Computing on GPUs

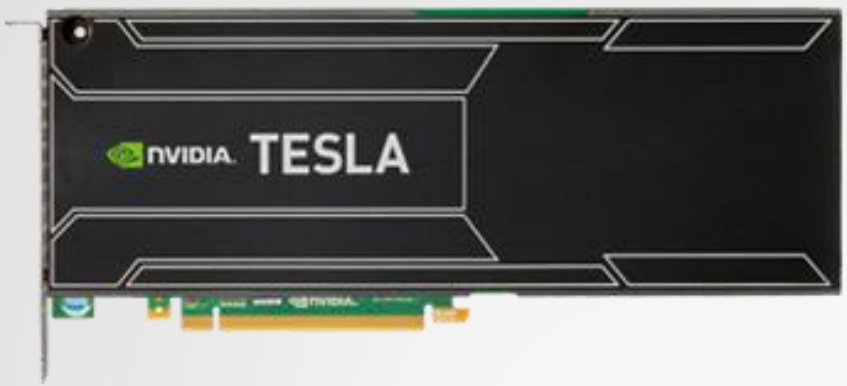
Loren Schwiebert
Department of Computer Science
Wayne State University
loren@wayne.edu

DESKTOP GPU – GTX 970

- ✓ 1664 Processor Cores
- ✓ 1050 MHz/1178 MHz Clock Rate
- ✓ 4 GB GDDR5 Memory
- ✓ 256-bit Memory Bus Width
- ✓ 224 GB/s Memory Bandwidth
- ✓ List Price: $449 in Dec. 2017

- ✓ 13 Streaming Multiprocessors
  Each has:
  - ✓ 128 Processor Cores
  - ✓ 32 Special Function Units
  - ✓ 64 KB of Shared Memory
  - ✓ 24 KB of L1 Cache
  - ✓ 64K 32-bit registers
  - ✓ 8 Instruction Issues/Cycle

