Veermata Jijabai Technological Institute, Mumbai 400019

Experiment No.: 02

Aim : Implementation of parallel search algorithm(BFS) using CUDA

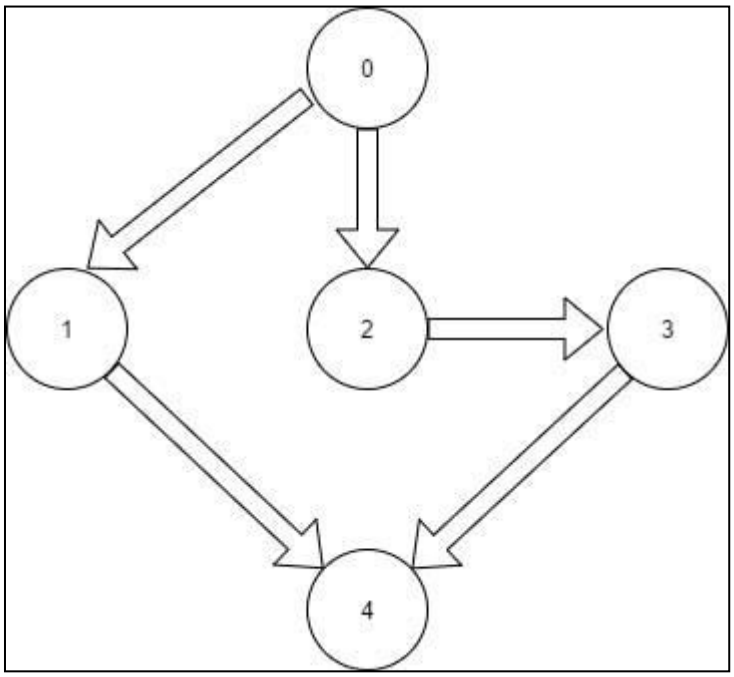
Name : Kiran K Patil

Enrolment No.: 211070904

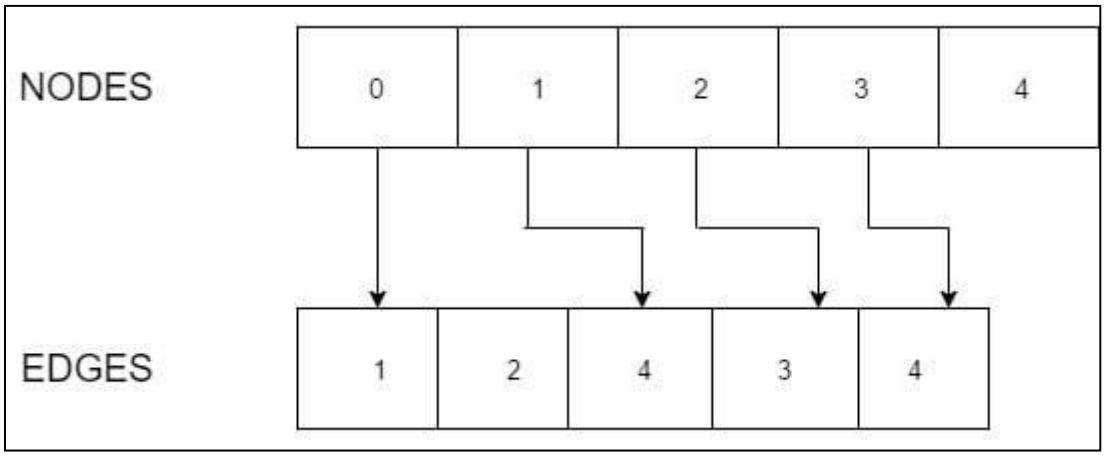
Branch: Computer Engineering

Batch: IV

Here I have taken the input as follows :



Nodes and the edges in the below example :



