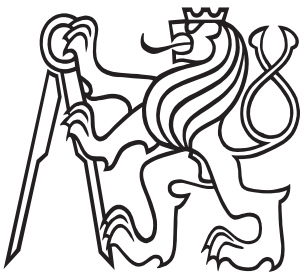
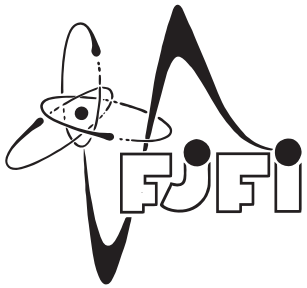
TNL:FDM on GPU in C++

Tomáš Oberhuber Vítězslav Žabka Vladimír Klement
Vít Hanousek Radek Fučík Jakub Klinkovský
Tomáš Sobotík Ondřej Székely Libor Bakajsa
Martin Schäfer Jakub Kaňuk Jan Vacata



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



CJPS 2016, Krakow

Template Numerical Library

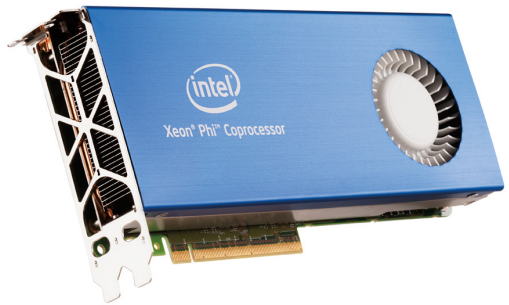
TNL = Template Numerical Library

Aim of the project is to develop a numerical library which is:

1. **efficient**
   - C++, CUDA for GPUs
2. **flexible**
   - C++ templates
3. **user friendly**
   - we hide C++ templates as much as possible

GPUs and MICs

- performance of CPUs does not grow as fast as it used to
- memory modules are $\approx 200 \times$ slower than CPU
- new accelerators appeared
  - GPU – graphical processing unit (Nvidia Tesla)
  - MIC – many integrated cores (Intel Xeon Phi)



- they are massively parallel – up to thousands of computing cores
- they have $\approx 20 \times$ faster memory modules

Difficulties in programming GPUs?

GPUs (and MICs)

- have own memory
- are connected to CPU by slow PCI Express
- require data stored in large contiguous blocks
- have many cores supporting vectorization

Therefore,

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

It is good reason for development of numerical library which makes GPUs (and MICs) easily accessible.

Outline

1. data structures
2. solvers
3. PDE solver
4. performance results

Arrays

- arrays are basic structures for memory management
- `TNL::Array< ElementType, DeviceType, IndexType >`
  - `DeviceType` - CPU (`TNL::Devices::Host`) or GPU (`TNL::Devices::Cuda`)
    - memory accesses to CPU and GPU are checked at compile time
- there are methods for
  - memory allocation – `setSize`, `setLike`
  - I/O operations – `load`, `save`
  - operators – =, ==, «
  - elements manipulation
    - `getElement`, `setElement` – callable only from host for both `Host`/`Cuda`
    - `__cuda_callable__ operator[]` – callable from host for `Host` and from CUDA kernels for `Cuda`

Vectors

`TNL::Vector< RealType, DeviceType, IndexType >`

- vectors extend arrays with algebraic operations (BLAS)
  - operators – `+=`, `-=`, `*=`, `/=`
  - scalar product – `scalarProduct`
  - parallel reduction operations – `lpNorm`, `min`, `max`, ...

