

ECE 408

Fall 2025

Review Notes for ECE 408 (Applied Parallel Programming)

This is in no way comprehensive.

Contents

1	Midterm 1	1
1.1	Why do GPUs exist?	1
1.1.1	End of Dennard Scaling	1
1.1.2	Amdahl's Law	1
1.2	Basic Organization of CUDA	2
1.3	High-Level Architecture of Modern GPUs	2
1.3.1	Block Scheduling	3
1.3.2	Barrier Synchronization	3
1.3.3	Warps and SIMD Hardware	3
1.3.4	Control/Branch Divergence	4
1.3.5	CUDA Memory Model	4
1.4	Matrix Multiplication	4
1.4.1	Naive Implementation	4
1.4.2	Tiled Matrix Multiplication	5
1.5	GPU Memory Systems	5
1.5.1	Memory Coalescing	6
1.5.2	Caches	6
1.6	Miscellaneous Optimization Strategies	7
1.7	Convolution	7
2	Midterm 2	8
2.1	Reduction	8
2.1.1	Parallel Reduction Algorithm	8
2.1.2	Further Optimizations for Parallel Reduction	9
2.2	Scan (Prefix Sum)	9
2.2.1	Kogge-Stone Scan	9
2.2.2	Brent-Kung Scan	10
2.2.3	Hierarchical Scan	11
2.3	Histograms and Atomics	11
2.4	Sparse Matrices & SpMV	12
2.4.1	COO (Coordinate) Format	12
2.4.2	CSR (Compressed Sparse Row) Format	13
2.4.3	ELLPACK Format	14
2.4.4	JDS (Jagged Diagonal Storage) Format	14
2.5	GPU Architecture, cont.	15
2.5.1	Guest Lecture, Tensor Cores	15
2.5.2	Dynamic Parallelism	15
3	Transformers	16
3.1	High-Level Overview of Transformers	16
3.2	Attention Mechanism	16

3.3 Feed Forward Network (FFN)	17
3.3.1 GeLU Activation	17
3.3.2 (Brief) Overview of Noteable Optimizations	18

1 Midterm 1

1.1 Why do GPUs exist?

Moores Law - observation that number of transistors on ICs double every 18-24 months.
Dennard Scaling - As feature sizes decrease, energy density remains constant and clock speeds increase.

- $P \propto C f V^2$ and capacitance C is proportional to area
- Exponential increase in clock speed
- Increased transistor density meant memory went from being expensive to effectively infinite

1.1.1 End of Dennard Scaling

Dennard Scaling ended around 2005/6, clock speeds stagnated, and we needed different methods to achieve performance expectations.

- ILP (Instruction Level Parallelism)
- Manycore Systems
- Specialization, including GPUs

CPUs vs. GPUs

- CPUs are latency-oriented (large ALUs, FUs, large caches, branch prediction, data bypassing, out-of-order execution, multithreading to hide short latency)
- GPUs are throughput-oriented with many small ALUs, small caches, simple control logic, and massive multithreading capabilities
- CPUs wins perf-wise for sequential, latency-heavy code. GPUs win perf-wise for parallelizable, throughput-focused code.

CUDA - Computing Unified Device Architecture

Threads - a PC, IR, and context (registers & memory)

- Many threads → context switching becomes inconvenient
- we'd like to avoid communication between threads as much as possible

1.1.2 Amdahl's Law

$$\begin{aligned} t &:= \text{sequential execution time} \\ p &:= \% \text{ parallelizable} \\ s &:= \text{speedup on the parallelizable part} \\ t_{\text{parallel}} &= \left(1 - p + \frac{p}{s}\right) \times t \end{aligned} \tag{1}$$

Effectively, the maximum speedup ($\frac{t_{\text{sequential}}}{t_{\text{parallel}}}$) is limited by the fraction of execution that is parallelizable.

1.2 Basic Organization of CUDA

CUDA integrates the device (GPU) and host (CPU) into one application. The host handles serial/moderately parallel tasks, whereas the device handles the highly parallel sections of the program.

