

# 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

CPU, GPU & LBM

TNL, Template Numerical Library

Memory management in TNL

Vector expressions

Parallel reduction

Solving the heat equation

# 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](http://www.tnl-project.org)**
- ▶ MIT licence
- ▶ affiliated project of Numfocus ([www.numfocus.org](http://www.numfocus.org))



TEMPLATE  
NUMERICAL  
LIBRARY



## Installation of TNL

The library can be downloaded using git as follows:

```
1 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:

```
1 cd tnl-dev
2 ./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:

```
1 namespace TNL::Containers;
2 template< typename Real = double,
3           typename Device = TNL::Devices::Host,
4           typename Index = int >
5 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

```
1  #include <iostream>
2  #include <list>
3  #include <vector>
4  #include <TNL/Containers/Vector.h>
5
6  using namespace TNL;
7  using namespace TNL::Containers;
8
9  int main( int argc, char* argv[] )
10 {
11     Vector< int > host_vector( 10 );           // vector with 10 elements on CPU
12     Vector< int, Devices::Cuda > device_vector; // empty vector on GPU
13
14     host_vector = 3;                          // set all elements to 3
15     device_vector = host_vector;              // copy the host vector on GPU
16
17     std::cout << "host_vector = " << host_vector << std::endl;
18     std::cout << "device_vector = " << device_vector << std::endl;
19     std::cout << std::endl;
```

## Memory allocation in TNL

```
1  std::list< int > list { 1, 2, 3, 4, 5 };
2  std::vector< int > vector { 6, 7, 8, 9, 10 };
3
4  Vector< int, Devices::Cuda > device_vector_list( list );
5  Vector< int, Devices::Cuda > device_vector_vector( vector );
6  Vector< int, Devices::Cuda > device_vector_init_list{ 11, 12, 13, 14, 15 };
7
8  std::cout << "device_vector_list = " << device_vector_list << std::endl;
9  std::cout << "device_vector_vector = " << device_vector_vector << std::endl;
10 std::cout << "device_vector_init_list = " << device_vector_init_list << std::endl;
11 }
```

The result looks as follows:

```
1  host_vector = [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ]
2  device_vector = [ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ]
3
4  device_vector_list = [ 1, 2, 3, 4, 5 ]
5  device_vector_vector = [ 6, 7, 8, 9, 10 ]
6  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:

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

## 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 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	<b>2.2</b>	194.7	236.5	<b>1.21</b>
200k	19.7	41.7	<b>2.1</b>	228.3	277.6	<b>1.21</b>
400k	17.3	35.9	<b>2.1</b>	218.3	330.9	<b>1.51</b>
800k	11.7	19.3	<b>1.6</b>	233.3	370.6	<b>1.58</b>
1.6M	10.4	17.0	<b>1.6</b>	249.6	403.4	<b>1.61</b>
3.2M	10.2	17.3	<b>1.7</b>	266.6	444.8	<b>1.66</b>
6.4M	10.2	17.3	<b>1.7</b>	276.6	471.3	<b>1.70</b>

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:

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

We may search for maximum and minimum:

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

The result looks as follows:

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

## Vector operations

We may combine vector operations with ET:

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

The result looks as follows:

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

## Vector operations

We may compute minimum and maximum componentwise:

```
1  cout << "min( a, b ) = " << min( a, b ) << endl
2      << "max( a, b ) = " << max( a, b ) << endl
3      << "min( max( a, b ) ) = " << min( max( a, b ) ) << endl
4      << "max( abs( a ), b ) = " << max( abs( a ), b ) << endl;
```

The result looks as follows:

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



## Vector operations

We may locate minimal and maximal elements:

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

The result looks as follows:

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

## Vector operations

We may perform componentwise operations:

```
1 cout << "a + b = " << a + b << endl
2   << "a - b = " << a - b << endl
3   << "a * b = " << a * b << endl
4   << "a / b = " << a / b << endl;
```

The result looks as follows:

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

## Vector operations

We may compute norms and scalar products:

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

The result looks as follows:

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

## Using lambda functions

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

```
1 using Vector = TNL::Containers::Vector< float, TNL::Devices::Cuda >;
2 Vector a( 6, 1.0 );
3 cout << "a = " << a << endl;
4
5 auto f = [] __cuda_callable__ ( int idx, float& value ) { // This all is
6     value += 0.5 * idx; // turned into
7 }; // a CUDA kernel
8 a.forAllElements( f ); // by C++ compiler.
9
10 cout << "a = " << a << endl;
```

The result looks as:

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

## Computations on GPU

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.

