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?

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

\[ \approx 8,000 \, \text{\$} \]
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 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 ...

1. Array< float, Devices::Cuda, int > a( 100 );
2. 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 ...
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

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

```c
float sum( 0.0 )
for( int i = 0; i < size; i++ )
    sum += a[ i ];
```
Parallel reduction on GPU = 150 lines of code

```c
int c1 = 1;
int c2 = 2;
int c3 = 3;
int c4 = 4;
int c5 = 5;
int c6 = 6;
int c7 = 7;
int c8 = 8;
int c9 = 9;
int c10 = 10;
int c11 = 11;
int c12 = 12;
int c13 = 13;
int c14 = 14;
int c15 = 15;
int c16 = 16;
int c17 = 17;
int c18 = 18;
int c19 = 19;
int c20 = 20;
int c21 = 21;
int c22 = 22;
int c23 = 23;
int c24 = 24;
int c25 = 25;
int c26 = 26;
int c27 = 27;
int c28 = 28;
int c29 = 29;
int c30 = 30;
int c31 = 31;
int c32 = 32;
int c33 = 33;
int c34 = 34;
int c35 = 35;
int c36 = 36;
int c37 = 37;
int c38 = 38;
int c39 = 39;
int c40 = 40;
int c41 = 41;
int c42 = 42;
int c43 = 43;
int c44 = 44;
int c45 = 45;
int c46 = 46;
int c47 = 47;
int c48 = 48;
int c49 = 49;
int c50 = 50;
int c51 = 51;
int c52 = 52;
int c53 = 53;
int c54 = 54;
int c55 = 55;
int c56 = 56;
int c57 = 57;
int c58 = 58;
int c59 = 59;
int c60 = 60;
int c61 = 61;
int c62 = 62;
int c63 = 63;
int c64 = 64;
int c65 = 65;
int c66 = 66;
int c67 = 67;
int c68 = 68;
int c69 = 69;
int c70 = 70;
int c71 = 71;
int c72 = 72;
int c73 = 73;
int c74 = 74;
int c75 = 75;
int c76 = 76;
int c77 = 77;
int c78 = 78;
int c79 = 79;
int c80 = 80;
int c81 = 81;
int c82 = 82;
int c83 = 83;
int c84 = 84;
int c85 = 85;
int c86 = 86;
int c87 = 87;
int c88 = 88;
int c89 = 89;
int c90 = 90;
int c91 = 91;
int c92 = 92;
int c93 = 93;
int c94 = 94;
int c95 = 95;
int c96 = 96;
int c97 = 97;
int c98 = 98;
int c99 = 99;
int c100 = 100;

// More code...
```

---

T. Oberhuber et al. (FNSPE CTU in Prague)
Take a look at scalar product:

```c
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:

```c
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 ) );
```
Another example - $l_p$-norm:

```cpp
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:

```cpp
1 bool zero = true;
2 const float p = 2.0;
3 float a[size], b[size];
...
5 auto fetch = [=] (int i) -> bool { return (a[i] == b[i]); };
6 auto reduce = [] (float& x, const float& y) { x = x && y; };
7
8 float result( zero );
9 for (int i = 0; i < size; i++)
10    reduce( result, fetch( i ) );
```
To perform the same on GPU in TNL just add \texttt{\_\_cuda\_callable\_\_} to lambdas...

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

... and for certain reasons, deliver \texttt{volatile} version of \texttt{reduce}:

\begin{verbatim}
auto volatileReduce = [] __cuda_callable__ (volatile float& x, volatile const float& y) {
    x = x && y;
};
\end{verbatim}

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

\begin{verbatim}
Reduction< Devices::Cuda >::reduce( size, reduce, volatileReduce, fetch, zero );
\end{verbatim}
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:

```c
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
We can use operators overloading in C++:

\[ x = a + 2 \times b + 3 \times c; \]

- it is very simple and easy to read
- can be performed in parallel on multicore CPUs or GPUs
- it is inefficient
We can use BLAS/cuBLAS:

1. `cublasHandle_t handle;`
2. `cublasSaxpy(handle, size, 1.0, y, 1, x, 1);`
3. `cublasSaxpy(handle, size, 2.0, b, 1, x, 1);`
4. `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...

\[ x = a + 2 \times b + 3 \times c; \]

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

In TNL, \(a\), \(b\) and \(c\) are VectorViews

\[ \text{VectorView}<\text{Real}, \text{Device}, \text{Index}> a, b, c; \]

- it is simple and easy to read
- works for any type \text{Real} (float/double) and any Device (CPU/GPU)
- it is very efficient
Example:

```cpp
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: \( r = (x, y) \).