CUDA kernels are executed as a grid of threads

- All threads in a grid run the same kernel (SIMT)
- Each thread has a unique index that can be used to index into memory/make control decisions

In CUDA, threads are organized within blocks

- Threads within a block can cooperate via shared memory, barrier synchronization, and atomic operations

Threads within a block are 3D, blocks within a grid are also 3D.

```
gridDim.x // gives you # of blocks in grid (in x axis)
blockDim.x // gives you # of threads within a block (x axis)
blockIdx.x // gives you the index of the block within the grid (x axis)
threadIdx.x // gives you the index of the thread within the block (x axis)
```

Host and Device have their own separate memories with some interconnect between them (PCIe, iirc). Thus, for most programs you have to:

1. Allocate GPU memory
2. Copy data from CPU to GPU memory
3. Perform computation using GPU memory
4. Copy data from GPU to CPU memory
5. Deallocate GPU memory

The `__global__` keyword defines a kernel (callable from host/device, but executes on device). There also exists `__host__` and `__device__` keywords that are callable/executes from host and device, respectively.

- `__global__` must return void, but the other two can return non-void

Example: `__global__ vecAdd(float* A, float* B, float* C, int n)`

To launch this kernel, you can do the following:

```
vecAdd<<<dimGrid, dimBlock>>>(A_d, B_d, C_d, n); where dimGrid is the number of
blocks per grid, and dimBlock is the number of threads per block.
```

There exists a `dim3` type in CUDA which makes multidimensional grids/blocks easier to launch.

Blocks can be executed in any order.

1.3 High-Level Architecture of Modern GPUs

- Organized into an array of highly threaded **streaming multiprocessors (SM)**

- Each SM has multiple streaming processors (CUDA cores), which share control logic and memory resources
- Memory Hierarchy - the shared global memory is DRAM (slow), local memory for each SM is SRAM (fast)

1.3.1 Block Scheduling

1. Kernel Called
2. CUDA runtime system launches the grid
3. Threads are assigned to SMs on a block-by-block basis. All threads in a block are assigned to the same SM. Usually multiple blocks per SM
4. Limited number of SMs - the runtime system keeps a list of blocks that need to be executed, and when a block finishes execution, a new block is assigned to that SM

1.3.2 Barrier Synchronization

CUDA allows threads in the same block to coordinate activity using the barrier synchronization method `__syncthreads()`

`__syncthreads()` holds a particular thread at the program location of the call (PC) until every thread in the same block reaches that location. **All threads need to be able to reach this program location, and execute `__syncthreads()`**

CUDA runtime system ensures all threads have the (memory) resources to arrive at the barrier.

Threads in different blocks can't perform barrier synchronization, but this is good because it allows the CUDA runtime system to execute blocks in any order relative to each other (thus, programs can scale easily).

1.3.3 Warps and SIMD Hardware

As a programmer, one should assume that threads in a block can execute in any order wrt. one another (hence why barrier synchronization is so important).

Once a block is assigned to a SM, it is divided into 32-thread units called **warps**.

- Warps are the unit of thread scheduling in SMs
- Blocks are partitioned into warps on the basis of thread indices
- If a block doesn't have a clean multiple of 32 threads, the last warp is padded with inactive threads
- Multidimensional blocks are projected onto a linearized row-major layout before being partitioned into warps

SM implements zero-overhead warp scheduling

- Warps are only eligible for execution once all of its operands are ready

Von Neumann Model - A basic computer Architecture

- data and programs are stored in the same memory unit

- control unit (which has PC, IR), processing unit (ALU, Register File), and I/O

Control units in modern processors are very complex, including fancy fetch logic, separate instruction/data caches, etc. SMs in GPUs are designed to execute all threads in a warp using SIMD (Single Instruction, Multiple Device)

- One instruction is fetched and executed for all threads
- Relatively simple control HW compared to CPUs, and its shared across multiple execution units
- Shared control units in SIMD designs result in significantly less power/area costs

1.3.4 Control/Branch Divergence

Control Divergence - different threads within a warp taking different branches. This is a disadvantage of SIMD designs.

When faced with control divergence, GPUs use predicated execution, where they sequentially execute both branches.

We can resolve control divergence issues by making branch granularity a multiple of warp size, so that all threads within a warp share control flow.

