COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Lecture 18
Heterogeneous Architectures

Prof. Brendan Kochunas

11/06/2019

NERS 590-004

Outline

- Introduction
- GPU Architecture
- Programming GPUs
  - CUDA
  - Others
  - Kokkos

Motivation

- Hardware limits of serial performance
  - Moore's law no longer holds
- Performance gain is now often gained through parallelism
- GPUs offer massive parallelism
- Required to use GPUs on some modern HPC clusters!

Learning Objectives

- Basic knowledge of GPU architecture
- Knowledge of how to program for GPUs
  - CUDA, Kokkos
  - What is important for efficiency?
    - Memory Layout
    - Thread Scheduling
  - Portability

What is a GPU? Why do we care?

- Graphics Processing Unit
  - As the name suggests, historically used for processing graphics
  - Card in computer (PCI/PCIe)
  - Unified shader pipeline – different “shaders” used in graphics processing
    - All shaders require: texturing (data loads), math operations (FLOPs)
    - Unified pipeline allows for better GPU utilization
  - Data loads + math operations? Sounds a lot like normal computing?
- GPGPU – General Purpose GPU
  - GPUs unified shader pipeline allows for us to use them in general computing!

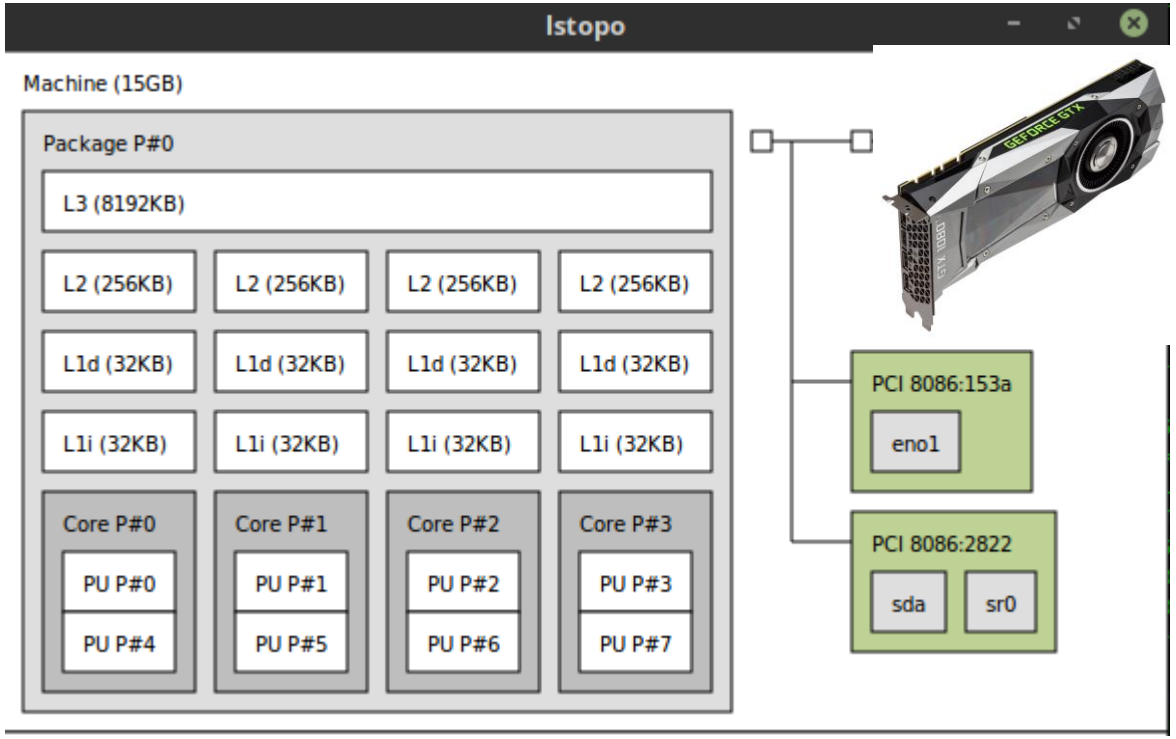
Why use a GPU?