<table>
<thead>
<tr>
<th>Size</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLAS</td>
<td>TNL</td>
</tr>
<tr>
<td></td>
<td>BW</td>
<td>BW</td>
</tr>
<tr>
<td>100k</td>
<td>20.8</td>
<td>2.7</td>
</tr>
<tr>
<td>200k</td>
<td>18.5</td>
<td>12.2</td>
</tr>
<tr>
<td>400k</td>
<td>18.4</td>
<td>13.3</td>
</tr>
<tr>
<td>800k</td>
<td>11.9</td>
<td>13.2</td>
</tr>
<tr>
<td>1.6M</td>
<td>13.6</td>
<td>15.6</td>
</tr>
<tr>
<td>3.2M</td>
<td>14.9</td>
<td>17.9</td>
</tr>
<tr>
<td>6.4M</td>
<td>16.7</td>
<td>17.0</td>
</tr>
</tbody>
</table>
Expression Templates in TNL

Vector addition: \( x += a \).

<table>
<thead>
<tr>
<th>Size</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLAS</td>
<td>TNL</td>
</tr>
<tr>
<td></td>
<td>BW</td>
<td>BW</td>
</tr>
<tr>
<td>100k</td>
<td>46.0</td>
<td>11.6</td>
</tr>
<tr>
<td>200k</td>
<td>42.3</td>
<td>7.0</td>
</tr>
<tr>
<td>400k</td>
<td>16.9</td>
<td>29.3</td>
</tr>
<tr>
<td>800k</td>
<td>16.8</td>
<td>23.9</td>
</tr>
<tr>
<td>1.6M</td>
<td>17.3</td>
<td>25.1</td>
</tr>
<tr>
<td>3.2M</td>
<td>17.5</td>
<td>25.3</td>
</tr>
<tr>
<td>6.4M</td>
<td>17.4</td>
<td>25.7</td>
</tr>
</tbody>
</table>
Vector addition: \( x \ += a + b \).

<table>
<thead>
<tr>
<th>Size</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLAS</td>
<td>TNL</td>
</tr>
<tr>
<td></td>
<td>BW</td>
<td>BW</td>
</tr>
<tr>
<td>100k</td>
<td>30.3</td>
<td>29.3</td>
</tr>
<tr>
<td>200k</td>
<td>30.5</td>
<td>31.8</td>
</tr>
<tr>
<td>400k</td>
<td>13.7</td>
<td>32.8</td>
</tr>
<tr>
<td>800k</td>
<td>11.6</td>
<td>23.0</td>
</tr>
<tr>
<td>1.6M</td>
<td>11.7</td>
<td>24.4</td>
</tr>
<tr>
<td>3.2M</td>
<td>11.7</td>
<td>24.8</td>
</tr>
<tr>
<td>6.4M</td>
<td>11.7</td>
<td>25.7</td>
</tr>
</tbody>
</table>
Expression Templates in TNL

Vector addition: \( x += a + b + c \).

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


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

\[ I_{0,1} = \begin{pmatrix} f_1 & f_2 & f_3 & f_4 & f_5 \\ v_1 & 1 & 1 \\ v_2 & 1 & 1 & 1 \\ v_3 & 1 & 1 \\ v_4 & 1 & 1 & 1 \end{pmatrix} \quad I_{0,2} = \begin{pmatrix} c_1 & c_2 \\ v_1 & 1 \\ v_2 & 1 \\ v_3 & 1 \\ v_4 & 1 \end{pmatrix} \]
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


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

Initial condition:

\[ Z_j(\vec{x}, 0) = Z_j^{\text{ini}}(\vec{x}), \quad \forall \vec{x} \in \Omega, \quad j = 1, \ldots, n, \]

Boundary conditions:

\[ Z_j(\vec{x}, t) = Z_j^D(\vec{x}, t), \quad \forall \vec{x} \in \Gamma_j^D \subset \partial\Omega, \quad j = 1, \ldots, n, \]
\[ \vec{v}_i(\vec{x}, t) \cdot \vec{n}_{\partial\Omega}(\vec{x}) = v_i^N(\vec{x}, t), \quad \forall \vec{x} \in \Gamma_i^N \subset \partial\Omega, \quad i = 1, \ldots, n, \]

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

\[ \vec{v}_i = - \sum_{j=1}^{n} D_{i,j} \nabla Z_j + w_i. \]
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

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
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

<table>
<thead>
<tr>
<th>DOFs</th>
<th>GPU</th>
<th>1 core</th>
<th></th>
<th>2 cores</th>
<th></th>
<th>CPU</th>
<th>4 cores</th>
<th></th>
<th>6 cores</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CT</td>
<td></td>
<td></td>
<td></td>
<td>CT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CT</td>
<td>GSp</td>
<td></td>
<td></td>
<td>CT</td>
<td>Eff</td>
<td>GSp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CT</td>
<td>GSp</td>
<td></td>
<td></td>
<td>CT</td>
<td>Eff</td>
<td>GSp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CT</td>
<td>GSp</td>
<td></td>
<td></td>
<td>CT</td>
<td>Eff</td>
<td>GSp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Orthogonal grids**

