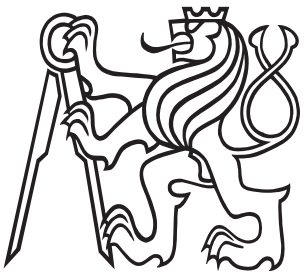
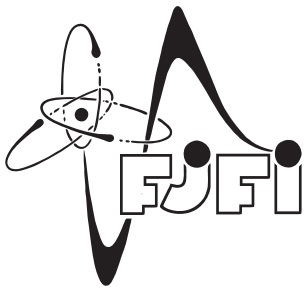
Programming GPU using TNL

Tomáš Oberhuber Jakub Klinkovský Radek Fučík
Aleš Wodecki

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





WSC 2019

Why GPU?

	Nvidia V100	Intel Xeon E5-4660
Cores	5120 @ 1.3GHz	16 @ 3.0GHz
Peak perf.	15.7/7.8 TFlops	0.4 / 0.2 TFlops
Max. RAM	32 GB	1.5 TB
Memory bw.	900 GB/s	68 GB/s
TDP	300 W	120 W

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

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
- www.tnl-project.org
- $\approx$ 300k lines of templated code
- documentation for few basic structures
- MIT license

Arrays

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 ...

```
1  Array< float, Devices::Cuda, int > a( 100 );
2  auto lamda = [] __cuda_callable__ ( int elementIdx ) {
3         return elementIdx%5; }
4  a.evaluate( lambda );
```

Vectors

Vectors add algebraic operations to arrays:

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

Vector and Array View

- arrays and vectors supports data sharing
- both are relatively complex structures
- TNL uses also lightweight counterparts `ArrayView`, `VectorView`
- both can be passed efficiently on GPU for example
- neither perform dynamic memory allocation/deallocation or deep copies

```
1  Vector< float, Devices::Cuda, int > v( 100 );
2  VectorView< float, Devices::Cuda, int > view( v );
```

Parallel reduction

Parallel reduction is operation taking 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 = 150 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 1024:
            cudaBlockReduction6< 1024 >
              <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
              ( currentSize, reductionInput, deviceAux1 );
            break;
         case 512:
            cudaBlockReduction6< 512 >
              <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
              ( currentSize, reductionInput, deviceAux1 );
            break;
         case 256:
            cudaBlockReduction6< 256 >
              <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
              ( currentSize, reductionInput, deviceAux1 );
            break;
         case 128:
            cudaBlockReduction6< 128 >
              <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
              ( currentSize, reductionInput, deviceAux1 );
            break;
         case 64:
            cudaBlockReduction6< 64 >
              <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
              ( currentSize, reductionInput, deviceAux1 );
            break;
         case 32:
            cudaBlockReduction6< 32 >
              <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
              ( currentSize, reductionInput, deviceAux1 );
            break;
         case 16:
            cudaBlockReduction6< 16 >
              <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
              ( currentSize, reductionInput, deviceAux1 );
            break;
         case 8:
            cudaBlockReduction6< 8 >
              <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
              ( currentSize, reductionInput, deviceAux1 );
            break;
         case 4:
            cudaBlockReduction6< 4 >
              <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
              ( currentSize, reductionInput, deviceAux1 );
            break;
         case 2:
            cudaBlockReduction6< 2 >
              <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
              ( currentSize, reductionInput, deviceAux1 );
            break;
         case 1:
            cudaBlockReduction6< 1 >
              <<< cudaGridSize, cudaBlockSize, sharedMemory >>>
              ( currentSize, reductionInput, deviceAux1 );
            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;
}
```

Parallel reduction in TNL

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  float a[ size ], b[size ];
2
3  ...
4
5  auto fetch = [=] (int i)->float {
6    return a[i]*b[i]; };
7  auto reduce = [] (float& x, const float& y) {
8    x += y; };
9
10 float result( 0.0 );
11 for( int i = 0; i < size; i++ )
12     reduce( result, fetch( i ) );
```

