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

Algoritmy 2016
**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?

- 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
- `tnlArray< ElementType, DeviceType, IndexType >`
  - DeviceType - CPU (`tnlHost`) or GPU (`tnlCuda`)
  - 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 `tnlHost/tnlCuda`
    - `__cuda_callable__` operator[] – callable from host for `tnlHost` and from CUDA kernels for `tnlCuda`
tnlVector< 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)
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:

\texttt{tnlGrid< Dimensions, Real, Device, Index >}

- it provides indexing and coordinates mapping for the mesh entities:
  - cells
  - faces
  - edges
  - vertices
2D grid with $3 \times 3$ cells
Solvers

- **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
• 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

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

• 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.

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

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

- setup mesh
- setup initial and boundary conditions
- allocate DOFs
- setup discrete solver
- evaluate numerical scheme
  - explicitly \( \rightarrow \) explicit update \( \frac{d}{dt} u_{ij}(t) = F_{ij} \quad \forall i,j \)
  - (semi-)implicitly \( \rightarrow \) assembly linear system \( A u^{k+1} = b \)
- perform snapshots of the time dependent solution

TNL aims to simplify these steps by implementing a skeleton of the PDE solver
PDE solver

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

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
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
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

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
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
- 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>Explicit scheme</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>GPU</td>
<td>Speed-up</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.005s</td>
<td>0.6s</td>
<td>0.0008</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0.04s</td>
<td>0.8s</td>
<td>0.05</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>0.33s</td>
<td>2.5s</td>
<td>0.13</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>2.62s</td>
<td>10.8s</td>
<td>0.24</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>20.9s</td>
<td>42.5s</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>2m 37.7s</td>
<td>2m 53.0s</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>22m 13.3s</td>
<td>11m 42.0 S</td>
<td>1.9</td>
<td></td>
</tr>
</tbody>
</table>
2D results

<table>
<thead>
<tr>
<th>DOFs</th>
<th>Explicit scheme</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>GPU</td>
<td>Speed-up</td>
<td></td>
</tr>
<tr>
<td>16(^2)</td>
<td>0.2 s</td>
<td>0.4 s</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>32(^2)</td>
<td>3.8 s</td>
<td>1.3 s</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>64(^2)</td>
<td>1m 04.5 s</td>
<td>5.8 s</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>128(^2)</td>
<td>17m 33.0 s</td>
<td>27.8 s</td>
<td>37.8</td>
<td></td>
</tr>
<tr>
<td>256(^2)</td>
<td>4h 43m 09.0 s</td>
<td>2m 36.6s</td>
<td>108</td>
<td></td>
</tr>
</tbody>
</table>
### 3D Results

<table>
<thead>
<tr>
<th>DOFs</th>
<th>Explicit scheme</th>
<th>CPU</th>
<th>GPU</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>16(^3)</td>
<td>09s</td>
<td>4.2s</td>
<td>2.1</td>
<td></td>
</tr>
<tr>
<td>32(^3)</td>
<td>5m 33s</td>
<td>18.8s</td>
<td>17.7</td>
<td></td>
</tr>
<tr>
<td>64(^3)</td>
<td>3h 11m 26s</td>
<td>129.8s</td>
<td>89</td>
<td></td>
</tr>
</tbody>
</table>
Future work

- publish TNL on the internet
- unstructured meshes (experimental)
- FEM, FVM
- support of MPI