TNL:FDM on GPU in C++

Tomáš Oberhuber Vítězslav Žabka Vladimír Klement
Radek Fučík Jiří Kafka Tomáš Sobotík Ondřej Székely
Libor Bakajsa Jakub Klinkovský Jan Vacata
Martin Heller Matěj Novotný

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

TNL = Template Numerical Library

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

1. **efficient**
   - Why? – numerical simulations take a lot of time
   - How? – we use C++, CUDA
2. **flexible**
   - Why? – to easily switch between different numerical methods
   - How? – we use C++ templates
3. **user friendly**
   - Why? – to be accessible to mathematicians and physicists
   - How? – we hide C++ templates as much as possible

GPUs and MICs

- development of TNL started $\approx$ 2005 $\equiv$ advent of GPGPU
- 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 SIMD-like cores

Therefore,

- the programmer must have good knowledge of the hardware
- porting a code to GPUs and MICs, in fact, 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
   - arrays, vectors, matrices, configuration
2. solvers
   - ODE, linear systems
3. PDE solver
4. future features

Arrays

- arrays are basic structure for memory management
- `tnlArray< ElementType, DeviceType, IndexType >`
  - `ElementType` - type of array elements
  - `DeviceType` - CPU (`tnlHost`) or GPU (`tnlCuda`)
    - memory accesses on CPU and GPU are checked at compile time
  - `IndexType` - indexing type – `int` or `long int`
- there are methods for
  - memory allocation – `setSize`, `setLike`
  - I/O operations – `load`, `save`
  - operators – =, ==, «
  - elements manipulation
    - `getElement`, `setElement` – callable only from host for both `tnlHost`/`tnlCuda`
    - `__cuda_callable__ operator[]` – callable from host for `tnlHost` and from CUDA kernels for `tnlCuda`
  - array bounds are checked by assertions (only in debug mode)

Vectors

`tnlVector< RealType, DeviceType, IndexType >`

- vectors extend arrays with algebraic operations (BLAS)
  - operators – `+=, -=, *=, /=`
  - scalar product – `scalarProduct`
  - vector addition – `addVectors`
  - parallel reduction operations – `lpNorm`, `min`, `max`, ...
  - prefix sum – `prefixSum`
    - $s_i = \sum_{j=0}^{i} a_i, i = 0, 1, 2, \ldots N - 1$

Everything is implemented in CUDA as well.

Shared arrays/vectors

To use TNL methods from non-TNL code, shared arrays/vectors can be used.

- `tnlSharedArray< ElementType,DeviceType,IndexType >`
- `tnlSharedVector< RealType,DeviceType,IndexType >`

```
double data = new[ size ];
...
tnlSharedArray< double,tnlHost,int > s;
s.bind( data, size );
```

Shared arrays and vectors

- shared arrays/vectors also help to organize degrees of freedom of the problem
- assume that we solve incompressible Navier-Stokes equations in 2D
- the unknown variables are u, v, p
- if we approximate them on mesh with n elements, we have $3n$ DOFs $d_1, \ldots d_{3n}$

Shared arrays and vectors

- the mapping might be as follows

$$\begin{array}{ccc} d_1, \ldots, d_n, & d_{n+1}, \ldots, d_{2n}, & d_{2n+1}, \ldots, d_{3n} \\ \downarrow & \downarrow & \downarrow \\ u_1, \ldots, u_n, & v_1, \ldots, v_n, & p_1, \ldots, p_n \end{array}$$

- the data are organized in SoA (*structure of arrays*) manner instead of AoS (*array of structures*)

$$\begin{array}{cccc} d_1, d_2, d_3, & d_4, d_5, d_6, & \ldots & d_{3n-2}, d_{3n-1}, d_{3n} \\ \downarrow & \downarrow & & \downarrow \\ u_1, v_1, p_1, & u_2, v_2, p_2, & \ldots, & u_n, v_n, p_n \end{array}$$

