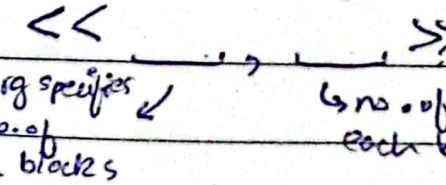
Date _____ Day _____

CHAPTER # 03

▷ CUDA THREAD ORGANIZATION:

→ All threads in a grid execute the same kernel func.
→ They rely on cordinates to distinguish themselves from one & another & identify appropriate portion of data to process
→ All threads in a block share the same block index which is blockIdx
→ Each thread has a thread index which is threadIdx
→ The thread Id & blockId together form a cordinate for a thread which helps to distinguish b/w 2 threads
→ The exact organization of grid is determined by the execution configuration parameter.



→ Each of this parameter is of type dim3
↳ it is a C structure

```
dim3 dimGrid(32, 1, 1)
dim3 dimBlock(128, 1, 1)
```

creates 1D of 32 blocks
each block have 128 thrs
~~total 128×32 = 4096 threads~~
these can be any C valid names.

```
vecAddKernel <<< dimGrid, dimBlock >>> (...);
```

Date _____ Day _____

→ All threads in a block share same blockIdx.x, blockIdx.y, blockIdx.z.
And value for each dimension ranges from
- blockIdx.x → 0 — gridDim.x - 1
- " .y → 0 — gridDim.y - 1
- " .z → 0 — gridDim.z - 1

→ The no. of threads in a block cannot be greater than 1024

→ blockDim (512, 1, 1) = 512 x 1 x 1 = 512 threads

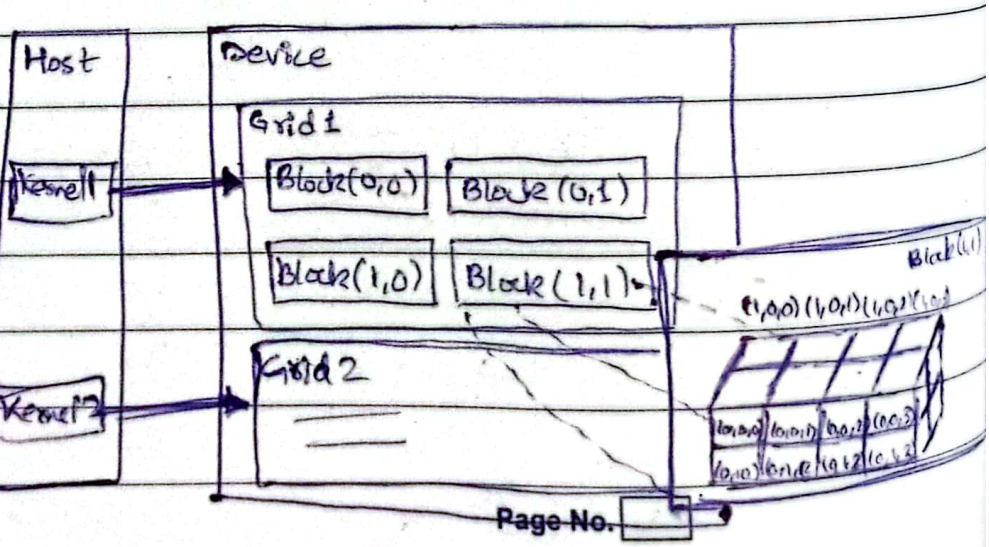
→ blockDim (8, 16, 4) = 8 x 16 x 4 = 512

→ blockDim (32 x 16 x 2) = 1024

→ blockDim (32 x 32 x 2) = 2048 (Not allowed)

→ dim3 dimGrid (2, 2, 1);

→ dim3 dimBlock (4, 2, 2);



Date Day

"GPU"

"Design Trajectories In Microprocessor"

a) Multicore Trajectory:

- Focus: Maintain execution speed of sequential programs while leveraging multiple cores.
- Example: Intel multicore processor
- Characteristics:
  - ↳ Out of order, multiple instr. issue processor
  - ↳ Large cache memories to reduce data access latencies
  - ↳ optimized for sequential code performance

b) Many-Thread Trajectory:

- Focus: Maximize execution throughput for // application.
- Example: NVIDIA Tesla P100 GPU
- Characteristics:
  - Thousands of threads executing in simple, in order pipelines
  - High floating point performance.

