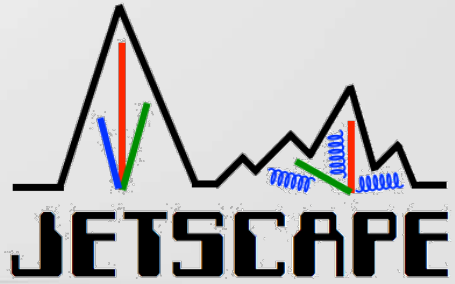
Computing on GPUs

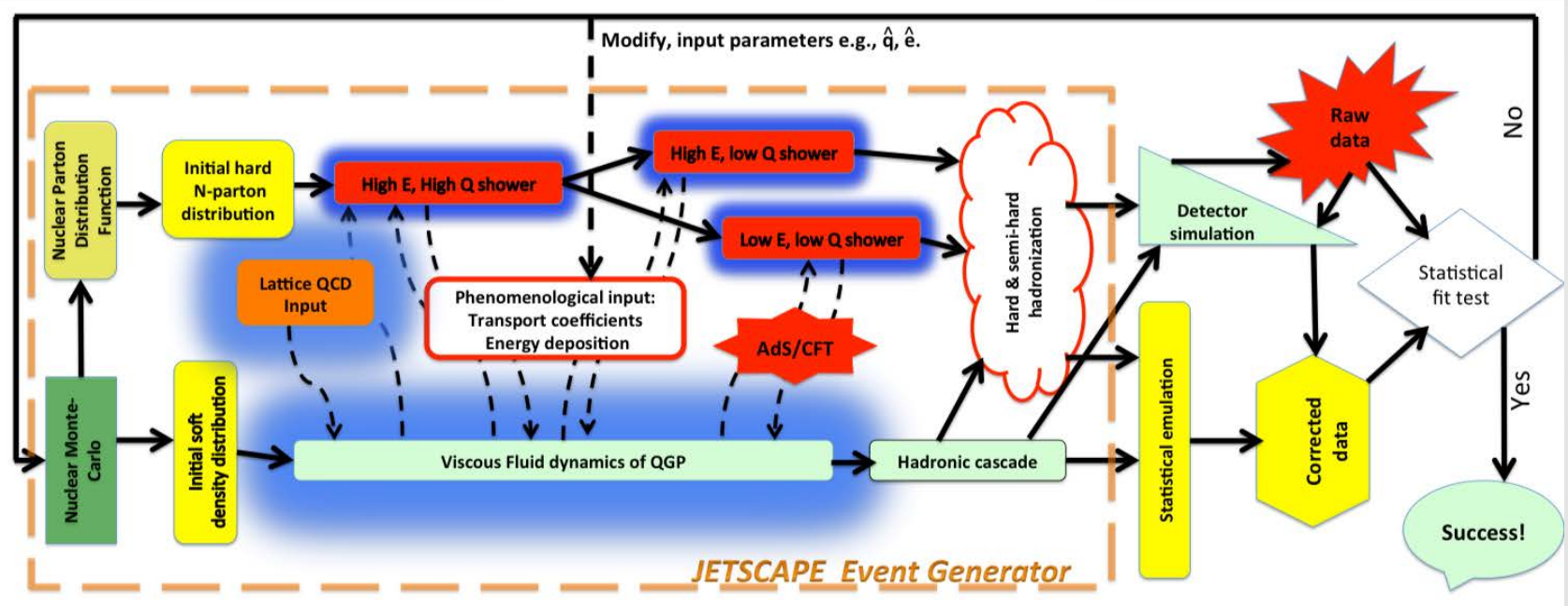
Presenter: Loren Schwiebert
Department of Computer Science
Wayne State University

On Behalf of the Jetscape Collaboration



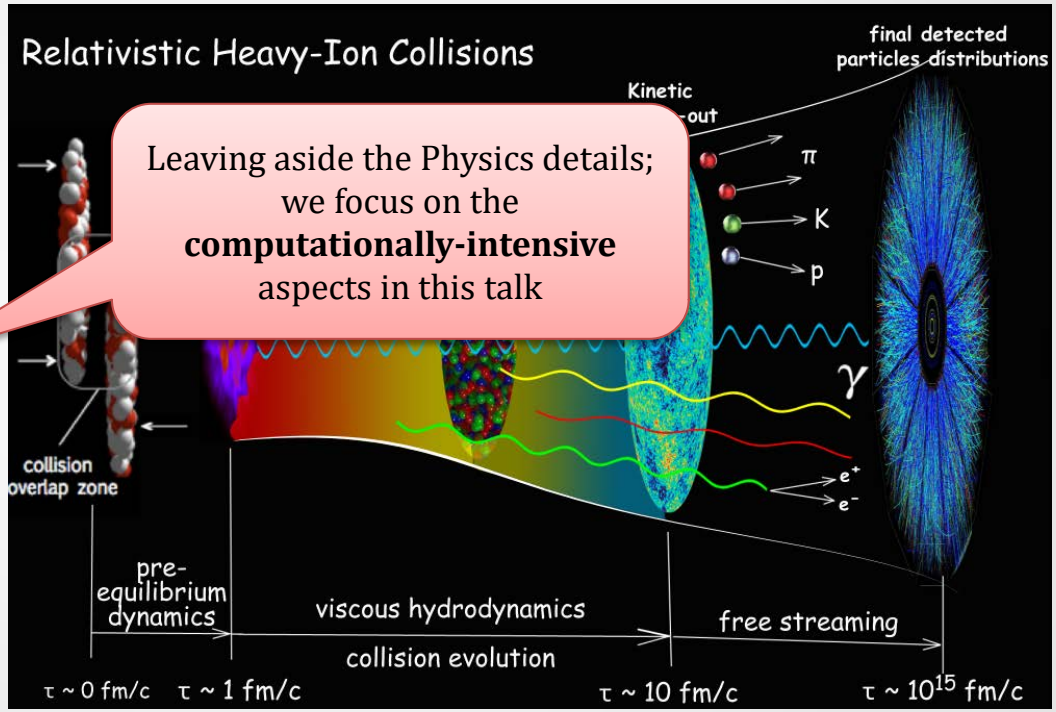
JETSCAPE Collaboration

- To simulate all aspects of the collision of heavy ions
  - From initial overlap to evaporation to conventional matter

