Lecture: Deep Learning and Differential Programming

3.3 GPUs - Software

https://liris.cnrs.fr/christian.wolf/teaching

INSA LYON Christian Wolf

APIs

GPUs can be programmed on multiple different levels of abstraction.

We will (very briefly) study two cases:

CUDA/C++: direct low-level GPU programming

```
__global__
void mult_kernel_simple(int mxWidth, float *mx1, float *mx2, float *output)
{
    int c = blockIdx.x*blockDim.x + threadIdx.x;
```

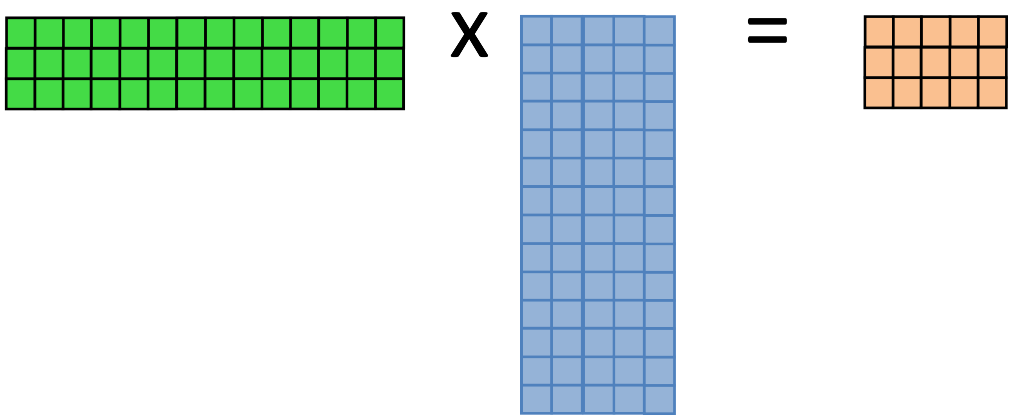
PyTorch/Python: high-level of abstraction

```
model = LeNet().to("cuda:0")

for batch_idx, (data, labels) in enumerate(train_loader):
    data = data.to("cuda:0")
```

1 CUDA – low level GPU programmng

2 GPUs & PyTorch

Example : matrix multiplication



The classical sequential solution:

```
for( int r = 0; r < h; r++)
{
    for( int c = 0; c < w; c++)
    {
        m1xm2[r*w + c] = 0;

        for( int k = 0; k < w; k++)
            m1xm2[r*w + c] += m1[r*w + k] * m2[k*w + c];
    }
}
```

The parallel solution

Parallel execution of a function called for each individual result value of the result matrix, with arguments being the indices of the value.

```
void mult_kernel_simple(int c, int r)
{
    output[r*mxWidth + c] = 0.0f;
    for( int k = 0; k < mxWidth; k++)
        output[r*mxWidth + c] += mx1[r*mxWidth + k] * mx2[k*mxWidth + c];
}
```

Called in parallel for all c,r

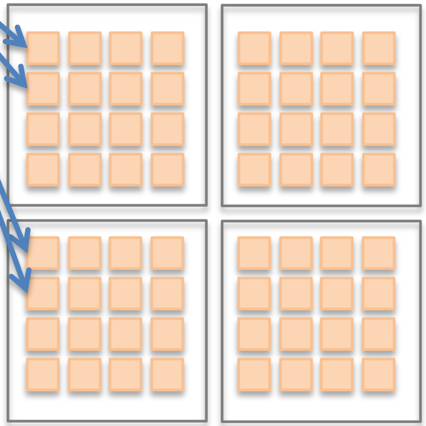
In CUDA, this function is called a *kernel.*

