Hands-on CUDA exercises

CUDA Exercises

- We have provided skeletons and solutions for 6 hands-on CUDA exercises
- In each exercise (except for #5), you have to implement the missing portions of the code
  - Finished when you compile and run the program and get the output "Correct!"
- Solutions are included in the "solution" folder of each exercise

© NVIDIA Corporation 2008

Compiling the Code: Windows

- Open the <project>.sln file in Microsoft Visual Studio
  - Build the project
  - 4 configuration choices:
    - Release,Debug,EmuRelease, EmuDebug
- To debug your code build EmuDebug configuration
  - Can set breakpoints inside kernels (__global__ or __device__ functions)
  - Can debug the code as normal, even printf!
  - One CPU thread per GPU thread
  - Threads not actually in parallel on GPU

© NVIDIA Corporation 2008

Compiling the Code: Linux

```
nvcc <filename>.cu [-o <executable>]
```

- Builds release mode

```
nvcc -g <filename>.cu
```

- Builds debug (device) mode
- Can debug host code but not device code (runs on GPU)

```
nvcc -deviceemu <filename>.cu
```

- Builds device emulation mode
- All code runs on CPU, but no debug symbols

```
nvcc -deviceemu -g <filename>.cu
```

- Builds debug device emulation mode
- All code runs on CPU, with debug symbols
- Debug using gdb or other linux debugger

© NVIDIA Corporation 2008

1: Copying between host and device

- Start from the "cudaMallocAndMemcpy" template.
- **Part1**: Allocate memory for pointers *d_a* and *d_b* on the device.
- **Part2**: Copy *h_a* on the host to *d_a* on the device.
- **Part3**: Do a device to device copy from *d_a* to *d_b*.
- **Part4**: Copy *d_b* on the device back to *h_a* on the host.
- **Part5**: Free *d_a* and *d_b* on the host.
- **Bonus**: Experiment with *cudaMallocHost* in place of *malloc* for allocating *h_a.*

© NVIDIA Corporation 2008

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.

© NVIDIA Corporation 2008

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

© NVIDIA Corporation 2008

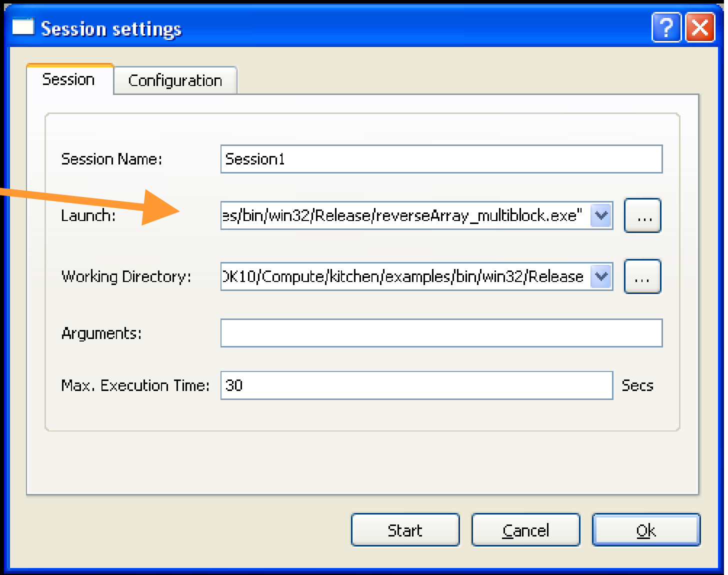
4: Reverse Array (multiblock)

- 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

© NVIDIA Corporation 2008

5: Profiling Array Reversal

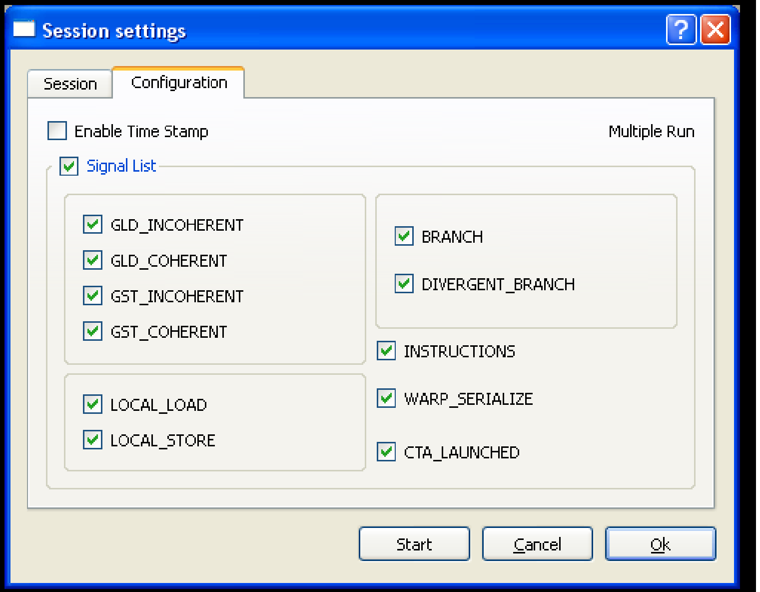
- Your array reversal has a performance problem
- Use the CUDA Visual Profiler to run your compiled program
  - Compile release mode, run “cudaprof”, create a new project
  - Browse to your executable file in the “launch box” of the session settings dialog



