

POWERING DISCOVERIES  
THAT CHANGE THE WORLD



# CUDA Programming

## Basic Concepts in C and Fortran



Lars Koesterke

Seventh Annual Texas A&M Research Computing Symposium

5/21/24

Intermediate Level:

Writing an Optimized Kernel using  
Shared Memory and Streams



# Overview

- Tutorial is meant for programmers (C/C++/Fortran)
  - No exposure to CUDA programming, yet
  - Basic knowledge of C/Fortran, or generally of a programming language
- Focus
  - Basic concepts: ‘software matching hardware’
  - Performance optimization and techniques beneficial under most circumstances
- Simple kernel: ‘Hello World’ equivalent for GPUs
- Lab
- Stencil update
  1. Using main memory on card
  2. Using shared memory of each SM (Streaming Multiprocessor)
  3. Interleaving data transfer and calculations
    - Using the host as well
    - Using multiple nodes with MPI

# Why GPUs?

GPUs promise a performance advantage vs CPUs

## Single precision GPUs, i.e. graphics cards

- Machine learning, pattern matching, etc.
- Undoubtedly, the performance per dollar is outstanding

This option (for compute) has gone away  
Frontera-GPU may be the last holdout

## Double precision GPUs, Nvidia A100/H100, AMD MI200

- ‘Classical’ number crunching
- Advantages v. CPUs are not so clear cut
- Apples to apples: dual-socket CPU node with a single GPU
- # of transistors, power consumption, price per device are similar
- The gain is often modest
- However, excellent performance for Machine Learning
  - Specialized hardware: Tensor cores

# Why CUDA?

- GPUs from a variety of vendors can be programmed in
  - OpenCL → tough going
  - OpenMP, OpenACC → Pragma based language extension
  - CUDA (Nvidia), HIP (AMD), SYCL (Intel)
- NVIDIA: language developed for specific hardware
- AMD is now developing HIP, a CUDA equivalent
- OpenMP offload or OpenACC not available everywhere
- Advantages and Disadvantages
  - Supposedly faster language evolution
  - How does CUDA translate to other options? Execution and development speed
  - CUDA is supported by C and Fortran compilers

**This class will be bi-lingual: C & Fortran**



# CUDA: C and Fortran

- CUDA (from Nvidia) is an addition to C
- Nvidia provides a separate compiler for Fortran
- User profile at TACC: 65% C/C++ and 35% Fortran
- Being able to teach both languages is a big plus!
- C + CUDA: all code is written in C
- Nvidia CUDA Fortran Compiler: all code is written in Fortran

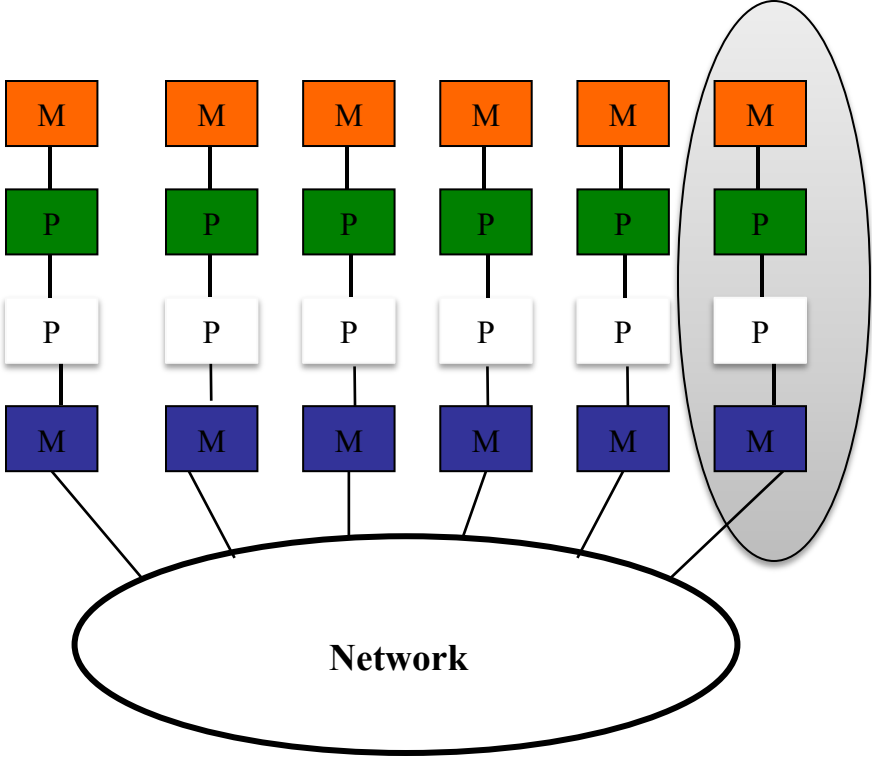
# Cluster with GPUs

Memory GPU

GPU

CPU

Memory CPU



Step 1:  
One node

Step 2:  
Multiple nodes

**Programming for a CPU/GPU pair is like programming for a small CPU cluster**

# GPU Memory Hierarchy

- GPUs have multiple memories
- Main Memory: accessible by all Threads executing on the SMs (Streaming Multiprocessors)
  - On the smaller side: 40 – 80 GB
  - High bandwidth: DDR5 → HBM
- Each SM has shared memory that is local to an SM
  - Very small: ~16KB per SM
  - Faster, compared to the Main GPU Memory. Can also act as a cache
- Additional Memories: Texture, Constants, etc.

# Hardware Overview:

## CPU vs. GPU characteristics

### CPU

- Dozens computation cores
  - Independent execution
  - Drawing from a pool of memory
- Lock-step execution
  - Vectorization (SIMD) with few (8/16) vector lanes
  - Hundreds of vector lanes
  -
- Optimized for serial and parallel exec.
- Memory latency hidden by
  - Caches, prefetching
  - Parallel execution (thread-# ~ core-#)

### GPU

- Dozens Streaming Multiprocessors
  - ‘Core’ = Streaming Multiprocessor
  - Independent execution
  - Drawing from a pool of memory
- Lock-step execution
  - Single Instruction Multiple Threads (SIMT)
  - Warps (bundles of 32) and many (1000’s) CUDA cores
- Optimized for parallel execution
- Memory latency hidden by
  - Parallel execution (many, many threads)
  - Caches, prefetching

# Hardware Comparison

(Frontera and Lonestar)

	Dual socket Intel Cascade Lake	Dual socket AMD Milan	Nvidia H100	Nvidia A100	Nvidia A30
Core/SM	56	128	144	108	56?
Speed (GHz)	2.7	2.45	1.6	1.41	1.24
SIMD / SIMT width (32 bit)	16	8	32	32	32
Vector lanes/ CUDA cores	896*2	1024*2	16896	6912	3584
	FMA	FMA			

# Programming should match the hardware:

## Thread configuration

- Two-tiered hierarchy: grids and blocks (to match hardware)
  - A kernel contains one grid *“The grid executes on all SMs”*
  - A grid is a grid of blocks *“Each block executes on one SM”*
  - A block contains threads *“The threads in a block are configured to Warps”*
- When CPU code is translated into GPU code:
  - A specific thread (typically) executes one specific “loop iteration”
    - Example: Thread “#1202” may execute loop iteration  $i=12, j=27$  (nested loop here)
    - Only one thread is mapped to a specific loop iteration
    - Only one loop iteration is mapped to a specific thread
    - This is called a bijection
  - Thread-# is not a linear construct, but has 6 components
    - Grid:3; Block:3; confusing in the beginning, but quite helpful later

# Thread configuration

- Why is the hierarchy two-tiered?
  - All threads of a block (an element of the grid) execute on one SM
  - Threads on a single SM can be synchronized (execution and memory)
  - Synchronization beyond a single block is more complicated and costly
  - The hardware provides multiple SMs. Therefore, a grid must contain multiple blocks to keep the SMs busy
  - If your hardware has 144 SMs (H100), you should select a setup with at least 144 blocks. For performance you will likely need (many) more!
- The blocks in a grid match the HW parallelism on the SM level → multiple blocks
- An SM is also executing in parallel: “CUDA” cores
- Multiple threads will execute concurrently on an SM



# Thread configuration

- All blocks in a grid are configured the same
- A specific thread has
  - A block-identifier (it's position in the block)
  - A grid-identifier (the position of the block in the grid)
- Let's assume that both identifiers are linear indices of type integer:
  - `threadidx`: index of thread in block
  - `blockidx`: index of block in the grid
- And the blocks and the grid have “sizes”
  - `blocksize`: size of the block, i.e. number of threads in a block
  - `gridsize`: number of blocks in the grid

# Thread configuration

- Example:
  - **Blocksize** = 32 (one warp)
  - **Gridsize** = 8 (Note: there are 80 SMs in Longhorn's GPUs)
  - Total number of threads = 256 (32 x 8)
- What block and thread position does **thread-# 72** have (C indexing)?  
Block-# 0: 32 threads (thread index: 0,...,31)  
Block-# 1: 32 threads (thread index: 32,...,63)  
Block-# 2: 32 threads (thread index: 64,65,66,67,68,69,70,71,72,...,95)  
9<sup>th</sup> position in block 2                      1, 2, 3, 4, 5, 6, 7, 8, 9 position (nth-thread)
- What block and thread position does **thread-# 72** have (Fortran indexing)?  
Block-# 1: 32 threads (thread index: 1,...,32)  
Block-# 2: 32 threads (thread index: 33,...,64)  
Block-# 3: 32 threads (thread index: 65,66,67,68,69,70,71,72,73,...,96)  
8<sup>th</sup> position in block 3                      1, 2, 3, 4, 5, 6, 7, 8 position (nth-thread)
- What is the thread-# for the 7th thread in 5th block?
  - C-Index:  $(5-1) \times 32 + 7 = 135$     F-Index:  $(5-1) \times 32 + 7-1 = 134$
- Note, that I have carefully use “position” at times.
  - Indices in C/Fortran start at 0 and 1, respectively

# Thread configuration

- Example:
  - `Blocksize = 32` (one warp)
  - `Gridsize = 8`
- What **block/grid** index does thread #72 have?
  - C: #72 is in Block w/ index **2** and has the index **7** in the block
  - Fortran: #72 is in Block w/ index **3** and has the index **8** in the block
- What is the thread-# for the 7th thread in 5th block?
  - C: `threadidx=6, blockidx=4`  
thread-# = `blockidx * blocksize + threadidx`
  - Fortran: `threadidx=7, blockidx=5`  
thread-# = `(blockidx-1) * blocksize + threadidx`

C, Fortran: thread-# starts at 0 and 1, respectively

# Thread configuration

- Linear configuration → Cubes
- A grid contains a cube of blocks: 3 coordinates, x/y/z
- A block contains a cube of threads: 3 coordinates, x/y/z
- There is some freedom on what coordinate to use
  - Some constraints are discussed later
  - minimum: 1 coordinate for the blocks, 1 coordinate for the grid
- Why x/y/z?

