ECE408: Parallel Programming

Pradyun Narkadamilli

Contents

1 Lecture 1

- *Dennard Scaling* - total power is constant for same chip area as transistors get smaller
  - No longer holds true, requires better computer architecture for performance improvements
  - Combatted via multi-core processors, vector engines, latency/throughput oriented heterogenous architecutres, 3D packaging
- If CPU is latency-oriented, then GPU is throughput-oriented
  - GPU has small caches, moderate clock, simple control (no BP, no forwarding), efficient ALUs (heavily pipelined)
  - Requires of thousands of threads for the throughput benefit to outweigh latency hit
  - The GK110 GPU had 2880 CUDA cores on a 28nm process, vs. 10th gen intel CPU w/ 10 cores on 14 nm
- Ideally, use CPU for sequential/control logic, then GPU for parallelizable bits
- We want to make scalable, portable, maintainable code and algorithms

2 Lecture 2

- We consider a *thread* the basic unit of computing (refers to the combination of a program, PC, context [memory, regs, etc.])

  - Can be considered a single execution context

- While some-threaded applications can afford inter-thread communication, in many-threaded context we want very little inter-thread comms required

- *CUDA* - is essentially an integrated host+device C program

  - Serial/parallel-ish parts run on CPU, highly parallel run on the *device* SPMD kernel C code
  - SPMD $\rightarrow$ Single Program Multiple Data
  - Executed on device as a grid (array) of threads - all threads in grid run the same kernel code, but by using thread idx we can vary behaviors

- CUDA kernel grid is a 3D array of thread blocks. Each block is a 3D array of threads (6D structure overall)

  - On SPMD model, we get each thread to execute the *same program* on *different data*

- Some handy dandy execution variables

  - `gridDim` and `blockDim` give us relevant dimensions - consistent per grid and block
  - Each block is assigned a unique `blockIdx`, and each thread is given a unique `threadIdx`
  - All vars have x, y, z members

- Intra-block communication done via *shared memory*, *atomic operations*, and *barrier synchronization*

  - Inter-block threads cannot cooperate

- CUDA functions can be declared with one of three descriptors

  - `__device__ <ret_type> FuncName()` - executed on device, only called from device
  - `__global__ <ret_type> FuncName()` - executed on device, only called from host
    - This is how we define a kernel function
  - `__host__ <ret_type> FuncName()` - executed on host, only called from host

2.1 Case Study: Vector Addition

- We need to migrate data to the GPU
  - Malloc memory on GPU, memcpy host's vectors to GPU vectors
- Kernel will operate per-idx of array (each thread handles an IDX)
- Memcpy output on device back into host
- Free device memory and unused host memory

3 Lecture 3

- Thread blocks have no fixed order - they are scheduled in some arbitrary order
- Threads assigned to *streaming multiprocessors* at a block-level
  - Up to 32 blocks per SM (on the Maxwell architecture)
  - Maxwell can also handle 2048 threads per SM
- Threads are run concurrently, and the SM will manage/schedule execution while maintaining thread/block ids
- Blocks are executed as 32-thread *warps* - these are the fundamental scheduling units of an SM
  - Warps are not exposed to the CUDA programming model, but are an implementation decision of the GPU
  - Warps are divided based on thread index ranges (0-31, 32-63, etc.)
- An SM will implement zero-overhead warp scheduling
  - A warp is eligible for execution if its operands are all ready
  - Eligible warps are selected via a prioty-based scheduling algorithm
  - *All threads in a warp execute the same instruction when selected*
- Different threads within a warp have the potential to *diverge* at branches
  - This is the big performance concern w.r.t control flow

  - GPUs use *predicated execution* - each thread will compute taken/not taken for each path, then the valid permutations of these paths are executed *serially* by the GPU
  - The resulting behavior is that all threads will execute all valid control paths for the warp, even it does not apply to that specific thread

- Performance can be improved by making branch granularity a multiple of warp size rather than granularly heterogenous branching behavior

- Keep in mind that SMs have limitations on both block count and thread count - pick block size/thread count accordingly to fully utilize SMs

4 Lecture 4

- *Von Neumann Model* - Instructions store in memory, decoded, then executed. CPU separates into memory, processing unit, control unit, and I/O
  - ”Fetch, Decode, Execute, Memory” instruction processing
  - Operation, Data Transfer, and Control Flow instructions