Date _____ Day _____

"Fundamental Design Philosophies"

a) CPU (Latency-Oriented Design)

- Purpose: Minimize the execution latency of a single thread.
- Features:
  - ↳ Sophisticated control logic for sequential and out of order execution
  - ↳ Large cache memories to reduce instruction and data access latencies.
  - ↳ Arithmetic units optimized for low-latency operations.

b) GPU (Throughput-Oriented Design)

- Purpose: Maximize total execution throughput of a large no. of threads.
- Features:
  - High memory bandwidth to handle large data volumes.
  - Small cache memories to support high DRAM bandwidth utilization.
  - Designed to tolerate high-latency operations by keeping many threads active.

Date _____ Day _____

"MEMORY BANDWIDTH"

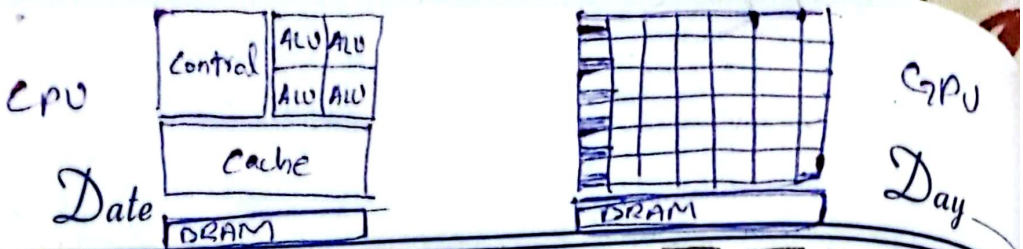
- GPU Advantage:
  - Graphic chips operate at 10x the memory bandwidth of CPUs due to the need for high-speed frame buffer operations.

- CPU Limitations:
  - General-purpose requirements (eg: OS, legacy applications) make bandwidth improvements challenging.

"SOFTWARE DESIGN CONSIDERATION"

- For GPUs:
  - Applications need to be written with many threads
  - Hardware overlaps computation and memory operations to hide latency

- For CPUs:
  - Designed for sequential and low-thread-count workloads
  - Large caches and low-latency operations reduce single-thread execution time.



Date

Day

CPU	GPU
Core	
Fewer no. of larger cores optimized for complex tasks	Thousands of smaller core optimized for parallel tasks.
Design Focus	
Latency oriented design (low latency sequential execution)	Throughput-oriented design (high throughput via parallelism)
Cache Memory	
Large caches to reduce latency	Small cache to support bandwidth needs.
Memory Bandwidth	
Limited (due to general purpose design)	High (for large scale data movement in graphical task)
Arithmetics Units	
Low latency arithmetic oper	High throughput arthim oper

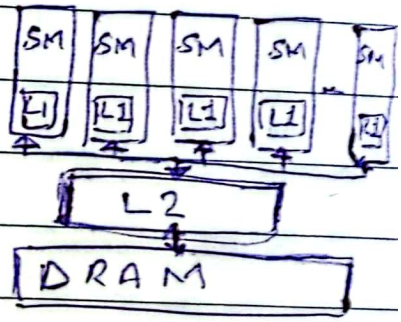
Page No.

streaming processor of a graphics processing unit (GPU) that allows a single instr. to be executed on multiple data sets simultaneously.