Maps nicely to numerical code that uses a  
1d (x), 2d (x/y), or 3d (x/y/z) grid

# Thread configuration

- The thread configuration is specified in the host code (code that is executed on the host)
- You are free to choose a name
  - Here I use `blocksize` and `gridsize`
- The configuration variables are structures of type `dim3`
  - `dim3` is predefined in the header file `cuda_runtime.h` and the module `mod_gpu`, respectively
  - `dim3` contains the integer components `x`, `y` and `z`

```
use mod_gpu
type(dim3) :: blocksize, gridsize

gridsize = dim3(8,1,1)
blocksize = dim3(32,1,1)
```

```
#include <cuda_runtime.h>

dim3 gridsize(8,1,1)
dim3 blocksize(32,1,1)
```

- Same example as before: 32 threads in a block (x), 8 blocks in the grid (x)

# Thread configuration

- **blocksize = 32,2,1; gridsize = 8,3,1**
- Total threads in a block:  $64 = 32 \times 2 \times 1$
- Total number of blocks in the grid:  $24 = 8 \times 3 \times 1$
- Total number of threads:  $1536 = 32 \times 24$
- Threads are arranged: first x, then y, then z;  
“threads in block” before “blocks in grid”

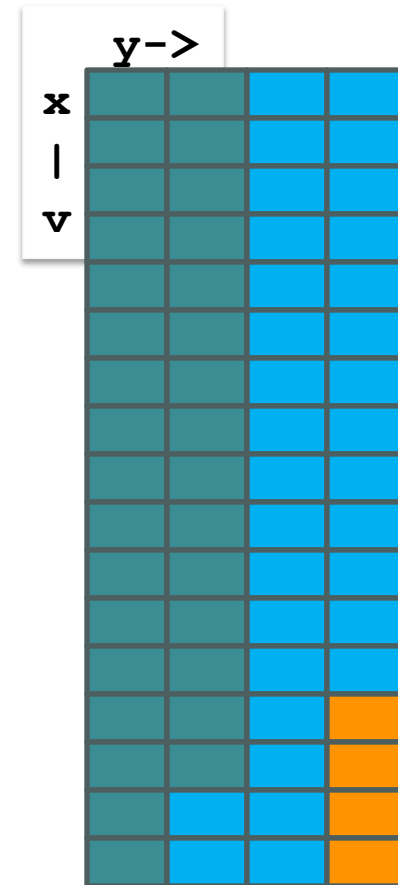
	<u>“thread in block”</u>	<u>“block in grid”</u>
• 1 <sup>st</sup> thread: position	( 1 , 1 , 1 )	( 1 , 1 , 1 )
• 2 <sup>nd</sup> thread:	( 2 , 1 , 1 )	( 1 , 1 , 1 )
• 32 <sup>nd</sup> thread:	( 32 , 1 , 1 )	( 1 , 1 , 1 )
• 33 <sup>rd</sup> :	( 1 , 2 , 1 )	( 1 , 1 , 1 )
• 64 <sup>th</sup> :	( 32 , 2 , 1 )	( 1 , 1 , 1 )
• 65 <sup>th</sup> :	( 1 , 1 , 1 )	( 2 , 1 , 1 )
• Etc.		

# Thread configuration: uneven setup

- `blocksize = 17,4,1; gridsize = 2,1,1`
- Total threads in a block:  $68 = 17 \times 4 \times 1$
- How are the warps formed? – in chunks of 32

**f**=full, **p**=partial row in **x**

- 1<sup>st</sup> warp:  $32 = 17+15$       **y**=1 (f) **y**=2 (p)
  - 2<sup>nd</sup> warp:  $32 = 2+17+13$     **y**=2 (p) **y**=3 (f) **y**=4 (p)
  - 3<sup>rd</sup> warp:  $4 = 4$             **y**=4 (p)
- Two full warps, one partial warp
    - The partial warp contains also 32 threads, but 28 are doing nothing
    - No spill-over to the next block in the grid; why?





# Kernel launch, Chevron syntax

- A kernel is a routine that is executed on the GPU
- Calling a kernel (from the host) is called “launching a kernel”
- Calling a kernel uses familiar syntax (function/subroutine call) augmented by Chevron syntax **containing configuration arguments**
- The Chevron syntax (`<<<...>>>`) communicates the thread configuration to the kernel
  - First **argument**: grid configuration
  - Second **argument**: block configuration

```
use mod_gpu
type(dim3) :: blocksize, gridsize

gridsize = dim3(8,1,1)
blocksize = dim3(32,1,1)

call do_this<<<gridsize,blocksize>>>()
```

```
#include <cuda_runtime.h>

dim3 gridsize(8,1,1)
dim3 blocksize(32,1,1)

do_this<<<gridsize,blocksize>>>()
```

No arguments  
passed to function  
in this example!

# Thread configuration on the GPU

- How does this information appear on the GPU?
  - Kernel launched with:  
`blocksize = 32,2,1; gridsize = 8,3,1`: 1536 threads total
- 1536 threads are started on the GPU
- Inside the kernel: The size of the blocks and the grid is stored in  
`blockDim` and `gridDim`
- Every thread holds a unique position in the block and grid space
- Every thread **can be uniquely identified with: `threadIdx`, `blockIdx`**
- These 4 variables are again of type `dim3`, and they are predefined in the kernel code

# Kernel

- A kernel is decorated with the **global** attribute
- Every thread executes the same code in parallel
- Block/grid variables are filled with different content

- |                                     | host             |   | kernel          |
|-------------------------------------|------------------|---|-----------------|
| • 1 <sup>st</sup> Chevron argument: | <b>gridsize</b>  | → | <b>gridDim</b>  |
| • 2 <sup>nd</sup> Chevron argument: | <b>blocksize</b> | → | <b>blockDim</b> |

```
module mod_gpu

!*** Use CUDA module
use, intrinsic :: cudafor

contains

attributes(global) subroutine do_this()

! blockDim, griddim, blockidx, threadIdx
! are predefined and of type dim3
```

```
#include <cuda_runtime.h>

__global__ void do_this(void)

/* blockDim, gridDim, blockIdx, threadIdx */
/* are predefined and of type dim3 */
```

# Thread configuration on the GPU

- 1<sup>st</sup> Chevron argument: `gridsize` → `gridDim`
- 2<sup>nd</sup> Chevron argument: `blocksize` → `blockDim`
- Every thread has a unique position in the block and the grid
  - `threadIdx` holds the 3 components of the position in the block
  - `blockIdx` holds the 3 components of the position in the grid
- `blockDim = 32,2,1; gridDim = 8,3,1`

```
Fortran      Thread-#
             1  2  32 33 65

threadIdx%x = 1  2  32  1  1
threadIdx%y = 1  1  1  2  1
threadIdx%z = 1  1  1  1  1
blockIdx%x  = 1  1  1  1  2
blockIdx%y  = 1  1  1  1  1
blockIdx%z  = 1  1  1  1  1
```

```
C           Thread-#
             0  1  31 32 64

threadIdx.x = 0  1  31  0  0
threadIdx.y = 0  0  0  1  0
threadIdx.z = 0  0  0  0  0
blockIdx.x  = 0  0  0  0  1
blockIdx.y  = 0  0  0  0  0
blockIdx.z  = 0  0  0  0  0
```

# Why is this so important?

- The thread-# relates to the loop indices in CPU code
- The translation of loop index to thread indices is crucial
- Remember, a specific thread executes only one loop iteration in this example

# First Kernel

- Let's consider this operation:  $x_i = x_i + 1.1$
- Vector  $x$  holds 256 elements
  - Pseudo code using variables  $x_f$  and  $x_c$ 
    - $x_f$ : 1 - 256
    - $x_c$ : 0 - 255

```
*** 1d arrays with 256 elements
x_f(1..256)
x_c[0..255]

* Fortran
Loop i from 1 to 256
  x_f(i) = x_f(i) + 1.1
Loop end

* C
Loop i from 0 to 255
  x_c[i] = x_c[i] + 1.1
Loop end
```

How do we translate  
this into CUDA?