Matrix formats

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

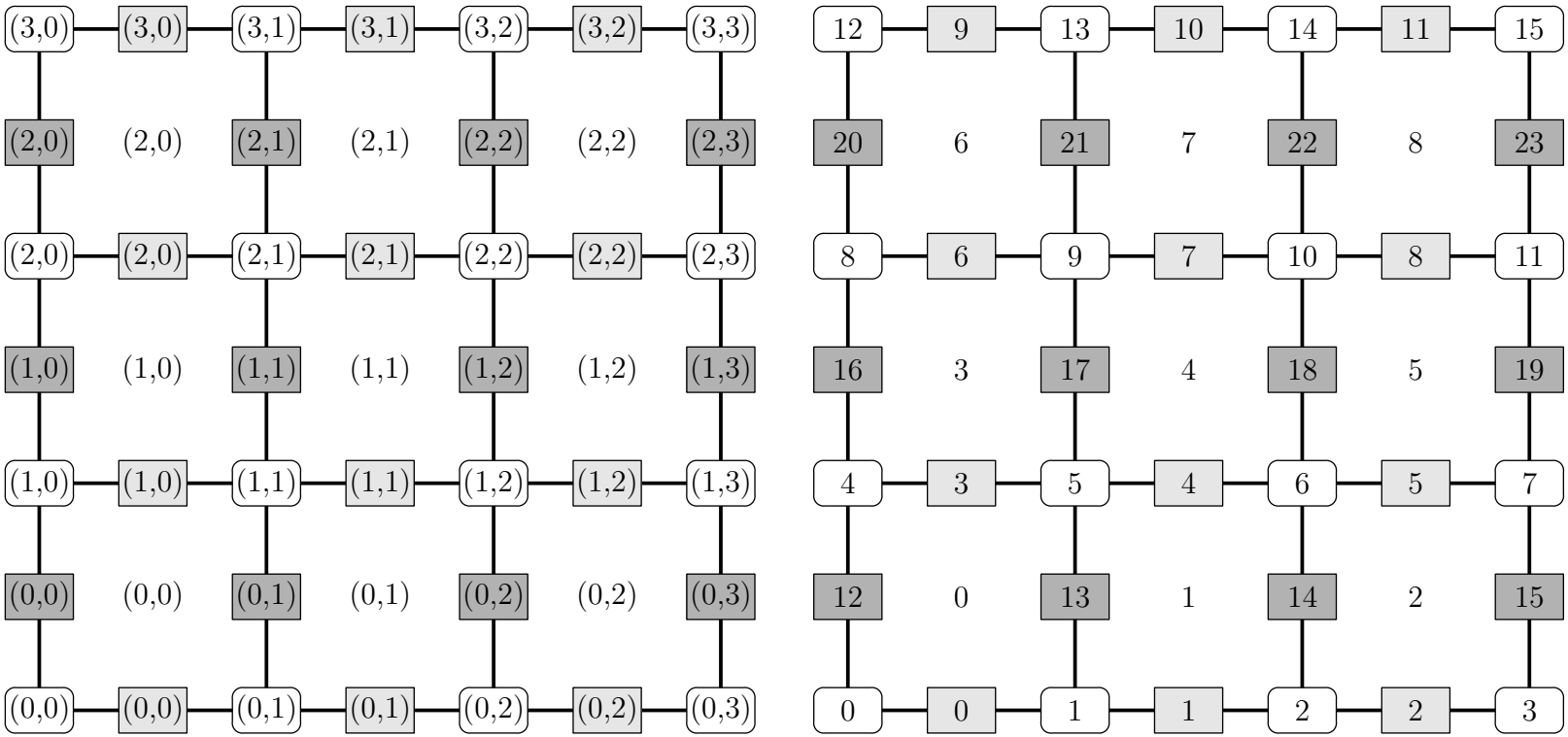
- dense matrix format
- tridiagonal and multidiagonal matrix format
- Ellpack format
- CSR format
- SlicedEllpack format
  - Oberhuber T., Suzuki A., Vacata J., *New Row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA*, Acta Technica, 2011, vol. 56, no. 4, pp. 447-466.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

Grids

TNL supports 1D, 2D and 3D structured grids:
`TNL::Meshes::Grid< Dimensions, Real, Device, Index >`