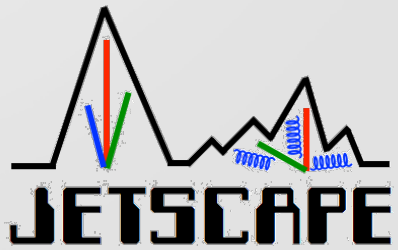


What Is JETSCAPE?

- Jet
  Energy Loss
  Tomography
- Statistical
- Computational
- Advanced
- Program
- Envelope

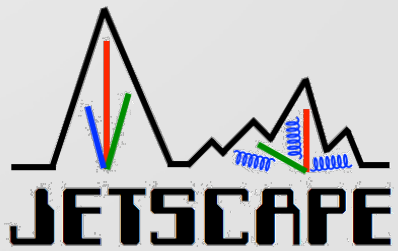


Leaving aside the Physics details; we focus on the **computationally-intensive** aspects in this talk



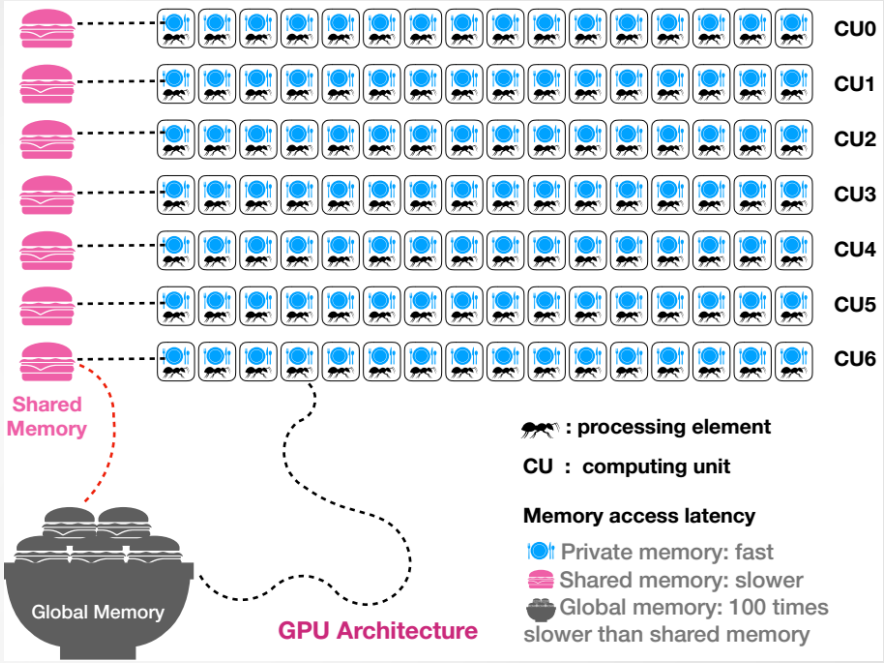
High Performance Computing (HPC)

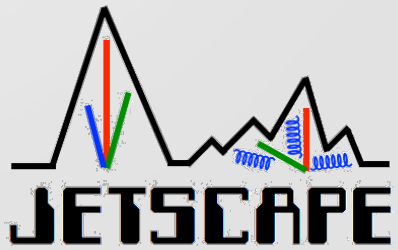
- Definition: the use of many computing resources over long periods of time to accomplish a computational task
- Goal: Accelerate completion of computational tasks
- Paradigms/Hardware: Graphics Processing Unit (GPU), Many Integrated Core (MIC), Many-Core CPUs, …
  - GPU: Composed of thousands of “simple” cores: compute-focused architecture. Cannot work independently and instead they work in lockstep.
  - MIC: Composed of many “smart” Intel Atom cores on “steroids”: capable of doing independent computation and Out-of-Order (OoO) execution, as of Knights Landings.
- Implementation: CUDA, MPI, OpenMP



