Harness the Power of GPUs:
An Introduction to GPGPU Programming
Lab 2: Data Movement and First Kernels

Guido Juckeland
Visiting Scholar
Technische Universität Dresden, Germany

June 16, 2014
## CUDA Cheat Sheet (1)

### Typical program structure

<table>
<thead>
<tr>
<th>Step</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initializing</td>
<td><code>cuInit()</code> [Optional, done with first CUDA call]</td>
</tr>
<tr>
<td>Memory allocation</td>
<td><code>cudaMalloc(DevicePointer, nBytes)</code></td>
</tr>
<tr>
<td>Copy input data to device</td>
<td><code>cudaMemcpy(dest, src, nBytes, cudaMemcpyHostToDevice)</code></td>
</tr>
<tr>
<td>Invoke kernel</td>
<td><code>kernelname&lt;&lt;&lt;numBlocks,threadsPerBlock&gt;&gt;&gt;(arguments)</code></td>
</tr>
<tr>
<td>Copy results back to host</td>
<td><code>cudaMemcpy(dest, src, nBytes, cudaMemcpyDeviceToHost)</code></td>
</tr>
<tr>
<td>Free device memory</td>
<td><code>cudaFree(DevicePointer)</code></td>
</tr>
</tbody>
</table>
CUDA Cheat Sheet (2)

**Kernel tricks**

- **Kernel declaration**
  - `__global__` `kernelname(args)` - callable from host
  - `__device__` `kernelname(args)` – callable only from GPU

- **Global 1D Thread-Index**
  - `int idx=blockIdx.x*blockDim.x+threadIdx.x;`

- **Shared Memory (only within a thread block)**
  - `__shared__` `float As[BLOCK_SIZE][BLOCK_SIZE];`

- **Thread synchronization (only within a thread block)**
  - `__syncthreads();`
CUDA Cheat Sheet (3)

**Working on BlueWaters**

- **ssh using your training account into bwbay.ncsa.illinois.edu**
  
  ```
  ssh tra123@bwbay.ncsa.illinois.edu
  ```

- **Grab an interactive session**
  
  ```
  qsub -I -l nodes=1:ppn=16:xk -l walltime=01:30:00
  ```
CUDA Cheat Sheet (4)

Compiling and running a CUDA program

- On remote systems make sure that the CUDA environment is available (usually requires a `module load cudatoolkit` or similar)
- Name your CUDA files with the suffix `.cu`
- Compile your program using `nvcc` (e.g. `nvcc myprogram.cu`)
- Execute your program by running `./a.out`
- On remote systems the node you compile on might not feature a GPU, you will have to use a batch system to access the “right” node
Tasks 1: Data movement

- Write a program that allocates a vector `a_host` of 1,000,000 elements of type `int`. Initialize the vector such that the value of each element is equal to its index (`a[i]=i`).

- Allocate the vectors `a_device`, `b_device`, and `b_host` of the same type and size. The *device vectors must be located on the device. Use the `cudaMemcpy` routine to transfer `a_host` to `a_device`, copy it to `b_device` and transfer it back to `b_host`. Check that `a_host` and `b_host` hold the same values. Don’t forget to free the used memory afterwards.

- Write a kernel to do the copying on the device and substitute the on-device memcpy with it.
Task 2: Vector operations

- Write a program that allocates three vectors $a_{\text{host}}, b_{\text{host}},$ and $c_{\text{host}}$ of type $\text{int}$ and 1,000,000 elements. Initialize the vectors $a_{\text{host}}$ and $b_{\text{host}}$ such that the value of each element is equal to its index ($a[i]=i$). Transfer $a_{\text{host}}$ and $b_{\text{host}}$ to the device and run the vector addition kernel using a block size of 500 and a grid size of 2000. Transfer the result back to the host and check it for correctness.
- Modify your program so that it works for arbitrary block sizes.