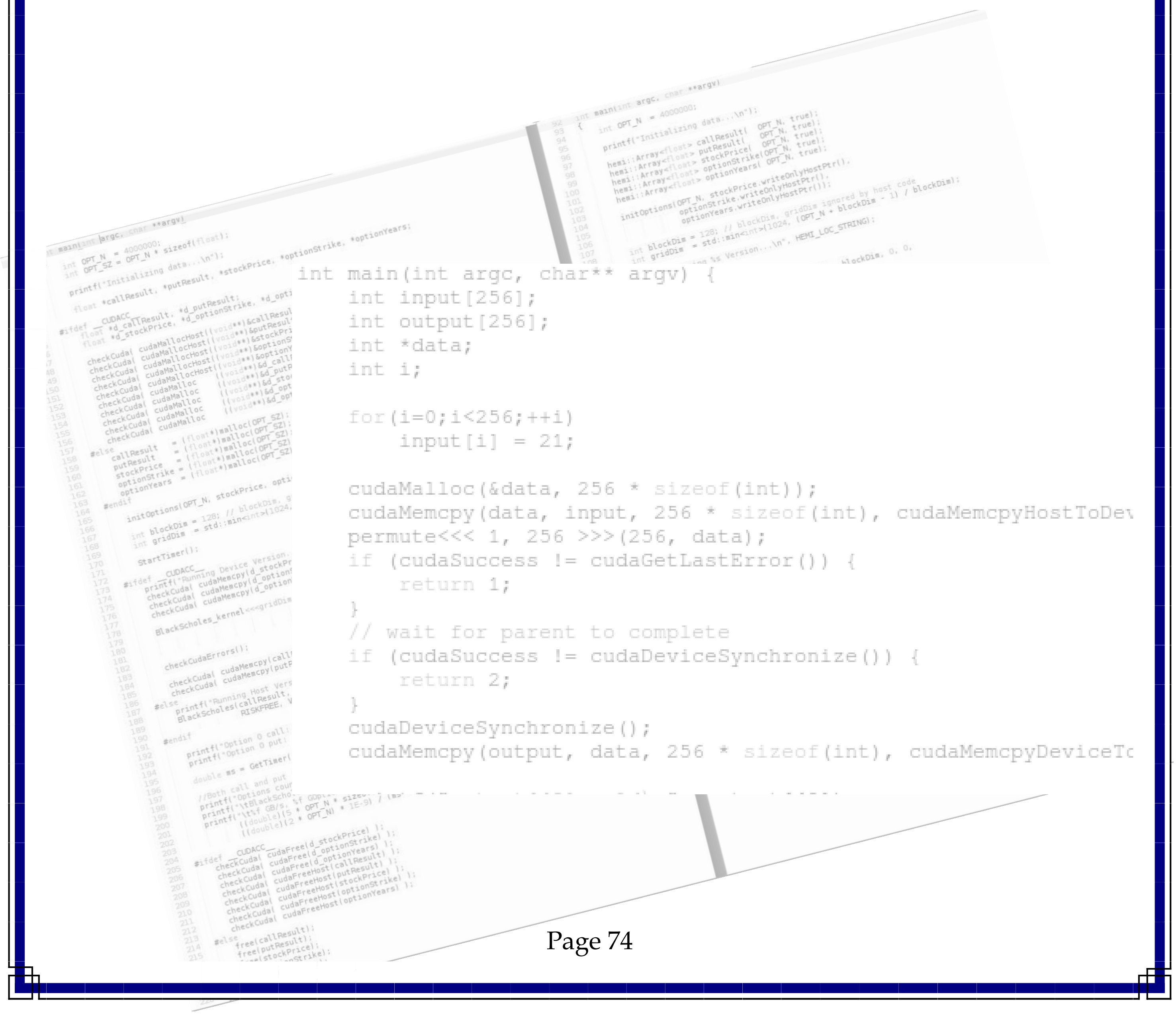
Appendix C

Program Codes



C.1. Parallel BFS

```
/**************************************************************************
  Implementing Breadth first search on CUDA based on the module provided by
  Rodinia benchmark.
 **************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cuda.h>

#define MAX_THREADS_PER_BLOCK 512

int no_of_nodes;
int edge_list_size;
FILE *fp;

//Structure to hold a node information
struct Node
{
	int starting;
	int no_of_edges;
};


__global__ void
Kernel( Node* g_graph_nodes, int* g_graph_edges, bool* g_graph_mask, bool*
g_updating_graph_mask, bool *g_graph_visited, int* g_cost, int no_of_nodes)
{
	int tid = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;
	if( tid<no_of_nodes && g_graph_mask[tid])
	{
		g_graph_mask[tid]=false;
		for(int i=g_graph_nodes[tid].starting;
i<(g_graph_nodes[tid].no_of_edges + g_graph_nodes[tid].starting); i++)
		{
			int id = g_graph_edges[i];
			if(!g_graph_visited[id])
			{
				g_cost[id]=g_cost[tid]+1;
				g_updating_graph_mask[id]=true;
			}
		}
	}
}
```

```
__global__ void
Kernel2( bool* g_graph_mask, bool *g_updating_graph_mask, bool* g_graph_visited,
bool *g_over, int no_of_nodes)
{
	int tid = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;
	if( tid<no_of_nodes && g_updating_graph_mask[tid])
	{

		g_graph_mask[tid]=true;
		g_graph_visited[tid]=true;
		*g_over=true;
		g_updating_graph_mask[tid]=false;
	}
}

////////////////////////////////////////////////////////////////////////////////
//Apply BFS using CUDA
////////////////////////////////////////////////////////////////////////////////
void BFS()
{
	char *input_f;
	//input_f = "B_graph4096.txt"; //argv[1];
	input_f = (char*)malloc(sizeof(char)*100);
	printf("File path:");
	scanf("%s",input_f);
	printf("Reading File\n");
	//Read in Graph from a file
	fp = fopen(input_f,"r");
	if(!fp)
	{
		printf("Error Reading graph file\n");
		return;
	}

	int source = 0;

	fscanf(fp,"%d",&no_of_nodes);

	int num_of_blocks = 1;
	int num_of_threads_per_block = no_of_nodes;

	//Make execution Parameters according to the number of nodes
	//Distribute threads across multiple Blocks if necessary
	if(no_of_nodes>MAX_THREADS_PER_BLOCK)
	{
		num_of_blocks = 
(int)ceil(no_of_nodes/(double)MAX_THREADS_PER_BLOCK);
		num_of_threads_per_block = MAX_THREADS_PER_BLOCK;
```

```
	}

	// allocate host memory
	Node* h_graph_nodes = (Node*) malloc(sizeof(Node)*no_of_nodes);
	bool *h_graph_mask = (bool*) malloc(sizeof(bool)*no_of_nodes);
	bool *h_updating_graph_mask = (bool*) malloc(sizeof(bool)*no_of_nodes);
	bool *h_graph_visited = (bool*) malloc(sizeof(bool)*no_of_nodes);

	int start, edgeno;
	// initalize the memory
	for( unsigned int i = 0; i < no_of_nodes; i++)
	{
		fscanf(fp,"%d %d",&start,&edgeno);
		h_graph_nodes[i].starting = start;
		h_graph_nodes[i].no_of_edges = edgeno;
		h_graph_mask[i]=false;
		h_updating_graph_mask[i]=false;
		h_graph_visited[i]=false;
	}

	//read the source node from the file
	fscanf(fp,"%d",&source);
//	source=0;

	//set the source node as true in the mask
	h_graph_mask[source]=true;
	h_graph_visited[source]=true;

	fscanf(fp,"%d",&edge_list_size);

	int id,cost;
	int* h_graph_edges = (int*) malloc(sizeof(int)*edge_list_size);
	for(int i=0; i < edge_list_size ; i++)
	{
		fscanf(fp,"%d",&id);
		fscanf(fp,"%d",&cost);
		h_graph_edges[i] = id;
	}

	if(fp)
		fclose(fp);

	printf("Read File\n");

	//Copy the Node list to device memory
	Node* d_graph_nodes;
	cudaMalloc( (void**) &d_graph_nodes, sizeof(Node)*no_of_nodes) ;
```

```
	cudaMemcpy( d_graph_nodes, h_graph_nodes, sizeof(Node)*no_of_nodes,
cudaMemcpyHostToDevice) ;

	//Copy the Edge List to device Memory
	int* d_graph_edges;
	cudaMalloc( (void**) &d_graph_edges, sizeof(int)*edge_list_size) ;
	cudaMemcpy( d_graph_edges, h_graph_edges, sizeof(int)*edge_list_size,
cudaMemcpyHostToDevice) ;

	//Copy the Mask to device memory
	bool* d_graph_mask;
	cudaMalloc( (void**) &d_graph_mask, sizeof(bool)*no_of_nodes) ;
	cudaMemcpy( d_graph_mask, h_graph_mask, sizeof(bool)*no_of_nodes,
cudaMemcpyHostToDevice) ;

	bool* d_updating_graph_mask;
	cudaMalloc( (void**) &d_updating_graph_mask, sizeof(bool)*no_of_nodes) ;
	cudaMemcpy( d_updating_graph_mask, h_updating_graph_mask,
sizeof(bool)*no_of_nodes, cudaMemcpyHostToDevice) ;

	//Copy the Visited nodes array to device memory
	bool* d_graph_visited;
	cudaMalloc( (void**) &d_graph_visited, sizeof(bool)*no_of_nodes) ;
	cudaMemcpy( d_graph_visited, h_graph_visited, sizeof(bool)*no_of_nodes,
cudaMemcpyHostToDevice) ;

	// allocate mem for the result on host side
	int* h_cost = (int*) malloc( sizeof(int)*no_of_nodes);
	for(int i=0;i<no_of_nodes;i++)
		h_cost[i]=-1;
	h_cost[source]=0;

	// allocate device memory for result
	int* d_cost;
	cudaMalloc( (void**) &d_cost, sizeof(int)*no_of_nodes);
	cudaMemcpy( d_cost, h_cost, sizeof(int)*no_of_nodes,
cudaMemcpyHostToDevice) ;

	//make a bool to check if the execution is over
	bool *d_over;
	cudaMalloc( (void**) &d_over, sizeof(bool));

	printf("Copied Everything to GPU memory\n");

	// setup execution parameters
	dim3  grid( num_of_blocks, 1, 1);
	dim3  threads( num_of_threads_per_block, 1, 1);
