Name : Manthan Gopal Dhole **TY IT**
Id : 201080020 **Parallel Computing**

Link : Click IT

Experiment 2

Aim: Write a code for implementing BFS(Breadth First Search) using CUDA.

Theory :

BFS (Breadth-First Search) is a graph traversal algorithm that visits all the vertices of a graph in breadth-first order. CUDA is a parallel computing platform and API that allows you to run computationally intensive tasks on the GPU (Graphics Processing Unit) rather than the CPU. By using CUDA to implement BFS, you can speed up the computation by utilizing the parallel processing capability of the GPU.

Here's a high-level explanation of how BFS can be implemented using CUDA:

Convert the graph into a data structure that can be processed in parallel on the GPU, such as an adjacency list or matrix.

Allocate GPU memory to store the graph and the distances from the source vertex to all other vertices.

Copy the graph and the distances from the CPU memory to the GPU memory.

Launch a CUDA kernel to perform the BFS. The kernel will be executed in parallel by multiple threads on the GPU.

In the CUDA kernel, each thread processes one vertex at a time. If the distance of the vertex has not been set yet, it is set to the distance of the source vertex plus one. The distance of all the

neighbors of the vertex are also updated using the atomicMin function, which ensures that only one thread can update the distance of a neighbor at a time.

The BFS continues until all vertices have been processed.

Finally, copy the updated distances from the GPU memory back to the CPU memory.

By using CUDA to parallelize the BFS, you can significantly speed up the computation, especially for large graphs with many vertices and edges.

Code :

```
%%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;
```

Name : Manthan Gopal Dhole **TY IT**
Id : 201080020 **Parallel Computing**

```
  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, cudaMemcpyHostToDevice);

int* Ea;
```

Name : Manthan Gopal Dhole **TY IT**
Id : 201080020 **Parallel Computing**

```
  cudaMalloc((void**)&Ea, sizeof(Node)*NUM_NODES);
  cudaMemcpy(Ea, edges, sizeof(Node)*NUM_NODES, cudaMemcpyHostToDevice);

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

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

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



  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("Order: \n\n");
  do {
    count++;
    done = true;
    cudaMemcpy(d_done, &done, sizeof(bool), cudaMemcpyHostToDevice);
    CUDA_BFS_KERNEL <<<num_blks, threads >>>(Va, Ea, Fa, Xa, Ca,d_done);
    cudaMemcpy(&done, d_done , sizeof(bool), cudaMemcpyDeviceToHost);

  } while (!done);
```

Name : Manthan Gopal Dhole **TY IT**
Id : 201080020 **Parallel Computing**

```
  cudaMemcpy(cost, Ca, sizeof(int)*NUM_NODES, cudaMemcpyDeviceToHost);

  printf("Number 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 :

```
Order:

0 1 2 3 4 Number of times the kernel is called : 3

Cost: 0    1    1    2    2
```

Conclusion:

In conclusion, BFS (Breadth-First Search) is a widely used graph traversal algorithm that visits all the vertices of a graph in breadth-first order. By using CUDA to implement BFS, the computation can be significantly sped up by utilizing the parallel processing capability of the GPU. With CUDA, BFS can be parallelized and executed in parallel by multiple threads on the GPU, which leads to faster computation times, especially for large graphs with many vertices and edges. Implementing BFS using CUDA requires a good understanding of both graph algorithms and parallel computing concepts, but the potential performance gains make it a worthwhile effort.

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing