**Lab Exercise 1.1 – Parallel Computing in CUDA Programming**

**Objective:**

* Understand the basics of parallel computing using CUDA.
* Implement a simple CUDA program that demonstrates parallel execution of threads.
* Measure the performance benefit of parallelism.

**1. Key Concepts of Parallel Computing in CUDA**

| **Concept** | **Description** |
| --- | --- |
| Kernel | A function that runs on the GPU and is executed by multiple threads. |
| Thread | A single execution unit within a CUDA block. |
| Block | A group of threads that can share data using shared memory. |
| Grid | A collection of blocks. |
| SIMT | Single Instruction, Multiple Thread execution model used by CUDA. |

**2. CUDA Program: Parallel Array Addition**

#include <iostream>

#include <cuda\_runtime.h>

\_\_global\_\_ void addArrays(int \*a, int \*b, int \*c, int N) {

int tid = blockIdx.x \* blockDim.x + threadIdx.x;

if (tid < N) {

c[tid] = a[tid] + b[tid];

}

}

int main() {

const int N = 1000;

int h\_a[N], h\_b[N], h\_c[N];

int \*d\_a, \*d\_b, \*d\_c;

for (int i = 0; i < N; i++) {

h\_a[i] = i;

h\_b[i] = i \* 2;

}

cudaMalloc(&d\_a, N \* sizeof(int));

cudaMalloc(&d\_b, N \* sizeof(int));

cudaMalloc(&d\_c, N \* sizeof(int));

cudaMemcpy(d\_a, h\_a, N \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, N \* sizeof(int), cudaMemcpyHostToDevice);

int threadsPerBlock = 256;

int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

addArrays<<<blocksPerGrid, threadsPerBlock>>>(d\_a, d\_b, d\_c, N);

cudaMemcpy(h\_c, d\_c, N \* sizeof(int), cudaMemcpyDeviceToHost);

std::cout << "First 10 results:\n";

for (int i = 0; i < 10; ++i)

std::cout << h\_c[i] << " ";

std::cout << std::endl;

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

return 0;

}

**3. Explanation**

* **Kernel Function** addArrays: Runs in parallel on the GPU. Each thread computes one index of the output array.
* **Thread Indexing**: Each thread computes tid using blockIdx.x \* blockDim.x + threadIdx.x to identify which part of the array it should handle.
* **CUDA Memory Management**: cudaMalloc, cudaMemcpy, and cudaFree handle GPU memory.

**Step 1: Include Headers**

#include <iostream>

#include <cuda\_runtime.h>

* #include <iostream>: Used for input and output operations (like std::cout).
* #include <cuda\_runtime.h>: Provides CUDA runtime API functions (like cudaMalloc, cudaMemcpy).

**Step 2: Define the Kernel Function**

\_\_global\_\_ void addArrays(int \*a, int \*b, int \*c, int N) {

int tid = blockIdx.x \* blockDim.x + threadIdx.x;

if (tid < N) {

c[tid] = a[tid] + b[tid];

}

}

* \_\_global\_\_: Declares a kernel function that will run on the GPU.
* blockIdx.x, threadIdx.x, blockDim.x:
  + blockIdx.x: Block's index in the grid.
  + threadIdx.x: Thread's index inside its block.
  + blockDim.x: Number of threads per block.
* tid: Global thread ID = identifies *which element* of the array this thread will work on.
* if (tid < N): Each thread checks if it is assigned a valid array element.
* c[tid] = a[tid] + b[tid];: Each thread adds one pair of elements (a[tid] + b[tid]) and stores it in c[tid].

**Step 3: Start of Main Function**

int main() {

const int N = 1000;

int h\_a[N], h\_b[N], h\_c[N];

int \*d\_a, \*d\_b, \*d\_c;

* Declare:
  + N: Size of the arrays (1000 elements).
  + h\_a, h\_b, h\_c: Arrays on the **host (CPU)**.
  + d\_a, d\_b, d\_c: Pointers for arrays on the **device (GPU)**.

**Step 4: Initialize Host Arrays**

for (int i = 0; i < N; i++) {

h\_a[i] = i;

h\_b[i] = i \* 2;

}

* Fill h\_a with values 0, 1, 2, ..., 999.
* Fill h\_b with values 0, 2, 4, ..., 1998.

**Step 5: Allocate Memory on the Device**

cudaMalloc(&d\_a, N \* sizeof(int));

cudaMalloc(&d\_b, N \* sizeof(int));

cudaMalloc(&d\_c, N \* sizeof(int));

* Allocate memory for d\_a, d\_b, and d\_c on the **GPU** (device).

**Step 6: Copy Data from Host to Device**

cudaMemcpy(d\_a, h\_a, N \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, N \* sizeof(int), cudaMemcpyHostToDevice);

* Copy the initialized host arrays h\_a and h\_b into the device arrays d\_a and d\_b.

**Step 7: Configure Threads and Blocks**

int threadsPerBlock = 256;

int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

* Set:
  + threadsPerBlock = 256 (maximum number of threads per block).
  + blocksPerGrid = Number of blocks needed = (1000 + 256 - 1) / 256 = 4.
* CUDA rounds up because 1000/256 is not a whole number. So, 4 blocks are needed.

**Step 8: Launch the Kernel**

addArrays<<<blocksPerGrid, threadsPerBlock>>>(d\_a, d\_b, d\_c, N);

* Launch the kernel with 4 blocks, each with 256 threads (up to 1024 threads total).
* Each thread handles **one** addition operation.

**Step 9: Copy Result Back to Host**

cudaMemcpy(h\_c, d\_c, N \* sizeof(int), cudaMemcpyDeviceToHost);

* After GPU computation is done, copy the result array d\_c back to host memory into h\_c.

**Step 10: Print the First 10 Results**

std::cout << "First 10 results:\n";

for (int i = 0; i < 10; ++i)

std::cout << h\_c[i] << " ";

std::cout << std::endl;

* Print the first 10 elements of the result array h\_c.
* Expected values:  
  0+0=0, 1+2=3, 2+4=6, 3+6=9, 4+8=12, etc.  
  So output will be:

0 3 6 9 12 15 18 21 24 27

**Step 11: Free Device Memory**

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

* Free the memory allocated on the GPU for d\_a, d\_b, and d\_c.

**Step 12: End of Program**

return 0;

* Exit the program successfully.

**4. Compile and Run**

nvcc -o parallel\_add parallel\_add.cu

./parallel\_add

Expected output:

First 10 results:

0 3 6 9 12 15 18 21 24 27

**5. Summary**

* This exercise demonstrates how to **parallelize array operations** using CUDA threads.
* The GPU executes hundreds or thousands of threads **concurrently**, greatly improving performance for data-parallel tasks.
* The key to effective CUDA programming is understanding how to **divide work among threads and blocks**.