1.3.5 CUDA Memory Model

Memory hierarchy once again: Registers (SRAM) are fast (~ 1 cycle), but few. Main memory is slow (~ 100 s of cycles), but huge (GBs or more)

Each Thread can:

- R/W per-thread registers (~ 1 cycle)
- R/W per-block shared memory (~ 5 cycles)
- R/W per-grid global memory (~ 500 cycles, but there are L2/L1 caches which can reduce this)
- Read-only per-grid constant memory (~ 5 cycles with caching)

1.4 Matrix Multiplication

1.4.1 Naive Implementation

Assign one thread to each element in the output matrix, read from global memory for each value in the output matrix.

This approach sucks because the global memory bandwidth cannot supply enough data to keep all of the SMs busy.

Let's assume we have a GPU which has 1000 GFLOP/s of compute power, and 150 GB/s memory bandwidth. In the naive implementation, each time we write into the output matrix, we perform two FP operations (multiply-add). Furthermore, every time we do these two operations, we have to read 8B of memory from global memory (float is 4B). Thus, its 4B/FLOP.

$(150 \text{ GB/s})/(4\text{B/FLOP}) = 37.5 \text{ GFLOP/s}$, which is significantly less than the theoretical maximum of 1000 GFLOP/s

1.4.2 Tiled Matrix Multiplication

A better approach at matrix multiplication, which uses shared memory to avoid unnecessary global memory reads.

Keep in mind, shared memory has a much lower latency than global memory!

To declare shared memory within a kernel, use the `_shared_` modifier

- example: `_shared_ float subTileN[TILE_WIDTH][TILE_WIDTH];`

High-level Idea:

- Break input matrices into NxN tiles
- Read tile into shared memory
- Each thread can then read this local tile from shared memory
- Repeat until we've computed the output matrix

While implementing tiled matmul, we need to use barrier synchronization to ensure that the shared memory tile has been completely loaded before we proceed with computation. This idea of:

doing some work → waiting for threads to catch up → repeat
is called **bulk synchronous execution** and dominates HPC applications.

The use of large enough shared memory tiles shifts the bottleneck in Matrix-Matrix multiplication. ex: Same GPU with 1000 GFLOP/s compute, 150 GB/s memory BW. If we use 16×16 tiles, we reduce global memory accesses by a factor of 16.

Thus, $(150 \text{ GB/s})/(4\text{B/FLOP}) \times 16 = 600 \text{ GFLOP/s}$.

If we use 32×32 tiles, we get a theoretical 1200 GFLOP/s, at which point memory bandwidth is no longer the bottleneck.

Shared Memory Limitations

- Implementation Dependent
- 64kB per SM in Maxwell architecture
- Ex: tile width of 16 → 256 threads/block → $2 \times 256 \times 4\text{B} = 2\text{kB}$ of shared memory/block → upper limit of 32 active blocks
- However, there is a maximum of 2048 threads/SM, which inherently limits number of blocks to 8.

1.5 GPU Memory Systems

SRAM - *dual inverter feedback loop with two NMOS transistors for R/W (6T design)*

DRAM - *literally a NMOS transistor and capacitor chained together, alongside a BIT and SELECT line.*

- destructive reads, must be rewritten (making it dynamic)

- many DRAM cells share a bit line ($\sim 1k$)
- DRAM bank - A 2D array of DRAM cells w/ sense amps for higher speed/reading tiny currents
- Row Address \rightarrow Row Decoder \rightarrow DRAM Array \rightarrow Sense Amps \rightarrow Column Latches & MUX
- DRAM never returns one bit, but rather a row burst
- Accessing data in different DRAM bursts is slow, but accessing data within the same burst is so much faster because of the column latches.

1.5.1 Memory Coalescing

Memory coalescing occurs when threads in the same warp access consecutive memory locations within the same burst, at which point the hardware coalesces them into one DRAM transaction.

- Multiple transactions within a warp is called memory divergence
- Without caching, DRAM accesses can be 100s of cycles, so we want to maximize memory coalescing if possible
- Use of shared memory generally enables coalescing

(Trivial) Example:

```
int i = blockDim.x * blockIdx.x + threadIdx.x;
z[i] = x[i] + y[i]; // consecutive threads access consecutive memory locations
```

1.5.2 Caches

Caches are an “array” of cache lines, each of which can hold data from several consecutive memory locations (spatial locality). When data is requested from global memory, an entire cache line that includes the specified data is loaded into cache.