```

```
	int k=0;
	printf("Start traversing the tree\n");
	bool stop;
	//Call the Kernel untill all the elements of Frontier are not false
	do
	{
		//if no thread changes this value then the loop stops
		stop=false;
		cudaMemcpy( d_over, &stop, sizeof(bool), cudaMemcpyHostToDevice) ;
		Kernel<<< grid, threads, 0 >>>( d_graph_nodes, d_graph_edges, 
d_graph_mask, d_updating_graph_mask, d_graph_visited, d_cost, no_of_nodes);
		// check if kernel execution generated and error


		Kernel2<<< grid, threads, 0 >>>( d_graph_mask, 
d_updating_graph_mask, d_graph_visited, d_over, no_of_nodes);
		// check if kernel execution generated and error


		cudaMemcpy( &stop, d_over, sizeof(bool), cudaMemcpyDeviceToHost) ;
		k++;
	}
	while(stop);


	printf("Kernel Executed %d times\n",k);

	// copy result from device to host
	cudaMemcpy( h_cost, d_cost, sizeof(int)*no_of_nodes, 
cudaMemcpyDeviceToHost) ;

	//Store the result into a file
	FILE *fpo = fopen("result.txt","w");
	for(int i=0;i<no_of_nodes;i++)
		fprintf(fpo,"%d) cost:%d\n",i,h_cost[i]);
	fclose(fpo);
	printf("Result stored in result.txt\n");


	// cleanup memory
	free( h_graph_nodes);
	free( h_graph_edges);
	free( h_graph_mask);
	free( h_updating_graph_mask);
	free( h_graph_visited);
	free( h_cost);
	cudaFree(d_graph_nodes);
	cudaFree(d_graph_edges);
```

```
	cudaFree(d_graph_mask);
	cudaFree(d_updating_graph_mask);
	cudaFree(d_graph_visited);
	cudaFree(d_cost);
}

int main( int argc, char** argv)
{
	no_of_nodes=0;
	edge_list_size=0;
	BFS();
}
```

C.2. Parallel st-CON

```
/*****************************************************************************
  Implementing s-t Connectivity algorithm on CUDA
*****************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cuda.h>

#define MAX_THREADS_PER_BLOCK 512

int no_of_nodes;
int edge_list_size;
FILE *fp;

//Structure to hold a node information
struct Node
{
      int starting;
      int no_of_edges;
};

__device__ int rf=0;
__device__ int gf=0;
__device__ bool d_stop=false;

__global__ void
Kernel( Node* g_graph_nodes, int* g_graph_edges, bool* g_graph_mask,
      bool* g_updating_graph_mask, bool *g_Red_graph_visited,
      bool *g_Green_graph_visited, bool *g_Red_updating_graph_visited,
      bool *g_Green_updating_graph_visited, int* g_cost, int no_of_nodes,
      bool* g_over)
{
      int tid = blockIdx.x*blockDim.x + threadIdx.x;
      if( tid<no_of_nodes && g_graph_mask[tid])
      {
            g_graph_mask[tid]=false;
            for(int i=g_graph_nodes[tid].starting;
      i<(g_graph_nodes[tid].no_of_edges + g_graph_nodes[tid].starting); i++)
            {
                  int nid = g_graph_edges[i];
                  if(g_Red_graph_visited[nid] || g_Green_graph_visited[nid])
                  {
                        if(g_Red_graph_visited[tid] &&
                              g_Green_graph_visited[nid])
```

```
				{
					rf = g_cost[tid]+1;
					*g_over = true;
					d_stop=true;
				}
				if(g_Green_graph_visited[tid] &&
					g_Red_graph_visited[nid])
				{
					*g_over=true;
					d_stop=true;
				}

			}
			else
			{
				if(g_Green_graph_visited[tid])
					g_Green_updating_graph_visited[nid] = true;
				if(g_Red_graph_visited[tid])
					g_Red_updating_graph_visited[nid] = true;

				g_updating_graph_mask[nid]=true;
				g_cost[nid] = g_cost[tid]+1;
			}

		}
	}
}

__global__ void
Kernel2( bool* g_graph_mask, bool * g_updating_graph_mask, bool *
	g_Red_graph_visited,bool * g_Green_graph_visited, bool *
	g_Red_updating_graph_visited, bool * g_Green_updating_graph_visited, bool
	*g_over, int no_of_nodes,int* g_cost)
{
	int tid = blockIdx.x*blockDim.x + threadIdx.x;
	if( tid<no_of_nodes && g_updating_graph_mask[tid] && !d_stop)
	{
		//printf("\n%dT",tid);
		g_graph_mask[tid]=true;
		if(g_Red_updating_graph_visited[tid])
		{
			g_Red_graph_visited[tid] = true;
			rf = g_cost[tid];
			g_Red_updating_graph_visited[tid] = false;
		}
		if(g_Green_updating_graph_visited[tid])
		{
			g_Green_graph_visited[tid] = true;
```

```
                        gf = g_cost[tid];
                        g_Green_updating_graph_visited[tid] = false;
                }
                g_updating_graph_mask[tid]=false;

                if(g_Green_graph_visited[tid] && g_Red_graph_visited[tid])
                        *g_over=true;
        }
}

__global__
void dummy(int *len){
        //printf("R%dG%d",rf,gf);
        *len = rf+gf;
}

////////////////////////////////////////////////////////////////////////////////
//Apply BFS on a Graph using CUDA
////////////////////////////////////////////////////////////////////////////////
void stCON() {
        char *input_f;
        input_f = (char*)malloc(sizeof(char)*100);
        printf("File path:");
        scanf("%s",input_f);

        printf("Reading File\n");
        //Read in Graph from a file
        fp = fopen(input_f,"r");
        if(!fp)
        {
                printf("Error Reading graph file\n");
                return;
        }

        int source = 0;
        int terminal = 0; //terminal  (Appended)
        //int h_R_length = 0;   //Rf  (Appended)
        //int h_G_length = 0;   //Gf  (Appended)

        fscanf(fp,"%d",&no_of_nodes);