- it provides indexing and coordinates mapping for the mesh entities:
- each grid/mesh consists of mesh entities referred by their dimensions
- in 2D
  - cells – 2 dimensions
  - faces – 1 dimension
  - vertices – 0 dimensions
- in 3D
  - cells – 3 dimensions
  - faces – 2 dimensions
  - edges – 1 dimensions
  - vertices – 0 dimensions

2D grid with 3×3 cells




```
auto neighbourEntities = entity.template getNeighbourEntities< 2 >();
Index& center = entity.getIndex(); // 4
Index& east  = neighbourEntities.template getEntityIndex< 1, 0>(); // 5
Index& west  = neighbourEntities.template getEntityIndex< -1, 0>(); // 3
Index& north = neighbourEntities.template getEntityIndex< 0, 1>(); // 7
Index& south = neighbourEntities.template getEntityIndex< 0,-1>(); // 1
```

Solvers

- ODEs solvers
  - Euler, Runge-Kutta-Merson – CPU and GPU
    - 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.
- solvers of linear systems
  - Krylov subspace methods (CG, BiCGSTab, GMRES, TFQMR) – CPU and GPU
    - 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.
  - SOR method – CPU only

Configuration parameters

- TNL offers configuration parameters management
- configuration description is done in methods `configSetup`
- one may define configuration parameter
  - type
  - default value
  - required
  - description
  - admissible values

```
static void configSetup( TNL::Config::ConfigDescription& config )
{
   config.addEntry< double >
                  ( "time-step",
                    "Time step for the time discretization.", 1.0 );
   config.addRequiredEntry< double >
                           ( "stop-time",
                             "Stop time of the time-dependent simulation." );
   config.addEntry< tnlString >
                  ( "boundary-conditions",
                    "Type of the boundary conditions." );
      config.addEntryEnum< tnlString >( "dirichlet" );
      config.addEntryEnum< tnlString >( "neumann" );
}
...

bool setup( TNL::Config::ParameterContainer& parameters )
{
   double timeStep = parameters.getParameter< double >( "time-step" );
}
```

PDE solver

- we have building blocks of PDE solvers
  - grids/meshes
  - sparse matrices
  - solvers (of ODEs and linear systems)
- but it is still far from the main PDE solver

PDE solver

- consider the heat equation as model problem

$$\begin{aligned} \frac{\partial u(\mathbf{x},t)}{\partial t} - \Delta u(\mathbf{x},t) &= 0 \quad \text{on } \Omega \times (0,T], && (1)\\ u(\mathbf{x},0) &= u_{ini}(\mathbf{x}) \quad \text{on } \Omega, && (2)\\ u(\mathbf{x},t) &= g(\mathbf{x},t) \quad \text{on} \partial\Omega \times (0,T]. && (3) \end{aligned}$$