Parallel reduction in TNL

Another example - l_p-norm:

```
1  const float p = 2.0;
2  float a[ size ];
3
4  auto fetch = [=] (int i)->float {
5          return pow( fabs( a[i] ), p ); };
6  auto reduce = [] (float& x, const float& y) {
7     x += y; };
8
9  float result( 0.0 );
10 for( int i = 0; i < size; i++ )
11     reduce( result, fetch( i ) );
```

Parallel reduction in TNL

Another example - arrays comparison:

```
1  bool zero = true;
2  const float p = 2.0;
3  float a[size], b[size];
4  ...
5  auto fetch = [=] (int i)->bool {
6     return ( a[i] == b[i] ); };
7  auto reduce = [] (float& x, const float& y) {
8     x = x && y; };
9
10 float result( zero );
11 for( int i = 0; i < size; i++ )
12     reduce( result, fetch( i ) );
```

Parallel reduction in TNL

To perform the same on GPU in TNL just add `__cuda_callable__` to lambdas...

```
1  auto fetch = [=] __cuda_callable__ (int i)->bool {
2     return ( a[i] == b[i] ); };
3  auto reduce = [] __cuda_callable__ (float& x,
4                                       const float& y) {
5     x = x && y; };
```

... and for certain reasons, deliver `volatile` version of `reduce`:

```
1  auto volatileReduce = [] __cuda_callable__ (
2     volatile float& x,
3     volatile const float& y) {
4     x = x && y; };
```

This could be avoided when CUDA compiler supports C++17 better. Now call

```
1  Reduction< Devices::Cuda >::reduce( size, reduce,
2     volatileReduce, fetch, zero );
```

Expression Templates in TNL

Expression

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

can be evaluated in C as follows:

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

It is:

- efficient
- relatively simple
- works only on CPU - sequentially

Expression Templates in TNL

We can use operators overloading in C++:

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

- it is very simple and easy to read
- can be performed in parallel on multicore CPUs or GPUs
- it is inefficient

Expression Templates in TNL

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

The code is at the end performed almost like this:

```
1  Vector tmp1( N ), tmp2( N ), tmp3( N );
2  for( int i = 0; i < N; i++ )
3     tmp1[ i ] = 2 * b[ i ];
4  for( int i = 0; i < N; i++ )
5     tmp2[ i ] = 3 * c[ i ];
6  for( int i = 0; i < N; i++ )
7     tmp3[ i ] = tmp1[ i ] + tmp2[ i ];
8  for( int i = 0; i < N; i++ )
9     x[ i ] = a[ i ] + tmp3[ i ];
```

Expression Templates in TNL

We can use BLAS/cuBLAS:

```
1  cublasHandle_t handle;
2  cublasSaxpy( handle, N, 1.0, a, 1, x, 1 );
3  cublasSaxpy( handle, N, 2.0, b, 1, x, 1 );
4  cublasSaxpy( handle, N, 3.0, c, 1, x, 1 );
```

- it is pretty hard to read
- works only for single precision
- more efficient than C++ version but still less efficient than C version

Expression Templates in TNL

The code is at the end performed almost like this:

```
1  for( int i = 0; i < tmp.size(); i++ )
2     x[ i ] = a[ i ];
3  for( int i = 0; i < tmp.size(); i++ )
4     x[ i ] += 2 * b[ i ];
5  for( int i = 0; i < tmp.size(); i++ )
6     x[ i ] += 3 * c[ i ];
```

Expression Templates in TNL

	Memory	Operations	Conditions
C-like	$4N$	$5N$	N
C++	$7N$	$8N$	$4N$
Blas	$4N$	$5N$	$3N$

Expression Templates in TNL

Expression templates take the formula...

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

... and parse it into a form of C++ typ:

```
1   Addition<
2      Vector,
3      Addition<
4         Multiplication< double, Vector >,
5         Multiplication< double, Vector > > > expr;
```

The expression is at the end evaluated like this:

```
1   for( int i = 0; i < N; i++ )
2      x[ i ] = expr[ i ];
```

- it is simple and easy to read
- works for any type `Real` (`float`/`double`) and any Device (CPU/GPU)
- it is very efficient

Expression Templates in TNL

Expression templates take the formula...