- Cache data is technically a copy of the original data, but we need to write-back to global memory if it has been modified (cache coherence)
- Employs tags and indexes (size dependent on cache associativity) to map data to/ from main memory
- Due to being substantially smaller than main memory, caches need some method to make room (eviction) for new lines once full. A commonly used eviction policy is LRU (least-recently used).

Spatial vs. Temporal locality

- Spatial: consecutive memory locations are caches
- Temporal: data accessed repeatedly in a short period of time is caches (may also move from L2 \rightarrow L1 cache)

The programmer can control shared memory contents, but only the microarchitecture controls caching behavior—except for the constant cache.

Constant Cache/Constant Memory

- Read-only, does not support WB to global memory
- Declared as global variable, outside of the kernel: `__constant__`
- Must initialize constant memory from host with `cudaMemcpyToSymbol()`
- Can only allocate up to 64kB

1.6 Miscellaneous Optimization Strategies

Thread Coarsening

- A thread is assigned *multiple* units of parallelizable work
- **Advantages**
 - reduces overhead incurred by parallelization
 - ex: redundant memory accesses/computations, control divergence
- **Disadvantages**
 - underutilization of resources
 - more resources per thread which may affect occupancy

Loop Unrolling

- Less loop iterations → fewer branches (long latency without good branch prediction)
- Exposes independent instructions for Instruction Scheduling
- Controlled w/ preprocessing directives (ex: `#pragma unroll 4`)

Double Buffering

- Eliminates false data dependences by using a different memory buffer for writing data than the memory buffer containing the data being read

1.7 Convolution

Idea: Convolution filter (kernel, mask) “slides” across input, we take a dot product between the two, and the result is ONE entry on the output.

Convolution Filter is unchanged in execution

- Good choice for constant memory (~5 cycles w/ caching)

Tiled Convolution

- Just like with matrix multiplication, reading input values from global memory each time will murder your performance.
- There is a lot of potential for reuse with convolution

Three Main Tiling Strategies:

- **Strategy 1**
 - Thread block size covers output tile
 - Include halos into shared memory
 - Advantages include global memory coalescing, and no branch divergence during computation

- ▶ Disadvantages include some threads doing >1 load operation, resulting in branch divergence. We also need slightly more shared memory (trivial)
- **Strategy 3**
 - ▶ Thread block size covers output tile
 - ▶ Threads read halo values directly from global memory, load only “core” values into shared memory
 - ▶ Advantages include optimal reuse of shared memory
 - ▶ Disadvantages include branch divergence, and no memory coalescing as you’ll only have a few threads accessing a few halo values from global memory (although this is less of an issue on modern GPU’s which have large caches)
- **Strategy 2**
 - ▶ Block size covers input tile (input tile size = output tile size + $2 \times$ mask radius)
 - ▶ Load input tile in one step
 - ▶ Some inactive threads when calculating output

2 Midterm 2

2.1 Reduction

- Reduction reduces a large array of data into a single value, using a **commutative** and **associative** operator ($+$, $*$, min, max, etc.)
- Sequential reduction is $O(N)$. Example sequential implementation is shown below:

```
// sequential reduction
float sum = 0;
for (int v : array) {
    sum += v;
}
```

- For relatively small N , we prefer the sequential algorithm due to the kernel launch & memory transfer overhead with launching the parallel reduction kernel.

2.1.1 Parallel Reduction Algorithm

- Reduction trees, which allow for $O(\log(N))$ “steps” and $O(N)$ total operations.
- Each thread is assigned to 2 elements in the input array
- The naive implementation uses a stride which doubles on each step, but the memory access pattern is uncoalesced, and it suffers from control divergence.

```
// naive parallel reduction
for (unsigned stride = 1; stride < blockDim.x; stride *= 2) {
    __syncthreads();
    if (threadIdx.x % stride == 0) {
        partialSum[2*tx.x] += partialSum[2*tx.x + stride];
    }
}
```

```
// the __syncthreads() is necessary to avoid errors stemming from race
conditions
```