- CUDA has some different memories
  - Per-thread registers
  - Per-block *shared memory* (~5 cycles) - declared as `__device__ __shared__` variable
  - Per-grid *global memory* (~500 cycles) - declared as `__device__` variable
  - Per-grid read-only *constant memory* (~5 cycles w/ caching) - declared as `__device__ __constant__` variable

- *Tiling* - we separate the output into a bunch of ”tiles” (sub-matrices of matrix product, for example) and assign each block to produce a different tile
  - When tiling, inefficient or excessive global memory accesses can significantly reduce computational throughput due to memory latencies

5 Lecture 5

- A naive implementation of matrix multiplication (tiling the output) is inefficient, due to bottlenecks on the global memory
  - Each thread requires a full row and column spanning the entire matrix, and it's all done from the original arrays!
  - Induces inefficient global memory accesses
- By tiling the *input data* instead, we can take advantage of locality and use shared memory on a per-block level
  - We can read tiles into shared memory using *multiple threads* to get better memory-level parallelism
- We can tile inputs and use partial products of matrix multiplication for better performance
  - Requires synchronization with barriers to do accumulation correctly
- *Barrier Synchronization* - threads stop at barrier until all threads covered by this barrier (e.g all threads in this block) are complete
- Tiling input via shraed memory changes bottleneck from memory to computational

6 Lecture 6

- *RAM*: Random Access Memory - we assume write/read time is identical
- *DRAM*: Dynamic RAM stores bits on capacitors, has discharge problems
  - Charge/discharge bitline to read/write, connection done via access transistors and a `SEL` line
  - Sense amps used to amplify minor peturbations in bitline voltage on read
  - Most large memories use DRAM
  - Assume that a single `SEL` line connects to ~1000 bitlines

  - These bit lines are muxed between via ~12 bits going into a mux

- DRAM is unclocked, but its interfaces must be clocked
  - Core speed is significantly slower than interface speed
  - DRAM will come out in bursts, usually bursted between different columsn on the same row
  - Can read between multiple banks in parallel to improve burst timing even more (less dead time)

- By default, when doing a matmul of $\boldsymbol{A} \times \boldsymbol{B}$, only the DRAM accesses of $\boldsymbol{B}$ will be coalesced
  - We can use shared memory to better coalesce the memory accesses for *both* matrices

7 Lecture 7

- *Convolution*: $f[x] * g[x] = \sum f[k]g[x-k]$
  - In this class, we assume *centered* filters usually
  - Odd-width filters are preferred for symmetry

- When convolving a *mask* with an *input*, the mask is a good candidate for constant memory

- CUDA Memories
  - Registers take ~1 cycle (per-thread storage)
  - R/W into shared memory is ~5 cycles (per-block storage)
  - R/W into global memory (per-grid storage)
  - R/W into constant memory is ~5 cycles w/ the cache hierarchy (per-grid storage)

- Shared memory is similar to L1 cache w.r.t timing
  - More predictable copy and eviction, since manually managed

- Special constant cache in GPU for constant data, will not be modified during kernel execution
  - Can have higher throughput than L1 cache

8 Lecture 8

- Generating an output block will require more inputs than output cells (halo cells) - 3 strategies for thread allocation
  - Strat 1: Block covers output tile, some threads also load halos into shared mem
    - Global memory accesses are coalesced, no computational branch divergence
    - Branch divergence incurred at tile-loading step, and some extra shared mem needed
  - Strat 2: Block covers *input* tile, some threads disabled during computation
    - Best performance overall, but wasted throughput because halo threads disabled at computation
    - Avoids narrow global accesses (only accessing small halo regions)
  - Strat 3: Block covers output tile, during execution halo cells are pulled from global memory
    - Branch divergence during computation and small halos won't fill memory bust
    - Better shared memory utilization, at the cost of throughput on rearely-used values

9 Lecture 9

- *Stencil Algorithms*: class of data processing algorithms that update an array based on some fixed pattern
- Tiling benefit can be calculated by looking at total accesses over shared memory elements

10 Lecture 10

- Machine learning is a good tool to automate tasks that are difficult to formalize, but easy to perform
  - Recognizing speech, facial detection,etc.

