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

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

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.