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 $\rightarrow$ context switching becomes inconvienient
- we'd like to avoid communication between threads as much as possible

1.1.2 Amdahl's Law

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

Effectively, the maximum speedup $\left(\frac{t_{\text{sequential}}}{t_{\text{parallel}}}\right)$ 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 ($\sim$1 cycle), but few. Main memory is slow ($\sim$100s of cycles), but huge (GBs or more)

Each Thread can:
- R/W per-thread registers ($\sim$1 cycle)
- R/W per-block shared memory ($\sim$5 cycles)
- R/W per-grid global memory ($\sim$500 cycles, but there are L2/L1 caches which can reduce this)
- Read-only per-grid constant memory ($\sim$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 $\rightarrow$ waiting for threads to catch up $\rightarrow$ 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) $\times$ 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 $16 \rightarrow 256$ threads/block $\rightarrow 2 \times 256 \times 4B = 2kB$ of shared memory/ block $\rightarrow$ upper limit of 32 active blocks
- However, there is a maximum of 2048 threads/SM, which inherently limits number of blocks to 8.

2 Midterm 2
Will update this later…