Each tuple (c,r) corresponds to a *thread.*

Organisation into *blocks*

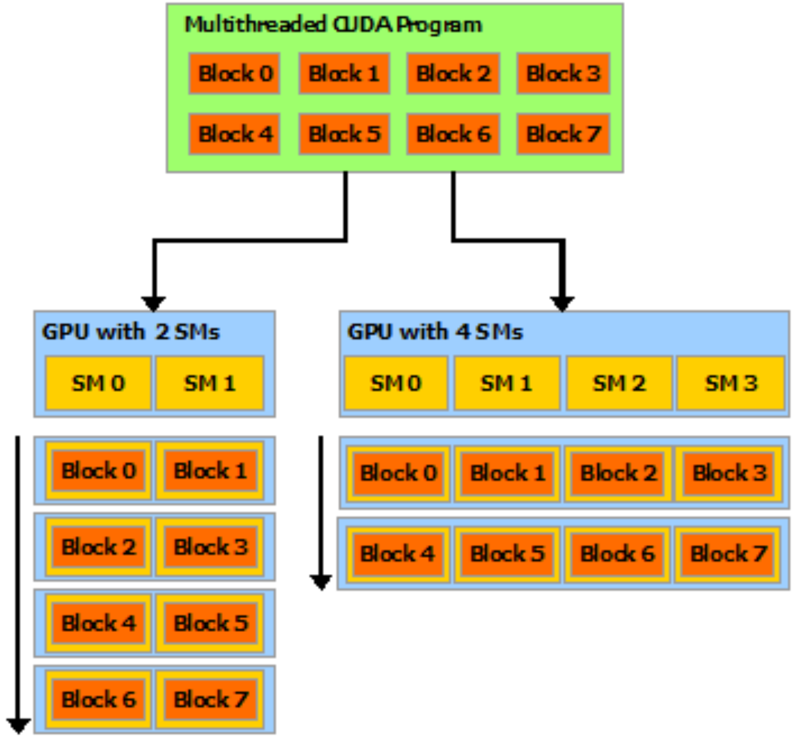
- Threads are organized into *blocks*
- Faster local memory can be shared by threads of the same block.

```
void mult_kernel_simple(int c, int r)
{
    output[r*mxWidth + c] = 0.0f;
    for( int k = 0; k < mxWidth; k++)
        output[r*mxWidth + c] += mx1[r*mxWidth + k] * mx2[k*mxWidth + c];
}
```