<table>
<thead>
<tr>
<th>DOFs</th>
<th>GPU</th>
<th>1 core</th>
<th>2 cores</th>
<th>CPU</th>
<th>4 cores</th>
<th>6 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>960</td>
<td>1.5</td>
<td>0.7</td>
<td>0.45</td>
<td>0.3</td>
<td>0.52</td>
<td>0.22</td>
</tr>
<tr>
<td>3720</td>
<td>11.0</td>
<td>13.2</td>
<td>1.20</td>
<td>7.6</td>
<td>0.87</td>
<td>0.69</td>
</tr>
<tr>
<td>14640</td>
<td>46.3</td>
<td>197.0</td>
<td>4.25</td>
<td>107.5</td>
<td>0.92</td>
<td>2.32</td>
</tr>
<tr>
<td>58080</td>
<td>380.0</td>
<td>4325.7</td>
<td>11.38</td>
<td>2360.6</td>
<td>0.92</td>
<td>6.21</td>
</tr>
<tr>
<td>231360</td>
<td>4449.9</td>
<td>91166.3</td>
<td>20.49</td>
<td>49004.3</td>
<td>0.93</td>
<td>11.01</td>
</tr>
</tbody>
</table>

**Unstructured meshes**

<table>
<thead>
<tr>
<th>DOFs</th>
<th>GPU</th>
<th>1 core</th>
<th>2 cores</th>
<th>CPU</th>
<th>4 cores</th>
<th>6 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>766</td>
<td>1.5</td>
<td>0.4</td>
<td>0.27</td>
<td>0.3</td>
<td>0.60</td>
<td>0.22</td>
</tr>
<tr>
<td>2912</td>
<td>8.9</td>
<td>6.2</td>
<td>0.70</td>
<td>3.7</td>
<td>0.84</td>
<td>0.42</td>
</tr>
<tr>
<td>11302</td>
<td>51.1</td>
<td>122.0</td>
<td>2.39</td>
<td>67.7</td>
<td>0.90</td>
<td>1.32</td>
</tr>
<tr>
<td>44684</td>
<td>396.1</td>
<td>2695.6</td>
<td>6.80</td>
<td>1480.7</td>
<td>0.91</td>
<td>3.74</td>
</tr>
<tr>
<td>178648</td>
<td>4008.3</td>
<td>57404.2</td>
<td>14.32</td>
<td>32100.5</td>
<td>0.89</td>
<td>8.01</td>
</tr>
</tbody>
</table>
## McWhorter–Sunada problem 3D

<table>
<thead>
<tr>
<th>DOFs</th>
<th>GPU</th>
<th>1 core</th>
<th>2 cores</th>
<th>CPU</th>
<th>4 cores</th>
<th>6 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CT</td>
<td>CT</td>
<td>GSp</td>
<td>CT</td>
<td>CT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orthogonal grids</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21 600</td>
<td>2,1</td>
<td>15,2</td>
<td><strong>7,30</strong></td>
<td>8,0</td>
<td>0,96</td>
<td><strong>3,82</strong></td>
</tr>
<tr>
<td>167 400</td>
<td>30,8</td>
<td>564,3</td>
<td><strong>18,33</strong></td>
<td>319,5</td>
<td>0,88</td>
<td><strong>10,38</strong></td>
</tr>
<tr>
<td>1 317 600</td>
<td>828,0</td>
<td>20 569,5</td>
<td><strong>24,84</strong></td>
<td>12 406,1</td>
<td>0,83</td>
<td><strong>14,98</strong></td>
</tr>
<tr>
<td>10 454 400</td>
<td>31 805,6</td>
<td>(not computed on 1, 2 and 4 cores)</td>
<td>234 066,0</td>
<td>7,36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unstructured meshes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 874</td>
<td>1,4</td>
<td>2,0</td>
<td><strong>1,48</strong></td>
<td>1,2</td>
<td>0,85</td>
<td><strong>0,88</strong></td>
</tr>
<tr>
<td>15 546</td>
<td>2,6</td>
<td>8,7</td>
<td><strong>3,30</strong></td>
<td>4,9</td>
<td>0,89</td>
<td><strong>1,85</strong></td>
</tr>
<tr>
<td>121 678</td>
<td>23,9</td>
<td>330,9</td>
<td><strong>13,87</strong></td>
<td>184,8</td>
<td>0,90</td>
<td><strong>7,75</strong></td>
</tr>
<tr>
<td>973 750</td>
<td>566,2</td>
<td>12 069,5</td>
<td><strong>21,32</strong></td>
<td>6 506,3</td>
<td>0,93</td>
<td><strong>11,49</strong></td>
</tr>
<tr>
<td>7 807 218</td>
<td>37 695,3</td>
<td>(not computed on CPU)</td>
<td>38/40</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Currently we are working on:

- MPI
- nd-arrays (⇒ nd-grids)
- adaptive grids
- documentation
TNL is available at

www.tnl-project.org

under MIT license.