HPC

__/__/____

* Warps, SIMD Hardware (Thread Execution Efficiency)

1) Cuda me threads block me hote hai.
2) Har block ko 32 thread ke groups me divide kiya jaata h.
3) 32 thread group is called warp.

Warp = Scheduled unit of SM (Streaming Multiprocessor)

4) Every warp has 32 threads.
5) Not a part of CUDA programming model, hardware implementation detail hai.
6) Future ~~warps~~ GPU's me warp size change ho SKta hai.
7) Warp ke andar threads SIMD model me execute hote hai. → Simple Instruction, Multiple data

* Multidimensional Thread Blocks

1) 2D/3D blocks ko pehle 1D linear order me convert ho kiya jaata hai
2) Order is :-
   X dimension → Y dimension → Z dimension
3) Warps are groups of linearized threads.

* Warp as Scheduling Unit

1) GPU ka SM (Streaming Multiprocessor) warp level par schedule karta hai.
2) SM at a time executes warps not threads solo

__/__/____

* Warp Rules and Synchronization

1.) Warp ke andar ya warp ke beech order par depend mt karo.
2.) Agar threads ke beech dependency ho → syncthreads() zaroori hai.
3.) " is also compulsory when shared memory is used.

* SM = SIMD Processor

1.) SM ke andar :- Have registers + shared memory
- Control unit common hota hai.
- Multiple ALU's

2.) Benefits :-
- Instruction fetch and decode ek baar hota hai
- Control overhead kam hota hai.

* Control Divergence (Performance Impact)

1.) When same warp ke threads different execution paths follow krte hai its known as control divergence.
2.) Control divergence slows why :-
- Warp can run 1 instruction at a time
- GPU pehle if path execute krega
- fir else ke saath execute krega

Threads jo us path me nahi hi, idle rehte hai

```
if (threadId.x < 16)
```

__/__/____

3.) Reasons of Divergence → if-else based on threadIdx
- Loop jisme → Some threads less iteration
  → Some threads more iteration

if (threadIdx.x > 2)
- Example of divergence
- Diff decision inside warp
- Divergence occurs (bad)

if (blockIdx.x > 2)
- Example of non divergence
- Block level decision
- All threads of the warp follows same path.

- Control Divergence → Boundary checks are imp.
  → Impact is negligible in large datasets.

Control flow zyada hone ka mean slow nahi hota hmesha.

* DRAM Bandwidth (Memory Access Performance)

1.) Importance of Memory Bandwidth
- GPU → Computes very fast
  → But if data is slow GPU will be idle

Thats why memory BW = first order performance factor.

2.) DRAM Internals
- Dram cell = capacitor
- To Capacitor me charge detect krna slow hota hai
- Core array speed << interface speed

Eg :- DDR3 → core speed = 1/8 interface speed.

__/__/____

* DRAM Bursting Concept

1.) DRAM stands for Dynamic random access memory.
2.) DRAM ek address read krta hai
3.) Poora row internal buffer me load hota hai
4.) Data burst mode me transfer hota hai

Sequential access = full benefit
Random access = data waste

* DRAM Banks

1.) Multiple banks = parallel row access
2.) Bank switching se latency hidi hoti hai

* GPU Memory Channels

1.) Eg :-
- → RTX 6000,
- → Required bw = 672 GB/s
- → Single channel ≈ 56 GB/s
- → Thats why 12 channels are used

* Memory Coalescing

Def 1.) Jab warp ke saare threads :-
- Same instruction
- Adjacent memory locations access karo.

Coalesed Access Pattern → GPU sirf ek DRAM request krta hai.