- The better reduction strategy stores values in contiguous segments of memory, which allows us to make better use of DRAM, and reduce the impact of control divergence.

```
// better parallel reduction
for (unsigned stride = blockDim.x; stride >= 1; stride /= 2) {
    __syncthreads();
    if (tx.x < stride) {
        partialSum[tx.x] += partialSum[tx.x + stride];
    }
}
```

2.1.2 Further Optimizations for Parallel Reduction

- We can always load a segment into shared memory, and perform reduction on that segment. This would involve an initial reduction step like:

```
__shared__ sums[BLOCK_SIZE];
sums[tx.x] = partialSum[tx.x] + partialSum[tx.x + BLOCK_SIZE];
```

- We can use warp-level primitives once the reduction tree hits a certain point (these warp-level operations use register memory instead of shared memory)
- Thread Coarsening

2.2 Scan (Prefix Sum)

- Similar to reduction, operator must also be commutative and associative.
- Sequential scan algorithm is as follows:

```
// sequential scan (inclusive)
output[0] = input[0];
for (unsigned i = 1; i < N; i++) {
    output[i] = input[i] + output[i - 1];
}
```

2.2.1 Kogge-Stone Scan

- A hardware adder topology as well
- Uses reduction trees (each element is a reduction of all previous elements)
- Low latency, but not work efficient.
- 1-1, number of threads is the same as number of elements in shared memory
- Following code implements the high-level idea of Kogge-Stone

```
// kogge-stone implementation (naive, has bugs)
__shared__ float buffer_s[BLOCK_SIZE];
buffer_s[tx.x] = input[tx.x];
__syncthreads();

for (unsigned stride = 1; stride < BLOCK_SIZE; stride *= 2) {
```

```

    if (tx.x >= stride) {
        buffer_s[tx.x] += buffer_s[tx.x - stride]; // this line has issues
    }
    __syncthreads();
}

```

- There are synchronization issues with this specific implementation—we can have different threads R/W the same location at the same time.
- We can separate out read/write operations with a `__syncthreads()`, but this enforces a false data dependency (WAR)
- The better solution is to double buffer:

```

__shared__ float buffer_s1[BLOCK_SIZE];
__shared__ float buffer_s2[BLOCK_SIZE];

float* ptr_buffer_s1 = buffer_s1;
float* ptr_buffer_s2 = buffer_s2;

// use ptr_buffer_s instead of buffer_s, and swap them on each reduction step

```

- As aforementioned, Kogge-Stone is low latency, but not work efficient. It does $\log(N)$ steps, and $O(N)$ operations per step. Thus, it does $O(N \log(N))$ operations, which is actually worse than the sequential scan.

2.2.2 Brent-Kung Scan

- Has a higher latency (more steps) than Kogge-Stone, but is work efficient.
- Each thread loads **two** elements.
- Relies on balanced trees (not to be confused with balanced trees from CS225 lmao)
- Idea: traverse from leaves to root building partial sums at particular internal nodes. Then have a post-scan step where you traverse back up the tree and complete the full scan.

```

__shared__ float T[2 * BLOCK_SIZE];
// omitted some initialization code

// REDUCTION STEP
int stride = 1;
while (stride < 2 * BLOCK_SIZE) {
    __syncthreads();
    int index = 2 * stride * (tx.x + 1) - 1;
    if (index < 2 * BLOCK_SIZE && (index - stride) >= 0) {
        T[index] += T[index - stride];
    }

    stride *= 2;
}

```

```

// POST-SCAN STEP
stride = BLOCK_SIZE/2;
while (stride > 0) {
    __syncthreads();
    int index = 2 * stride * (tx.x + 1) - 1;
    if ((index + stride) < 2*BLOCK_SIZE) {
        T[index + stride] = T[index];
    }

    stride /= 2;
}

```

- In contrast to Kogge-Stone, Brent-Kung is $O(N)$ work, so its work efficient. But it takes twice the number of steps, and uses half the number of threads.

