Lecture 19 – Heterogeneous Architectures

Prof. Brendan Kochunas

NERS/ENGR 570 - Methods and Practice of Scientific Computing (F22)

COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Outline

- Coarse Grained vs. Fine Grained Parallelism
- SIMD
- GPU Architectures
- Overview of GPU Programming Models
- Hands-on Stuff

Learning Objectives: By the end of Today's Lecture you should be able to

Coarse vs Fine Grained Parallelism

And other ingredients for parallel algorithms

Parallel Algorithm Ingredients

- What is the ***programming model***? (distributed, shared, both)
  - If distributed, what is the communication model?
- What should the ***granularity*** of the parallelism be?
- How are you going to ***decompose*** the problem in parallel?
- How are you going ***partition*** the problem to obtain a balanced decomposition?
- Can all this be done once for a single simulation?
- What synchronizations are required?

Coarse Grained vs. Fine Grained

Coarse Grained

- Divide work into large tasks
  - Example: executing several functions
- Coarse grained parallelism usually has better strong scaling than fine-grained parallelism.
  - Although smaller limits to the maximum parallelism
- More susceptible to load imbalance.

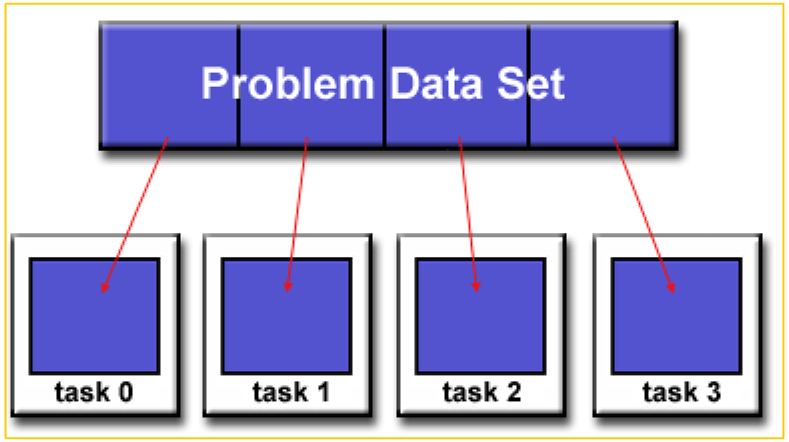
Fine Grained

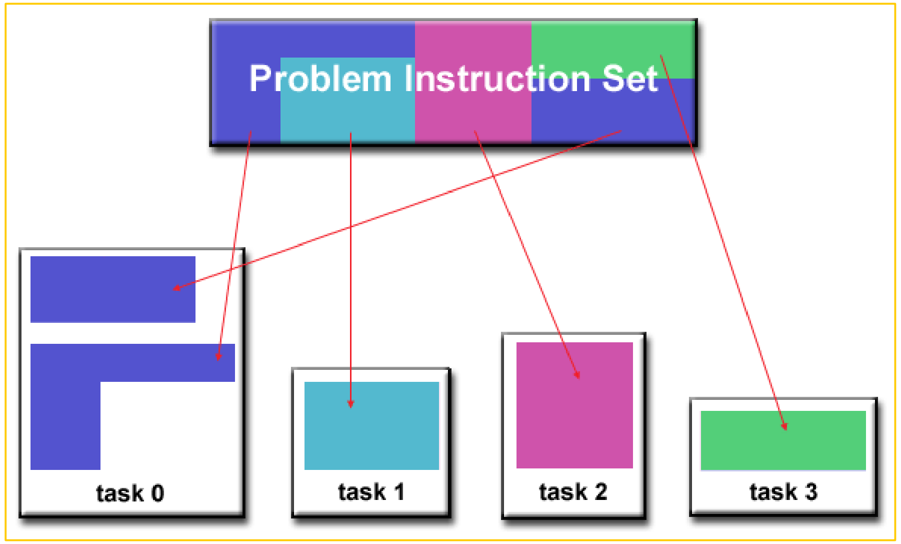
- Divide work into many small tasks
  - Example: iterations of a loop
- Usually has good load balance
- Difficult to hide overhead from parallelism
- Works well for things like SIMD & vector computing

