GPU101 Project

Breadth-First Search Algorithm

Lorenzo Di Napoli

lorenzo.dinapoli@mail.polimi.it

Project Goal

Breadth-First Search Algorithm

- Implementation and optimisation of the ***Breadth-First Search (BFS)*** Algorithm in *CUDA* programming language for the *GPU.*

Starting Materials

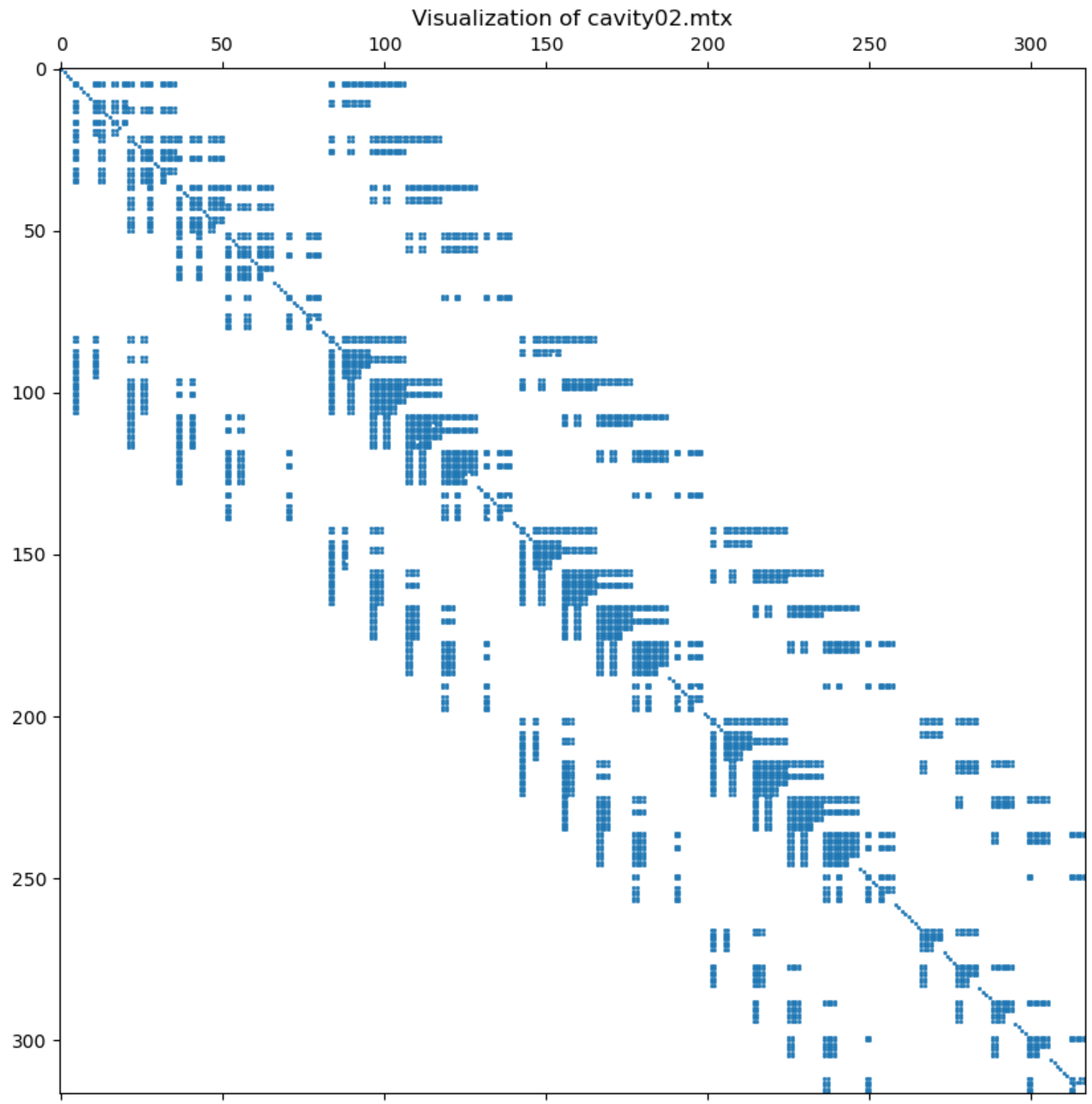
- a .mtx file (size 317x317) containing the Tree Structure.
- an already working *C++* script wich:
    - reads and represents the .mtx file in CSR format (*Compressed Sparse Row*).
    - executes via *CPU* the ***BFS_sequential( )*** Algorithm.

