# Programming GPU with TNL

**Tomáš Oberhuber**, J. Klinkovský, R. Fučík

Katedra matematiky,
Fakulta jaderná a fyzikálně inženýrská,
České vysoké učení technické v Praze

Winter Kindergarten School on LBM in Krakow 2022

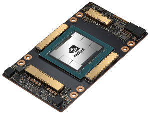
# Overview

# Lattice Boltzmann Method

**LBM** - Lattice Boltzmann Method

- LBM is an efficient numerical method
- it is easy to implement
- perfectly profits from the GPU architecture

# Comparison of GPU and CPU






| | Nvidia A100 | AMD Instinct MI100 | AMD EPYC 2 Rome 7H12 |
|---|---|---|---|
| Počet jader | 6912 @ 1.41GHz | 7680 @ 1.2GHz | 64 @ 2.4GHz |
| Max. výkon | 19.5/9.7 TFlops | 23/11 TFlops | 2 / 1 TFlops |
| Max. výkon s tenzory | 156 /– TFlops | –/– | –/– |
| Max. RAM | 80 GB | 32 GB | 2 TB |
| Datová propustnost | 2039 GB/s | 1200 GB/s | 204 GB/s |
| Energetická náročnost | 400 W | 300 W | 280 W |

**Unfortunately, programming GPU is difficult.**

# Template Numerical Library

**TNL** = Template Numerical Library

- numerical library for modern parallel architectures
- written in C++ and profiting from features of C++14 and C++17
- offers unified interface for both multi-core CPUs a GPUs
- **www.tnl-project.org**
- MIT licence
- affiliated project of Numfocus (www.numfocus.org)

# Installation of TNL

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`.

# Working with address spaces

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

# Memory allocation in TNL

Memory allocation is done by templated class `Vector`:

```
namespace TNL::Containers;
template< typename Real = double,
          typename Device = TNL::Devices::Host,
          typename Index = int >
class 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 GPU and global memory
- `Index` is type for indexing the elements in the vector

# Memory allocation in TNL

```
#include <iostream>
#include <list>
#include <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;
```

# Memory allocation in TNL

```
    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 ]
```

# Expression templates for vectors

The following vector expression

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

can be evaluated with Blas/Cublas as follows:

```
cublasHandle_t c_h;
cublasSaxpy(c_h,N,1.0,a,1,x,1); // -> x = a (assume x = [0,...,0] at the beginning)
cublasSaxpy(c_h,N,2.0,b,1,x,1); // -> x = x + 2 * b
cublasSaxpy(c_h,N,3.0,c,1,x,1); // -> x = x + 3 * c
```

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

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

The later is **more efficient**.

# Expression templates for vectors

- 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 epxression on-the-fly, i.e.

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

- on CPU, the assignment would be evaluated as follows

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

# Vector operations

We setup the following vectors:

```
    using namespace std;
    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
```

# Vector operations

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
```

# Vector operations

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 ]
```

# Vector operations

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
```

# Vector operations

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 ]
```

# Vector operations

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
```

# Using lambda functions

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 ]
```

# Computations on GPU

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

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

const float h = 0.5;  // capture this variable h
auto f = [=] __cuda_callable__ ( int idx, float& value ) {
  value += h * idx;  // use it here inside the lambda function
};
a.forAllElements( f );
cout << "a = " << a << endl;
```

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

# Computations on GPU

- if the lambda function is executed on GPU, the captured **variable must be copied on the GPU**
- this is why all variables **must be captured as a copy** not as a reference
  - we must use the statement `[=]` in the definition of the lambda function
  - we cannot use `[&]`
- with the lambda functions, the captured variables are transferred on the GPU **automatically**
- if we write CUDA kernels, we have to transfer all variables explicitly = very annoying

# Vector view

- what if we need to work with **two vectors**?
- vector **cannot be captured as a copy**:
  - it would create a new copy of the vector ⇒ **additional memory** is needed
  - we would not be able **to modify the original vector**, only the copy
  - the copy is temporal and it is lost when the lambda function finishes
- we have to use a **vector view** instead of vector
  - vector view shares data with another vector
  - copy of a vector view is not a deep copy ⇒ elements of the original vector are not duplicated

