Programming GPUs with TNL

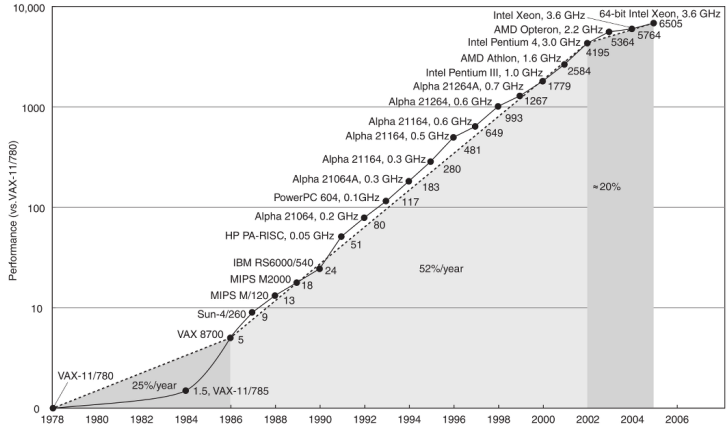
T. Oberhuber[1], J. Klinkovský[1], T. Halada[2], R. Fučík[1], L. Čejka[1], I. Kolesnik[3]

[1]Faculty of Nuclear Science and Physical Engineerings, Czech Technical University in Prague,
[2]Faculty of Mechanical Engineering, Czech Technical University in Prague,
[3]Faculty of Information Technology, Czech Technical University in Prague.

Hora Informaticae
UTIA 2023

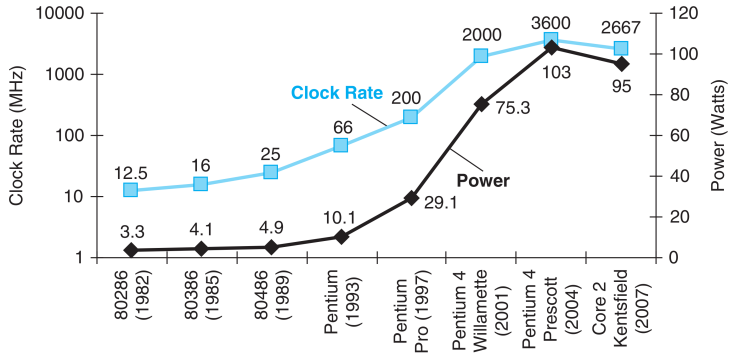
Overview

History of CPU performance



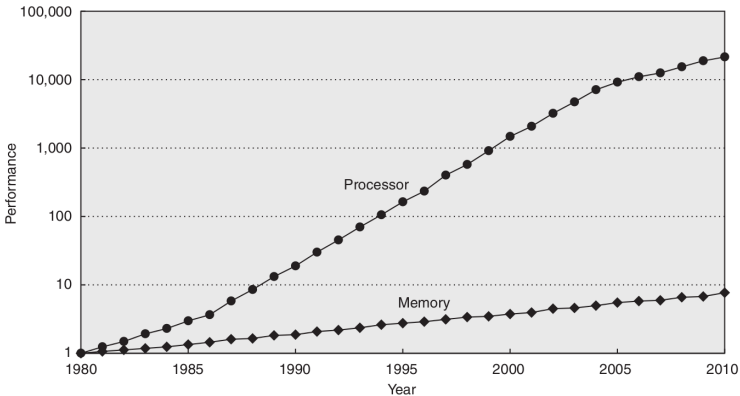
Henessy, Patterson, *Computer architecture: A quantitative approach*, Morgan Kaufmann, 2011.

Power wall



Henessy, Patterson, *Computer architecture: A quantitative approach*, Morgan Kaufmann, 2011.

Memory wall



Henessy, Patterson, *Computer architecture: A quantitative approach*, Morgan Kaufmann, 2011.

GPU vs CPU

	Nvidia Hopper H100	**AMD Epyc 9004 Genoa 96 Cores**
Transistors count	80 billions	90 billions
Frequency	1.78 GHz	2.4-3.7 GHz
Cores	16896	96
Peak Performance	60 TFlops	$\approx$ 2.8 TFlops
Bandwidth	3 TB/s	460 GB/s
RAM	80 GB	12$\times$6 TB
Energy Consumption	700 W	360 W

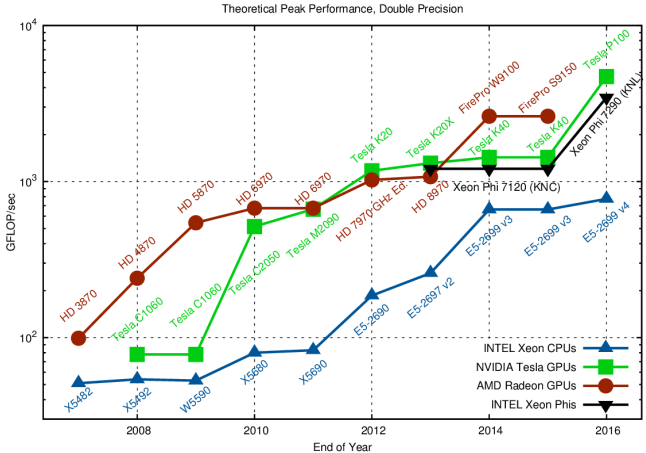


Figure: Performance comparison between GPU and CPU - source Karl Rupp.

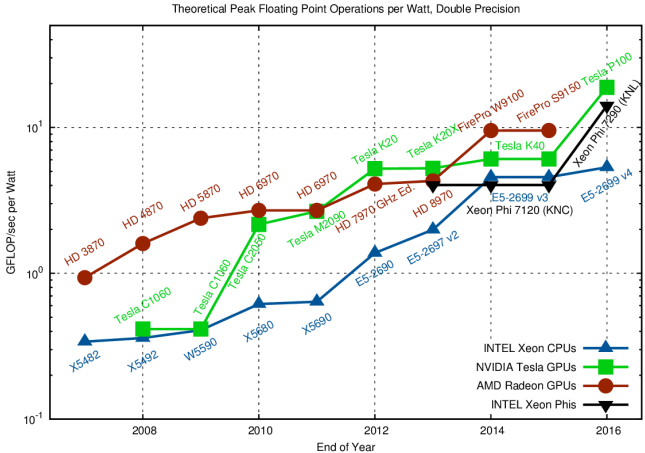


Figure: Performance comparison between GPU and CPU - source Karl Rupp.