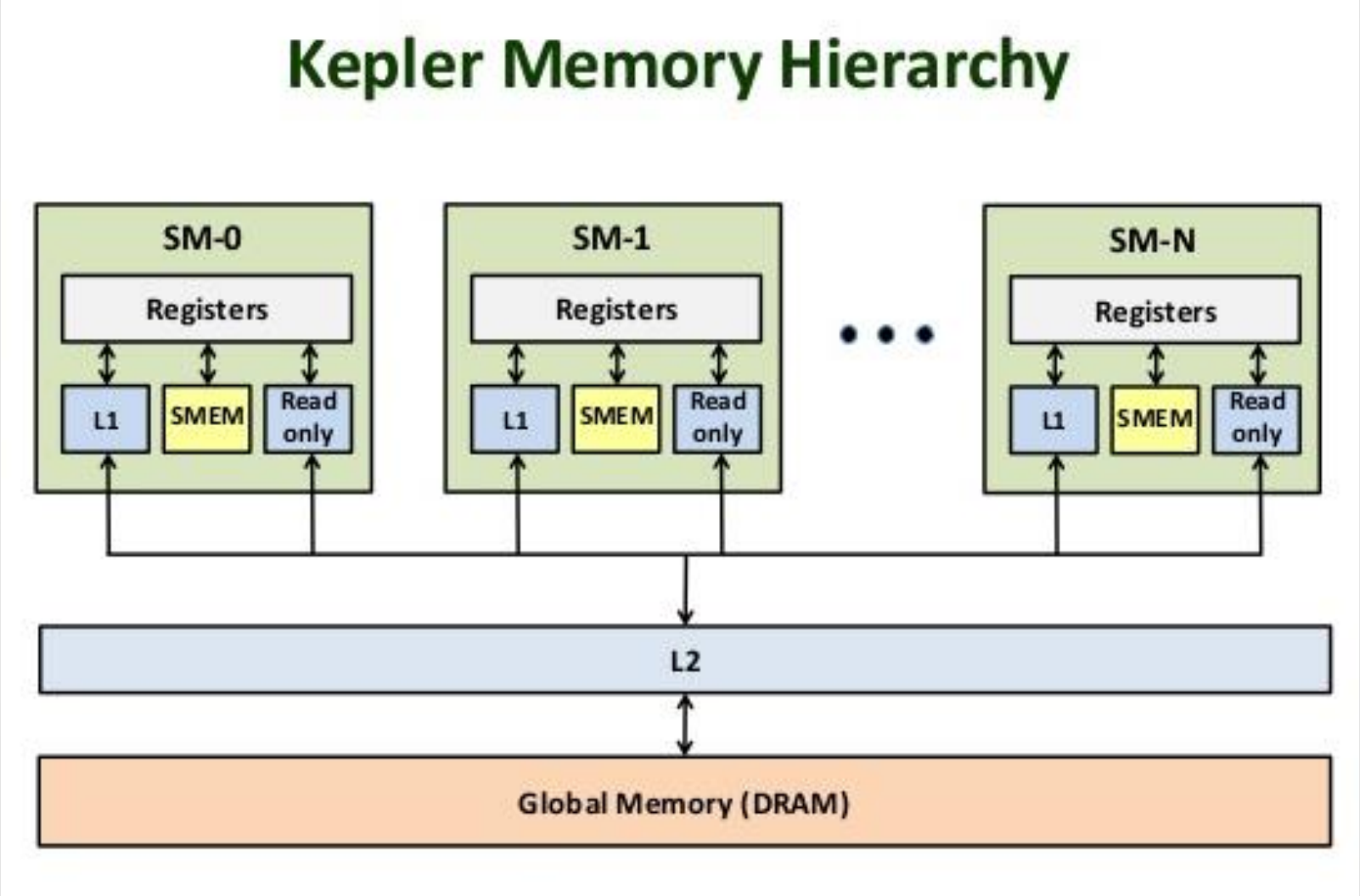
HIGH-END COMPUTE GPU – Kepler K40



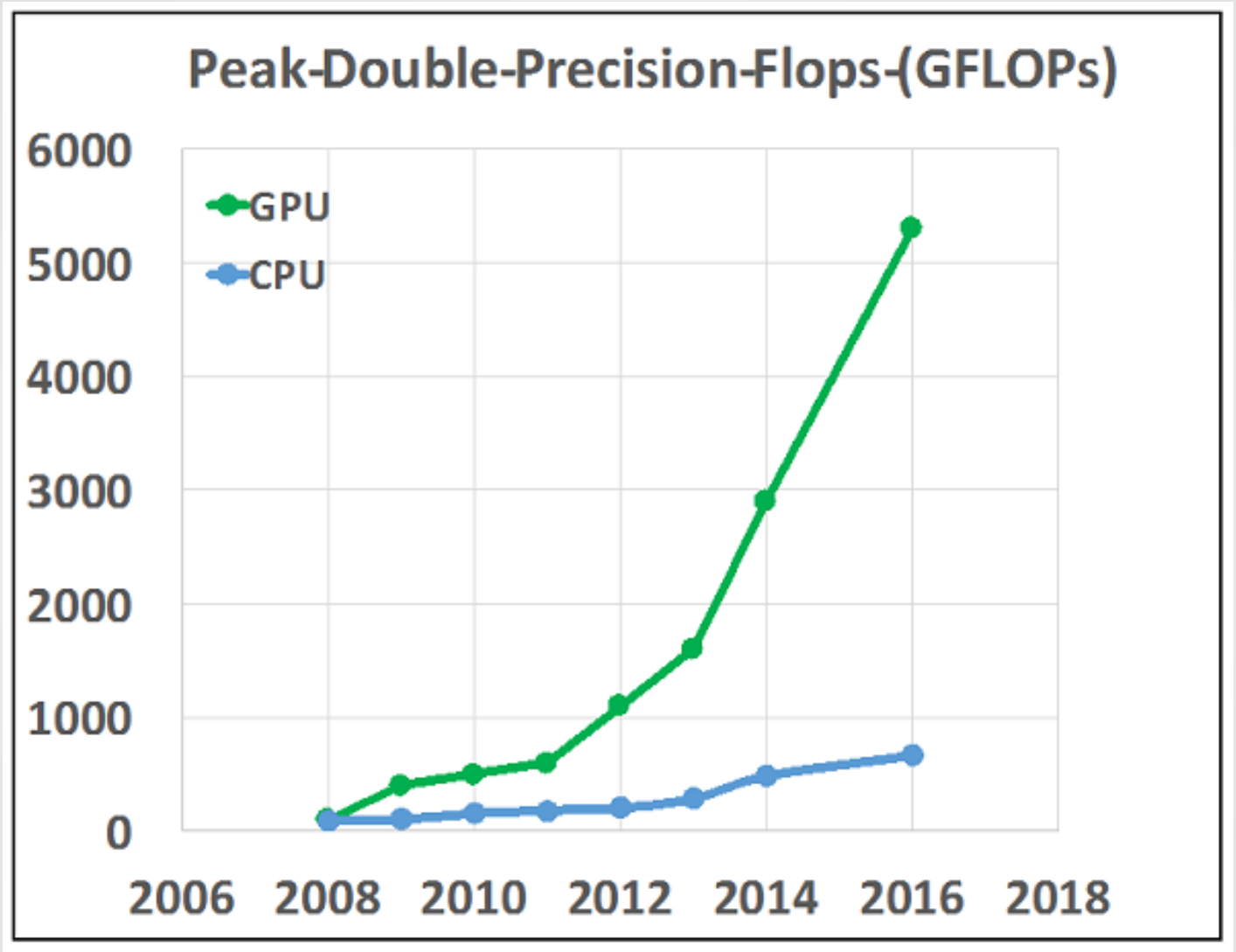
- ✓ 2880 Processor Cores
- ✓ 745 MHz Clock Rate (+Boost)
- ✓ 12 GB DDR5
- ✓ 384-bit Memory Bus Width
- ✓ 288 GB/s Memory Bandwidth
- ✓ List Price: $3400 in Dec. 2017

- ✓ 15 Streaming Multiprocessors
  Each has:
  - ✓ 192 Processor Cores
  - ✓ 32 Special Function Units
  - ✓ 48 KB of Shared Memory
  - ✓ 16 KB of L1 Cache
  - ✓ 64K 32-bit registers
  - ✓ 8 Instruction Issues/Cycle