- Easily describable tasks that are hard to perform may be better done as an algorithm, however some subcomponents may benefit from ML
  - Data representation can have a big impact on how easily identifiiable features are
- *Machine Learning*: ability to acquire knowledge or intuition by performing pattern analysis on data
- *Deep Learning*: Representations can be expressed in terms of other representations
  - by layering these conversions and creating a ”conceptual hierarchy” so to speak, the machine can create more complex feature mappings
- A *linear classifier* will take a single vector, multiply it into a weight vector, and produce a scalar
  - A *fully-connected* layer can produce a vector output rather than a simple scalar
- A *multilayer* classifier can sequence an input layer, multiple fully-connected *hidden* layers, then pass into an output layer to produce vector outputs
  - The output layer can be put through an argmax or some other manipulation to produce the final output
  - Ex: each index is a proability of what digit the input is, and then the output is the highest probability
- *Inference* is done via forward-propagation through the layers
  - To *train*, we need to backpropagate through the layers
  - On training data we can compute an error E then adjust the weights proportional to that E
- Layers will sometimes include an *activation function* to ”reposition” the output
  - Sigmoid, Sgn, ReLU, etc.
  - Smoother activation functions make it easier to backpropagate, since the derivative is better defined
  - Softmax function is used to ”normalize” a vector into a probability distribution

11 Lecture 11

- Weights can be calculated via *stochastic gradient descent*
  - Do forward inference, then use the proabilities of each pre-argmax output and the intended label T to calculate error
  - Backpropagate the derivative of said error E to dinf derivative of each network parameter
  - Peturb the network parameters in the opposite direction to the derivative to reduce error
  - Rinse and repeat this process - though doing this too much can cause *overfit*
- An *epoch* in training refers to a single pass through the full training set
- For a large image, fully-connected layer for an MLP will be *huge*, and hidden layers make the computation infeasible
  - Filters are better suited for feature detection in image processing applications
  - A neural network can have *convolutional* layers in addition to its fully-connected layers - these are *CNNs*

12 Lecture 12

- Convolution layers have A input features, B convolution kernels, and a total of $A \times B$ outputs
  - Each input feature can be multi-channel, with a different kernel being used for each channel (but aggregatng into the same final output)
- Subsampling layer used to make representation invariant to transformations like scaling or translation
  - Examples include max, average, L2 norm, etc.
- Trivial parallelization of 2D convolution layer can be done as follows:
  - Grid dimensions parallelize batch, feature, and 2D tile (linearized via row-major order)

  - Each index in each tile computes a different pixel for a specific feature of a specific batch
  - Potential for shared memory usage, but need to analyze the reuse patterns

- Subsampling by Scale N: averages an NxN block then calculates the sigmoid on that value
  - Same higher order output dimensions as conv ouptut, but smaller H/W per feature
  - Can be merged into the convolution layer to save bandwidth

- Convolutions can be converted to matmuls by unrolling the filters and input features
  - Outputs directly correspond to a pixel in the output feature

- Transformer Language Models add self-attention layers to the expected feed-forward layers
  - Usually a tokenized version of input text fed into transformer-based LLM, which then spits out tokenized output that needs to be converted to text

13 Guest Lecture: Nsight Profiling

- Want to be able to understand CUDA tools to unpack and understand GPU performance
  - Specifically *GPU utilization*

- Nsight Compute used for *kernel-level* profiling
  - Mostly GPU execution speed

- Nsight Systems used to profile at system-level
  - System dispatch speed (sending work to GPU)
  - System-GPU parallelism
  - System-GPU data transfer speed
  - How much time CPU takes to control GPU
  - CPU/GPU asynchronicity