- explicit scheme (by method of lines) reads as

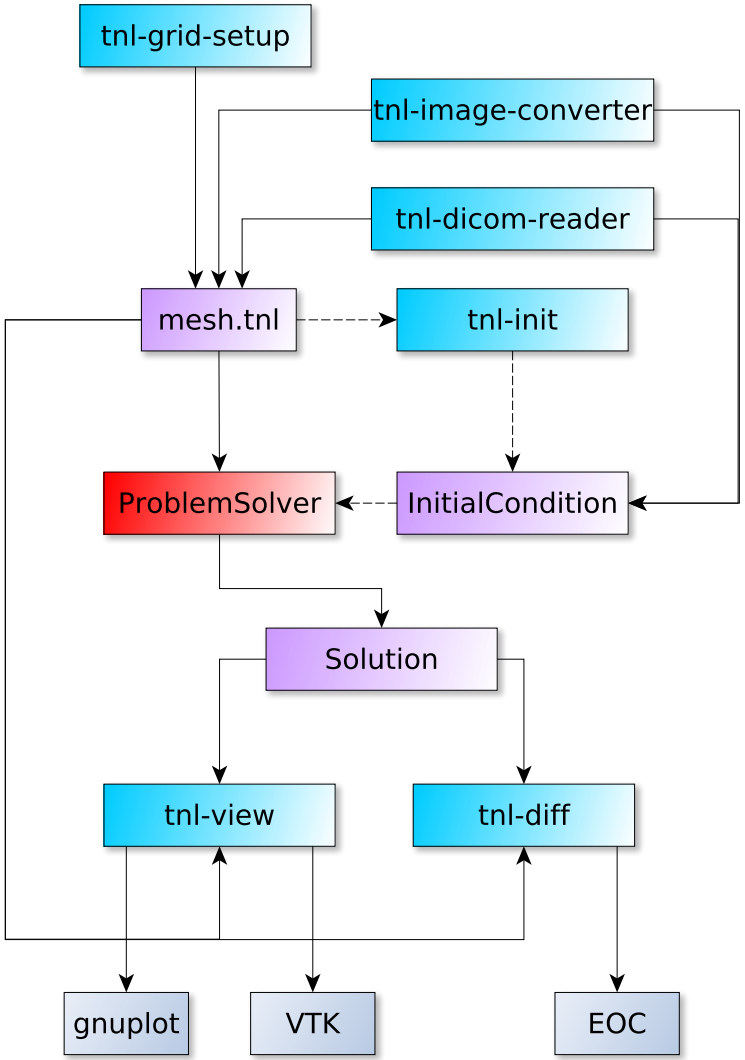
$$\frac{\mathrm{d}}{\mathrm{d}t} u_{ij}(t) = \frac{1}{h^2}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}\right) = F_{ij},$$

- semi-implicit scheme reads as
  - $$\frac{u_{ij}^{k+1} - u_{ij}^{k}}{\tau} - \frac{1}{h^2}\left(u_{i+1,j}^{k+1} + u_{i-1,j}^{k+1} + u_{i,j+1}^{k+1} + u_{i,j-1}^{k+1} - 4u_{ij}^{k+1}\right) = 0,$$
  - i.e.
  $$\lambda u_{i+1,j}^{k+1} + \lambda u_{i-1,j}^{k+1} + \lambda u_{i,j+1}^{k+1} + \lambda u_{i,j-1}^{k+1} + \left(1 - 4\lambda u_{ij}^{k+1}\right) = u_{ij}^{k}$$
  - which is linear system $\mathbb{A}\mathbf{u}^{k+1} = \mathbf{b}$