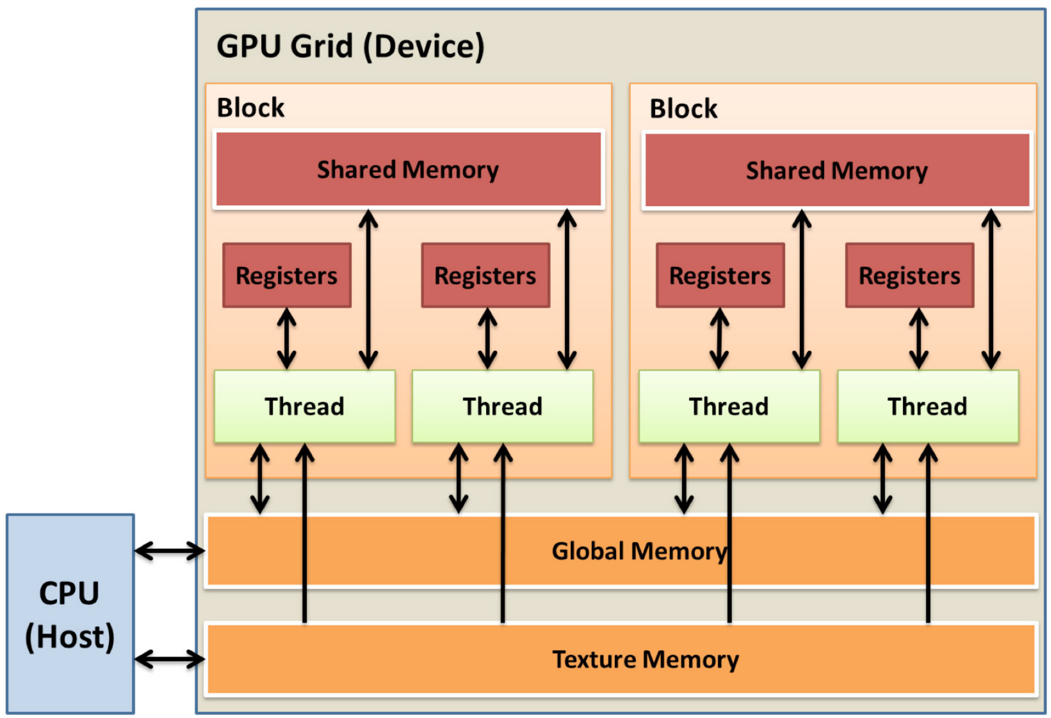
SM

A block will be sent to a SM (*streaming multi-processor)*



The classical CUDA sequence

- The CPU allocates memory on the GPU
- The CPU copies the data to GPU global memory
- The CPU launches the kernel on the GPU
- The GPU executes the kernel in parallel
- The CPU copies the result data back to CPU host memory



The CUDA syntax of the kernel

Key word declares the *kernel*

```
__global__
void mult_kernel_simple(int mxWidth, float *mx1, float *mx2, float *output)
{
    int c = blockIdx.x*blockDim.x + threadIdx.x;
    int r = blockIdx.y*blockDim.y + threadIdx.y;

    output[r*mxWidth + c] = 0.0;
    for( int k = 0; k < mxWidth; k++)
        output[r*mxWidth + c] += mx1[r*mxWidth + k] * mx2[k*mxWidth + c];
}
```

Index of the block in the grid

Index of the thread in the block

Calling the kernel

```
int main(void)
{
    ...
    // allocate memory on GPU
    cudaMalloc( (void**) &gpuMatrix1, matrixSizeInBytes);
    ...

    // copy data from CPU memory to GPU memory
    cudaMemcpy(gpuMatrix1, matrix1, matrixSizeInBytes, cudaMemcpyHostToDevice);
    ...

    // Set grid and block size
    dim3 dimBlock(32, 32);
    dim3 dimGrid(matrixWidth/dimBlock.x, matrixWidth/dimBlock.y);

    // run kernel
    mult_kernel_simple<<<dimGrid, dimBlock>>>( matrixWidth, gpuMatrix1, gpuMatrix2, gpuO

    // copy back results from GPU memory to CPU memory
    cudaMemcpy( outputData, gpuOutput, matrixSizeInBytes, cudaMemcpyDeviceToHost);
    ...
}
```