- SoA is better for parallel vector architectures like GPUs and MICs
- in incompressible Navier-Stokes problem
    - AoS $\rightarrow$ Vanka type solvers
        - works efficiently only for certain finite elements and 2D
    - SoA $\rightarrow$ Schur complement methods

Shared arrays and vectors

The code may look as follows:

```
tnlVector< double > d;
tnlSharedVector< double > u,v,p;
d.setSize( 3*n );
u.bind( d, 0, n );
v.bind( d, n, n,);
p.bind( d, 2*n, n );
```

Matrix formats

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

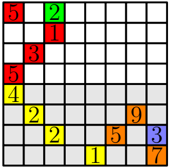
- dense matrix format
- tridiagonal and multidiagonal matrix format
- Ellpack format
- SlicedEllpack format
    - Oberhuber T., Suzuki A., Vacata J., *New Row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA*, Acta Technica, 2011, vol. 56, no. 4, pp. 447-466.
- ChunkedEllpack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

Matrix formats

- most of the matrix formats are developed for fast matrix-vector multiplication
- for non-linear problems, we need to recompute matrices efficiently even on GPUs
- we do it in 3 steps

1. estimate number of non-zero matrix elements in each row – user
2. allocate the matrix and set-up format metadata – TNL
3. set-up the non-zero matrix elements – user

Sliced Ellpack format



values	5	1	3	5	2	0	0	0	4	2	2	1	0	9	5	7	0	0	3	0
columns	0	2	1	0	2	*	*	*	0	1	2	4	*	6	5	7	*	*	7	*

sliceRowLengths	2	3
slicePointers	0	8

1. numbers of non-zero matrix elements in each row (= compressed row lengths)
   - $[2, 1, 1, 1, 1, 2, 3, 2]$
2. setting metadata
   - padding zeros $\rightarrow [2, 2, 2, 2, 3, 3, 3, 3]$
   - numbers of non-zero elements in slices $\rightarrow [8, 12, 0]$
   - exclusive prefix sum $\rightarrow [0, 8, 20]$
   - offsets of the slices $\rightarrow [0, 8]$
   - number of allocated elements $\rightarrow [20]$
3. setting the matrix elements – with help of `MatrixRow`

Setting matrix elements

- most of the matrix formats can directly access matrix rows
- seeking for a matrix element in a row with given column index is less efficient

There are three ways for setting matrix elements:

1. inserting one-by-one by column index – it is slow
2. precompute whole row in a buffer – requires additional memory
   - this can be problem on the GPUs with limited shared memory (64 kB)
3. inserting directly one-by-one by position in compressed row

Matrix rows

- each matrix format has its own supporting type `MatrixRow`
- it helps to directly manipulate the matrix elements ...
- ... even from CUDA kernels

```
void setMatrixElements( const IndexType rowIndex,
                        Matrix& matrix )
{
   typename Matrix::MatrixRow matrixRow =
      matrix.getRow( rowIndex );
   matrixRow.setElement( 0,rowIndex-1, -1.0 );
   matrixRow.setElement( 1,rowIndex,    2.0 );
   matrixRow.setElement( 2,rowIndex+1, -1.0 );
}
```

- local variable `matrixRow` helps to keep a matrix row metadata in fast shared memory of GPU

Grids

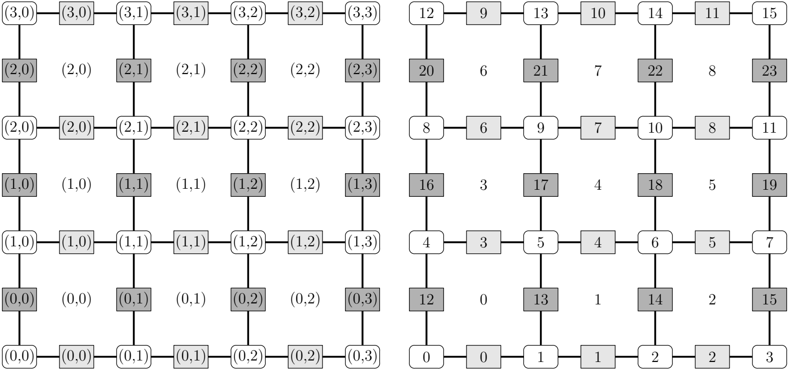
TNL supports 1D, 2D and 3D structured grids:
`tnlGrid< Dimensions, Real, Device, Index >`