- GPUs have a lot of small cores
  - Opportunity for **massive** parallelism
  - But… threads are executed in SIMD within warps! (Data parallelism)
- CPU is “low latency, low throughput”
  - Serial and moderate parallel calculations
- GPU is “high latency, high throughput” → streaming processor
  - i.e. good for *most* highly parallel applications (think simulations)

GPU Architecture

lstopo

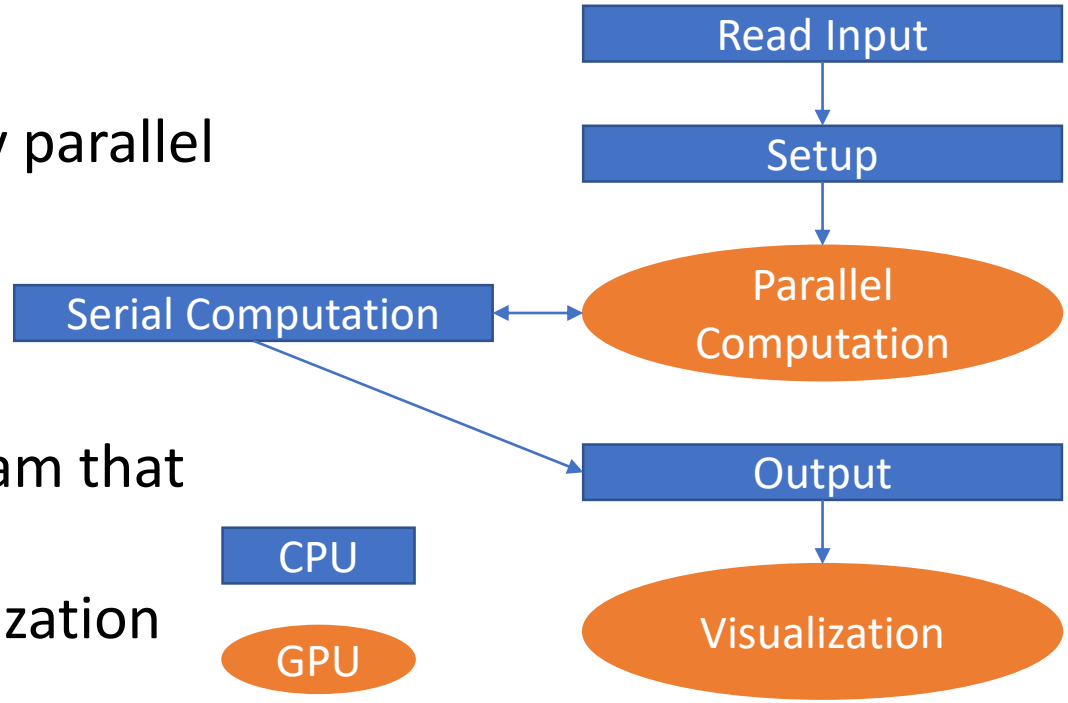
- CPU shown on the left (NUMA node)
  - See each core, cache-level
- GPU is shown as a PCI device (right)



https://www.google.com/search?q=image+of+gpu&source=lnms&tbm=isch&sa=X&ved=0ahUKEwixsvnYnMDeAhUE0oMKHSXXC7IQ_AUIEygB&biw=1368&bih=722#imgrc=J4rnsk1P30xTOM:

Heterogeneous System Architectures

- Serial code sections
  - Few applications are completely parallel
- Amdahl's law:
  - $S = \frac{1}{(1-p)+\frac{p}{s}}$
  - p is the proportion of the program that can be parallelized
  - s is the speedup of that parallelization



GPU Architecture

- A modern GPU has several Streaming Multiprocessors (SMs)
  - Pascal SM - 64 CUDA cores
  - Tesla GP100 – 56 SMs
  - Totally: 3584 CUDA cores
- Each SM has “warps” (2)
  - Each warp performs operations in SIMT fashion
    - Single Instruction Multiple Thread

https://devblogs.nvidia.com/inside-pascal/

CPU vs GPU

CPUs

- Designed for **Task Parallelism:**
  - Each thread executes a task
  - Tasks have different instructions
  - Relatively low number of threads
  - Within core/thread SIMD (4)
- Large cache to hide latency
- Only option for **serial** applications

