COSC 407 Midterm Study Guide

Topic 11 – Speed and Efficiency

Overview

This topic introduces key performance metrics in parallel computing: **speedup**, **efficiency**, and how they are affected by factors such as **parallelism**, **overhead**, and **problem size**. It also covers two important theoretical models: **Amdahl's Law** and **Gustafson's Law**.

Performance Metrics

- **Response Time (Execution Time):** Time to complete a single task (in seconds).
- **Throughput:** Number of tasks completed per unit time (e.g. transactions per second).
- **CPU Time Breakdown:**
  - *User time:* Time the CPU spends running user code.
  - *System time:* Time spent running OS services.
  - *Wait time:* Time waiting on I/O or other processes.

Instruction-Level Metrics

- **IPS (Instructions Per Second):** Approximate speed of CPU execution.
- **MIPS / MFLOPS / GFLOPS:** Measure specific types of operations (especially useful in scientific computing).
- **CPI (Cycles Per Instruction):**

  ```
  CPU Time = (CPI × Instruction Count) / Clock Rate
  ```

Benchmarks

- Standardized programs used to evaluate performance.
- **SPEC** benchmark suite: widely used, realistic workloads.
- Website: http://www.spec.org

Overhead in Parallelism

- **Overhead includes:**
  - Thread creation/destruction
  - Synchronization
  - Communication
  - Waiting due to load imbalance or mutual exclusion

Speedup and Efficiency

- **Speedup (S):**

  ```
  S = T_serial / T_parallel
  ```

  - Ideal max speedup = p (number of cores).
  - *Linear speedup* is rare in practice.

- **Efficiency (E):**

  ```
  E = S / p
  ```

  - Measures how well computational resources are utilized.

Practical Observations

1. Increasing `p` → S increases (nonlinearly), E decreases (due to overhead).
2. Increasing problem size → both S and E improve (less relative overhead).

Amdahl's Law

- Focus: **fixed problem size**
- Speedup is bounded by the serial portion of the program.

  ```
  S = 1 / [(1 - r) + r/p]
  ```

  - `r`: fraction of program that is parallelizable.
  - As p → ∞, S → 1 / (1 - r)

Example: If `r = 0.9` (90% parallel), max speedup = 10.

Gustafson's Law

- Focus: **scalable problem size**
- Larger datasets → higher parallel fraction.
  - As r → 100%, S → p
- Ideal for workloads where problem size can grow with hardware.

Analogy: Car Travel

- **Amdahl's:** You already drove slowly for part of the trip — no matter how fast you drive after, average speed is capped.
- **Gustafson's:** Keep driving longer at higher speed → average increases.

Estimating Parallelism

To achieve a desired speedup `S` with `p` cores:

```
r = (1 - 1/S) / (1 - 1/p)
```

Example: Want `S = 10` on 20 cores → need ~94.7% parallelism.

Scalability

- **Strong Scalability:** E remains constant as p increases (problem size fixed).
- **Weak Scalability:** E remains constant as both p and problem size increase.

Summary

- Speedup ≠ efficiency.
- More cores ≠ guaranteed performance gains.
- For small tasks, serial execution may outperform parallel.
- Use parallelism on larger tasks to overcome overhead.

Topic 12 – Intro to CUDA

Overview

This topic introduces **CUDA**, NVIDIA's platform for GPU programming, and compares **CPU vs. GPU** architectures. It explains **latency vs. throughput**, CUDA's hardware model, thread structure, and programming workflow. Examples show how to launch kernels and write simple CUDA programs.

Parallel vs. Serial Execution

- **Serial (CPU):** One thread handles all operations:

  ```
  for(i = 0; i < 100; i++) C[i] = A[i] + B[i];
  ```

- **Parallel (CPU with 4 threads):** Work divided among threads:

  ```
  for(i = 0; i < 25; i++) C[i] = A[i] + B[i]; // Thread 0
  ...
  ```

- **Parallel (GPU with 100 threads):** One thread per data element:

  ```
  C[i] = A[i] + B[i];  // Each thread does one
  ```

Latency vs. Throughput

- **Latency:** Time to complete one task.
- **Throughput:** Number of tasks completed per unit time.

Example Analogy:

- 1 fast worker (low latency) vs. 10 slower workers (higher throughput).
- Depends on whether speed per task or total volume matters more.

