TNL: Framework for numerical computing on modern parallel architectures

Tomáš Oberhuber Radek Fučík Jakub Klinkovský
Vítězslav Žabka Vladimír Klement Vít Hanousek

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

Acomen 2017, Ghent

Why GPU?

	Nvidia Tesla P100	Intel Xeon E5-4660
Cores	3584 @ 1.3GHz	16 @ 3.0GHz
Peak perf.	10.6/5.3 TFlops	0.4 / 0.2 TFlops
Max. RAM	16 GB	1.5 TB
Memory bw.	720 GB/s	68 GB/s
TDP	330 W	120 W

$$\approx 8,000\ \$$$

Difficulties in programming GPUs?

Unfortunately,

- the programmer must have good knowledge of the hardware
- porting a code to GPUs often 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 easily accessible.

Template Numerical Library

TNL = Template Numerical Library

- is written in C++ and profits from meta-programming
- provides unified interface to multi-core CPUs and GPUs (via CUDA)
- wants to be user friendly

Outline

1. TNL design
2. Multiphase flow in porous media
3. Performance results

Arrays and vectors

Arrays are basic structures for memory management

- `TNL::Array< ElementType, DeviceType, IndexType >`
- `DeviceType` says where the array resides
  - `TNL::Devices::Host` for CPU
  - `TNL::Devices::Cuda` for GPU
- memory allocation, I/O operations, elements manipulation ...

Vectors extend arrays with algebraic operations

- `TNL::Vector< RealType, DeviceType, IndexType >`
- addition, multiplication, scalar product, l_p norms ...

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

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.

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.

Numerical meshes

Numerical mesh consists of *mesh entities* referred by their dimension:

	Mesh entity dimension			
Mesh dimension	0	1	2	3
1	vertex	cell	–	–
2	vertex	face	cell	–
3	vertex	edge	face	cell

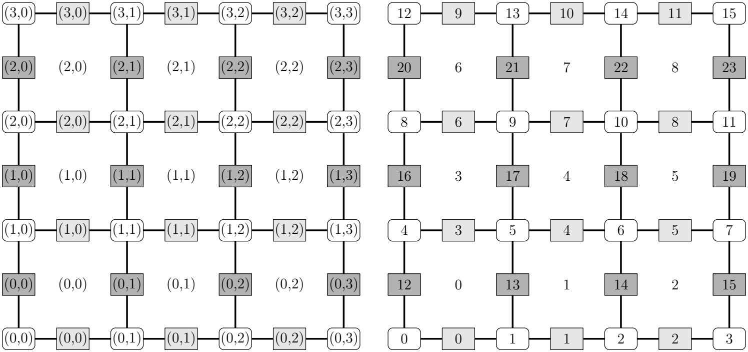
Numerical meshes

TNL supports

- structured orthogonal **grids** – 1D, 2D, 3D
  - mesh entities are generated on the fly
- unstructured **meshes** – nD
  - mesh entities are stored in memory

Structured grids

`TNL::Meshes::Grid< Dimensions,Real,Device,Index >`



Grid provides mapping between coordinates and global indexes.

Unstructured meshes

Unstructured numerical mesh is defined by:

- set of vertexes, cells, faces (and edges)
- coordinates of the vertexes
- each mesh entity may store subentities and superentities
    - see the next slide

The mesh does not store:

- mesh entity volume
- mesh entity normal
- etc.

Unstructured meshes

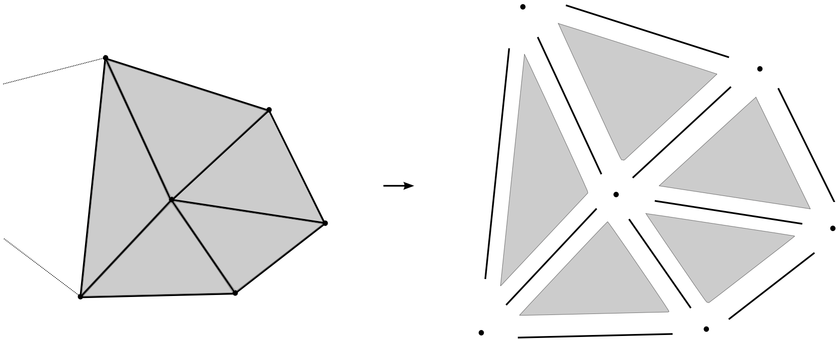
Subentities = mesh entities adjoined to another mesh entity with **higher** dimension

- faces adjoined to cell
- vertexes adjoined to cell
- ...