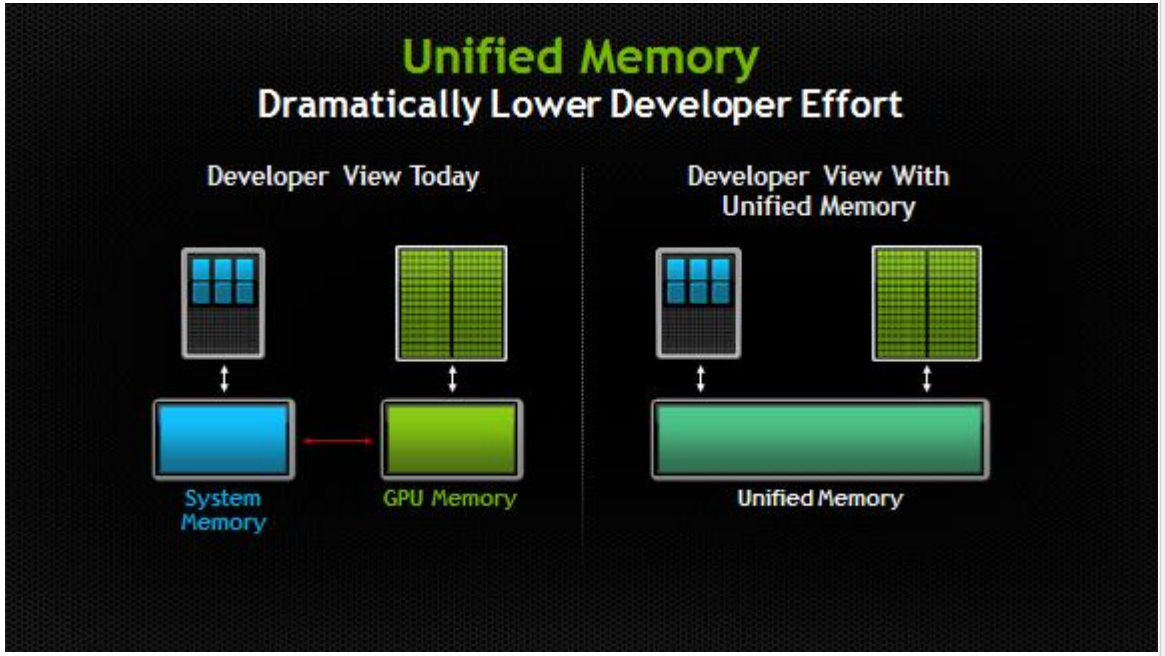
GPUs

- **Data Parallelism:**
  - SIMT model (SIMD)
  - Same instruction on different data
  - Large number of threads (10,000+)
- Stream Processing:
  - Large set of data (stream)
  - Run same series of operations on all the data
    - Series of operations is called a **kernel**

CUDA

CUDA

- CUDA is a platform and API for utilizing GPUs
  - C/C++ and Fortran
  - Only works for Nvidia GPUs!
  - Host = CPU
  - Device = GPU
- Memory management
  - Host vs device memory
  - Explicit data passing
  - Unified memory (CUDA 6+)



https://devblogs.nvidia.com/unified-memory-in-cuda-6/

Basics – Kernels

- Kernel
  - Essentially a function to be performed on the device (GPU)
  - Use **__global__** keyword to indicate device code

```
__global__ void mykernel(…);
```