        int num_of_blocks = 1;
        int num_of_threads_per_block = no_of_nodes;

        //Make execution Parameters according to the number of nodes
        //Distribute threads across multiple Blocks if necessary
        if(no_of_nodes>MAX_THREADS_PER_BLOCK)
        {
```

```
		num_of_blocks = 
(int)ceil(no_of_nodes/(double)MAX_THREADS_PER_BLOCK); 
		num_of_threads_per_block = MAX_THREADS_PER_BLOCK; 
	}

	// allocate host memory
	Node* h_graph_nodes = (Node*) malloc(sizeof(Node)*no_of_nodes);
	bool *h_graph_mask = (bool*) malloc(sizeof(bool)*no_of_nodes);
	//Fa
	bool *h_updating_graph_mask = (bool*) malloc(sizeof(bool)*no_of_nodes);
	//Fva
	bool *h_Red_graph_visited = (bool*) malloc(sizeof(bool)*no_of_nodes);
	//Ra  (Appended)
	bool *h_Green_graph_visited = (bool*) malloc(sizeof(bool)*no_of_nodes);
	//Ga  (Appended)
	bool *h_Red_updating_graph_visited = (bool*) 
malloc(sizeof(bool)*no_of_nodes);	//Rva  (Appended)
	bool *h_Green_updating_graph_visited = (bool*) 
malloc(sizeof(bool)*no_of_nodes);	//Gva  (Appended)

	int start, edgeno;   
	// initalize the memory
	for( unsigned int i = 0; i < no_of_nodes; i++) 
	{
		fscanf(fp,"%d %d",&start,&edgeno);
		h_graph_nodes[i].starting = start;
		h_graph_nodes[i].no_of_edges = edgeno;
		h_graph_mask[i]=false;
		h_updating_graph_mask[i]=false;
		h_Red_graph_visited[i]=false; //Ra=false (Appended)
		h_Green_graph_visited[i]=false;		//Ga=false (Appended)
		h_Red_updating_graph_visited[i]=false;		//Rva=false 
(Appended)
		h_Green_updating_graph_visited[i]=false;  //Gva=false (Appended)
	}

	//read the source node from the file
	fscanf(fp,"%d",&source);
//	source=0;
	fscanf(fp,"%d",&terminal);	//take input terminal from file (Appended)

	//set the source node as true in the mask
	h_graph_mask[source]=true;
	h_graph_mask[terminal]=true;  //Added line
	h_Red_graph_visited[source]=true;   //Appended line (Ra)
	h_Green_graph_visited[terminal]=true;	//Apended line (Ga)

	fscanf(fp,"%d",&edge_list_size);
```

```
	int id,cost;
	int* h_graph_edges = (int*) malloc(sizeof(int)*edge_list_size);
	for(int i=0; i < edge_list_size ; i++)
	{
		fscanf(fp,"%d",&id);
		fscanf(fp,"%d",&cost);	//needed because of the format of the file
		h_graph_edges[i] = id;
	}

	if(fp)
		fclose(fp);

	printf("Read File\n");

	//Copy the Node list to device memory
	Node* d_graph_nodes;
	cudaMalloc( (void**) &d_graph_nodes, sizeof(Node)*no_of_nodes) ;
	cudaMemcpy( d_graph_nodes, h_graph_nodes, sizeof(Node)*no_of_nodes, 
cudaMemcpyHostToDevice) ;

	//Copy the Edge List to device Memory
	int* d_graph_edges;
	cudaMalloc( (void**) &d_graph_edges, sizeof(int)*edge_list_size) ;
	cudaMemcpy( d_graph_edges, h_graph_edges, sizeof(int)*edge_list_size, 
cudaMemcpyHostToDevice) ;

	//Copy the Mask to device memory
	bool* d_graph_mask;
	cudaMalloc( (void**) &d_graph_mask, sizeof(bool)*no_of_nodes) ;
	cudaMemcpy( d_graph_mask, h_graph_mask, sizeof(bool)*no_of_nodes, 
cudaMemcpyHostToDevice) ;

	bool* d_updating_graph_mask;
	cudaMalloc( (void**) &d_updating_graph_mask, sizeof(bool)*no_of_nodes) ;
	cudaMemcpy( d_updating_graph_mask, h_updating_graph_mask, 
sizeof(bool)*no_of_nodes, cudaMemcpyHostToDevice) ;

	//Copy the Visited nodes array to device memory (Appended)
	bool* d_Red_graph_visited;
	cudaMalloc( (void**) &d_Red_graph_visited, sizeof(bool)*no_of_nodes) ;
	cudaMemcpy( d_Red_graph_visited, h_Red_graph_visited, 
sizeof(bool)*no_of_nodes, cudaMemcpyHostToDevice) ;

	//Copy the Visited nodes array to device memory (Appended)
	bool* d_Green_graph_visited;
	cudaMalloc( (void**) &d_Green_graph_visited, sizeof(bool)*no_of_nodes) ;
```

```
	cudaMemcpy( d_Green_graph_visited, h_Green_graph_visited, 
sizeof(bool)*no_of_nodes, cudaMemcpyHostToDevice) ;

	//Copy the Visited nodes array to device memory (Appended)
	bool* d_Red_updating_graph_visited;
	cudaMalloc( (void**) &d_Red_updating_graph_visited, 
sizeof(bool)*no_of_nodes) ;
	cudaMemcpy( d_Red_updating_graph_visited, h_Red_updating_graph_visited, 
sizeof(bool)*no_of_nodes, cudaMemcpyHostToDevice) ;

	//Copy the Visited nodes array to device memory (Appended)
	bool* d_Green_updating_graph_visited;
	cudaMalloc( (void**) &d_Green_updating_graph_visited, 
sizeof(bool)*no_of_nodes) ;
	cudaMemcpy( d_Green_updating_graph_visited, 
h_Green_updating_graph_visited, sizeof(bool)*no_of_nodes, 
cudaMemcpyHostToDevice) ;

	// allocate mem for the result on host side
	int* h_cost = (int*) malloc( sizeof(int)*no_of_nodes);
	for(int i=0;i<no_of_nodes;i++)
		h_cost[i]=-1;
	h_cost[source]=0;
	h_cost[terminal]=0;		//(Appended)

	// allocate device memory for result
	int* d_cost;
	cudaMalloc( (void**) &d_cost, sizeof(int)*no_of_nodes);
	cudaMemcpy( d_cost, h_cost, sizeof(int)*no_of_nodes, 
cudaMemcpyHostToDevice) ;

	//make a bool to check if the execution is over
	bool *d_over;
	cudaMalloc( (void**) &d_over, sizeof(bool));

	printf("Copied Everything to GPU memory\n");

	// setup execution parameters
	dim3  grid( num_of_blocks, 1, 1);
	dim3  threads( num_of_threads_per_block, 1, 1);

	int k=0;
	printf("Start traversing the tree\n");
	bool stop=false;
	cudaMemcpy( d_over, &stop, sizeof(bool), cudaMemcpyHostToDevice) ;
	//Call the Kernel untill all the elements of Frontier are not false
