ECE 408

Fall 2025

Review Notes for ECE 408 (Applied Parallel Programming)

This is in no way comprehensive.

Aniketh Tarikonda (aniketh8@illinois.edu)

Contents

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.

- $P \propto CfV^2$ 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 inconvienient
- we'd like to avoid communication between threads as much as possible

1.1.2 Amdahl's Law

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

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 reult 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 (∼100s 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 - Labs 2 & 3

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 GB/s)/(4B/FLOP) = 37.5 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 procede 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 GB/s)/(4B/FLOP) × 16 = 600 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 → $2 \times 256 \times 4$B = 2kB 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 (∼1k)
- DRAM bank - A 2D array of DRAM cells w/ sense amps for higher speed/reading tiny currents
- Row Address → Row Decoder → DRAM Array → Sense Amps → 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 → 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 (Lab 4)

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 operaiton, resulting in branch divergence. We also nede 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 large caches)
- **Strategy 2**
    - Block size covers input tile (input tile size = output tile size + 2 × mask radius)
    - Load input tile in one step
    - Some inactive threads when calculating output

2 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.

2.1 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**.

2.2 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.

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

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.

2.3 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).

2.3.1 GeLU Activation

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

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

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

2.3.2 Residual Correction

- Adds the input to the output of a sub-layer (FFN, self-attention, etc.)
- When implemented in CUDA it reduces down to a vec-add kernel (trivial).
- Helps preserve information & prevents issues with vaninishing gradients during backprop

2.3.3 Layer Normalization

3 Midterm 2

Will update this later as midterm 2 rolls around :D