Algorithm & Hardware will ultimately determine which is better. However, coarse-grained will usually be better

Decomposition

- What is being divided into parallel work?
- Most typical is domain decomposition
  - Divide up part of your equation "phase space"
    - Phase space = dependent variables of unknown (e.g. Cartesian space)
  - Slightly different is data decomposition
    - e.g. decompose a matrix in parallel
      - Matrix is usually a discretization of the phase space(s)
- Also have functional decomposition
  - Decompose by computation or operation
    - e.g. fluid on one process, solid on another for convective/conductive heat transfer

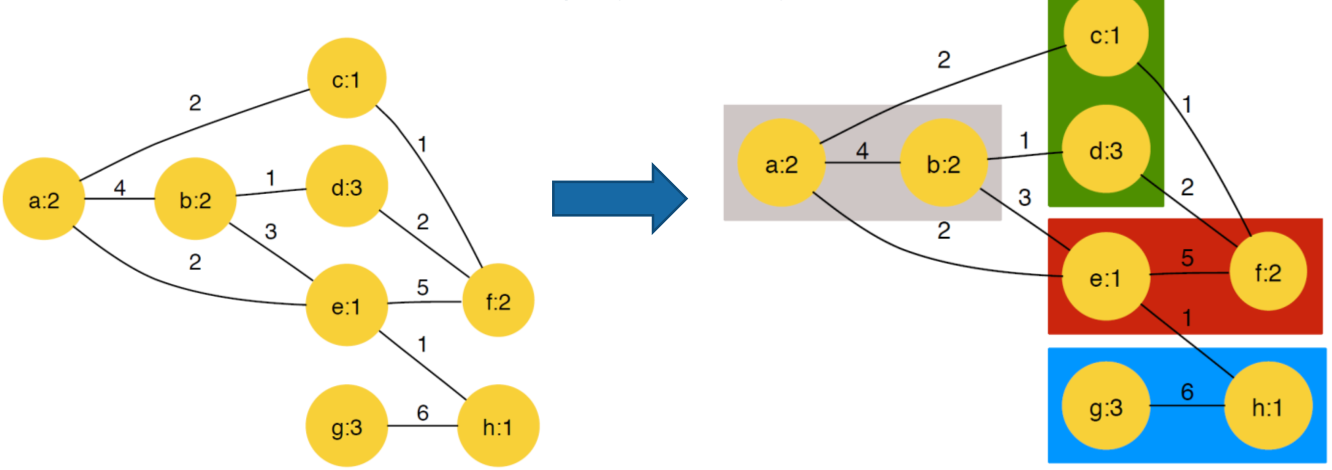




Figures from: https://computing.llnl.gov/tutorials/parallel_comp/

Partitioning

- How do you decompose the problem in parallel?
  - Example: Matrix partitioning
- In general this is a much harder problem.
  - Especially for the general case.
    - Involves a lot of graph theory



Figures from: R. Vuduc, “Graph Partitioning,” Lecture in CSE/CS 8803, Georgia Institute of Technology, April 2008