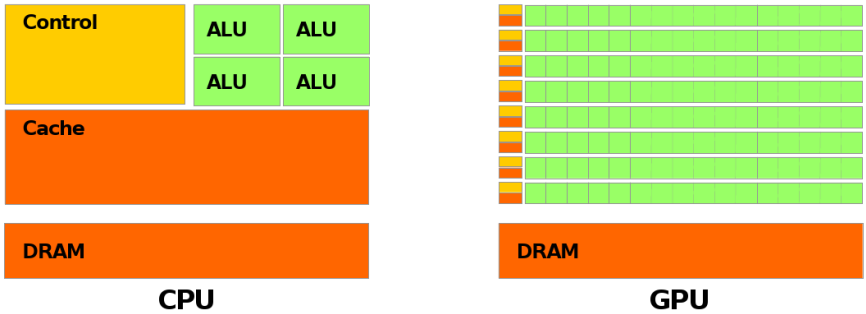


Figure: Comparison between GPU and CPU - source Nvidia.

CUDA = Compute Unified Device Architecture - Nvidia 15 February 2007

- significantly simplified programming of GPUs,
- completely avoided necessity of use of OpenGL,
- it is based on simple extension of C/C++,
- it works only with GPUs by Nvidia.

ROCM and HIP = similar framework for GPUs by AMD.

It is simple to write a program in CUDA but one needs very good knowledge of the architecture to write efficient code.

Programming of GPUs is difficult

1. GPU has its own memory.
2. Connection between CPU and GPU is very slow.
3. GPU contains independent multiprocessors.

Example: Summation of a sequence of numbers

```
1  float sum( 0.0 )
2  for( int i = 0; i < size; i++ )
3  sum += a[ i ];
```

Programming of GPUs is difficult

```
template <unsigned int blockSize>
__global__ void cudaBlockReduction(int size, int* dInput, int* dOutput, )
{
    extern __shared__ volatile int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
    unsigned int gridSize = blockSize*gridDim.x;
    sdata[tid] = 0;
    while (gid < size)
    {
        sdata[tid] += dInput[gid] + dInput[gid+blockSize];
        gid += gridSize;
    }
    __syncthreads();

    if (blockSize >= 10240) {
        if( tid<5120 ) { sdata[ tid ] += sdata[ tid+5120 ]; } __syncthreads(); }
    if (blockSize >= 5120) {
        if( tid<2560 ) { sdata[ tid ] += sdata[ tid+2560 ]; } __syncthreads(); }
    if (blockSize >= 2560) {
        if( tid<1280 ) { sdata[ tid ] += sdata[ tid+1280 ]; } __syncthreads(); }
    if (blockSize >= 1280) {
        if( tid<640 ) { sdata[ tid ] += sdata[ tid+640 ]; } __syncthreads(); }
    if (tid < 32) {
            if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
            if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
            if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
            if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
            if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
            if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
    }

    if (tid == 0) dOutput[blockIdx.x] = sdata[0];
}

bool cudaGridReduction( const int size,
const int deviceData,
const int blockSize,
const int deviceGridSize,
int& result )
{
    const int elementsInBlock = 2 * blockSize;
    int gridSize = size / elementsInBlock + ( size % elementsInBlock != 0 );
    if ( gridSize > deviceGridSize ) gridSize = deviceGridSize;
    int sharedMemSize = gridSize * sizeof( int );
    int* deviceSum1, *deviceSum2;
    if( cudaMalloc( (void**) &deviceSum1, sharedMemSize ) != cudaSuccess ||
    cudaMalloc( (void**) &deviceSum2, sharedMemSize ) != cudaSuccess )
    {
        cerr << "Not enough memory on device." << endl;
        return false;
    }

    int currentSize = size;
    const int* reductionInput = deviceData;
    while( currentSize > 1 )
    {
        dim3 cudaBlockFour( 2, cudaBlockSize( 0 ), 1 );
        dim3 cudaGridOne( 1, 1, 1 );
        gridSize = currentSize / elementsInBlock +
        ( currentSize % elementsInBlock != 0 );
        cudaBlockSize = ( gridSize < deviceGridSize ) gridSize : deviceGridSize;

        size_t sharedMemory = blockSize * sizeof( int );
        switch( blockSize )
        {
                case 2048:
                    cudaGridReduction<2048> <<
                    <<< cudaGridOne, cudaBlockSize, sharedMemory >>>
                    ( currentSize, reductionInput, deviceSum1 );
                    break;
                    break;

        }
        cudaError error = cudaGetLastError();
        if( error != cudaSuccess )
        {
                cerr << "The CUDA kernel ended with error code " << error << "."
                     << endl;
                return false;
        }
        currentSize = cudaGridSize. x;

        reductionInput = deviceSum1;
        int* tmp = deviceSum1;
        deviceSum1 = deviceSum2;
        deviceSum2 = tmp;
    }
    if( cudaMemcpy( &result,
    reductionInput,
    sizeof( int ),
    cudaMemcpyDeviceToHost ) != cudaSuccess )
    {
        cerr << "An error occured while transferring data from device to host."
             << endl;
        return false;
    }
    cudaFree( deviceSum1 );
    cudaFree( deviceSum2 );
    return true;
}
```

Programming of GPUs is difficult

- Lack of support in numerical libraries for numerical simulations:
  - especially software for FEM or FVM,
  - porting of older libraries is impossible.
- Much better situation in AI:
  - DNN are based mainly on matrix-matrix multiplication,
  - libraries for DNN appeared after $\approx$ 2010 and they were designed for GPUs.

TNL = Template Numerical Library

- numerical library for modern parallel architectures
- offers unified interface for both multi-core CPUs and GPUs
- written in C++ profiting from features of C++17
- $\approx$ 340k lines of code and documentation
- **www.tnl-project.org**
- MIT license
- affiliated project of Numfocus (www.numfocus.org)

NUMFOCUS [AFFILIATED PROJECT]

TNL TEMPLATE NUMERICAL LIBRARY

The library can be downloaded using `git` as follows:

```
git clone gitlab@mmg-gitlab.fjfi.cvut.cz:tnl/tnl-dev.git tnl-dev
```

TNL is header-only library and so installation is very fast:

```
cd tnl-dev
./install
```

This installs TNL into `${HOME}/.local/include`.

Memory management:

- **CPU** has its own system memory,
- **GPU** has its own global memory,
- both are connected by slow PCI Express bus,
- programmer must carefully **explicitly** distinct the two address spaces.

Execution of parallel kernels:

- passing all arguments on GPU,
- scheduling of thousands of CUDA threads.

Memory management in TNL is done mainly by arrays and vectors.