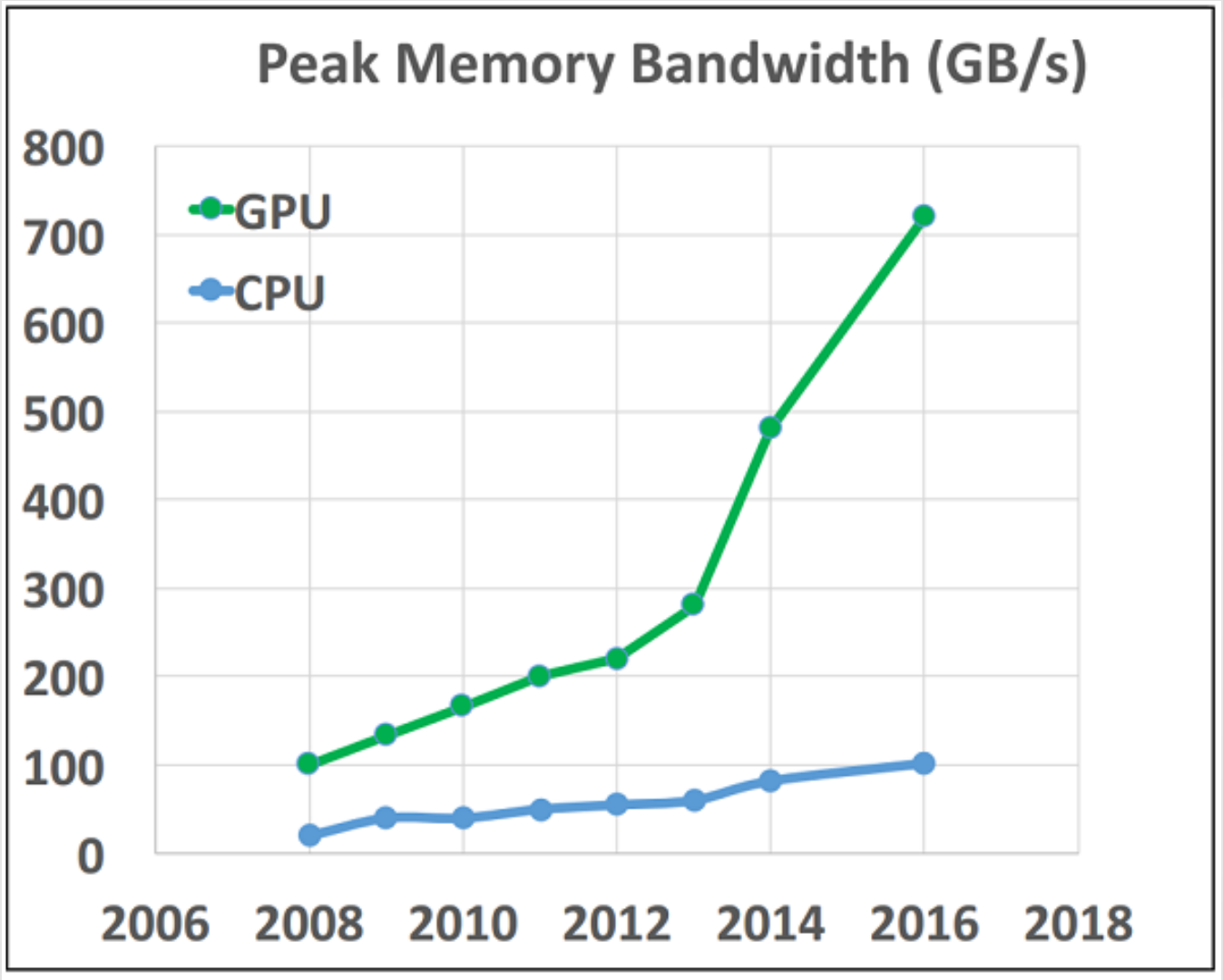
MEMORY CONFIGURATION



GPU VS. CPU FP PERFORMANCE



GPU VS. CPU MEMORY BANDWIDTH



GPU PROGRAMMING

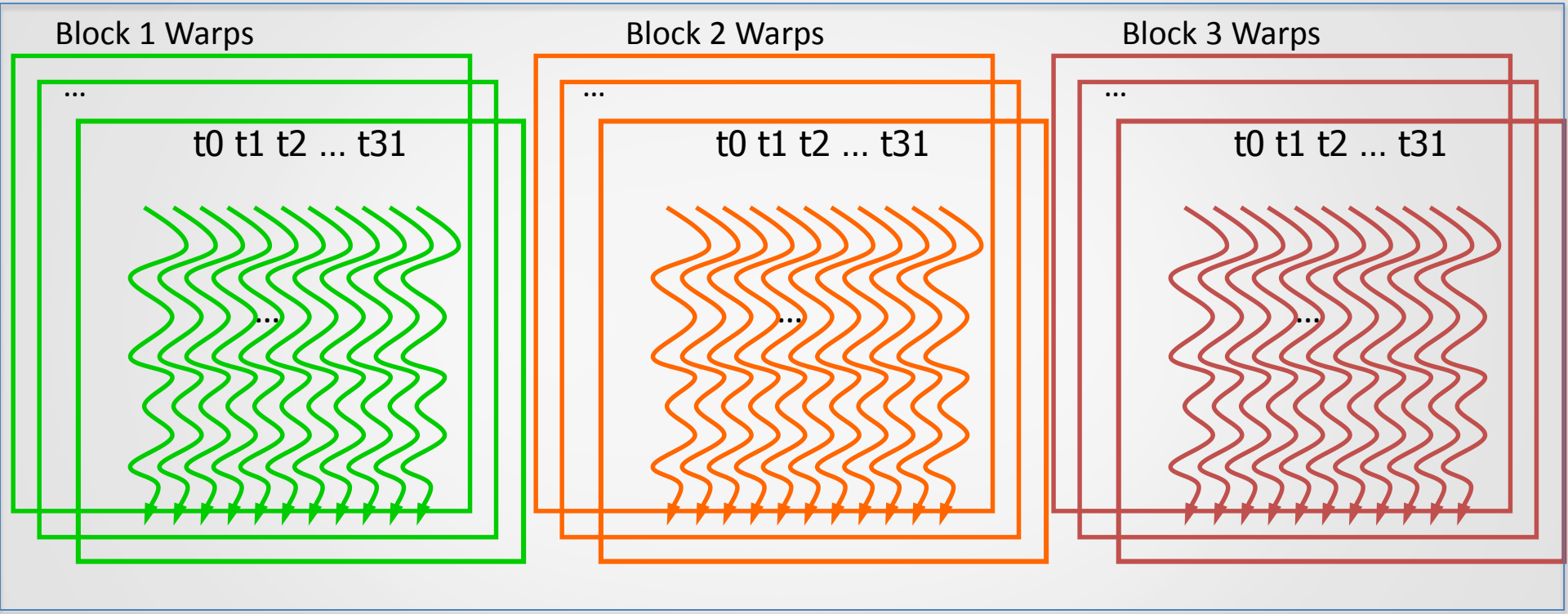
- CUDA: https://developer.nvidia.com/cuda-zone
  - Programming API.
  - NVIDIA GPUs only. Not an open standard.
  - Offers many advanced features.
- OpenCL: https://www.khronos.org/opencl/
  - Open Standard.
  - Also supports other multicore and many-core architectures: AMD-ATI GPUs, Xeon Phi.
- OpenACC: https://www.openacc.org/
  - Open Standard.
  - Programming based on directives: #pragma

GPU PROGRAMMING

- ✓ A multi-threaded many-core model of computation
  - ✓ Threads are lightweight processes
  - ✓ Each thread solves part of the problem – data parallel apps
- ✓ The threads can all run simultaneously – massively parallel
  - ✓ You can create millions of threads
- ✓ A grid contains all the threads
- ✓ The grid is divided into blocks
- ✓ Each block has many threads
  - ✓ Each block runs independently
  - ✓ Blocks are assigned to streaming multiprocessors (SM)
  - ✓ Can have more than one block on the same SM
  - ✓ Run as many blocks simultaneously as possible
- ✓ Threads are grouped into warps