Date _____ Day _____

Characteristics Of GPU

- Specialized electronic circuits designed to process and manipulate memory for creating digital images
- Thousands of lightweight cores optimized for // processing.
- Efficiently handles graphical and mathematical workloads requiring parallel computations.
- Leverages data parallelism by executing the same operation on multiple data points
- Typically based on SIMD



Architecture Of GPU:

- It is organized into an array of highly threaded streaming multiprocessor (SMs)
- Two SMs form a Building block.
- However the no. of SMs in building block can vary from one generation to another.
- Each SMs has no. of streaming processors (SPs) that share control logic and instr. cache.
- Each GPU currently comes with gigabytes of

Date _____ Day

of Graphics Double Data Rate (GDDR), Synchronous DRAM (SDRAM) offered to as global memory.

"CUDA"

- Compute Unified Device Architecture -

→ It is a parallel computing platform and API developed by NVIDIA.
→ Provides APIs for optimizing GPU resource usage without needing specialized graphics knowledge.
→ Focuses on // processing for intensive computational tasks.

⇒ CUDA COMPUTE HIERARCHY:

o Threads:

→ A CUDA core is a parallel processor that executes floating-point calculations
→ Each thread has it's own memory registes, inaccessible to other threads
→ GPUs typically contain hundreds to thousands of CUDA cores

Day _____

Date _____

- Thread Block :
  - → A logical grouping of CUDA cores (threads), executed in series or parallel.
  - → Blocks share memory on a per-block basis
  - → CUDA architecture caps threads per block at 1024 threads
  - → Shared memory improves efficiency within a block.

- Kernel Grids :
  - → A grouping of thread blocks executing the same kernel.
  - → Used for large-scale parallel computations requiring more than 1024 threads.
  - → No shared memory accross blocks, limiting synchronization capabilities at this level.

- Threads within block can sync and share memory, improving performance for tasks needing frequent interaction.

Date Day

• CUDA Memory Hierarchy :

1. Registers :

- Private, on-chip memory allocated to individual threads (CUDA core)
- Fastest memory in the hierarchy because it is local to threads and directly on-chip.
- Managed automatically by the CUDA compiler
- Ideal for frequent accessed data within a single thread.

2. Read Only (RO) Memory :

- Specialized on-chip memory used for specific tasks like texture memory. Located on SMs.
- Fetching data from RO is faster than global memory.
- Used for tasks that involve constant data or textures that don't change during time.

3. L1 Cache And Shared Memory :

- Both are on-chip memory shared with thread blocks
  - ↳ L1 is managed by hardware
  - ↳ Shared memory managed by developers
- Faster than L2 & global mem.
- Suitable for thread-level data sharing.

Day _____

Date _____

4. L2 Cache :
- Shared cache accessible by all threads across all CUDA blocks.
- Slower than L1 but faster than global memory
- Temporary storage for frequently accessed data from global memory.

5. Global Memory:
- Memory residing in the device's DRAM like RAM in CPU.
- Slowest in hierarchy due to higher access latency.
- Used for large dataset that needs to be accessed across multiple threads & blocks.

Parallel Programming Languages & Models:

① OpenMP & OpenACC:

OpenMp was originally designed for CPU execution. A variation called OpenACC has been proposed and supported by multiple computer vendors for programming on heterogenous computing systems. OpenACC provides compiler automation & runtime support for abstracting away many // programing details.

Date Day

② MPI & CUDA :

MPI is a model where computing nodes in a cluster don't share memory. All interaction done thru explict msg passing.

CUDA is an effective interface ~~with each node~~, most

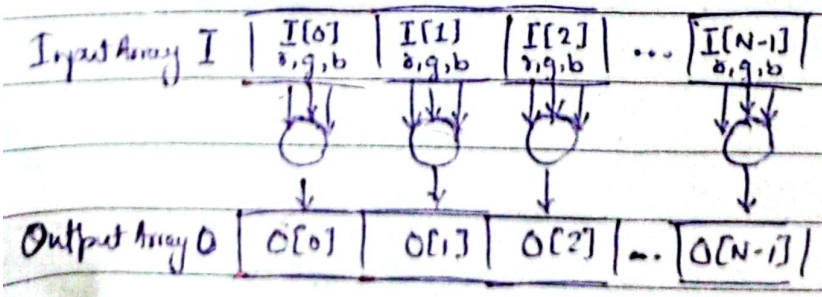
③ OpenCL :

Similar to CUDA, the OpenCL programming model defines language extensions and runtime APIs to allow programmers to manage ||ism and data delivery in massively || processors.