	do
	{
```

```
		Kernel<<< grid, threads, 0 >>>(d_graph_nodes, d_graph_edges,
			d_graph_mask, d_updating_graph_mask,
			d_Red_graph_visited, d_Green_graph_visited,
			d_Red_updating_graph_visited, d_Green_updating_graph_visited,
			d_cost, no_of_nodes, d_over);
		// check if kernel execution generated and error

		Kernel2<<< grid, threads, 0 >>>(d_graph_mask, d_updating_graph_mask,
			d_Red_graph_visited, d_Green_graph_visited,
			d_Red_updating_graph_visited, d_Green_updating_graph_visited,
			d_over, no_of_nodes,d_cost);
		// check if kernel execution generated and error


		cudaMemcpy( &stop, d_over, sizeof(bool), cudaMemcpyDeviceToHost) ;
		k++;
	}
	while(!stop);


	printf("Kernel Executed %d times\n",k);




	// copy result from device to host
	cudaMemcpy( h_cost, d_cost, sizeof(int)*no_of_nodes,
cudaMemcpyDeviceToHost) ;

	int *d_len,len;
	cudaMalloc( (void**) &d_len, sizeof(int));
	dummy<<<1,1>>>(d_len);
	cudaMemcpy( &len, d_len, sizeof(int), cudaMemcpyDeviceToHost);
	printf("\nlength == %d",len);



	//Store the result into a file
	FILE *fpo = fopen("result.txt","w");
	for(int i=0;i<no_of_nodes;i++)
		fprintf(fpo,"%d) cost:%d\n",i,h_cost[i]);
	fclose(fpo);
	printf("Result stored in result.txt\n");


	// cleanup memory
	free( h_graph_nodes);
	free( h_graph_edges);
	free( h_graph_mask);
```

```
      free( h_updating_graph_mask);
      free( h_Red_graph_visited);         //(Appended)
      free( h_Green_graph_visited); //(Appended)
      free( h_Red_updating_graph_visited);      //(Appended)
      free( h_Green_updating_graph_visited);    //(Appended)
      free( h_cost);
      cudaFree(d_graph_nodes);
      cudaFree(d_graph_edges);
      cudaFree(d_graph_mask);
      cudaFree(d_updating_graph_mask);
      cudaFree(d_Red_graph_visited);            //(Appended)
      cudaFree(d_Green_graph_visited);    //(Appended)
      cudaFree(d_Red_updating_graph_visited);   //(Appended)
      cudaFree(d_Green_updating_graph_visited); //(Appended)

//    cudaFree(d_graph_visited);
      cudaFree(d_cost);
      cudaFree(d_len);
}

int main( int argc, char** argv) {
      no_of_nodes=0;
      edge_list_size=0;
      stCON();
}
```

C.3. Parallel SSSP

```
/******************************************************************************
 Implementing Single Source Shortest Path on CUDA
******************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <assert.h>

#define MAX_COST 10000000

unsigned int no_of_nodes;
unsigned int edge_list_size;
FILE *fp;

__global__ void
DijkastraKernel1(unsigned int* g_graph_nodes, unsigned int* g_graph_edges,
      unsigned int* g_graph_weights, unsigned int* g_graph_updating_cost, bool*
      g_graph_mask, unsigned int* g_cost , unsigned int no_of_nodes,
      unsigned int edge_list_size)
{
      unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
      unsigned int i,id;
      unsigned int end = edge_list_size;
      if(tid<no_of_nodes && g_graph_mask[tid])
      {
      if(tid < no_of_nodes-1)
            end = g_graph_nodes[tid+1];
      for(i = g_graph_nodes[tid]; i< end; i++)
      {
            id = g_graph_edges[i];
            atomicMin(&g_graph_updating_cost[id],
                        g_cost[tid]+g_graph_weights[i]);
      }
      g_graph_mask[tid]=false;
      }
}

__global__ void
DijkastraKernel2(unsigned int* g_graph_nodes, unsigned int* g_graph_edges,
      unsigned int* g_graph_weights, unsigned int* g_graph_updating_cost, bool*
      g_graph_mask, unsigned int* g_cost , bool *d_finished, unsigned int
      no_of_nodes, unsigned int edge_list_size)
{
      unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
```

```
	if(tid<no_of_nodes && g_cost[tid] > g_graph_updating_cost[tid])
		{
		g_cost[tid] = g_graph_updating_cost[tid];
		g_graph_mask[tid] = true;
		*d_finished = true;
		}
	if(tid<no_of_nodes)
	g_graph_updating_cost[tid] = g_cost[tid];
}

////////////////////////////////////////////////////////////////////////////////
//Apply Shortest Path on a Graph using CUDA
////////////////////////////////////////////////////////////////////////////////
void Dijkstra()
{
	printf("Reading File\n");
	fp = fopen("g.txt","r");
	if(!fp)
	{
		printf("Error Reading graph file\n");
		return;
	}

	unsigned int source = 0;

	fscanf(fp,"%d",&no_of_nodes);
	printf("No of Nodes: %d\n",no_of_nodes);

	cudaDeviceProp dev;
	cudaGetDeviceProperties(&dev,0);
	//printf("thread = %d",dev.maxThreadsPerBlock);

	unsigned int MAX_THREADS_PER_BLOCK;
	unsigned int num_of_blocks = 1;
	unsigned int num_of_threads_per_block = no_of_nodes;
	MAX_THREADS_PER_BLOCK = dev.maxThreadsPerBlock;

	//Make execution Parameters according to the number of nodes
	//Distribute threads across multiple Blocks if necessary
	if(no_of_nodes>MAX_THREADS_PER_BLOCK)
	{
	num_of_blocks = (unsigned int)ceil(no_of_nodes
	                  /(double)MAX_THREADS_PER_BLOCK);
	num_of_threads_per_block = MAX_THREADS_PER_BLOCK;
	}

	// allocate host memory
```

```
    unsigned int* h_graph_nodes = (unsigned int*) malloc(sizeof(unsigned
int)*no_of_nodes);
    bool *h_graph_mask = (bool*) malloc(sizeof(bool)*no_of_nodes);
    unsigned int *h_graph_updating_cost = (unsigned int*) malloc(sizeof(unsigned