How to Read the Tree

- In order to graphically visualise the .mtx file, I wrote a phyton script.
- The script works this way: from an .mtx imput file it displays a graphic of the matrix.

```python
import matplotlib.pyplot as plt
import numpy as np
import os

# Check if the file exists
file_path = 'cavity02.mtx'
if not os.path.exists(file_path):
    print(f"File {file_path} not found.")
    exit(1)

try:
    # Read the matrix manually
    with open(file_path, 'r') as f:
        lines = f.readlines()

    # Skip comments and read the header
    lines = [line for line in lines if not line.startswith('%')]
    header = lines[0].strip().split()
    num_rows, num_cols, num_entries = map(int, header)

    # Initialize the matrix
    matrix = np.zeros((num_rows, num_cols))

    # Read the entries
    for line in lines[1:]:
        row, col, value = map(float, line.strip().split())
        matrix[int(row)-1, int(col)-1] = value

    # Print matrix details for debugging
    print(f"Matrix type: {type(matrix)}")
    print(f"Matrix shape: {matrix.shape}")

    # Plot the matrix
    plt.spy(matrix, markersize=1)
    plt.title('Visualization of ' + file_path)
    plt.show()
except Exception as e:
    print(f"An error occurred: {e}")
```

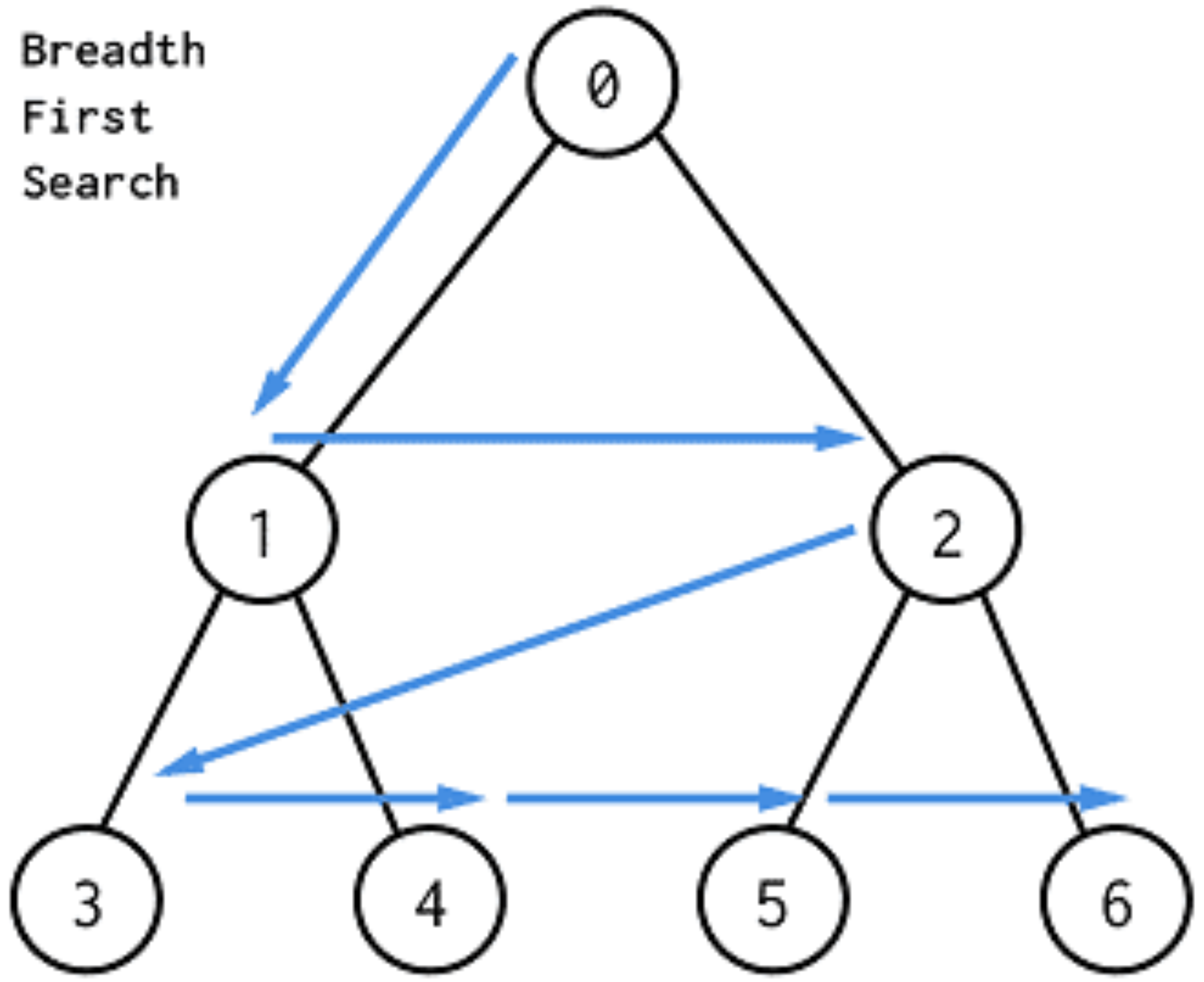
Visualization of the Tree

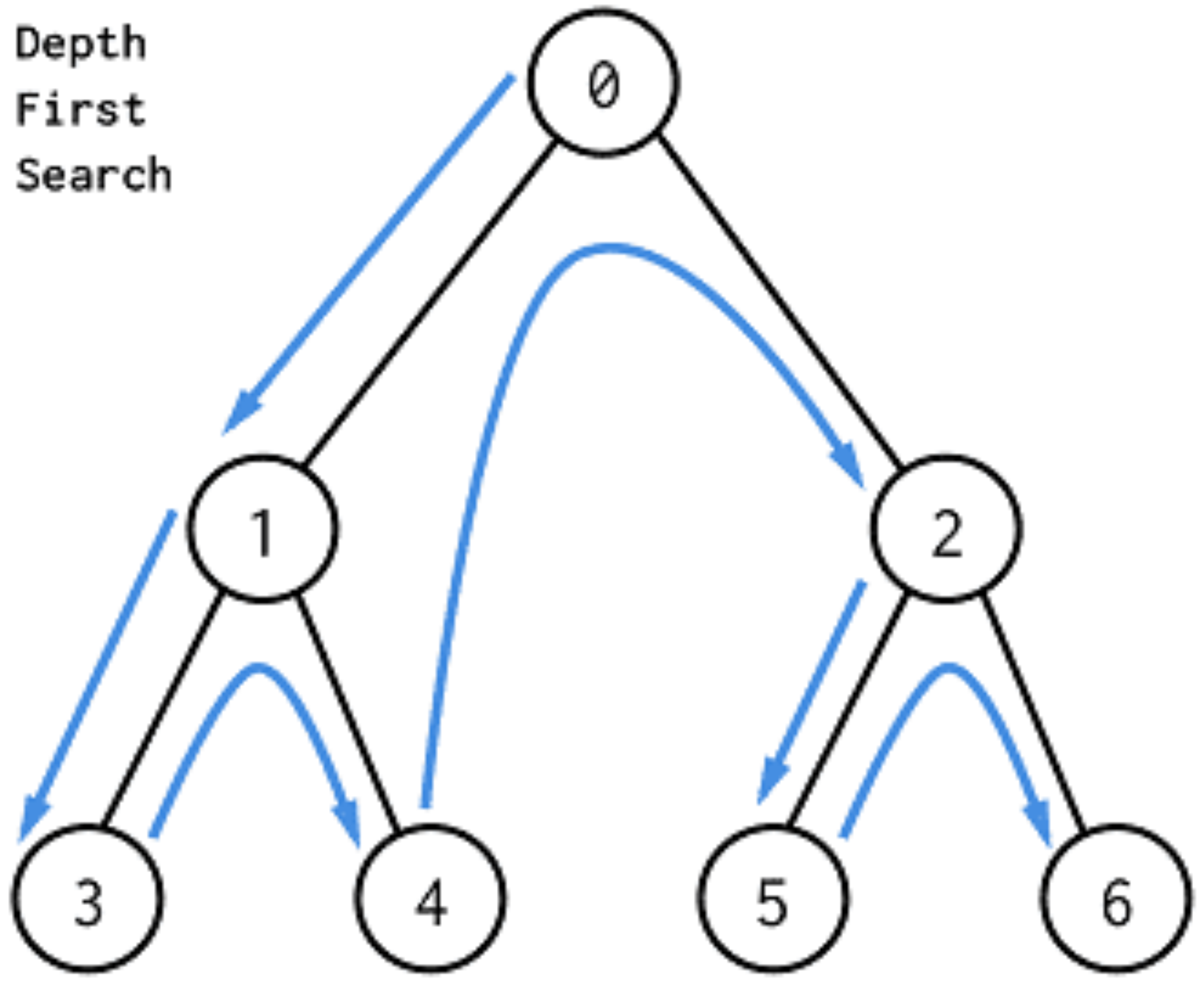


Explanation of the Algorithm

BFS vs DFS

- ***Breadth-First Search (BFS)*** is a traversing algorithm wich works by exploring the totality of the frontier befor going deeper (Breadth analysis).
- It is different from the ***Depth-First Search (DFS)***, wich works by exploring first by depth and then by frontier (Depth analysis).





Procedure

main( ) adptation

- In order to call the *<u>GPU-kernell</u>*, it is necessary to adapt the ***main( )*** function:
  - have to be inizialised some pointers for the variables to give at the kernell, paying attention to the double cases using *host_variable* / *gpu_variable*.
  - have to be allocated the *GPU* device memory via **cudaMalloc( )** and copied the data in the direction *HostToDevice* via **cudaMemcpy( )**, after the kernell had executed the *BFS*, the memory is freed via **cudaFree( )**.
  - all the *CUDA* calls are checked via the macro CHECK(call), the kernel launch is checked via the macro CHECK_KERNELLCALL( ).