  - Call with

```
<<<gridDim, blockDim>>>mykernel(...);
```

Basics – Threads, Blocks, Grids

```
<<<gridDim, blockDim>>>mykernel(...);
```

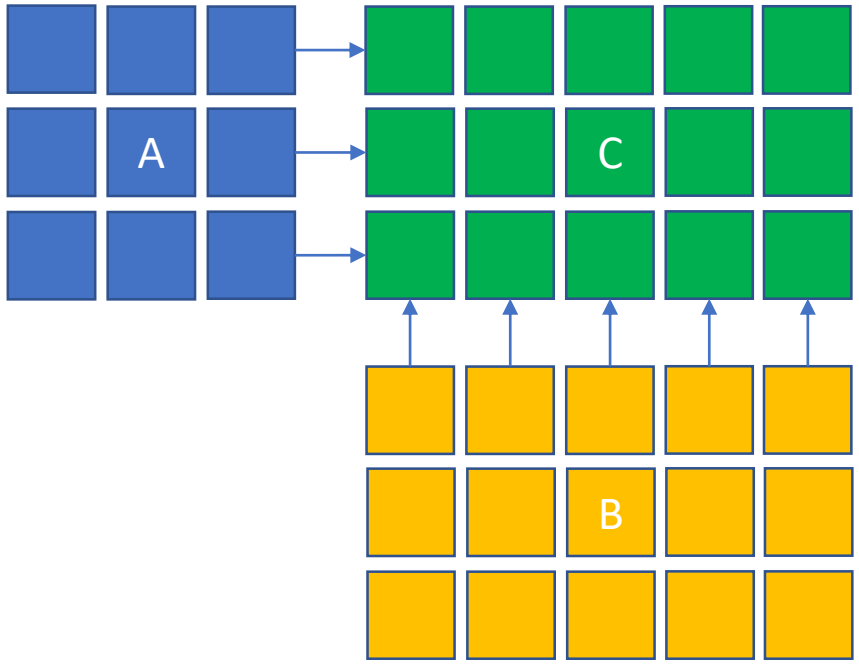
- Threads are organized into a grid of blocks
  - A grid is a collection of blocks – Executed on a single GPU
  - A block is a collection of threads – Executed on a single SM
  - A thread is the smallest unit – Executed on a single scalar core
- A thread represents a single scalar core performing work
  - Threads are grouped into “warps” (32 threads) – SIMT
- Indexing can be multidimensional

Basics – Indexing/Parallelism

- Within a kernel
  - the grid and block sizes are available: gridDim.x, blockDim.x
  - the block and thread indices are available: blockIdx.x, threadIdx.x
  - Calculate **global** index from these
  - Multidimensional indexing → replace x with y,z!
- What is the point of having blocks **and** threads?
  - Threads can communicate/synchronize!
  - Shared memory

Let's do an example…

- Matrix-matrix multiplication
  - C = AB
- Naïve approach
- Each element in C is given by
  - Dot product of corresponding row and column in A and B, respectively



Matrix-matrix multiplication (1)

- To make memory access patterns more obvious – flatten arrays

- Why are memory access patterns important?
  - Threads (cores) are grouped in warps
  - Warps operate in SIMT fashion (lock-step)
  - If 1 has a cache-miss **all** have a cache-miss (effectively)
- Linear memory accesses
  - A, C should be row-major
  - B should be column-major

Matrix-matrix Multiplication (2)

- On CPU – Naïve matmul
  - Loop through elements in C
    - Loop over corresponding row/column in A/B
- Matrices have been flattened
  - A, C = Row-Major
  - B = Column-Major

```
void matmul(unsigned n, unsigned m, unsigned p, double* A, double* B, double* C){
  unsigned A_idx = 0; unsigned B_idx = 0; unsigned C_idx = 0;
  for(unsigned row = 0; row < n; ++row){
    for(unsigned col = 0; col < p; ++col) {
      // Compute element of C at (row, col)
      A_idx = row * m; B_idx = col * m;

      // Initialize to zero
      C[C_idx] = 0.0f;
      // Loop over row/col of A/B (respectively)
      for(unsigned idx = 0; idx < m; ++idx){
        // Summation for the element
        C[C_idx] += A[A_idx] * B[B_idx];
        // Increment indices in A, B
        ++A_idx; ++B_idx;
      }
      // Increment the global C index
      ++C_idx;
    }
  }
}
```

Matrix-matrix Multiplication (3) - Threads

- On GPU
  - Parallelize w/ threads
  - Hardware limits number of threads
- Not general
- Not fast

```
__global__ void matmul(unsigned n, unsigned m, unsigned p, double* A,
double* B, double* C){
  // Each thread gets an INDEPENDENT element of C
  // From the index...we compute row and column
  unsigned C_idx = threadIdx.x;
  unsigned row = C_idx / p;
  unsigned col = C_idx - p * row;
  // Then compute A, B indices (starting)
  unsigned A_idx = m * row;
  unsigned B_idx = m * col;
  // Initialize to zero
  C[C_idx] = 0.0f;
  // Loop over row/col of A/B (respectively)
  for(unsigned idx = 0; idx < m; ++idx){
    // Summation for the element
    C[C_idx] += A[A_idx] * B[B_idx];
    // Increment indices in A, B
    ++A_idx; ++B_idx;
  }
}
```

