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?

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

\[ \approx 8,000 \] $
**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](http://www.tnl-project.org)
- ≈ 300k lines of templated code
- documentation for few basic structures
- MIT license
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 ...

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

```plaintext
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
Parallel reduction in TNL

Take a look at scalar product:

```c++
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:

```c++
float a[ size ], b[ size ];
...

auto fetch = [=] (int i)→float {
    return a[i]*b[i]; }

auto reduce = [] (float& x, const float& y) {
    x += y; }

float result( 0.0 );
for( int i = 0; i < size; i++ )
    reduce( result, fetch( i ) );
```
Another example - $l_p$-norm:

```cpp
const float p = 2.0;
float a[size];

auto fetch = [=] (int i) -> float {
    return pow(fabs(a[i]), p);
};

auto reduce = [] (float& x, const float& y) {
    x += y;
};

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

```cpp
auto fetch = [=] __cuda_callable__ (int i)->bool {
    return (a[i] == b[i]);
};

auto reduce = [] __cuda_callable__ (float& x,
     const float& y) {
    x = x && y;
};

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

```cpp
auto volatileReduce = [] __cuda_callable__ (volatile float& x,
     volatile const float& y) {
    x = x && y;
};
```

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

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

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

can be evaluated in C as follows:

```c
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
We can use operators overloading in C++:

\[ x = a + 2 * b + 3 * c; \]

- 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 \[ x = a + 2 \times b + 3 \times c; \]

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 \times b[ i ];
4 for( int i = 0; i < N; i++ )
5    tmp2[ i ] = 3 \times 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
The code is at the end performed almost like this:

```cpp
for(int i = 0; i < tmp.size(); i++ )
    x[i] = a[i];
for(int i = 0; i < tmp.size(); i++ )
    x[i] += 2 * b[i];
for(int i = 0; i < tmp.size(); i++ )
    x[i] += 3 * c[i];
```
### Expression Templates in TNL

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
<th>Operations</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-like</td>
<td>4N</td>
<td>5N</td>
<td>N</td>
</tr>
<tr>
<td>C++</td>
<td>7N</td>
<td>8N</td>
<td>4N</td>
</tr>
<tr>
<td>Blas</td>
<td>4N</td>
<td>5N</td>
<td>3N</td>
</tr>
</tbody>
</table>
Expression Templates in TNL

Expression templates take the formula...
\[ x = a + 2 \cdot b + 3 \cdot c; \]
...and parse it into a form of C++ typ:

```
Addition<int, Vector, Addition<int, Vector, Multiplication<double, Vector>>, Multiplication<double, Vector>> expr;
```

The expression is at the end evaluated like this:
```
for (int i = 0; i < N; i++)
    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  \( x = a + 2 \times b + 3 \times c; \)

... 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  \( \text{for ( int } i = 0; i < N; i++ ) } \)
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

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
<th>Operations</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-like</td>
<td>$4N$</td>
<td>$5N$</td>
<td>$N$</td>
</tr>
<tr>
<td>C++</td>
<td>$7N$</td>
<td>$8N$</td>
<td>$4N$</td>
</tr>
<tr>
<td>Blas</td>
<td>$4N$</td>
<td>$5N$</td>
<td>$3N$</td>
</tr>
<tr>
<td><strong>ET</strong></td>
<td>$4N$</td>
<td>$5N$</td>
<td>$N$</td>
</tr>
</tbody>
</table>
Example:

```cpp
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 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: $r = (x, y)$.

<table>
<thead>
<tr>
<th>Size</th>
<th>CPU</th>
<th></th>
<th></th>
<th>GPU</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BLAS</td>
<td>TNL</td>
<td>cuBLAS</td>
<td>TNL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BW</td>
<td>BW</td>
<td>Speed-up</td>
<td>BW</td>
<td>BW</td>
<td>Speed-up</td>
</tr>
<tr>
<td>100k</td>
<td>20.8</td>
<td>2.7</td>
<td>0.1</td>
<td>49.3</td>
<td>69.9</td>
<td>1.41</td>
</tr>
<tr>
<td>200k</td>
<td>18.5</td>
<td>12.2</td>
<td>0.6</td>
<td>90.1</td>
<td>108.6</td>
<td>1.20</td>
</tr>
<tr>
<td>400k</td>
<td>18.4</td>
<td>13.3</td>
<td>0.7</td>
<td>142.2</td>
<td>159.1</td>
<td>1.11</td>
</tr>
<tr>
<td>800k</td>
<td>11.9</td>
<td>13.2</td>
<td>1.1</td>
<td>207.4</td>
<td>233.4</td>
<td>1.12</td>
</tr>
<tr>
<td>1.6M</td>
<td>13.6</td>
<td>15.6</td>
<td>1.1</td>
<td>313.6</td>
<td>333.3</td>
<td>1.06</td>
</tr>
<tr>
<td>3.2M</td>
<td>14.9</td>
<td>17.9</td>
<td>1.2</td>
<td>381.0</td>
<td>403.7</td>
<td>1.05</td>
</tr>
<tr>
<td>6.4M</td>
<td>16.7</td>
<td>17.0</td>
<td>1.0</td>
<td>417.1</td>
<td>431.8</td>
<td>1.03</td>
</tr>
</tbody>
</table>
Vector addition: \( x += a \).

<table>
<thead>
<tr>
<th>Size</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLAS</td>
<td>TNL</td>
</tr>
<tr>
<td></td>
<td>BW</td>
<td>BW</td>
</tr>
<tr>
<td>100k</td>
<td>46.0</td>
<td>11.6</td>
</tr>
<tr>
<td>200k</td>
<td>42.3</td>
<td>7.0</td>
</tr>
<tr>
<td>400k</td>
<td>16.9</td>
<td>29.3</td>
</tr>
<tr>
<td>800k</td>
<td>16.8</td>
<td>23.9</td>
</tr>
<tr>
<td>1.6M</td>
<td>17.3</td>
<td>25.1</td>
</tr>
<tr>
<td>3.2M</td>
<td>17.5</td>
<td>25.3</td>
</tr>
<tr>
<td>6.4M</td>
<td>17.4</td>
<td>25.7</td>
</tr>
</tbody>
</table>
### Expression Templates in TNL

Vector addition: \( x += a + b \).

<table>
<thead>
<tr>
<th>Size</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLAS</td>
<td>TNL</td>
</tr>
<tr>
<td></td>
<td>BW</td>
<td>BW</td>
</tr>
<tr>
<td>100k</td>
<td>30.3</td>
<td>29.3</td>
</tr>
<tr>
<td>200k</td>
<td>30.5</td>
<td>31.8</td>
</tr>
<tr>
<td>400k</td>
<td>13.7</td>
<td>32.8</td>
</tr>
<tr>
<td>800k</td>
<td>11.6</td>
<td>23.0</td>
</tr>
<tr>
<td>1.6M</td>
<td>11.7</td>
<td>24.4</td>
</tr>
<tr>
<td>3.2M</td>
<td>11.7</td>
<td>24.8</td>
</tr>
<tr>
<td>6.4M</td>
<td>11.7</td>
<td>25.7</td>
</tr>
</tbody>
</table>
Expression Templates in TNL

Vector addition: \( x += a + b + c \).
More about TNL ...

TNL is available at

www.tnl-project.org

under MIT license.