# Vector view - example

```
using Vector = TNL::Containers::Vector< float, TNL::Devices::Cuda >;
Vector a( 6, 1.0 ), b( 6, 3.0 );
cout << "a = " << a << endl;

const float h = 0.5;
auto b_view = b.getView();         // this view can be captured by making copy
auto f = [=] __cuda_callable__ ( int idx, float& value ) {
  value = h * idx + b_view[ idx ];  // we can use the view even on the GPU
};
a.forAllElements( f );
cout << "a = " << a << endl;
```

The result looks as follows:

```
a = [ 1, 1, 1, 1, 1, 1 ]
a = [ 3, 3.5, 4, 4.5, 5, 5.5 ]
```

# Vector view - example

Vector view can be used for encapsulating data not allocated by TNL:

```
using VectorView = TNL::Containers::VectorView< float, TNL::Devices::Host >;
float* data = new float[ 6 ];
VectorView a( data, 6 );
a = 1.0;
cout << "a = " << a << endl;

auto f = [] __cuda_callable__ ( int idx, float& value ) {
   value += 0.5 * idx;
};
a.forAllElements( f );
cout << "a = " << a << endl;

delete[] data;
```

## Reduction

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

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

# Parallel reduction on GPU = 100 lines of code

```
template <unsigned int blockSize>
__global__ void cudaBlockReduction6(int size, int* dInput, int* dOutput, )
{
   extern __shared__ volatile int sdata[];
   unsigned int tid = threadIdx.x;
   unsigned int gid = blockIdx.x*blockSize*2 + threadIdx.x;
   unsigned int gridSize = blockSize*2*gridDim.x;
   sdata[tid] = 0;
   while (gid < size)
   {
      sdata[tid] += dInput[gid] + dInput[gid+blockSize];
      gid += gridSize;
   }
   __syncthreads();

   if (blockSize == 1024) {
      if( tid<512 ) { sdata[ tid ] += sdata[ tid+512 ]; } __syncthreads(); }
   if (blockSize >= 512) {
      if( tid<256 ) { sdata[ tid ] += sdata[ tid+256 ]; } __syncthreads(); }
   if (blockSize >= 256) {
      if( tid<128 ) { sdata[ tid ] += sdata[ tid+128 ]; } __syncthreads(); }
   if (blockSize >= 128) {
      if( tid< 64 ) { sdata[ tid ] += sdata[ tid+ 64 ]; } __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 cudaGridReduction6( const int size,
                         const int* deviceData,
                         const int blockSize,
                         const int desiredGridSize,
                         int& result )
{
   const int elementsInBlock = 2 * blockSize;
   int gridSize = size / elementsInBlock + ( size % elementsInBlock != 0 );
   gridSize = gridSize < desiredGridSize ? gridSize : desiredGridSize;
   int auxArraySize = gridSize * sizeof( int );
   int *deviceAux1, *deviceAux2;
   if( cudaMalloc( ( void** ) &deviceAux1, auxArraySize ) != cudaSuccess ||
       cudaMalloc( ( void** ) &deviceAux2, auxArraySize ) != cudaSuccess )
   {
      cerr << "Not enough memory on device." << endl;
      return false;
   }

   int currentSize = size;
   const int* reductionInput = deviceData;
   while( currentSize > 1 )
   {
      dim3 cudaBlockSize( 0 ), cudaGridSize( 0 );
      cudaBlockSize. x = blockSize;
      gridSize = currentSize / elementsInBlock +
                 ( currentSize % elementsInBlock != 0 );
      cudaGridSize. x = gridSize < desiredGridSize ? gridSize : desiredGridSize;

      size_t sharedMemory = blockSize * sizeof( int );
      switch( blockSize )
      {
         case 256:
            cudaBlockReduction6< 256 >
               <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
               ( currentSize, reductionInput, deviceAux1 );
            break;
            break;

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

      reductionInput = deviceAux1;
      int* tmp = deviceAux1;
      deviceAux1 = deviceAux2;
      deviceAux2 = tmp;
   }
   if( cudaMemcpy( &result,
                   reductionInput,
                   sizeof( int ),
                   cudaMemcpyDeviceToHost ) != cudaSuccess )
   {
      cerr << "An error occured while transfering data from device to host."
           << endl;
      return false;
   }
   cudaFree( deviceAux1 );
   cudaFree( deviceAux2 );
   return true;
```

