GPU Course follow-up

Ben van Werkhoven

Topics for today

- Host code
  - What is host code?
  - Practical info on CUDA Runtime
  - Overlapping Computation and Communication

Download the slides!

- Get your own copy of the slides so you can read along and click on links
  See: https://github.com/benvanwerkhoven/gpu-course/

- My slides are sometimes very wordy, this is intentional, so they may serve as a reference that you can read again later

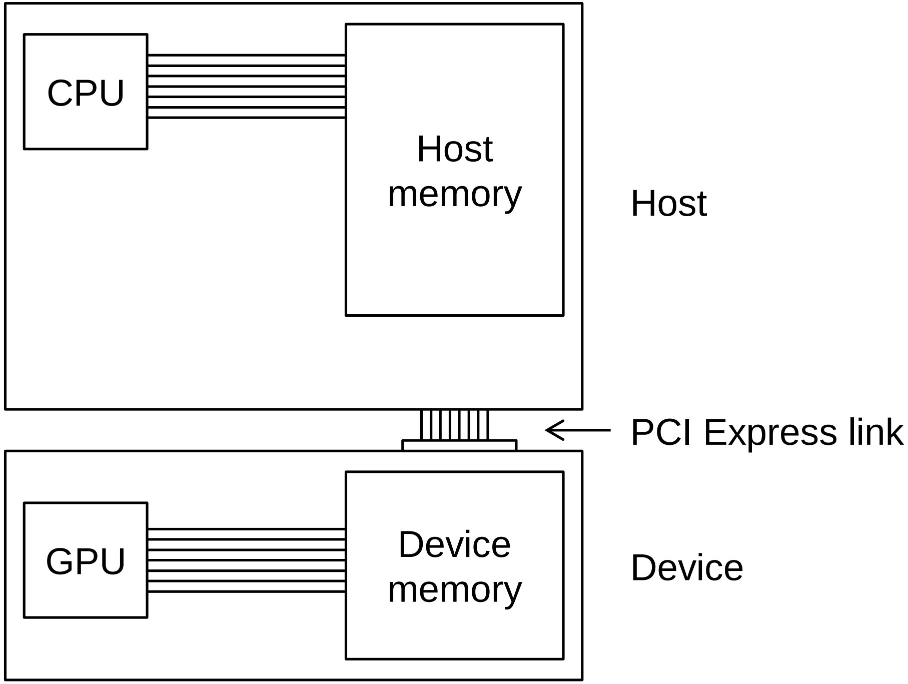
- As requested, the goal for today is to provide a lot of very practical information that will help to get you started with writing CUDA programs

- In code samples in the slides I sometimes leave out '{' and '}' and the checks for return values of called API functions to save space

Host Code

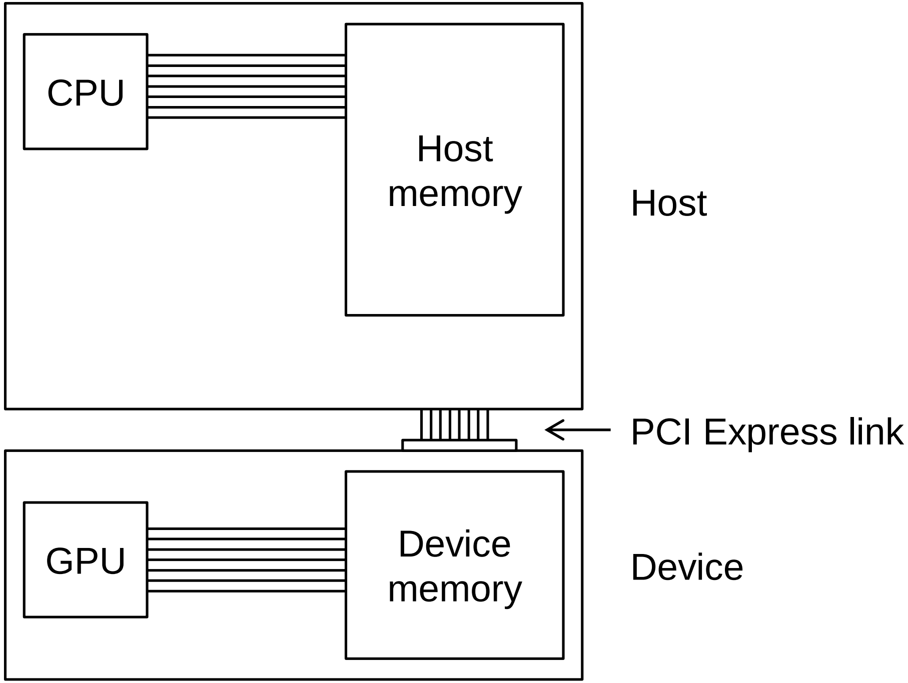
What is host code?