# First kernel

- Pseudo code and kernels in C and Fortran
- Loop index i from the CPU code translates to index i in the kernel
- Thread configuration variables are used

```
*** 1d arrays with 256 elements
x_f(1..256)
x_c[0..255]

* Fortran
Loop i from 1 to 256
  x_f(i) = x_f(i) + 1.1
Loop end

* C
Loop i from 0 to 255
  x_c[i] = x_c[i] + 1.1
Loop end

Blocksize: x/y/x = 64,1,1
Gridsize:  x/y/z =  4,1,1
```

```
#include <cuda_runtime.h>

__global__ void add(float *x, int n)
{
  /* blockDim, gridDim, blockIdx, threadIdx */
  /*   are predefined and of type dim3      */

  i = threadIdx.x + blockIdx.x * blockDim.x

  x[i] = x[i] + 1.1
}
```

```
attributes(global) subroutine add(x,n)

! blockDim, griddim, blockIdx, threadIdx
!   are predefined and of type dim3

real, dimension(n) :: x
integer, value      :: n

i = threadIdx%x + (blockIdx%x-1) * blockDim%x

x(i) = x(i) + 1.1

end subroutine
```



# Data Copy and Kernel Launch (C)

- 256 results are being calculated
- Each thread calculates 1 result → 256 threads are needed
- Example: 4 blocks with 64 threads each

```
!*** Copy first array to GPU
    x_d = x      ! Arrays were allocated with 256 elements

!*** Kernel launch 4 blocks with 64 threads each
    blocksize = dim3(4, 1, 1)
    gridsize  = dim3(64, 1, 1)

!*** Kernel launch
    call add<<<gridsize,blocksize>>>(x_d, n)

!*** Barrier
    ierr = cudaDeviceSynchronize()

!*** Copy result back from GPU
    x = x_d
```

```
/* copy x to x_d on GPU */
cudaMemcpy(x_d, x, 256*sizeof(float), cudaMemcpyHostToDevice);

/* launch kernel on GPU 4 blocks with 64 threads each */
dim3 blockSize(4);
dim3 gridSize(64);

/* Kernel launch */
add<<<gridSize,blockSize>>>(x_d, n);

/* Barrier */
cudaDeviceSynchronize();

/* copy x_d to x on CPU */
cudaMemcpy(x, x_d, 256*sizeof(float), cudaMemcpyDeviceToHost);
```

# CUDA Mini-Lab

Lars Koesterke  
Hanning Chen

Please find the slides at

# Introduction to a Short Lab

What you will learn

ACES

- Access ~~Lonestar 6~~ and its GPU nodes (Nvidia A30)
- Inspect source code
- Compile code
- Run jobs
- Interpret results

Unfortunately, time does not permit a full lab with source code manipulation

# How to compile and run CPU and GPU demo codes on Texas A&M's Launch cluster

- Step 1: Log onto TAMU's Launch cluster by clicking on <https://portal-aces.hprc.tamu.edu/>
- Step 2: Open a terminal: Clusters > \_launch Shell Access
- Step 3: Copy the demo codes by typing: `cp /scratch/training/tamu_cuda_labs_2024.tar.gz`
- Step 4: Decompress the tar ball: `tar xvfz tamu_cuda_labs_2024.tar.gz`
- Step 5: Read the file named "README" under folder "1D\_ARRAY" for more instructions
- Step 6: To compile the CPU demo code, enter the CPU folder and type: `make -f Makefile_cpu`
- Step 7: To execute the CPU demo code, type: `./runjob_interactive.sh`
- Step 8: Enter the GPU folder and load the CUDA module by typing: `module load CUDA`
- Step 9: To compile the GPU demo code, type: `make -f Makefile_gpu`
- Step 10: To execute the GPU demo code, type: `./runjob_interactive.sh`

## Codes

```
array_cpu.c  
array_gpu.cu
```

# Lab: CPU & 1<sup>st</sup> GPU experiment

```
srun --reservation training_may21 --time=00:04:00 --mem=10G --cpus-per-task=1 --pty bash
```

- 1 • Compile and run the CPU baseline
  - `cd CPU`
  - `make -f Makefile_cpu`
  - `sbatch runjob_CPU.sh`
- 2 • Compile and run the GPU experiment
  - `cd GPU_Experiment_1`
  - `module load cuda`
  - `make -f Makefile_gpu`
  - `sbatch runjob_EXP1.sh`

```
#SBATCH -A training_may21
```

- 3 Inspect the source code
  - Is the CPU code serial or parallel?
  - How many cores does the CPU code use?
  - How many threads are being used on the GPU?
  - How many SM's are being used?
  - Is the GPU code faster?
  - Do you see good scaling?
  - How fast is the data transfer?

CPU code

```
for(unsigned long i = 0; i<SIZE; i++){  
cpu_array[i] = sqrt((float) i) * sqrt((float) i);  
}
```

## Monitor the queues

```
showq -l -u # your jobs  
showq -l | grep gpu-a100 # All jobs in the A100 GPU queue  
# Jobs with run-time = 3 mins are from this lab exercise
```

# Lab: CPU & first GPU experiment (2)

## GPU source code

### GPU kernel

```
__global__ void kernel(float* gpu_array, unsigned long SIZE) {  
// Helper indices:  
//  global_index:  global index in the gpu_array  
//  ntotal_threads: total number of threads  
//  niter:         number of iterations for each GPU thread  
//  ipos:         position of the thread  
//  global_index:  index of array element  
  
unsigned global_index;  
unsigned long ntotal_threads = blockDim.x*blockDim.x;  
unsigned long niter = SIZE / ntotal_threads;  
unsigned long ipos = blockIdx.x*blockDim.x + threadIdx.x;  
for(unsigned long i = 0; i < niter; i++){  
    global_index = ipos + ntotal_threads*i;  
    gpu_array[global_index] = sqrt((float) global_index)*sqrt((float) global_index);  
}
```

### Launching the kernel

```
kernel<<<gridsize,blocksize>>>(gpu_array, SIZE);
```

### Synchronizing the Device

```
cudaDeviceSynchronize();
```

### Copy from device to host

```
cudaMemcpy(cpu_array, gpu_array, SIZE*sizeof(float), cudaMemcpyDeviceToHost);
```

Every threads calculates multiple results

Simple example:

128 elements (SIZE) and 32 threads

Every thread calculates 4 results

niter = 4 (i-loop)

Thread (pos)	Elements
0	0, 32, 64, 96
1	1, 33, 65, 97
...	
31	31, 63, 95, 127

index = (32 x i) + pos

array[index] = <actual calculation>

### The job

```
a.out 320000000 1 64 # 1 x 64 threads # 1 SM 64 threads  
a.out 320000000 2 64 # 2 x 64 threads # 2 SMs 128 threads  
a.out 320000000 4 64 # 4 x 64 threads # 4 SMs 256 threads
```

# Lab: CPU & 1<sup>st</sup> GPU experiment

## 1 Similarities CPU vs GPU code

- Vector lanes = threads
- Lane #0: element 0, 15, 31, etc.
- Cuda-core #0: 0, 31, 63, etc.

2

## Differences

- CPU hides latency by prefetching data → the code should move in order through array
- GPU hides latency by using very large number of threads → no specific order required

3

## Scaling is pretty linear

- 1 SM w/ 64 threads
- 2 SM w/ 128 threads
- 4 SM w/ 256 threads

The linear scaling will end when memory bandwidth is exhausted, which will happen before you employ all 144 SMs.

The situation is similar for parallel CPU code



# Lab: CPU & 2<sup>nd</sup> GPU experiment

## 1 • Compile and run the GPU experiment

- `cd GPU_Experiment_2`
- `make -f Makefile_gpu`
- `sbatch runjob_EXP2.sh`

## 2 • Inspect the job file

### The job

```
a.out 320000000 1 1 # 1 x 1 threads # 1 SM, partial warp
a.out 320000000 1 2 # 1 x 2 threads
a.out 320000000 1 4 # 1 x 4 threads
...
a.out 320000000 1 32 # 1 x 32 threads # 1 SM, 1 warp
a.out 320000000 1 64 # 1 x 64 threads # 1 SM, 2 warps
...
```

## 3 Analysis

- Below 32 threads
  - CUDA-cores of a warp are left idle
  - Linear speedup 1 to 32 threads
- Above 32 threads
  - Multiple warps compete for resources
  - More parallelism helps
  - Scaling quite good

Experiment (on your own)

Write a new job that uses 2048 threads  
What will happen?

# Nvidia Hardware

Comparison of Compute Capability: GP100 vs GV100 vs GA100<sup>[12]</sup>

GPU features	NVIDIA Tesla P100	NVIDIA Tesla V100	NVIDIA A100
GPU codename	GP100	GV100	GA100
GPU architecture	NVIDIA Pascal	NVIDIA Volta	NVIDIA Ampere
Compute capability	6.0	7.0	8.0
Threads / warp	32	32	32
Max warps / SM	64	64	64
Max threads / SM	2048	2048	2048
Max thread blocks / SM	32	32	32
Max 32-bit registers / SM	65536	65536	65536
Max registers / block	65536	65536	65536
Max registers / thread	255	255	255
Max thread block size	1024	1024	1024
FP32 cores / SM	64	64	64
Ratio of SM registers to FP32 cores	1024	1024	1024
Shared Memory Size / SM	64 KB	Configurable up to 96 KB	Configurable up to 164 KB

- A100 GPU allow for
  - Up to 1024 threads in one block
  - One SM can host up to 2048 threads
- The experiment with 1 block and 2048 threads fails

This concludes the lab

# Stencil Update

Homework example that I give to my students

A common operation in scientific computing is digital convolution, by which each element in a multi-dimensional grid is replaced by a weighted sum of its neighbors. This has applications in graphics, in which such operations are done to both blur and sharpen images, and in numerical simulations, in which this may be a single step in a Poisson solver.

Write serial code for CPUs either in Fortran90 or in C. In the next homework you will port the code to the GPU and will compare timings on the CPU and the GPU.

Outline:

Define two 2D arrays ( $x$ ,  $y$ ) in your code. Initialize one array ( $x$ ) with random numbers between 0 and 1. Derive the elements of the second array ( $y$ ) from the first array by smoothing over the elements of the first array, using the constants  $a$ ,  $b$  and  $c$ :

$$\begin{aligned} y(i, j) = & a \cdot (x(i-1, j-1) + x(i-1, j+1) + x(i+1, j-1) + x(i+1, j+1)) + \\ & b \cdot (x(i-1, j+0) + x(i+1, j+0) + x(i+0, j-1) + x(i+0, j+1)) + \\ & c \cdot x(i+0, j+0) \end{aligned}$$

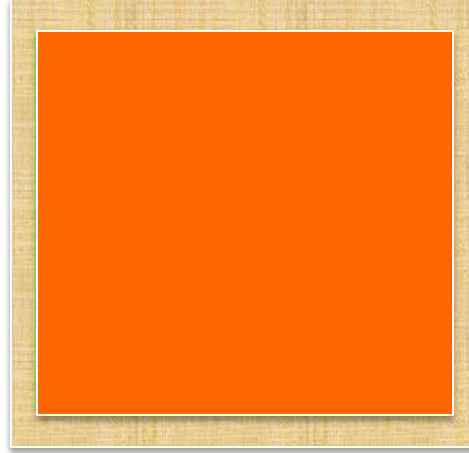
Count the elements that are smaller than a threshold  $t$  in both arrays and print the number for both arrays.