- each grid/mesh consists of mesh entities referred by their dimensions
- in 2D
  - cell – 2 dimensions
  - face – 1 dimension
  - vertex – 0 dimensions
- in 3D
  - cell – 3 dimensions
  - face – 2 dimensions
  - edge – 1 dimensions
  - vertex – 0 dimensions

`tnlGrid` provides only indexing, topology and coordinates of the mesh entities. It does not store any DOFs.

Grids

2D grid with 3×3 cells




```
getFaceOfCell< 1, 0 >( CoordinatesType( 1, 1 ) )  =  18
getFaceOfCell< 0, 1 >( CoordinatesType( 1, 1 ) )  =  7
getFaceOfCell< -1, 0 >( CoordinatesType( 1, 1 ) ) =  17
getFaceOfCell< 0, -1 >( CoordinatesType( 1, 1 ) ) =  4
```

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( tnlConfigDescription& 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( tnlParameterContainer& parameters )
{
   double timeStep = parameters.getParameter< double >( "time-step" );
}
```

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) – 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

PDE solver

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

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

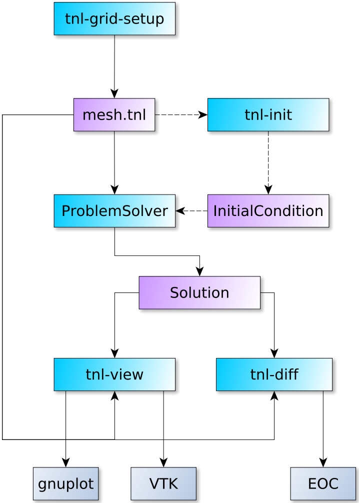
- semi-implicit scheme reads as
  - $$\frac{u_{ij}^{k+1} - u_{ij}^{k}}{\tau} - \frac{1}{h^2}\left(u_{i+1,j}^{k+1} + u_{i-1,j}^{k+1} + u_{i,j+1}^{k+1} + u_{i,j-1}^{k+1} - 4u_{ij}^{k+1}\right) = 0,$$
  - i.e.

    $$\lambda u_{i+1,j}^{k+1} + \lambda u_{i-1,j}^{k+1} + \lambda u_{i,j+1}^{k+1} + \lambda u_{i,j-1}^{k+1} + \left(1 - 4\lambda u_{ij}^{k+1}\right) = u_{ij}^{k}$$
  - which is linear system $\mathbb{A}\mathbf{u} = \mathbf{b}$

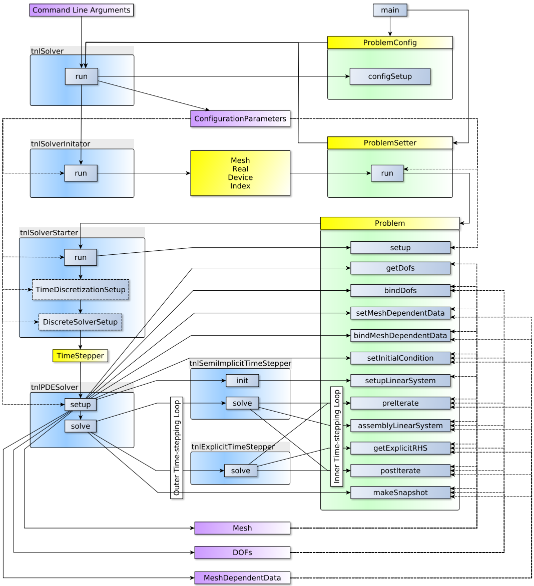
PDE solver

- we need to
  - setup mesh
  - setup initial and boundary conditions
  - allocate DOFs and supporting functions (for non-linear PDEs)
  - 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 $\mathbb{A}\mathbf{u} = \mathbf{b}$
  - perform snapshots of the time dependent solution
- TNL aims to simplify this step by
  - offering some command-line tools
  - implementing a skeleton of PDE solver

TNL tools


tnl-grid-setup
mesh.tnl
tnl-init
ProblemSolver
InitialCondition
Solution
tnl-view
tnl-diff
gnuplot
VTK
EOC

PDE solver



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

Mesh traversers

- the method `Problem::getExplicitRightHandSide` 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++ )
      {
         typename MeshType::CoordinatesType
            cellCoordinates( i, j );
         if( mesh.isBoundaryCell( cellCoordinates ) )
         {
            /****
             * Set boundary conditions
             */
             ...
         }
      }

   for( int i = 0; i < mesh.getDimensions().x(); i++ )
      for( int j = 0; j < mesh.getDimensions().y(); j++ )
      {
         typename MeshType::CoordinatesType
            cellCoordinates( i, j );
         if( ! mesh.isBoundaryCell( cellCoordinates ) )
         {
            /****
             * Approximate the differential operator
             */
            ...
         }
      }
}
```