- GPU Programs consist of 2 parts:
  - Host code (C/C++/Python/Java/…)
  - Device code (CUDA/OpenCL)
- Host code runs on the CPU and uses (for example) the CUDA Runtime API to steer the GPU Computations
- Host code is responsible for:
  - GPU memory management
  - Transferring data between host and device
  - GPU kernel launches



Host-Device Parallelism

- The host (CPU) and device (GPU) are independent processors, they may execute their programs in parallel
- Some of the things you do in GPU programs are asynchronous with respect to the host, such as kernel launches
- You can again synchronize the host and device by making the host wait for all operation on the device to be completed



Practical info on CUDA

- The complete CUDA documentation is here:
  - http://docs.nvidia.com/cuda/
- The CUDA Programming guide is here:
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/
- The CUDA Runtime API can be found here:
  - http://docs.nvidia.com/cuda/cuda-runtime-api/index.html

Error handling

- **CUDA Runtime API functions return a** `cudaError_t` **value that you can pass to**
  `const char* cudaGetErrorString ( cudaError_t error )`
  to get an error message string

- **Kernel launches don't return anything, so you can use:**
  `cudaError_t cudaGetLastError ( void )`
  to retrieve the last error

- **Always check the return value for every call to the CUDA Runtime API**

Memory Management

- **GPU Memory:**
  - `cudaError_t cudaMalloc ( void** devPtr, size_t size )`
  - `cudaError_t cudaFree ( void* devPtr )`

- **Host memory:**
  - `cudaError_t cudaHostAlloc ( void** pHost, size_t size, unsigned int  flags )`
    allocates page-locked (pinned) memory on the host
    - **flag:** `cudaHostAllocMapped` **Maps the allocation into the CUDA address space**
    - `cudaSetDeviceFlags()` **must have been called at the start of the program with the** `cudaDeviceMapHost` **flag in order for the** `cudaHostAllocMapped` **flag to have any effect**
  - `cudaError_t cudaFreeHost ( void* ptr )`
    frees memory that must have been allocated by `cudaHostAlloc()`

Data transfers

- `cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count, cudaMemcpyKind kind )`
  Copies memory from src to dst, count is in bytes

- `cudaMemcpyKind` **specifies the direction of the copy**
  - `cudaMemcpyHostToHost`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

- **This is the most common way to transfer data between host and device memory**
  - **(in addition to** `cudaMemcpyToSymbol` **for copying to constant memory)**