int)*no_of_nodes);

    unsigned int start, edgeno;
    // initalize the memory
      unsigned int no=0;
    for(  unsigned int i = 0; i < no_of_nodes; i++)
    {
            fscanf(fp,"%d %d",&start,&edgeno);
            if(edgeno>100)
                  no++;
        h_graph_nodes[i] = start;
            h_graph_updating_cost[i] = MAX_COST;
        h_graph_mask[i]=false;
    }

    //read the source unsigned int from the file
    fscanf(fp,"%d",&source);
printf("Source %d\n",source);

    //set the source unsigned int as true in the mask
    h_graph_mask[source]=true;
      //h_graph_counter[source]=0;

    fscanf(fp,"%d",&edge_list_size);

    unsigned int id;
    unsigned int* h_graph_edges = (unsigned int*) malloc(sizeof(unsigned
int)*edge_list_size);
    unsigned int* h_graph_weights = ( unsigned int*) malloc(sizeof( unsigned
int)*edge_list_size);

    unsigned int i;
    for(i=0; i < edge_list_size ; i++)
    {
            fscanf(fp,"%d",&id);
            h_graph_edges[i] = id;
            fscanf(fp,"%d",&id);
            h_graph_weights[i] = id;

    }
      if(fp)
      fclose(fp);

      printf("Read File\n");
```

```
      //Copy the unsigned int list to device memory
    unsigned int* d_graph_nodes;
    cudaMalloc( (void**) &d_graph_nodes, sizeof(unsigned int)*no_of_nodes);
    cudaMemcpy( d_graph_nodes, h_graph_nodes, sizeof(unsigned int)*no_of_nodes,
cudaMemcpyHostToDevice);

      //Copy the Edge List to device Memory
      unsigned int* d_graph_edges;
    cudaMalloc( (void**) &d_graph_edges, sizeof(unsigned int)*edge_list_size);
    cudaMemcpy( d_graph_edges, h_graph_edges, sizeof(unsigned
int)*edge_list_size, cudaMemcpyHostToDevice);

       unsigned int* d_graph_weights;
    cudaMalloc( (void**) &d_graph_weights, sizeof( unsigned
int)*edge_list_size);
    cudaMemcpy( d_graph_weights, h_graph_weights, sizeof( unsigned
int)*edge_list_size, cudaMemcpyHostToDevice);

    //Copy the Mask to device memory
    bool* d_graph_mask;
    cudaMalloc( (void**) &d_graph_mask, sizeof(bool)*no_of_nodes);
    cudaMemcpy( d_graph_mask, h_graph_mask, sizeof(bool)*no_of_nodes,
cudaMemcpyHostToDevice);

    // allocate mem for the result on host side
      unsigned int* h_cost = (unsigned int*) malloc( sizeof(unsigned
int)*no_of_nodes);
      for(unsigned int i=0;i<no_of_nodes;i++)
      h_graph_updating_cost[i] = h_cost[i] = MAX_COST;
      h_cost[source]=0;
      // allocate device memory for result
    unsigned int* d_cost;
    cudaMalloc( (void**) &d_cost, sizeof(unsigned int)*no_of_nodes);
    cudaMemcpy( d_cost, h_cost, sizeof(unsigned int)*no_of_nodes,
cudaMemcpyHostToDevice);

      unsigned int* d_graph_updating_cost;
    cudaMalloc( (void**) &d_graph_updating_cost, sizeof(unsigned
int)*no_of_nodes);
    cudaMemcpy( d_graph_updating_cost, h_graph_updating_cost, sizeof(unsigned
int)*no_of_nodes, cudaMemcpyHostToDevice);

    //make a bool to check if the execution is over

      bool *d_finished;
      bool finished;
      cudaMalloc( (void**) &d_finished, sizeof(bool));
```

```
    // setup execution parameters
    dim3  grid( num_of_blocks, 1, 1);
    dim3  threads( num_of_threads_per_block, 1, 1);

      unsigned int k=0;

      do
      {
            DijkastraKernel1<<< grid, threads, 0 >>>( d_graph_nodes,
                  d_graph_edges, d_graph_weights, d_graph_updating_cost,
                  d_graph_mask, d_cost, no_of_nodes, edge_list_size);
            cudaDeviceSynchronize();
            k++;
            finished=false;
            cudaMemcpy( d_finished, &finished, sizeof(bool),
cudaMemcpyHostToDevice);
            DijkastraKernel2<<< grid, threads, 0 >>>( d_graph_nodes,
                  d_graph_edges, d_graph_weights, d_graph_updating_cost,
                  d_graph_mask, d_cost, d_finished, no_of_nodes, edge_list_size);
            cudaDeviceSynchronize();
            cudaMemcpy( &finished, d_finished, sizeof(bool),
cudaMemcpyDeviceToHost);
      }
      while(finished);


    // copy result from device to host
    cudaMemcpy( h_cost, d_cost, sizeof(unsigned int)*no_of_nodes,
cudaMemcpyDeviceToHost);
      cudaDeviceSynchronize();

      //Store the result unsigned into a file
      FILE *fpo = fopen("result.txt","w");
      for(unsigned int i=0;i<no_of_nodes;i++)
      fprintf(fpo,"%d) cost:%d\n",i,h_cost[i]);
      fclose(fpo);
      printf("Result stored in result.txt\n");


    // cleanup memory
    free( h_graph_nodes);
    free( h_graph_edges);
    free( h_graph_mask);
    free( h_graph_weights);
      free( h_graph_updating_cost);
    free( h_cost);
    cudaFree(d_graph_nodes);
    cudaFree(d_graph_edges);
```

```
    cudaFree(d_graph_mask);
    cudaFree(d_graph_weights);
      cudaFree(d_graph_updating_cost);
    cudaFree(d_cost);
      cudaFree(d_finished);
}

int main( int argc, char** argv)
{
      no_of_nodes=0;
      edge_list_size=0;
      Dijkstra();
}
```

C.4. Parallel 8-Puzzle Solver

```
/*****************************************************************************
 Implementing parallel 8-puzzle solver on CUDA
*****************************************************************************/
#include <stdio.h>
#include <assert.h>

#define LEN 9