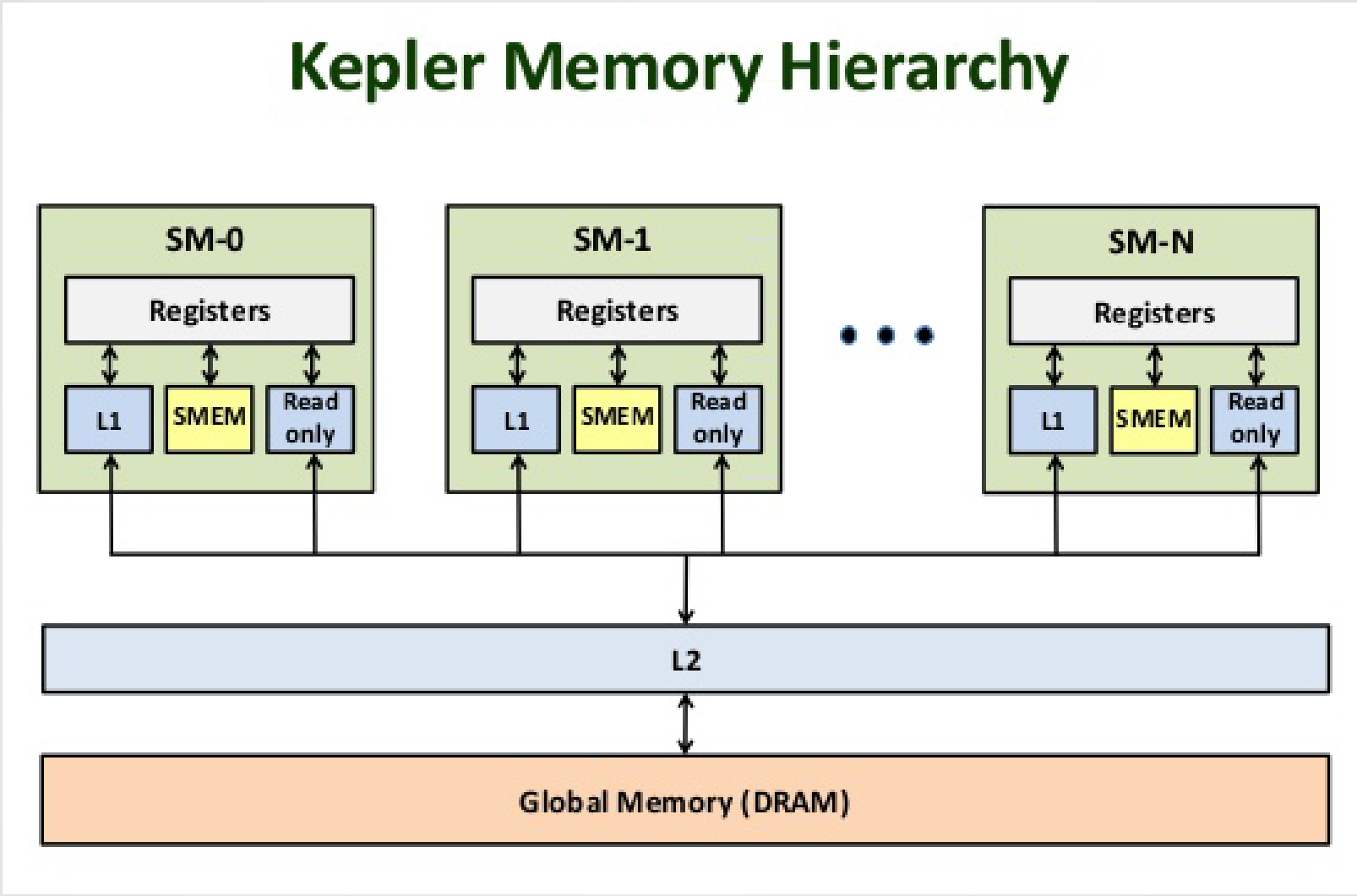
GPU Architecture

- Job is assigned to GPU cores via a higher level structure, i.e. a compute unit (CU) on AMD or a Streaming Multiprocessor (SM) on Nvidia.
- GPUs are ideal for embarrassingly parallel code where a single instruction is executed on a large data set.
  - Ideal case: large Matrix-Matrix multiplication





Memory Configuration



Commodity GPU – RTX 2080

- ✓ 2944 Processor Cores
- ✓ 1515 MHz/1710 MHz Clock Rate
- ✓ 8 GB GDDR6 Memory
- ✓ 256-bit Memory Bus Width
- ✓ 448 GB/s Memory Bandwidth
- ✓ Price: $700 in Sep. 2018

- ✓ 46 Streaming Multiprocessors

  Each has:
  - ✓ 64 FP32, 64 INT32 Cores
  - ✓ 4 Warp Schedulers
  - ✓ 32/64 KB of Shared Memory
  - ✓ 64/32 KB of L1 Cache
  - ✓ 64K 32-bit registers
  - ✓ 4 Dispatch Units