2.2.3 Hierarchical Scan

- If the input array is too large, each block can only compute a segment of the overall input.
- To consolidate these segments, we use a three-kernel approach:
- Kernel 1: Performs scan operation on each block, stores partial sums of each block into a separate array.
- Kernel 2: Performs scan operation on the block of partial sums.
- Kernel 3: Adds partial sum i to all elements in scan block $i + 1$ (output of Kernel 1)

2.3 Histograms and Atomics

- Histogramming is a method of extracting features/patters from data. For each element in a set, you identify its corresponding “bin” and increment it.
- If you try to do a naive implementation (without locks/atomics), you can potentially have multiple threads accessing the same memory location (and at least one of which can write)
- To avoid data races, concurrent read-modify-write operations need to be made mutually exclusive. CPU’s often implement this using locks (mutexes), but this is a terrible idea for SIMD execution models.
- Atomic operations are a single ISA instruction which guarantee exclusivity without the risk of deadlock which comes with locks. Atomicity is enforced by microarchitecture.
- Example histogramming code:

```

// setup code omitted (trivial)

unsigned char b = input[idx];
atomicAdd(&bins[b], 1); // this is built in to CUDA and the ISA

```

- An optimized histogram kernel has coalesced memory access patterns:

```

// better histogram code
int idx = tx.x + bx.x * bd.x;
int stride = bd.x * gd.x; // total number of threads

while (i < size) {
    unsigned char b = input[i];
    atomicAdd(&histogram[b], 1);
    i += stride;
}

```

- Atomic operations on global memory have terrible latency.
- Throughput is the rate at which an application can execute an atomic operation on a particular location. If many threads attempt to do atomic operations on the same address, the memory bandwidth is reduced significantly. You have to pray that values propagate to L2 cache.
- Better approach is privatizing the histogram (making a local copy in shared memory)
- Similar to previous implementation, except you're writing to private histogram. You also have to write back to output:

```

// after writing to private histogram
__syncthreads();

if (tx.x < 256) {
    atomicAdd(&histogram[tx.x], private_histogram[tx.x]);
}

```

2.4 Sparse Matrices & SpMV

- Often times we need to do operations with sparse matrices (mostly/significant amount of zeros)
- **SpMV - Sparse Matrix-Vector multiplication**
- Spatially and computationally inefficient to use regular matmul algorithms for sparse matrices
- There are multiple representations for sparse matrices, each of which is viable in different scenarios.

2.4.1 COO (Coordinate) Format

- The most basic sparse matrix representation format—each non-zero value is stored in an array, 1-1 map between value and its respective row and column.
- Parallel SpMV strategy involves assigning a thread to each non-zero value, each thread needs to **atomicAdd** its respective product to the output.

```

unsigned i = threadIdx.x + blockIdx.x * blockDim.x;
if (i < numNonZero) {
    unsigned row = rows[i];
    unsigned col = cols[i];

```

```

    float val = input_matrix[i] * input_vector[col];
    atomicAdd(&out[row], val);
}

```

- This works best for *really* sparse matrices.
- Non-zero elements can be placed in any order, memory accesses to input are coalesced, no control divergence.
- Atomic operations significantly limit the performance of COO SpMV at scale, and it's not terribly space/work efficient.

2.4.2 CSR (Compressed Sparse Row) Format

- Idea: 1-1 map between values and their respective columns, but keep values in the same row adjacent to each other, and have an array of row pointers which tell you the start of each row in the column/value array.
- Parallel SpMV strategy involves assigning a thread to each row pointer, and update the corresponding output element (eliminates the need for atomic operations)

```

int r = threadIdx.x + blockIdx.x * blockDim.x;
if (r < nrows) {
    float val = 0.0f;

    int start = rows[r];
    int end = rows[r + 1];

    for (int i = start; i < end; i++) {
        val += data[i] * input_vector[cols[i]]
    }

    out[r] = val;
}

```

- This is more space efficient than COO, and avoids atomic operations, so it scales better.
- Has control divergence and uncoalesced memory accesses to the input matrix.
- Hard to add new elements to the matrix (you'd have to put it in the right block within the data/column arrays, and then modify row pointers accordingly)

2.4.2.1 CSC (Compressed Sparse Column) Format

- This exists, and is kinda similar to CSR, but also not really.
- Has the same disadvantages as CSR, but also needs atomics to write to output vector.
- Not widely used.