A[base + threadIdx.x] → Fast, BW efficient

__/__/____

2.) Uncoalesced Access
- Threads scattered addresses access krte hain
- Multiple DRAM requests
- Slow execution

3.) Matrix Multiplication Case
- B Matrix access → coalesced
- A " " → uncoalesced

Solution → Tiling, Shared Memory, Corner Turning

* Histogram

1.) Data ko bins me count krna is histogram
- Example → Alphabet frequency
  → Img intensity count

2.) Parallel histogram challenge
- Multiple threads
- Same bin update
- Memory Access Ineffecient

3.) Sectioned Partitioning (Bad)
- Each thread continuous chunk leta hai
- Adjacent threads → far memory
- No coalescing

4.) Interleaved Partitioning (Good)
- Threads alternate data access
- Coalesced memory, Better bandwidth (BW)

A bin is a memory location that stores the count of data elements falling into a specific range or category

__/__/____

* Data Races (DR)

1) DR are :- When Read-Modify-Write
- Multiple threads
- Same memory location
- Read-Modify-write
- without synchronization

Eg :- `x = x + 1;`

2 → threads

→ Output is unpredictable. result kabhi 1 kabhi 2

2) Real life eg :- Bank cash counting
Ticket booking, customer numbering system

* Atomic Operations (AO)

1) AO are :-
- Single hardware instruction
- Read + Modify + Write together
- No interference allowed

2) CUDA Atomic functions
- atomicAdd, atomicMin/Max
- atomicSub, atomicCAS

3) Histogram with Atomics
- Threads input ko stride pattern me process krte hai.
- Histogram update atomicAdd se hota hai
- Data race eliminate ho jata hai.

__/__/____

* Kernel and SPMD parallelism

1.) Kernel = GPU par run hone wala fxn.
CPU se launch hota h, but parallel threads GPU par execute krte hai.

2.) SPMD = Single Program Multiple Data, means same code (kernel) but different data for every thread
Eg :- Img ke har pixel ke liye same kernel
Har thread ko ek pixel process krta hai

Imp: Har threads ka unique ID hota hai
Same kernel sab threads ke liye run hota hai
GPU massive parallelism achieve krta hai

- CPU follows SPMD model where the same kernel program is executed by multiple threads on diff data elements.

* Multidimensional Kernels

MD Kernels are used to naturally map threads to 2D or 3D like images and matrices

1.) 1D vs 2D vs 3D grids.
↓ ↳ ↳ → Volume Data
Vector Image/
Matrix

x = threadIdx.x + blockIdx.x * blockDim.x
y = threadIdx.y + blockIdx.y * blockDim.y

Isse har thread apna pixel (x,y) find krta hai

2.) Why imp :- Img are naturally 2D
Matrix multiplication also uses 2D mapping

__/__/____

* Color to Grayscale (Img Processing)

1) RGB img → Har pixel = (R, G, B)
   → Values range = 0-255

2) Grayscale img → Sirf intensity value
   → Color info. remove ho jata hai

Formula Gray = 0.21R + 0.71G + 0.07B (Green is max bcz human eye is more sensitive to green)

CUDA Approach
↳ Har thread = 1 pixel
RGB read → grayscale write

RGB to grayscale conversion is done using a weighted sum to preserve percieved brightness

* Blur Kernel (Image Blurring)

1) Pixel ko neighbours ke avg se replace krna is Image blurring, img smooth ho jati hai
2) Blur box → Center pixel ke aas pass ka square region
   → Eg:- 3x3, 5x5 window.
3) CUDA working → Har thread = 1 pixel
   → Neighbour pixels read
   → Avg calculate, Output img me store

if (row < height && col < width)

Image blurring uses a 2D kernel where each thread computes the avg. of neighbouring pixels

__/__/____

* Thread Scheduling and Transparent Scalability

1.) Transparent Scalability
- Programmer ko GPU size ka tension nahi
- Block can be executed in any order
- Hardware automatically schedule krta hai

2.) SM (Streaming Multiprocessor)
- Threads block granularity me SM par assign hote h, Ek SM me multiple blocks ho skte hai

3.) Warps (32 threads)
- Scheduling unit
- Warp ke sab threads same instruction execute karte h (SIMD)

4.) Zero Overhead Scheduling
- Jab 1 warp wait krta hai (memory), dusra warp execute ho jata hai, latency hide ho jati hai.