- Profiling data is recorded on Delta server, then we analyze it locally
- Host code requires specific includes and linker arguments to be profiled well
  - Make sure kernel is memory-safe before profiling, since it can cause new errors
    - cuda-memcheck
  - Normally, system may not always catch memory failures, among other types of silent errors
- Remove debug flags when profiling - these kinds of flags can induce runtime slowdowns and prevent optimizations
  - Add line number annotations so that profiler can map from compiled code to codebase locations
- Profiling good to figure out behavior and tendencies of program, but should not be considered a *benchmark*
  - Benchmarks require optimal compile parrameters, and profiling can impede kernel performance to collect data
- Stream refers to queue of sequential CUDA events
  - Program can use multiple CUDA streams
  - Operations are overlapped using different streams
- Events record the state of a CUDA stream
- For Nsight Compute, data is collected via a CLI on the target platform, then analyzed with GUI on client
  - Main thing to know about on the GUI side is *SOL* or speed of light
  - SOL shows the utilization% compared to the theoretical maximum
  - GUI can also show scheduler statistics, like warp stalling, how many warps were issued/given to SM, etc.
  - Warp state statistics avilable, like how many cycles it takes for instructions to evaluate, and what the inter-instruction latency is

- Nsight Systems used to analyze system-GPU interactions and performance
  - Similar CLI/GUI flow
  - Many of our labs can be decomposed into CPU-to-GPU transfer, GPU execution time, then GPU-to-CPU transfer time
  - We could parallelize the memory transfer and kernel execution for performance improvements

14 Lecture 14

- *Scan*: given an array, calculate the sum of an operand and all the elements that came before it
- Various scan algorithms exist
  - *Naive*: Each thread calculates an element in the scan array
- *Reduction Trees*: Also called Kogge-Stone or a Kogge-Stone Tree
  - Calculate each output element as a reduction of all previous elements, share some reduction partial sums
  - Iterate from a stride of 1 to $\frac{N}{2}$. For each stride amount, set `a[k] = a[k] + a[k - stride]`, updating the top $(N - \text{stride})$ elements on each iteration
  - Runtime is $O(\log(n))$, but total add ops is $O(n \log(n))$ - much less efficient than naive sequential's $O(n)$
  - Typically used on a per-block basis
  - Blocksize is generally matched to shared array size

```
kogge_stone:
  shared = in

  for (int i=0; i<logstrides; i++){
     if (elem >= stride)
        temp = shared[i] + shared[i - stride];
     __syncthreads();
     if (elem >= stride)
        shared[i] = temp;
     __syncthreads();
}

  out = shared
```

Can double buffer the shared array used in $\text{kogge}_{\text{stone}}$ to remove a syncthreads without a race condition. Output/input roles swap per iteration.

15 Lecture 15

- Balanced trees can help improve work efficiency - e.g a *Brent-Kung Parallel Scan*
- *Brent-Kung Scan*: Consists of a Scan step and a Post-Scan step
  - Generally done with the the blocksize as half of the block's data size
  - Scan step is an adder tree, constructing the highest order sum with the tree
    - Update intermediate elements as you go, of course. Can use double buffer here for better performance
  - Post-Scan step is to update all the lower order, elements after the initial scan
    - Iterate over strides again - the highest order element of each stride segment is used to element the element ahead of it by `stride/2`.
    - Ex: Suppose you have a scan of size 8. Final reduction stride was 4. You take element 3, add its value to element 5. Keep going until the stride segment size of 2.
    - Effectively the same as the reduction tree, but the tree is shifted to the right by (stride-1) - one node removed per level
- Brent-Kung algorithm will perform a total of $O(2\log(n))$ iterations as opposed to Kogge Stone's $O(\log(n))$. However, it does it on a smaller blocksize for the same amount of data, so warps*iterations is identical for a fixed length array.
  - Kogge-Stone is more popular in GPUs due to the lower iteration count

- Scan generally has to be done in ”segments” - you take scan of segments of the array
    - Take highest order element of each scanned subarray, store it into an aux array, then perform a scan on *that*
    - Used the first $N-1$ scanned elements of this aux array to increment elements of the last $N-1$ subarrays
    - This incrementation strategy will produce a final array that is fully scanned
- *Exclusive scan* will take sum of prefix elements, with the last element of the array not being the total sum anymore
    - More useful in some applications, like finding starting memeory addresses
    - An efficient exclusive scan is identical to an inclusive scan algorith, you just shift the loaded shared array to the right by one element, with the gap in position 0 filled with a zero
    - Inclusive scan can be derived from exclusive by adding first element to every element in array
    - Inverse for deriving inclusive from exclusive