Program :

```
%%cu
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_NODES 5

typedef struct
{
  int start;     // Index of first adjacent node in Ea

  int length;    // Number of adjacent nodes
} Node;

__global__ void CUDA_BFS_KERNEL(Node *Va, int *Ea, bool
 *Fa, bool *Xa, int *Ca,bool *done)
{

  int id = threadIdx.x + blockIdx.x * blockDim.x;
  if (id > NUM_NODES)
    *done = false;


  if (Fa[id] == true && Xa[id] == false)
  {
    printf("%d ", id); //This printf gives the order of
 vertices in BFS
    Fa[id] = false;
    Xa[id] = true;
```

```
    __syncthreads();
    int k = 0;
    int i;
    int start = Va[id].start;
    int end = start + Va[id].length;
    for (int i = start; i < end; i++)
    {
      int nid = Ea[i];

      if (Xa[nid] == false)
      {
        Ca[nid] = Ca[id] + 1;
        Fa[nid] = true;
        *done = false;
      }

    }

  }

}

// The BFS frontier corresponds to all the nodes being
processed at the current level.


int main()
{

   Node node[NUM_NODES];


  //int edgesSize = 2 * NUM_NODES;
  int edges[NUM_NODES];
```

```
  node[0].start = 0;
  node[0].length = 2;

  node[1].start = 2;
  node[1].length = 1;

  node[2].start = 3;
  node[2].length = 1;

  node[3].start = 4;
  node[3].length = 1;

  node[4].start = 5;
  node[4].length = 0;

  edges[0] = 1;
  edges[1] = 2;
  edges[2] = 4;
  edges[3] = 3;
  edges[4] = 4;

  bool frontier[NUM_NODES] = { false };
  bool visited[NUM_NODES] = { false };
  int cost[NUM_NODES] = { 0 };

  int source = 0;
  frontier[source] = true;

  Node* Va;
  cudaMalloc((void**)&Va, sizeof(Node)*NUM_NODES);
  cudaMemcpy(Va, node, sizeof(Node)*NUM_NODES, cudaMemc
pyHostToDevice);
```

```
  int* Ea;
  cudaMalloc((void**)&Ea, sizeof(Node)*NUM_NODES);
  cudaMemcpy(Ea, edges, sizeof(Node)*NUM_NODES, cudaMem
cpyHostToDevice);

  bool* Fa;
  cudaMalloc((void**)&Fa, sizeof(bool)*NUM_NODES);
  cudaMemcpy(Fa, frontier, sizeof(bool)*NUM_NODES, cuda
MemcpyHostToDevice);

  bool* Xa;
  cudaMalloc((void**)&Xa, sizeof(bool)*NUM_NODES);
  cudaMemcpy(Xa, visited, sizeof(bool)*NUM_NODES, cudaM
emcpyHostToDevice);

  int* Ca;
  cudaMalloc((void**)&Ca, sizeof(int)*NUM_NODES);
  cudaMemcpy(Ca, cost, sizeof(int)*NUM_NODES, cudaMemcp
yHostToDevice);

  int num_blks = 1;
  int threads = 5;

  bool done;
  bool* d_done;
  cudaMalloc((void**)&d_done, sizeof(bool));
  printf("\n\n");
  int count = 0;

  printf("************ Implementation of parallel searc
h algorithm(BFS) using CUDA ******************** \n\n"
);

  printf("Order: \n\n");
```

```
  do {
    count++;
    done = true;
    cudaMemcpy(d_done, &done, sizeof(bool), cudaMemcpyH
ostToDevice);
    CUDA_BFS_KERNEL <<<num_blks, threads >>>(Va, Ea, Fa
, Xa, Ca,d_done);
    cudaMemcpy(&done, d_done , sizeof(bool), cudaMemcpy
DeviceToHost);

  } while (!done);


  cudaMemcpy(cost, Ca, sizeof(int)*NUM_NODES, cudaMemcp
yDeviceToHost);

  printf("\n\nNumber of times the kernel is called : %d
 \n", count);


  printf("\nCost: ");
  for (int i = 0; i<NUM_NODES; i++)
    printf( "%d    ", cost[i]);
  printf("\n");

}
```

Output :

```
********** Implementation of parallel search algorithm(BFS) using CUDA ********************

Order:

0 1 2 3 4

Number of times the kernel is called : 3

Cost: 0    1    1    2    2
```

Conclusion:

- Thus we have implemented the parallel search algorithm (BFS) using CUDA.
- This program uses a CUDA kernel to perform BFS in parallel on a GPU.
- Each thread in the kernel processes one node in the graph and updates the distances of its neighbours.
- The number of blocks and threads per block can be adjusted to control the degree of parallelism.