# Serial Code: Stencil Update

- I'll show the whole source code for a serial CPU version on the slides, so that you'll be able to look at the complete code later on.
- I'll focus on the “smoothing” routine, which we will port to the GPU using CUDA

# Data Layout

- Arrays  $x$  and  $y$  with  $(n+2)$  times  $(n+2)$  elements



- Inner elements ( $n$  by  $n$ ) are going to be modified
- Ghost-layers, used in  $x$  and  $y$  direction, are not updated
- Provided boundaries “eliminates” the need for if statements in the code

# Data Allocation

- Allocation on the heap:
  - More flexible
  - Stack size is limited
- Allocation in 1 step (for C programmers: no array of pointers pointing to non-consecutively stored rows or columns)

```
!*** Number of array elements in one direction
integer                :: n = 2**14
!*** Smoothing constants
real                   :: a = 0.05, &
                       b = 0.1, &
                       c = 0.4

!*** Threshold
real                   :: t = 0.1

!*** Input and output array
real, dimension(:,:), allocatable :: x, y

!*** Allocate input array
allocate(x(0:n+1,0:n+1), stat=istat)
allocate(y(0:n+1,0:n+1), stat=istat)
```

```
int n,nbx,nby;
float a,b,c,
      t,
      *x, *y;
Timer timer;

/* n - number of elements in one direction */
n = 1 << 14; /* 2^14 == 16384 */

/* a,b,c - smoothing constants */
a = 0.05;
b = 0.1;
c = 0.4;

/* t - threshold */
t = 0.1;

/* allocate x */
x = new float[(n+2)*(n+2)];

/* allocate y */
y = new float[(n+2)*(n+2)];
```

# Calls to Initialize, Smooth, Count

- All are subroutine / void function calls
- All information through subprogram parameters
- No global variables!

```
!*** Initialize array x  
call initialize(x, n)  
  
!*** Derive second array from first array  
call smooth(y, x, n, a, b, c)  
  
!*** Count elements in first array  
call count(x, n, t, nbx)  
  
!*** Count elements in second array  
call count(y, n, t, nby)
```

```
/* initialize x */  
initialize(x, n);  
  
/* smooth x into y */  
smooth(y, x, n, a, b, c);  
  
/* count elements in first array */  
count(x, n, t, nbx);  
  
/* count elements in second array */  
count(y, n, t, nby);
```



# Initialization and Smoothing

- In C: Index calculation “by hand”
- $x_{i,j} = x[ (n-1)*i + j ]$
- Fortran: `random_number` is an elemental function

The function/subroutine *smooth* will become a kernel later!

```
!*** Initialize with random numbers
subroutine initialize(x, n)
real, dimension(0:n+1,0:n+1) :: x
call random_number(x)
end subroutine

!*** Smooth data
subroutine smooth(y, x, n, a, b, c)
real, dimension(0:n+1,0:n+1) :: x, y
do j=1, n
  do i=1, n
    y(i,j) = a * (x(i-1,j-1) + x(i-1,j+1) + &
                  x(i+1,j-1) + x(i+1,j+1)) + &
              b * (x(i-0,j-1) + x(i-0,j+1) + &
                  x(i-1,j-0) + x(i+1,j+0)) + &
              c * x(i,j)
  enddo
enddo
end subroutine
```

```
void initialize( float* x, int n )
{
  int n2 = n+2;
  for (int i=0; i<n2; ++i)
    for (int j=0; j<n2; ++j)
      x[i*n2 + j] = random() / (float) RAND_MAX;
}

void smooth( float* y, float* x, int n,
            float a, float b, float c )
{
  int n2 = n+2;
  for (int i=1; i<=n; ++i)
    for (int j=1; j<=n; ++j)
      y[i*n2 + j] = a * (x[(i-1)*n2 + (j-1)] +
                        x[(i-1)*n2 + (j+1)] +
                        x[(i+1)*n2 + (j-1)] +
                        x[(i+1)*n2 + (j+1)])
                    + b * (x[i*n2+(j-1)] +
                        x[i*n2+(j+1)] +
                        x[(i-1)*n2 + j] +
                        x[(i+1)*n2 + j])
                    + c * x[i*n2 + j];
}
```

# Counting

- Count could have been a function!
- Return value is communicated through an argument

```
!*** Count elements below threshold
subroutine count(x, n, t, nbx)

real, dimension(0:n+1,0:n+1) :: x

nbx = 0

do j=1, n
  do i=1, n
    if (x(i,j) < t) then
      nbx = nbx + 1
    endif
  enddo
enddo

end subroutine
```

```
void count( float* x, int n, float t, int &nbx )
{
  nbx = 0;
  int n2 = n+2;
  for (int i=1; i <= n; ++i)
    for (int j=1; j <= n; ++j)
      if (x[i*n2 + j] < t)
        ++nbx;
}
```

# Sample Output

The output of your code may look like this:

```
Summary
-----
Number of elements in a row/column      :: 16386
Number of inner elements in a row/column :: 16384
Total number of elements                :: 268500996
Total number of inner elements          :: 268435456
Memory (GB) used per array              :: 1.00024
Threshold                               :: 0.10
Smoothing constants (a, b, c)           :: 0.05 0.10 0.40
Number of elements below threshold (X)   :: 26847453
Fraction of elements below threshold     :: 1.00015E-01
Number of elements below threshold (Y)   :: 2950
Fraction of elements below threshold     :: 1.09896E-05

Action      :: time/s Time resolution = 1.0E-04
-----
CPU: Alloc-X :: 0.000
CPU: Alloc-Y :: 0.000
CPU: Init-X  :: 3.904
CPU: Smooth  :: 1.434
CPU: Count-X :: 0.397
CPU: Count-Y :: 0.384
```

# Formatted Output (Fortran)

```
!*** Print number of elements below threshold
write (0,*)
write (0,'(a)')      'Summary'
write (0,'(a)')      '-----'
write (0,'(a,i14)')  'Number of elements in a row/column      :: ', n+2
write (0,'(a,i14)')  'Number of inner elements in a row/column :: ', n
write (0,'(a,i14)')  'Total number of elements      :: ', (n+2)**2
write (0,'(a,i14)')  'Total number of inner elements  :: ', n**2
write (0,'(a,f14.5)') 'Memory (GB) used per array      :: ', real((n+2)**2 * 4. / (1024.**3))
write (0,'(a,f14.2)') 'Threshold                        :: ', t
write (0,'(a,3(f4.2,1x))') 'Smoothing constants (a, b, c)      :: ', a, b, c
write (0,'(a,i14)')  'Number of elements below threshold (X) :: ', nbx
write (0,'(a,es14.5)') 'Fraction of elements below threshold :: ', real(nbx) / n**2
write (0,'(a,i14)')  'Number of elements below threshold (Y) :: ', nby
write (0,'(a,es14.5)') 'Fraction of elements below threshold :: ', real(nby) / n**2
```

# Formatted Output (C++ & C)

```
std::cout << std::endl;
std::cout << "Summary" << std::endl;
std::cout << "-----" << std::endl;
std::cout << "Number of elements in a row/column      :: " << n+2 << std::endl;
std::cout << "Number of inner elements in a row/column    :: " << n << std::endl;
std::cout << "Total number of elements                        :: " << (n+2)*(n+2) << std::endl;
std::cout << "Total number of inner elements                 :: " << n*n << std::endl;
std::cout << "Memory (GB) used per array                     :: " << (n+2)*(n+2)*sizeof(float) / (float)(1024*1024*1024) << std::endl;;
std::cout << "Threshold                                       :: " << t << std::endl;
std::cout << "Smoothing constants (a, b, c)                 :: " << a << " " << b << " " << c << std::endl;
std::cout << "Number of elements below threshold (X)        :: " << nbx << std::endl;
std::cout << "Fraction of elements below threshold          :: " << nbx / (float)(n*n) << std::endl;
std::cout << "Number of elements below threshold (Y)        :: " << nby << std::endl;
std::cout << "Fraction of elements below threshold          :: " << nby / (float)(n*n) << std::endl;
```

```
printf("\n");
printf("Summary\n");
printf("-----\n");
printf("Number of elements in a row/column      :: %d\n", n+2);
printf("Number of inner elements in a row/column :: %d\n", n);
printf("Total number of elements                :: %d\n", (n+2)*(n+2));
printf("Total number of inner elements          :: %d\n", n*n);
printf("Memory (GB) used per array              :: %g\n", (n+2)*(n+2)*sizeof(float) / (float)(1024*1024*1024));
printf("Threshold                               :: %g\n", t);
printf("Smoothing constants (a, b, c)           :: %g %g %g\n", a,b,c);
printf("Number of elements below threshold (X)  :: %d\n", nbx);
printf("Fraction of elements below threshold     :: %g\n", nbx / (float)(n*n));
printf("Number of elements below threshold (Y)  :: %d\n", nby);
printf("Fraction of elements below threshold     :: %g\n", nby / (float)(n*n));
```

# Timing

- We will be using timers left and right
- You can use a simple timer
- Our implementation uses a timer object.
- What is an object?
- How to declare an object?
- How to call an object?

```
!*** Timing information
class(cls_timer) :: timer

!*** Count elements in second array
call timer%start('CPU: Count-Y')
call count(y, n, t, nby)
call timer%stop
```

```
/* timer of class Timer */
Timer timer;

/* count elements in second array */
timer.start("CPU: Count-Y");
count(y, n, t, nby);
timer.stop();
```