"DATA PARALLELISM"

→ To convert the color image to greyscale we compute the luminance value L for each pixel by applying the following weighted sum formula

$$L = r * 0.21 + g * 0.72 + b * 0.07$$

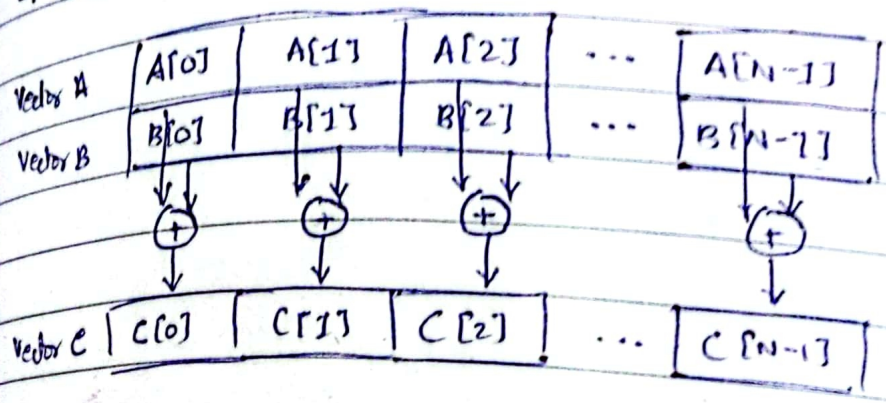


Day

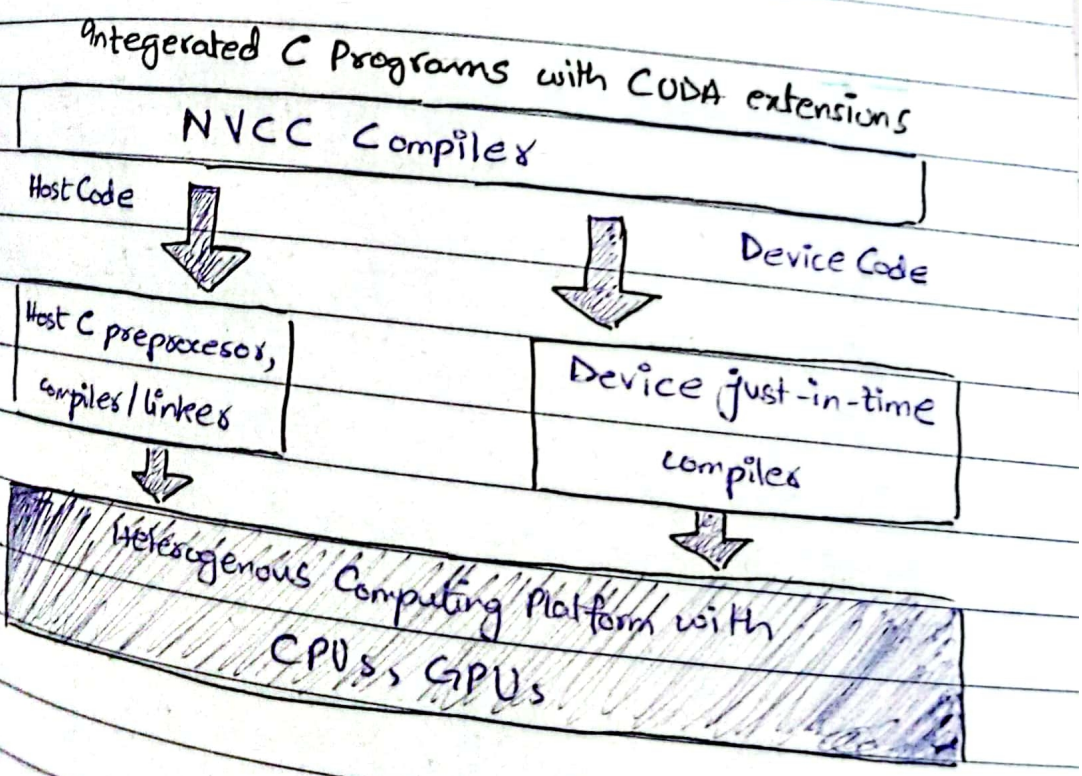
Date

→ Each array value can be calculated independently.