CPU vs. GPU Architecture

Feature	CPU	GPU
Control logic	Complex	Simple
Threads	Few	Thousands
Memory bandwidth	Lower	Higher
Latency	Optimized	Higher (hidden)
Use case	Serial work	Parallel work

- GPUs use **SIMD (Single Instruction, Multiple Data)** execution.
- Hundreds/thousands of threads can run concurrently.

CUDA Terminology

- **Host:** The CPU
- **Device:** The GPU
- **Kernel:** Function run on the device (executed in parallel by many threads)
- **Thread/Block/Grid:**
  - Thread: basic unit of execution
  - Block: group of threads (executes on one SM)
  - Grid: collection of blocks

Program Structure

- CUDA programs = **host code (CPU)** + **device code (GPU)**

```
// Host
cudaMalloc();
cudaMemcpy();
kernel<<<blocks, threads>>>();
cudaMemcpy();
cudaFree();
```

```
// Device
__global__ void kernel(...){ ... }
```

Launching Kernels

- Launch format:

  ```
  kernel<<<num_blocks, num_threads>>>(...);
  ```

- Threads organized into **blocks**, blocks into **grids**.

Built-in CUDA Functions

- **cudaMalloc, cudaFree:** Allocate/free memory on device
- **cudaMemcpy:** Copy data between host and device
- **cudaMemset:** Initialize memory
- **cudaDeviceSynchronize():** Wait for device to finish all tasks

---

Function Qualifiers

Qualifier	Runs on	Callable from
global	Device	Host
device	Device	Device
host	Host	Host
host device	Both	Both

Memory Transfers

- **Host to Device:** `cudaMemcpy(d_ptr, h_ptr, size, cudaMemcpyHostToDevice)`
- **Device to Host:** `cudaMemcpy(h_ptr, d_ptr, size, cudaMemcpyDeviceToHost)`
- **Device to Device:** `cudaMemcpy(dst, src, size, cudaMemcpyDeviceToDevice)`

Example: Adding Two Numbers on GPU

```
__global__ void add(int a, int b, int* c) {
    *c = a + b;
}

int main() {
    int *d_c;
    cudaMalloc(&d_c, sizeof(int));
    add<<<1,1>>>(2, 7, d_c);
    int c;
    cudaMemcpy(&c, d_c, sizeof(int), cudaMemcpyDeviceToHost);
    printf("2 + 7 = %d\n", c);
    cudaFree(d_c);
}
```

Example: Vector Addition

- **Serial:**

  ```
  for (int i = 0; i < N; i++) C[i] = A[i] + B[i];
  ```

- **Parallel (GPU):**

  ```
  __global__ void vectorAdd(int* A, int* B, int* C, int N) {
      int i = threadIdx.x;
      if (i < N) C[i] = A[i] + B[i];
  }
  ```

Summary

- CUDA enables high-performance computing on NVIDIA GPUs.
- Use many lightweight threads to handle large data sets.
- Efficient programming requires understanding memory layout, launch configs, and GPU execution model.

Topic 13 – CUDA Threads

Overview

This topic dives into **CUDA's thread hierarchy**, **error handling**, **launch configuration**, and how to compute **array indices** across threads. It also covers memory layouts and introduces the concepts of **grids**, **blocks**, **thread IDs**, and synchronization points.

Error Handling in CUDA

1. **CUDA API Errors:**
    - Functions return `cudaError_t`
    - Check for success:

    ```
    cudaError_t err = cudaMalloc(&d_a, num_bytes);
    if (err != cudaSuccess) {
        printf("Memory allocation failed!\n");
    }
    ```

2. **Kernel Launch Errors:**
    - Use:

    ```
    CHK(cudaGetLastError());
    CHK(cudaDeviceSynchronize());
    ```

3. **Macro for cleaner error checks:**

    ```
    #define CHK(call) { \
        cudaError_t err = call; \
        if (err != cudaSuccess) { \
            printf("Error %d: %s\n", err, cudaGetErrorString(err)); \
            cudaDeviceReset(); \
            exit(1); \
        } \
    }
    ```

---

cudaDeviceSynchronize()

- CUDA and CPU code are asynchronous by default.
- Use `cudaDeviceSynchronize()` to wait until all launched kernels finish.
- Useful for **timing kernel execution**.