WAYNE STATE

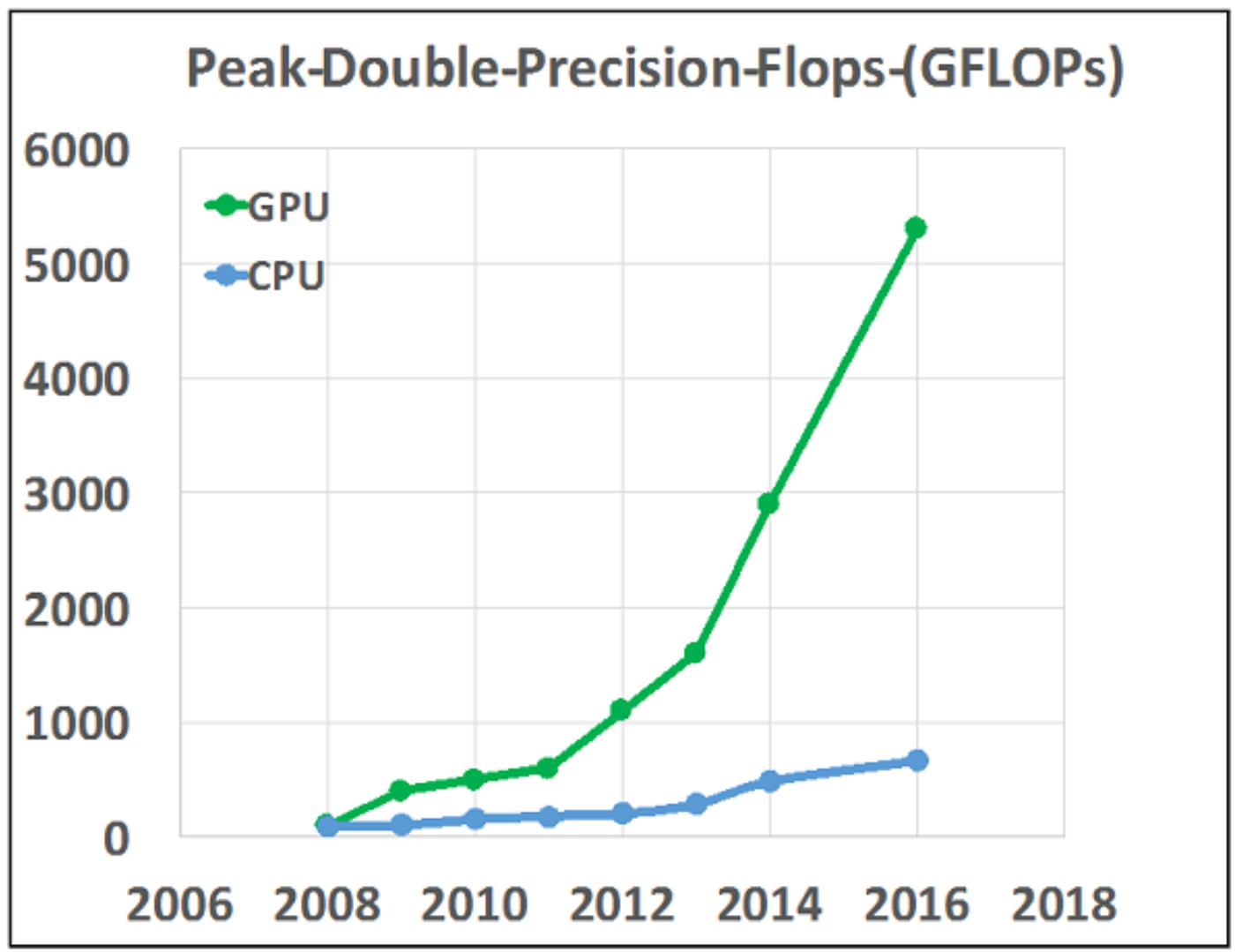
High-End Compute GPU – Kepler K80

- ✓ 5760 Processor Cores
- ✓ 745 MHz Clock Rate (+Boost)
- ✓ 24 GB DDR5
- ✓ 384-bit Memory Bus Width
- ✓ 288 GB/s Memory Bandwidth
- ✓ Price: $4000 in Sep. 2018

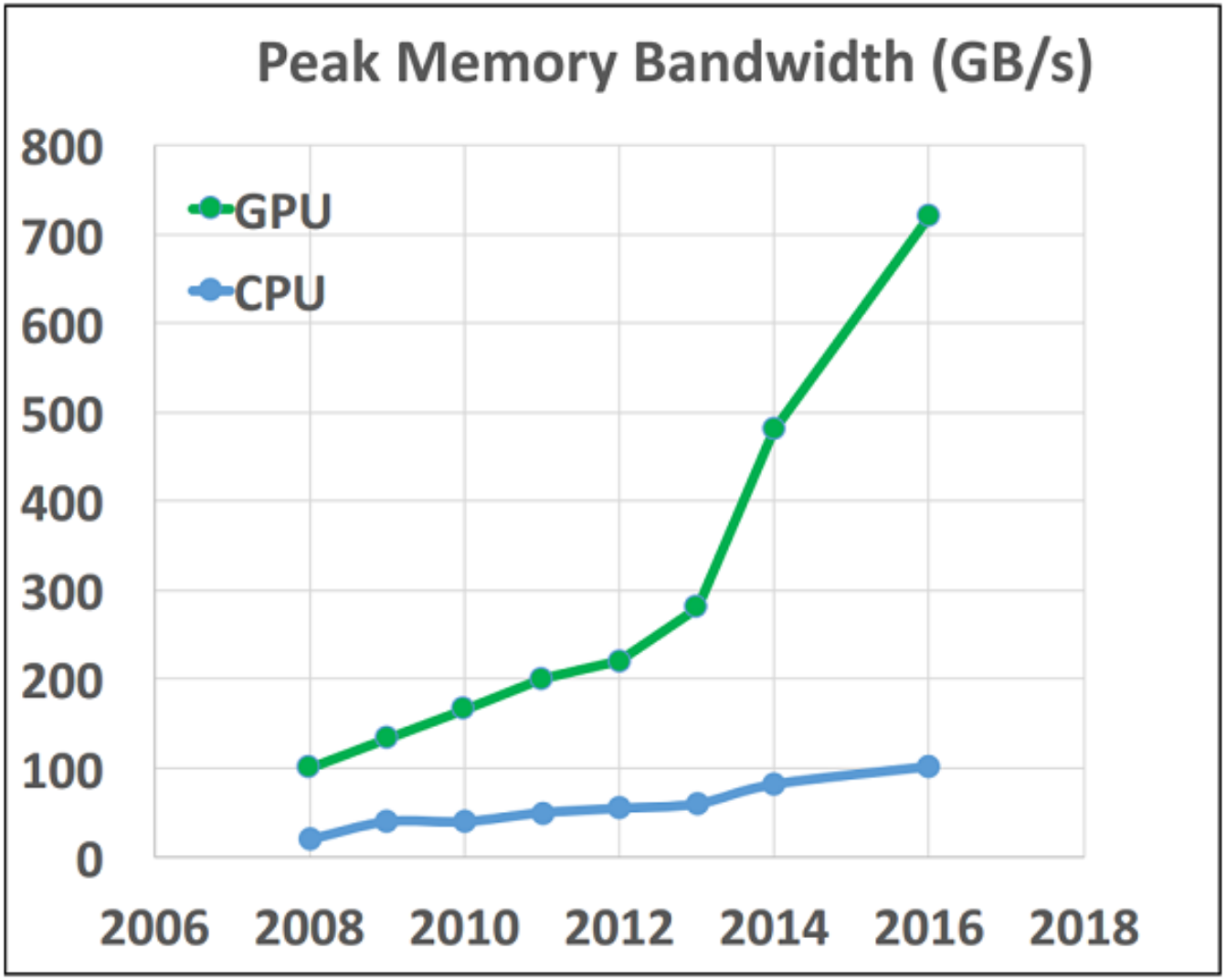
- ✓ 30 Streaming Multiprocessors
  Each has:
  - ✓ 192 Processor Cores
  - ✓ 32 Special Function Units
  - ✓ 48 KB of Shared Memory
  - ✓ 16 KB of L1 Cache
  - ✓ 64K 32-bit registers
  - ✓ 8 Instruction Issues/Cycle