THREAD MODEL

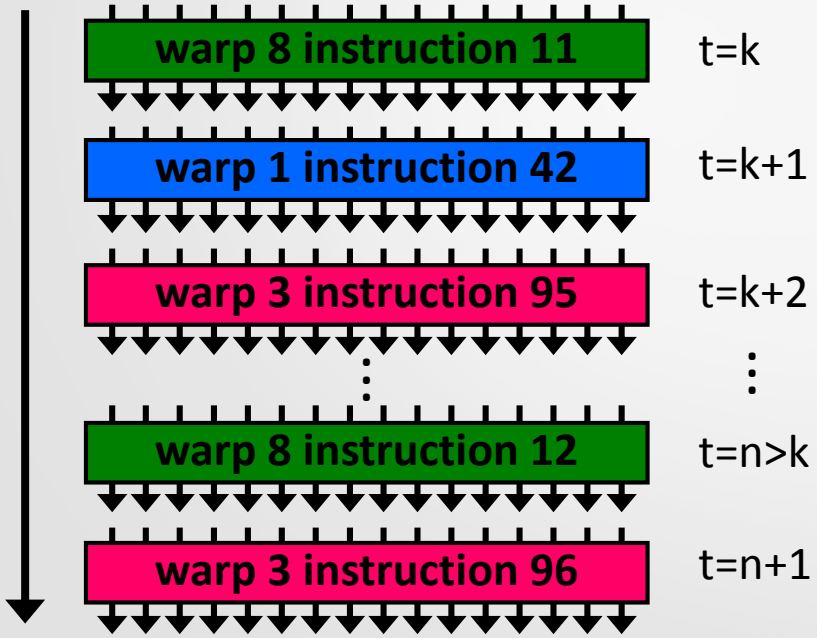
The Grid



WARP SCHEDULING EXAMPLE

- Consider three separate instruction streams on one streaming multiprocessor: warp1, warp3, and warp8

OK, but why warps??