16 Lecture 16

- In highly parallel programs, writes to shared memory w/o synchronization can cause read invalidation
    - Data that was read out of an address may be changed by the time you overwrite it
    - Need to use synchronization, critical sections, or atomics to avoid this artifact
- Atomicity does not constrain relative order, cannot be used to enforce a global memory access ordering
- Many ISAs have atomic instructions (ex: RISC-V CSR modification/read instructions)
    - Test and set, compare+swap/exchange, swap/xchange, fetch and add

- Threads are queued when making accesses to a memory location atomically - atomics performed serially
- *Histogramming*: Perform frequency analysis on a large data set to extract features/patterns
  - Map each element to a bin, increment that
  - Easy enough for sequential code, hard to parallelize because of race conditions
  - Long DRAM delays mean that it is bottlenecking to use atomics directly on global mem for every op
  - Better to perform intrablock atomic instructions into shared memory, then coalesce data with other blocks into global memory via more atomic increments
- **General Histogramming Algorithm**
  - Make a shared coyp of the intended histogram
  - Iterate block over the input segment, while using a coalescable access pattern
  - Use atomic operations on shared memory copy to construct a block-level histo
  - Use atomicadds onto a global output bector to construct global histo
- General principle above called *privatization*

17 Lecture 17

- Sparse Matrix matmul is very irregular, has low input data reuse, and compiler transformations are not super helpful
- Want to somehow improve the regularity of the sparse matrix structure and improve layout for DRAM bursts
- Many possible sparse matrix representations, each with their own benefits and drawbacks
  - *Compressed Sparse Row*: Nonzero elements of each row are serialized into a data array

    * Second corresponding array, denotes column index for each element
    * Keep an array w/ one element per row (and an extra sentinel), denotes the index in data array where that row starts
    * A simple kernel could assign one thread per row, use row pointer to pull out iteration bounds on data array
    * Lots of control divergence, since iteration bounds can vary per element, and therefore per thread
    * Adjacent elements in data array are not adjacent rows, so bad memory coalescing in kernel
  - *ELL Format*: take CSR, pad all rows in data array to the same length, then transpose the data/column arrays
    * Move forward by `num_rows` as you move across the row in data array (it is now column major)
    * ELL *does not* ignore rows with all zeroes - that gets padded too. Duplicate column indices indicate invalid elements
  - *COO Format*: Each element in data is given a column index and row index (2 aux arrays)
    * Allows for element reordering
    * Easy to make a sequential kernel for this, but COO will usually end up requiring atomic operations
  - *Hybrid Format*: Encode the first N nonzero elements in each row with ELL (pad rows with less than N)
    * Any ”outliers” (elements above the first N nonzeroes, where N is typical to most rows) are encoded in COO
    * COO elements are used to peturb the initial array computd from the ELL elements
    * Hybrid format helps reduce space overhead induced by ELL padding, while retaining benefits of ELL

- Note: for a matrix w/ column or row pointer array, the last element will be the total number of elements in data array

18 Lecture 18

- With traditional CSR, adjacent elements can have wildly different nonzero elements

    - With ELL, padding can waste space or cause unecessary iteration counts on smaller rows

- Can sort CSR rows by number of nonzeros to form a *Jagged Diagonal Sparse* matrix (JDS)
    - CSR is sorted, meaning data array is no longer ordered by row
    - In addition to `row_ptr` array, keep a second array like `row_perm` which tells you which row each range in `row_ptr` corresponds to
    - Can use algorithm very similar to typical CSR, but now adjacent threads have more similar iteration counts
    - Less likely to have control divergence, even though we basically just use the CSR algo
    - Can do the transpose JDS data array to improve memory coalescing
    - After transposition, typicall `col_ptr` array is used instead of `row_ptr` - helps easily determine which threads should be ”on” for each iteration

19 Lecture 19

- Performance bottleneck will probably end up being bandwidth between key components
- Important to understand how old computer architectures looked
    - North bridge connected to CPU, connects it to high speed devices (AGP bus, South bridge, RAM)
    - South bridge connected to slower devices and I/O, i.e SATA/ATA devices and PCI buses, acts as a concentrator
    - NVIDIA GPUs used to be connected over the AGP bus, 2GB/s
- PCI bus originally was a shared bus on the south bridge, shared bus with arbitration
    - PCI’s device registers were mapped into the memory map for the CPU, addresses assigned at bootup
- PCIe (PCI express) is instead a switched point-to-point connection direct from the CPU