Call the *kernel* in parallel for a set of threads

Compilation

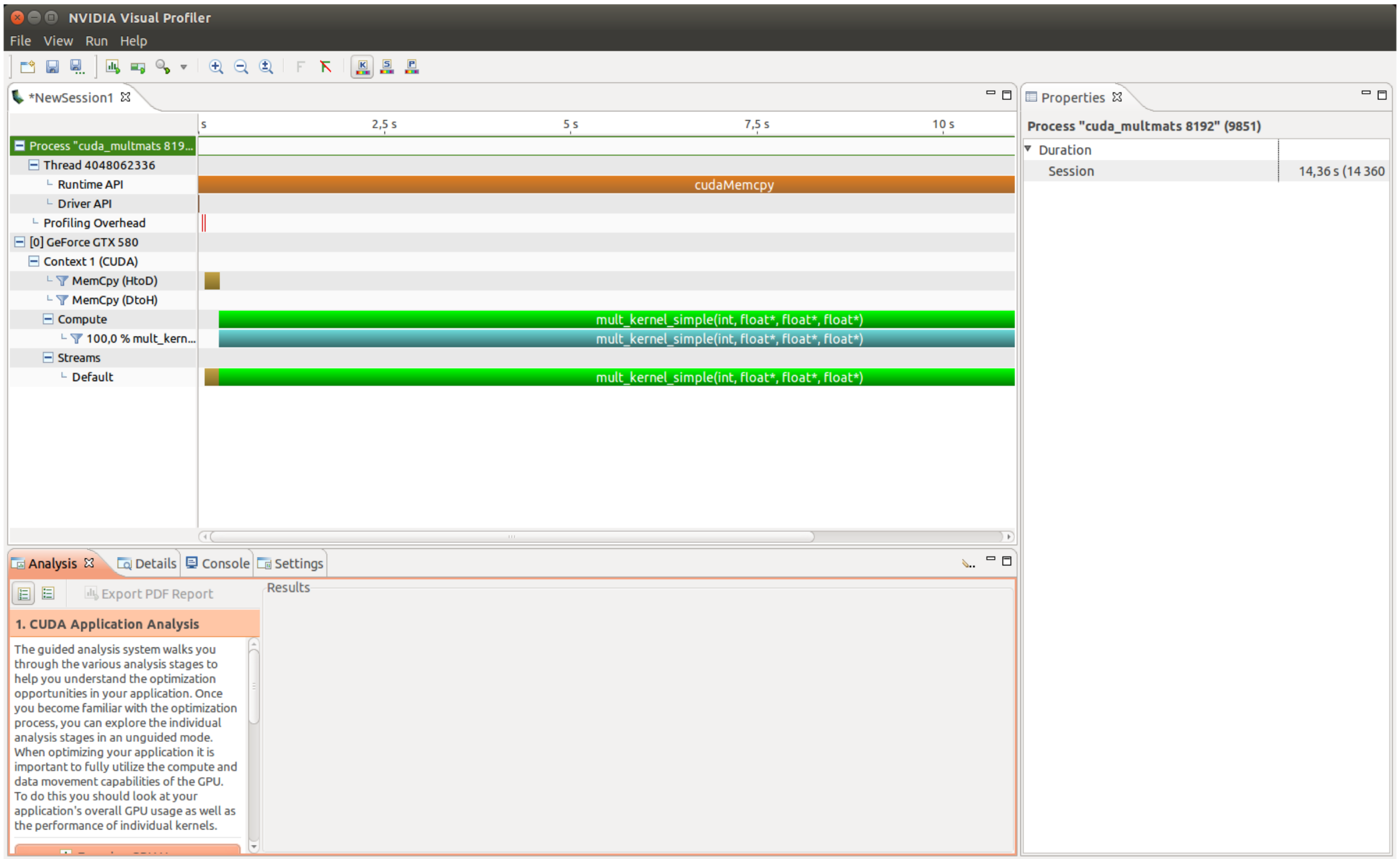
CUDA uses a specific compiler, which is based on a generic C++ compiler (gcc)

```
g++ -c  -O3 -Wall file_seqcode.cpp -o file_seqcode.o

nvcc -c -O3 file_parcode.cu -o file_parcode.o

g++ file_seqcode.o file_parcode.o -o executable
```

Classical C++ code   CUDA code   Linking

Debugging and profiling

The nvvp profiler is part of the CUDA toolkit.



1 CUDA – low level GPU programmng

2 GPUs & PyTorch

The PyTorch GPU interface

The model is transferred to GPU memory with `.to(device)`
Device can be "cpu", "cuda:0", "cuda:1" etc.

```
model = LeNet()
model = model.to("cuda:0")
```

We also send the data to GPU memory.
We get data back to the CPU with the `.cpu()` method:

```
# Cycle through batches
for idx, (data,labels) in enumerate(train_loader):
  data = data.to("cuda:0")
  optimizer.zero_grad()
  y = model(data)
  loss = crossentropy(y, labels)
  loss.backward()
  running_loss += loss.cpu().item()
  optimizer.step()

  _, predicted = torch.max(y.data.cpu(), 1)
```

PyTorch vs. Cuda

- For standard functions (Linear, Conv2d, Pooling etc.), PyTorch ships GPU support.
- PyTorch allows to write custom neural network layers (requiring to specify the forward and and the backward pass)
- Custom layers require CUDA programming to run on GPUs.