→ VECTOR ADDITION :



"CUDA C PROGRAM STRUC."



Page No.

Date _____ Day _____

→ The structure of a CUDA program reflects the coexistence of a Host (CPU) and one or more dev (GPU).

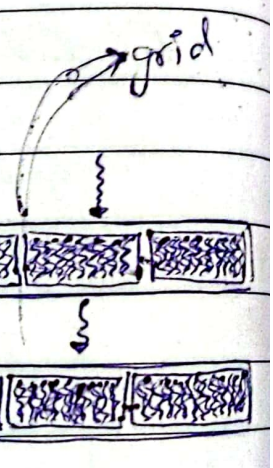
→ Each CUDA src file can have a mixture of both host & device code.

→ NVCC (NVIDIA C) compiler is used.

→ The device code is marked with CUDA keywords for labelling data-parallel functions, called kernels.

→ The device code is run on GPU device.

⇒ Execution of CUDA Program



→ The execution starts with a host (CPU) execution.

→ When a kernel function is called or launched it is executed by large no. of threads on a device.

→ All the threads that are generated by kernel launch are collectively called **grid**

↳ In the color-to-grey scale conversion, each thread in the grid can be used to compute one pixel of the output array O.

↳ The no. of threads = no. of pixels in image.

→ When all threads of kernel complete their execution,

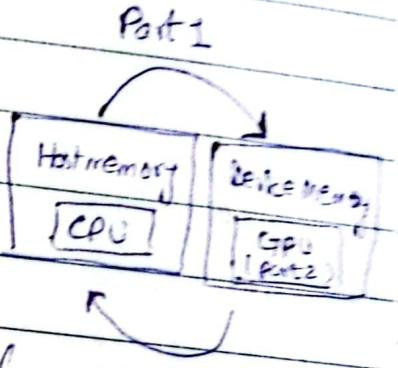
Date _____ Day

corresponding grid terminates and the execution continues on the host until another kernel is launched.

- Vector Addition Pseudo Code

```
#include <cuda.h>
void vecAdd (float *A, float *B, float *C, int n) {
    int size = n * sizeof (float);
    float *d_A, *d_B, *d_C;

1. // Allocate device memory
   // for A, B & C. Copy A & B
   // to device memory
2. // Kernel launch code - to have
   // the device to perform the actual
   // vector addition
3. // Copy C from the device memory
   // free device vectors
```



→ Names of variables processed by host are prefixed with "h_"
Names of variables processed by device are prefixed with "d_"

Date _____ Day _____

▷ Device Global Memory and Data Transfer:

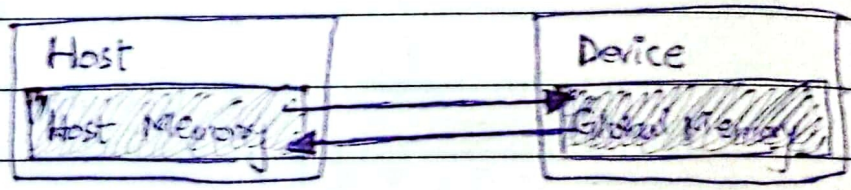
(GR)

→ Devices are often hardware cards that come with their own dynamic DRAM.

→ In CUDA host & devices have seperate memory spaces.

→ To execute a kernel on a device, the programmer needs to allocate global memory on the device and transfer relevant data from host to device memory.

→ After execution the programmer needs to transfer



result data from device to host memory.

→ After that free up the device memory.

- cudaMalloc() & cudaFree();

```
float *d_A;
int size = n * sizeof(float);
cudaMalloc((void**)&d_A, size);
....
cudaFree(d_A);
```

↳ d_A will be pointing to memory in global mem space

Day

cudaMemcpy() ;

→ Used for transfering data from Host ⇆ Device.

