INTERNSHIP REPORT

GPU Programming in CUDA Framework

**SUBMITTED BY:**

Name: Vartika Singh

Institute: Veer Surendra Sai University of Technology, Burla

Current Education: Bachelors of Technology (2017 – 2021)

Department: Dept. of Computer Science and Engineering

Under the Guidance of Dr. Pabitra Mitra, Professor, Computer Sc. And Engg.

Dept., IIT Khargpur.

CONTENTS

1. Acknowledgement
2. Introduction
3. Introduction to GPU
4. GPU Pipeline
5. CPU vs GPU
6. GPU Design
7. Introduction to CUDA
8. Programming Structure of CUDA
9. CUDA Thread Organisation
10. Resource Assignment to Blocks
11. Synchronisation between Threads
12. Thread Scheduling
13. Memory in CUDA
14. Writing a CUDA Program
15. Kernel Function
16. CUDA Keywords
17. Some Programming Applications of CUDA
    1. Matrix Addition
    2. Matrix Multiplication
    3. Path using Breadth First Search(BFS)
18. Other Applications of CUDA
19. Conclusion
20. References

**ACKNOWLEDGEMENT**

I wish to express my sincere gratitude to Dr. Pabitra Mitra, Professor, Computer Sc. And Engg. Dept., IIT Kharagpur, for providing me an opportunity to do my internship and project work in “Dept. of Computer Sc. and Engg., IIT Kharagpur”.

I sincerely thank Dr. Pabitra Mitra for his giudance and encouragement in carrying out this project work. His help in the clarification of doubts during the entire project work made the experience more interesting and easier.

I would also like to thank “IIT Kharagpur” for serving the perfect environment of learning in cooperation with other interns, which proved to be a great motivation in the long run.

This was an extremely overwhelming and dedicated learning experience which helped me build a stronger knowledge on the topic studied as well as its practical applications.

**INTRODUCTION**

This report is a short overview of the project “Parallel Prgramming in CUDA Framework” that was assigned.

The project is to study the basic concepts of parallelism in GPU and exploiting the parallelising capabilities of GPU using CUDA platform. It involves the study of the structure of a CUDA program and the different tyes of memory and ways to handle them.

An introduction to the Thread and Block concept is also included for better understanding.

Three CUDA programming codes along with explanation are included to elaborate on the practical working of the CDUA programs and its interaction with the users.

Finally, some other applications and futuristic overview is provided so as to acknowledge with the effective use of the CUDA API platform.

**INTRODUCTION TO GPU**

GPU is a microprocessor that has been designed specifically for processing 3-D graphics. It is capable of handling millions of math-intensive processes per second.

**GPU PIPELINE**

* The GPU recieves geometry information from the CPU as an input and provides a picture as an output.
* The host interface is the communication bridge between the CPU and the GPU.
* It recieves commands from the CPU and also pulls geometry information from system memory.
* It outputs a stream of verticesin object space will all their associated information.
* The vertex processing stage recieves vertices from the host interface in object space and outputs them in screen space.
* It may be a simple linear transformation, or a complex operation involving morphing effects.

Triangle setup

Host

interface

Memory

interface

Pixel

processing

Vertex

processing

**CPU vs GPU**

**CPU**

* It works with multi-cores.
* It is faster for sequential execution.
* It has a larger instruction set than GPU.
* Complex ALU.
* Better branch prediction logic.
* Has more sophisticated caching/ pipeline schemes.
* Instuction cycles are a lot faster.

**GPU**

* It deals with many-cores.
* It is used to deal with large chunks of data.
* It utilises many cores.
* Each core is slower than CPU.
* It focuses on executon throughput of massively parallel-programming.
* It works on SIMD(Single Instruction Multiple Data).
* FLOPS(Floating point Operations Per Second) is more.
* Designed for data extensive applications.
* DRAM of GPU has a higher bandwidth.
* Originally designed of 3-D rendering.
* Inefficient for SPMD(Single Programming Multiple Data).

**GPU DESIGN**

A CUDA capable GPU consists of a fixed number of streaming multiprocessors. Again, each SM contains a fixed set of streaming processors. Each SP has an MAD(Multiply and Addition Unit) and an additional MU (Multiply Unit). Each SP is masively threaded , and can run thousands of threads per application i.e. it consists of a fixed number of threads that can run parallely and access same set of resources.