Procedure

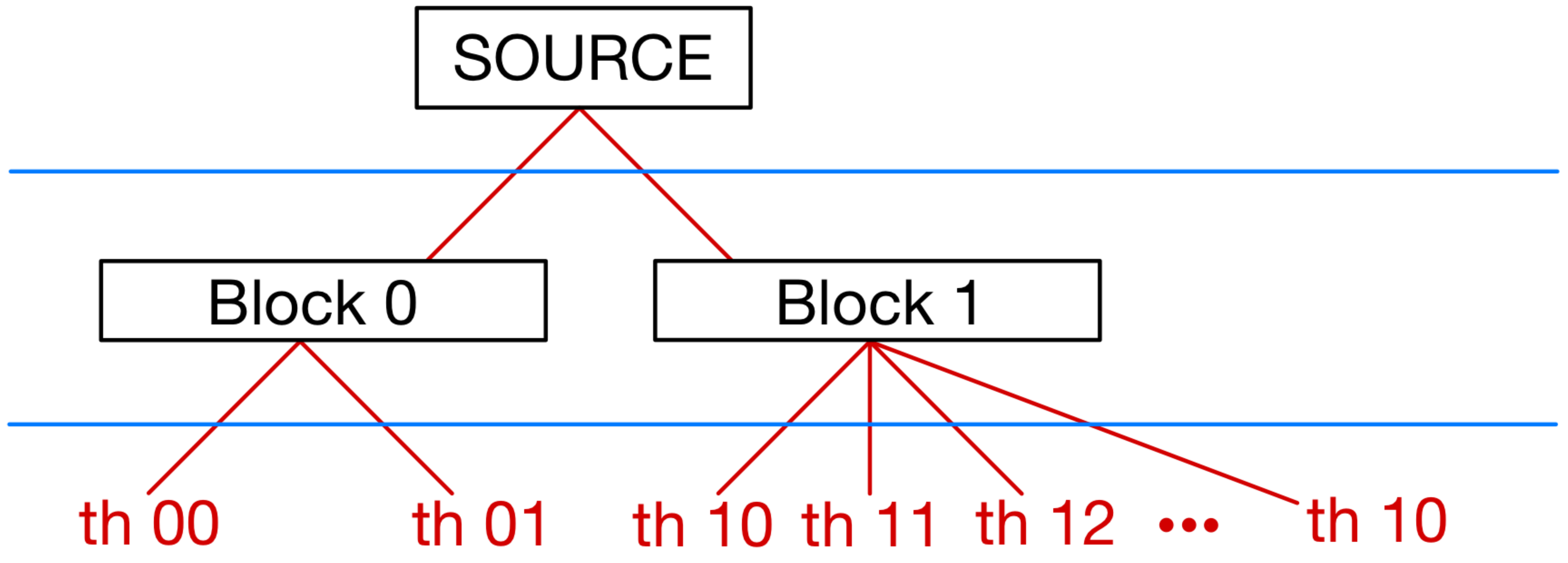
main( ) adptation

```
// gpu variables allocation
int *gpu_row_ptr;
int *gpu_col_ind;
int *gpu_dist;
int *gpu_source;

// gpu memory allocation
CHECK(cudaMalloc(&gpu_source, sizeof(int)));
CHECK(cudaMalloc(&gpu_row_ptr, host_row_ptr.size() * sizeof(int)));
CHECK(cudaMalloc(&gpu_col_ind, host_col_ind.size() * sizeof(int)));
CHECK(cudaMalloc(&gpu_dist, num_vals * sizeof(int)));

// Copy data from host to device
CHECK(cudaMemcpy(gpu_source, &host_source, sizeof(int), cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(gpu_row_ptr, host_row_ptr.data(), host_row_ptr.size() * sizeof(int), cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(gpu_col_ind, host_col_ind.data(), host_col_ind.size() * sizeof(int), cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(gpu_dist, host_dist.data(), num_vals * sizeof(int), cudaMemcpyHostToDevice));

// Call the gpu_kernel function
BFS_gpu<<<DIM_GRID,DIM_BLOCK>>>(gpu_source, gpu_row_ptr, gpu_col_ind, gpu_dist);
CHECK_KERNELCALL();

// gpu memory free
CHECK(cudaFree(gpu_source));
CHECK(cudaFree(gpu_row_ptr));
CHECK(cudaFree(gpu_col_ind));
CHECK(cudaFree(gpu_dist));
```