Thread Organization

- **Threads → Blocks → Grids**
    - Threads are organized into 1D/2D/3D blocks.
    - Blocks are organized into 1D/2D/3D grids.
- Each thread/block has its own ID:

```
threadIdx.x, threadIdx.y, threadIdx.z
blockIdx.x, blockIdx.y, blockIdx.z
```

- **Dimension Variables:**

```
blockDim.x, gridDim.y, etc.
```

GPU Hardware Model

- **SM (Streaming Multiprocessor):** Runs thread blocks.
- **SP (Streaming Processor):** Executes one thread at a time.
- A GPU has many SMs → many blocks can run in parallel.

Thread Lifecycle (Simplified)

1. Grid is launched
2. Blocks are scheduled to SMs
3. SMs divide blocks into **warps** (groups of 32 threads)
4. Each SP executes one thread in a warp
5. Warps are scheduled in SIMT (Single Instruction, Multiple Threads) model

---

Example: Vector Addition

```
__global__ void vec_add(float *A, float *B, float* C, int N) {
    int i = threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}
```

Computing Global Thread Index

- For 1D block & grid:

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

- For 2D grid:

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

Launch Configuration

- To compute total threads and blocks:

```
int nthreads = 256;
int nblocks = (N + nthreads - 1) / nthreads;
vectorAdd<<<nblocks, nthreads>>>(...);
```

Block & Grid Dimensions

- **Threads in block:**

```
dim3 blockDim(16, 16);
```

- **Blocks in grid:**

```
dim3 gridDim((width+15)/16, (height+15)/16);
```

Thread Indexing for Multidimensional Arrays

- Example for image processing:

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

- Always check bounds!

```
if (x < width && y < height) {
    // safe access
}
```

Summary

- Understand the thread hierarchy: threads → blocks → grids
- Use built-in ID variables to calculate data access indices
- Use synchronization and error checking to write correct GPU code
- Launch configuration must match your data dimensions

Topic 14 – Scheduling and Performance

Overview

This topic covers how CUDA **schedules threads on hardware**, how **warps** are used for execution, and how **shared memory** and **tiling** can improve performance in matrix multiplication. It also details the different **types of memory** and how to utilize them efficiently.

Thread Scheduling & Warps

- **Thread blocks** are assigned to **SMs (Streaming Multiprocessors)**.
- Each SM divides threads into **warps** (typically 32 threads).
- All threads in a warp execute the **same instruction** (SIMT model).
- **Warp scheduling**:
  - Warps ready to execute are selected.
  - If a warp stalls (e.g. waiting on memory), another warp can be scheduled.

Warp Size: Usually 32 threads
Execution: All threads in a warp run together, no guaranteed order across warps.

Scalability & Zero-Overhead Scheduling

- **Transparent scalability:** GPU handles scheduling without user involvement.
- Each block can execute on **any available SM**.
- **Zero-overhead scheduling:** SMs switch between warps with minimal delay.
- **Latency hiding:** When one warp waits, others execute → better utilization.

GPU Hardware & Limits

Architecture	SMs	Threads/SM	Max Threads
G80	16	768	12,288
G200	30	1024	30,720

- CUDA imposes **software limits** (e.g., 1024 threads per block).
- GPUs also have **hardware limits** on active threads/blocks per SM.

Matrix Multiplication: One Block

- Threads in a **single block** compute a matrix `P = M × N`
- Each thread computes one element in result `P`.
- Size limited to **32x32**, i.e. 1024 threads max per block.

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int width) {
    int r = threadIdx.y;
    int c = threadIdx.x;
    if (r < width && c < width) {
        float value = 0;
        for (int k = 0; k < width; k++)
            value += d_M[r*width + k] * d_N[k*width + c];
        d_P[r*width + c] = value;
    }
}
```

Matrix Multiplication: Multiple Blocks

- Break the work into **tiles** and assign to multiple blocks.
- Each block handles a **TILE_WIDTH × TILE_WIDTH** chunk.
- Coordinates use both `blockIdx` and `threadIdx`.

```
int r = blockIdx.y * TILE_WIDTH + threadIdx.y;
int c = blockIdx.x * TILE_WIDTH + threadIdx.x;
```

Using Shared Memory (Tiling)

- Load tile data into **shared memory** to reduce global memory reads.
- All threads in a block share tiles of M and N.