Mesh traversers

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

Therefore we introduce *mesh traversers*.

Mesh traversers

Mesh traversers are objects for mesh traversing and performing certain operation on mesh entities.

```
tnlExplicitUpdater< Mesh,
                    DofVectorType,
                    DifferentialOperator,
                    BoundaryCondition,
                    RightHandSide >
                       explicitUpdater;
explicitUpdater.template
   update< Mesh::Dimensions >
         ( time,
           mesh,
           this->differentialOperator,
           this->boundaryCondition,
           this->rightHandSide,
           u,
           fu );
```

Mesh traversers

This will

- traverse all mesh entities with `Mesh::Dimensions` dimensions i.e. cells
- for the boundary cells it calls method
  - `__cuda_callable__ setBoundaryConditions` of `this->boundaryConditions`
- for the interior cells it calls method
  - `__cuda_callable__ getValue` of `this->differentialOperator` and `this->rightHandSide` and sum up

Mesh traversers

Assembling of the linear system for (semi-)implicit methods is similar:

```
tnlLinearSystemAssembler< Mesh,
                          DofVectorType,
                          DifferentialOperator,
                          BoundaryCondition,
                          RightHandSide,
                          tnlBackwardTimeDiscretisation,
                          Matrix > systemAssembler;
systemAssembler.template assembly< Mesh::Dimensions >
                                 ( time,
                                   tau,
                                   mesh,
                                   this->differentialOperator,
                                   this->boundaryCondition,
                                   this->rightHandSide,
                                   u,
                                   matrix,
                                   b );
```

Flexibility

- the solver may now run even on GPUs
  - hopefully even other parallel architectures
- adding other schemes (3D, unstructured mesh) = adding template specialization of the differential operator
- adding geometric multigrid might be simple as well

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
    - we need to write good documentation
- C++ templates standard is not perfect
- it leads to compiler error messages difficult to read
- compilation may take a lot of time
    - TNL performs explicit template instantiation during installation
    - one may restrict number of admissible template types for the development builds

Future work

- unstructured meshes (experimental – V. Žabka)
- FEM (V.Žabka), FVM
- support of MPI (V.Hanousek)
- geometric and algebraic multigrid on GPU (V.Klement)
- sparse matrix formats (L.Bakajsa)
- image processing, image import from DICOM (J. Kafka)
- incompressible Navier-Stokes (V.Klement)
- level-set method
  - mean-curvature flow (O.Székely)
  - signed distance function on GPU (T.Sobotík)
- high-precision arithmetic (experimental – M. Novotný)