Vectors are represented by templated class `Vector`:

```
template< typename Real = double,
          typename Device = TNL::Devices::Host,
          typename Index = int >
class TNL::Containers::Vector { ... };
```

where

- `Real` is type of elements stored in the vector
- `Device` says where the vector will be allocated
  - `TNL::Devices::Host` for CPU and system memory
  - `TNL::Devices::Cuda` for CUDA GPU and global memory
- `Index` is type for indexing the elements in the vector

```
#include <TNL/Containers/Vector.h>

using namespace TNL;
using namespace TNL::Containers;

int main( int argc, char* argv[] )
{
        Vector< int > host_vector( 10 );             // vector with 10 elements on CPU
        Vector< int, Devices::Cuda > device_vector;  // empty vector on GPU

        host_vector = 3;                             // set all elements to 3
        device_vector = host_vector;                 // copy the host vector on GPU

        std::cout << "host_vector = " << host_vector << std::endl;
        std::cout << "device_vector = " << device_vector << std::endl;
        std::cout << std::endl;
```

```
        std::list< int > list { 1, 2, 3, 4, 5 };
        std::vector< int > vector { 6, 7, 8, 9, 10 };

        Vector< int, Devices::Cuda > device_vector_list( list );
        Vector< int, Devices::Cuda > device_vector_vector( vector );
        Vector< int, Devices::Cuda > device_vector_init_list{ 11, 12, 13, 14, 15 };

        std::cout << "device_vector_list = " << device_vector_list << std::endl;
        std::cout << "device_vector_vector = " << device_vector_vector << std::endl;
        std::cout << "device_vector_init_list = " << device_vector_init_list << std::endl;
}
```

The result looks as follows:

```
        host_vector = [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ]
        device_vector = [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ]

        device_vector_list = [ 1, 2, 3, 4, 5 ]
        device_vector_vector = [ 6, 7, 8, 9, 10 ]
        device_vector_init_list = [ 11, 12, 13, 14, 15 ]
```

TNL supports **expression templates** for algebraic vector expressions.
The following vector expression

$$\vec{x} = \vec{a} + 2\vec{b} + 3\vec{c}$$

can be evaluated with Blas/Cublas as follows:

```
1  cublasHandle_t c_h;
2  cublasSaxpy(c_h,N,1.0,a,1,x,1); // -> x = a
3  cublasSaxpy(c_h,N,2.0,b,1,x,1); // -> x = x + 2 * b
4  cublasSaxpy(c_h,N,3.0,c,1,x,1); // -> x = x + 3 * c
```

And using the **expression templates (ET)** in TNL as follows:

```
1  x = a + 2 * b + 3 * c;
```

The later is **more efficient**.

- ET is a proxy object for vector expression
- it allows to do **lazy evaluation of the expression**
- it can evaluate $i-$th element of the expression on-the-fly, i.e.

```
1  ET[ i ] = a[ i ] + 2 * b[ i ] + 3 * c[ i ]
```

- on CPU, the assignment would be evaluated as follows

```
1  for( int i = 0; i < n; i++ )
2  x[ i ] = a[ i ] + 2 * b[ i ] + 3 * c[ i ];
3  //       ^       = ET[ i ]              ^
```

- there is only **one write** to $\vec{x}$ instead of **3 writes and 2 reads** in case of Blas

Expression templates for vectors

Vector addition: `x += a + b + c`.

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	19.3	41.5	**2.2**	194.7	236.5	**1.21**
200k	19.7	41.7	**2.1**	228.3	277.6	**1.21**
400k	17.3	35.9	**2.1**	218.3	330.9	**1.51**
800k	11.7	19.3	**1.6**	233.3	370.6	**1.58**
1.6M	10.4	17.0	**1.6**	249.6	403.4	**1.61**
3.2M	10.2	17.3	**1.7**	266.6	444.8	**1.66**
6.4M	10.2	17.3	**1.7**	276.6	471.3	**1.70**

BW = effective memory bandwidth in GB/s,
tested on GPU Nvidia P100 (16 GB HBM2 @ 732 GB/s, 3584 CUDA cores)
and Intel Core i7-5820K (3.3GHz, 16MB cache).

We setup the following vectors:

```
using Vector = TNL::Containers::Vector< float, TNL::Devices::Cuda >;
Vector a{  8, 4, 2, 0, -2,  -4,  -8 };
Vector b{  0, 2, 4, 6,  8,  10,  12 };
```

We may search for maximum and minimum:

```
cout << "min( a ) = " << min( a ) << endl
     << "max( a ) = " << max( a ) << endl;
```

The result looks as follows:

```
min( a ) = -8;
max( a ) = 8
```

We may combine vector operations with ET:

```
cout << "abs( a ) = << abs( a ) << endl
     << "min( abs( a ) ) = " << min( abs( a ) ) << endl
     << "max( abs( a ) ) = " << max( abs( a ) ) << endl;
```

The result looks as follows:

```
abs( a ) = [ 8, 4, 2, 0, 2, 4, 8 ]
min( abs( a ) ) = 0
max( abs( a ) ) = 8
```

We may compute minimum and maximum componentwise:

```
cout << "min( a, b ) = " << min( a, b ) ) << endl
     << "max( a, b ) = " << max( a, b ) ) << endl
     << "min( max( a, b ) ) = " << min( max( a, b ) ) << endl
     << "max( abs( a ), b ) = " << max( abs( a ), b ) << endl;
```

The result looks as follows:

```
min( a, b ) = [ 0, 2, 2, 0, -2, -4, -8 ]
max( a, b ) = [ 8, 4, 4, 6, 8, 10, 12 ]
min( max( a, b ) ) = -8
max( abs( a ), b ) = [ 8, 4, 4, 6, 8, 10, 12 ]
```

We may locate minimal and maximal elements:

```
auto arg_min_a = argMin( a );      // -> std::pair< float, int >
auto arg_max_a = argMax( a + b ); // -> std::pair< float, int >
cout << "min( a ) = " << arg_min_a.first << " at " << arg_min_a.second << endl
     << "max( a + b ) = " << arg_max_a.first << " at " << arg_max_a.second << endl;
```

The result looks as follows:

```
min( a ) = -8 at 6
max( a + b ) = 8 at 0
```

We may perform componentwise operations:

```
cout << "a + b = " << a + b << endl
     << "a - b = " << a - b << endl
     << "a * b = " << a * b << endl
     << "a / b = " << a / b << endl;
```

The result looks as follows:

```
a + b = [ 8, 6, 6, 6, 6, 6, 4 ]
a - b = [ 8, 2, -2, -6, -10, -14, -20 ]
a * b = [ 0, 8, 8, 0, -16, -40, -96 ]
a / b = [ inf, 2, 0.5, 0, -0.25, -0.4, -0.666667 ]
```