- Libraries exist to do this for us: METIS & ParMETIS

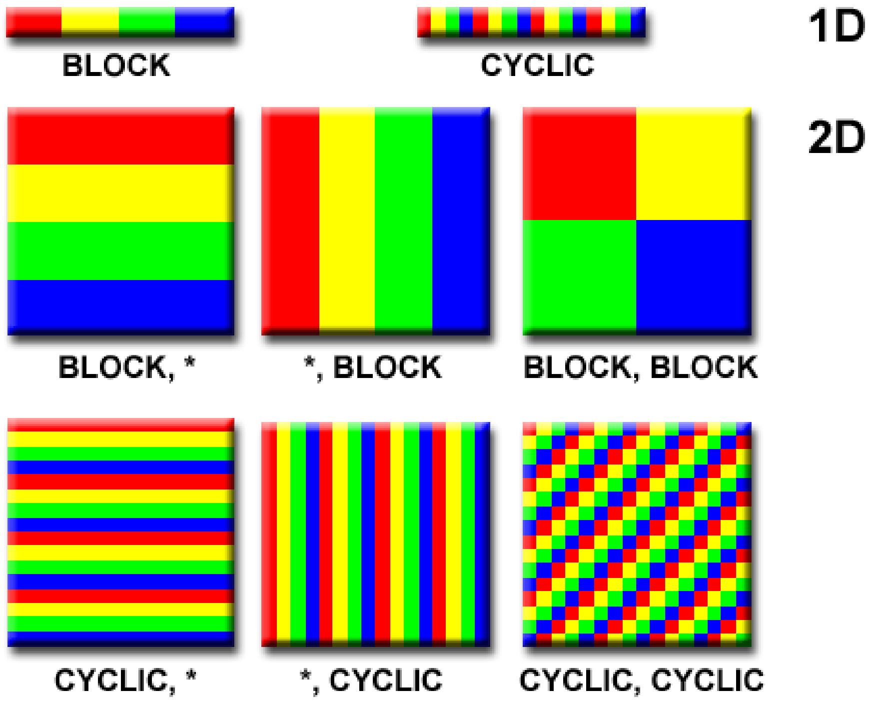


Figure from: https://computing.llnl.gov/tutorials/parallel_comp/

Dynamic vs. Static

Static

- Determine decomposition and partitioning once up-front prior to execution.
- Execute without changing number of processors or decomposition or partitioning
  - Fork/Join is not considered dynamic if the number of threads always the same
- More likely you will encounter this case

Dynamic

- Necessary to achieve better performance if computation load changes during run time.
- Change number of processors during run time.
- Change partitioning during run time.

Synchronization

- Generally, best to avoid as much as possible
  - In practice, never completely avoidable.
- In shared memory parallelism this includes the fork and join operations.
- Synchronization usually occurs whenever you encounter an integral.
  - More generally it occurs with "reduction" operations.
  - In a reduction operation you reduce parallel data to a single process
    - E.g. computing a sum, finding a max, computing a product, logical operators
- In distributed memory parallelism (more specifically MPI), it is any collective operation (not just reduce)
- Critically important to be aware of collective operations