→ Requires 4 parameters

① Pointer to destination

② Pointer to source

③ Nuber of bytes to be copied

④ Type / Direction of transfer.

- → cudaMemcpyHostToDevice
- → cudaMemcpyDeviceToHost
- → cudaMemcpyDeviceToDevice
- → cudaMemcpyHostToHost

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n) {
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void**) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void**) &d_C, size);
    // Kernel invocation code
    ...

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device mem
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
```

Page No.

Date Day

Error handling:

```
cudaError_t err = cudaMalloc((void**)&d_A, size);
if (err != cudaSuccess){
    printf("%s in %s at line %d\n",
        cudaGetErrorString(err),
        __FILE__, __LINE__);
    exit(EXIT_FAILURE);
}
```

KERNEL FUNCTIONS AND THREADING:

→ A kernel function specifies the code to be executed by all threads during a parallel phase.

→ When a CUDA kernel is launched from the host code, the CUDA runtime system organizes the threads into a two-level hierarchy consisting of grids & blocks.

Grid:

↳ Composed of multiple blocks, where all blocks in a grid are of the same size.

↳ The no. of blocks & threads per block is specified in host code when kernel is launched.

↳ Example:

↳ If a grid contains 10 blocks with 256 threads

Day _____

ale _____

each, the total threads = 10 × 256 = 2560

- Blocks: (Thread Block)
  - ↳ A block contains upto 1024 thread
  - ↳ Threads in a block work together, often sharing resources like shared memory.
  - ↳ Blocks execute independently but can sync within a single block.

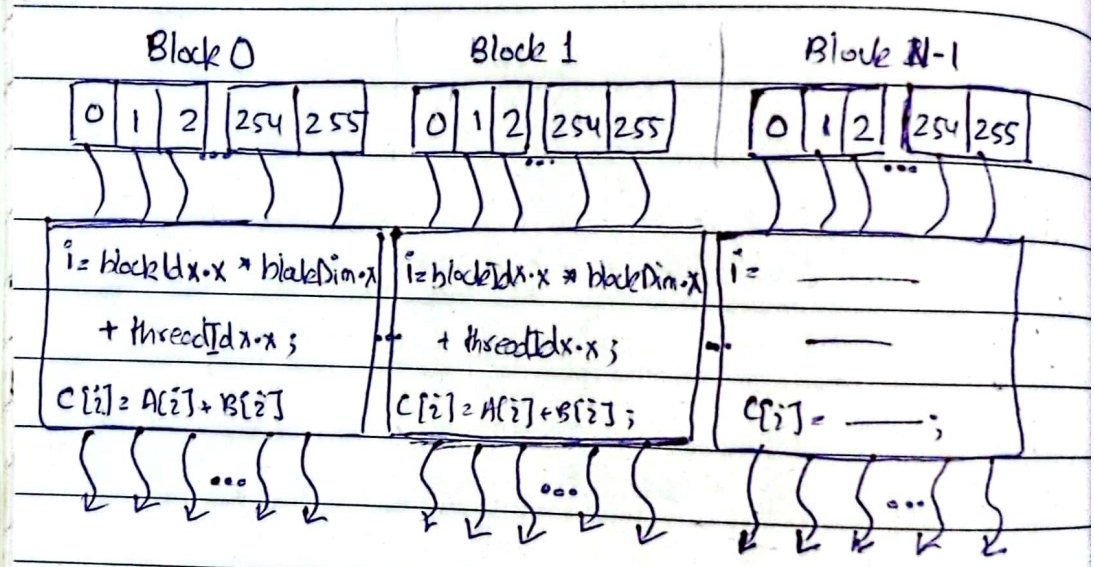
- Built-in Variable : blockDim
  - → Represents the dimensions of a block in terms of threads
  - → Available as a struct with three unsigned integer fields: x, y and z.

blockDim.x ⟶ No. of threads in x-dimension
blockDim.y ⟶ " " " y-dimension
blockDim.z ⟶ " " " z-dimension

- Thread Organization:
  - → Threads are often organized to reflect the shape of the data being processed:
    - ↳ 1D array: use blockDim.x
    - ↳ 2D ~~array~~ matrix: use blockDim.x & blockDim.y
    - ↳ 3D volume: use all three field x, y & z

Date _____ Day _____

SPMD is not the same as SIMD. In an SPMD system, the parallel processing units execute the same program on multiple parts of the data. However these processing units do not need to be executing the same instr. at the same time. In an SIMD sys, all processing units are executing the same instr. at any instant.



→ For a given grid of threads, the no. of threads in a block is available in the blockDim variable.

→ The value of blockDim.x is 256.

→ The dimensions of thread block should be multiples of 32 due to hardware efficiency.

→ Each thread in a block has a unique threadIdx value.

→ CUDA kernel have access to two more built in variables threadIdx, blockIdx

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

Day _____

Date _____

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition

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

→ this variable is private to each thread.

Each thread in the grid corresponds to one iteration of the original loop

		Executed on the	Only callable from the
__device__	float DeviceFunc()	device	device
__global__	void KernelFunc()	device	host
__host__	float HostFun()	host	host

→ By default all functions in a CUDA program are host functions if they donot have any of the above keywords

→ One can use "__host__" & "__device__" in a function declaration.

→ this combination tells the compiles to generate two versions of the object files for the same function & this function can be called from host & used in host & can also be called from kernel & ~~call~~ used in kernel.

Date _____ Day _____

→ When the host code launches a kernel, it sets the grid and thread block dimensions via execution configuration parameters. These are given between the "<<<" and ">>>".

↳ The 1st parameter → no. of thread blocks in grid

↳ The 2nd → no. of threads in each block.