Matrix-matrix Multiplication (4) - Blocks

- On GPU
  - Parallelize w/ blocks of threads
  - General
  - Not very fast though
- Memory accesses
  - Each thread has a single element of C
  - Each thread accesses contiguous chunks of A, B
  - Very small cache
    - Reloading A, B values!

```
__global__ void matmul(unsigned n, unsigned m, unsigned p, double* A,
double* B, double* C){
  // Each thread gets an INDEPENDENT element of C
  // From the index...we compute row and column
  unsigned C_idx = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned row = C_idx / p; unsigned col = C_idx - p * row;
  // Then compute A, B indices (starting)
  unsigned A_idx = m * row; unsigned B_idx = m * col;
  // Initialize to zero
  C[C_idx] = 0.0f;
  // Loop over row/col of A/B (respectively)
  for(unsigned idx = 0; idx < m; ++idx){
    // Summation for the element
    C[C_idx] += A[A_idx] * B[B_idx];
    // Increment indices in A, B
    ++A_idx; ++B_idx;
  }
}
```

Matrix-matrix Multiplication (5) - Tiling

- On CPU
  - Tiling increases cache locality → fewer misses
- On GPU
  - Tiling increases cache locality → fewer misses
  - A, B tiles (submatrices) can be loaded into shared memory (cache)
    - Fewer loads, fewer misses
    - Relatively small tiles → fits in cache!
  - Significantly better performance!

Summary of Cuda

- Control
  - Over where data is stored
  - Over where functions are executed
- **Kernels** are executed on GPU
  - Kernel functions marked by keyword **__global__**
- Parallelism is hierarchical
  - Grid of Blocks of Threads
  - A block of threads should work on contiguous memory
  - A single thread should move through memory with stride-1 access

Other Approaches for GPGPUs

OpenCL

- What is OpenCL?
  - Programming framework for heterogeneous systems
    - Includes CPUs, GPUs, DSPs, FPGAs, etc
  - Maintained by Khronos Group
- Portability
  - …at what cost?

https://en.wikipedia.org/wiki/OpenCL

OpenCL – Execution Model

- Abstract Execution Model
  - Work Item – Basic unit of work on compute device (GPU)
  - Kernel – Code that runs on a work item
  - Program – Collection of kernels and other functions
  - Context – Environment where work items execute
  - Command Queue – Queue used by host to submit work to the device(s)
- Work size
  - Global work size, work-group size
  - Effectively same as grids, blocks in CUDA

CUDA and OpenCL

Cuda	OpenCl
Streaming Multi Processor(SM)	Compute Unit
Streaming Processor(SP)	Processing Element
Global Memory	Global Memory
Shared Memory	Local Memory
Local Memory	Private Memory
Kernel	Kernel
Warp	Wavefront
Thread	Work-item
Block	Work-group

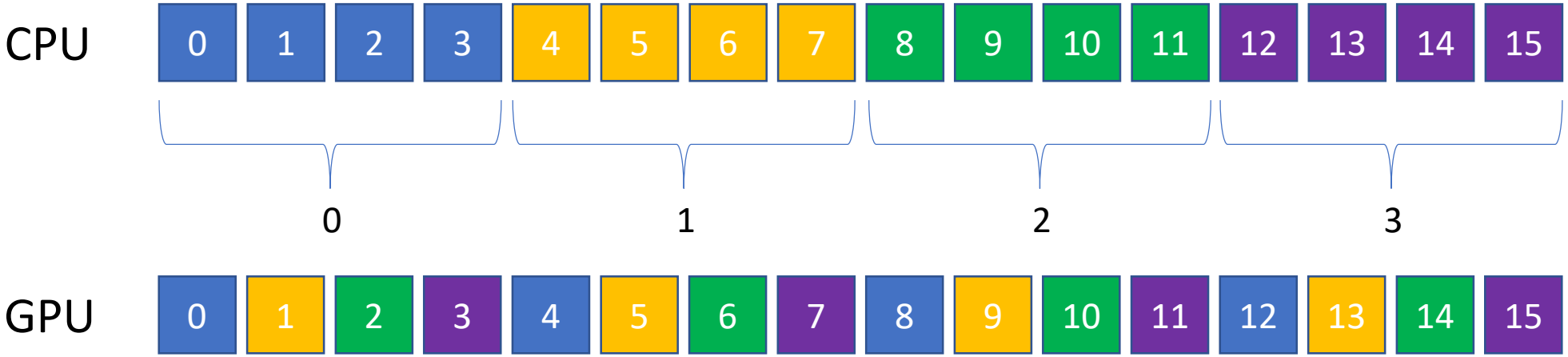
https://leonardoaraujosantos.gitbooks.io/opencl/chapter1.html