  - CPU connects to a central switch, which has a link to each device
  - Effectively acts as a network switch, packet switching applies here, with packet priorities for QoS

- PCIe links can have a different number of lanes

  - A PCIe gen 3 link consists of 1 or more lanes - each lane is 1 bit, and has 4 wires
  - Each lane has 2 inbound and 2 outbound wires. Differential signalling used for each bit, and inbound/outbound can be simultaneous
  - Each lane transmits data as a rate of 1GB/s on its own
  - The link width is varying at the switch itself, since the PCIe switch has a singular connection to the CPU
  - Bytes are encoded w/ 128b/130b encoding, which has an equal number of 1/0s
  - 128/130 encoding maintains DC balance while having sufficient state transition for clock recovery
    - Each 128 bit combination is mapped into a 130 bit sequence
    - Only 1.5% overhead instead of 8/10!
    - Runs of 1s are vanishingly small, mostly long runs of 0s
    - One shift guaranteed every 66 bits

- Recently PCIe has been used as the interconnect ”backbone” in PC - Northbridge/Southbridge are PCIe switches

  - May need a PCIe-to-PCI bridge to support older PCI devices. PCIe controllers are integrated on-chip with the CPU itself

- PCIe data transfer is done via DMA controller, dumps data directly into the DRAM

  - Requires pinned memory, so that PCIe/CPU can reliably use the same region for communication
  - If not pinned, DMA’s physical memory region can be ”paged out” and assigned to a different virtual page during transaction
  - `cudaHostAlloc` will allocate *pinned* memory, prevents being paged out, free it with `cudaFreeHost`

- `cudaMemcpy` will only work with pinned memory, so copying into a malloc region is actually *two* transactions
    - Copy to pinned region
    - Copy from pinned region to actual malloc'd region
- If using hostAlloc to get pinned memroy, cudaMemcpy will be 2x faster
- Overallocation of pinned memory is bad - CPU rotates out virtual memory to ensure large virtual address space can be mapped onto smaller PADDR space, don't want to fuck this up
- NVLink is a Multi-GPU and GPU-CPU interconnect - connects GPUs between multiple systems together in a mesh topology of some sort
    - EX: Connecting 8 GPUs together in a hybrid cube mesh
    - GPU-GPU transfers over NVLink are 160GB/s bidirectionally, ~5x faster than over PCIe
    - Some CPUs, like IBM's Power9, can use NVLink to connect to the GPU itself. This makes GPU-CPU connection ~150GB/s
    - Based on diagram, Delta exclusively uses PCIe gen 4 to connect GPUs together?

20 Lecture 20

- Most CUDA devices allow interleaving Device-Host data transfer with GPU kernel execution
    - Overlap transfer and compute of adjacent segments
    - PCIe is bidirectional, output transfers can be overlapped with computation and input transfers as well
- CUDA abstracts this interleaving as "streams" - queue of kernel/cudaMemcpy operations
    - Different streams can be executed in parallel - take advantage of this to absolve yourself of serial dependencies
    - Host thread enqueues operations, driver will dequeue them - one queue per stream

- Stream operations are enqueued via *asynchronous functions*, e.g non-blocking functoin calls
  - Kernel can take an additional launch parameter determining which stream it links its execution to
- Keep in mind that there is *a single copy engine and kernel engine* - they get shared by all streams
  - AFAIK age-ordering is the arbiter strategy between multiple conflicting streams, so want to make sure Async calls are made in intended order to maximize parallelism
  - Fermi architecture and earlier used to serialize multiple stream queues into a single operational queue - bad for inter-stream parallelism, leads to Head-of-Line blocking
- Task runtimes are affine functions, so moderate segment/stream blocksize is required to make it yield performance benefits
  - Best "moderate" size can vary from system to system - profile to figure it out!