```
__shared__ float Ms[TILE_WIDTH][TILE_WIDTH];
__shared__ float Ns[TILE_WIDTH][TILE_WIDTH];
```

- Threads perform computation on shared data.

Execution Configuration

```
dim3 blockSize(TILE_WIDTH, TILE_WIDTH);
dim3 gridSize((width+TILE_WIDTH-1)/TILE_WIDTH, (height+TILE_WIDTH-1)/TILE_WIDTH);
MatrixMul<<<gridSize, blockSize>>>(...);
```

Memory Types in CUDA

Type	Scope	Speed	Notes
Registers	Thread	Very Fast	Private
Shared	Block	Fast	Shared across threads in a block

Type	Scope	Speed	Notes
Global	All grids	Slow	Accessible by all threads
Constant	All grids	Fast (cached)	Read-only, limited size (64 KB)
Local	Thread	Slow (cached)	Used when registers spill

Summary

- Threads are scheduled as **warps** for efficiency.
- **Latency hiding** ensures better hardware utilization.
- Use **shared memory and tiling** to reduce global memory access.
- Understand memory types and **limit reliance on global memory** for performance.

Topic 15 – Making Things Faster (Best Practices)

Overview

This topic presents **CUDA best practices** for performance optimization. It covers the **APOD framework**, **memory optimization**, **synchronization**, **atomic operations**, and strategies to reduce **warp divergence**, improve **instruction-level performance**, and efficiently perform **reductions**.

APOD Framework

Assess – Identify performance bottlenecks
Parallelize – Decide what can be parallelized (Amdahl’s/Gustafson’s)
Optimize – Improve memory usage, instruction performance
Deploy – Measure and compare performance

Memory Optimization Guidelines

1. **Minimize Host-Device Transfers**

   - Batch small transfers
   - Keep intermediate structures on the device

2. **Use Fast Memory Types:**

Memory	Speed	Scope
Register	Very fast	Thread
Shared	Fast	Block
Constant	Fast	Grid
Global	Slow	Grid
Local	Slow	Thread

3. **Reduce Global Memory Traffic:**
    - Use **tiling**: load data into shared memory, compute, write back
    - Coalesced access patterns

Tiling and Shared Memory Example

```
__shared__ float shrArr[128];
int idx = threadIdx.x;
shrArr[idx] = arr[idx];
__syncthreads();
// process shrArr[idx]
```

Synchronization

- **Barriers**:
    - `__syncthreads();` synchronizes threads within a block
    - Implicit barrier at the end of a kernel
- **Avoid Data Races**:
    - RAW (Read After Write)
    - WAR (Write After Read)
    - WAW (Write After Write)

Synchronization Example Fix

Problem:

```
array[i] = array[i-1]; // data race!
```

Fix:

```
__syncthreads();
int temp = array[i-1];
__syncthreads();
array[i] = temp;
```

Atomic Operations

Used to ensure **mutual exclusion**:

```
__global__ void increment_atomic(int* x) {
    atomicAdd(x, 1);
}
```

- **Limitations:**
  - Slower (serialized)
  - Not all types supported
  - Use sparingly!

Critical Sections

- Can be implemented using `atomicCAS`:

```
__device__ void lock() {
    while (atomicCAS(mutex, 0, 1) != 0);
    __threadfence();
}
__device__ void unlock() {
    atomicExch(mutex, 0);
    __threadfence();
}
```

Coalesced Global Memory Access

Access pattern matters!

- Best: threads access contiguous memory
- Avoid:
  - Strided access
  - Random access
  - Misaligned blocks

Access Pattern Examples

```
x = A[i];            // Coalesced
x = A[2 * i];        // Strided
x = A[128 - i];      // Strided
A[A[i]] = 7;         // Random
```

Instruction-Level Optimization

- Use **bitwise ops** for divisions by powers of 2:

```
int x = i >> 1; // i / 2
int r = i & 0x1; // i % 2
```

- Prefer faster math:
  - `__expf`, `__logf`, `__sinf` vs. `exp`, `log`, `sin`

Avoid Warp Divergence

- Threads in the same warp taking different branches leads to **serialization**.
- Fix:

```
if (threadIdx.x / WARP_SIZE == warp_id) { ... }
```

- Balance work: give complex tasks to whole warps

Reduction Pattern (Summing an Array)

Naive:

