* Calculating average peak performance
  + Scenario 1: CPUs executes first than GPUs
    - max(Total CPU FLOPs, Total GPU FLOPs)
  + Scenario 2: CPUs and GPUs work together
    - Total CPU FLOPs + Total GPU FLOPs
* Basics of processors
  + Why move from single-core to many/multi-core?
    - Power constraints became severe (too much heat created and too much power required)
  + Why do we have distributed memory systems if shared memory exists?
    - As the number of cores increases, there is a requirement for more interconnects between those cores and the shared memory, creating more contention over the memory → Memory does not scale to serve many threads/processes
  + What makes code parallelizable?
    - Loops can be made independent (loops can be split into sub-loops)
    - If the loops don’t depend on one another:
      * If they access shared data:
        + OpenMP
      * If they DO NOT access shared data:
        + MPI/CUDA
    - Each instance requires a lot of memory and/or computation
* GPUs
  + SIMD units
  + Criteria for good GPU applications (GPU-friendly)
    - Computationally intensive
    - Independent computations
    - Similar computations (limited or no branch divergence)
    - Problem size is large enough to ensure parallelism
  + Why can GPU code be slower than sequential code?
    - Data transfer to/from the GPU can exceed the runtime of the sequential code (Communication between host/device memory is high overhead)
    - Global memory access overhead
    - Blocks require pre-allocation of resources → No sharing SM
  + SM → Streaming Multiprocessor
  + SP → Streaming Processor
* MPI (Processes) → No Shared memory
  + Communication Basics
    - Messages can arrive out of order → If they took different paths in the interconnect
    - MPI buffers messages → Ensuring out of order does not happen → In-order delivery enforced
      * You can also use MPI\_ANY\_TAG to receive messages with any tag
  + When can 1 core with MPI run faster than simple sequential?
    - If I/O is in one process and the other does computation, and the IO causes blocking as the operation waits for data
    - The two processes are doing different types of computations → Hyperthreading without contention
  + When will two MPI processes run on four cores faster than two cores?
    - Each process is doing multithreading
    - Some of the cores become hot and we can migrate to a cooler core
  + Reasons for performance loss:
    - Communication overhead
    - Load imbalance
    - Process creation overhead
  + Race Conditions (race to modify a variable in shared memory)
    - DNE, because no shared variables or shared memory
  + Example Questions:
    - Why would 2 cores in MPI execute slower than 1 core in MPI?
      * There is a lot of communication between the processes and the overhead slows down the process
      * The 2 cores combined runs slower than the 1 core for whatever hardware reason (for example, the 1 core is designed to run faster than the speed of both cores combined)
    - When to use split communicator command?
      * When you need to do collective communication on subset of processes but one-to-one amongst all the processes in the subgroup will be too much overhead
  + Collective Calls (Example: MPI\_Reduce) → Blocking, each process is required to wait
    - Deadlock happens if all processes do not call this
    - The output is only saved to the root process → If you send all the data as a reduction to process 0, no other process will have the reduced array values
  + MPI\_Finalize()
    - Before this call, we have N number of processes
    - After execution → All four processes still exist
  + MPI\_Comm\_Split → If you have a question on this, look at spring 2020 exam question 1, and remember that the question is most likely very easy → When you split a process, the split starts at 0 → Processes are reindexed
    - Also, thread ranks are split by *colors,* The processes need to have the same colors to communicate with one another
    - Creates **two** disjoint communicators
* OpenMP (Threads) → Shared memory
  + Loop index is made private by default in OpenMP
  + MAKE SURE THE OMP PRAGMA CALL SAYS PARALLEL OR ONLY ONE THREAD IS MADE
  + Nowait → Schedules the current pragma block and moves on → Otherwise it waits until block is done to move on
  + Reasons for performance loss
    - Synchronization
    - Coherence Cache
    - Communication (memory to cores)
    - Load imbalance (divergence)
  + Race Conditions → Is a variable shared and modified
    - Shared memory exists, so it can have race conditions
    - Nowait implemented → Possible race condition (check)
      * Both or any one thread has dynamic → race condition
  + If unspecified, num\_threads = min(num\_cores, iterations\_in\_loop)
  + Scheduling
    - Static → Iterations being assigned prior to execution
    - Dynamic → Iterations are assigned during execution
      * When to use it
        + Independent loop iterations where they each depend on the iteration index (i)
        + Also good if you have no preference because it’s more flexible to performance issues like cache misses
        + Best if work in for-loop body scales non-linearly
        + (Work in loop body increases with loop index)
        + Each thread has a different amount of work → WHEN IN DOUBT, DO DYNAMIC
  + Loop unrolling
    - When the iterations of a loop are predictable
    - When there are not enough cores
  + How to make threads do different tasks (different loop body itself)?
    - If-else/switch-case statements dependent on thread ID
    - Tasks
    - Sections
      * Good if you know number of tasks ahead of time
  + Ways to manage/deal with critical sections?