Illustrations of collective operations

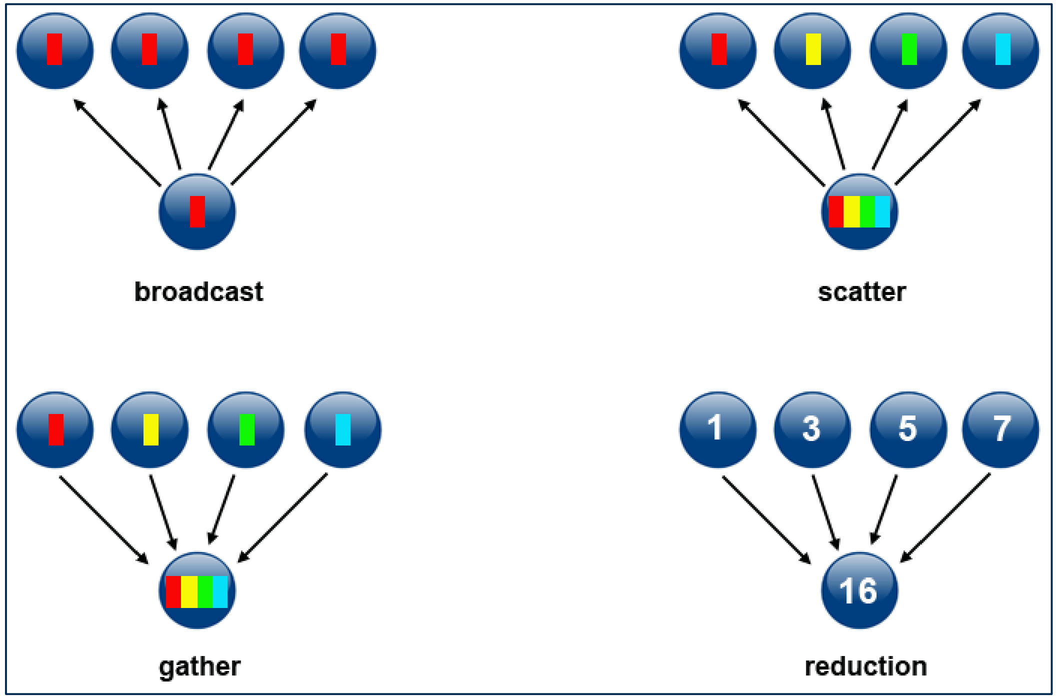


Figure from: https://computing.llnl.gov/tutorials/parallel_comp/

SIMD Parallelism

Single Instruction Multiple Data

Single Instruction Multiple Data

More illustrative examples

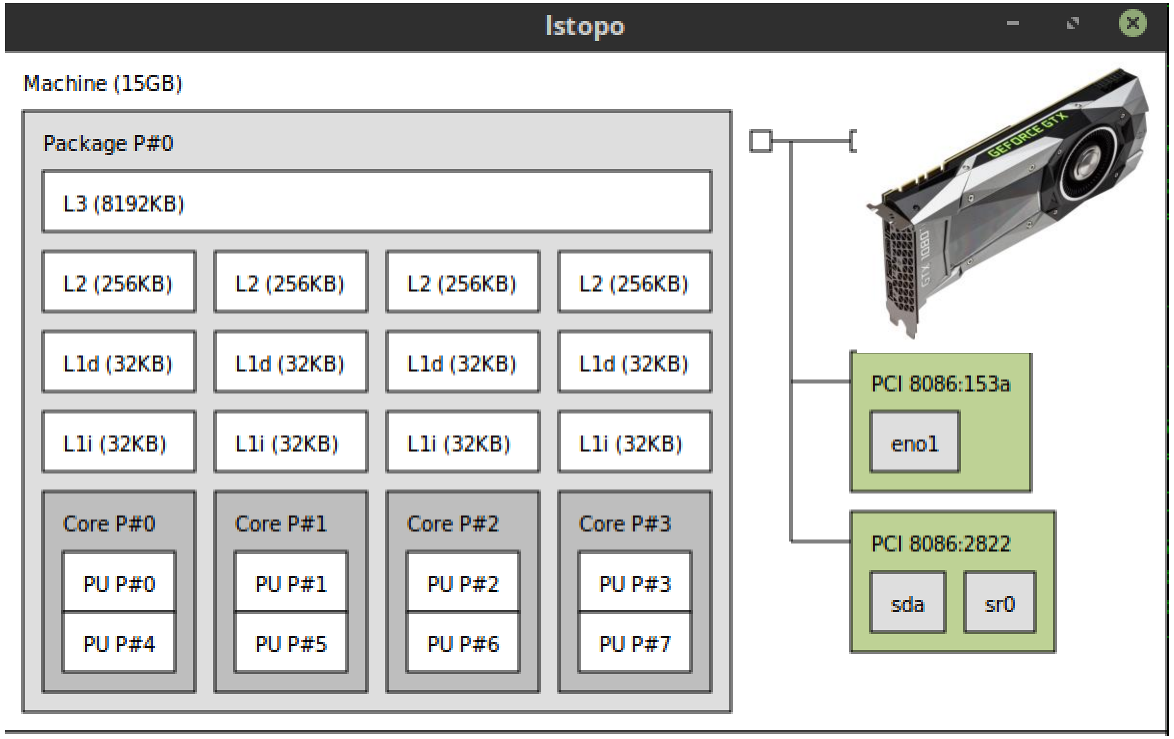
GPU Architecture

Motivation for GPUs

Why use a GPU?

