



**NVIDIA**®

**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**

# 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**

# 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

# 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*.

## 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.

# 3: Reverse Array (single block)



- Given an input array  $\{a_0, a_1, \dots, a_{n-1}\}$  in pointer  $d\_a$ , store the reversed array  $\{a_{n-1}, a_{n-2}, \dots, a_0\}$  in pointer  $d\_b$
- Start from the “**reverseArray\_singleblock**” template
- Only one thread block launched, to reverse an array of size  $N = \text{numThreads} = 256$  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

# 4: Reverse Array (multiblock)

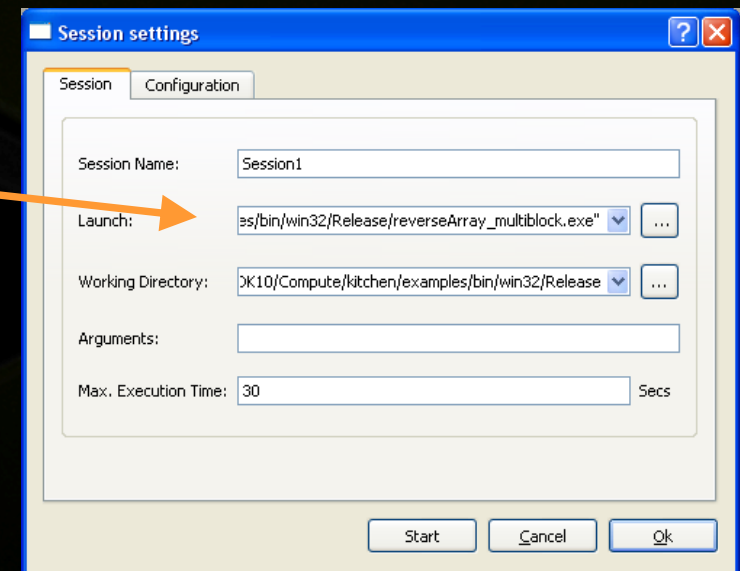


- Given an input array  $\{a_0, a_1, \dots, a_{n-1}\}$  in pointer  $d\_a$ , store the reversed array  $\{a_{n-1}, a_{n-2}, \dots, a_0\}$  in pointer  $d\_b$
- Start from the “**reverseArray\_multiblock**” template
- Multiple 256-thread blocks launched
  - To reverse an array of size  $N$ ,  $N/256$  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

# 5: Profiling Array Reversal



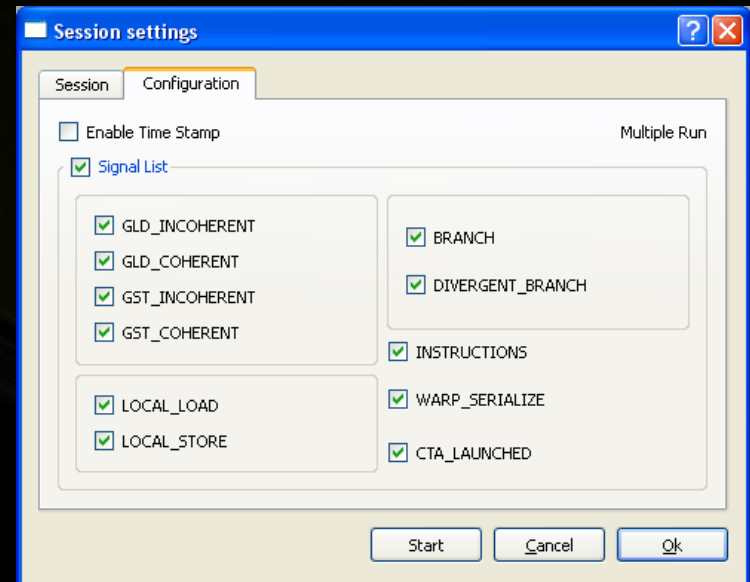
- Your array reversal has a performance problem
- Use the CUDA Visual Profiler to run your compiled program
  - Compile release mode, run “`cuda_prof`”, create a new project
  - Browse to your executable file in the “launch box” of the session settings dialog



