

Programming Multi / Many cores using

CUDA & OpenCL

-- Accelerator → Hardware device or Software for enhancing overall performance of Computer.

-- GPU has higher parallelism than CPU.

↓
many
small
GPU cores.

↓
multiple
cores.

-- 3 Ways to accelerate applications. { Libraries, OpenACC Directives, Language }

• Libraries (Ease of use, Quality, Performance)

• OpenACC 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) → Embedded in hardware
↓ help in

• Crypto Mining

• Video Rendering

• Machine Learning

• CUDA Architecture →

• Use GPU cores for general-purpose computing.

• CUDA C / C++ →

-- Heterogeneous Computing (Use different processors)
Host (CPU + its memory)
Device (GPU + its memory)

• Simple Processing Flow

① CPU's memory say input data ko GPU ki memory mein copy krj.
↳ krj ky (PCI bus say)

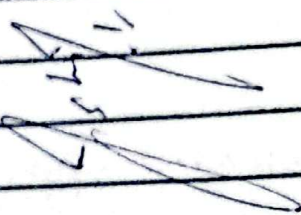
CUDA

Gpu computing $\xrightarrow{\text{purpose}}$ move parallelism

Runs on device, called from host code.



- global void.
- nvcc (Nvidia Compiler) Separates code in 2 ways.
 - Device function \rightarrow myKernelC) nvcc compiler.
 - Host function \rightarrow main() gcc handles
- myKernel <<< 1, 1 >>> C);
 - Triple angle brackets mark call from \rightarrow host to device.
 - Also called Kernel launch.



Memory Management

- Both (comp. pass or receiver) {
 - Device Pointers (Host code main direct use natively)
 - Host Pointers (Device code main direct use natively)
- 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 = NxSize of(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, cudaMemcpy HostTo Device);

CudaMemcpy (db, b, size, cudaMemcpy HostTo Device);

Add <<< N, 1>>> - (da, db, dc);

CudaMemcpy (c, dc, size, cudaMemcpy DeviceTo Host);

free (a)

free (b)

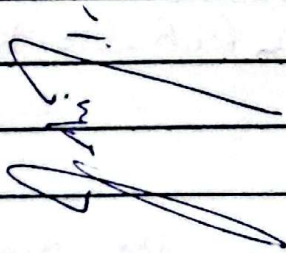
free (c)

CudaFree (da);

CudaFree (db);

CudaFree (dc);

}



blockIdx (grid main block ki jagah)

threadIdx (block main thread ki jagah)

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 x, a, int x, b, int x, 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.

int index = threadIdx.x + blockIdx.x * blockDim.x;

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

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

Same main but.

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

- When kernel launched it works like grid.

many blocks
↓
many threads.

CPU send code to GPU. CPU do not stop.
Executes asynchronously.

• Kernel launched are asynchronous.

- Control returns to CPU.
- Return say پہلے Synchronize karna karni hai.

`cudaMemcpy()` → - Blocks CPU until copy is completed
- Copy tab shuru hti hai jab phay wadi
CUDA call complete hti hai.

`cudaMemcpyAsync()` → does not block CPU.

`cudaDeviceSynchronize()` → Blocks CPU until phay wadi
sari CUDA call finish na hojaye.

`cudaGetLastError()`

`cudaGetErrorString()`

`cudaGetDeviceCount(int * count)` - System main kitni GPU hai.

`cudaGetDevice(int * count)` - Abhi koi GPU selected hai.

`cudaSetDevice(int count)` - Select 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 device select
karna pata hai.

PDC: Practice

- 1) NXM.
- 2) `cudaGetLastError()` launched. immediately.
`return cudaSuccess.`

- CUDA Program

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

__global__ void mul(const int* a, int* b, int* c)
{
    int idx = threadIdx.x * blockDim.x + dimblockIdx.x;
    if (idx < n) {
        a[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;
mod 224 blo, 256 >>> (d-a, d-b, d-c, n);
cudaMemcpy(h-c, d-c, size, cudaMemcpyDeviceToHost);

```

```

#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 n = 4;
    int h-a[n] = {1, 2, 3, 4};
    int h-b[n] = {5, 6, 7, 8};
    int size = n * sizeof(int);
    void *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);
    add <<< blo, >>> (d-a, d-b, d-c, n);
    cudaMemcpy(h-c, d-c, size, cudaMemcpyDeviceToHost);
}

```