Exercises

HPC COMPUTING WITH CUDA AND TESLA HARDWARE

Exercise 0: Run a Simple Program

- Log on to test system
- Compile and run pre-written CUDA program — deviceQuery

```
CUDA Device Query (Runtime API) version (CUDART static linking)
There are 2 devices supporting CUDA

Device 0: "Tesla C1060"
   CUDA Capability Major revision number:         1
   CUDA Capability Minor revision number:         3
   Total amount of global memory:                 4294705152 bytes
   Number of multiprocessors:                     30
   Number of cores:                               240
   Total amount of constant memory:               65536 bytes
   Total amount of shared memory per block:       16384 bytes
   Total number of registers available per block: 16384
   Warp size:                                     32
   Maximum number of threads per block:           512
   Maximum sizes of each dimension of a block:    512 x 512 x 64
   Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
   Maximum memory pitch:                          262144 bytes
   Texture alignment:                             256 bytes
   Clock rate:                                    1.44 GHz
   Concurrent copy and execution:                 Yes
   Run time limit on kernels:                     No
   Integrated:                                    No
   Support host page-locked memory mapping:       Yes
   Compute mode:                                  Exclusive (only
one host thread at a time can use this device)
```

Exercise 1: Move Data between Host and GPU

- Start from the "cudaMallocAndMemcpy" template.
- Part 1: Allocate memory for pointers *d_a* and *d_b* on the device.
- Part 2: Copy *h_a* on the host to *d_a* on the device.
- Part 3: Do a device to device copy from *d_a* to *d_b.*
- Part 4: Copy *d_b* on the device back to *h_a* on the host.
- Part 5: Free *d_a* and *d_b* on the host.
- Bonus: Experiment with *cudaMallocHost* in place of *malloc* for allocating *h_a.*

Exercise 2: Launching Kernels

- Start from the "myFirstKernel" template.
- Part1: Allocate device memory for the result of the kernel using pointer *d_a.*
- Part2: Configure and launch the kernel using a 1-D grid of 1-D thread blocks.
- Part3: Have each thread set an element of *d_a* as follows:

```
idx = blockIdx.x*blockDim.x + threadIdx.x
d_a[idx] = 1000*blockIdx.x + threadIdx.x
```

- Part4: Copy the result in *d_a* back to the host pointer *h_a.*
- Part5: Verify that the result is correct.

Exercise 3: Reverse Array, Single Block

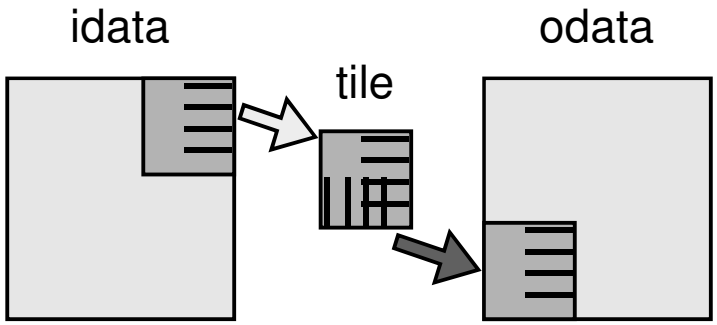
- Given an input array $\{a_0, a_1, \ldots, a_{n-1}\}$ in pointer *d_a,* store the reversed array $\{a_{n-1}, a_{n-2}, \ldots, a_0\}$ in pointer *d_b*
- Start from the "reverseArray_singleblock" template
- Only one thread block launched, to reverse an array of size
  N = numThreads = 256 elements
- Part 1 (of 1): All you have to do is implement the body of the kernel "reverseArrayBlock()"
- Each thread moves a single element to reversed position
  - Read input from *d_a* pointer
  - Store output in reversed location in *d_b* pointer

Exercise 4: Reverse Array, Multi-Block

- Given an input array $\{a_0, a_1, \ldots, a_{n-1}\}$ in pointer *d_a,* store the reversed array $\{a_{n-1}, a_{n-2}, \ldots, a_0\}$ in pointer *d_b*
- Start from the "reverseArray_multiblock" template
- Multiple 256-thread blocks launched
  - To reverse an array of size N, N/256 blocks
- Part 1: Compute the number of blocks to launch
- Part 2: Implement the kernel reverseArrayBlock()
- Note that now you must compute both
  - The reversed location within the block
  - The reversed offset to the start of the block

Exercise 5: Matrix Transpose

- Access columns of a tile in shared memory to write contiguous data to global memory
- Requires __syncthreads() since threads write data read by other threads
- Pad shared memory array to avoid bank conflicts



Exercise 5: Matrix Transpose

- There are further optimisations: see the New Matrix Transpose SDK example.