⇒ CUDA provides transparent scalability by allowing blocks to execute in any order across SM's.

* CUDA Memories

	Memory	Scope	Speed
•	Shared	Block	Very fast
•	Register	Thread	fastest
•	Global	Grid	Slow
•	Constant	"	Cached

__/__/____

2) Registers → per thread, automatic variables, fast but limited
Shared Memory
↳ per block, global se kaafi fast, stored b/w threads
Global Memory
↳ sabse slow, high latency, sab threads access kar skte hai

Efficient use of shared memory is critical to improve CUDA kernel performance.

* Tiled Parallel Algorithms

1) Problem → Global memory is slow
↳ Har operation me global access → low performance

2) Tiling Idea → Data ko small tiles me break karo
- Tiles ko shared memory me load karo.
- Baar baar use karo.

Real life analogy → Carpooling = tiling
↳ Zyada log kam gaadi = kam traffic

3) Steps of tiling
- Tile identify kro
- Global → shared load
- __syncthreads()
- Computation
- Next tile

⇒ Tiling reduces global memory access by reusing data from fast on chip shared memory

__/__/____

* Tiled Matrix Multiplication (TMM)

1) Basic MM
- Har thread : 1 output element
- Global memory access is very much

2) Tiled approach
- Matrix ko block width x block width tiles me
- M aur N ke tiles shared memory me load divide
- Phase by phase computation

3) Phases
- Phase 0 :- Tile 0 load + compute
- Phase 1 :- Tile 1 load + compute
- .. repeat

Tiled matrix multiplication improves arithmetic intensity by inc. computation per memory access

* TMM Kernel

1) Shared Memory Usage
```
__shared__ float ds_M (TILE) (TILE);
__shared__ float ds_N (TILE) (TILE);
```

2) Kernel flow :- Thread index calculate
- Tile loaded into shared memory
- __syncthreads ()
- Partial sum calculation
- Next phase
- Final result write

Proper tile size selection balances shared memory usage and thread occupancy.

__/__/____

* Course Intro and Overview

1) Aim of course :-

- Heterogeneous parallel computing systems ke program krna
- Achieve karna :-

→ High performance
Energy efficiency
Maintainability
Scalability (future hardware ke liye)
Portability (diff vendors ke devices)

(EK hi program GPU's/CPU's par bhi acha chle bina dobara likhe)

* Heterogeneous Parallel Computing

HC kya hota h

1) Jb system me diff types ke processors milker kaam krte hai :-

- CPU, GPU, DSP, Specialised hardware
  ↓ L, ~~Global~~ Graphics
  Central Processing Unit

DSP - Digital Signal Processor

2) CPU vs GPU

CPU	GPU
• Latency Oriented	• Throughput Oriented
• Powerful Cores	• Bohot zyada cores
• Large Cache	• Small cache
• Fast response time	• Slow
• Best for sequential code	• Thousands of threads
• Eg:- OS tasks control logic	• Eg:- Img processing Matrix Multiplication

3.) Why use both :- CPU → Sequential part
GPU → Parallel part
Sequential code : CPU 10x faster
Parallel " : GPU " "

4) Applications of Heterogeneous Computing
- Scientific simulation
- Medical Imaging
- Video processing
- Machine Learning
- Financial Analysis

* Portability and Scalability

1.) Software cost is imp :- Software lines faster grow than hardware
- Har naya hardware kar sakte h par code rewrite is costly.