So, total number of threads in an entire GPU = Number of SMs \* Number of SPs per SM \* Number of Threads per SP.

**INTRODUCTION TO CUDA**

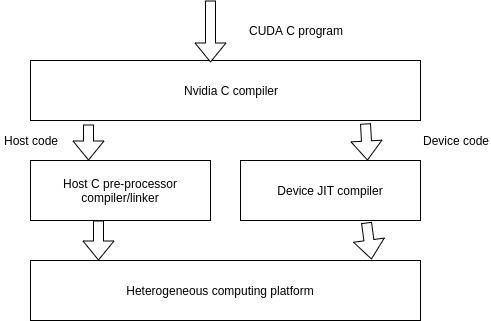
CUDA is a GPU based parallel programming platform developed by Nvidia to promote parallel use of multi threads for fast and advanced computing performance. CUDA stands for Compute Unified Device Architecture. It is an extension of C programming and therfore a person who has a liitle knowledge of C can learn CUDA comfortably.

**PROGRAM STRUCTURE OF CUDA**

A general CUDA code includes both the application of CPU and GPU. The CPU serves as the Host and the GPU serves as the Device.

Whereas the host code can be compiled by a traditional C compiler as the GCC, the device code needs a special compiler to understand the api functions that are used. For Nvidia GPUs, the compiler is called the NVCC (Nvidia C Compiler).

The device code runs on the GPU, and the host code runs on the CPU. The NVCC processes a CUDA program, and separates the host code from the device code. To accomplish this, special CUDA keywords are looked for. The code intended to run of the GPU (device code) is marked with special CUDA keywords for labelling data-parallel functions, called ‘Kernels’. The device code is further compiled by the NVCC and executed on the GPU.



**CUDA THREAD ORGANISATION**

The threads in CUDA are organised in a 3 heirarchy levels. All the threads are divided into 3-D Blocks which again are divided into 3-D Grids. The programmer has the previlage to decide the dimensions of the blocks and the grids i.e. the number to threads and their arrangement along each 3-D Block and the number of blocks and their arrangement along each 3-D Grid. Moreover, their are keywords to represent these arrangements:

**gridDim -** arrangement of blocks in a grid.

**blockDim -** arrangement of threads in a block.

**blockIdx** **-** block index parameter along all the 3 dimensions(blockIdx.x, blockIdx.y and blockIdx.z).

**threadIdx** **-** thread index parameter along all 3 dimensions(threadIdx.x, threadIdx.y, threadIdx.z).

To fix the dimensions of the grid and block, **dim3** is used.

After assigning dimensions to the grids and blocks, they are passed to the device while calling the kernel function.

**RESOURCE ASSIGNMENT TO BLOCKS**

* Execution resources are organised into Streming Multiprocessors.
* Maximum numberof threads that can be assigned to a ‘SM’ is fixed, but not the number of blocks.
* Number of threads that can run parallel on a CUDA Device is the number of SM \* maximum number of threads each SM can support.

**SYNCHRONISATION BETWEEN THREADS**

* CUDA API method: ‘\_\_syncthreads() ’ is used to synchronize threads.
* When the method is encountered in the Kernel, all threads in a block get be locked at the calling location until each of them reaches the location.
* It ensures phase synchronization i.e. all the threads of a block start executing their next phase only after they have finished the previous one.
* All the threads should undergo the ‘\_\_syncthreads() ’ function even if an ‘ if else’ statement or similar kind is used in the program. If threads take different paths, then they get blocked forever.

**THREAD SCHEDULING**

* Blocks of threads is assigned to an SM.
* SM is divided into sets of 32 threads each called a warp. Size of warp depends on implementation.
* Block is divided into warps for synchronisation.
* Warp is a unit of thread-scheduling in SMs.
* SM is composed of SPs(actual CUDA cores).
* Threads are always scheduled in a group.
* Warp consists of threads with consecutive threadIdx.x values.
* All threads in a warp follow SIMD model.