# Flexible parallel reduction

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 ) );
```

# Flexible parallel reduction

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

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

result = TNL::Algorithms::reduce< Device >( 0, a.getSize(), fetch, reduction, 0.0 );
```

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

# Flexible parallel reduction

Comparison of two vectors can be evaluated as follows:

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

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

# Flexible parallel reduction

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(),
            [=] (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.

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

# Solving the heat equation

At the end, we show FDM solver for the heat equation given as:

$$\begin{aligned}
\frac{\partial u(\vec{x},t)}{\partial t} - \Delta u(\vec{x},t) &= 0 \text{ on } \Omega \times [0,T], \\
u(\vec{x},0) &= u_{ini}(\vec{x}) \text{ on } \Omega, \\
u(\vec{x},t) &= 0 \text{ on } \partial\Omega \times [0,T].
\end{aligned}$$

We approximate it by the finite difference method:

$$\begin{aligned}
u_{ij}^{k+1} &= u_{ij}^{k} + \frac{\tau}{h^2}\left(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1} - 4u_{ij}\right) \text{ on } \Omega_h, \\
u_{ij}^{k+1} &= = 0 \text{ on } \partial\Omega_h.
\end{aligned}$$

# Solving the heat equation

```
#include <TNL/Containers/Vector.h>
#include <TNL/Algorithms/reduce.h>
#include <TNL/Algorithms/ParallelFor.h>

using Device = TNL::Devices::Host;

int main( int argc, char* argv[] )
{
   using Vector = TNL::Containers::Vector< float, Device >;

   /////
   // Parameters of the discretization
   int size = 11;
   float h = 1.0 / ( size - 1 );
   float tau = 0.1 * h * h;
   float final_T = 1.0;
   float h_inv_sqr = 1.0 / ( h * h );
```

# Solving the heat equation

```
    /////
    // Allocation of the grid functions
    Vector u( size * size, 0.0 ), aux( size * size, 0.0 );
    auto u_view = u.getView();
    TNL::Algorithms::ParallelFor2D< Device >::exec( 0, 0, size, size,
     [=] __cuda_callable__ ( int i, int j ) mutable {
        float x = i * h - 0.5;
        float y = j * h - 0.5;
        int idx = j * size + i;
        if( x * x + y * y < 0.25 )
           u_view[ idx ] = 1.0; } );
```

# Solving the heat equation

```
    /////
    // Time loop
    float t = 0.0;
    float residue = 10.0;
    while( t < final_T && residue > 1.0e-6 )
    {
       /////
       // Update with FDM approximation of the Laplace operator
       auto u_view = u.getView();
       auto aux_view = aux.getView();
```

# Solving the heat equation

```
      residue = sqrt( TNL::Algorithms::reduce< Device >( 0, size * size,
         [=] __cuda_callable__ ( int idx ) mutable -> float {
            int i = idx % size;
            int j = idx / size;
            if( i == 0 || j == 0 || i == size - 1 || j == size - 1 )
               return 0.0;
            float update = h_inv_sqr * (
               u_view[ idx - size ] + u_view[ idx - 1 ] +
               u_view[ idx + size ] + u_view[ idx + 1 ]
               - 4.0 * u_view[ idx ] );
            aux_view[ idx ] = u_view[ idx ] + tau * update;
            return update * update;
         },
         TNL::Plus() ) );
```

# Solving the heat equation

```
        t += tau;
        u.swap( aux );
        std::cout << "t = " << t << " residue = " << residue << std::endl;
    }
    return EXIT_SUCCESS;
}
```

# Summary

- TNL allows simple and efficient way how to develop code for GPUs
- there is no performance drop
- the code can be written and **DEBUGGED** on CPU
- with a little bit of luck it runs on GPU as well