Performance Metrics

- **Response Time** The time taking to complete one task
- **Throughput** is the number of task completed per unit of time
- **CPU Time breakdown**
  - User time: Time the CPU spends running the users code, System time: Time CPU spent running the OS's code, Wait time: Time spent waiting for I/O or other services

Instruction-Level Metrics

- **IPS (Instructions Per Second):** Approximate speed of CPU execution.
- **CPI (Cycles Per Instruction):** CPU Time = (CPI × Instruction Count) / Clock Rate

Overhead in Parallelism

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

Speedup and Efficiency

formulas:

```
Speedup = T_serial / T_parallel
Efficiency = E = S / p      // p is the number
```

As p increases then E decreases due to overhead. If the problem size increases then both Speedup (S) and Efficiency (E) increase. due to less overhead

Amdahl's Law

S is the max speedup, r is the percentage/fraction of the program that is parallelizable, p is the number of cores. r/p is also the parallel speedup

```
S = 1 / ((1 - r) + (r/p))
S = 1 / ((1 - r) + r) // as p approaches infinity
```

Gustafson's law

This formula is for scalable/large problem sizes. If 'r' or the parallelizable portion is 100% then S = P **Strong scalability** if E remains constant as p increases (that means the problem size is fixed). **Weak scalability** if E remains constant as both p and problem size increase.

CUDA

Latency is the time taken to complete one task. **Throughput** is the number of tasks completed per unit of time.

CPU vs. GPU Architecture

(Feature, CPU, GPU), (Control logic, Complex, Simple), (Threads, Few, Thousand), (Memory bandwidth, Lower, higher), (Latency, Optimized, Higher), (User Case, Serial Work, Parallel Work) GPU uses SIMD (single instruction multiple data). That's why GPU's are optimized for parallelism. Host is the CPU, Device is the GPU, Kernel is the function run on the device (executed in parallel by many threads). A grid is a collection of blocks, a block is a collection of threads

Program Structure

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

Function Qualifiers

(Qualifier, Runs on, Callable from), (**global**, Device, Host), (**device**, Device, Device), (**host**, Host, Host), (**host device**, Both, Both)

```
__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); //num blocks, num threads
    int c;
    cudaMemcpy(&c, d_c, sizeof(int), cudaMemcpyDeviceToHost);
    printf("2 + 7 = %d\n", c);
    cudaFree(d_c);
}
```

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 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.
  ```

```
__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;
  // 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 fimension
  dim3 blockDim(16, 16);
  //blocks in grid
  dim3 gridDim((width+15)/16, (height+15)/16);
  //thread indexing
  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
  }
  ```

Thread Scheduling & Warps

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;
```

- 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), (**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)

APOD Framework

Assess identify performance bottlenecks, **P**arallelize decide what can be parallelized (Amdahl's/Gustafson's), **O**ptimize Improve memory usage, instruction performance, **D**eploy 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:**
   - (Register, Thread), (Shared, Fast), (Constant, Grid), (Global, Slow), (Local, Slow)

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 Example Fix

Problem:

```
array[i] = array[i-1]; // data race!
//fix:
__syncthreads();
int temp = array[i-1];
__syncthreads();
array[i] = temp;
//atomic sections
__global__ void increment_atomic(int* x) {
    atomicAdd(x, 1);
}
//critical sections
__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
  // faster math: `__expf`, `__logf`, `__sinf` vs. `exp`, `log`, `sin`
  ```

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

Memory Location Summary

(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