```
for (int stride = 1; stride < blockDim.x; stride *= 2)
  if (i % (2 * stride) == 0)
    partialSum[i] += partialSum[i + stride];
```

Improved (no divergence):

```
for (int stride = blockDim.x / 2; stride > 0; stride /= 2)
  if (i < stride)
    partialSum[i] += partialSum[i + stride];
```

Summary

- Use shared memory, coalesced access, and synchronization correctly
- Avoid warp divergence and unnecessary atomics
- Optimize memory access and arithmetic operations
- Use **tiling** and **in-place reductions** for scalable performance

Topic 16 – Performance (Part 2)

Overview

This topic revisits CUDA **memory performance**, **timing strategies**, **bandwidth estimation**, and the concept of **CGMA (Compute to Global Memory Access ratio)**. It also reinforces the benefits of **tiling** in matrix multiplication and memory access patterns in CUDA.

Measuring Performance

(1) Using CPU Timers

```
double t = clock();
kernel<<<..,..>>>();
cudaDeviceSynchronize();
t = 1000 * (clock() - t) / CLOCKS_PER_SEC;  // milliseconds
```

(2) Using GPU Events

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
kernel<<<..,..>>>();
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);  // milliseconds
```

Measuring Bandwidth (BW)

- **Effective Bandwidth:**

  ```
  BW_effective = (RB + WB) / time
  ```

  - RB = read bytes
  - WB = write bytes
- Compare this to theoretical bandwidth (from hardware specs) to determine efficiency.

CGMA: Compute to Global Memory Access Ratio

- Measures how much **useful computation** you get per memory access.
- Low CGMA means memory bottleneck.
- Aim for **high CGMA** to maximize FLOPS.

Example:

- Matrix multiplication with no tiling: CGMA = 1 → underutilizes GPU.

Example: Matrix Multiplication

Without tiling:

```
for (int k = 0; k < width; k++)
  value += M[y * width + k] * N[k * width + x];
```

Each value in M/N is accessed multiple times → low CGMA.

Improving CGMA with Tiling

Tiling Strategy:

- Divide work into TILE_WIDTH × TILE_WIDTH sub-matrices.
- Load each tile into **shared memory**:

  ```
  __shared__ float Ms[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Ns[TILE_WIDTH][TILE_WIDTH];
  ```

- Each thread computes its portion using:

  ```
  for (int k = 0; k < TILE_WIDTH; k++)
    value += Ms[ty][k] * Ns[k][tx];
  ```

CGMA with Tiling

With TILE_WIDTH = 16:

- 256 threads × 16 iterations × 2 FLOP = 8192 FLOPs
- 256 threads load 2 elements each = 512 global loads
- **CGMA = 8192 / 512 = 16**
- Drastically improves utilization

Execution Configuration

```
dim3 blockSize(TILE_WIDTH, TILE_WIDTH);
dim3 gridSize((Width + TILE_WIDTH - 1) / TILE_WIDTH,
              (Width + TILE_WIDTH - 1) / TILE_WIDTH);
MatrixMul<<<gridSize, blockSize>>>(...);
```

Memory Location Summary

Variable Type	Memory	Lifetime	Speed
`int x`	Register	Thread	Very fast
`int arr[10]`	Local	Thread	Slow
`__shared__`	Shared	Block	Fast
`__device__`	Global	Application	Slow
`__constant__`	Constant	Application	Fast (cached)

L1 and L2 Caches

- L1: Per SM, fast but **not coherent**
- L2: Shared across GPU, coherent
- Reads use both caches; writes go through L2 only

Constant Memory

- Limited (64 KB), cached, read-only from device
- Great for parameters or lookup tables

Local Memory

- Private to thread
- Physically located in global memory (but cached)
- Used when registers are insufficient

Image Processing with CUDA

- CUDA is well-suited for:
  - Convolution (e.g., blur, edge detection)
  - Histogram operations
  - Noise reduction
  - Compression, correlation

Each pixel can be mapped to one thread.

Summary

- Use **shared memory** and **tiling** to reduce global memory access
- Measure **effective bandwidth** and maximize **CGMA**

- Use **GPU timers** for accurate profiling
- Understand memory hierarchy for performance tuning
- Carefully choose thread/block/grid dimensions