **1.06** |
| 3.2M | 12.8 | 4.6 | **0.4** | 381.0 | 403.7 | **1.05** |
| 6.4M | 12.7 | 4.6 | **0.4** | 417.1 | 431.8 | **1.03** |

BW = effective memory bandwidth in GB/s

# Expression Templates in TNL

Vector addition: `x += a`.

| Size | CPU | | | GPU | | |
|---|---|---|---|---|---|---|
| | BLAS | TNL | | cuBLAS | TNL | |
| | BW | BW | Speed-up | BW | BW | Speed-up |
| 100k | 34.0 | 42.0 | **1.2** | 152.2 | 174.8 | **1.14** |
| 200k | 35.1 | 43.0 | **1.2** | 196.6 | 216.1 | **1.09** |
| 400k | 31.7 | 36.7 | **1.2** | 277.6 | 294.4 | **1.06** |
| 800k | 19.7 | 19.3 | **0.97** | 326.2 | 333.6 | **1.02** |
| 1.6M | 18.1 | 17.3 | **0.95** | 362.5 | 374.2 | **1.03** |
| 3.2M | 18.4 | 17.6 | **0.95** | 422.4 | 436.8 | **1.03** |
| 6.4M | 18.3 | 17.5 | **0.95** | 456.6 | 469.8 | **1.02** |

BW = effective memory bandwidth in GB/s

# 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 | 23.6 | 42.0 | **1.8** | 188.3 | 190.9 | **1.01** |
| 200k | 23.5 | 41.8 | **1.8** | 218.0 | 230.7 | **1.05** |
| 400k | 20.9 | 37.4 | **1.8** | 243.1 | 305.9 | **1.25** |
| 800k | 13.7 | 18.7 | **1.4** | 263.8 | 353.0 | **1.33** |
| 1.6M | 12.2 | 16.8 | **1.4** | 285.9 | 389.4 | **1.36** |
| 3.2M | 12.3 | 17.5 | **1.4** | 312.9 | 442.8 | **1.41** |
| 6.4M | 12.2 | 17.3 | **1.4** | 327.3 | 471.9 | **1.44** |

BW = effective memory bandwidth in GB/s

# 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 | 19.3 | 41.5 | **2.2** | 194.7 | 236.5 | **1.21** |
| 200k | 19.7 | 41.7 | **2.1** | 228.3 | 277.6 | **1.21** |
| 400k | 17.3 | 35.9 | **2.1** | 218.3 | 330.9 | **1.51** |
| 800k | 11.7 | 19.3 | **1.6** | 233.3 | 370.6 | **1.58** |
| 1.6M | 10.4 | 17.0 | **1.6** | 249.6 | 403.4 | **1.61** |
| 3.2M | 10.2 | 17.3 | **1.7** | 266.6 | 444.8 | **1.66** |
| 6.4M | 10.2 | 17.3 | **1.7** | 276.6 | 471.3 | **1.70** |

BW = effective memory bandwidth in GB/s

# 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.

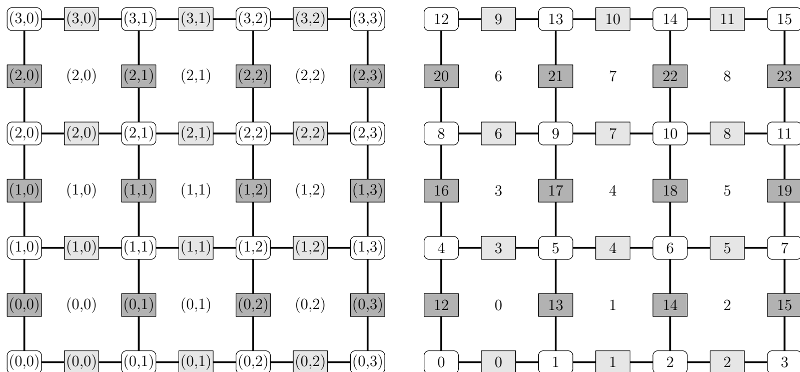
# Numerical meshes

TNL supports

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

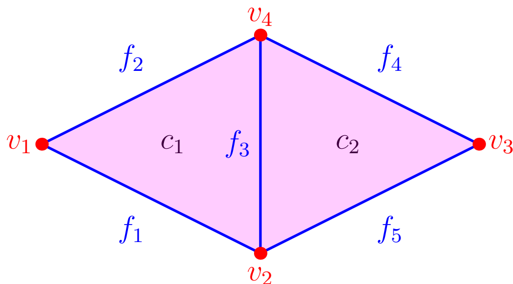
# Structured grids

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




Grid provides mapping between coordinates and global indexes.

# Unstructured meshes




$$I_{0,1} = \left( \begin{array}{c|ccccc} & f_1 & f_2 & f_3 & f_4 & f_5 \\ \hline v_1 & 1 & 1 & & & \\ v_2 & 1 & & 1 & & 1 \\ v_3 & & & & 1 & 1 \\ v_4 & & 1 & 1 & 1 & \end{array} \right) \quad I_{0,2} = \left( \begin{array}{c|cc} & c_1 & c_2 \\ \hline v_1 & 1 & \\ v_2 & 1 & 1 \\ v_3 & & 1 \\ v_4 & 1 & 1 \end{array} \right)$$

# 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

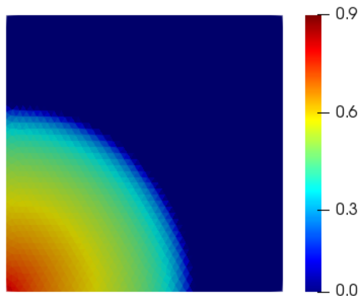
R. Fučík, J.Klinkovský, T. Oberhuber, J. Mikyška, *Multidimensional Mixed–Hybrid Finite Element Method for Compositional Two–Phase Flow in Heterogeneous Porous Media and its Parallel Implementation on GPU*, Computer Physics Communications 238, pp. 165–180, 2019.

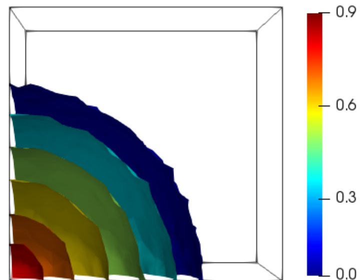
# McWhorter–Sunada problem

Benchmark problem – generalization of the McWhorter–Sunada problem

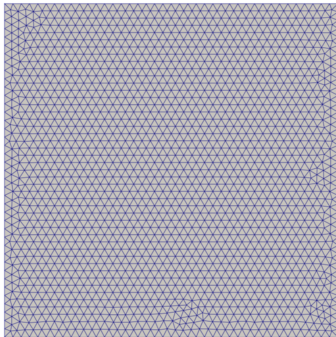
- Two phase flow in porous media
- General dimension (1D, 2D, 3D)
- Radial symmetry
- Point injection in the origin
- Incompressible phases and neglected gravity
- Semi-analytical solution by McWhorter and Sunada (1990) and Fučík *et al.* (2016)

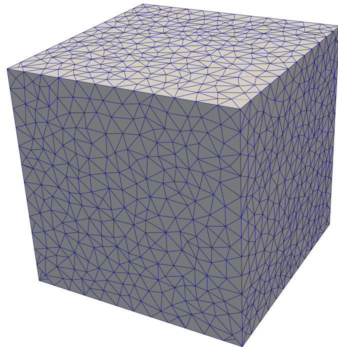
# McWhorter–Sunada problem



# 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.