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

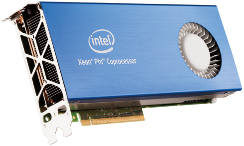
TNL = Template Numerical Library

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.

Outline

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`

Vectors

`tnlVector< 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.

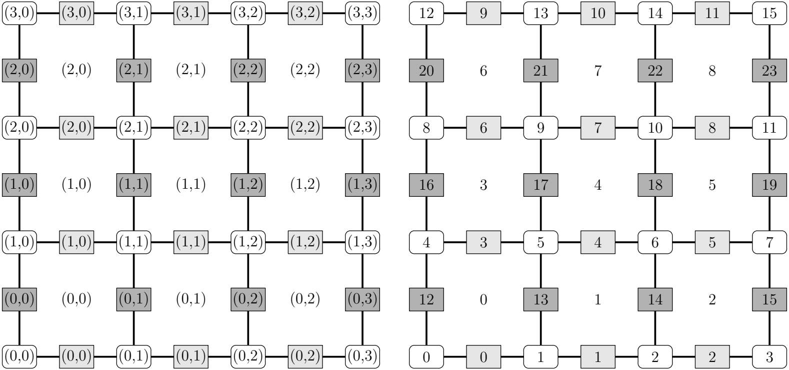
Grids

TNL supports 1D, 2D and 3D structured grids:

`tnlGrid< Dimensions, Real, Device, Index >`

- it provides indexing and coordinates mapping for the mesh entities:
  - cells
  - faces
  - edges
  - vertices

2D grid with 3×3 cells



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

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

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

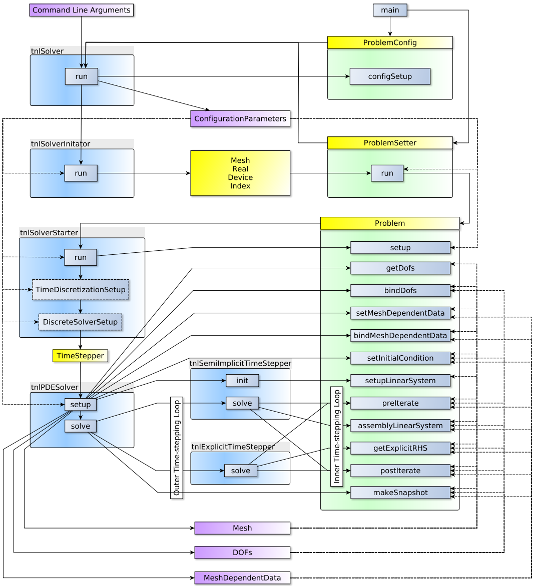
- semi-implicit scheme reads as
  - $$\frac{u_{ij}^{k+1} - u_{ij}^{k}}{\tau} - \frac{1}{h^2}\left(u_{i+1,j}^{k+1} + u_{i-1,j}^{k+1} + u_{i,j+1}^{k+1} + u_{i,j-1}^{k+1} - 4u_{ij}^{k+1}\right) = 0,$$
  - i.e.

    $$\lambda u_{i+1,j}^{k+1} + \lambda u_{i-1,j}^{k+1} + \lambda u_{i,j+1}^{k+1} + \lambda u_{i,j-1}^{k+1} + \left(1 - 4\lambda u_{ij}^{k+1}\right) = u_{ij}^{k}$$
  - 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 implementing a skeleton of the PDE solver

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
tnlPDESolver
setup
solve
tnlSemiImplicitTimeStepper
init
solve
tnlExplicitTimeStepper
solve
Outer Time-stepping Loop
Inner Time-stepping Loop
setup
getDofs
bindDofs
setMeshDependentData
bindMeshDependentData
setInitialCondition
setupLinearSystem
preIterate
assemblyLinearSystem
getExplicitRHS
postIterate
makeSnapshot
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

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

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`

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

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	Explicit scheme		
	CPU	GPU	Speed-up
16	0.005s	0.6 s	**0.0008**
32	0.04s	0.8 s	**0.05**
64	0.33s	2.5 s	**0.13**
128	2.62s	10.8 s	**0.24**
256	20.9s	42.5 s	**0.5**
512	2m 37.7s	2m 53.0 s	**0.9**
1024	22m 13.3s	11m 42.0 S	**1.9**

- 2D results

DOFs	Explicit scheme		
	CPU	GPU	Speed-up
16^2	0.2 s	0.4s	**0.5**
32^2	3.8 s	1.3s	**2.9**
64^2	1m 04.5 s	5.8s	**11**
128^2	17m 33.0 s	27.8s	**37.8**
256^2	4h 43m 09.0 s	2m 36.6s	**108**

Results

- 3D results

DOFs	Explicit scheme		
	CPU	GPU	Speed-up
16^3	09s	4.2s	**2.1**
32^3	5m 33s	18.8s	**17.7**
64^3	3h 11m 26s	129.8s	**89**

Future work

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