**MEMORY IN CUDA**

There are 5 types of memories used in CUDA:

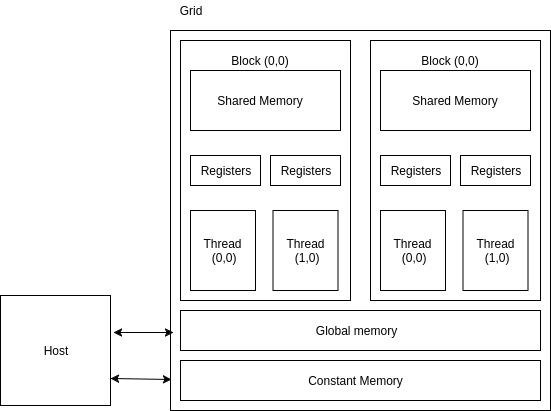
* **Global Memory –** The global memory is a high-latency memory.Main means of communicating R/W. Stores data between Host and Device. Contents visible to all threads.
* **Local Memory –** It is a thread’s unique and private chunk of global memory. Access to this is slow.
* **Constant Memory –** The constant memory can be written into and read by the host. It is used for storing data that will not change over the course of kernel execution. It is read-only accessible by the device and all the threads access te same memory location. Contents visible to all threads.
* **Shared Memory –**  Shared memory can be used for inter-thread communication. Each block has its own shared-memory. It is an on-chip memory. Contents visible to Threads belonging to the same Block.
* **Resistor Memory –** It is also an on-chip memory. No load operation is required in accessing this memory and so it is faster than shared memory.

Accessibililty of the Device code on each type of memory is be given by:

* R/W per-thread registers
* R/W per-thread local memory
* R/W per-block shared memory
* R/W per-grid global memory
* Read only per-grid constant memory

Accessibility of the Host code on memory is given by:

* R/W per-grid global and constant memory



**WRITING A CUDA PROGRAM**

* A cuda program starts with incuding all the necessary header files needed to execute any library function going to be used in the program.
* Then comes the main function which contains the host code and is executed by the CPU. This part of the program is capable of interaction with the user. All the input and output is performed within the main function and all the data and variables declared inside the main function is stored in the CPU memory which is inaccessible by GPU. GPU can only access its own memory. To execute a kernel on the GPU, the programmer needs to allocate separate memory on the GPU by writing code. The CUDA API provides specific functions for accomplishing this. Here is the flow sequence –
* Memory for device is needed to be allocated.
* After allocating memory on the device, data has to be transferred from the host memory to the device memory.
* Then the Kernel function is launched:

kernelName<<<dimGrid,dimBlock>>>(parameter1, parameter2, ...) − Launch the actual kernel.

Here, kernelName is the name of the kernel function of the device. dimGrid refers the grid dimensions. dimBlock refers to block dimensions. parameter1, parameter2, ... refers to the variables on device memory passed to the kernel function to be worked upon.

* After the kernel is executed on the device, the result has to be transferred back from the device memory to the host memory.
* The allocated memory on the device has to be freed-up. The host can access the device memory and transfer data to and from it, but not the other way round.
* Now, after the execution of the program within the kernel , the result is again served back to the user within the main function.

**KERNEL FUNCTION**

Kernel function starts with the keyword ‘\_\_global\_\_’ as this function is executed by the device and callable from the host. It is followed by the return type, the kernel name and the passed parameters.

Within the function, the device code is written which is executed by each and every thread parallely and independently.

**CUDA KEYWORDS**

* Keywords used while declaring a cuda function:

|  |  |  |
| --- | --- | --- |
|  | **EXECUTED ON THE −** | **CALLABLE FROM −** |
| **\_\_device\_\_** float function() | GPU (device) | CPU (host) |
| **\_\_global\_\_** void function() | CPU (host) | GPU (device) |
| **\_\_host\_\_**float function() | GPU (device) | GPU (device) |

* cudaMalloc() : This method is used to allocate memory on device.
* cudaMemcpy() : This method is used to copy values stored in variables from host to device.
* cudaFree() : This method is used to release objects from device memory.