    - #pragma omp atomic → Only one-line body
      * Only one thread can change variable at a time
    - #pragma omp critical → Multi-line body
      * Only one thread can run that section at a time
    - Locks
      * Lock and unlock a variable → Mutex locks
  + Speeds of critical section management tools
    - Atomic is fastest → Critical → Locks are slowest
  + After block of pragma code is done running
    - All threads collapse into thread 0
* CUDA (Threads)
  + Local variables → Stored in local memory (Not necessarily registers)
  + All threads in only the same block can access shared memory
  + Data in Global Memory
    - Lasts lifetime of the program
    - Can be viewed by any kernel in process
  + Reasons for performance loss:
    - Global Memory Access
    - Data transfer (cudaMemCpy)
    - Branch divergence (causing load imbalance)
  + Race Conditions
    - Exist, because CUDA is shared memory (global variables can be changed by multiple threads)
      * Only if threads in same warp try to modify same variable
    - A better breakdown of this
      * 2 threads in same warp → Can cause race condition because two threads can access same memory location
      * 2 threads in same block but different warp → Can cause RC because they access shared/global memory
      * 2 threads in different blocks but same kernel → Can cause RC, Access same global memory location
      * 2 threads in different kernels but same app → Can cause RC, Access same global memory location
  + Stream → Sequence of operations executed by the GPU (in-order)
    - Examples of use:
      * cudamemcpy and kernel are non-blocking → Make sure they run in-order
      * Executing two kernels in parallel
      * Overlapping data-transfer and computation
  + Branch Divergence
    - NO IF/ELSE = NO BRANCH DIVERGENCE
    - When if/else statements cause one kernel instance to take longer than another to run (load imbalance) → can make loop-unrolling useless
  + Thread divergence
    - Definition: If/Else statement in kernel where some threads in warp go down if part and some go down else part
    - Problem: Threads in a warp need to move in lockstep and the if-part and else-part both will require time to complete
    - How to find out if a kernel has it:
      * It runs the same for all the threads in the block
      * If it is always true or always false for All the threads in the block, then no thread divergence
  + Streaming Multiprocessor (SM)
    - Assigned blocks after resources are acquired
      * Makes context switching instantaneous
      * Scheduling of warp for execution takes 0 cycles
    - Blocks are assigned 1-to-1, after which, leftover blocks are scheduled
  + Warps
    - Execute in lockstep → Each line executes the same time for all threads in warp
      * If they are in the same block but not same warp → They just execute in traditional parallel fashion
      * Two different blocks → Execute at different times
    - \_\_shared\_\_ → Only shared in block
    - Only 32 threads in a warp → Consecutive thread indexes (group of threads)
    - One Warp = executed on = One CUDA core
      * Thread block is divided into a number of warps for execution on cores of SM → Leftover warps will be scheduled
    - Know about warps helps you write better code
      * If num\_thread%32 != 0, you are underutilizing the GPU
      * Knowing about branch divergence can advise you to limit if-else statements in a kernel
      * Memory accessing is more friendly because memory accesses happen per warp:
        + Decoupling number of threads per block from SP/SM
    - Calculate number of warps in each block?
      * Divide number of threads per block by 32
  + Threads create blocks
    - Blocks → Threads in different blocks cannot sync or communicate
    - Blocks are divided into warps (32 per warp)
    - SPs execute the threads in a warp in parallel → NOT THE THREADS IN A BLOCK, BUT THREADS IN A WARP
  + Kernel basics
    - Kernels in a single stream (default stream) are executed in-order, so if one kernel depends on another, just make sure they are sequentially executed
  + How to find best configuration for threads per block
    - Best configuration = Block dimensions and number of blocks that maximize number of threads in SM (has to be less than max num threads in SM)
    - Find some combination of
      * Block\_x\_dim \* Block\_y\_dim \* num\_blocks = max\_blocks/SM
* Amadahl’s Law → Gives upper bound of speedup
  + Problems:
    - Doesn’t take into account memory access overhead
    - Sequential part is hard to calculate
* Cache
  + L2 Miss in GPU → Results in coalesced global memory access
    - We need to go check with global memory because its not in L1 or L2 in GPU
* Processes
  + Can they share cache memory → Yes if they are on the same core
* CGMA (Compute to Global Memory Access) Ratio
  + Floating point calculations to each access to global memory
  + CGMA = FP operations / Memory Accesses
    - # of FLOPs / Memory Accesses
    - Memory Accesses → Accessing global memory or pointers
* Cache Blocks and False Sharing → Fall 2020, Problem 1d
* Superscalar → Execution instructions come from same thread
  + Does not need branch prediction → Will stall when faces a conditional branch instruction
* Hyperthreading → Execution instructions come from different threads
  + Must be superscalar → Executes several instructions at same time