We may compute norms and scalar products:

```
cout << "l2Norm( a - b ) = " << l2Norm( a - b ) << endl
     << "l2Norm( a + 3 * sin( b ) ) = " << l2Norm( a + 3 * sin( b ) ) << endl
     << "Scalar product: ( a, b ) =  " << ( a, b ) << endl
     << "Scalar product: ( a + 3, abs( b ) / 2 ) =  " << ( a + 3, abs( b ) / 2 )
     << endl;
```

The result looks as follows:

```
l2Norm( a - b ) = 28.3549
l2Norm( a + 3 * sin( b ) ) = 15.3312
Scalar product: ( a, b ) =  -136
Scalar product: ( a + 3, abs( b ) / 2 ) =  -5
```

More complex operations can be done with **lambda functions**.

```
using Vector = TNL::Containers::Vector< float, TNL::Devices::Cuda >;
Vector a( 6, 1.0 );
cout << "a = " << a << endl;

auto f = [] __cuda_callable__ ( int idx, float& value ) { // This all is
        value += 0.5 * idx;                                  // turned into
};                                                        // a CUDA kernel
a.forAllElements( f );                                    // by C++ compiler.

cout << "a = " << a << endl;
```

The result looks as:

```
a = [ 1, 1, 1, 1, 1, 1 ]
a = [ 1, 1.5, 2, 2.5, 3, 3.5 ]
```

The lambda functions can **capture** variables from the surrounding code:

```
1  using Vector = TNL::Containers::Vector< float, TNL::Devices::Cuda >;
2  Vector a( 6, 1.0 );
3  cout << "a = " << a << endl;
4
5  const float h = 0.5;   // capture this variable h
6  auto f = [=] __cuda_callable__ ( int idx, float& value ) {
7          value += h * idx;  // use it here inside the lambda function
8  };
9  a.forAllElements( f );
10 cout << "a = " << a << endl;
```

We do not have to copy all necessary variables on GPU explicitly.

Reduction is an operation that takes all array/vector elements as input and returns one value as output:

- array comparison
- scalar product
- l_p norm
- minimal/maximal value
- sum of all elements

Take a look at scalar product:

```
float result( 0.0 );
for( int i = 0; i < size; i++ )
result += a[ i ] * b[ i ];
```

Let us rewrite it using C++ lambda functions as:

```
auto fetch = [=] __cuda_callable__ (int i)->float { return a[i]*b[i]; };
auto reduction = [] __cuda_callable__ (float x, float y) -> float { return x+y; };

float result( 0.0 );
for( int i = 0; i < size; i++ )
reduction = reduction( result, fetch( i ) );
```

In TNL, the for-loop is replaced with call of `reduce` function:

```
1  auto a_view = a.getView();
2  auto b_view = b.getView();
3  auto fetch = [=] __cuda_callable__ ( int i)->float {
4          return a_view[ i ] * b_view[ i ]; };
5  auto reduction = [] __cuda_callable__ (float x, float y) -> float { return x + y; };
6
7  result = TNL::Algorithms::reduce< Device >( 0, a.getSize(), fetch, reduction, 0.0 );
```

The last parameter is the identity element for given reduction operation.

Comparison of two vectors can be evaluated as follows:

```
1  auto a_view = a.getView();
2  auto b_view = b.getView();
3  auto fetch = [=] __cuda_callable__ (int i)->bool {
4          return a_view[ i ] == b_view[ i ]; };
5  auto reduction = [] __cuda_callable__ (float x, float y)->float { return x && y; };
6
7  result = TNL::Algorithms::reduce< Device >( 0, a.getSize(), fetch, reduction, true );
```

Largest element in absolute value can be evaluated as follows:

```
auto a_view = a.getView();
auto fetch = [=] __cuda_callable__ (int i)->float { return abs( a_view[ i ] ); };
auto reduction = [] __cuda_callable__ (float x, float y)->float {
        return max( x, y ); };

result = TNL::Algorithms::reduce< Device >( 0, a.getSize(), fetch, reduction,
std::numeric_limits< float >::lowest() );
```

Or more compact way:

```
auto a_view = a.getView();
result = TNL::Algorithms::reduce< Device >( 0, a.getSize(),
    [=] __cuda_callable__ (int i)->float { return abs( a_view[ i ] ); },
    TNL::Max() );
```

Flexible parallel reduction

We can merge two operations together:

- update/addition of a vector
- computation of the norm of the update

It appears often in Runge-Kutta solvers.

```
1  auto a_view = a.getView();
2  auto b_view = b.getView();
3  result = TNL::Algorithms::reduce< Device >( 0, a.getSize(),
4     [=] __cuda_callable__ (int i)->float mutable {
5              // mutable because we change a_view
6              float update = 0.5 * ( a_view[ i ] + b_view[ i ];
7              a_view[ i ] += update;
8              return abs( update ); },
9      TNL::Plus() );
```

TNL supports several types of matrices:

- dense matrices,
- tridiagonal matrices,
- multidiagonal or band matrices,
- lambda matrices for matrix-free methods,
  - matrix elements are not stored explicitly in memory
  - they are given by lambda functions
- **sparse matrices**.

Sparse matrix is defined as follows:

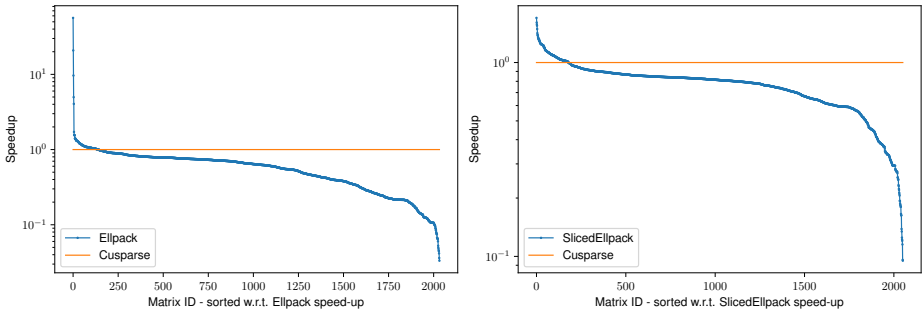
```
template<
    typename Real = double,
    typename Device = TNL::Devices::Host,
    typename Index = int,
    typename MatrixType =
        TNL::Matrices::GeneralMatrix,
    typename Format =
        TNL::Algorithms::Segments::CSR>
class TNL::Matrices::SparseMatrix;
```

Matrices offer flexible interface based on C++ lambda functions for matrix operations.