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

... and parse it into a form of C++ typ:

```
1   Addition<
2      Vector,
3      Addition<
4         Multiplication< double, Vector >,
5         Multiplication< double, Vector > > > expr;
```

The expression is at the end evaluated like this:

```
1   for( int i = 0; i < N; i++ )
2      x[ i ] = expr[ i ];
```

- it is simple and easy to read
- works for any type `Real` (`float`/`double`) and any Device (CPU/GPU)
- it is very efficient

Expression Templates in TNL

	Memory	Operations	Conditions
C-like	$4N$	$5N$	N
C++	$7N$	$8N$	$4N$
Blas	$4N$	$5N$	$3N$
ET	$4N$	$5N$	N

Expression Templates & Parallel Reduction in TNL

Example:

```
1  using Vector = Vector< float, Devices::Cuda, int >;
2  using View = VectorView< float, Devices::Cuda, int >;
3  Vector av( 100 ), bv( 100 ), cv( 100 ), dv( 100 );
4  View a( av ), b( bv ), c( cv ), d( dv );
5  ...
6  float scalarProduct = ( a, b + 3 * c );
7  d = a + b * c + sin( d );
8  a = min( b, c );
9  float min_a = min( a );
10 float total_min = min( min( a, b ) );
```

Performance comparison

Performance was tested on:

- GPU Nvidia P100
  - 16 GB HBM2 @ 732 GB/s
  - 3584 CUDA cores, 4.7 TFlops in double precision
- CPU
  - AMD Ryzen 5 2600, 8MB L3 cache

Expression Templates in TNL

Scalar product: `r = ( x, y )`.

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	20.8	2.7	0.1	49.3	69.9	1.41
200k	18.5	12.2	0.6	90.1	108.6	1.20
400k	18.4	13.3	0.7	142.2	159.1	1.11
800k	11.9	13.2	1.1	207.4	233.4	1.12
1.6M	13.6	15.6	1.1	313.6	333.3	1.06
3.2M	14.9	17.9	1.2	381.0	403.7	1.05
6.4M	16.7	17.0	1.0	417.1	431.8	1.03

Expression Templates in TNL

Vector addition: `x += a`.

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	46.0	11.6	0.2	152.2	174.8	1.14
200k	42.3	7.0	0.1	196.6	216.1	1.09
400k	16.9	29.3	1.7	277.6	294.4	1.06
800k	16.8	23.9	1.4	326.2	333.6	1.02
1.6M	17.3	25.1	1.4	362.5	374.2	1.03
3.2M	17.5	25.3	1.4	422.4	436.8	1.03
6.4M	17.4	25.7	1.4	456.6	469.8	1.02

Expression Templates in TNL

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

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	30.3	29.3	0.9	188.3	190.9	1.01
200k	30.5	31.8	1.0	218.0	230.7	1.05
400k	13.7	32.8	2.3	243.1	305.9	1.25
800k	11.6	23.0	1.9	263.8	353.0	1.33
1.6M	11.7	24.4	2.0	285.9	389.4	1.36
3.2M	11.7	24.8	2.1	312.9	442.8	1.41
6.4M	11.7	25.7	2.1	327.3	471.9	1.44

Expression Templates in TNL

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

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	25.4	7.7	0.30	194.7	236.5	1.21
200k	23.7	16.1	0.67	228.3	277.6	1.21
400k	13.0	31.0	2.38	218.3	330.9	1.51
800k	10.0	24.5	2.45	233.3	370.6	1.58
1.6M	9.9	23.6	2.38	249.6	403.4	1.61
3.2M	9.8	25.2	2.57	266.6	444.8	1.66
6.4M	9.8	25.9	2.64	276.6	471.3	1.70

More about TNL ...

TNL is available at

`www.tnl-project.org`

under MIT license.