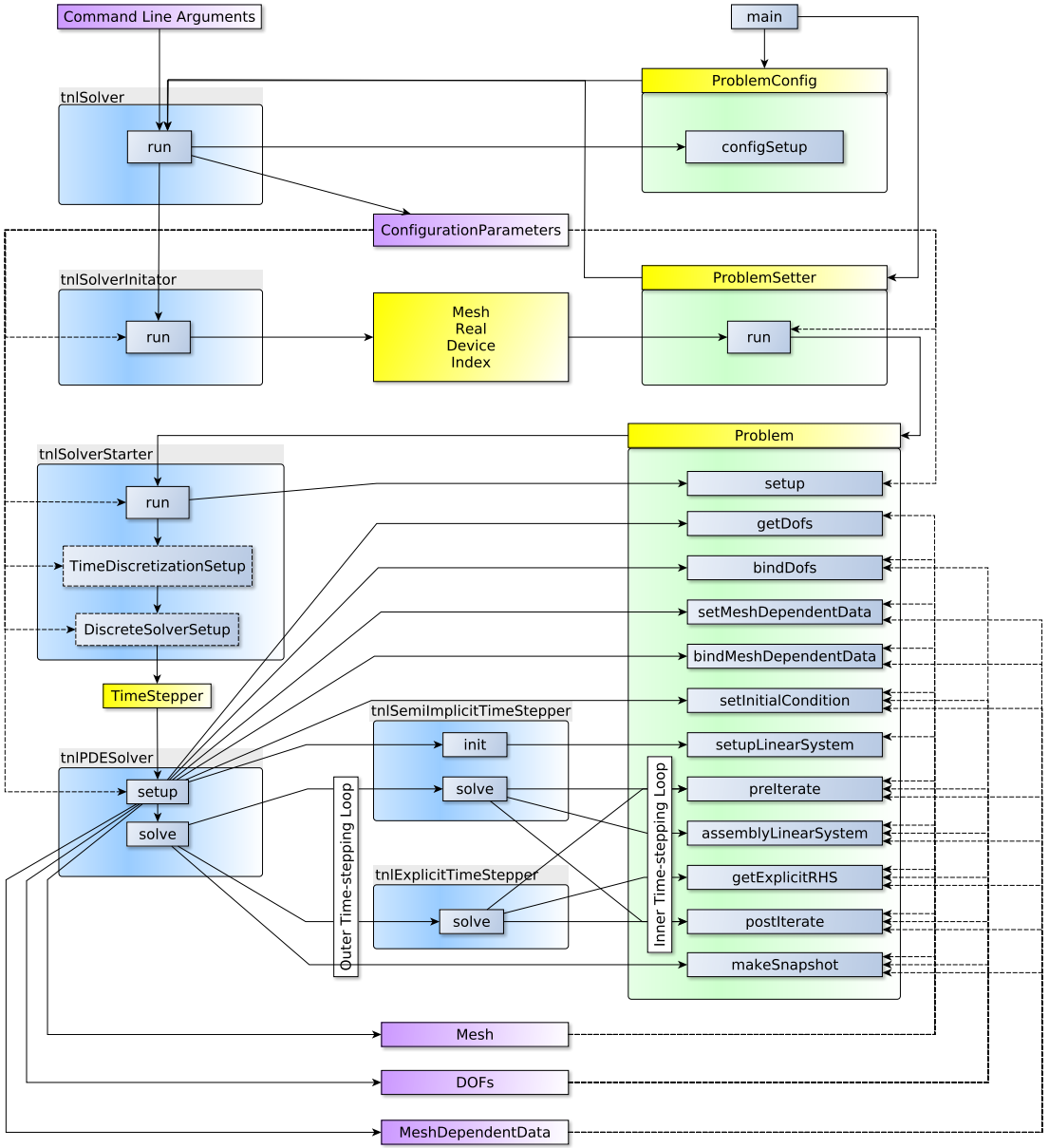
PDE solver

- we need to
  - setup mesh
  - setup initial and boundary conditions
  - allocate DOFs
  - setup discrete solver
  - evaluate numerical scheme
    - explicitly $\rightarrow$ explicit update $\frac{\mathrm{d}}{\mathrm{d}t} u_{ij}(t) = F_{ij} \ \forall ij$
    - (semi-)implicitly $\rightarrow$ assembly linear system $\mathbb{A}\mathbf{u}^{k+1} = \mathbf{b}$
  - perform snapshots of the time dependent solution
- TNL aims to simplify these steps by
  - offering some command-line tools
  - implementing a skeleton of PDE solver

TNL tools


tnl-grid-setup
tnl-image-converter
tnl-dicom-reader
mesh.tnl
tnl-init
ProblemSolver
InitialCondition
Solution
tnl-view
tnl-diff
gnuplot
VTK
EOC

PDE solver


Command Line Arguments
main
tnlSolver
run
ProblemConfig
configSetup
ConfigurationParameters
tnlSolverInitator
run
Mesh
Real
Device
Index
ProblemSetter
run
Problem
tnlSolverStarter
run
TimeDiscretizationSetup
DiscreteSolverSetup
TimeStepper
setup
getDofs
bindDofs
setMeshDependentData
bindMeshDependentData
setInitialCondition
setupLinearSystem
preIterate
assemblyLinearSystem
getExplicitRHS
postIterate
makeSnapshot
tnlSemiImplicitTimeStepper
init
solve
tnlIPDESolver
setup
solve
tnlExplicitTimeStepper
solve
Outer Time-stepping Loop
Inner Time-stepping Loop
Mesh
DOFs
MeshDependentData