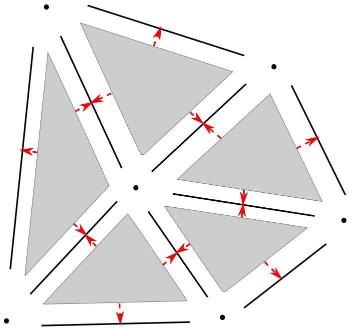
Superentities = mesh entities adjoined to another mesh entity with **lower** dimension

- cells adjoined to vertex
- cells adjoined to face
- faces adjoined to vertex
- ...

Unstructured meshes

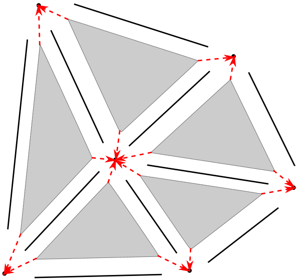


Subentities storage



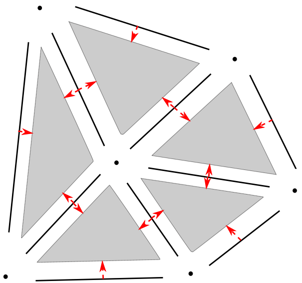
cell		faces
c_1	$\rightarrow$	f_1, f_4, f_3
c_2	$\rightarrow$	f_3, f_6, f_2
c_3	$\rightarrow$	f_5, f_8, f_4
c_4	$\rightarrow$	f_7, f_9, f_6
c_5	$\rightarrow$	f_8, f_{10}, f_7

Subentities storage



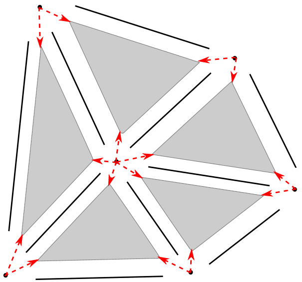
cell		vertexes
c_1	$\rightarrow$	v_2, v_3, v_1
c_2	$\rightarrow$	v_1, v_3, v_5
c_3	$\rightarrow$	v_2, v_4, v_3
c_4	$\rightarrow$	v_3, v_6, v_5
c_5	$\rightarrow$	v_4, v_6, v_3

Superentities storage



face		cells
f_1	$\to$	c_1
f_2	$\to$	c_2
f_3	$\to$	c_1, c_2
f_4	$\to$	c_1, c_3
f_5	$\to$	c_3
f_6	$\to$	c_2, c_4
f_7	$\to$	c_4, c_5
f_8	$\to$	c_3, c_5
f_9	$\to$	c_4
f_{10}	$\to$	c_5

Superentities storage



vertex		cells
v_1	$\rightarrow$	c_1, c_2
v_2	$\rightarrow$	c_1, c_3
v_3	$\rightarrow$	c_1, c_3, c_5, c_4, c_2
v_4	$\rightarrow$	c_3, c_5
v_5	$\rightarrow$	c_2, c_4
v_6	$\rightarrow$	c_4, c_5

Unstructured meshes

`TNL::Meshes::Mesh< MeshConfig, Device >`

- can have arbitrary dimension
- `MeshConfig` controls what mesh entities, subentities and superentities are stored
- it is done in the compile-time thanks to C++ templates

Based on `MeshConfig`, the mesh is fine-tuned for specific numerical method in compile-time.

Mesh traversers

- PDE solvers need to iterate over mesh entities with given dimension
- it is usually done by for-loops, iterators or CUDA kernel calls
- instead, we use *mesh traversers*

Mesh traverser = object for traversing the mesh.

- it has unified interface independent on the device where the mesh is stored
- on specific mesh entities, it performs given action
  - evaluate numerical scheme
  - compute matrix elements of some discrete operator

Functions and operators

Functions

- they are defined on cells, faces, edges or vertexes of a numerical mesh
- `operator()( const MeshEntity& entity, const Real& time)`

Operators

- `operator()(const Function& f, const MeshEntity& entity, const Real& time)`
- `setMatrixElements( const MeshFunction& f, const MeshEntity& entity, const Real& time, Matrix& matrix )`

Why traversers, functions and operators?

The concept of mesh traversers, operators and functions allows to separate HW dependent code (in the traversers) from numerical scheme (in operators and functions).

Solvers

ODEs solvers

- Euler, Runge-Kutta-Merson

Linear systems solvers

- Krylov subspace methods (CG, BiCGSTab, GMRES, TFQMR)

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.

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.