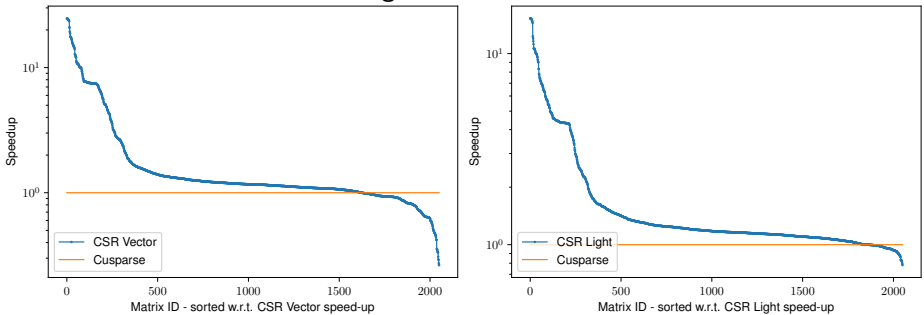
- matrix type can be
  - general matrix
  - symmetric matrix
  - binary matrix
- several sparse-matrix formats are available
  - Ellpack
  - Sliced Ellpack
  - Chunked Ellpack
  - Bisection Ellpack
  - CSR - scalar, vector, light
  - Adaptive CSR

	CPU	GPU
Model	Intel Xeon Gold 6146	Nvidia Tesla V100
Cores	12	5120
Core frequency	3.2-4.2 GHz	1.455 GHz
L2 cache size	12×1 MiB	6 MiB
L3 cache size	24.75 MiB	—
Amount of memory	384 GiB	32 GiB
Peak memory throughput	6×21 GB/s	900 GB/s

Speed-up comparison of **Ellpack** and **Sliced Ellpack** compared to Cusparse in logarithmic scale.



Speed-up comparison of **CSR Vector** and **CSR Light** compared to Cusparse in logarithmic scale.



TNL offers N-dimensional array with compile-time definition of element ordering:

```
NDArray< int,                                   // value type
         SizesHolder< int, 0, 0, 0 >,           // all sizes are set dynamicaly
         std::index_sequence< 2, 1, 0 >,        // first index changes the fastest,
                                                // last index changes the slowest
         TNL::Devices::Host > array_3d;
```

```
NDArray< int,                                   // value type
         SizesHolder< int, 0, 0, 3 >,           // first index is staticaly set to 3
         std::index_sequence< 0, 1, 2 >,        // first index changes the slowest,
                                                // last index changes the fastest
         TNL::Devices::Host > array_3d;
```

This can be used for computations on Cartesian grids.

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

- can have arbitrary dimension
- `MeshConfig` controls what mesh entities and links between them 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.

J. Klinkovský, T. Oberhuber, R. Fučík, V. Žabka, *Configurable open-source data structure for distributed conforming unstructured homogeneous meshes with GPU support*, ACM Transactions on Mathematical Software, vol. 48, no. 3., art. no. 3, 2022.

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:

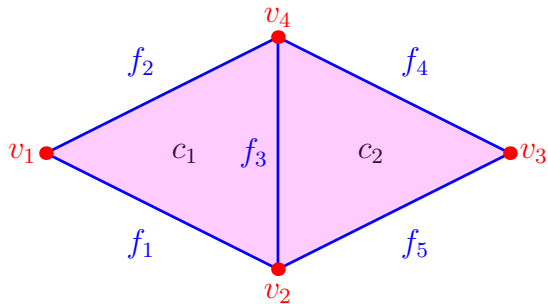
- mesh entity volume
- mesh entity normal
- etc.

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
- ...



$$I_{0,1} = \left(\begin{array}{c|ccccc} & f_1 & f_2 & f_3 & f_4 & f_5 \\ \hline v_1 & 1 & 1 & & & \\ v_2 & 1 & & 1 & & 1 \\ v_3 & & & & 1 & 1 \\ v_4 & & 1 & 1 & 1 & \end{array}\right) \qquad I_{0,2} = \left(\begin{array}{c|cc} & c_1 & c_2 \\ \hline v_1 & 1 & \\ v_2 & 1 & 1 \\ v_3 & & 1 \\ v_4 & 1 & 1 \end{array}\right)$$

Id.	h [m]	$\#\mathcal{V}$	$\#\mathcal{F}$	$\#\mathcal{C}$	N_{dof}
$2D_1^{\triangle}$	$6{,}71 \cdot 10^{-2}$	142	383	242	766
$2D_2^{\triangle}$	$3{,}49 \cdot 10^{-2}$	513	1 456	944	2 912
$2D_3^{\triangle}$	$1{,}64 \cdot 10^{-2}$	1 938	5 651	3 714	11 302
$2D_4^{\triangle}$	$8{,}73 \cdot 10^{-3}$	7 555	22 342	14 788	44 684
$2D_5^{\triangle}$	$4{,}23 \cdot 10^{-3}$	29 989	89 324	59 336	178 648
$3D_1^{\triangle}$	$2{,}13 \cdot 10^{-1}$	395	2 937	1 312	5 874
$3D_2^{\triangle}$	$1{,}27 \cdot 10^{-1}$	824	7 773	3 697	15 546
$3D_3^{\triangle}$	$6{,}29 \cdot 10^{-2}$	5 740	60 839	29 673	121 678
$3D_4^{\triangle}$	$3{,}48 \cdot 10^{-2}$	43 293	486 875	240 372	973 750
$3D_5^{\triangle}$	$1{,}84 \cdot 10^{-2}$	336 608	3 903 609	1 939 413	7 807 218

Calculation of cell measure

			1 th.		12 threads			GPU			
		types	CT	EBW	CT	EBW	Sp_{12}	CT	EBW	GSp_1	GSp_{12}
TNL	$2D_5^{\triangle}$	f, i, s	0,103	9,2	0,013	73	7,9	0,012	79	8,6	1,1
		f, l, i	0,104	16,0	0,014	119	7,4	0,012	139	8,7	1,2
		d, i, s	0,114	10,5	0,015	80	7,6	0,012	99	9,5	1,2
		d, l, i	0,120	15,9	0,015	127	8,0	0,012	159	10,0	1,2
	$3D_5^{\triangle}$	f, i, s	8,837	4,0	0,795	44	11,1	0,084	418	105,2	9,5
		f, l, i	10,173	6,5	0,930	71	10,9	0,123	537	82,7	7,6
		d, i, s	9,283	4,2	0,846	46	11,0	0,114	343	81,4	7,4
		d, l, i	10,757	6,5	0,998	70	10,8	0,144	487	74,7	6,9
MOAB	$2D_5^{\triangle}$	d, l, i	1,579	1,2	0,240	8	6,6				
	$3D_5^{\triangle}$	d, l, i	59,312	1,2	5,664	12	10,5				