Starting a kernel

- **The host program sets the number of threads and thread blocks when it launches the kernel**

- ```
  //create variables to hold grid and thread block dimensions
  dim3 threads(x, y, z)
  dim3 grid(x, y)

  //launch the kernel
  vector_add<<<grid, threads>>>(c, a, b);

  //wait for the kernel to complete
  cudaDeviceSynchronize();
  ```
  ```

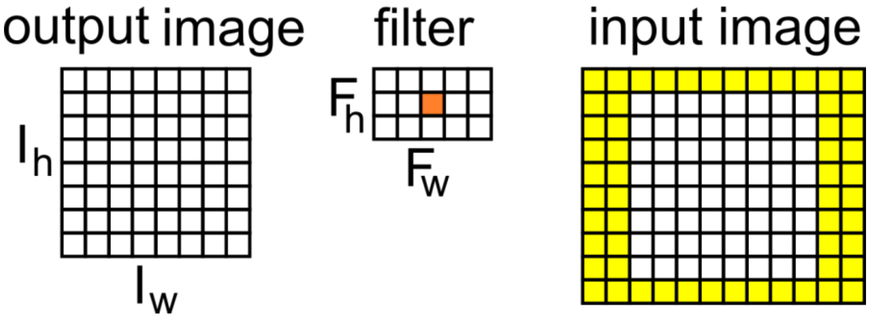
First hands-on session

- **Remember 2D Convolution:**

```
//for each pixel in the output image
for (y=0; y < image_height; y++) {
 for (x=0; x < image_width; x++) {

  //for each filter weight
  for (i=0; i < filter_height; i++) {
   for (j=0; j < filter_width; j++) {
    output[y][x] += input[y+i][x+j] * filter[i][j];

}}}}
```



First hands-on session

- **Login on the DAS5 (type or add to .bashrc):**
  - `module load cuda75 slurm`
  - `alias gpurun="srun -N 1 -C TitanX --gres=gpu:1 numactl --physcpubind=0"`
- **Checkout the repository: https://github.com/benvanwerkhoven/gpu-course**
- **Change to the directory overlap**
- **Compile by typing** `make`**, run by typing** `gpurun ./conv`

- **Make sure you understand everything in the code, if not ask me!**

- **Write the host code in** `convolution2d_explicit()` **by following the directions in the comments**

Asynchronous calls

- **Remember the host and device may run independently**

- `cudaError_t cudaMemcpyAsync ( void* dst, const void* src, size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0 )`
  - **Like a normal cudaMemcpy, but notice the additional stream argument**
  - **If you put a 0 there, the call will happen in the 'default stream' like with regular cudaMemcpy() calls**
  - **If the host memory pointer is not in page-locked (or pinned) memory this call will not happen asynchronously**
  - **This call is asynchronous with respect to the host, in that this function may return before the copy is complete**
  - **Only if you pass a non-zero stream argument this call may overlap with operations in different streams**

CUDA Streams

- Streams are sequences of commands that are sequentially consistent:
  - Commands issued in a stream will happen one after the other
  - Commands issued in different streams may execute out of order with respect to one another or concurrently
- Example use:

```
cudaStream_t my_stream;                                //placeholder for stream id
cudaStreamCreate(&my_stream);                          //create a stream
//issue commands to my_stream
my_kernel<<<grid, threads, 0, my_stream>>>(...);       //kernel launch in a stream
cudaStreamDestroy(my_stream);                          //cleanup the stream
```

- Also see:
  - cudaError_t cudaStreamCreate (cudaStream_t* pStream)
  - cudaError_t cudaStreamDestroy (cudaStream_t stream)

Synchronization

- **You may want to synchronize the host and device programs, for example to wait for the results of a computation on the GPU**

- `cudaError_t cudaDeviceSynchronize ( void )`
  forces the host to wait for all pending operations on the device to be completed

- `cudaError_t cudaStreamSynchronize ( cudaStream_t stream )`
  forces the host to wait for all pending operations in the given stream to be completed

Streams Example

```
cudaStream_t stream[2];
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);

cudaMemcpyAsync(d_A, h_A, A_size,
        cudaMemcpyHostToDevice, stream[0]);
cudaMemcpyAsync(d_B, h_B, B_size,
        cudaMemcpyHostToDevice, stream[1]);
kernel_A<<<grid, threads, 0, stream[0]>>>(d_A);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_B);

cudaDeviceSynchronize();