Heterogenous Architecture

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



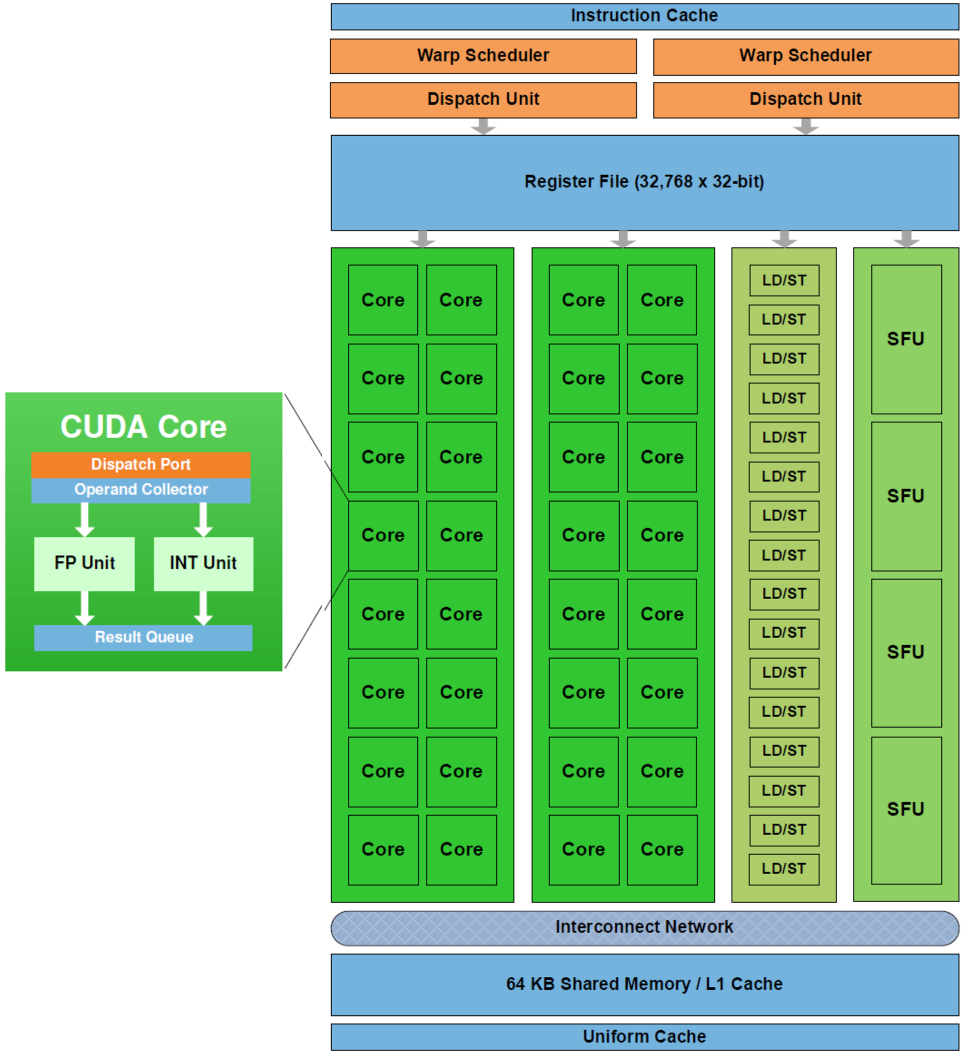
https://www.google.com/search?q=image+of+gpu&source=lnms&tbm=isch&sa=X&ved=0ahUKEwixsvnYnMDeAhUE0oMKHSXXC7IQ_AUIEygB&biw=1368&bih=722#imgrc=J4rnsk1P30xTOM:

GPU Architecture

- A modern GPU has several Streaming Multiprocessors (SMs)
  - Pascal SM - 64 CUDA cores
  - Tesla GP100 – 56 SMs
  - Total: 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/