## 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  $\Rightarrow$  **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  $\Rightarrow$  elements of the original vector are not duplicated

## Vector view - example

```
1 using Vector = TNL::Containers::Vector< float, TNL::Devices::Cuda >;
2 Vector a( 6, 1.0 ), b( 6, 3.0 );
3 cout << "a = " << a << endl;
4
5 const float h = 0.5;
6 auto b_view = b.getView();           // this view can be captured by making copy
7 auto f = [=] __cuda_callable__ ( int idx, float& value ) {
8     value = h * idx + b_view[ idx ]; // we can use the view even on the GPU
9 };
10 a.forAllElements( f );
11 cout << "a = " << a << endl;
```

The result looks as follows:

```
1 a = [ 1, 1, 1, 1, 1, 1 ]
2 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:

```
1  using VectorView = TNL::Containers::VectorView< float, TNL::Devices::Host >;
2  float* data = new float[ 6 ];
3  VectorView a( data, 6 );
4  a = 1.0;
5  cout << "a = " << a << endl;
6
7  auto f = [] __cuda_callable__ ( int idx, float& value ) {
8      value += 0.5 * idx;
9  };
10 a.forAllElements( f );
11 cout << "a = " << a << endl;
12
13 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

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

# Parallel reduction on GPU = 100 lines of code

```
1 template<typename ScalarType>
2 auto reduce(ScalarType const& src, ScalarType const& dst, ScalarType const& src0, ScalarType const& dst0,
3            ScalarType const& src1, ScalarType const& dst1, ScalarType const& src2, ScalarType const& dst2,
4            ScalarType const& src3, ScalarType const& dst3, ScalarType const& src4, ScalarType const& dst4,
5            ScalarType const& src5, ScalarType const& dst5, ScalarType const& src6, ScalarType const& dst6,
6            ScalarType const& src7, ScalarType const& dst7, ScalarType const& src8, ScalarType const& dst8,
7            ScalarType const& src9, ScalarType const& dst9, ScalarType const& src10, ScalarType const& dst10,
8            ScalarType const& src11, ScalarType const& dst11, ScalarType const& src12, ScalarType const& dst12,
9            ScalarType const& src13, ScalarType const& dst13, ScalarType const& src14, ScalarType const& dst14,
10           ScalarType const& src15, ScalarType const& dst15, ScalarType const& src16, ScalarType const& dst16,
11           ScalarType const& src17, ScalarType const& dst17, ScalarType const& src18, ScalarType const& dst18,
12           ScalarType const& src19, ScalarType const& dst19, ScalarType const& src20, ScalarType const& dst20,
13           ScalarType const& src21, ScalarType const& dst21, ScalarType const& src22, ScalarType const& dst22,
14           ScalarType const& src23, ScalarType const& dst23, ScalarType const& src24, ScalarType const& dst24,
15           ScalarType const& src25, ScalarType const& dst25, ScalarType const& src26, ScalarType const& dst26,
16           ScalarType const& src27, ScalarType const& dst27, ScalarType const& src28, ScalarType const& dst28,
17           ScalarType const& src29, ScalarType const& dst29, ScalarType const& src30, ScalarType const& dst30,
18           ScalarType const& src31, ScalarType const& dst31, ScalarType const& src32, ScalarType const& dst32,
19           ScalarType const& src33, ScalarType const& dst33, ScalarType const& src34, ScalarType const& dst34,
20           ScalarType const& src35, ScalarType const& dst35, ScalarType const& src36, ScalarType const& dst36,
21           ScalarType const& src37, ScalarType const& dst37, ScalarType const& src38, ScalarType const& dst38,
22           ScalarType const& src39, ScalarType const& dst39, ScalarType const& src40, ScalarType const& dst40,
23           ScalarType const& src41, ScalarType const& dst41, ScalarType const& src42, ScalarType const& dst42,
24           ScalarType const& src43, ScalarType const& dst43, ScalarType const& src44, ScalarType const& dst44,
25           ScalarType const& src45, ScalarType const& dst45, ScalarType const& src46, ScalarType const& dst46,
26           ScalarType const& src47, ScalarType const& dst47, ScalarType const& src48, ScalarType const& dst48,
27           ScalarType const& src49, ScalarType const& dst49, ScalarType const& src50, ScalarType const& dst50,
28           ScalarType const& src51, ScalarType const& dst51, ScalarType const& src52, ScalarType const& dst52,