Simple implementation of explicit scheme

- the method `Problem::getExplicitRHS` for explicit scheme may look as

```
void Problem::getExplicitRHS( const RealType& time,
                              const RealType& tau,
                              const MeshType& mesh,
                              DofVectorType& u,
                              DofVectorType& fu )
{
   for( int i = 0; i < mesh.getDimensions().x(); i++ )
      for( int j = 0; j < mesh.getDimensions().y(); j++ )
      {
         if( mesh.isBoundaryCell( i, j ) )
         {
            /****
             * Set boundary conditions
             */
             IndexType idx = mesh.getCellIndex( i, j );
             ...
         }
      }

   for( int i = 0; i < mesh.getDimensions().x(); i++ )
      for( int j = 0; j < mesh.getDimensions().y(); j++ )
      {
         if( ! mesh.isBoundaryCell( i, j ) )
         {
            /****
             * Approximate the differential operator
             */
             IndexType idx = mesh.getCellIndex( i, j );
            ...
         }
      }
}
```

More flexible implementation of explicit schemes

It is simple but it works only ...

- on CPU
- for structured grids
- 2D problems

The user would have to write template specialization for

- GPU
- unstructured meshes
- 1D or 3D problems
- other parallel architectures like MIC or MPI

We replace:

- code inside the for loops by
  - differential operators – `operator()`, `setMatrixElements`
  - functions – `operator()`
- for loops by objects iterating over the grids
  - explicit updater
  - linear system assembler

Functions

We distinguish:

- *analytical functions*
  - they are defined on $\mathbb{R}^n$
  - `operator()(const Point& v, const Real& time)`
- *discrete functions* (`TNL::Functions::MeshFunction`)
  - they are defined on cells, faces, edges or vertices of a numerical mesh
  - values are stored in an array
  - `operator()( const MeshEntity& entity, const Real& time)`
  - `operator[]( const Index& entityIndex )`

Operators

- *analytical operators*
  - they act on analytical functions
  - `operator()(const Function& f, const Point& v, const Real& time)`
- *discrete operators*
  - they act on discrete mesh functions
  - `operator()(const Function& f, const MeshEntity& entity, const Real& time)`
  - `setMatrixElements( const MeshFunction& f, const MeshEntity& entity, const Real& time, Matrix& matrix )`
- boundary conditions are operators defined on the boundary mesh entities

Laplace operator

```
Real operator()( const EntityType& entity,
                 const MeshFunction& u,
                 const Real& time ) const
{
   auto neighbourEntities = entity.getNeighbourEntities();
   const Mesh& mesh = entity.getMesh();
   Real& hxSquareInverse = mesh.template getSpaceStepsProducts<-2, 0>();
   Real& hySquareInverse = mesh.template getSpaceStepsProducts< 0,-2>();
   Index& east  = neighbourEntities.template getEntityIndex<-1, 0>();
   Index& west  = neighbourEntities.template getEntityIndex< 1, 0>();
   Index& south = neighbourEntities.template getEntityIndex< 0,-1>();
   Index& north = neighbourEntities.template getEntityIndex< 0, 1>();
   Index& center = entity.getIndex();
   return ( u[ east ] + u[ west ] ) * hxSquareInverse +
          ( u[ south ] + u[ north ] ) * hySquareInverse
          - 2.0 * u[ center ] * ( hxSquareInverse + hySquareInverse );
}
```

Laplace operator

```
void setMatrixElements( const EntityType& entity,
                        const MeshFunction& u,
                        const RealType& time,
                        Matrix& matrix ) const
{
    auto matrixRow = matrix.getRow( index );
    Real lambdaX = tau*mesh.template getSpaceStepsProducts< -2, 0 >();
    Real lambdaY = tau*mesh.template getSpaceStepsProducts< 0, -2 >();
    auto neighbourEntities = entity.getNeighbourEntities();
    Index& east  = neighbourEntities.template getEntityIndex<-1, 0>();
    Index& west  = neighbourEntities.template getEntityIndex< 1, 0>();
    Index& south = neighbourEntities.template getEntityIndex< 0,-1>();
    Index& north = neighbourEntities.template getEntityIndex< 0, 1>();
    Index& center = entity.getIndex();
    matrixRow.setElement( 0, south,  -lambdaY );
    matrixRow.setElement( 1, west,   -lambdaX );
    matrixRow.setElement( 2, center, 2.0 * ( lambdaX + lambdaY ) );
    matrixRow.setElement( 3, east,   -lambdaX );
    matrixRow.setElement( 4, north,  -lambdaY );
}
```