A Closer Look



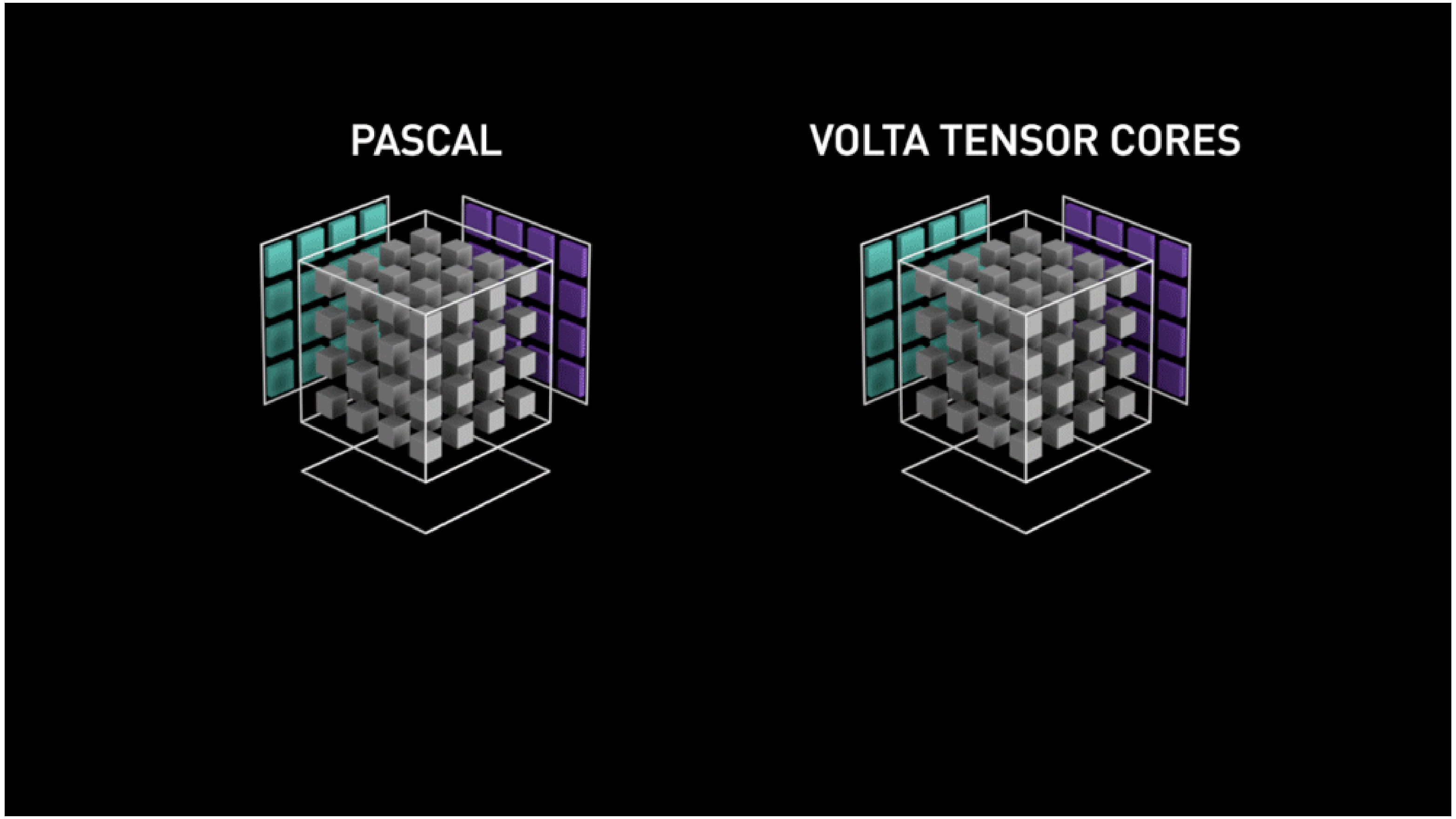
Fermi Streaming Multiprocessor (SM)