WAYNE STATE

GPU VS. CPU FP Performance



GPU VS. CPU Memory Bandwidth



WAYNE STATE

GPU Programming

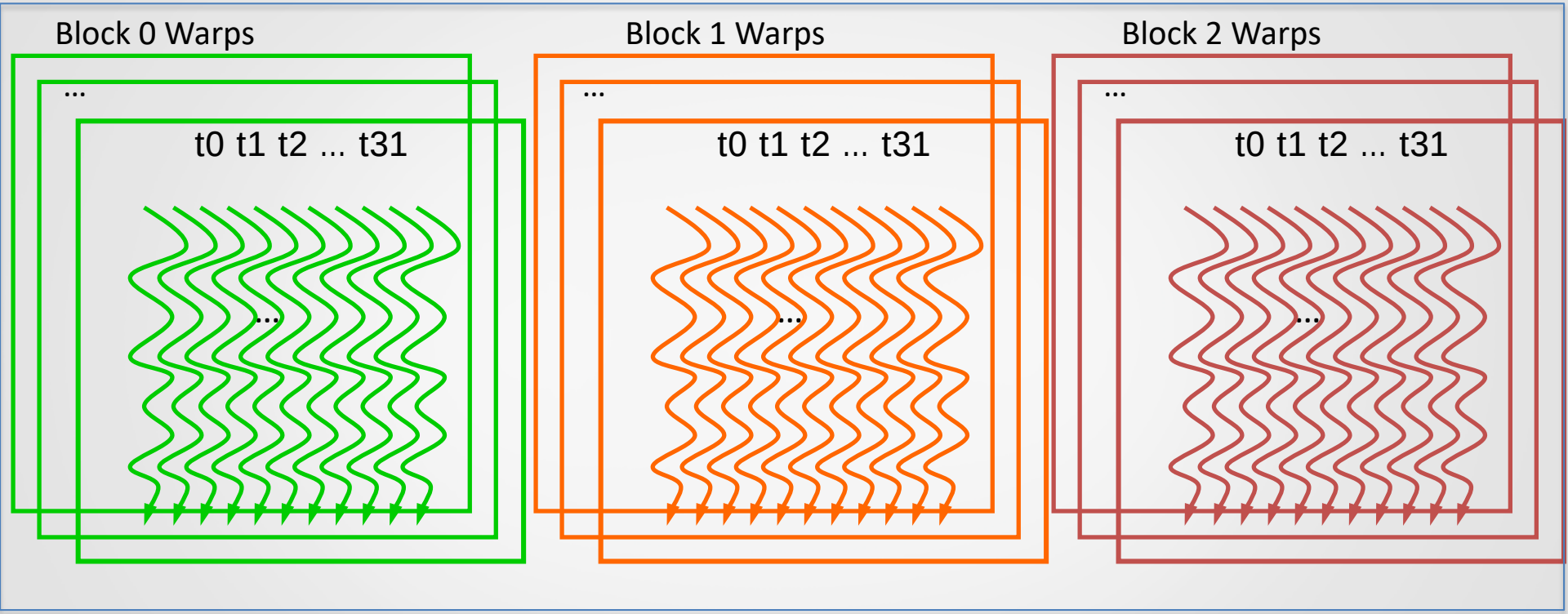
- CUDA: https://developer.nvidia.com/cuda-downloads
  - Programming API.
  - NVIDIA GPUs only. Not an open standard.
  - Offers many advanced features.
- OpenCL: https://www.khronos.org/opencl/
  - Open Standard.
  - Also supports other multicore and many-core architectures.
- OpenACC: https://www.openacc.org/
  - Open Standard.
  - Automated Parallelism.
  - 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

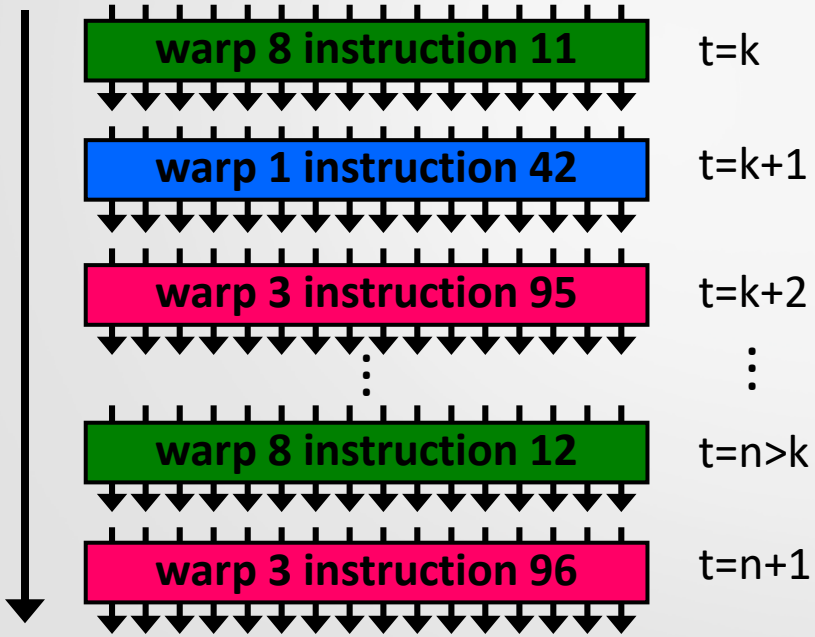
The Grid



Warp Scheduling Example

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

OK, but why warps??