**SOME PROGRAMMING APPLICATIONS OF CUDA**

**\*for the below programs, 9.0 version of CUDA and Microsoft Visual Studio 2015 has been used.\***

**MATRIX ADDITION**

#include<iostream>

#include<conio.h>

#include<stdio.h>

#include"cuda.h"

#include<cuda\_runtime\_api.h>

using namespace std;

/\*Kernel function: It takes the 3 matrices and their size as arguments. Each thread

calculates an element of resultant matrix.\*/

\_\_global\_\_ void addKernel(float \*a1, float \*b1, int n1, float \*c1)

{

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

/\*finding the sum of 2 corresponding elements from the 2 matrices represented

by a thread\*/

if(i < n1)

c1[i] = a1[i] + b1[i]; //sum of corresponding elements

}

int main(void)

{

float \*A, \*B, \*C; //Variables for matrices in Host memory

float \*a, \*b, \*c; //Variables for matrices in Device memory

int m, n; //variables for no of rows and columns of the matrices

cout<<"Enter the dimensions of the array:";

cin>>m>>n; //To enter the dimensions of matrices

int x = sizeof(float) \* n \* m; //to find the size of the matrices in bytes

//Memory allocation for host variables

A = (float\*)malloc(x);

B = (float\*)malloc(x);

C = (float\*)malloc(x);

int y = m \* n;//Calculation of total no. of elements in each matrix

//Memory allocation for device variables

cudaMalloc((void\*\*)&a, x);

cudaMalloc((void\*\*)&b, x);

cudaMalloc((void\*\*)&c, x);

//Input to the 1st array

cout<<"Enter the elements in the 1st array:"<<endl;

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

{

cin>>A[i];

}

//Input to th 2nd array

cout<<"Enter the elements in the 2nd array:"<<endl;

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

{

cin>>B[i];

}

//Copying the arrays from Host to Device

cudaMemcpy(a, A, x, cudaMemcpyHostToDevice);

cudaMemcpy(b, B, x, cudaMemcpyHostToDevice);

/\*Kernel Launch: Each matrix has dimensions m\*n and so a total of m blocks

and n threads per block are alotted in a way that each thread corresponds to

an element in the resultant matrix. Moreover, all the matrices are passed as

arguments to the kernel along with the total no. of elements in each

matrix.\*/

addKernel<<<m, n>>>(a, b, y, c);

cudaDeviceSynchronize();

//Copying the resultant array from Host to Device

cudaMemcpy(C, c, x, cudaMemcpyDeviceToHost);

//Output the resultant array

cout<<"the sum array:"<<endl;

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

{

cout<<C[i]<<endl;

}

getch();

//freeing the memory

free(A);

free(B);

free(C);

cudaFree(a);

cudaFree(b);

cudaFree(c);

return 0;

}

**MATRIX MULTIPLICAION**

#include<iostream>

#include<conio.h>

#include"cuda.h"

#include<stdio.h>

#include<cuda\_runtime\_api.h>

using namespace std;

/\*Kernel Function: All the 3 matrices along with the width(no. of elements in each row of 1st matrix) are recieved as arguments for the function. Each thread calculates one element of the product

matrix i.e. (i\*width+j)th thread calculates the ith row and jth column element of the product matrix by multiplying each element of ith row of the 1st matrix to the corresponding elements of the jth column of 2nd matrix.\*/

\_\_global\_\_ void mulKernel(float \*A1, float \*B1, float \*C1, int width)

{

int row = blockIdx.x;//row no.

int col = threadIdx.x;//column no.

if(row<width && col<width)

{

C1[row\*width + col] = 0;//initialising element to 0.

/\*To calculating the sum of the product of the corresponding elements of

the (row)th row of 1st array and (col)th column of the 2nd array inorder to

get the element of the (row)th row and (col)th column of the product

matrix\*/

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

C1[row\*width + col] += A1[row\*width + i] \* B1[i\*width + col];

}

}

int main(void)

{

//declaration of Host and Device variables

float \*A, \*B, \*C;

float \*a, \*b, \*c;

int w, am, an, bm, bn;

//Input the dimensions of the input array

cout<<"Enter the dimensions of the 1st array:";

cin>>am>>an;

cout<<"Enter the dimensions of the 2nd array:";

cin>>bm>>bn;

w = an;//width of the product matrix

//memory allocation on Host Memory

A = (float\*)malloc(sizeof(float)\*am\*an);