Volta Streaming Multiprocessor (SM)

Evolution of Hardware

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 (+64 mixed INT/FP32 cores)
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

Tensor Cores



Evolution of Performance

GPU PERFORMANCE COMPARISON

	P100	V100	Ratio
DL Training	10 TFLOPS	120 TFLOPS	12x
DL Inferencing	21 TFLOPS	120 TFLOPS	6x
FP64/FP32	5/10 TFLOPS	7.5/15 TFLOPS	1.5x
HBM2 Bandwidth	720 GB/s	900 GB/s	1.2x
STREAM Triad Perf	557 GB/s	855 GB/s	1.5x
NVLink Bandwidth	160 GB/s	300 GB/s	1.9x
L2 Cache	4 MB	6 MB	1.5x
L1 Caches	1.3 MB	10 MB	7.7x

Summary of 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**

Thing 1 and Thing 2

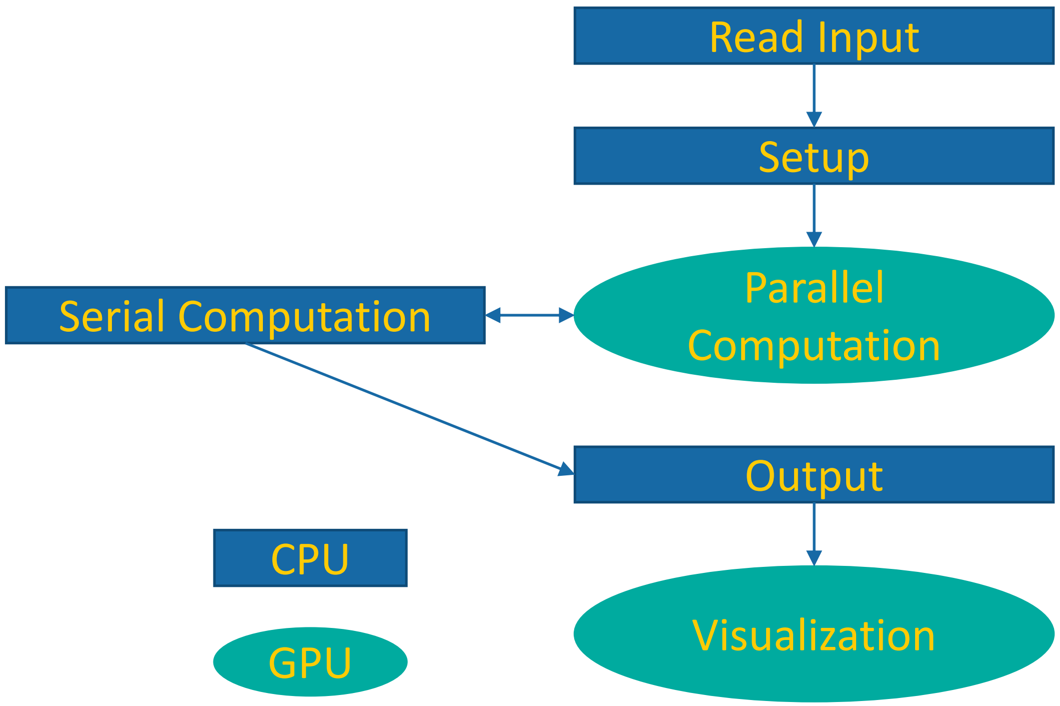
- On GPUs
  - Single Precision is 2x faster than double precision
  - Branching really hurts performance
  - Everyone. Does. The. Same. Thing.

- Need to more explicit about when data is on the GPU or CPU and when you move data between them.

Overview of Programming Models

General Structure of Parallel Codes with GPUs

- 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
  - *S* is the total speedup



The Players Club

- CUDA
- ROCm
- OpenMP
- Kokkos and RAJA

- OpenACC
- OpenCL
- CuPy

Hands On

Hello world Cuda C and Fortran

Hello world OpenMP