2.4.3 ELLPACK Format

- Similar to CSR, you group non-zero values and their respective columns by row. Pad each row so its the same size, and store this padded array in column major order.
- Parallel SpMV strategy involves assigning a thread to each row, iterating through it (just like CSR)

```
int r = threadIdx.x + blockIdx.x * blockDim.x;

if (r < nrows) {
    float val = 0.0f;

    for (int i = 0; i < num_elems; i++) {
        val += data[nrows * i + r] * input_vector[cols[nrows * i + r]];
    }

    out[r] = val;
}
```

- Has memory coalescing of input matrix, and its easier to add new non-zero elements.
- Also has control divergence, reduced space efficiency due to the extra padding.
- If you have several “exceptional” rows with a lot of elements, you can use **hybrid ELL + COO**, where the longest rows of the padded matrix are represented with COO.
- hybrid ELL + COO allows us to reduce the padding (spatial overhead) and have less control divergence.
- ELL is great for roughly random sparse matrices, may need hybrid ELL + COO when you have high variance with row size.

2.4.4 JDS (Jagged Diagonal Storage) Format

- Group non-zero elements by row (CSR), then sort these rows by size (keeping track of which row maps to where)
- Like ELL, we can transpose this (column-major format) in order to enable better memory coalescing
- Allows control divergence to be easily scalable
- Implementation of JDS with transposition is below:

```
int r = threadIdx.x + blockIdx.x * blockDim.x;

if (r < nrows) {
    float val = 0.0f;
    unsigned int sec = 0;

    while (jds_col_t_ptr[sec + 1] - jds_col_t_ptr[sec] > r) {
        val += data[jds_col_t_ptr[sec] + r] *
input_vector[cols[jds_col_t_ptr[sec] + r]];
```

```

    sec++;
}

out[jds_row_permutations_t[r]] = val;
}

```

- Has good memory coalescing & reduced control divergence.
- In general JDS is best for roughly triangular matrices.

2.5 GPU Architecture, cont.

- **PCI** - Peripheral Component Interconnect, originally 33 Mhz (slow), 32-bit wide, used memory-mapped I/O
- **PCIe** - switched, point-to-point connection. Each gen of PCIe doubles bandwidth and power efficiency.
- First gen. of PCIe had bandwidth of 2 GT/s
- Uses bit encoding scheme (8b/10b in PCIe 1, 128b/130 in PCIe3)
- Data is transferred over PCIe via a DMA (Direct-Memory Access), needs physical address for source and destination. **Needs pinned memory**
- If memory is pageable, OS could page out data being R/W to → page fault happens → CPU takes time to page in the data (slow because you're reading from SSD/Disk).
- NVLink - GPU-to-GPU interconnect with a higher bandwidth than PCIe.

2.5.1 Guest Lecture, Tensor Cores

- Specialized hardware unit specifically built for matrix-multiplication operations (AB + C)
- Each thread loads 2 values from shared memory to register file, routed to ALU (multiplier/adders) by a hardware routing unit.
- wmma exists for using these tensor cores
- multiple reasons for increasing performance for inferencing: smaller representations, complex instructions, model efficiency, shrinking process nodes, etc.
- Google TPUs have similar hardware (matmul-specific hardware, interfacing with systolic array data setup)
- Intel AMX allows cuda-like instructions in x86 processors.

2.5.2 Dynamic Parallelism

- idk if this really falls within the category of gpu architecture, but idc
- Dynamic Parallelism refers to being able to launch a kernel from within another kernel
- useful for load balancing, recursion, library calls
- Syntax: `kernel<<<dimGrid, dimBlock, Ns, S>>>([args]);`, where Ns is the amount of shared memory per thread block for this call, S is the associated `cudaStream_t` object.