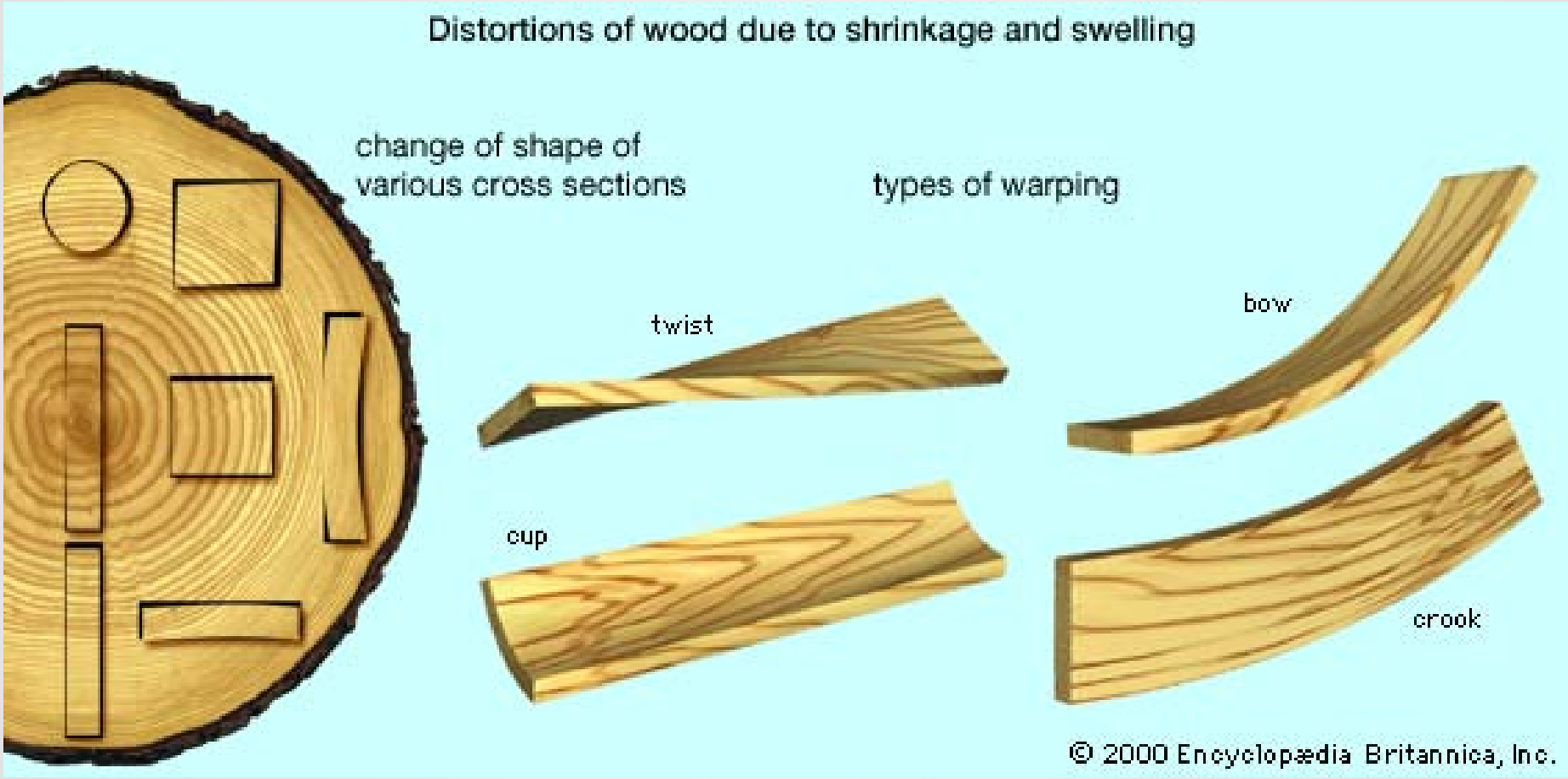


Warp	Current Instruction	Instruction State
Warp 1	42	Ready to write result
Warp 3	95	Computing
Warp 8	11	Computing
…		

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 10.0 released in September 2018
  - ✓ Runs on Linux, macOS, 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
  - ✓ Performance Analysis and Debugging tools (Nsight)
  - ✓ Mathematical libraries
  - ✓ Image Processing libraries
  - ✓ Deep Learning libraries
- ✓ Its Free