for (i=0; i<2; i++)
    cudaStreamDestroy(stream[i]);
```

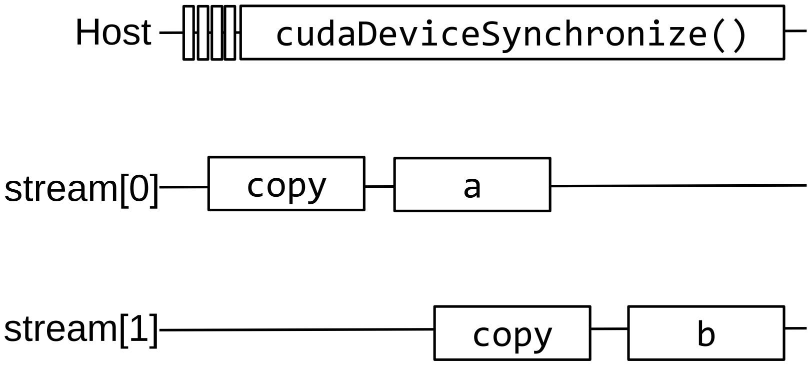
Streams Example

```
cudaStream_t stream[2];
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);

cudaMemcpyAsync(d_A, h_A, A_size,
        cudaMemcpyHostToDevice, stream[0]);
cudaMemcpyAsync(d_B, h_B, B_size,
        cudaMemcpyHostToDevice, stream[1]);
kernel_A<<<grid, threads, 0, stream[0]>>>(d_A);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_B);

cudaDeviceSynchronize();

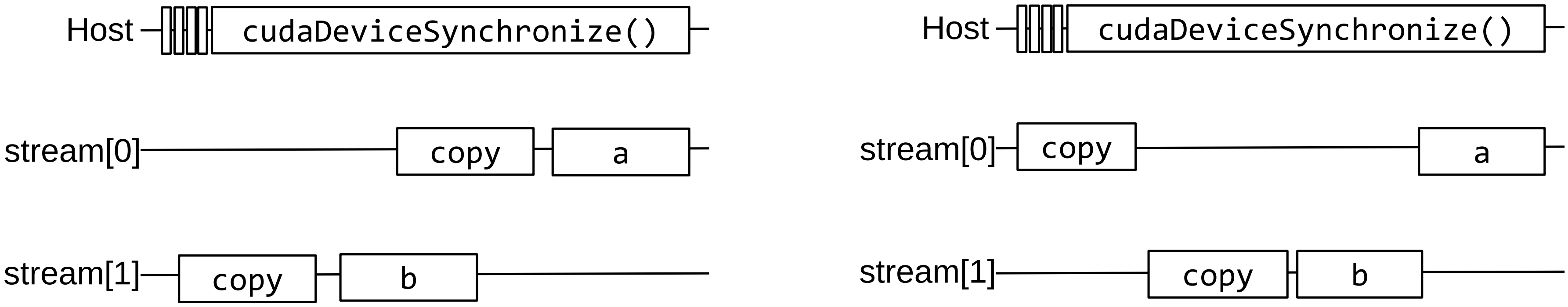
for (i=0; i<2; i++)
    cudaStreamDestroy(stream[i]);
```



Sequential Consistency

- Within a stream commands are executed in order, but among streams operations may execute in any order
- Therefore, program correctness should next never depend on this order

- These executions are also possible under the sequential consistency model:



CUDA Events

- **Event creation and destruction**
  - `cudaEventCreate ( cudaEvent_t* event )`
  - `cudaEventDestroy ( cudaEvent_t event )`
- **Record an event at this point in the program or stream**
  - `cudaEventRecord ( cudaEvent_t event, cudaStream_t stream = 0 )`
- **Get elapsed time between events**
  - `cudaEventElapsedTime (float* ms, cudaEvent_t start, cudaEvent_t end )`
- **Make the host wait for an event**
  - `cudaEventSynchronize ( cudaEvent_t event )`
- **Force the operations in a stream to wait for an event, possibly in another stream. Allows to synchronize different streams without delaying the host.**
  - `cudaStreamWaitEvent ( cudaStream_t stream, cudaEvent_t event, unsigned int  flags )`

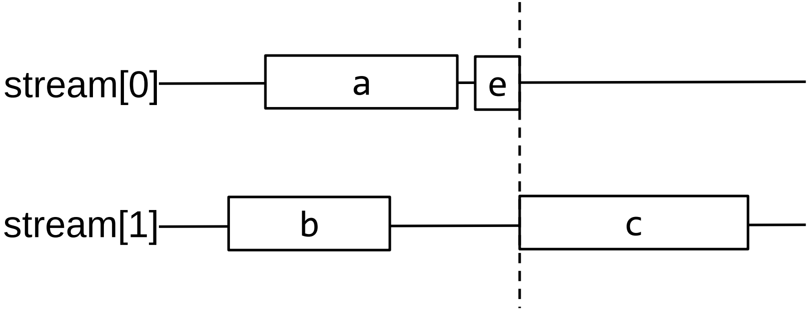
Cross-Stream Synchronization