B = (float\*)malloc(sizeof(float)\*bm\*bn);

C = (float\*)malloc(sizeof(float)\*am\*bn);

//memory allocation on Device Memory

cudaMalloc((void\*\*)&a, sizeof(float)\*am\*an);

cudaMalloc((void\*\*)&b, sizeof(float)\*bm\*bn);

cudaMalloc((void\*\*)&c, sizeof(float)\*am\*bn);

//Input to 1st array in vector form

cout<<"Enter the 1st array:"<<endl;

for(int i=0; i<an\*am; i++)

cin>>A[i];

//Input to 2nd array in vector form

cout<<"Enter the 2nd array:"<<endl;

for(int i=0; i<bn\*bm; i++)

cin>>B[i];

//Copying input matrices from Host to Device

cudaMemcpy(a, A, sizeof(float)\*am\*an, cudaMemcpyHostToDevice);

cudaMemcpy(b, B, sizeof(float)\*bm\*bn, cudaMemcpyHostToDevice);

/\*Kernel Launch: here ‘am’ is the no. of rows of the 1st matrix and ‘bn’ is the no.

of columns of the 2nd matrix, and hence the dimensions of the product

matrix. Each Block corresponds to a row in the product matrix and each thread

corresponds to each element of the product matrix. Moreover, the 2 input

matrices, the product matrix and the width are passed as vector arguments to

the Kernel Function.\*/

mulKernel<<<am, bn>>>(a, b, c, w);

cudaDeviceSynchronize();

//Copying product matrix from Device to Host

cudaMemcpy(C, c, sizeof(float)\*am\*bn, cudaMemcpyDeviceToHost);

//Output the product matrix

cout<<"resultant array:"<<endl;

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

{

for(int j=0; j<bn; j++)

cout<<C[i\*am + j];

cout<<endl;

}

getch();

return 0;

}

**PATH USING BREADTH FIRST SEARCH(BFS)**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include<iostream>

#include<conio.h>

using namespace std;

/\*Kernel Function:

Variables passed to the Kernel:

* darr – Stores the input matrix.
* dqueu – Stores the weight calculated for each vertex.
* ddept – Stores the depth of each vertex.
* dvis – Initially stores the value for visited vertices and later the final output path.
* st- Stores the starting vertex.
* nw – Stores the total no. of vertices.
* d – Stores the value of maximum depth.

\*/

\_\_global\_\_ void myKernel(int \*darr, int \*dqueu, int \*ddept, int \*dvis, int st, int nw, int \*d)

{

int x = threadIdx.x;

int y = blockIdx.x;

ddept[st] = 0;//updating the depth of the starting vertex with 0.

dqueu[st] = 0;//Initialising the weight ‘dqueu’ for the starting vertex with 0.

dvis[st] = 1; //Marking the dvis for starting vertex as visited.

d[0]=0; //Initialising the maximum depth with 0.

//Calculating depth, weight(queue), dvis of each vertex and maximum depth

while(ddept[x]==-1)

{

if(darr[y\*nw +x] == 1)

{

if(ddept[x] == -1)

{

if(ddept[y] != -1)

{

dvis[x] = 1;

ddept[x] = ddept[y]+1;

dqueu[x] = dqueu[y]\*10 + x;

if(d[0]<ddept[x])

d[0]=ddept[x];

}

}

}

\_\_syncthreads();

if(darr[y\*nw +x] == 1)

{

if(ddept[y] == -1)

{

if(ddept[x] != -1)

{

dvis[y] = 1;

ddept[y] = ddept[x]+1;

dqueu[y] = dqueu[x]\*10 + y;

if(d[0]<ddept[y])

d[0]=ddept[y];

}

}

}

\_\_syncthreads();

}

\_\_syncthreads();

//Updating weight of vertices having more than one parent vertices.

for(int i=0; i<d[0]; i++)

if(darr[y\*nw+x] == 1 && ddept[x] != ddept[y])

{

if(ddept[x]>ddept[y] && dqueu[y]<dqueu[x]/10)

{

if(x<10)

dqueu[x]=dqueu[y]\*10 + x;

else if(x<100)

dqueu[x]=dqueu[y]\*100 + x;

else if(x<1000)

dqueu[x]=dqueu[y]\*1000 + x;