#define SIZE 362880 //9!

#define CUDA_CHECK_RETURN(value) {                                          \
      cudaError_t _m_cudaStat = value;                                      \
      if (_m_cudaStat != cudaSuccess) {                                     \
            fprintf(stderr, "Error %s at line %d in file %s\n",             \
                        cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__); \
            exit(1);                                                        \
      } }

enum Move{UP, DOWN, LEFT, RIGHT};

char s[LEN+1], g[LEN+1];
bool *frontier, *Ufrontier;
int *visited;

__device__ char *start, *goal;
__device__ int sIndex, gIndex;
__device__ char *state, *charH, *charT;

__device__
int fact(int n){ //computes factorial
      int x=1;
      for (int i = 1; i <= n; i++) {
            x *= i;
      }
      return x;
}

///////////////////////////////////////////////////////////////////////////////
//Implementing minimal perfect hashing
//This will generate the index/rank of a given permutation
///////////////////////////////////////////////////////////////////////////////
__device__
int getHash(char *n,int tid){ //implements perfect hashing
      int h=0;
      char *num;
      num = &charH[tid*9];
```

```
	for (int i = 0; i < LEN; ++i) {
		num[i] = n[i] - '0';
	}

	int f=LEN;
	for (int i = 0; i < LEN ; ++i) {
		f--;
		if(num[i] > 0)
			h += num[i]*fact(f);
		for (int j = i+1; j < LEN; ++j) {
			if(num[j] > num[i]){
				num[j]--;
			}
		}
	}
	return h;
}

__global__
void init(char *s, char *g, int *visited, bool *frontier, bool *Ufrontier, char
		*h, char *t, char *st){
	start=s;
	goal=g;

	charH = h
	if(charH==NULL){
		printf("charH NULL");
		assert(0);
	}
	charT = t;
	if(charT==NULL){
		printf("charT NULL");
		assert(0);
	}
	state = st;
	if(state==NULL){
		printf("state NULL");
		assert(0);
	}

	sIndex=getHash(s,0);
	int offset = sIndex*9;
	for (int i = 0; i < LEN; ++i) {
		state[offset+i] = s[i];
	}

	gIndex=getHash(g,0);
```

```
	if(visited==NULL){
		printf("visited NULL");
		assert(0);
	}
	if(frontier==NULL){
		printf("frontier NULL");
		assert(0);
	}
	if(Ufrontier==NULL){
		printf("Ufrontier NULL");
		assert(0);
	}
}

__global__
void clean(int *visited, bool *frontier, bool *Ufrontier){
	int tid = blockIdx.x * blockDim.x + threadIdx.x;
	if(tid < SIZE){
		visited[tid] = -1;
		frontier[tid] = false;
		Ufrontier[tid] = false;
	}
}

__device__
char* swap(char *c, int pos, int p, int offset){
	char *a;
	a = charT+offset;//(char*)malloc(sizeof(char)*LEN);
	if(a==NULL){
		printf("a NULL");
		assert(0);
	}
	for (int i = 0; i < LEN; ++i) {
		a[i] = c[offset+i];
	}
	int x=pos;
	int y=pos+p;
	char tmp = a[x];
	a[x] = a[y];
	a[y] = tmp;
	return a;
}

__device__
char* move(char *s, int pos, Move m, int tid){
	int offset = tid*9;
	int i,j;
```

```
        i=pos/3;
        j=pos%3;
        switch(m){
              case UP:
                    if(i==0)
                          return NULL;
                    return swap(s, pos, -3, offset);
              case DOWN:
                    if(i==2)
                          return NULL;
                    return swap(s, pos, 3, offset);
              case LEFT:
                    if(j==0)
                          return NULL;
                    return swap(s, pos, -1, offset);
              case RIGHT:
                    if(j==2)
                          return NULL;
                    return swap(s, pos, 1, offset);

              default: return NULL;
        }
}


__global__
void compute(int *visited, bool *frontier, bool *Ufrontier, bool *fin){
        char *adj;
        int tid = blockIdx.x * blockDim.x + threadIdx.x;
        int index, pos;
        int offset = tid*9;
        if(tid < SIZE && frontier[tid]){
              frontier[tid] = false;
              if(tid==gIndex)
                    *fin = true;
              for (int i = 0; i < LEN; ++i) {
                    if(state[offset+i] == '0')
                          pos = i;
              }
              for (int i = UP; i <= RIGHT; ++i) {
                    adj=move(state, pos, (Move)i, tid);
                    if(adj == NULL){
                          continue;
                    }
                    index = getHash(adj,tid);
                    offset = index*9;
                    if(visited[index] < 0){
                          Ufrontier[index] = true;
```

```
                        visited[index] = tid;
                        for (int i = 0; i < LEN; ++i) {
                              state[offset+i] = adj[i];
                        }
                  }
            }
      }
}

__global__
void save(bool *frontier, bool *Ufrontier, bool *fin){
      int tid = blockIdx.x * blockDim.x + threadIdx.x;
      if(tid < SIZE && Ufrontier[tid]){
            Ufrontier[tid] = false;
            frontier[tid] = true;
      }
}
__global__
void dummy(bool *frontier, bool *fin){
      frontier[sIndex] = true;
}


int main(int argc, char **argv) {
      printf("Enter start: ");
      scanf("%s",s);

      printf("Enter goal: ");
      scanf("%s",g);

      char *start, *goal;
      cudaMalloc((void**)&start,sizeof(char)*10);