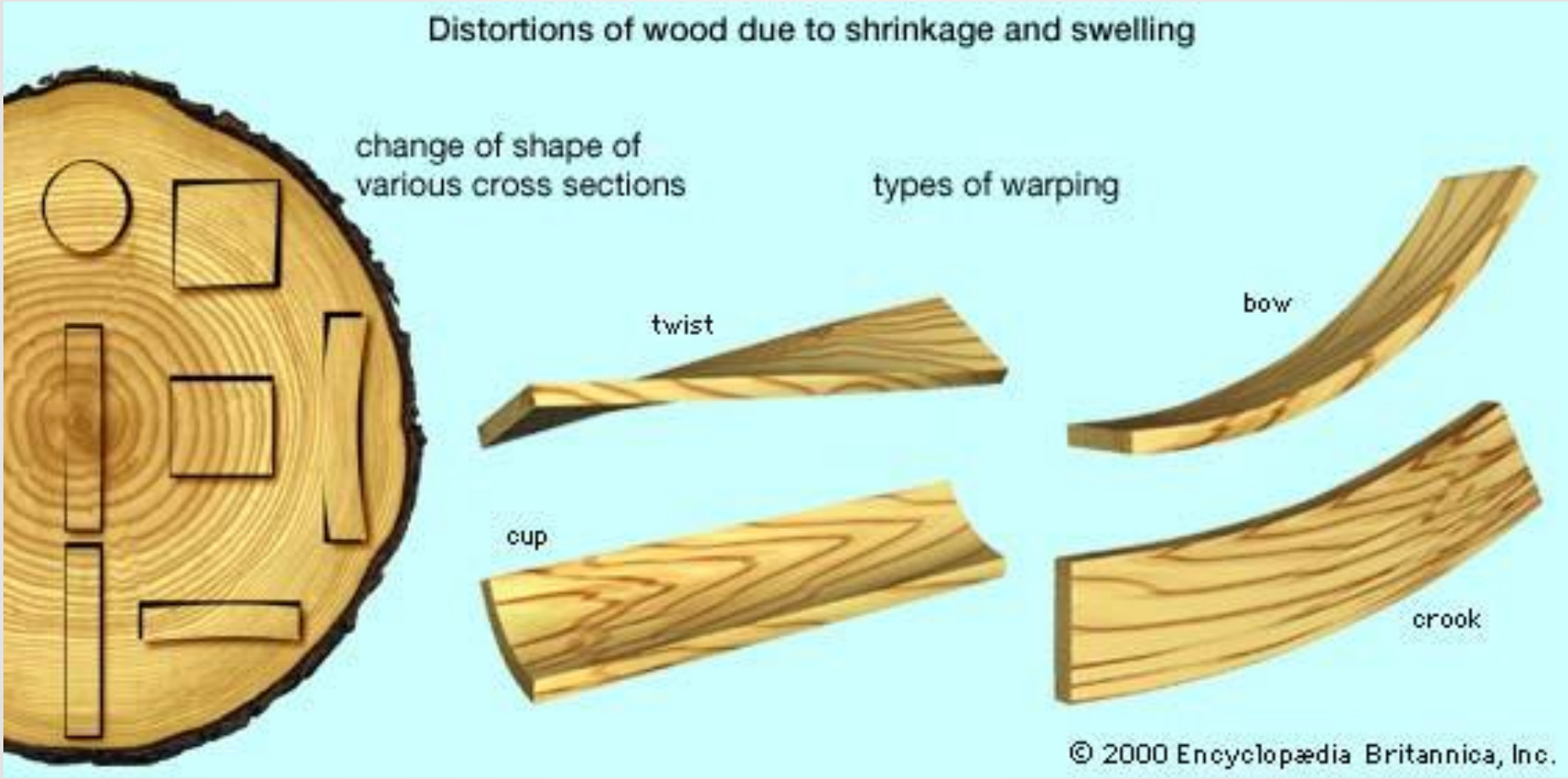


Warp	Current Instruction	Instruction State
Warp 1	42	Ready to write result
Warp 3	95	Computing
Warp 8	11	Computing
…		

Schedule at time k+1 ←

IT'S NOT THIS

AND IT'S NOT THIS



IT'S THIS!

CUDA

- ✓ Acronym for Common Unified Device Architecture
  - ✓ An API for General-Purpose GPU (GPGPU) Programming
- ✓ Version 9.1 released in December 2017
  - ✓ Runs on Linux and Windows
  - ✓ Comes with Software Developer's Kit – (Code examples)
- ✓ Supports programming in C/C++ and Fortran
  - ✓ Supports a large subset of C++
- ✓ Includes additional functionality
  - ✓ Debugging tools
  - ✓ Mathematical libraries
  - ✓ Image Processing libraries
  - ✓ Deep Learning libraries
- ✓ Its Free – See https://developer.nvidia.com/cuda-downloads

CUDA PROGRAMMING

To call the GPU kernel, you call it like a function, but between the function name and the parameters, you enclose the number of blocks and number of threads per block in triple angle brackets <<< and >>>.

```
//The function call
Kernel<<<BlocksPerGrid, ThreadsPerBlock>>>(params);
```

NEED TO MAKE SURE THINGS ARE DONE IN THE RIGHT ORDER

If there are two steps that share variables, we have to make sure that the first is done before the second starts.

```
__global__ void compute(int *d_in, int
   *d_out, int *d_sum) {

  d_out[threadIdx.x] = 0;

  for (i=0; i<SIZE/BLOCKSIZE; i++) {

     int val = d_in[i*BLOCKSIZE +
       threadIdx.x];

     d_out[threadIdx.x] +=
             compare(val, 6);

   }

    __threadfence();

   if (threadIdx.x == 0) {

      for (i=0; i<BLOCKSIZE-1; i++)

         *d_sum += d_out[i];

     }
}
```

This makes every thread in the block stop until all threads reach this point.