# Timing

- An object is a structure (class)
  - That contains data (instance variables)
  - That has "procedures" attached (methods)
  - The data is hidden and can only be accessed through the methods
  - A method may have arguments
- Name-of-object: **delimiter method**
  - Fortran: **timer%start**
  - C: **timer.start**

```
!*** Timing information
class(cls_timer) :: timer

!*** Count elements in second array
call timer%start('CPU: Count-Y')
call count(y, n, t, nby)
call timer%stop
```

```
/* timer of class Timer */
Timer timer;

/* count elements in second array */
timer.start("CPU: Count-Y");
count(y, n, t, nby);
timer.stop();
```

# Timing

- A class contains **private data** and **public methods**

```
type, public :: cls_timer

private
integer                :: n = 0
integer(8), dimension(2,m) :: it
integer(8)              :: itr
character(len=16), dimension(m) :: c

contains

procedure, public :: reset
procedure, public :: start
procedure, public :: stop
procedure, public :: print

endtype cls_timer

contains

subroutine reset(this)

class(cls_timer) :: this
this%n = 0
this%it = -1
this%c = 'undef'

end subroutine
```

```
class Timer
{
public:
    Timer(): n(0) { }
    void start(std::string label)
    {
        if (n < 20)
            { labels[n] = label; times[2*n] = clock(); }
        else { std::cerr << "No more timers, " << label
            << " will not be timed." << std::endl; }
    }

    void stop() { times[2*n+1] = clock(); n++;}
    void reset() { n=0; }
    void print();

private:
    std::string labels[20];
    float times[40];
    int n;
};
...
```



# Most important method: Printing

```
subroutine print(this)

  class(cls_timer) :: this

  write (0,*)
  write (0,'(a,es7.1)') 'Action          ::  time/s      Time resolution = ', 1./real(this%itr)
  write (0,'(a)')      '-----'
  do i=1, this%n
    write (0,'(a16,a, f7.3)') this%c(i), ' :: ',          &
                               (real(this%it(2,i) - this%it(1,i))) / real(this%itr)
  enddo

end subroutine
```

```
void Timer::print()
{
  std::cout << std::endl;
  std::cout << "Action          ::  time/s      Time resolution = " << 1.f/(float)CLOCKS_PER_SEC << std::endl;
  std::cout << "-----" << std::endl;
  for (int i=0; i < n; ++i)
    std::cout << labels[i] << " :: " << (times[2*i+1] - times[2*i+0])/(float)CLOCKS_PER_SEC << std::endl;
}
```

# Invoke printing by

- Invoking the method `print`

```
!*** Print timings  
call timer%print
```

```
/* Print timings */  
timer.print();
```

- If you need another (probably nested) timer: create another object

```
!*** Timing information  
class(cls_timer) :: timer, timer2  
  
!*** Count elements in second array  
call timer2%start('CPU: Count-XY')  
  call timer%start('CPU: Count-X')  
  call count(x, n, t, nbx)  
  call timer%stop  
  call timer%start('CPU: Count-Y')  
  call count(y, n, t, nby)  
  call timer%stop  
call timer2%stop
```

```
/* timer of class Timer */  
Timer timer timer2;  
  
/* count elements in second array */  
timer2.start("CPU: Count-XY");  
  timer.start("CPU: Count-X");  
  Count(x, n, t, nbx);  
  timer.stop();  
  timer.start("CPU: Count-Y");  
  count(y, n, t, nby);  
  timer.stop();  
timer2.stop();
```

# Header files; Modules

- Fortran: Subroutines/Functions/Class are declared in Modules
  - Modules become available by use association
  - This implies also an order of compilation: Modules first, then code that uses the module
- C/C++: Header files are used to communicate the interface

```
! Example for a module
Module my_mod
Contains
Subroutine do_this(x, y, z)
...
End subroutine
End module

Program main
Use my_mod
Call do_this(a,b,c)
End program
```

```
/* Header files needed for HW0
I/O
timer
strings */
#include <iostream>
#include <sys/time.h>
#include <string>
```

# Converting the Serial Code to CUDA Code

# Why a Stencil Update and not a Matrix-Matrix-Multiply

- Matrix-Matrix-Multiply (MMM) is a very simple example
  - Matrices are divided up into tiles
  - Operations on a tile require only data of that tile (no boundary needed)
  - Most books cover MMM
  - You will almost never code a MMM (or any Linear Algebra routine) yourself, but rather use a library
- The setup of a Stencil update is more complex
  - A stencil update needs the boundaries around a tile
  - The operation is simpler, though

# Pseudo Code

Simple execution scheme: CPU, Transfer, GPU, ...

Loop until work is done

1. Setup data on CPU
2. Transfer input data to GPU
3. Perform work on GPU (3 different schemes)
  1. using "Main Memory"
  2. using "Shared Memory"
  3. Different ways to setup threads
4. Transfer results back from GPU

EndLoop

Efficient Kernel Execution on GPU

# Pseudo Code

Interleaved execution scheme: CPU, **Transfer**, **GPU**, ...

```
Loop until work is done. Parts (1) (2) (3) (4) (...)  
1. Setup (1)  
2. Transfer-in (1)      Setup (2)  
3. GPU-work (1)        Transfer-in (2)  
4. Transfer-out (1)   GPU-work (2)  
5. Transfer-out (2)  
EndLoop
```

Efficient Kernel Execution on GPU  
Efficient Data Transfer

# Pseudo Code

Interleaved execution scheme with CPU: CPU, **Transfer**, **GPU**, ...

Loop until work is done

- |    |                         |                         |              |
|----|-------------------------|-------------------------|--------------|
| 1. | Setup (1)               |                         | CPU-work (3) |
| 2. | <b>Transfer-in (1)</b>  | Setup (2)               | CPU-work (3) |
| 3. | <b>GPU-work (1)</b>     | <b>Transfer-in (2)</b>  | CPU-work (3) |
| 4. | <b>Transfer-out (1)</b> | <b>GPU-work (2)</b>     | CPU-work (3) |
| 5. |                         | <b>Transfer-out (2)</b> | CPU-work (3) |

EndLoop

Efficient Kernel Execution on GPU

Efficient Data Transfer

Simultaneous use of the GPU *and* CPU



# Data Allocation: Fortran

- Declaration with special attribute: **device**
- Allocation with normal **allocate**

```
real, dimension(:, :), allocatable, device :: d_x, &
                                         d_y

!*** Allocate input array on device
call timer%start('GPU: Alloc-D_X')
allocate(d_x(0:n+1,0:n+1), stat=istat)
call timer%stop
if (istat /= 0) then
    write (*,*) 'Allocation of d_x failed'
    stop 'Fatal ERROR in Main Program HW2'
endif

!*** Allocate output array on device
call timer%start('GPU: Alloc-D_Y')
allocate(d_y(0:n+1,0:n+1), stat=istat)
call timer%stop
if (istat /= 0) then
    write (*,*) 'Allocation of d_y failed'
    stop 'Fatal ERROR in Main Program HW2'
endif
```

# Data Allocation: C

- Allocation with special malloc call: `cudaMalloc()`

```
real, dimension(:,,:), allocatable, device

!*** Allocate input array on device
call timer%start('GPU: Alloc-D_X')
allocate(d_x(0:n+1,0:n+1), stat=istat)
call timer%stop
if (istat /= 0) then
    write (*,*) 'Allocation of d_x failed'
    stop 'Fatal ERROR in Main Program HW2'
endif

!*** Allocate output array on device
call timer%start('GPU: Alloc-D_Y')
allocate(d_y(0:n+1,0:n+1), stat=istat)
call timer%stop
if (istat /= 0) then
    write (*,*) 'Allocation of d_y failed'
    stop 'Fatal ERROR in Main Program HW2'
endif
```

```
float *x_d, *y_d;

/* allocate device input array x_d */
timer.start("GPU: Alloc-X_D ");
cudaMalloc((void**) &x_d, (n+2)*(n+2)*sizeof(float));
timer.stop();

/* allocate device output array y_d */
timer.start("GPU: Alloc-Y_D ");
cudaMalloc((void**) &y_d, (n+2)*(n+2)*sizeof(float));
timer.stop();
```

# Why Tiling?

- A GPU has multiple Streaming Multiprocessors (SMs), which are similar to the multiple cores of a CPU
- To use more than 1 SM, the problem must be divided up
- Each SM will work on one (or more) tiles

# Data Copy and Kernel Launch (F90)

- Block size: 16x16
- Assumption: Grid (n) is divisible by 16
- Coding for arbitrary adds a bit of complexity

```
!*** Copy first array to GPU
call timer%start('GPU: Copy-in')
d_x = x
call timer%stop

!*** Kernel launch
nblock      = 16
blocksize   = dim3( nblock,  nblock,  1)
gridsize    = dim3(n/nblock, n/nblock, 1)
call timer%start('GPU: Smooth')
call smooth_gpu<<<gridsize,blocksize>>>(d_y, d_x, n, a, b, c)
ierr = cudaThreadSynchronize()
call timer%stop

!*** Copy result back from GPU
call timer%start('GPU: Copy-out')
yd = d_y
call timer%stop
```

# Data Copy and Kernel Launch (C)

- Block size: 16x16
- Assumption: Grid (n) is divisible by 16
- Coding for arbitrary grid sizes adds a bit of complexity

```
!*** Copy first array to GPU
call timer%start('GPU:
d_x = x
call timer%stop

!*** Kernel launch
nblock      = 16
blocksize = dim3( nbl
gridsize  = dim3(n/nbl
call timer%start('GPU:
call smooth_gpu<<<grid
ierr = cudaThreadSynch
call timer%stop

!*** Copy result back from
call timer%start('GPU:
yd = d_y
call timer%stop

/* copy x_h to x_d on GPU */
timer.start("GPU: Copy-in  ");
cudaMemcpy(x_d, x, (n+2)*(n+2)*sizeof(float), cudaMemcpyHostToDevice);
timer.stop();

/* launch smooth() on GPU */
int nBlocks = 16;
dim3 blockSize(nBlocks, nBlocks);
dim3 gridSize(n/nBlocks, n/nBlocks);
timer.start("GPU: Smooth  ");
smooth_gpu<<<gridSize,blockSize>>>(y_d, x_d, n, a, b, c);
cudaDeviceSynchronize();
timer.stop();

/* copy y_d to y_h on CPU */
timer.start("GPU: Copy-out  ");
cudaMemcpy(y_h, y_d, (n+2)*(n+2)*sizeof(float), cudaMemcpyDeviceToHost);
timer.stop();
```