}

\_\_syncthreads();

if(ddept[y] > ddept[x] && dqueu[x] < dqueu[y]/10)

{

if(y<10)

dqueu[y] = dqueu[x]\*10 + y;

else if(y<100)

dqueu[y] = dqueu[x]\*100 + y;

else if(y<1000)

dqueu[y] = dqueu[x]\*1000 + y;

}

\_\_syncthreads();

}

\_\_syncthreads();

//Arranging the vertices in increasing order of their weights

int n1=0;

x = threadIdx.x;

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

if(dqueu[i]<dqueu[x])

n1++;

dvis[n1]=x;

\_\_syncthreads();

}

int main(void)

{

/\*

Firstly, all the required variables are declared,

Variables on the Host:

* array- Stores a matrix of size n\* n (where n is the total no. of vertices) on Host, which displays the connections between the different vertices i.e. element has a value 1 if its row no. and column no. are connected, else stores 0.
* queue – n dimensional vector that stores the weights alotted to each vertex as they are transversed by the threads.
* depth – n dimensional vector that stores the depth or heirachy level of each vertex.
* vis- n dimensional vector whose all elements are initialised to 0 at first and then , after the kernel launch , updated with different values. It is basically a vector which stores the info whether a particular vertex is visited or not and later it is overwritten by the output.
* dmax – Stores the value of maximum depth.
* start – Stores the starting vertex.
* n – Stores the total no. of vertices.

Variables on the Device:

* darray – n\*n matrix to store the values of matrix ‘array’ on Device.
* dqueue- n dimensional vector to store values for ‘queue’ on Device.
* ddepth – n dimensional vector to store the values for ‘depth’ on Device.
* dvist – n dimensional vector to store the values for vis on Device.
* dd – Stores the value of maximum depth.

\*/

//Declaration of variables

int \*array, \*queue, \*depth, \*vis, \*q, \*dmax;

int \*darray, \*dqueue, \*ddepth, \*dvist, \*dque, \*dd;

int n, start;

//To input the total no. of vertices

cout<<"Enter the no. of vertices: ";

cin>>n;

//memory allocation on Host

array = (int\*)malloc(sizeof(int)\*n\*n);

queue = (int\*)malloc(sizeof(int)\*n);

depth = (int\*)malloc(sizeof(int)\*n);

vis = (int\*)malloc(sizeof(int)\*n);

q = (int\*)malloc(sizeof(int)\*n);

dmax = (int\*)malloc(sizeof(int)\*2);

//Initialising all the elements of the input matrix to 0

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

for(int j=0; j<n; j++)

{

array[i\*n+j]=0;

}

//Input to the vertices and their connections

int c;

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

{

int x;

cout<<"Enter the no. of vertices connected with "<<i<<": ";

cin>>x;

cout<<"Enter the vertices: ";

for(int j=0; j<x; j++)

{

cin>>c;

array[i\*n+c]=1;

}

}

//Displaying the matrix representing the connected vertices

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

{

for(int j=0; j<n; j++)

cout<<array[i\*n+j]<<" ";

cout<<endl;

}

//Input to the starting vertex

cout<<"Enter the starting vertex: ";

cin>>start;

//memory allocation on Device

cudaMalloc((void\*\*)&darray,sizeof(int)\*n\*n);

cudaMalloc((void\*\*)&dqueue,sizeof(int)\*n);

cudaMalloc((void\*\*)&dvist,sizeof(int)\*n);

cudaMalloc((void\*\*)&ddepth,sizeof(int)\*n);

cudaMalloc((void\*\*)&dd,sizeof(int)\*2);

cudaMalloc((void\*\*)&dque,sizeof(int)\*n);

//Initialising the vector dvist(vector to store the visited to 0 and ddepth to -1

cudaMemset(dvist,0,sizeof(int)\*n);

cudaMemset(ddepth,-1,sizeof(int)\*n);

//Copying the input matrix from Host to Device.

cudaMemcpy(darray,array,sizeof(int)\*n\*n,cudaMemcpyHostToDevice);

//Kernel Launch

myKernel<<<n, n>>>(darray, dqueue, ddepth, dvist, start, n, dd);

cudaDeviceSynchronize();