Multiphase flow in porous media

We consider the following system of n partial differential equations in a general coefficient form

$$\sum_{j=1}^{n} N_{i,j} \frac{\partial Z_j}{\partial t} + \sum_{j=1}^{n} \mathbf{u}_{i,j} \cdot \nabla Z_j + \nabla \cdot \left[m_i \left(- \sum_{j=1}^{n} D_{i,j} \nabla Z_j + \mathbf{w}_i \right) + \sum_{j=1}^{n} Z_j \mathbf{a}_{i,j} \right] + \sum_{j=1}^{n} r_{i,j} Z_j = f_i$$

for $i = 1, ..., n$, where the unknown vector function $\vec{Z} = (Z_1, ..., Z_n)^T$ depends on position vector $\vec{x} \in \Omega \subset \mathbb{R}^d$ and time $t \in [0, T]$, $d = 1, 2, 3$.

Multiphase flow in porous media

Initial condition:

$$Z_j(\vec{x},0) = Z_j^{ini}(\vec{x}), \quad \forall \vec{x} \in \Omega,\ j = 1,\dots,n,$$

Boundary conditions:

$$\begin{aligned} Z_j(\vec{x},t) &= Z_j^{\mathcal{D}}(\vec{x},t), \quad \forall \vec{x} \in \Gamma_j^{\mathcal{D}} \subset \partial\Omega,\ \ j = 1,...,n, \\ \vec{v}_i(\vec{x},t) \cdot \vec{n}_{\partial\Omega}(\vec{x}) &= v_i^{\mathcal{N}}(\vec{x},t), \quad \forall \vec{x} \in \Gamma_i^{\mathcal{N}} \subset \partial\Omega,\ \ i = 1,...,n, \end{aligned}$$

where $\vec{v}_i$ denotes the conservative velocity term

$$\vec{v}_i = -\sum_{j=1}^{n} \mathbf{D}_{i,j} \nabla Z_j + \mathbf{w}_i.$$

Numerical method

- Based on the mixed-hybrid finite element method (MHFEM)
  - one global large sparse linear system for traces of $(Z_1, , \dots, Z_n)$ (on faces) per time step
- Semi-implicit time discretization
- General spatial dimension (1D, 2D, 3D)
- Structured and unstructured meshes

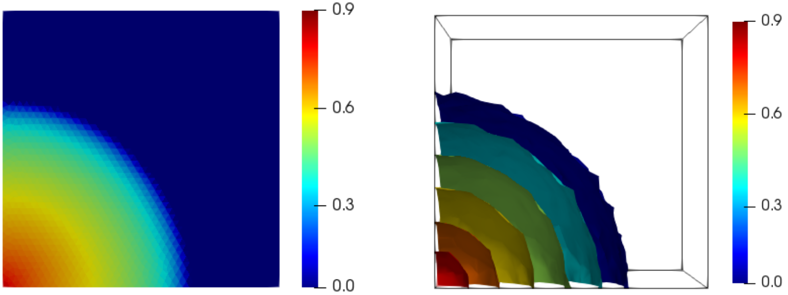
R. Fučík, J.Klinkovský, T. Oberhuber, J. Mikyška, *Multidimensional Mixed–Hybrid Finite Element Method for Compositional Two–Phase Flow in Heterogeneous Porous Media and its Parallel Implementation on GPU*, submitted to Computer Physics Communications.

McWhorter–Sunada problem

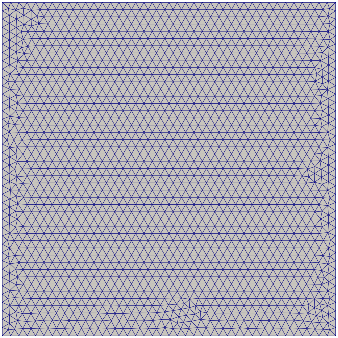
Benchmark problem – generalization of the McWhorter–Sunada problem

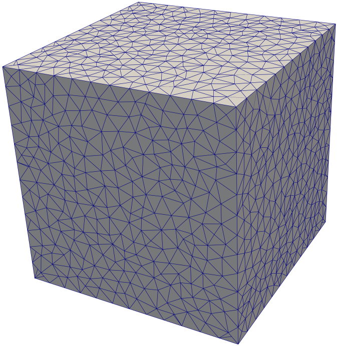
- Two phase flow in porous media
- General dimension (1D, 2D, 3D)
- Radial symmetry
- Point injection in the origin
- Incompressible phases and neglected gravity
- Semi-analytical solution by McWhorter and Sunada (1990) and Fučík *et al.* (2016)

