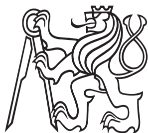
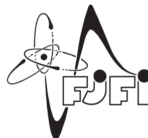
Modern trends in programming of GPUs

Tomáš Oberhuber Jakub Klinkovský Radek Fučík Pavel Eichler
Aleš Wodecki

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



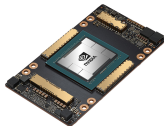


DAQFEET 2021

Outline

1. Why GPU?
2. Programming of GPUs
3. Template Numerical Library
4. Conclusion

Best HW available



	Nvidia A100	AMD Instinct MI100	AMD EPYC 2 Rome 7H12
Cores	6912 @ 1.41GHz	7680 @ 1.2GHz	64 @ 2.4GHz
Peak perf.	19.5/9.7 TFlops	23/11 TFlops	2 / 1 TFlops
Tensor peak. perf.	156 /– TFlops	–/–	–/–
Max. RAM	80 GB	32 GB	2 TB
Memory bw.	2039 GB/s	1200 GB/s	204 GB/s
TDP	400 W	300 W	280 W

CPU vs GPU

GPUs are more suitable for many algorithms.

- Machine learning (CNN) - 50x
- Computational Fluid Dynamics (LBM) - 50x - 100x
- Linear algebra (SpMV) - 10x
- Monte Carlo - 10x
- ...

CPU vs GPU

Unfortunately:

- the programmer must have good knowledge of the hardware
  - slow connection between CPU and GPU
  - memory optimised for transfer of large continuous blocks
  - GPU consists of several independent multiprocessors
- porting a code to GPUs often means rewriting the code from scratch
- lack of support in older HPC libraries

CUDA and software stack for Nvidia GPUs

CUDA = Compute Unified Device Architecture

- **nvcc** - C/C++ compiler
- **cuBLAS** - BLAS for GPUs
- **cuFFT** - Fast Fourrier transformation
- **CUDA Math Library** - mathematical functions
- **cuRAND** - random number generation
- **cuSOLVER** - dense and sparse direct solvers
- **cuSPARSE** - BLAS for sparse matrices
- **cuTENSOR** - tensor linear algebra
- **AmgX** - solver for PDEs
- **Thrust** - C++ library for parallel algorithms
- Deep learning libraries - **cuDNN**, **TensorRT**, **Jarvis**, ...
- Image libraries - **nvJPEG**, ...
- ...

ROCm and software stack for AMD GPUs

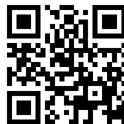
ROCm = Radeon Open Compute (open-source alternative to CUDA)

- **HIP** - C/C++ compiler, very simillar to CUDA, also for GPUs by Nvidia
- **rocBLAS**, **hipBLAS** - BLAS for GPUs
- **rocRAND** - random number generation
- **rocFFT** - Fast Fourrier transformation
- **rocSPARSE**, **hipSPARSE** - BLAS for sparse matrices
- **rocSOLVER** - dense and sparse direct solvers
- **rocALUTION** - sparse linear algebra
- **rocTHRUST** - C++ library for parallel algorithms
- Deep learning libraries - **TensorFlow**, **Pytorch**, **Caffe2**
- ...

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**
- MIT license



Arrays

Arrays are basic structures for memory management

- `TNL::Array< ElementType, DeviceType, IndexType, Allocator >`
- `DeviceType` says where the array resides
  - `TNL::Devices::Host` for CPU
  - `TNL::Devices::Cuda` for CUDA GPUs
  - (`TNL::Devices::Hip` for CUDA/ROCm GPUs - very experimental)
- `Allocator` performs the memory allocation
  - common CPU and GPU memory allocators
  - page-locked memory allocator
  - CUDA Unified Memory allocator
- I/O operations, elements manipulation ...

Parallel reduction

Parallel reduction is an 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

```
float sum( 0.0 )
for( int i = 0; i < size; i++ )
    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;
}
```

Flexible parallel reduction in TNL

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

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

Flexible parallel reduction in TNL

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

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

Now call

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

Calling

```
Reduction< Devices::Host >::reduce( size,reduce,fetch,zero );
```

performs the same on CPU.