OpenMP(4+) and OpenACC

- OpenMP and OpenACC
  - Offer directive based approach for writing GPU code
- Typically simpler than writing CUDA/OpenCL codes
  - Typically worse performance
  - ...Compilers are making progress
- Portability
  - At what cost?

Portability?

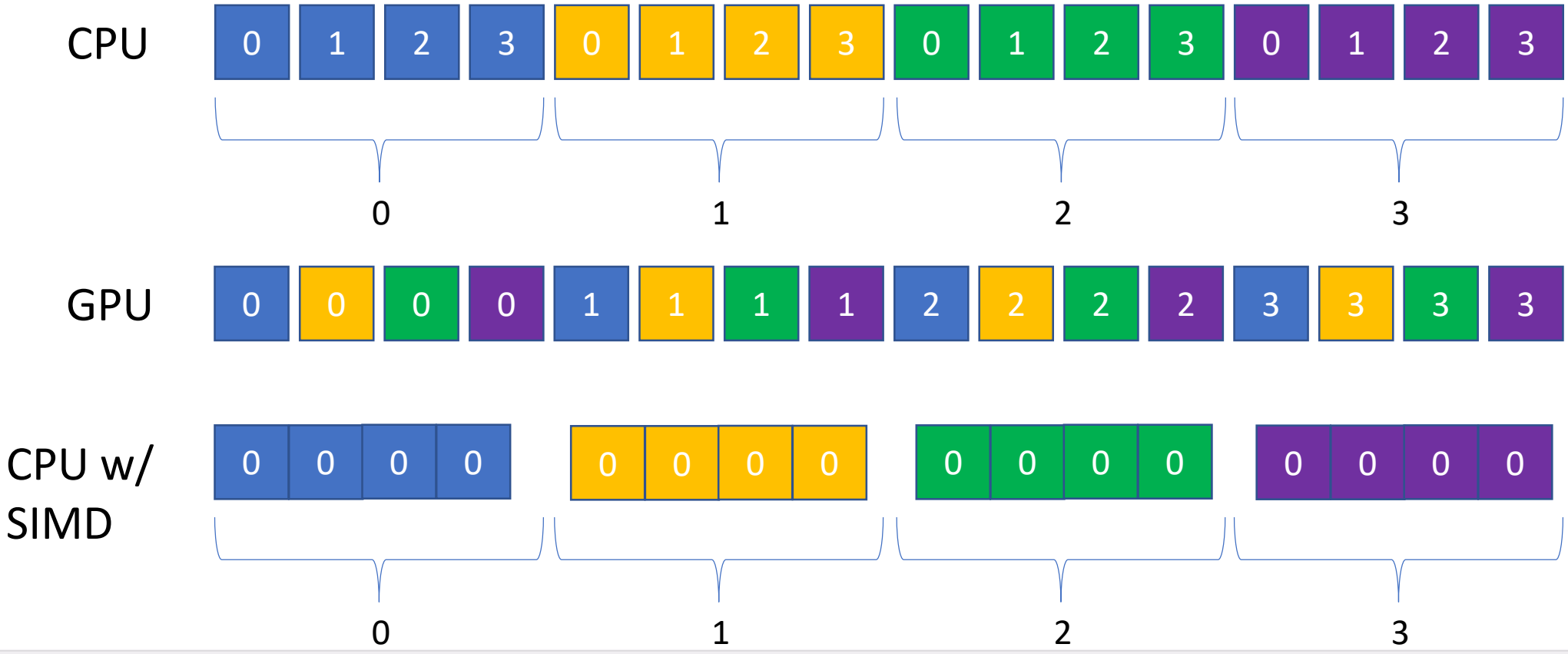
- Is portability a good thing?
- Code may be portable…but not efficient!