```
int vectAdd(float *A, float *B, float *C, int n) {
    vecAddKernel<<<ceil(n/256.0), 256>>>
        (d_A, d_B, d_C, n);
}
```

(total no. of values)

if total element = 1000
so 4 blocks created
if 4000 then 16.

⊙ ~~KERNE LAUNCH~~

yaha par gridDim.x & blockDim.x ki value kitni initialize hogayegi.

gridDim.x = 4 (n=1000)
blockDim.x = 256

Date _____ Day _____

CHAPTER # 03

CUDA THREAD ORGANIZATION:

→ All threads in a grid execute the same kernel func.

→ They rely on cordinates to distinguish themselves from one & another & identify appropriate portion of data to process

→ All threads in a block share the same block index which is blockIdx

→ Each thread has a thread index which is threadIdx

→ The thread Id & blockId together form a cordinate for a thread which helps to distinguish b/w 2 threads

→ The exact organization of grid is determined by the execution configuration parameter.

<< ..... >>

1st arg specifies the no. of thread blocks ↙

↳ no. of threads in each block.

→ Each of this parameter is of type dim3

↳ it is a C structure

```
dim3 dimGrid(32, 1, 1)
dim3 dimBlock(128, 1, 1)
```

} creates 1D of 32 blocks, each block have 128 threads. total 128 x 32 = 4096 threads

these can be any C valid names.

```
vecAddKernel <<< dimGrid, dimBlock >>> (...);
```

Date _____ Day _____

→ All threads in a block share same blockIdx.x, blockIdx.y, blockIdx.z.

And value for each dimension ranges from

- blockIdx.x → 0 — gridDim.x - 1
- " .y → 0 — gridDim.y - 1
- " .z → 0 — gridDim.z - 1

→ The no. of threads in a block cannot be greater than 1024

→ blockDim (512, 1, 1) = 512 x 1 x 1 = 512 threads

→ blockDim (8, 16, 4) = 8 x 16 x 4 = 512

→ blockDim (32 x 16 x 2) = 1024

→ blockDim (32 x 32 x 2) = 2048 (Not allowed)

→ dim3 dimGrid (2, 2, 1);

→ dim3 dimBlock (4, 2, 2);