      cudaMalloc((void**)&goal,sizeof(char)*10);

      cudaMemcpy(start,&s,sizeof(char)*10, cudaMemcpyHostToDevice);
      cudaMemcpy(goal,&g,sizeof(char)*10, cudaMemcpyHostToDevice);

      cudaMalloc((void**)&visited,sizeof(int)*SIZE);
      cudaMalloc((void**)&frontier,sizeof(bool)*SIZE);
      cudaMalloc((void**)&Ufrontier,sizeof(bool)*SIZE);
      char *h, *t, *st;
      cudaMalloc((void**)&h,sizeof(char)*SIZE*9);
      cudaMalloc((void**)&t,sizeof(char)*SIZE*9);
      cudaMalloc((void**)&st,sizeof(char)*SIZE*9);


      int threads = 504;
      int blocks = SIZE/threads;
```

```
	init<<<1,1>>>(start, goal, visited, frontier, Ufrontier, h, t, st);
	CUDA_CHECK_RETURN(cudaDeviceSynchronize());
	clean<<<blocks, threads>>>(visited, frontier, Ufrontier);
	CUDA_CHECK_RETURN(cudaDeviceSynchronize());


	bool fin = false, *dfin;
	cudaMalloc((void**)&dfin,sizeof(bool));
	cudaMemcpy(dfin,&fin,sizeof(bool), cudaMemcpyHostToDevice);
	dummy<<<1,1>>>(frontier, dfin);
	CUDA_CHECK_RETURN(cudaDeviceSynchronize());
	int k=0;
	while(!fin){
		compute<<<blocks, threads>>>(visited, frontier, Ufrontier, dfin);
		CUDA_CHECK_RETURN(cudaDeviceSynchronize());
		save<<<blocks, threads>>>(frontier, Ufrontier, dfin);
		cudaMemcpy(&fin,dfin,sizeof(bool), cudaMemcpyDeviceToHost);
		CUDA_CHECK_RETURN(cudaDeviceSynchronize());
		k++;
	}
	printf("\n%d",k-1);

//free allocated memories

}
```

C.5. Complete 8-Puzzle Solver

```
/*****************************************************************************
 Implementing parallel 8-puzzle solver on CUDA
*****************************************************************************/
#include <ctime>
#include <iostream>
#include <stdio.h>
#include <assert.h>

#define CUDA_CHECK_RETURN(value) {                                          \
      cudaError_t _m_cudaStat = value;                                      \
      if (_m_cudaStat != cudaSuccess) {                                     \
            fprintf(stderr, "Error %s at line %d in file %s\n",             \
                        cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__); \
            exit(1);                                                        \
      } }

#define LEN 9
#define mySIZE 362880 //9!

using namespace std;

class Queue{
      private:
            int front,rear;
            int qarr[30000]; //a heuristic value is taken
            int s;
      public:
            __device__
            void init(){
                  if(&qarr == NULL){
                        printf("Queue failed\n");
                        assert(0);
                        return;
                  }
                  front = 0;
                  rear = 0;
                  s=0;
            }

            __device__
            void enQueue(int x){
                  if(s==30000){
                        printf("Q full");
                        assert(0);
                  }
```

```
			qarr[rear] = x;
			rear++;
			if(rear == 30000)
				rear = 0;
			s++;
		}

		__device__
		int deQueue(){
			if(isEmpty())//{
				return -1;  //empty
			s--;
			int x = qarr[front];
			front++;
			if(front == 30000)
				front = 0;
			return x;
		}

		__device__
		bool isEmpty(){
			if(s == 0)
				return true;
			return false;
		}
		__device__
		int size(){
			return s;
		}
};

__device__
int fact(int n){
	int x=1;
	for (int i = 1; i <= n; i++) {
		x *= i;
	}
	return x;
}

__device__ int factof[9];

__global__
void storeFact(){
	for (int i = 0; i < LEN; ++i) {
		factof[i] = fact(i);
	}
}
```

```
/////////////////////////////////////////////////////////////////////////////////
//Implementing minimal perfect hashing
//This will generate the index/rank of a given permutation
/////////////////////////////////////////////////////////////////////////////////
__device__
int getHash(int n, char *numH){
	int h=0;
	for (int i = LEN-1; i >= 0; --i) {
		numH[i] = n%10;
		n = n/10;
	}
	int f=LEN;
	for (int i = 0; i < LEN ; ++i) {
		f--;
		if(numH[i] > 0)
			h += numH[i]*factof[f];
		for (int j = i+1; j < LEN; ++j) {
			if(numH[j] > numH[i]){
				numH[j]--;
			}
		}
	}
	return h;
}

__device__ int state[mySIZE];
__device__ int k = 0;

__device__ __noinline__
void generate(int x[], int l){
	if (l == LEN) {
		int num = 0;
		for (int i = 0; i < LEN; i++) {
			num = num * 10 + x[i];
		}
		state[k++] = num;
		return;
	}
	int j;
	for (int i = 0; i < LEN; i++) {
		for (j = 0; j < l; j++) {
			if(x[j] == i)
				break; //next i
		}
		if(j==l){ //fully iterated, i not in x[]
			x[l] = i;
			generate(x,l+1);
		}
```

```
    }
}

__global__ __noinline__
void populate()
{
    int x[LEN] = { 0, 1, 2, 3, 4, 5, 6, 7, 8 };
    generate(x,0);

}

enum Move{UP, DOWN, LEFT, RIGHT};

__device__ int cost[mySIZE];
__device__ int count = 0;
__device__ int gIndex;

__device__
void dump(char str[]){
    for (int i = 0; i < 9; ++i) {
        printf("%d",str[i]);
    }
    printf("\n");
}

__device__
char* swap(char str[],int pos,int p){
    int x=pos;
    int y=pos+p;
    char tmp = str[x];
    str[x] = str[y];
    str[y] = tmp;
    pos = y;
    return str;
}

__global__
void setGoal(int n){
    char tmp[LEN];
    gIndex = getHash(n,tmp);
}

__device__
int move(int state,Move m,char *nextM){