29           ScalarType const& src53, ScalarType const& dst53, ScalarType const& src54, ScalarType const& dst54,
30           ScalarType const& src55, ScalarType const& dst55, ScalarType const& src56, ScalarType const& dst56,
31           ScalarType const& src57, ScalarType const& dst57, ScalarType const& src58, ScalarType const& dst58,
32           ScalarType const& src59, ScalarType const& dst59, ScalarType const& src60, ScalarType const& dst60,
33           ScalarType const& src61, ScalarType const& dst61, ScalarType const& src62, ScalarType const& dst62,
34           ScalarType const& src63, ScalarType const& dst63, ScalarType const& src64, ScalarType const& dst64,
35           ScalarType const& src65, ScalarType const& dst65, ScalarType const& src66, ScalarType const& dst66,
36           ScalarType const& src67, ScalarType const& dst67, ScalarType const& src68, ScalarType const& dst68,
37           ScalarType const& src69, ScalarType const& dst69, ScalarType const& src70, ScalarType const& dst70,
38           ScalarType const& src71, ScalarType const& dst71, ScalarType const& src72, ScalarType const& dst72,
39           ScalarType const& src73, ScalarType const& dst73, ScalarType const& src74, ScalarType const& dst74,
40           ScalarType const& src75, ScalarType const& dst75, ScalarType const& src76, ScalarType const& dst76,
41           ScalarType const& src77, ScalarType const& dst77, ScalarType const& src78, ScalarType const& dst78,
42           ScalarType const& src79, ScalarType const& dst79, ScalarType const& src80, ScalarType const& dst80,
43           ScalarType const& src81, ScalarType const& dst81, ScalarType const& src82, ScalarType const& dst82,
44           ScalarType const& src83, ScalarType const& dst83, ScalarType const& src84, ScalarType const& dst84,
45           ScalarType const& src85, ScalarType const& dst85, ScalarType const& src86, ScalarType const& dst86,
46           ScalarType const& src87, ScalarType const& dst87, ScalarType const& src88, ScalarType const& dst88,
47           ScalarType const& src89, ScalarType const& dst89, ScalarType const& src90, ScalarType const& dst90,
48           ScalarType const& src91, ScalarType const& dst91, ScalarType const& src92, ScalarType const& dst92,
49           ScalarType const& src93, ScalarType const& dst93, ScalarType const& src94, ScalarType const& dst94,
50           ScalarType const& src95, ScalarType const& dst95, ScalarType const& src96, ScalarType const& dst96,
51           ScalarType const& src97, ScalarType const& dst97, ScalarType const& src98, ScalarType const& dst98,
52           ScalarType const& src99, ScalarType const& dst99)
53 {
54     auto src0_ptr = src0;
55     auto dst0_ptr = dst0;
56     auto src1_ptr = src1;
57     auto dst1_ptr = dst1;
58     auto src2_ptr = src2;
59     auto dst2_ptr = dst2;
60     auto src3_ptr = src3;
61     auto dst3_ptr = dst3;
62     auto src4_ptr = src4;
63     auto dst4_ptr = dst4;
64     auto src5_ptr = src5;
65     auto dst5_ptr = dst5;
66     auto src6_ptr = src6;
67     auto dst6_ptr = dst6;
68     auto src7_ptr = src7;
69     auto dst7_ptr = dst7;
70     auto src8_ptr = src8;
71     auto dst8_ptr = dst8;
72     auto src9_ptr = src9;
73     auto dst9_ptr = dst9;
74     auto src10_ptr = src10;
75     auto dst10_ptr = dst10;
76     auto src11_ptr = src11;
77     auto dst11_ptr = dst11;
78     auto src12_ptr = src12;
79     auto dst12_ptr = dst12;
80     auto src13_ptr = src13;
81     auto dst13_ptr = dst13;
82     auto src14_ptr = src14;
83     auto dst14_ptr = dst14;
84     auto src15_ptr = src15;
85     auto dst15_ptr = dst15;
86     auto src16_ptr = src16;
87     auto dst16_ptr = dst16;
88     auto src17_ptr = src17;
89     auto dst17_ptr = dst17;
90     auto src18_ptr = src18;
91     auto dst18_ptr = dst18;
92     auto src19_ptr = src19;
93     auto dst19_ptr = dst19;
94     auto src20_ptr = src20;
95     auto dst20_ptr = dst20;
96     auto src21_ptr = src21;
97     auto dst21_ptr = dst21;
98     auto src22_ptr = src22;
99     auto dst22_ptr = dst22;
100    auto src23_ptr = src23;
101    auto dst23_ptr = dst23;
102    auto src24_ptr = src24;
103    auto dst24_ptr = dst24;
104    auto src25_ptr = src25;
105    auto dst25_ptr = dst25;
106    auto src26_ptr = src26;
107    auto dst26_ptr = dst26;
108    auto src27_ptr = src27;
109    auto dst27_ptr = dst27;
110    auto src28_ptr = src28;
111    auto dst28_ptr = dst28;