21 Lecture 21

- CUDA is just one model for parallel computation
  - Many other models for compute accleration and parallelism
  - Ex: OpenCL, ROCm, etc.
- General set of traits for any acceleration API
  - *Hardware*: Some hierarchy of lightweight cores, local (scratchpad) memories, no HW coherence for SPEED, some slower global atomics, and a threading model
  - *Software*: Kernel-based acceleration, device/host memory separation, software-managed memory, Grid/Block/Thread
- OpenCL: generic framework for CPUs, GPUs, DSP cores, FPGAs, etc.
  - Models system as a host/context separation - context is the "hardware", with work-groups instead of SMs/Blocks

    - Work-groups have local memory, and work-items have private memory (registers, I think)

- Some frameworks, like OpenACC, use pragmas on top of typical C++ code instead of defining language extensions
    - Allows for a ”single version of code” for both sequential and parallelized version
    - OpenACC can infer the parallelism constructs, and pragmas can be ignored by other compilers
    - Nice idealism, but code may not work if pragmas are ignored and sequentially compiled
    - Strong dependence on compiler may not be ideal - parallelism is less user-specified and more compiler-inferred
- OpenACC models device with execution units, each executing a set of threads concurrently
    - No user-specified cross-thread synchronizatoin
- OpenACC has a ”gang” consisting of many ”works”, which execute many vector operations in parallel
    - Like block vs. thread basiclaly, and set of gangs is the grid
- Pragmas tell the compiler about data movement, available parallelism, and what to treat as a parallel region
    - `copyin` and `copyout` denote how to send data on and off the device
- Parallel region/construct is executed on accelerator, number of gangs/works can be specified (grid/block size)
    - Can specify how to stripe work across gangs, or gangs will just execute same work on each one
- *MPI Model*: Many ”nodes” with distributed processes
    - Each process computes partial output, and different processes communicate via message-passing
    - Saome global synchronization constructions, but data communication is all message-passing

22 Large-Scale Inference/Training

- Power is the main limiter for AI training - need to improve power generation, delivery, and efficiency
  - Bottleneck will arise by 2030
- Batch size of training determined by memory bandwidth and desired backward pass frequency
  - Can do it faster by separating data into multiple batches (mini-batching), evaluate those parallely on many GPUs to aggregate gradient and figure out an update
- Data parallelism is done via a parameter server
- *Parameter Server*: Many GPUs can infer model in parallel, send deltas to a parameter server, which propagates changes to all GPU before next per-GPU batch
  - Centralized server means that all workers communicate with it, bandwidth/connection constraints limit scaling
- Can perform AllReduce, so element-wise reduction across multiple GPUs to generate a more consistent output, require fewer GPUs contacting parameter server
- Our mini-batch parallelism relies on having each GPU have a different copy of the model - bottlenecked by GPU memory capacity, so you can't have larger models
- This led to *model parallelism*, so model is split between multiple GPUs by either assigning layers to each GPU or splitting a single layer's operation into multiple GPUs
- *Tensor Model Parallelism*
  - partitioned Tensor Model Parallelism separates the output into multiple GPUs (quadrants of matrix generated by different GPUs)
  - reduction Tensor Model Parallelism computes partial sums for the output on each GPU, then performs a sum reduction across the partial outputs (partition inputs)
- We also have *pipeline parallelism*

  - Split the sequential layers into multiple GPUs (pipeline the GPUs in sequence!)
  - Can run into GPU utilization issues, due to bubbling as the model propagates through, but can allow for larger models
  - Can separate minibatch into *microbatch* to reduce bubbling (less dead time) - different layers/GPUs will have parallelism while processing different microbatches
  - There will be a bubble between the forward pass and backward pass, and then a flush to kick off a new forward pass

- Modern implementations will combine pipeline parallelism, data parallelism, and tensor model parallelism
  - Data parallelism for large batch, Tensor Model Parallelism to run larger models, and Pipeline Model Parallelism to support deeper models
  - All this parallelism requires strong communication networks in between the GPUs

- Computer/Power Density is constrained by copper and cooling, getting better due to liquid pooling

- Parallelism of LLMs is very similar for inference and training - performance optimizations are a bit different
  - KV caching - memoizes prior computation state, since as you grow the context you can reuse some of the computations by caching the KV product
  - Attention is getting appended to the KV state, so you can cache that, so you only really generate the APPENDATIONS as you go forwards

- Reducing floating point precision can improve power-to-performance ratio, and some algorithms can reduce energy consumption at all

23 Lecture 22

Intel has AMX that is very similar to NVIDIA's tensor cores, but it's used in CPU code (SIMD engine). Google TPU is also effectively a very complex matmul accelerator.