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);
```

Make Sure Things Are Done in the Right Order

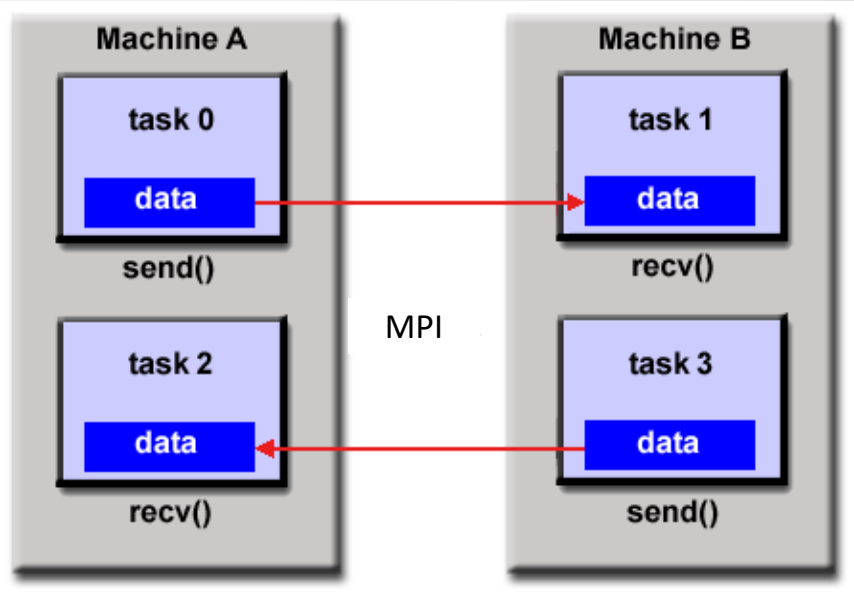
If there are two steps that share variables, we have to make sure that the first is done before the second starts.

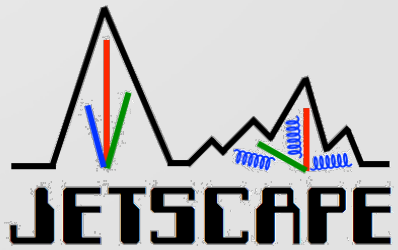
```
__global__ void compute(int *d_in, int
   *d_out, int *d_sum) {

  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);

   }

    __syncthreads();

   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.

Parallel Computation on CPUs

- Parallelizing independent events using Message Passing Interface (MPI)
  - MPI is a standardized and portable solution
  - Scalable across CPUs
- Our MPI implementation can run on:
  - Multicore CPU machines
  - Intel Xeon Phi co-processors (MIC)



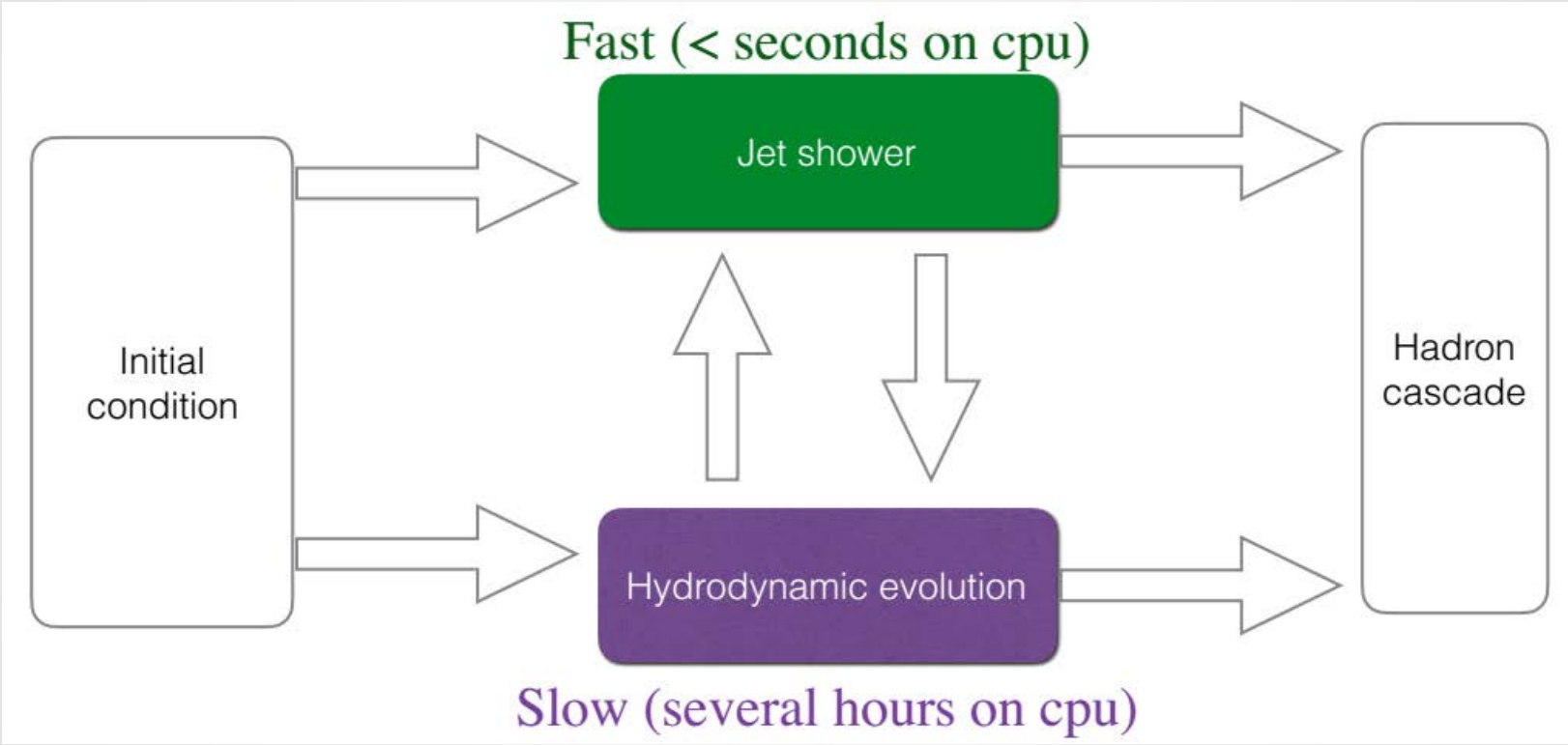


Parallel Implementation

- Focus on Compute-Intensive Routines
  - Data Parallel Functions
  - SIMD – Single Instruction, Multiple Data
- May Require Redesign of Data Structures
  - Better cache-aware data processing
  - Should benefit Sequential Code, too
- May Require Algorithmic Changes
  - Recomputation can be more efficient
  - Sometimes a totally new approach