# Fortran Kernel

- Kernel operates on the data stored in the main (GPU) memory
- Indices i and j are calculated from blockdim and blockidx
- Fortran example shown with full module environment

```
__global__ void smooth_gpu( float* y, float* x, int n, float a, float b, float c )
```

```
module mod_gpu
!*** Use CUDA module
use, intrinsic :: cudafor

contains
!*** Smooth data
attributes(global) subroutine smooth_gpu(d_y, d_x, n, a, b, c)

real, dimension(0:n+1,0:n+1) :: d_x, d_y
real, value :: a, b, c
integer, value :: n
integer :: i, j

i = blockdim%x * (blockidx%x - 1) + threadidx%x
j = blockdim%y * (blockidx%y - 1) + threadidx%y

d_y(i,j) = a * (d_x(i-1,j-1) + d_x(i-1,j+1) + d_x(i+1,j-1) + d_x(i+1,j+1)) + &
           b * (d_x(i-0,j-1) + d_x(i-0,j+1) + d_x(i-1,j-0) + d_x(i+1,j+0)) + &
           c * d_x(i,j)

end subroutine
end module mod_gpu
```

# C Kernel

- Kernel operates on the data in the main (GPU) memory
- Indices i and j are calculated from blockDim and blockIdx

```
module mod_gpu
!*** Use CUDA module
use, intrinsic ::

contains
!*** Smooth data
attributes(global
real, dimension(0
real, value
integer, value
integer

i = blockDim%x *
j = blockDim%y * (blockIdx%y - 1) + threadIdx%y

d_y(i,j) = a * (d_x(i-1,j-1) + d_x(i-1,j+1) + d_x(i+1,j-1) + d_x(i+1,j+1)) + &
           b * (d_x(i-0,j-1) + d_x(i-0,j+1) + d_x(i-1,j-0) + d_x(i+1,j+0)) + &
           c * d_x(i,j)

end subroutine
end module mod_gpu
```

```
__global__ void smooth_gpu( float* y, float* x, int n, float a, float b, float c )
{
    int n2 = n+2, i, j;

    i = blockDim.x * blockIdx.x + threadIdx.x + 1; // add one to start from index 1
    j = blockDim.y * blockIdx.y + threadIdx.y + 1;

    y[i*n2 + j] = a * (x[(i-1)*n2 + (j-1)] + x[(i-1)*n2 + (j+1)] +
                       x[(i+1)*n2 + (j-1)] + x[(i+1)*n2 + (j+1)])
                  + b * (x[i*n2+(j-1)] + x[i*n2+(j+1)] +
                       x[(i-1)*n2 + j] + x[(i+1)*n2 + j])
                  + c * x[i*n2 + j];
}
```

# Using “Shared Memory”

- If data is re-used a huge performance gain can be achieved by copying sections of the data into the (small) shared memory of the SM
- In the example every element is used 9 times to calculate the average
- In other words, each cell can be the northern, southern, eastern, or western, or diagonal' neighbor of an element
- I will show how allocate shared memory, and how to copy (inside the kernel), data from main GPU memory to the shared memory
- Note: In a Matrix-Matrix-Multiply every element is used  $n$  times (loosely speaking)



# Using “Shared Memory”

- I will employ 3 different strategies
  1. Simple, but very slow: Only one thread is copying the data
  2. Fast (16x16 threads):
    1. Most threads copy one element
    2. Some threads are copying more than one element
    3. All threads are calculating one result
  3. Fast (18x18) threads:
    1. All threads are copying exactly one element
    2. Not all threads are calculating a result

– Remember: A tile contains 16x16 elements, but the data covers an 18x18 patch (Boundary condition)

  - 16x16 elements are calculated from 18x18 elements



# Shared Memory: Fortran Kernel (sh1)

- Only the **first thread** copies data to shared memory
- **Synchronization** prevents threads from moving ahead
- **Shared memory holds 18x18 elements**

```
attributes(global) subroutine smooth_gpu_sh1(d_y, d_x, n, a, b, c)
  integer, value                :: n
  integer                       :: i, j
  real, dimension(0:n+1,0:n+1)  :: d_x, d_y
  real, dimension(0:17,0:17), shared :: xb
  real, value                   :: a, b, c
  k = blockdim%x * (blockidx%x - 1) + threadidx%x
  l = blockdim%y * (blockidx%y - 1) + threadidx%y
  i = threadidx%x
  j = threadidx%y
  *** Data copy
  if (i == 1 .and. j == 1) then
    do n2=0, blockdim%y+1
      do n1=0, blockdim%x+1
        xb(n1,n2) = d_x(k+n1-1,l+n2-1)
      enddo
    enddo
  endif
  *** Synchronize
  call syncthreads()
  *** Action
  d_y(k,l) = a * (xb(i-1,j-1) + xb(i-1,j+1) + xb(i+1,j-1) + xb(i+1,j+1)) + &
             b * (xb(i-0,j-1) + xb(i-0,j+1) + xb(i-1,j-0) + xb(i+1,j+0)) + &
             c * xb(i,j)
end subroutine
```

# Shared Memory: C Kernel (sh1)

- Only the **first thread** copies data to shared memory
- **Synchronization** prevents threads from moving ahead
- **Shared memory holds 18x18 elements**

```
__global__ void smooth_gpu_sh1( float* y, float* x, int n, float a, float b, float c )
{
    int n2 = n+2, i, j, k, l, block=18;
    __shared__ float xb[18*18];

    k = blockDim.x * blockIdx.x + threadIdx.x + 1; // add one to start from index 1
    l = blockDim.y * blockIdx.y + threadIdx.y + 1;
    i = threadIdx.x + 1;
    j = threadIdx.y + 1;

    /* copy data into shared buffer */
    if (i == 1 & j==1)
    {
        for (int ii=0; ii <= (blockDim.x+1); ++ii)
            for (int jj=0; jj <= (blockDim.y+1); ++jj)
                xb[ii*block + jj] = x[(k+ii-1)*n2 + (l+jj-1)];
    }
    __syncthreads();

    y[l*n2 + k] = a * (xb[(j-1)*block+(i-1)] + xb[(j-1)*block+(i+1)] +
                    xb[(j+1)*block+(i-1)] + xb[(j+1)*block + (i+1)])\
        + b * (xb[j*block + (i-1)] + xb[j*block + (i+1)] +
                xb[(j-1)*block + i] + xb[(j+1)*block + i])
        + c * xb[j*block + i];
}
```

# Shared Memory: Calling the Kernel (sh1)

- Block size: 16x16
- Assumption: Grid (n) is divisible by 16

```
!*** Block size: A tile must be 16x16!  
    nblock    = 16  
    blockSize = dim3( nblock,  nblock, 1)  
    gridSize  = dim3(n/nblock, n/nblock, 1)  
  
    call smooth_gpu_sh1<<<gridSize,blockSize>>>(d_y, d_x, n, a, b, c)
```

```
{  
    nBlocks    = 16;  
    blockSize  = dim3( nBlocks,  nBlocks);  
    gridSize   = dim3( n/nBlocks), n/nBlocks) );  
    smooth_gpu_sh1<<<gridSize,blockSize>>>(y_d, x_d, n, a, b, c);  
}
```

# Shared Memory: F90 Kernel (sh16)

```
attributes(global) subroutine smooth_gpu_sh16(d_y, d_x, n, a, b, c)
integer, value :: n
integer :: i, j
real, dimension(0:n+1,0:n+1) :: d_x, d_y
real, dimension(0:17,0:17), shared :: xb
real, value :: a, b, c
k = blockdim%x * (blockidx%x - 1) + threadidx%x
l = blockdim%y * (blockidx%y - 1) + threadidx%y
i = threadidx%x
j = threadidx%y
!*** Copy inner part: 16x16 elements
xb(i,j) = d_x(k,l)
!*** Copy boundaries
!*** : x + corners
if (i == 1) then
  xb(i-1,j) = d_x(k-1,l)
  if (j == 1) then
    xb(i-1,j-1) = d_x(k-1,l-1)
    xb(i-1,j+1) = d_x(k-1,l+1)
  else if (j == blockdim%y) then
    xb(i-1,j+1) = d_x(k-1,l+1)
  endif
endif
```

Each thread copies one element

Threads close to boundary copy boundary

Threads close to the corner copy corner

```
else if (i == blockdim%x) then
  xb(i+1,j) = d_x(k+1,l)
  if (j == 1) then
    xb(i+1,j-1) = d_x(k+1,l-1)
    xb(i+1,j+1) = d_x(k+1,l+1)
  else if (j == blockdim%y) then
    xb(i+1,j+1) = d_x(k+1,l+1)
  endif
endif
!*** : y
if (j == 1) then
  xb(i,j-1) = d_x(k,l-1)
else if (j == blockdim%y) then
  xb(i,j+1) = d_x(k,l+1)
endif

!*** Synchronize
call syncthreads()

!*** Action
d_y(k,l) = a * (xb(i-1,j-1) + xb(i-1,j+1) + etc.

end subroutine
```