Calculation of curve length or surface area around vertices

			1 th.		12 threads			GPU			
		types	CT	EBW	CT	EBW	Sp_{12}	CT	EBW	GSp_1	GSp_{12}
TNL	$2D_5^{\triangle}$	f, i, s	2,310	1,2	0,219	12	10,5	0,016	171	144,4	13,7
		f, l, i	2,211	2,4	0,208	25	10,6	0,019	276	116,4	10,9
		d, i, s	2,331	1,3	0,219	14	10,6	0,017	175	137,1	12,9
		d, l, i	2,149	2,6	0,209	26	10,3	0,021	261	102,3	10,0
	$3D_5^{\triangle}$	f, i, s	141,762	0,9	13,380	10	10,6	0,360	358	393,8	37,2
		f, l, i	157,331	1,6	15,773	16	10,0	0,496	512	317,2	31,8
		d, i, s	144,816	0,9	13,864	10	10,4	0,388	343	373,2	35,7
		d, l, i	162,191	1,6	16,194	16	10,0	0,529	488	306,6	30,6
MOAB	$2D_5^{\triangle}$	d, l, i	56,155	0,1	14,650	0	3,8				
	$3D_5^{\triangle}$	d, l, i	8 902,000	0,0	3 914,120	0	2,3				

ODEs solvers:

- Euler, Runge-Kutta-Merson

Linear systems solvers:

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

Preconditioners:

- Jacobi, ILU0(CPU only), ILUT(CPU only)

Efficient linear preconditioners for GPUs are still an open problem in general.

Solving PDEs with TNL

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 $t \in [0, T]$, $d = 1, 2, 3$.

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

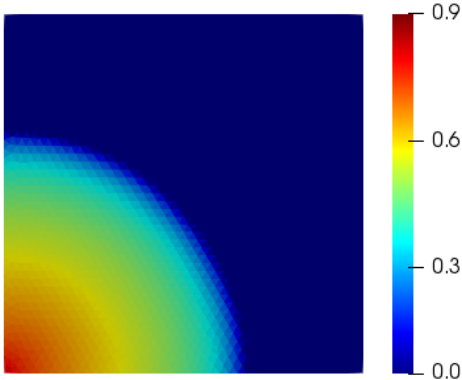
$$\vec{v}_i = -\sum_{j=1}^{n} \mathbf{D}_{i,j} \nabla Z_j + \mathbf{w}_i.$$

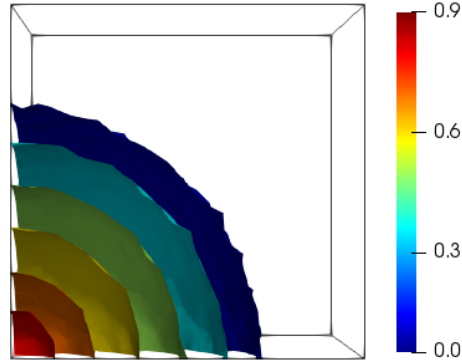
McWhorter–Sunada–Fučík problem from porous media flow

- Based on the mixed-hybrid finite element method (MHFEM)
- Semi-implicit time discretization
- General spatial dimension (1D, 2D, 3D)
- Structured and unstructured meshes

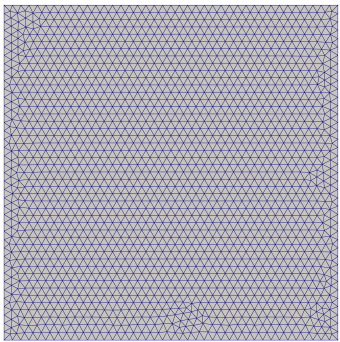
R. Fučík, J.Klinkovský, T. Oberhuber, J. Mikyška, *Multidimensional Mixed–Hybrid Finite Element Method for Compositional Two–Phase Flow in Heterogeneous Porous Media and its Parallel Implementation on GPU*, Computer Physics Communications, vol. 238, pp. 165-180, 2019.

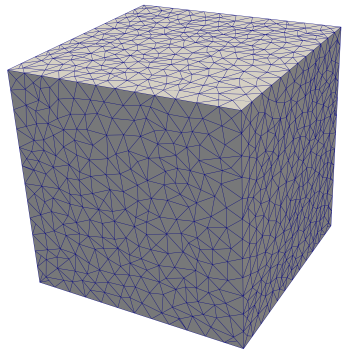
McWhorter–Sunada–Fučík problem





McWhorter–Sunada–Fučík problem





McWhorter–Sunada problem 2D

	GPU	CPU										
		1 core		2 cores			6 cores			12 cores		
Id.	CT	CT	GSp_1	CT	Eff	GSp_2	CT	Eff	GSp_6	CT	Eff	GSp_{12}
$2D_1^{\triangle}$	0,5	0,1	0,16	0,1	0,48	0,17	0,1	0,12	0,22	0,1	0,04	0,30
$2D_2^{\triangle}$	2,6	1,0	0,39	0,7	0,67	0,29	0,7	0,25	0,26	0,7	0,12	0,28
$2D_3^{\triangle}$	13,8	19,1	1,38	10,7	0,89	0,77	6,1	0,52	0,44	5,1	0,31	0,37
$2D_4^{\triangle}$	87,6	468,6	5,35	243,4	0,96	2,78	99,1	0,79	1,13	58,9	0,66	0,67
$2D_5^{\triangle}$	528,6	10 800,2	20,43	5 693,5	0,95	10,77	2 506,0	0,72	4,74	1 096,8	0,82	2,07

McWhorter–Sunada–Fučík problem 3D

	GPU	CPU										
		1 core		2 cores			6 cores			12 cores		
Id.	CT	CT	GSp_1	CT	Eff	GSp_2	CT	Eff	GSp_6	CT	Eff	GSp_{12}
$3D_1^{\triangle}$	0,3	0,4	1,09	0,2	0,80	0,68	0,1	0,41	0,44	0,2	0,18	0,50
$3D_2^{\triangle}$	0,6	1,6	2,78	0,9	0,87	1,59	0,5	0,60	0,78	0,4	0,36	0,65
$3D_3^{\triangle}$	3,5	67,4	19,35	35,8	0,94	10,28	15,7	0,71	4,52	7,8	0,72	2,24
$3D_4^{\triangle}$	54,2	2 918,9	53,87	1 551,1	0,94	28,63	786,7	0,62	14,52	396,1	0,61	7,31
$3D_5^{\triangle}$	2 654,8						46 014,7		17,33	23 949,5		9,02