Flexible parallel reduction in TNL

Max norm:

```
auto fetch = [=] __cuda_callable__ (int i)->float {
        return TNL::abs( a[i] ); };
auto reduce = [] __cuda_callable__ (float x, float y)->float {
        return TNL::max( x, y ); };
```

Vectors comparison:

```
auto fetch = [=] __cuda_callable__ (int i)->bool {
        return ( a[i] == b[i] ); };
auto reduce = [] __cuda_callable__ (bool x, bool y) -> bool {
        return x && y; };
```

Flexible parallel reduction in TNL

Count occurencies:

```
auto fetch = [=] __cuda_callable__ (int i)->int {
        return ( a[i] == 0 ); };
auto reduce = [] __cuda_callable__ (int x, int y)->int {
        return x + y; };
```

Vector addition and L_2 norm:

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

Algebraic vector expressions

Consider algebraic vector expressions like this:

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

We can use BLAS/cuBLAS:

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

- it is pretty hard to read
- works only for single precision
- not very efficient - requires call of three CUDA kernels

Expression Templates in TNL

In TNL, algebraic vector expressions are handled by **expression templates**:

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

- simple
- works for both CPU and GPU
- efficient
  - **one** loop on CPU
  - specialized **one** CUDA kernel for each expression on GPU

Expression Templates & Parallel Reduction in TNL

Example:

```
using Vector = Vector< float, Devices::Cuda, int >;
Vector a( 100 ), b( 100 ), c( 100 ), d( 100 );
...
float scalarProduct = ( a, b + 3 * c );
d = a + b * c + sin( d );
a = min( b, c );
float min_a = min( a );
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
  - Intel Core i7-5820K, 3.3GHz, 16MB 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	17.4	5.0	**0.3**	49.3	69.9	**1.41**
200k	17.4	5.0	**0.3**	90.1	108.6	**1.20**
400k	17.7	5.0	**0.3**	142.2	159.1	**1.11**
800k	13.7	4.8	**0.3**	207.4	233.4	**1.12**
1.6M	12.6	4.8	**0.4**	313.6	333.3	**1.06**
3.2M	12.8	4.6	**0.4**	381.0	403.7	**1.05**
6.4M	12.7	4.6	**0.4**	417.1	431.8	**1.03**

BW = effective memory bandwidth in GB/s

Expression Templates in TNL

Vector addition: `x += a`.

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	34.0	42.0	**1.2**	152.2	174.8	**1.14**
200k	35.1	43.0	**1.2**	196.6	216.1	**1.09**
400k	31.7	36.7	**1.2**	277.6	294.4	**1.06**
800k	19.7	19.3	**0.97**	326.2	333.6	**1.02**
1.6M	18.1	17.3	**0.95**	362.5	374.2	**1.03**
3.2M	18.4	17.6	**0.95**	422.4	436.8	**1.03**
6.4M	18.3	17.5	**0.95**	456.6	469.8	**1.02**

BW = effective memory bandwidth in GB/s

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	23.6	42.0	**1.8**	188.3	190.9	**1.01**
200k	23.5	41.8	**1.8**	218.0	230.7	**1.05**
400k	20.9	37.4	**1.8**	243.1	305.9	**1.25**
800k	13.7	18.7	**1.4**	263.8	353.0	**1.33**
1.6M	12.2	16.8	**1.4**	285.9	389.4	**1.36**
3.2M	12.3	17.5	**1.4**	312.9	442.8	**1.41**
6.4M	12.2	17.3	**1.4**	327.3	471.9	**1.44**

BW = effective memory bandwidth in GB/s

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	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

Conclusion

- Lambda functions and template specializations in C++ are great tools for GPU programming.
- Developers implement highly-optimised skeletons of parallel algorithms.
- Physicists/mathematicians add (simple) lambda functions.
- We obtain specialised GPU kernels for given tasks $\Rightarrow$ better performance.

Conclusion

Current and future work in TNL development:

- extension of flexible reduction to sparse matrices and simillar structures
- support of MPI for GPU clusters
- unstructured numerical meshes
- hashing and B-trees
- interface for Julia (julialang.org)

Conclusion

Thank you for your attentation.