PERFORMANCE TIP #1

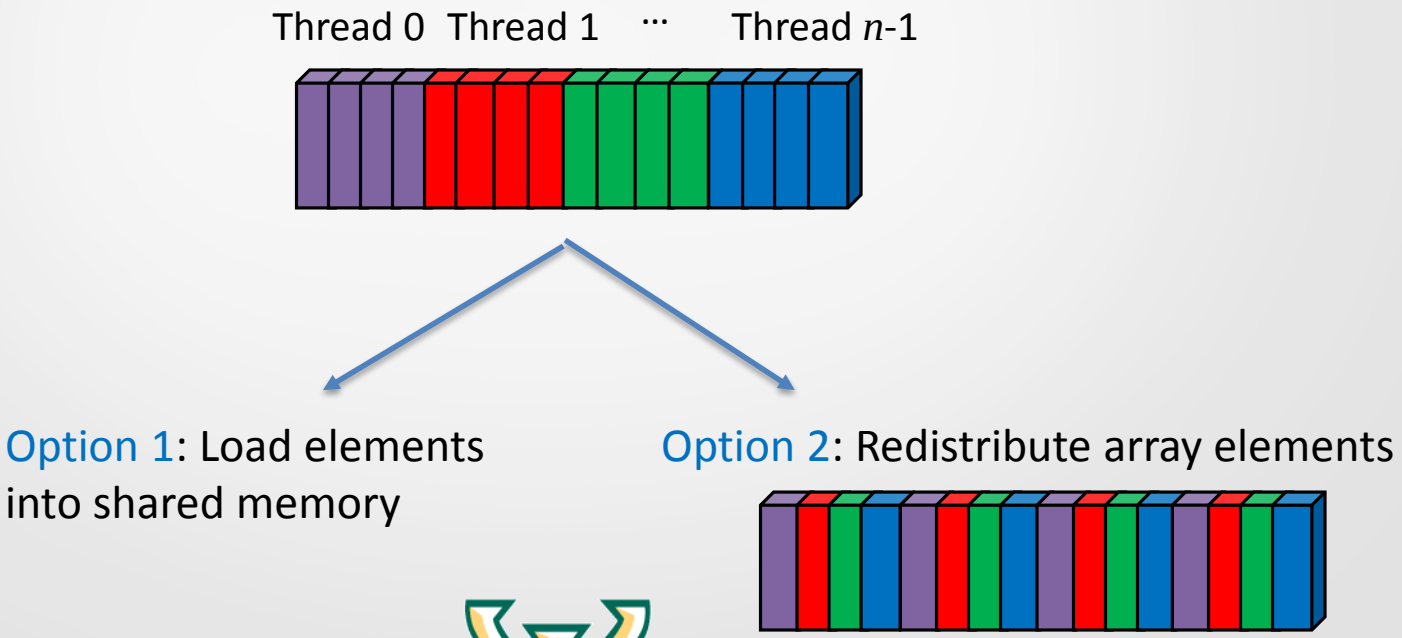
Premature optimization is the root of all evil – Donald Knuth

- "Maximize" Simultaneous Threads on each SM
- SM Resource Bottlenecks
  - Number of Registers
  - Number of Threads
  - Number of Blocks
  - Shared Memory
- Compile-Time Directives
  - __launch_bounds__(Max Threads per Block, Min # Blocks)
  - Advanced Idea

PERFORMANCE TIP #2

Coalesced Memory Accesses

- Global Memory Accesses are Expensive
  - Tens or hundreds of clock cycles
  - Can access up to 128 consecutive bytes in one memory operation
  - Combine reads/writes of threads to adjacent memory locations



CODING CONSIDERATIONS

- Single Codebase Approach
  - GPU Functionality Embedded in an Object-Oriented Framework
  - Interface should remain consistent
  - Multiple Parallel APIs could be supported
    - CUDA, OpenCL, OpenMP, OpenACC, etc.
  - #ifdef __NVCC__ … #endif
- Gradual Implementation
  - Simple loops with many iterations
  - Compute intensive functions

SUMMARY

- GPUs have a great deal of computing power
- Software tools like CUDA, OpenCL, and OpenACC make GPUs easier to use for non-graphics apps
- Performance depends on the GPU architecture
  - Data parallel applications
  - You have to think parallel
  - You often need different algorithms
  - At least, if you want to get good performance
- Good performance is possible with some effort

THANKS!!!

For some of the Slides and Figures Courtesy of:

Mary Hall – University of Utah
NVIDIA Corporation Documentation

QUESTIONS??

Picture courtesy of Google Images