Distributed McWhorter–Sunada problem 2D based on MPI

	GPU						
	1 rank	2 ranks		3 ranks		4 ranks	
Id.	CT	CT	Sp_2	CT	Sp_3	CT	Sp_4
$2D_1^{\triangle}$	0,5	0,8	0,6	0,9	0,5	1,1	0,4
$2D_2^{\triangle}$	2,6	3,7	0,7	4,4	0,6	5,1	0,5
$2D_3^{\triangle}$	13,8	19,1	0,7	23,0	0,6	26,7	0,5
$2D_4^{\triangle}$	87,6	113,3	0,8	126,2	0,7	148,7	0,6
$2D_5^{\triangle}$	528,6	566,1	0,9	642,5	0,8	709,7	0,7

Distributed McWhorter–Sunada–Fučík problem 3D based on MPI

	GPU						
	1 rank	2 ranks		3 ranks		4 ranks	
Id.	CT	CT	Sp_2	CT	Sp_3	CT	Sp_4
$3D_1^{\triangle}$	0,3	0,5	0,6	0,7	0,5	0,8	0,4
$3D_2^{\triangle}$	0,6	0,9	0,7	1,1	0,5	1,3	0,5
$3D_3^{\triangle}$	3,5	4,2	0,8	4,8	0,7	5,7	0,6
$3D_4^{\triangle}$	54,2	36,7	1,5	33,4	1,6	30,6	1,8
$3D_5^{\triangle}$	2 654,8	1 415,4	1,9	996,7	2,7	793,3	3,3

Method for solving the Boltzmann equation:

$$\frac{\partial f(\vec{v},\vec{x},t)}{\partial t} = C(\vec{v},\vec{x},t) + S(\vec{v},\vec{x},t),$$

where

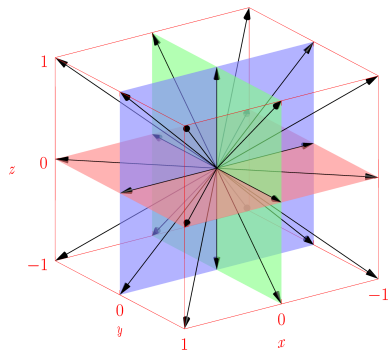
- $f(\vec{v},\vec{x},t)$ is distribution function of particles with velocity $\vec{v}$ at position $\vec{x}$ and time t
- $C(\vec{v},\vec{x},t)$ describes particles collisions
- $S(\vec{v},\vec{x},t)$ is a source term

The discretization is done as:

- $\vec{v} \rightarrow v_i$ for $i = 1, \dots q$ (9 in 2D and 27 in 3D)
- $\vec{x} \rightarrow \Delta x$ (lattice)
- $t \rightarrow \Delta t$ time steps
- $C(\vec{v}, \vec{x}, t) \rightarrow$ SRT, MRT, Entropic, Cumulant, ...

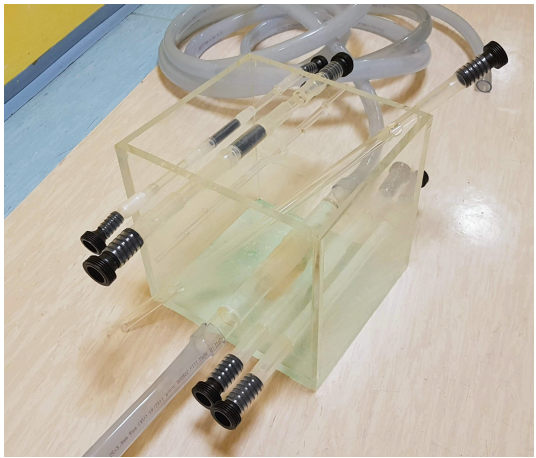
Macroscopic quantities:

- $\rho = \sum_{i=1}^{q} f_i$
- $\rho \vec{v} = \sum_{i=1}^{q} f_i \vec{c}_i$
- + energy, stress tensor, ...



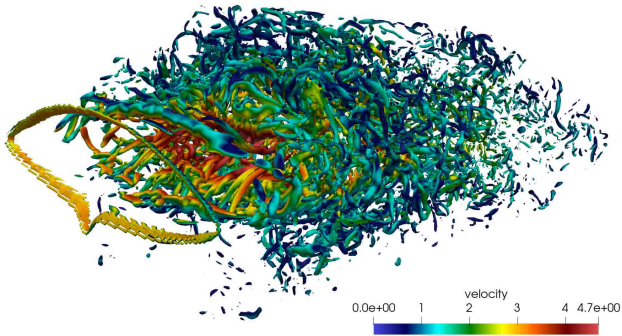
Blood flow through heart valve

Blood flow through heart valve



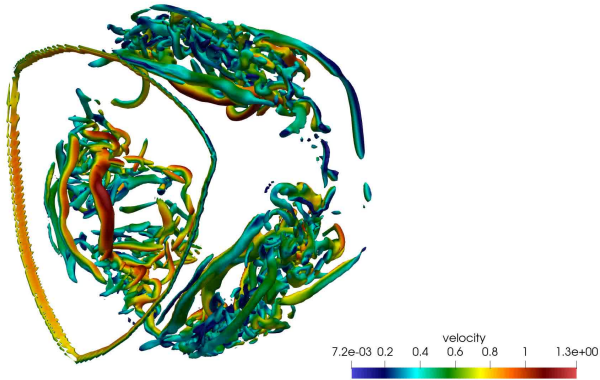
Blood flow through heart valve

Blood flow through heart valve



Flow simulation with $Re = 5 \cdot 10^4$.

Blood flow through heart valve

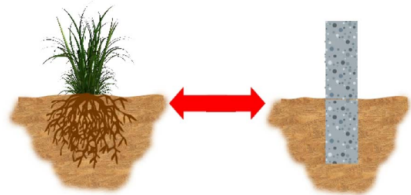


Flow simulation with $Re = 5 \cdot 10^4$.

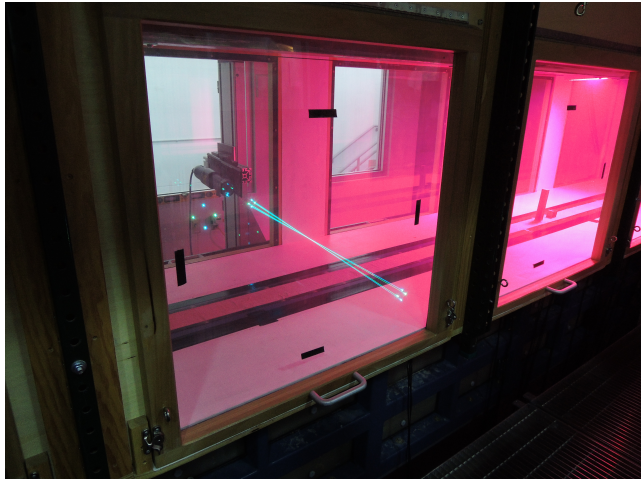
Coupled LBM-MHFEM simulator for vapor transport in air over a moist soil

Simulation of:

- Water evaporation from soil and vapor transport above soil layer.
- Live vegetation approximated with limestone blocks.
- Measurements: air flow characteristics, relative humidity (vapor concentration), soil moisture.
- Solved by coupled LBM-MHFEM solver running completely on GPU.