- Allows for non-uniform workloads, helps avoid kernel launch overheads

3 Transformers

Note: This section is for the GPT project - this is less relevant for the CNN project. This also isn't an AI class per se, but some of these topics are fair game for demo questions. (**update: nevermind! this is apparently fair game for MT2 as well, FML!**)

3.1 High-Level Overview of Transformers

From a high level, transformers:

- Require supervised training
- Have text inputs translated into numerical representations (tokens)
- These tokens are converted into vectors in some n-dim vector space and during training, the transformer derives meaning by settling on embeddings in which direction and semantic meaning are related.
- Number of tokens (vectors) which get passed through the transformer block → **context size**
- The output of the transformer would be a probability distribution, which aggregates in **one embedding**,
 - We only use the final embedding to calculate this probability distribution because it is more efficient to use each vector in the final layer to make predictions on the subsequent vector.
 - The final probability distribution is the result of $\text{softmax}(W_U \cdot V_F)$ where V_F is the final embedding vector, and W_U is the unembedding matrix.
 - W_U has dimensions correlating to the number of words, and the number of words in the embedding dimension.
 - The resulting unnormalized (pre-softmax) parameters of $W_U \cdot V_F$ are called **logits**.

3.2 Attention Mechanism

Begins with text being converted into embeddings (which contain non-contextual meaning & position in the text)

An attention head encapsulates a particular relationship between data (e.g “adjectives updating nouns”)

- The transformer architecture uses a multi-head attention mechanism
- Queries within the attention head are vectors, which are determined by $W_Q \cdot E_n$, where E_n is the embedding. W_Q is determined by training, and encapsulates the nature of the query (e.g “adjectives preceding nouns”)
- Each embedding is also multiplied by W_K , called the key matrix, which can be thought of as the answer to the query.

- We take the dot product of key and query vectors to see our relationships arise from the text
- Attention should be **causal**, so we remove the influence of future words by masking them (set them to negative INF)

At this point we have a matrix of key-query dot products, each product can be thought of conceptually as a ‘score’ which represents how well one word updates other words. We then normalize using the softmax function.

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{K^T \cdot Q}{\sqrt{d_k}} \right) V \quad (2)$$

Above equation is from “Attention Ss All You Need.” d_k is an extra factor which is added to preserve numerical stability.

The V vector(s) are generated by the result of $W_V \cdot E_n$. By multiplying these V vectors by the weights generated from the previous step, we generate a list of “changes” to the original embedding vectors—reflecting the relationship(s) of that particular head of attention.

The transformer architecture is multi-headed, and inherently parallelized. This allows for the transformer to “learn” many distinct relationships between context and meaning.

Multiple attention blocks and in-between operations allow the transformer to determine more nuanced and abstract ideas about a given input. After each self-attention block is finished, the “updated” embeddings are concatenated, and merged via projection by another weight matrix W_O .

3.3 Feed Forward Network (FFN)

In the transformer architecture, the FFN is placed right after the self-attention mechanism, and is responsible for introducing non-linearity into the system (key component of machine learning).

3.3.1 GeLU Activation

Better than ReLU in many aspects because its differentiable at the origin, and smoother.

$$\text{GELu}(x) = x \cdot \Phi(x) \quad (3)$$

where $\Phi(x)$ is the CDF of the Gaussian Distribution. When implemented, we use an approximation to minimize compute resources.

Other components are relatively straightforward

3.3.2 (Brief) Overview of Noteable Optimizations

- `__restrict__` - compiler doesn't know if pointers alias or not, so it needs to be conservative (leading to potentially redundant load/stores).
- ex:

```
// compiler will do 4 loads here. If we use __restrict__, it will optimize this
down to like 2
C[i] = A[i] * B[i];
C[i] += A[i] * B[i];
```

- Joint Register and Shared Memory Tiling - registers have lower latency than shared memory, more efficient memory access pattern
- cuBLAS/Tensor Cores - both optimizations use tensor cores so I'll group them together.
- Reduction trees for Softmax - softmax() uses the max operation which is associative and commutative, so you can use reduction trees to optimize this.
- Flash Attention - Tiles across Q/K/V, the full matrix is never fully computed/materialized (which is where majority of the performance increase comes from as opposed to shared memory tiling), also eliminates some kernel launch overhead.
- Local Attention - Smaller window of nearby tokens, less computational complexity, but result isn't as good.
- Split-K - parallelizing matmul operations by splitting it along the K dimension, leads to better hardware utilization.
- KV-Cache - avoid redundant computations by cacheing the result of each token, reuse it for subsequent tokens in a sequence.

...and that's all for ECE 408.