Procedure

GPU architecture implementation

- Since the *BFS Algorithm* operates by <u>Breadth Analysis</u>, for each frontier are defined the following associations:
  - each vertex corrisponds to a block of the *GPU*.
  - each edge of a vertex is analysed by a thread of the corrispective block.
  - if there are more edges in one vertex than thread per block, each thread may have to analyse sequentially more edges.



Procedure

kernell call

- The <u>*GPU-kernell*</u> call contains four variables:
  - a pointer to the root vertex <u>const int *source_ptr</u>.
  - a pointer to the first edge of each vertex <u>const int *rowPointers</u>.
  - a pointer to the destination of each edge <u>const int *destinations</u>.
  - a pointer wich contains the distances of all vertices from the root and is set at -1 from the ***main( )*** <u>int *distances</u>.

```
// BFS algorithm optimized for GPU
__global__ void BFS_gpu(const int *source_ptr, const int *rowPointers, const int *destinations, int *distances)
{
```

Procedure

frontiers initialization

- For each block are initialised four __shared__ variables:
    - an array representing the previous frontier of the Tree int previousFrontier[ ].
    - an array representing the current frontier of the Tree int currentFrontier[ ].
    - two integers used to indicate the size of the two frontiers (the number of outgoing edges from the frontier) int previousFrontierSize and int currentFrontierSize.

```
// initialize frontiers
__shared__ int currentFrontier[MAX_FRONTIER_SIZE];
__shared__ int currentFrontierSize;
__shared__ int previousFrontier[MAX_FRONTIER_SIZE];
__shared__ int previousFrontierSize;
```

Procedure

BFS loop setting

- Given the pointer to the root const int *source_ptr, the following step it to define the first frontier as previousFrontier[ ] and the first size as previousFrontierSize via the __*device*__ function ***insertIntoFrontier( )***.

- Only then is possible to start a while( ) loop, wich is the core of the *BFS*.

- This loop will go on until the given frontier previosFrontier[ ] has at least a new child vertex to analyse (previousFrontierSize > 0).