//Copying the updated values from Device to Host.

cudaMemcpy(array,darray,sizeof(int)\*n\*n,cudaMemcpyDeviceToHost);

cudaMemcpy(queue,dqueue,sizeof(int)\*n,cudaMemcpyDeviceToHost);

cudaMemcpy(depth,ddepth,sizeof(int)\*n,cudaMemcpyDeviceToHost);

cudaMemcpy(vis,dvist,sizeof(int)\*n,cudaMemcpyDeviceToHost);

cudaMemcpy(dmax,dd,sizeof(int)\*2,cudaMemcpyDeviceToHost);

//Displaying the output

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

{

cout<<i<<" "<<depth[i]<<" ";

cout<<queue[i]<<" ";

cout<<endl;

}

cout<<”Maximum depth: ”<<dmax[0]<<endl;

cout<<”The Result of BFS: “;

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

cout<<vis[i]<<" ";

//Feeing the memory.

free(array);

free(queue);

free(depth);

free(vis);

free(dmax);

cudaFree(darray);

cudaFree(dqueue);

cudaFree(ddepth);

cudaFree(dvist);

cudaFree(dd);

getch();

}

**Explanation:**

* Initially, all the reqiured variables on host and device are declared.Then, the total no. of vertices are taken as input.
* The vertices and their connections are taken as input and simultaneously entered into an n\*n matrix where each element is either ‘1’ or ‘0’ depending on the presence of the connection between their corresponding row and column, i.e. ‘1’ if row and column are connected and ‘0’ if row and column are not connected.
* Then, the vertex from where to the start the search is taken as input. All the variables are allocated memory on host and Device respectively followed by the initialisation of the ‘vis’ and ‘depth’ for each vertex with ‘0’ and ‘-1’ respectively.
* Then, the array is copied from Host to Device.
* At the kernel launch, all the required variables on device are passed as arguments to the kernel. During the kernel launch, n blocks and n threads in each block are alloted where each thread represents one element from the input array i.e. absence or presence of a particular connection between any 2 vertices.
* Inside the Kernel, the weight and depth of the starting vertex and the maximum depth are initialised to 0.
* Then a while loop run for each thread containing ‘\_\_syncthreads()’ function which waits till the depth of all the vertices are assigned along with the calculation of the maximum depth and the weight of each vertex and marking the vertices as visited.
* Then another for loop runs to update the weights of some specific vertices which are connected to more than 1 parents so as to consider the connection with the parent with lower value.
* Then, all the vertices are arranged in ascending order of their weights and stored in the ‘dvis’ vector.
* Back in the main function, all the variables after having been modified in the Kernel, are copied back from the Device to the Host and the Result is displayed.
* At the end, all the memory is freed.

**OTHER APPLICATIONS OF CUDA**

* 3D Image Analysis
* Automobile Vision
* Bio Informatics
* Broadcast
* Computational Fluid Dynamics
* Computer Vision
* Cryptography
* Data Mining
* Mine Planning
* Network Processing
* Neural Network
* Robotic Vision/AI
* Robotic Surgery
* Satalite Data Analysis
* Seismic Imaging

**CONCLUSION**

* CUDA is a powerful parallel programming model.
* CUDA on GPUs can achieve great results on data parallel computations with a few simple performance optimization strategies.
* Structure your application and select execution configurations to maximize exploitation of the GPU’s parallel capabilities.
* Minimize CPU-GPU data tranfers.
* Coalesce global memory accesses.
* Take advantage of shared memory.
* Minimize divergent warps.
* Minimize use of low-throughput instructions.

**REFERENCES**

### “[CUDA Zone | NVIDIA Developer](https://developer.nvidia.com/cuda-zone)” by developer.nvidia.com/cuda-zone

* “CUDA tutorial” by [www.tutorialspoint.com](http://www.tutorialspoint.com).
* “CUDA” by wikipedia.org/wiki/CUDA

### “[CUDA C/C++ Basics” - Nvidia](http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf) by Cyril Zeller, Nvidia Corperation.

### “Optimiing Parallel Reduction” in CUDA by Mark Harris, Nvidia Developer Technology.