```
cudaStream_t stream[2];
cudaEvent_t event;
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);
cudaEventCreate(&event);

kernel_A<<<grid, threads, 0, stream[0]>>>(d_out_A, d_A);
cudaEventRecord(event, stream[0]);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_out_B, d_B);
cudaStreamWaitEvent(stream[1], event, 0);
kernel_C<<<grid, threads, 0, stream[1]>>>(d_out, d_out_A, d_out_B);
```

Cross-Stream Synchronization

```
cudaStream_t stream[2];
cudaEvent_t event;
for (i=0; i<2; i++)
    cudaStreamCreate(&stream[i]);
cudaEventCreate(&event);
```




```
kernel_A<<<grid, threads, 0, stream[0]>>>(d_out_A, d_A);
cudaEventRecord(event, stream[0]);
kernel_B<<<grid, threads, 0, stream[1]>>>(d_out_B, d_B);
cudaStreamWaitEvent(stream[1], event, 0);
kernel_C<<<grid, threads, 0, stream[1]>>>(d_out, d_out_A, d_out_B);
```

2nd hands-on session

- **Typically your program would look like:**

```
//copy inputs (host to device)
cudaMemcpy(d_input, h_input, input_size, cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(d_filter, h_filter, filter_size, cudaMemcpyHostToDevice);

//call the kernel
convolution_kernel<<<grid, threads>>>(d_output, d_input, d_filter);