112    auto src29_ptr = src29;
113    auto dst29_ptr = dst29;
114    auto src30_ptr = src30;
115    auto dst30_ptr = dst30;
116    auto src31_ptr = src31;
117    auto dst31_ptr = dst31;
118    auto src32_ptr = src32;
119    auto dst32_ptr = dst32;
120    auto src33_ptr = src33;
121    auto dst33_ptr = dst33;
122    auto src34_ptr = src34;
123    auto dst34_ptr = dst34;
124    auto src35_ptr = src35;
125    auto dst35_ptr = dst35;
126    auto src36_ptr = src36;
127    auto dst36_ptr = dst36;
128    auto src37_ptr = src37;
129    auto dst37_ptr = dst37;
130    auto src38_ptr = src38;
131    auto dst38_ptr = dst38;
132    auto src39_ptr = src39;
133    auto dst39_ptr = dst39;
134    auto src40_ptr = src40;
135    auto dst40_ptr = dst40;
136    auto src41_ptr = src41;
137    auto dst41_ptr = dst41;
138    auto src42_ptr = src42;
139    auto dst42_ptr = dst42;
140    auto src43_ptr = src43;
141    auto dst43_ptr = dst43;
142    auto src44_ptr = src44;
143    auto dst44_ptr = dst44;
144    auto src45_ptr = src45;
145    auto dst45_ptr = dst45;
146    auto src46_ptr = src46;
147    auto dst46_ptr = dst46;
148    auto src47_ptr = src47;
149    auto dst47_ptr = dst47;
150    auto src48_ptr = src48;
151    auto dst48_ptr = dst48;
152    auto src49_ptr = src49;
153    auto dst49_ptr = dst49;
154    auto src50_ptr = src50;
155    auto dst50_ptr = dst50;
156    auto src51_ptr = src51;
157    auto dst51_ptr = dst51;
158    auto src52_ptr = src52;
159    auto dst52_ptr = dst52;
160    auto src53_ptr = src53;
161    auto dst53_ptr = dst53;
162    auto src54_ptr = src54;
163    auto dst54_ptr = dst54;
164    auto src55_ptr = src55;
165    auto dst55_ptr = dst55;
166    auto src56_ptr = src56;
167    auto dst56_ptr = dst56;
168    auto src57_ptr = src57;
169    auto dst57_ptr = dst57;
170    auto src58_ptr = src58;
171    auto dst58_ptr = dst58;
172    auto src59_ptr = src59;
173    auto dst59_ptr = dst59;
174    auto src60_ptr = src60;
175    auto dst60_ptr = dst60;
176    auto src61_ptr = src61;
177    auto dst61_ptr = dst61;
178    auto src62_ptr = src62;
179    auto dst62_ptr = dst62;
180    auto src63_ptr = src63;
181    auto dst63_ptr = dst63;
182    auto src64_ptr = src64;
183    auto dst64_ptr = dst64;
184    auto src65_ptr = src65;
185    auto dst65_ptr = dst65;
186    auto src66_ptr = src66;
187    auto dst66_ptr = dst66;
188    auto src67_ptr = src67;
189    auto dst67_ptr = dst67;
190    auto src68_ptr = src68;
191    auto dst68_ptr = dst68;
192    auto src69_ptr = src69;
193    auto dst69_ptr = dst69;
194    auto src70_ptr = src70;
195    auto dst70_ptr = dst70;
196    auto src71_ptr = src71;
197    auto dst71_ptr = dst71;
198    auto src72_ptr = src72;
199    auto dst72_ptr = dst72;
200    auto src73_ptr = src73;
201    auto dst73_ptr = dst73;
202    auto src74_ptr = src74;
203    auto dst74_ptr = dst74;
204    auto src75_ptr = src75;
205    auto dst75_ptr = dst75;
206    auto src76_ptr = src76;
207    auto dst76_ptr = dst76;
208    auto src77_ptr = src77;
209    auto dst77_ptr = dst77;
210    auto src78_ptr = src78;
211    auto dst78_ptr = dst78;
212    auto src79_ptr = src79;
213    auto dst79_ptr = dst79;
214    auto src80_ptr = src80;
215    auto dst80_ptr = dst80;
216    auto src81_ptr = src81;
217    auto dst81_ptr = dst81;
218    auto src82_ptr = src82;
219    auto dst82_ptr = dst82;
220    auto src83_ptr = src83;
221    auto dst83_ptr = dst83;
222    auto src84_ptr = src84;
223    auto dst84_ptr = dst84;
224    auto src85_ptr = src85;
225    auto dst85_ptr = dst85;
226    auto src86_ptr = src86;
227    auto dst86_ptr = dst86;
228    auto src87_ptr = src87;
229    auto dst87_ptr = dst87;
230    auto src88_ptr = src88;
231    auto dst88_ptr = dst88;
232    auto src89_ptr = src89;
233    auto dst89_ptr = dst89;
234    auto src90_ptr = src90;
235    auto dst90_ptr = dst90;
236    auto src91_ptr = src91;
237    auto dst91_ptr = dst91;
238    auto src92_ptr = src92;
239    auto dst92_ptr = dst92;
240    auto src93_ptr = src93;
241    auto dst93_ptr = dst93;
242    auto src94_ptr = src94;
243    auto dst94_ptr = dst94;
244    auto src95_ptr = src95;
245    auto dst95_ptr = dst95;
246    auto src96_ptr = src96;
247    auto dst96_ptr = dst96;
248    auto src97_ptr = src97;
249    auto dst97_ptr = dst97;
250    auto src98_ptr = src98;
251    auto dst98_ptr = dst98;
252    auto src99_ptr = src99;
253    auto dst99_ptr = dst99;
254 }
255
```