McWhorter–Sunada problem



McWhorter–Sunada problem





McWhorter–Sunada problem

Numerical simulations were performed on:

- 6-core CPU Intel i7-5820K at 3.3 GHz with 15 MB cache
- GPU Tesla K40 with 2880 CUDA cores at 0.745 GHz

McWhorter–Sunada problem 2D

	GPU	CPU										
		1 core		2 cores			4 cores			6 cores		
DOFs	*CT*	*CT*	*GSp*	*CT*	*Eff*	*GSp*	*CT*	*Eff*	*GSp*	*CT*	*Eff*	*GSp*
Orthogonal grids												
960	1,5	0,7	**0,45**	0,4	0,79	**0,28**	0,3	0,52	**0,22**	0,3	0,41	**0,18**
3 720	11,0	13,2	**1,20**	7,6	0,87	**0,69**	4,8	0,68	**0,44**	4,0	0,55	**0,37**
14 640	46,3	197,0	**4,25**	107,5	0,92	**2,32**	65,7	0,75	**1,42**	52,6	0,62	**1,14**
58 080	380,0	4 325,7	**11,38**	2 360,6	0,92	**6,21**	1 448,1	0,75	**3,81**	1 195,8	0,60	**3,15**
231 360	4 449,9	91 166,3	**20,49**	49 004,3	0,93	**11,01**	29 182,1	0,78	**6,56**	24 684,0	0,62	**5,55**
Unstructured meshes												
766	1,5	0,4	**0,27**	0,3	0,60	**0,22**	0,2	0,45	**0,15**	0,2	0,32	**0,14**
2 912	8,9	6,2	**0,70**	3,7	0,84	**0,42**	2,3	0,66	**0,26**	2,0	0,52	**0,23**
11 302	51,1	122,0	**2,39**	67,7	0,90	**1,32**	40,3	0,76	**0,79**	32,5	0,63	**0,64**
44 684	396,1	2 695,6	**6,80**	1 480,7	0,91	**3,74**	855,2	0,79	**2,16**	671,7	0,67	**1,70**
178 648	4 008,3	57 404,2	**14,32**	32 100,5	0,89	**8,01**	18 814,1	0,76	**4,69**	16 414,0	0,58	**4,09**

McWhorter–Sunada problem 3D

	GPU	CPU										
		1 core		2 cores			4 cores			6 cores		
DOFs	*CT*	*CT*	*GSp*	*CT*	*Eff*	*GSp*	*CT*	*Eff*	*GSp*	*CT*	*Eff*	*GSp*
Orthogonal grids												
21 600	2,1	15,2	**7,30**	8,0	0,96	**3,82**	4,4	0,86	**2,13**	3,4	0,75	**1,62**
167 400	30,8	564,3	**18,33**	319,5	0,88	**10,38**	186,7	0,76	**6,07**	150,3	0,63	**4,88**
1 317 600	828,0	20 569,5	**24,84**	12 406,1	0,83	**14,98**	7 092,6	0,73	**8,57**	5 533,7	0,62	**6,68**
10 454 400	31 805,6	(not computed on 1, 2 and 4 cores)								234 066,0		7,36
Unstructured meshes												
5 874	1,4	2,0	**1,48**	1,2	0,85	**0,88**	0,7	0,68	**0,54**	0,6	0,54	**0,46**
15 546	2,6	8,7	**3,30**	4,9	0,89	**1,85**	2,9	0,75	**1,10**	2,3	0,64	**0,86**
121 678	23,9	330,9	**13,87**	184,8	0,90	**7,75**	107,9	0,77	**4,53**	93,4	0,59	**3,92**
973 750	566,2	12 069,5	**21,32**	6 506,3	0,93	**11,49**	3 771,0	0,80	**6,66**	3 306,2	0,61	**5,84**
7 807 218	37 695,3	(not computed on CPU)										

Conclusion

We have presented:

- data structures and solvers in TNL
- MHFEM method for multiphase flow in porous media on GPU
- speed-up on the GPU is up to 7

Future work

Experimental features:

- unstructured meshes
- support of Intel Xeon Phi and distributed clusters using MPI

Future plans:

- support of clusters with GPUs
- geometric and algebraic multigrid
- FEM, FVM, LBM

More about TNL ...

TNL is available at

`www.tnl-project.org`

under MIT license.

Oberhuber T., Klinkovský J., Žabka V., Klement V., Fučík R., *TNL: Framework for rapid development of numerical solvers for modern parallel architectures*, submitted to Computer Physics Communications.