Flexibility

- the solver may now run even on GPUs
  - hopefully even other parallel architectures like MPI or MIC
- implementing other schemes (3D, unstructured mesh) = implementing another discrete differential operator

TNL Quickstart

- the user still have to write a lot of code
- TNL offers a tool `tnl-quickstart`
- it generates Makefile and all common files

TNL Quickstart

```
tnl-quickstart
TNL Quickstart -- solver generator
---------------------------------------

Problem name: Heat Equation
Problem class base name (base name acceptable in C++ code): HeatEquation
Operator name: Laplace

ls
HeatEquation.cpp  HeatEquation-cuda.cu  HeatEquation.h
HeatEquationProblem.h  HeatEquationProblem_impl.h
HeatEquationRhs.h  Laplace.h  Laplace_impl.h
Makefile  run-HeatEquation
```

Compile it by

```
make
g++  -I/home/oberhuber/local/include/tnl-0.1 -std=c++11 -DNDEBUG -c -o
 HeatEquation.o HeatEquation.cpp
g++ -o HeatEquation HeatEquation.o -L/home/oberhuber/local/lib -ltnl-0.1
```

or

```
make WITH_CUDA=yes
nvcc  -I/home/oberhuber/local/include/tnl-0.1 -DHAVE_CUDA -DHAVE_NOT_CXX11
-gencode arch=compute_20,code=sm_21 -DNDEBUG -c -o HeatEquation-cuda.o
HeatEquation-cuda.cu
...
nvcc -o HeatEquation HeatEquation-cuda.o -L/home/oberhuber/local/lib -ltnl-0.1
```

It creates executable `HeatEquation`

You may run it with:

```
./HeatEquation
Some mandatory parameters are missing. They are listed at the end.
Usage of: ./HeatEquation


Heat Equation settings:

        --boundary-conditions-type       string    Choose the boundary conditions type.
                                                   - Can be: dirichlet, neumann
                                                   - Default value is: dirichlet
    --boundary-conditions-constant         real    This sets a value in case of the constant boundary conditions.

 === General parameters ===

                         --real-type       string    Precision of the floating point arithmetic.
                                                   - Can be: double
                                                   - Default value is: double
                           --device        string    Device to use for the computations.
                                                   - Can be: host, cuda
                                                   - Default value is: host
                       --index-type        string    Indexing type for arrays, vectors, matrices etc.
                                                   - Can be: int
                                                   - Default value is: int
...
Add the following missing  parameters to the command line:
   --final-time ... --snapshot-period ... --time-discretisation ... --discrete-solver ...
```

Or you may use a generated script:

```
./run-HeatEquation
```

Disadvantages

Disadvantages of C++ templates:

- object interfaces are given implicitly
- it leads to compiler error messages difficult to read
- compilation may take a lot of time

Recent work (summer 2016)

- refactoring
  - introducing namespaces $\Rightarrow$ renaming of all objects in TNL
  - `tnlVector` $\rightarrow$ `TNL::Containers::Vector`
  - `tnlGrid` $\rightarrow$ `TNL::Meshes::Grid`
  - `tnlCSRMatrix` $\rightarrow$ `TNL::Matrices::CSR`
  - `tnlGMRESSolver` $\rightarrow$ `TNL::Solvers::Linear::GMRES`
- switching from CppUnit to GTest

Recent work (summer 2016)