# Shared Memory: C Kernel (sh16)

```
__global__ void smooth_gpu_sh16( float* y, float* x, int n, float a, float b, float c )
{
    int n2 = n+2, i, j, k, l, block=18;
    __shared__ float xb[18*18];
    k = blockDim.x * blockIdx.x + threadIdx.x + 1; // add one to start from index 1
    l = blockDim.y * blockIdx.y + threadIdx.y + 1;
    i = threadIdx.x + 1;
    j = threadIdx.y + 1;
    /* copy inner part: block x block elements */
    xb[j*block + i] = x[l*n2 + k];
    /* copy boundaries */
    /* x and corners */
    if (i == 1)
    {
        xb[j*block + (i-1)] = x[l*n2 + (k-1)];
        if (j == 1)
        {
            xb[(j-1)*block + (i-1)] = x[(l-1)*n2 + (k-1)];
            xb[(j+1)*block + (i-1)] = x[(l+1)*n2 + (k-1)];
        }
        else if (j == blockDim.y)
        {
            xb[(j+1)*block + (i-1)] = x[(l+1)*n2 + (k-1)];
        }
    }
}
```

```
else if (i == blockDim.x)
{
    xb[j*block + (i+1)] = x[l*n2 + (k+1)];
    if (j == 1)
    {
        xb[(j-1)*block + (i+1)] = x[(l-1)*n2 + (k+1)];
        xb[(j+1)*block + (i+1)] = x[(l+1)*n2 + (k+1)];
    }
    else if (j == blockDim.y)
    {
        xb[(j+1)*block + (i+1)] = x[(l+1)*n2 + (k+1)];
    }
}
/* y */
if (j == 1)
    xb[(j-1)*block + i] = x[(l-1)*n2 + k];
else if (j == blockDim.y)
    xb[(j+1)*block + i] = x[(l+1)*n2 + k];
__syncthreads();
y[l*n2 + k] = a * (xb[(j-1)*18 + (i-1)] + etc.
}
```

Each thread copies one element

Threads close to boundary copy boundary

Threads close to the corner copy corner

# Shared Memory: Calling the Kernel (sh16)

- Block size: 16x16
- Assumption: Grid (n) is divisible by 16

```
!*** Block size: A tile must be 16x16!  
    nblock    = 16  
    blocksize = dim3( nblock,  nblock, 1)  
    gridsize  = dim3(n/nblock, n/nblock, 1)  
  
    if (ckernel == 'global') then  
        call timer%start('GPU: Smooth')  
        call smooth_gpu16<<<gridsize,blocksize>>>(d_y, d_x, n, a, b, c)
```

```
{  
    nBlocks = 16;  
    blockSize = dim3(nBlocks, nBlocks);  
    gridSize = dim3( (int)ceilf(n/(float)nBlocks), (int)ceilf(n/(float)nBlocks) );  
    smooth_gpu_sh16<<<gridSize,blockSize>>>(y_d, x_d, n, a, b, c);  
}
```

# Calling the first three kernels

1. No shared memory (16x16 threads)
2. Shared memory (16x16 threads)
  - One thread copies data (sh1)
  - All threads are copying data; some multiple elements
- To call these three kernels the same block/grid structure is used
  - Blocksize : 16x16
  - Grid divided in tiles of 16x16 elements



# Shared Memory: Calling the Kernel (sh18)

- Now, we would like to have **18x18 threads**
  - Each thread will copy exactly one data element
  - Not all threads are joining the computation
- Tile is still **16x16**

```
!*** Block size: A tile must be 16x16, but the thread block holds 18x18 threads
nblock = 18
blocksize = dim3(nblock, nblock, 1)
gridsize = dim3(n/16, n/16, 1)

call timer%start('GPU: Smooth')
call smooth_gpu_sh18<<<gridsize,blocksize>>>(d_y, d_x, n, a, b, c)
ierr = cudaThreadSynchronize()
call timer%stop
```

```
{
    nBlocks = 18;
    blockSize = dim3(nBlocks, nBlocks);
    gridSize = dim3( n/16, n/16 );
    smooth_gpu_sh18<<<gridSize,blockSize>>>(y_d, x_d, n, a, b, c);
}
```

# Shared Memory: Fortran Kernel (sh18)

- Each thread copies exactly one element
- Only the first 256 threads calculate

```
attributes(global) subroutine smooth_gpu_sh18(d_y, d_x, n, a, b, c)
  integer, value                :: n
  integer                       :: i, j
  real, dimension(0:n+1,0:n+1)  :: d_x, d_y
  real, dimension(0:17,0:17), shared :: xb
  real, value                   :: a, b, c

  k = 16 * (blockidx%x - 1) + threadidx%x - 1
  l = 16 * (blockidx%y - 1) + threadidx%y - 1
  i = threadidx%x
  j = threadidx%y
  *** Copy tile from GPU memory to shared memory
  xb(i-1,j-1) = d_x(k,l)
  *** Synchronize
  call syncthreads()
  *** Linear thread index
  lind = blockdim%x * (threadidx%y - 1) + threadidx%x
  *** Select first 256 threads
  if (lind <= 256) then
    j = (lind-1) / 16 + 1
    i = lind - (j-1) * 16
    k = 16 * (blockidx%x - 1) + i
    l = 16 * (blockidx%y - 1) + j
  *** Action
    d_y(k,l) = a * (xb(i-1,j-1) + etc.
  endif
end subroutine
```

# Shared Memory: C Kernel (sh18)

- Each thread copies exactly one element
- Only the first 256 threads calculate

```
__global__ void smooth_gpu_sh18( float* y, float* x, int n, float
a, float b, float c )
{
    int n2 = n+2, i, j, k, l, lind;
    __shared__ float xb[18*18];

    k = 16 * blockIdx.x + threadIdx.x;
    l = 16 * blockIdx.y + threadIdx.y;
    i = threadIdx.x + 1;
    j = threadIdx.y + 1;

    /* copy tile to shared memory */
    xb[threadIdx.y*18 + threadIdx.x] = x[l*n2 + k];

    __syncthreads();

    /* compute linear thread index */
    lind = blockDim.x * threadIdx.y + threadIdx.x + 1;

    if (lind <= 256)
    {
        i = (lind-1) / 16 + 1;
        j = lind - (i-1) * 16;
        k = 16 * blockIdx.x + i;
        l = 16 * blockIdx.y + j;

        y[l*n2 + k] = a * (xb[(j-1)*18 + (etc.
    }
}
```

# Performance

- Which kernel is the fastest?
- Obviously, using only 1 thread for the copy is slow
  - I added this one only for educational purposes
  - Any code that uses only 1 thread on an SM is slow
- But for the other 2 kernels it is not obvious, which one is faster
  - Analysis of the occupancy: registers, shared memory, etc.
  - Trial and error

# Overlapping communication and computation

- With an efficient kernel at hand, we can now tackle the second step
- To overlap communication with computation, we need
  - Break the problem into several pieces (4 in the example)
- This requires to go from a 2d array to a 3d array
  - 2d for the data, the last dimension (set to 4) is for the 4 pieces
- We can use the kernels that we have written, but we need to
  - Call the kernels for one 2d slab of the 3d array
  - Setup the data streams

This topic will become much less important in the future

- Nvidia's new architecture (Grace-Hopper) integrates CPU and GPU into one device
- Host can see GPU memory
- GPU can see host memory

Other vendors (AMD, later Intel) are doing the same

# Grace-Hopper architecture

CPU: Grace (ARM architecture)

72 cores

GPU: Hopper (based on H100)

144 SMs

CPU memory: LPDDR5

GPU memory: HBM3

NVLINK between Grace and Hopper

One address space

- CPU can access HBM memory
- GPU can access DDR memory
- Bandwidth decreased and latency increased for 'cross' access

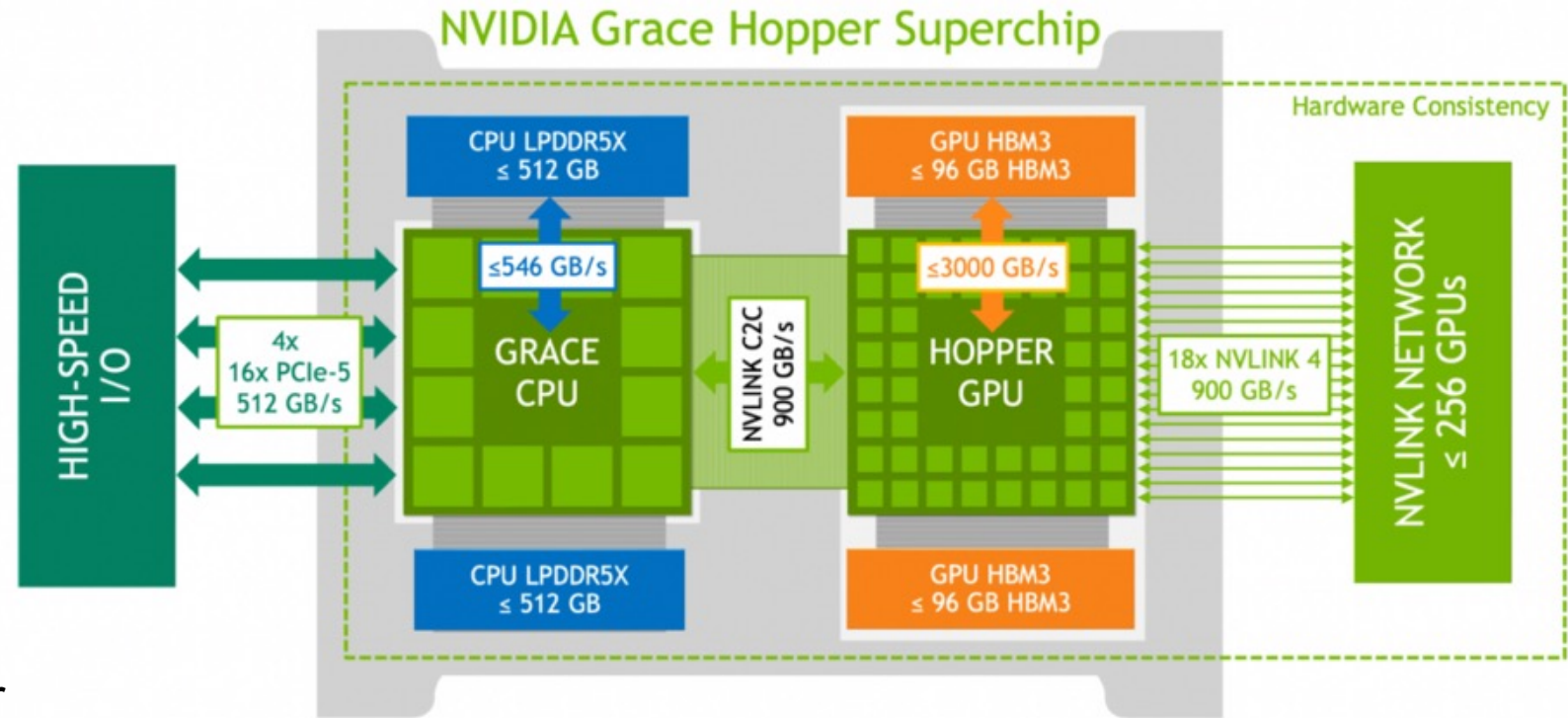
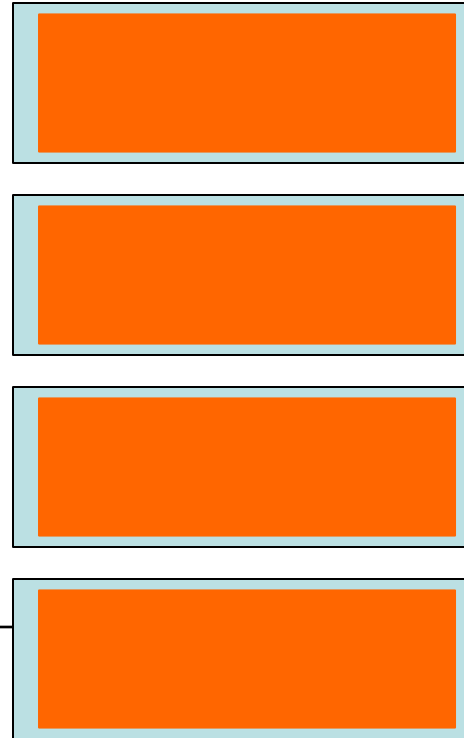
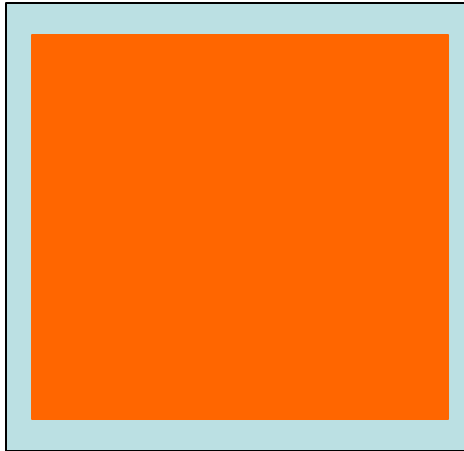