"You cannot make a solution independent of the actual platform or finite set of platforms. Reality is not a hack you're forced to deal with to solve your abstract, theoretical problem. Reality is the actual problem."

– Mike Acton, CppCon 2014 "Data-Oriented Design and C++"
https://www.youtube.com/watch?v=rX0ItVEVjHc

Another look



CPU and GPU Programming

- Threads on GPUs are (somewhat) analogous to Instruction Level Parallelism (ILP) on CPU
- Blocks on GPUs are (somewhat) similar to threads on CPUs
  - No communication though
- Portability may be possible, but we must be aware of architecture differences!

Kokkos

- Open Source – Developed by Sandia National Laboratories
  - https://github.com/kokkos
  - Detailed training slides at https://github.com/kokkos/kokkos-tutorials
- Kokkos offers performant portable code
- Aims for simplicity
  - Shouldn't be more difficult than OpenMP!
  - (You need some modern C++ knowledge though)
- It's a library
- A lot to cover. We will see examples on Friday!

Kokkos – Introduction (1)

Pattern Policy

```
for (element = 0; element < numElements; ++element) {
  total = 0;
  for (qp = 0; qp < numQPs; ++qp) {
    total += dot(left[element][qp], right[element][qp]);
  }
  elementValues[element] = total;
}
```

Body

https://github.com/kokkos/kokkos-tutorials/blob/master/Intro-Full/Slides/KokkosTutorial_ORNL18.pdf

- Pattern – structure for computations
- Policy – Range, schedule, thread teams, etc.
- Body – Work to be done (kernel)

Kokkos – Introduction (2)

- For performance, memory access patterns **must** be architecture dependent!
- The Kokkos library maps work to cores
  - Provide Kokkos with range and body
  - Kokkos will map indices to cores (may be on GPU!)
- If Kokkos is a library…how do we provide a body?
  - Functors! (function with data)

```
ParallelFunctor functor;
Kokkos::parallel_for(numberOfIterations, functor);
```

Kokkos – Functors

- Parallel functor must have access to the data it needs through the functor's data members
- Functors are simply structs
  - with void operator()(const size_t index) {…};
- Or, lambdas (C++11)
  - Lambdas auto-generate a functor
  - Tends to be less tedious (more productivity)

Functor vs Lambda Example

Functor

```
struct AtomForceFunctor {
  ForceType _atomForces;
  AtomDataType _atomData;
  void operator()(const size_t atomIndex) const {
    _atomForces[atomIndex] = calculateForce(_atomData);
  }
}
```

Lambda

```
    [=] (const size_t atomIndex) {
    atomForces[atomIndex] = calculateForce(data);
  }
```

Kokkos – Comparison

- Not more difficult than OpenMP, in concept

Serial

```
for (size_t i = 0; i < N; ++i) {
  /* loop body */
}
```

OpenMP

```
#pragma omp parallel for
for (size_t i = 0; i < N; ++i) {
  /* loop body */
}
```

Kokkos

```
parallel_for(N, [=] (const size_t i) {
  /* loop body */
});
```

Kokkos – Views

- Views allow us to store/communicate data between host and device
  - They are like pointers (you should copy, not reference)
  - Templated – on data type
    - And others!
    - Memory Space, Layout, Memory Traits, etc. (more details to come)
  - Multidimensional
    - Ease of use
    - Rectangular (not jagged)
  - No hidden allocations! (important for performance)

Kokkos – Execution Space

- Execution Space tells us where the code will run
  - Serial, Threads, OpenMP, Cuda, etc.
- Can control via default Execution Space – Compile Time
- Can specify Execution Space in the policy
  - A template parameter
- Need to mark non-CPU space lambda/functors with macro
  - #define KOKKOS_INLINE_FUNCTION inline __device__ __host__