- bug in `nvcc` 7.5
  - reported as a bug number 1786375 *slow pointers initiation in kernel*
  - it seems to be fixed in CUDA 8 RC

Recent work (summer 2016)

- bug in gcc-5.4 and older
  - gcc crashes when we compile TNL with OpenMP support
  - to be reported soon
  - currently we test Clang compiler

```
template< typename Device >
class Grid
{
};


template< typename Grid >
class GridTraverser
{
};

template< typename Device >
class GridTraverser < Grid< Device > >
{
   public:

      typedef Grid< Device > GridType;
      typedef typename GridType::CoordinatesType CoordinatesType;

      template< typename GridEntity >
      static void
      processEntities( const CoordinatesType begin, const CoordinatesType end )
      {
         GridEntity entity;
         {
      #pragma omp parallel for
            for( entity.getCoordinates().y() = begin.y();
                 entity.getCoordinates().y() <= end.y();
                 entity.getCoordinates().y() ++ )
               {
               }
         }
      }
};
```

Results

- solving heat equation in 1D, 2D and 3D on time interval $[0, 1]$
- CPU is Intel Xeon CPU E5-2630 at 2.4-3.2 GHz with 20MB cache
- GPU is Tesla K40 2880 CUDA cores at 0.745 GHz

Results

- 1D results

DOFs	CPU -O0	CPU -O3	Speed-up	GPU	Speed-up
16	0.005s	0.007s	**0.7**	0.6s	**0.001**
32	0.04s	0.027s	**1.5**	0.8s	**0.03**
64	0.33s	0.12s	**2.8**	2.5s	**0.05**
128	2.62s	0.52s	**5**	10.8s	**0.05**
256	20.9s	2.95s	**7**	42.5s	**0.07**
512	2m 37.7s	21.5s	**7.3**	2m 53.0s	**0.12**
1024	22m 13.3s	2m 41.62s	**8.25**	11m 42.0s	**0.23**

Results

- 2D results

DOFs	CPU -O0	CPU -O3	Speed-up	GPU	Speed-up
16^2	0.2s	0.07s	**2.85**	0.4s	**0.17**
32^2	3.8s	0.59s	**6.4**	1.3s	**0.45**
64^2	1m 04.5s	8.16s	**7.9**	5.8s	**1.4**
128^2	17m 33.0s	2m 13.7s	**7.8**	27.8s	**4.84**
256^2	4h 43m 09.0s	36m 08.9s	**7.1**	2m 36.6s	**15.2**

Results

- 3D results

DOFs	CPU -O0	CPU -O3	Speed-up	GPU	Speed-up
16^3	9s	0.96s	**9.3**	4.2s	**0.22**
32^3	5m 33s	30.7s	**10.8**	18.8s	**1.6**
64^3	3h 11m 26s	17m 29.7s	**10.9**	2m 09.8s	**8.16**

Results

- Comparison TNL vs. pure C implementation on CPU
  - heat equation in 2D

DOFs	TNL	Pure C	Speed-up
32^2	0.1s	0.04s	**0.4**
64^2	0.19s	0.12s	**0.63**
128^2	0.41s	0.25s	**0.6**
256^2	1.34s	1.1s	**0.8**
512^2	4.9s	3.7s	**0.75**
1024^2	19s	21.8s	**1.14**

Future work

Other experimental features:

- unstructured meshes - V. Žabka
- mean-curvature flow, complementary finite volumes - O. Székely
- GEM on GPU - J. Kaňuk
- GMRES via Householder tranformations - J. Klinkovský
- parallel fast sweeping method, narrow band method - O. Sobotík
- adaptive grids - L. Bakajsa
- solver for the Euler equations - J. Schafer
- incompressible Navier-Stokes solver - V. Klement
- suppport of Intel Xeon Phi - V. Hanousek

Future plans:

- geometric and algebraic multigrid on GPU - J. Klinkovský
- support of MPI
- FEM, FVM, LBM, IBM

Results

TNL will be released soon under MIT License.