Figure 2. NVIDIA Grace Hopper Superchip logical overview

# Partitioning of the Problem

- Inner elements ( $n \times n$ ) are going to be modified
- “2D” array (square)  $\rightarrow$  “3D” array (rectangles  $\times$  “number of streams”)
- Example with 4 streams:  $n_s = n/4$ ; Inner elements:  $n \times n_s$  or  $n_s \times n$
- All rectangles need their own “boundaries”



Not relevant for new architectures  
with unified shared memory

# Streams

- CUDA allows you to execute multiple “streams” simultaneously in your code
- Parts of a stream are: (that we are interested in)
  - Asynchronous data transfer operations (default is synchronous)
  - Asynchronous kernel execution (default is asynchronous)
- How would this be executed if 4 streams are used?

“Pseudo code”

```
Loop over #-of streams: istream
  Asynchronous Copy xs{istream} to d_xs{istream}
  Kernel call for istream: smooth_gpu_sh1_p "Calculate d_ys from d_xs"
  Asynchronous Copy d_ys{istream} to ys{istream}
End Loop

Wait
```

Not relevant for new architectures  
with unified shared memory

- All 12 “operations” would be initiated at once
- GPU scheduler would decide in what order to execute
- The dependences inside a stream is maintained



# Stream Creation

- Integer array that holds stream identifier
- Call to `cudaStreamCreate` creates a stream and returns an identifier, which is an integer number

Fortran

```
integer, dimension(4) :: id_stream  
  
do istream=1, 4  
    istat = cudaStreamCreate(id_stream(istream))  
enddo
```

C/C++

```
int id_stream[4];  
  
for (int istream=0; istream<4; ++i)  
    cudaStreamCreate(id_stream[istream]);
```

Not relevant for new architectures  
with unified shared memory

# Asynchronous copy

- Very similar the synchronous copy, but **stream identifier is added**

Not relevant for new architectures  
with unified shared memory

```
"istat = " cudaMemcpyAsync(destination, source, "size", direction, id_stream(istream))
```

#### Notes Fortran:

Call returns an integer

Size is the number of elements

Direction is optional

#### Notes C/C++:

Function has no return value

Size is in bytes

Direction is not optional

# Kernel Invocation with Stream Reference

- Very similar to the normal Kernel launch
- 2 parameters added
- Size for shared memory transactions
- Stream ID

Not relevant for new architectures  
with unified shared memory

```
"call" kernel<<<gridsize, blocksize, 0, id_stream(istream)>>>()
```

Notes Fortran:  
Subroutine call

Notes C/C++:  
Function call

# Events

- Events work like “Barriers”
- An event is recorded when the code “passes” the barrier
- A barrier can be passed when all streams have finished operating (stream number = 0)
- A barrier can be passed when a particular stream has finished (stream number >0)
- Events are of the type cudaEvent
- Syntax: “istat = “ cudaEventRecord(Event, stream-number)

Not relevant for new architectures with unified shared memory

Declaration: An event is of a special type

Fortran:

```
type(cudaEvent) :: startEvent, stopEvent
```

C/C++:

```
cudaEvent startEvent, stopEvent;
```

## Example

```
"istat =" cudaEventRecord(startEvent, 0)  
"Start Timer"
```

```
"Do something with streams"
```

```
"Code will wait at the barrier until all  
stream operations are finished"
```

```
"istat =" cudaEventRecord(stopEvent, 0)  
"Stop timer"
```

# Hardware and Scheduling

- Different scheduling ordering
- Scheme 1: One loop with Copy-Kernel-Copy
- Scheme 2: Three loops; Loop bodies are: Copy, Kernel, or Copy

"Pseudo code: Scheme 1"

```
Loop over #-of streams: istream
  Asynchronous Copy xs{istream} to d_xs{istream}
  Kernel call for istream: smooth_gpu_shl_p "Calculate d_ys from d_xs"
  Asynchronous Copy d_ys{istream} to ys{istream}
End Loop
```

Not relevant for new architectures  
with unified shared memory

"Pseudo code: Scheme 2"

```
Loop over #-of streams: istream
  Asynchronous Copy xs{istream} to d_xs{istream}
End Loop
Loop over #-of streams: istream
  Kernel call for istream: smooth_gpu_shl_p "Calculate d_ys from d_xs"
End Loop
Loop over #-of streams: istream
  Asynchronous Copy d_ys{istream} to ys{istream}
End Loop
```

# Lonestar

- GPUs have multiple copy engines
  - In this example, one is used to copy data in, one to copy data out

Loop over all 'blocks' (This is how your stream is organized in the code)

Copy-in

Compute

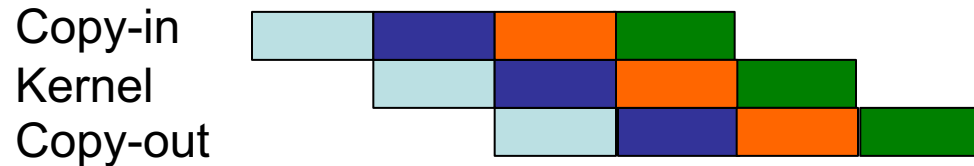
Copy-out

End loop

4 different blocks of work

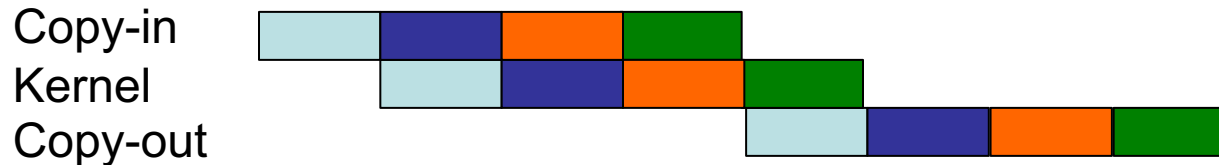
1 2 3 4

Engine With multiple copy engines



Not relevant for new architectures with unified shared memory

Engine If there was only one copy engine (old hardware)



# Using Host and GPU together

- Simple expansion of the “overlap” concept
- Think of dividing the domain into 5 pieces
  - 4 would be executed on the GPU (w/ overlap)
  - The 5<sup>th</sup> piece is executed on the host
- Challenges
  - Load-balancing
  - For static problems
    - Determine the relative speeds
    - Adjust the tile sizes accordingly
- In real applications this will may not be applicable

# Multiple nodes (MPI)

- One node with a GPU is already a cluster
  - Multiple memories, multiple address spaces
- Easiest way to expand to a cluster
  - Use MPI to divide the work among the nodes and GPUs
  - Potentially use CUDA to divide work assigned to a node between the host CPU(s) and the GPU(s)
  - A process on the host communicates with
    - MPI processes communicate with other MPI tasks
    - ‘Processes’ / kernels on the GPU communicate with kernels on other GPUs
      - Direct communication, host not involved
- MPI is the outer frame-work
- CUDA kernels are on the inside
  - CUDA kernels may contain MPI calls
- Comparable to: MPI/OpenMP — Outside/Inside



# Summary

- Writing efficient code for GPUs requires
  - An efficient kernel that uses (if applicable) shared memory (or texture/constant)
  - Efficient data transfer that overlaps with computations
  - Uses the cores of the CPU while also asynchronously executing on the GPU; applies to HPC clusters that have powerful hosts (most clusters available in XSEDE)
- Translation of host code to CUDA
  - Use a grid of blocks to keep all SMs busy; an SM executes one or more blocks
  - Use the blocks to keep all CUDA cores in an SM busy; execution in lockstep, like the Vector lanes of a CPU core
  - Attention to detail is required

Thank You