//copy outputs (device to host)
cudaMemcpy(h_output, d_output, output_size, cudaMemcpyDeviceToHost);
```

- **But this achieves no overlap between computation and communication!**

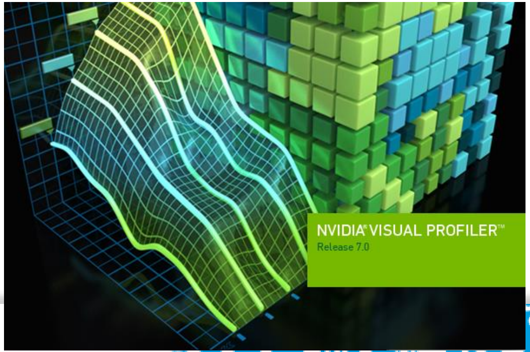
2nd hands-on session

- Go to directory overlap, type: `git checkout explicit`
- Make sure you understand everything in the code
- Implement the function `convolution_streams()` following the guidelines in the comments
- Use CUDA Streams to allow overlap between host to device transfers, kernel execution, and device to host transfers

- Hint #1 Divide the input data into a number of chunks, let the kernel in each stream operate on a chunk, copy the output back to host in chunks as well
- Hint #2 Use CUDA events to synchronize between operations in different streams when necessary
- Hint #3 Using offsets when passing kernel arguments, you can avoid changing the kernel code

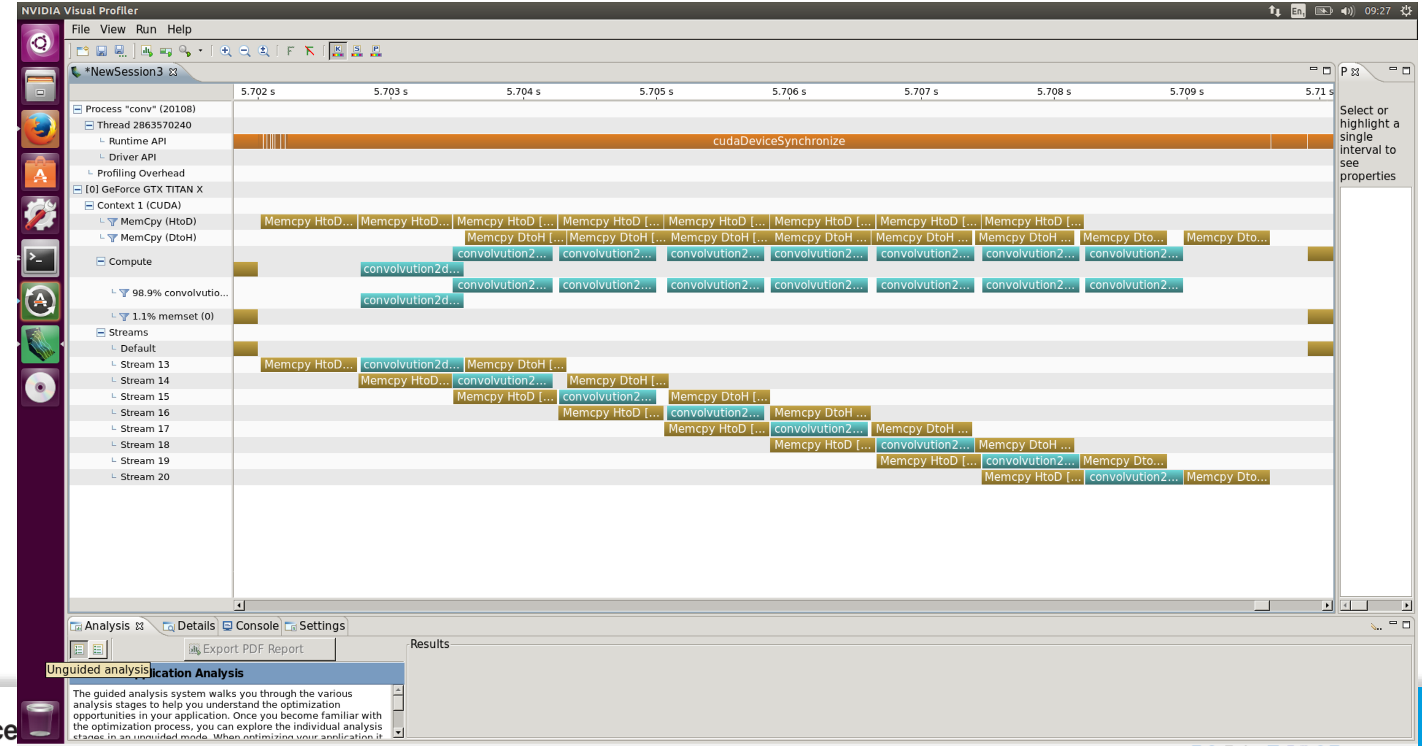
Using Nvidia Visual Profiler

- When using streams the Nvidia Visual Profiler is a great tool to get insight in all the parallel operations in your program
- To start this program on DAS5:
  - Login to DAS5
  - Type `srun -N 1 -C TitanX --gres=gpu:1 --pty bash`
  - Open another terminal and login to DAS5 with X-forwarding (e.g. use `ssh –XY`)
  - ssh with X-forwarding to the node reserved by the previous srun command
    - WARNING: Be *extremely* careful to ssh to the correct node and exit before your reservation ends!
  - On the node type `nvvp` to start the Nvidia Visual Profiler
  - The load image should appear
  - If you get the error:
    - Nvvp:
      An error has occurred. See the log file
    - You probably don't have correct X-forwarding



Using Nvidia Visual Profiler

- **For example for our streamed convolution kernel we could see:**



Performance Modeling