Procedure

BFS loop setting

```
// initialize block's previous frontier from source
if (threadIdx.x == 0)
{
  currentFrontierSize = 0;
  previousFrontierSize = 0;
  const int source = *source_ptr;
  insertIntoFrontier(source, previousFrontier, &previousFrontierSize);
  distances[source] = 0;
}

__syncthreads();

// BFS with parallel vertices
while(previousFrontierSize > 0)       // while there are new vertices to visit
{
```

Procedure

BFS loop explanation

- At each iteration is analysed a single frontier, wich is represented by the couple previousFrontier[ ] and previousFrontierSize.
- Each frontier use only the necessary number of blocks in order to represent all the vertices via the check condition: if (blockIdx.x < previousFrontierSize).
- For each block are defined:
    - the current vertex as int currentVertex = previousFrontier[blockIdx.x].
    - the address of the first edge of the currentVertex, wich is defined as int row_start = rowPointers[currentVertex].
    - the address of the first edge of the next vertex, wich is used in order to establish the limit for the currentVertex edges, as int row_end = rowPointers[currentVertex +1].

Procedure

BFS loop explanation

```
// BFS with parallel vertices
while(previousFrontierSize > 0)        // while there are new vertices to visit
{
  // visit all vertices on the previus frontier
  if(blockIdx.x < previousFrontierSize)
  {
    int currentVertex = previousFrontier[blockIdx.x];
    int row_start = rowPointers[currentVertex];
    int row_end = rowPointers[currentVertex + 1];
```

Procedure

BFS loop explanation

- The threads are managed in a more organised way:
- For each vertex represented by a block:
  - each edge is analysed by a single thread, via the association between the edge `int row_i = row_start + threadIdx.x`
  - a thread represents one edge only if is true that `row_i < row_end`.
  - if the edges of a vertex are more than the number of threads per block, some threads may have to analyse more edges.
  - in order to cover this eventuality, the check is part of a `for()` loop, wich is defined as `for ( int row_i; row_i < row_end; row_i += DIM_BLOCK)`.

Procedure

BFS loop explanation

- Then is carried out a check for each int row_i, wich analyse the distance from the currentVertex and the destination.
  - if the vertex has not been explored yet, the check gives as a result: distances[destinations[row_i]] == -1.
  - in this case the currentFrontier[ ] and the currentFrontierSize have to be updated with the new vertex destinations[row_i] via the __*device*__ function ***insertIntoFrontier( )***.
  - it also has to be calculated the distance of the new vertex as one more than the currentVertex distance, as distances[destinations[row_i]] = distances[currentVertex] +1.

Procedure

BFS loop explanation

```
// check all outgoing edges
for(int row_i = row_start + threadIdx.x; row_i < row_end; row_i += DIM_BLOCK)      // parallelize over all outgoing edges even if they are more than the block size
{
  if(distances[destinations[row_i]] == -1)
  {
    // this vertex has not been visited yet
    insertIntoFrontier(destinations[row_i], currentFrontier, &currentFrontierSize);
    distances[destinations[row_i]] = distances[currentVertex] + 1;
  }
}
```

Procedure

BFS loop explanation

- Once all the vertices have been analysed, all the new distances are updated in the frontier currentFrontier[ ].

- When all threads of each block have ended the analysis, the first thread per block calls the *__device__* function ***swap( )*** between previousFrontier[ ] and currentFrontier[ ].

- When all the ***swap( )*** executions have come to an end, the loop is ready to begin a new iteration on the next frontier.

```
    // wait for all vertices to be visited
    __syncthreads();

    // swap to the next frontier
    if(threadIdx.x == 0)
    {
        swap(&currentFrontier, &previousFrontier);
        previousFrontierSize = currentFrontierSize;
        currentFrontierSize  = 0;
    }

    // synchronize with the swap
    __syncthreads();

}
```

Conclusions

BFS implementation and total parallelisation

- We have seen an implementation of the ***Breadth-First Search (BFS)*** *Algorithm* in *CUDA* programming language, in order to use the capability of a *GPU* device.
- This implementation is also totally optimised, since it parallelise each level of the *GPU* architecture:
    - each vertex of the frontier is analysed by a block in parallel.
    - each edge of the vertex is analysed by a thread in parallel.
- The only limitation of this implementation is definition of *BFS* as a *Breadth Analysis Algorithm*, wich implies the sequentiality of the frontier analysis.