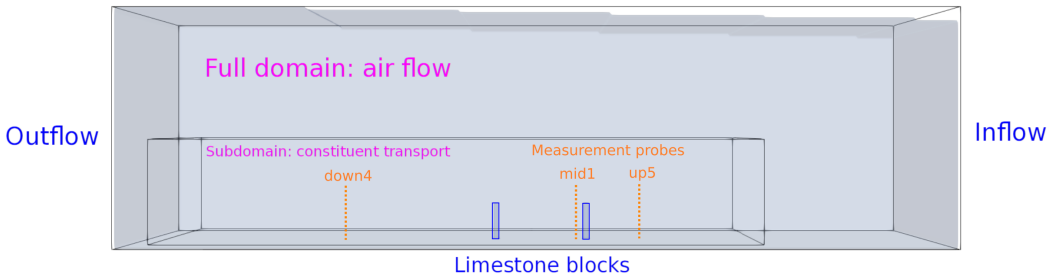


Coupled LBM-MHFEM simulator for vapor transport in air over a moist soil

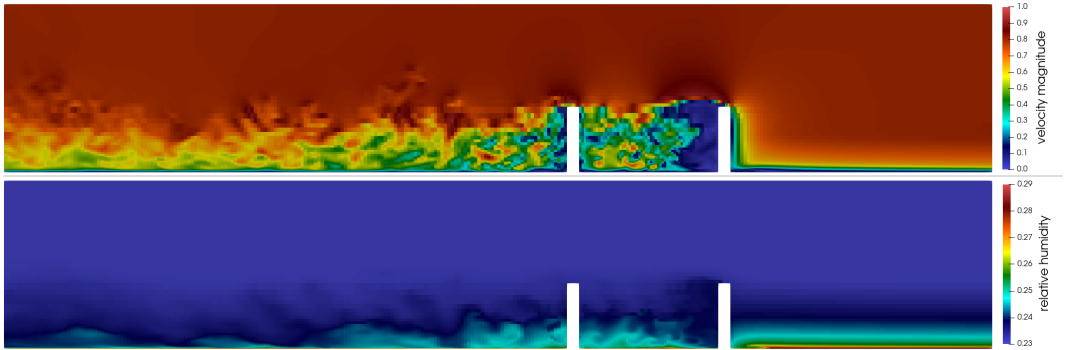


Wind tunnel (CESEP, Colorado School of Mines, USA).

Coupled LBM-MHFEM simulator for vapor transport in air over a moist soil



Coupled LBM-MHFEM simulator for vapor transport in air over a moist soil

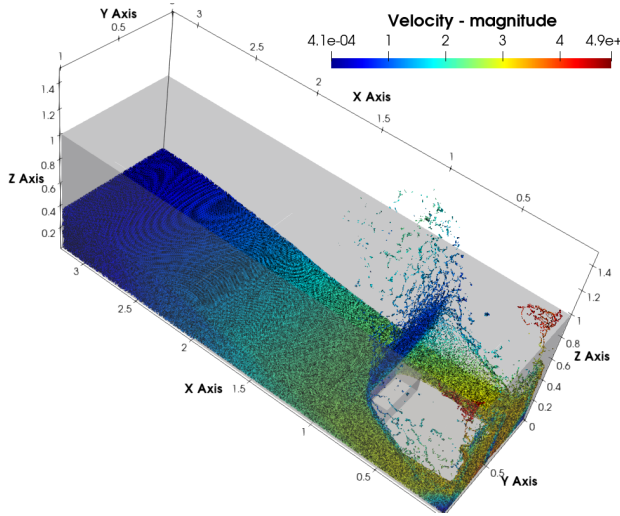


Simulations – velocity and relative humidity profiles

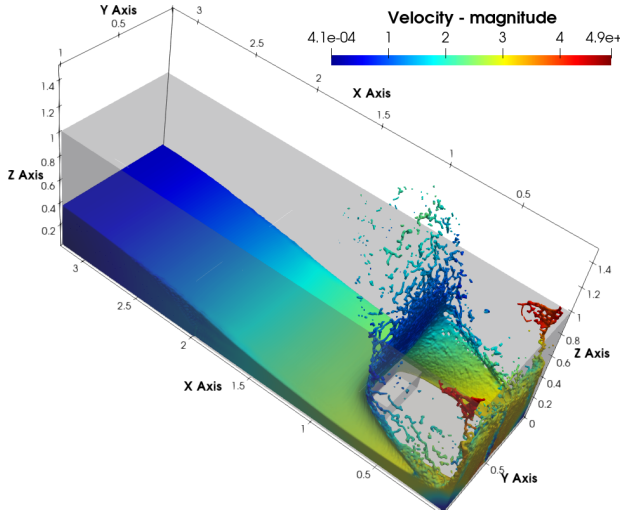
SPH - Smoothed-particle hydrodynamics

- SPH is meshfree Lagrangian method for simulation of fluid flow
- It is especially efficient for flow with free boundary
- since it is meshfree, it is much more suitable for parallel computing on GPUs
- implementation in TNL is by Tomáš Halada, FMI at CTU in Prague.

SPH - Smoothed-particle hydrodynamics



SPH - Smoothed-particle hydrodynamics



Comparison of TNL::SPH and DualSPHysics:

Particles count	Dual SPHysics [s]	TNL::SPH [s]	Speed-up
172,422	0.0014	0.0012	**1.16**
1,015,809	0.0055	0.0040	**1.37**
6,751,766	0.0396	0.0282	**1.40**

Computed on Nvidia A40 and AMD EPYC 7543 32-Core Processor.

We have presented:

- key features of TNL
- MHFEM solver for porous media flow
- LBM solver for simulation of turbulent flow
- coupled LBM-MHFEM solver runnig completely on GPU
- implementation of SPH (=smoothed-particle hydrodynamics)

We are working on:

- support of GPUs by AMD
- orthogonal grids, polyhedral and adaptive meshes
- implementation of FEM
- and many other things ...