Åbo Akademi University

IT00CG19-3002
GPU Programming

Slide set #3: GPU Execution Model

Fall 2023, period 1

Jan Westerholm
Åbo Akademi University

GPU Hardware Architecture

- cudaGetDeviceProperties() on dione.utu.fi gives us (additional code for pretty typing):

```
Found 4 GPU devices
    GPU device 0:
        Device Tesla V100-PCIE-16GB has compute capability 7.0
        totalGlobalMemory                =         16.95 GB
        l2CacheSize                      =         6291456 B
        regsPerBlock                     =         65536
        streaming multiprocessor         =         80
        maxThreadsPerMultiprocessor      =         2048
        sharedMemPerBlock                =         49152 B
        clockRate                        =         1.380 GHz
        maxThreadsPerBlock               =         1024
        concurrentKernels                =         yes
        maxGridSize             =        2147483647 x 65535 x 65535
        maxThreadsDim           =        1024 x 1024 x 64
```

Åbo Akademi University

Execution Mechanism

- A GPU has a number of *streaming multiprocessors* SM to which thread blocks are assigned.
- Threads within a thread block are executed with a *Single Instruction Multiple Threads* SIMT mechanism in groups of 32 threads called *warps*.
  - ***All 32 threads in a warp execute the same instruction concurrently except when serializing (e.g. atomic operations)***
  - In case of divergent codes, some of the threads within a warp may be inactive
- A thread has 32 bit local *registers* available for eg. 32-bit integers and single precision floats. A double precision floating point value uses two registers.

Execution Mechanism

- All resources in a GPU are limited, hence there are bounds to the capabilities of the GPU and the SMs:
  - The total amount of memory on the GPU, (<= 48GB)
  - Maximum x, y and z-dimension of the grid of thread blocks
  - Maximum number of threads per thread block
  - Warp size is fixed, today NVIDIA = 32, AMD = 64.
  - Maximum # of blocks resident per SM
  - Maximum # of active threads per SM
  - Number of 32-bit registers per SM
  - Maximum number of 32-bit registers per block
  - Maximum amount of shared memory per SM

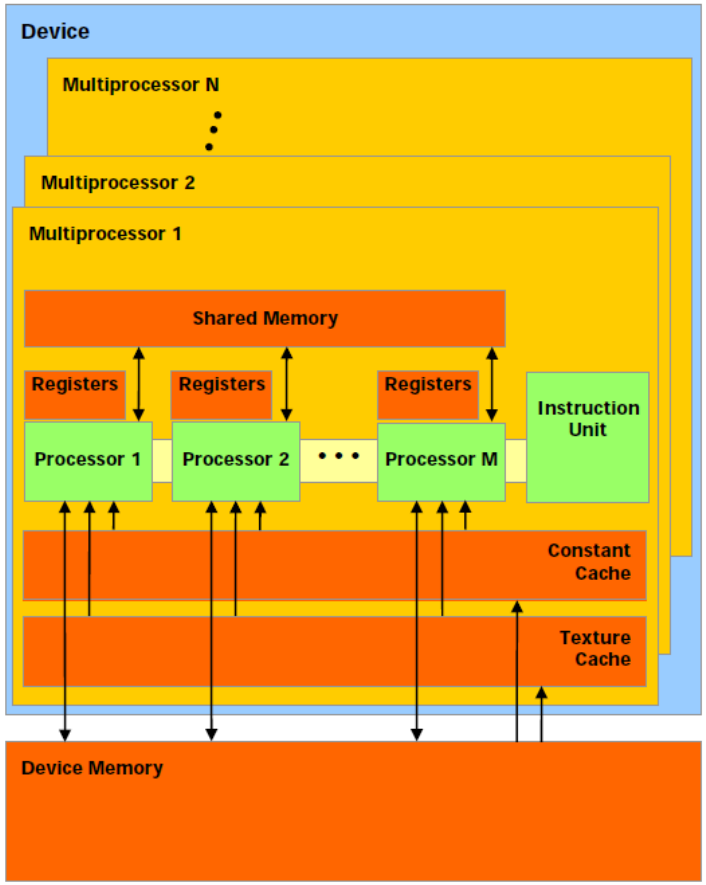
Åbo Akademi University

Execution Mechanism

- All numbers related to the execution mechanism will change in the future as the *compute capabilities* of the GPUs evolve.
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities, Tables 14 and 15
- The numbers specific to your device(s) can be obtained at runtime using two APIs and a structure:
  - cudaGetDeviceCount(&deviceCount)
  - cudeDeviceProp deviceProp
  - cudaGetDeviceProp(&deviceProp, iterate_deviceCount)

GPU Hardware Architecture

- Compute capabilities are classified according to the major revision number
  - *Kepler* architecture 3.x
  - *Maxwell* architecture 5.x
  - *Pascal* architecture 6.x
  - *Volta* architecture 7.x
  - *Ampere* architecture 8.x
  - *Hopper* architecture 9.x
- Sample numbers for dione:
  - N = 80
  - 5120 cuda cores
  - M = 5120/80 = 64



Åbo Akademi University

CUDA Multiple Devices

- In case you have more than one GPU card on your machine
  - make sure your power supply can deliver enough power, typical rating is 200–300 W / GPU card
- Choose your device in your program by calling

  cudaSetDevice(int device_number);

Åbo Akademi University

Default Stream

- In your program, all CUDA APIs and kernel invocations are put into a default ***stream***, a FIFO queue (first in, first out, "in order" execution).
- Some of the APIs are blocking
  - `cudaMemcpy( , )`
- Some of the commands are non-blocking
  - kernel launches with managed memory

    ```
    kernel<<< , >>>(args)
    ```
  - `cudaMemcpyAsync( )`
- Use device synchronization `cudaDeviceSynchronize()` if you need to wait until all non-blocking commands have finished

Multiple (Concurrent) Streams

- If your GPU supports concurrent kernels (in device properties, the value of concurrentKernels = 1) and asynchronous data transfers between host and device (deviceOverlap = 1) you can create and use multiple streams (here N streams):

```
cudastream_t      stream[N];
cudaStreamCreate(&stream[i]);
…
cudaMemcpyAsync( …, stream[i]);
MyKernel <<<100, 512, 0, stream[i]>>>( … );
cudaMemcpyAsync(…, stream[i]);
…
cudaStreamDestroy(stream[i]);
```