© NVIDIA Corporation 2008

5: Profiling Array Reversal

- Click on configuration tab
- Select the check box next to “signal list”
- Click OK, then “Start”
- Check if any of these are non-zero:
  - GLD_INCOHERENT
  - GST_INCOHERENT
  - WARP_SERIALIZE
- Take a note of the “GPU Time”



© NVIDIA Corporation 2008

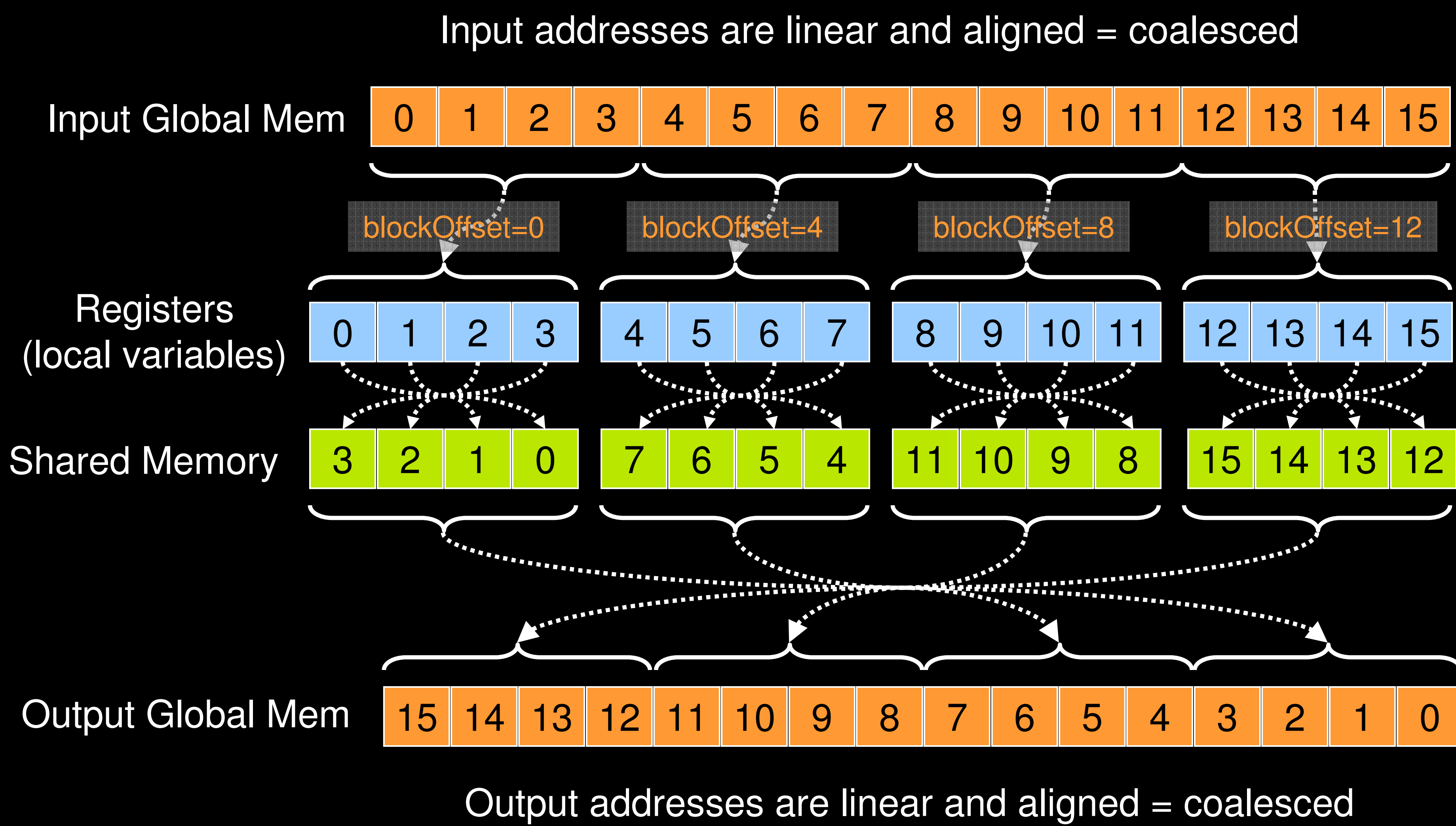
6: Optimizing Array Reversal

- Goal: Get rid of incoherent loads/stores and improve performance
  - Use shared memory to reverse each block

- Part 1: compute the number of bytes of shared mem
  - One element per thread
- Part 2: implement the kernel
  - Comments should help
  - Don’t forget to compute the correct block offset!
- Part 3: Profile the working code
  - Compare value of GLD/GST_INCOHERENT to previous
  - Compare GPU Time to previous

© NVIDIA Corporation 2008

Reverse Data in shared memory



© NVIDIA Corporation 2008