Kokkos – Memory Space

- Memory Space tells us where the data is stored
- Template parameter on Views
  - HostSpace, CudaSpace, CudaUVMSpace, etc.
- Each Execution Space has a default memory space
- If in CudaSpace
  - Data is stored on GPU
  - Metadata is also stored on CPU
- Mirror/deep_copy

Kokkos – Layouts

- Multidimensional arrays in C/Fortran have different “Layouts”
  - Row-major is good in some places
  - Column-major is good in some places
- Kokkos gives a simple approach to multidimensional array layout:
  - Layout is a templated parameter on a View
  - LayoutLeft → Left-most index is stride 1
  - LayoutRight → Right-most index is stride 1
  - More, and you can create your own

Kokkos – Scheduling

- On the CPU we wanted each thread to work over a contiguous chunk of memory
- On the GPU we wanted each thread ***block*** to work over a contiguous chunk of memory (thread indexing strided)
- Kokkos maps indices to cores in contiguous chunks on CPU and strided for GPU
- Using this knowledge in addition to Layouts
  - We can achieve performant portable code
  - …if we choose the right layout for the architecture

Kokkos – Subviews and Multiloop Parallelism

- Subview
  - Like a “slice” of a View
  - Uses colon notation from Fortran, Matlab, Python, etc.
  - No allocation occurs in construction! – but not free!
    - Avoid usage if not accessing a lot
- MDRangePolicy
  - MultiDimensional Range Policy
  - Parallelism over multiple loop levels
  - Similar to collapse in OpenMP

Kokkos – Hierarchical Parallelism

- Hierarchical Parallelism allows for nested parallelism
  - League of Teams of Threads vs Grid of Blocks of Threads
  - Thread teams – Threads that work concurrently and can synchronize
    - Think block in Cuda
    - Functor/lambda operations get a team member
      - teamMember.league_rank();
      - teamMember.team_rank();
      - blockIdx, threadIdx in Cuda
  - Vector level parallelism
    - Vectorizable loops on CPU
    - (Sub-)warp level parallelism on GPUs

Matrix-matrix multiplication

- Parallel implementation of naïve matrix-matrix multiplication

```
void matmul(unsigned n, unsigned m, unsigned p, mat_r A, mat_l B, mat_r C)
{
  typedef Kokkos::MDRangePolicy< Kokkos::Rank<2> > mdrange_policy;
  Kokkos::parallel_for("matmul", mdrange_policy({0, 0} , {n, p}),
    KOKKOS_LAMBDA (unsigned row, unsigned col) {
      for(unsigned idx = 0; idx < m; ++idx){
        C(row, col) += ( A(row, idx) * B(idx, col) );
      }
    }
  );
}
```

  - mat_l, mat_r are aliases for Kokkos::View<double**>
    - LayoutLeft, LayoutRight

Kokkos – Atomics/MemoryTraits

- Atomics – Atomic Operations
  - Solution to multiple threads trying to write to same location
  - Locks are not scalable to 10,000+ threads
  - Data replication is not scalable to 10,000+ threads
  - Example: Kokkos::atomic_add(…);
- Memory Traits
  - Template parameter on Views
  - Atomic, Read, Write, ReadWrite, ReadOnce, Contiguous, RandomAccess
  - Tells compiler/Kokkos what you will do with the memory

Kokkos – Scratch Space

- Scratch space offers (roughly) a programmer managed cache
- GPUs have small dedicated scratch space
  - Kokkos lets you use these much more easily than Cuda
- CPUs don't have special hardware for this
  - But, memory access patterns are cache-aware
- Accessing data in this scratch space is **much** faster than in main memory
  - Level 0 fastest
  - Level 1 fast

Kokkos – Summary

- Offers Portability
  - **without** compromising performance
- Simple things are just as simple as in OpenMP
- Advanced tuning is simpler than native implementations
  - For GPUs
  - Not so much for CPUs

Useful resources

- https://devblogs.nvidia.com
  - https://devblogs.nvidia.com/even-easier-introduction-cuda/
  - https://devblogs.nvidia.com/unified-memory-cuda-beginners/
  - https://devblogs.nvidia.com/inside-pascal/
- http://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/CUDA_C_Programming_Guide.pdf
- https://github.com/kokkos/kokkos-tutorials