Is liye :- → Scalability
→ Portability

2.) Scalability :- Same app efficiently run kare
- New CPU ~~generators~~ generations par
- Zyada cores par
- " threads par

3.) Achieving Scalability → Fine grained parallelism
H/w details pe depend ← → Dynamic thread scheduling na krna

4) Portability :- Same program efficiently run kare
- CPU, GPU par
- X86, ARM par
- SIMD, threading par
- Shared and distributed memory systems par

__/__/____

* CUDA C vs Thrust vs Libraries

1.) Libraries → Easy, fast
Compiler Directives → Portable, simple
Programming Lang. (CUDA C) → Max performance

2.) Libraries → Ready made GPU made code
Minimal changes required
Eg:→ Thrust, cuBLAS
Jab pattern already available ho → libraries best

3.) Thrust → STL jaisa interference
Easy vector operations, less code, less control

4.) Compiler Directives (OpenACC)
- #pragma based
- Compiler parallelism handle krta hai
- Performance compiler pe depend krta hai

5.) CUDA C → Most powerful, full control, max performance

CUDA C offers max. flexibility and performance at the cost of programming complexity.

* Memory Allocation and Data transfer

1.) CUDA execution model
- Host (CPU) → serial code
- Device (GPU) → parallel kernel

2.) Explicit Memory Management Flow

- Host memory allocate
- Device " "
- Data copy Host → Device
- Kernel launch
- Result copy Device → Host
- Device memory free

Explicit memory management requires manual allocation and data transfer b/w host and device.

* Threads and Kernel functions

1.) CUDA thread model
- Kernel → grid of threads
- Sab threads same code run krte hai (SPMD)

2.) Threads as Virtual Processor
- Har thread ke pass :-

→ Registers, program counter, control logic
Thread = Virtual Von-Neumann processor

3.) Thread Indexing

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

- Har thread apna data element calculate krta h.

4.) Thread Blocks
- Threads ko blocks me divide kiya jaata hai
- Block ke andar :→ Shared memory, synchronization
- Blocks ke beech no communication possible

⇒ Thread blocks enable scalable cooperation among threads using shared memory and synchronization.

__/__/____

* Unified Memory (UM)

1.) UM is a single memory space.
- CPU aur GPU dono access kar sakte hai
- Data migration automatically hota hai

2.) Benefits → Code simple, less bugs, easy learning

3.) Advanced UM
- Compute capability 6.x ke baad :-

→ Page faulting, on demand migration, memory oversubscription possible.

⇒ Unified memory simplifies programming by providing a single address space managed by hardware and software.

* FORMULAS

1.) Warp Size = 32 threads / warp

2.) Warps per block = $\frac{\text{Threads per block}}{\text{Warp Size}}$

3.) Blocks = $\frac{\text{Total elements}}{\text{Threads per block}}$

4.) Threads in grid = Block x Threads per block

5.) Idle threads = Threads in grid − Useful threads

6.) $\text{Block}_{\text{threads}} = \frac{\text{Max threads per SM}}{\text{Threads per block}}$

__/__/____

7.) $\text{Blocks}_{\text{registers}} = \dfrac{\text{Total registers per SM}}{\text{Registers per thread} \times \text{Thread per block}}$

8.) $\text{Blocks}_{\text{shared memory}} = \dfrac{\text{Shared memory per SM}}{\text{" " " block}}$

9.) Blocks per SM = min (6, 4, 8, HW block limit)

10.) $\text{Active warps per SM} = \text{Blocks per SM} \times \dfrac{\text{Threads per block}}{\text{Warp Size}}$

11.) $\text{Active threads per SM} = \text{Active warps per SM} \times \text{Warp Size}$

12.) $\text{Max warps per SM} = \dfrac{\text{Max threads per SM}}{\text{Warp Size}}$

13.) $\text{Occupancy} = \dfrac{\text{Active threads per SM}}{\text{Max " " "}}$

threads ki jagh warps. new formula.

14.) Unutilized warps = Max warps per SM - Active warps

threads = Max threads per SM - " threads

15.) Required warps = $1 + \left(\dfrac{L}{M}\right)$ L: Memory latency

M: useful compute cycles b/w memory access