# 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”



# 6: Optimizing Array Reversal

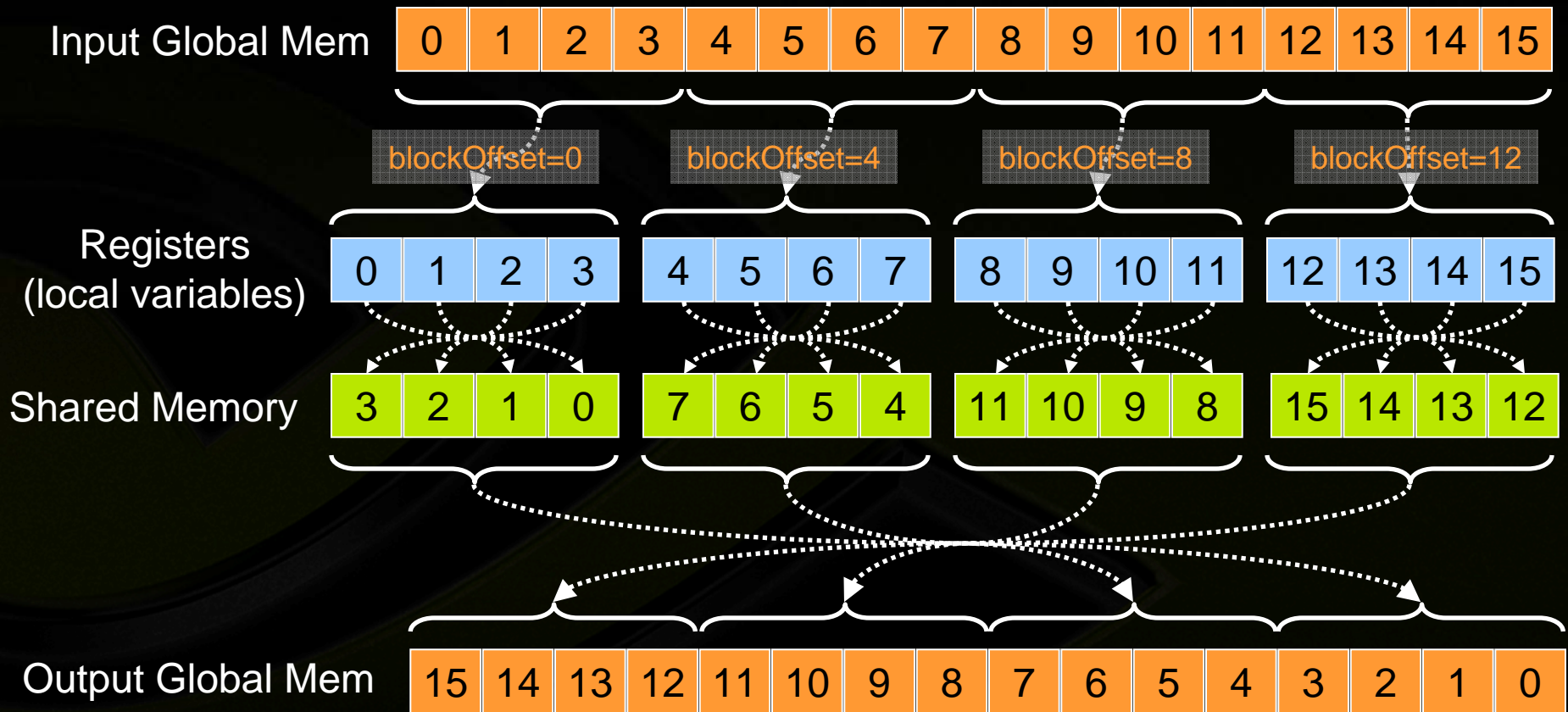


- **Goal: Get rid of incoherent loads/stores and improve performance**
  - Use shared memory to reverse each block
- **Part 1: compute the number of bytes of shared mem**
  - One element per thread
- **Part 2: implement the kernel**
  - Comments should help
  - Don't forget to compute the correct block offset!
- **Part 3: Profile the working code**
  - Compare value of `GLD/GST_INCOHERENT` to previous
  - Compare GPU Time to previous

# Reverse Data in shared memory



Input addresses are linear and aligned = coalesced



Output addresses are linear and aligned = coalesced