    int pos,i,j;
    for (int k = LEN-1; k >= 0; --k) {
        nextM[k] = state%10;
        state = state/10;
```

```
            if(nextM[k] == 0)
                  pos = k;
      }
      i=pos/3;
      j=pos%3;
      switch(m){
            case UP:
                  if(i==0)
                        return -1;
                  swap(nextM,pos,-3);
                  break;
            case DOWN:
                  if(i==2)
                        return -1;
                  swap(nextM,pos,3);
                  break;
            case LEFT:
                  if(j==0)
                        return -1;
                  swap(nextM,pos,-1);
                  break;
            case RIGHT:
                  if(j==2)
                        return -1;
                  swap(nextM,pos,1);
                  break;
            default: return -1;
      }
      int x=0;
      for (int i = 0; i < LEN; ++i) {
            x = x*10 + nextM[i];
      }
      return x;
}

class eightPuzzle{
      private:
            Queue Q;
            bool visited[mySIZE];
            //for statistical purpose we do not need to store the path
            int source;

      public:
            int sIndex;

            __device__
            void start(int s){
                  Q.init();
```

```
        sIndex = s;
        source = state[s];
        if(&visited == NULL){
              printf("%d: malloc failed for path",sIndex);
              assert(0);
              return;
        }
        for (int i = 0; i < mySIZE; ++i) {
              visited[i] = false;//-1;
        }
        bool fin=false;
        fin = doBFS();

        if(fin){
              count++;
        }
  }
  __device__
  ~eightPuzzle(){
        clean();
  }

  __device__
  void clean(){
        while(!Q.isEmpty()){
              Q.deQueue();
        }
  }

  __device__
  bool doBFS(){
        int currIndex,index;
        int c=0;
        //__shared__
        char tmp[LEN];

        Q.enQueue(source);
        visited[sIndex] = true; //initial;  //mark as visited

        Q.enQueue(-1);    //level marker

        int curr,child;
        while(!Q.isEmpty()){
              curr = Q.deQueue();
              if(curr == -1){   //a level has completed
                    c++;
                    if(Q.isEmpty()){
```

```
                        cost[sIndex] = 0;
                        break;
                    }
                    Q.enQueue(-1);    //set marker for next level
                    //level completion = next level children are
already in the Q
                    continue;
                }
                currIndex = getHash(curr,tmp);
                if(currIndex == gIndex){      //Reached Goal
                    cost[sIndex] = c;
                    return true;
                }

                //for each child enQueue and mark
                for (int i = UP; i <= RIGHT; ++i) {
                    child = move(curr,(Move)i,tmp);
                    if(child == -1){
                        continue;
                    }
                    index = getHash(child,tmp);
                    if(!visited[index]){
                        visited[index] = true;
                        Q.enQueue(child);
                    }
                }
            }
            return false;
        }
};

__device__ eightPuzzle *game;

__global__
void dummy(eightPuzzle *g){
    game = g;
    if(game == NULL)
        printf("Abort");
}

__device__ int iteration;

__global__
void compute() {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    game[tid].start(tid+iteration);
}
```

```
__global__ void nextI(){
	iteration += 1008;
}

int main(int argc, char **argv) {
	system("date");
	clock_t t;
	t=clock();

	storeFact<<<1,1>>>();
	cout<<"factorials stored"<<endl;

	populate<<<1,1>>>();
	CUDA_CHECK_RETURN(cudaDeviceSynchronize());
	cout<<sizeof(eightPuzzle)<<endl;
	cout<<sizeof(Queue)<<endl;

	cout<<"alloc done"<<endl;

	cout<<(float(clock()-t))/CLOCKS_PER_SEC<<endl;

	setGoal<<<1,1>>>(123456780);
	CUDA_CHECK_RETURN(cudaDeviceSynchronize());
	cout<<"goal set"<<endl;
	cout<<(float(clock()-t))/CLOCKS_PER_SEC<<endl;

	eightPuzzle *game;
	cudaMalloc((void**)&game,sizeof(eightPuzzle)*1008);

	dummy<<<1,1>>>(game);
	CUDA_CHECK_RETURN(cudaDeviceSynchronize());
	cout<<"game ready"<<endl;
	cout<<(float(clock()-t))/CLOCKS_PER_SEC<<endl;

	int loop = mySIZE/(1008); //9*8*7 = 504
	for (int i = 0; i < loop; ++i) {
		compute<<<1,1008>>>();
		CUDA_CHECK_RETURN(cudaDeviceSynchronize());
		nextI<<<1,1>>>();
		//cout<<(float(clock()-t))/CLOCKS_PER_SEC<<endl;
		//cout<<"loop:"<< i <<endl;
	}
	CUDA_CHECK_RETURN(cudaDeviceSynchronize());
	int c;
	CUDA_CHECK_RETURN(cudaMemcpyFromSymbol(&c,count,sizeof(int),0,cudaMemcpyDe
viceToHost));
	cout<<c<<endl;
	cout<<(float(clock()-t))/CLOCKS_PER_SEC<<endl;
```

```
	system("date");

	cout<<"===================================="<<endl;
	int moves[mySIZE];
	cudaMemcpyFromSymbol((void**)&moves,cost,sizeof(int)*mySIZE,0,cudaMemcpyDe
viceToHost);
	CUDA_CHECK_RETURN(cudaDeviceSynchronize());
	cout<<"result copied to host!"<<endl;
	cout<<"dumping result"<<endl;
	for (int i = 0; i < mySIZE; ++i) {
		cout<<i<<":"<<moves[i]<<endl;
	}
}
```