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

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

T. Oberhuber et al. (FNSPE CTU in Prague) 7/38
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
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
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
ODEs solvers

- Euler, Runge-Kutta-Merson – CPU and GPU

solvers of linear systems

- Krylov subspace methods (CG, BiCGSTab, GMRES, TFQMR) – CPU and GPU

- SOR method – CPU only
TNL offers configuration parameters management
configuration description is done in methods configSetup
one may define configuration parameter
  - type
  - default value
  - required
  - description
  - admissible values

```cpp
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" );
}
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
consider the heat equation as model problem

\[
\frac{\partial u(x, t)}{\partial t} - \Delta u(x, t) = 0 \quad \text{on } \Omega \times (0, T],
\]
\[
u(x, 0) = u_{ini}(x) \quad \text{on } \Omega,
\]
\[
u(x, t) = g(x, t) \quad \text{on } \partial \Omega \times (0, T].
\]

explicit scheme (by method of lines) reads as

\[
\frac{d}{dt} u_{ij}(t) = \frac{1}{h^2} (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}) = F_{ij},
\]

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.

\[
\lambda u_{i+1,j}^{k+1} + \lambda u_{i-1,j}^{k+1} + \lambda u_{i,j+1}^{k+1} + \lambda u_{i,j-1}^{k+1} + (1 - 4\lambda u_{ij}^{k+1}) = u_{ij}^k
\]

which is linear system \( A u^{k+1} = b \)
we need to

- setup mesh
- setup initial and boundary conditions
- allocate DOFs
- setup discrete solver
- evaluate numerical scheme
  - explicitly → explicit update \( \frac{d}{dt} u_{ij}(t) = F_{ij} \; \forall ij \)
  - (semi-)implicitly → assembly linear system \( A u^{k+1} = b \)

- 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

T. Oberhuber et al. (FNSPE CTU in Prague)
Simple implementation of explicit scheme

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

```cpp
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
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 )`
analytical operators
- they act on analytical functions
- \texttt{operator() (const Function\& f, const Point\& v, const Real\& time)}

discrete operators
- they act on discrete mesh functions
- \texttt{operator() (const Function\& f, const MeshEntity\& entity, const Real\& time)}
- \texttt{setMatrixElements( const MeshFunction\& f, const MeshEntity\& entity, const Real\& time, Matrix\& matrix )}

boundary conditions are operators defined on the boundary mesh entities
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 );
}
```cpp
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);
}
```
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
- the user still have to write a lot of code
- TNL offers a tool tnl-quickstart
- it generates Makefile and all common files
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
make WITH_CUDA=yes

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 of C++ templates:

- object interfaces are given implicitly
- it leads to compiler error messages difficult to read
- compilation may take a lot of time
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 \texttt{nvcc} 7.5
  - reported as a bug number 1786375 \textit{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

```cpp
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() ++ )
        {
          ...
        }
      }
    }
};
```
- 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
### 1D results

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

- **2D results**

<table>
<thead>
<tr>
<th>DOFs</th>
<th>CPU -00</th>
<th>CPU -03</th>
<th>Speed-up</th>
<th>GPU</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16^2$</td>
<td>0.2s</td>
<td>0.07s</td>
<td>2.85</td>
<td>0.4s</td>
<td>0.17</td>
</tr>
<tr>
<td>$32^2$</td>
<td>3.8s</td>
<td>0.59s</td>
<td>6.4</td>
<td>1.3s</td>
<td>0.45</td>
</tr>
<tr>
<td>$64^2$</td>
<td>1m 04.5s</td>
<td>8.16s</td>
<td>7.9</td>
<td>5.8s</td>
<td>1.4</td>
</tr>
<tr>
<td>$128^2$</td>
<td>17m 33.0s</td>
<td>2m 13.7s</td>
<td>7.8</td>
<td>27.8s</td>
<td>4.84</td>
</tr>
<tr>
<td>$256^2$</td>
<td>4h 43m 09.0s</td>
<td>36m 08.9s</td>
<td>7.1</td>
<td>2m 36.6s</td>
<td>15.2</td>
</tr>
</tbody>
</table>
Results

- 3D results

<table>
<thead>
<tr>
<th>DOFs</th>
<th>CPU -O0</th>
<th>CPU -O3</th>
<th>Speed-up</th>
<th>GPU</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16^3$</td>
<td>9s</td>
<td>0.96s</td>
<td>9.3</td>
<td>4.2s</td>
<td>0.22</td>
</tr>
<tr>
<td>$32^3$</td>
<td>5m 33s</td>
<td>30.7s</td>
<td>10.8</td>
<td>18.8s</td>
<td>1.6</td>
</tr>
<tr>
<td>$64^3$</td>
<td>3h 11m 26s</td>
<td>17m 29.7s</td>
<td>10.9</td>
<td>2m 09.8s</td>
<td>8.16</td>
</tr>
</tbody>
</table>
- Comparison TNL vs. pure C implementation on CPU
  - heat equation in 2D

<table>
<thead>
<tr>
<th>DOFs</th>
<th>TNL (s)</th>
<th>Pure C (s)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>32^2</td>
<td>0.1</td>
<td>0.04</td>
<td>0.4</td>
</tr>
<tr>
<td>64^2</td>
<td>0.19</td>
<td>0.12</td>
<td>0.63</td>
</tr>
<tr>
<td>128^2</td>
<td>0.41</td>
<td>0.25</td>
<td>0.6</td>
</tr>
<tr>
<td>256^2</td>
<td>1.34</td>
<td>1.1</td>
<td>0.8</td>
</tr>
<tr>
<td>512^2</td>
<td>4.9</td>
<td>3.7</td>
<td>0.75</td>
</tr>
<tr>
<td>1024^2</td>
<td>19</td>
<td>21.8</td>
<td>1.14</td>
</tr>
</tbody>
</table>
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
- support of Intel Xeon Phi - V. Hanousek

Future plans:

- geometric and algebraic multigrid on GPU - J. Klinkovský
- support of MPI
- FEM, FVM, LBM, IBM
TNL will be released soon under MIT License.