Programming Multi / Many Cores using CUDA & OpenCL

- Accelerator → Hardware device or Software for enhancing overall performance of Computer.
- GPU has higher parallism than CPU.
  - GPU → many small GPU cores.
  - CPU → multiple cores.
- 3 Ways to accelerate applications.
  - Libraries
  - OpenACC Directives
  - Language
  - Libraries (Ease of use, Quality, Performance)
  - Open ACC directives → - Simple Compiler hints
    - Easy, open, Powerful.
    - Works on GPU cores + CPU cores performance is very high.
    - Avoids restructuring of code for production application.
  - GPU Programming languages → Fortran, C, C++, Python, F#
    - Maximum Flexibility
- CUDA (Compute Unified Device Architecture) → Embeded in hardware
  - helps in
    - Crypto Mining
    - Video Rendering
    - Machine learning.
  - CUDA Architecture → • Use GPU cores for general-purpose computing.
  - CUDA C / C++ →
- Hetrogeneous Computing (Use different processors)
  - Host (CPU + its memory)
  - Device (GPU + its memory)
- Simple Processing Flow
  1. CPU ki memory say input data ko GPU ki memory main copy kiya jata hy (PCI bus say)

CUDA

Gpu computing $\xrightarrow{\text{Purpose}}$ more parallelism

Runs on device, called from host code.

- global void.
- nvcc (nvidia Compiler) Separates code in 2 hissay.
  - Device function → mykernel() nvidia compiler.
  - Host function → main() gcc handles

- mykernel <<< 1,1 >>> ();
  - Triple angle brackets mark call from → host to device.
  - Also called Kernal launch.

- Memory Managment

Both can pass or receive.

{ Device Pointers (Host code main direct use ni hty)
{ Host Pointers (Device code mein direct use ni hty)

cudaMalloc () (allocate)
cudaFree () (free)
cudaMemcpy () (copy)

- Vector addition like $\begin{pmatrix}3\\4\end{pmatrix} + \begin{pmatrix}2\\1\end{pmatrix} = \begin{pmatrix}5\\5\end{pmatrix}$

```
global void add (int* a, int* b, int* c)
{ c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x]; }
```

Example.

```
global void add (int* a,
  int* b, int* c)
{
c[blockIdx.x] = a[blockIdx.x] +
        b[blockIdx.x].
int main (void)
{int *a, *b, *c; // host copies
int *da, *db, *dc,
```

```
int size = N*sizeof(int)
cudaMalloc ((void **) &da, size)
cudaMalloc ((void **) &db, size)
cudaMalloc ((void **) &dc, size)
```

Allocate space for device copies

```
// Allocate space for host copies.

a = (int *) malloc (size);
b = (int *) malloc (size);
c = (int *) malloc (size);

// copy inputs to device.

CudaMemcpy (da, a, size, cudaMemcpyHostToDevice);
Cuda Memcpy (db, b, size, cudaMemcpyHostToDevice);

add <<< N, 1>>> (da, db, dc);

cudaMemcpy (c, dc, size, cudaMemcpyDeviceToHost);

free (a)
free (b)
free (c)

cudafree (da);
cuda free (db);
cudafree (dc);

}
```

blockIdx (grid main block kis index)

threadIdx (block main thread ki index)
blockDim (EK block main total threads)
~~gridDim (EK grid main total blocks)~~

- CUDA Threads. (A block can be split into parallel threads).

```
__global__ void add (int* a, int* b, int* c) {
c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

Same as previous code but threads more.

```
add <<< 1, N >>> (da, db, dc)
```

- 1 block main M threads hain. (Threads per block)

```
int index = threadIdx.x + blockIdx.x * M;
```

- Built-in variable (blockDim.x) For threads per block.

```
__global__ void add (int* a, int* b, int* c)
{
int index = threadIdx.x + blockIdx.x * blockDim.x;
c[index] = a[index] + b[index];
}
```

Same main but.

```
#define N (2048 x 2048)
#define THREADS_PER_BLOCK 512
add <<< N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >>> (da, db, dc)
```

- When kernel launched it works like grid.
  many blocks
  many threads.

CPU launch code to GPU. CPU do not stop.
Executes aur wala code.

- Kernal launches are asynchronous.
  - Control returns to CPU.
  - Results say pehlay synchronize karna zaroori hy.

cudaMemcpy() → - Blocks CPU until copy is completed
- Copy tab shuru hti hy jab phly wali CUDA calls complete hti hain.

cudaMemcpyAsync() → does not block CPU.

cudaDeviceSynchronize() → Blocks CPU until phly wali saari CUDA calls finish na ho jaye.

cudaGetLastError()
cudaGetErrorString()
cudaGetDeviceCount(int* count) - System main kitnay GPU hain.
cudaGetDevice(int* count) - Abhi kn sa GPU selected hy.
cudaSetDevice(int count) - Selects specific GPU jo chahiye.
cudaGetDeviceProperties(cudaDeviceProp* prop, int device)
↓
prop likes memory size, cores etc.

- Multiple threads can use 1 GPU.
- Single thread can select multiple devices but cudaSetDevice() call kr kr kay device select krna prta hy.

PDC: Practice

1) N×M.
2) cudaGetLastError() launched immediately.
   return cudaSuccess.

- CUDA Program

```
#include <stdio.h>
# include <cuda-runtime.h>.

__global__ void mul(int* a, int* b, int* c, int n)
{
        int idx = threadIdx.x + blockIdx.x × blockDim.x;
        if(idx < n){
        c[idx] = a[idx] × b[idx]; }
}
int main () {
 int n=10;
 int size = n* sizeof(int);
 int h_a [2] = {1,2}
 int h_b [2] = {3,4}
 int h_c [2];
   int* d_a, *d_b, *d_c;
   cudaMalloc((void**) &d_a, size);
cudaMalloc((void**) &d_b, size);
cudaMalloc((void**) &d_c, size);
```

```
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
blo = (n+256-1) / 256
mul <<< blo, 256 >>> (d_a, d_b, d_c, n);

cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceTo
                                    Host);
```

```
# include <stdio.h>
# include <cuda_runtime.h>

__global__ void add (int *a, int *b, int *c, n)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x
    if (idx < n)
    {   c[idx] = a[idx] + b[idx];  }
}
int main ()                int t=256;
{
    int n = 4              int blo = (n+256-1)/256;
    int h_a [n] = {1,2,3,4};
    int h_b [n] = {5,6,7,8,9};
    int size = n * sizeof (int); int *d_a, *d_b, *d_c;
cudaMalloc ((void**) &d_a, size)
cudaMalloc ((void**) &d_b, size)
cudaMalloc ((void**) &d_c, size)
cudaMemcpy (d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_b, h_b, size, cudaMemcpyDeviceToHost);
add <<<blo, t>>> (d_a, d_b, d_c, n);
cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceTo
                                    Host);
```