## Flexible parallel reduction

Take a look at scalar product:

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

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

```
1 auto fetch = [=] __cuda_callable__ (int i)->float { return a[i]*b[i]; };
2 auto reduction = [] __cuda_callable__ (float x, float y) -> float { return x+y; };
3
4 float result( 0.0 );
5 for( int i = 0; i < size; i++ )
6     reduction = reduction( result, fetch( i ) );
```

## Flexible parallel reduction

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.

## Flexible parallel reduction

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 );
```

## Flexible parallel reduction

Largest element in absolute value can be evaluated as follows:

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

Or more compact way:

```
1 auto a_view = a.getView();
2 result = TNL::Algorithms::reduce< Device >( 0, a.getSize(),
3     [=] (int i)->float { return abs( a_view[ i ] ); },
4     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() );
```



## 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:

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

## Solving the heat equation

```
1  #include <TNL/Containers/Vector.h>
2  #include <TNL/Algorithms/Reduce.h>
3  #include <TNL/Algorithms/ParallelFor.h>
4
5  using Device = TNL::Devices::Host;
6
7  int main( int argc, char* argv[] )
8  {
9      using Vector = TNL::Containers::Vector< float, Device >;
10
11     /////
12     // Parameters of the discretization
13     int size = 11;
14     float h = 1.0 / ( size - 1 );
15     float tau = 0.1 * h * h;
16     float final_T = 1.0;
17     float h_inv_sqr = 1.0 / ( h * h );
```

## Solving the heat equation

```
18  /////
19  // Allocation of the grid functions
20  Vector u( size * size, 0.0 ), aux( size * size, 0.0 );
21  auto u_view = u.getView();
22  TNL::Algorithms::ParallelFor2D< Device >::exec( 0, 0, size, size,
23  [=] __cuda_callable__ ( int i, int j ) mutable {
24      float x = i * h - 0.5;
25      float y = j * h - 0.5;
26      int idx = j * size + i;
27      if( x * x + y * y < 0.25 )
28          u_view[ idx ] = 1.0; } );
```

## Solving the heat equation

```
29  /////  
30  // Time loop  
31  float t = 0.0;  
32  float residue = 10.0;  
33  while( t < final_T && residue > 1.0e-6 )  
34  {  
35      /////  
36      // Update with FDM approximation of the Laplace operator  
37      auto u_view = u.getView();  
38      auto aux_view = aux.getView();
```

## Solving the heat equation

```
39     residue = sqrt( TNL::Algorithms::reduce< Device >( 0, size * size,
40         [=] __cuda_callable__ ( int idx ) mutable -> float {
41             int i = idx % size;
42             int j = idx / size;
43             if( i == 0 || j == 0 || i == size - 1 || j == size - 1 )
44                 return 0.0;
45             float update = h_inv_sqr * (
46                 u_view[ idx - size ] + u_view[ idx - 1 ] +
47                 u_view[ idx + size ] + u_view[ idx + 1 ]
48                 - 4.0 * u_view[ idx ] );
49             aux_view[ idx ] = u_view[ idx ] + tau * update;
50             return update * update;
51         },
52         TNL::Plus() ) );
```

## Solving the heat equation

```
53     t += tau;
54     u.swap( aux );
55     std::cout << "t = " << t << " residue = " << residue << std::endl;
56 }
57 return EXIT_SUCCESS;
58 }
```

## 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