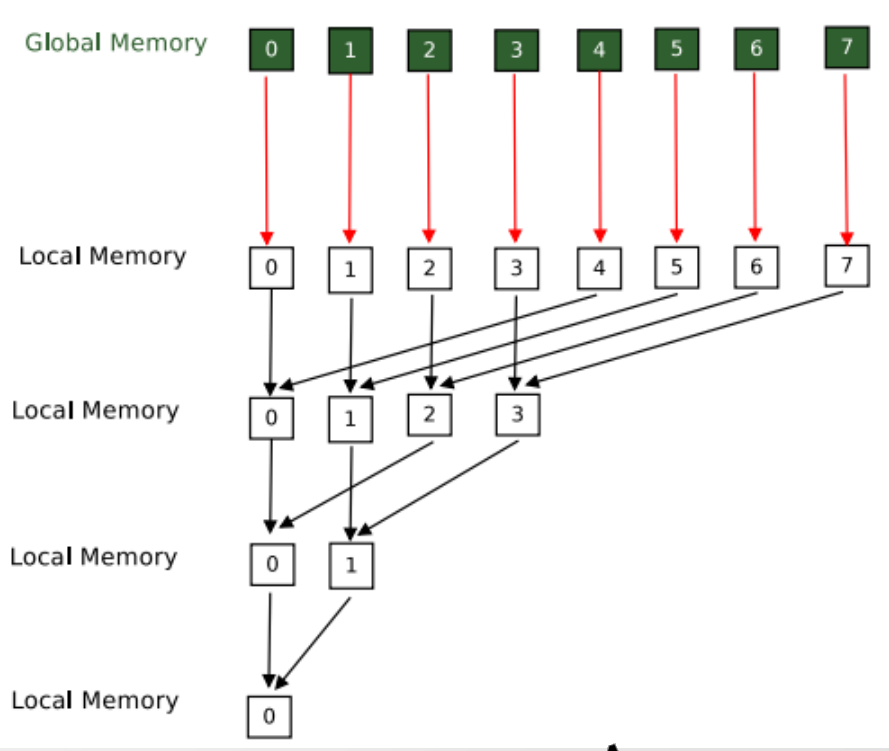
JETSCAPE: Opportunities To Apply HPC

- Hydrodynamics is the bottleneck in JETSCAPE



Hydrodynamics: First Application of GPU Parallelization

- Parallel reduction to get maximum, minimum, summation for a big array
- Stop hydro evolution when maximum temperature of QGP smaller than freeze out temperature
- Calculate spectra by summation over all the freeze out hyper-surface elements



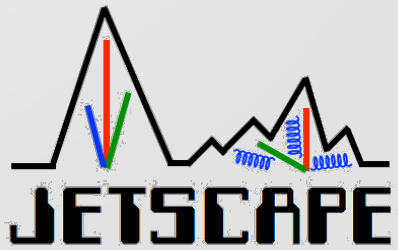
Hydrodynamics: Speedup Using GPU

Pb+Pb 2.76TeV/n, 20-25%

	CPU (i5-430M)	GPU (GT-240M)	GPU (K20)
Smooth spec. for π^+	7 minutes	30 seconds	0.5 seconds

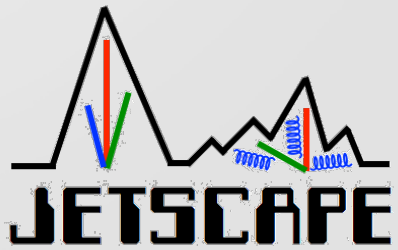
Performance of KT evolution for most central Pb+Pb collisions ideal hydrodynamics

CPU (E5-2650)	K20 ($5*5*5$ block)	K20 ($7*7*7$ block)
1 hour	3 minutes	50 seconds



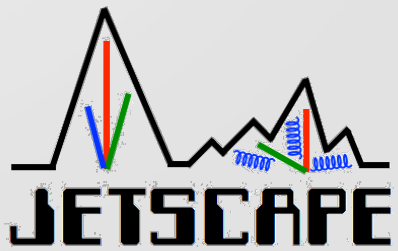
JETSCAPE: Status of Parallelization

- MUSIC: 3+1D viscous hydrodynamic code from McGill group
  - Parallelized on CPU using MPI
  - Already merged to master branch
- CLVisc: 3+1D viscous hydrodynamics code from CCNU-LBNL group
  - Parallelized on GPU using OpenCL
  - add_clvisc – merged to master branch soon
- SMASH: New hadronic transport code in C++
  - Parallelized on event-by-event basis on CPU using MPI (Future Work)



JETSCAPE: Parallel Performance

- CLVisc: 3+1D Mode
  - ~12 minutes on NVIDIA K20M, 2 minutes on NVIDIA on Tesla V100
  - 1 event at top LHC energy for Pb+Pb central collisions (spatial lattice size 201 x 201 x 121, ~1500 time steps)
- CLVISC: 2+1D Mode
  - Fluid evolution on GPU: 2 second/event using AMD-S9150 GPU (grid size 160 x 160)
- Smash: Hadronic Transport Code
  - Takes 30-80 seconds per event on a single core CPU
- Note: These numbers are rough approximations for a typical configuration



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 make it much easier to use GPUs for non-graphics applications
- 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