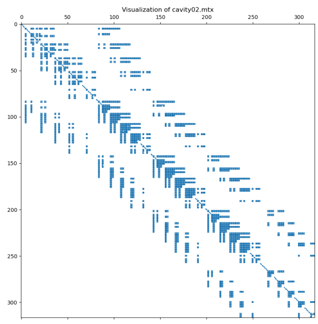
Passion-In-Action Class:	GPU101
Student:	Di Napoli Lorenzo / lorenzo.dinapoli@mail.polimi.it
Assigned Project:	Breadth-First Search (BFS) Algorithm

PROJECT GOAL

This Project Goal is to implement and optimise in *CUDA* programming language, for *GPU* devices, the ***Breadth-First Search (BFS)*** Algorithm, starting from a sequential *BFS* script wrote in *C++* programming language.

STARTING MATERIAL

The starting material is the following one:

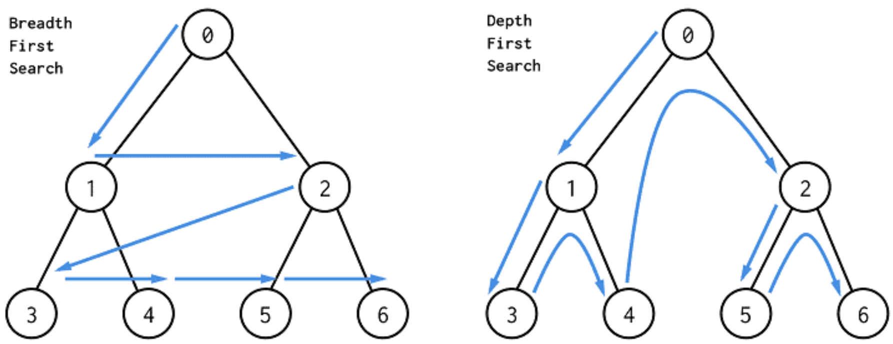


- .mtx file rappresenting a Tree Structure as a matrix (in this case a square and simmetrical matrix of size 317x317).
- script wrote in *C++* programming language which performs the following:
  - reading and representation of the previously mentioned .mtx file in CSR format (*Compressed Sparse Row*).
  - sequential execution by the host of the *BFS Algorithm* in order to analyze the Tree Structure.

ALGORITHM EXPLANATION

The ***Breadth-First Search (BFS)*** Algorithm performs a Breadth-Analysis of a Tree Structure which, starting form a root vertex, analyses al the arising vertices by breadth (unlike the ***Depth-First Search (DFS)*** Algorithm, which performs a Depth Analysis).
The BFS Algorithm analyses first the child vertex from the root (*first frontier*), then the child of the child (*second frontier*) and so on by frontiers, until he has analysed the entire the Tree Structure, or until a vertex satisfying a certain property is found.

In this Project, using the *BFS Algortihm* allows to measure the distances between the vertices and the root, that is given by the user via the variable `const int host_source`.



PROCEDURE

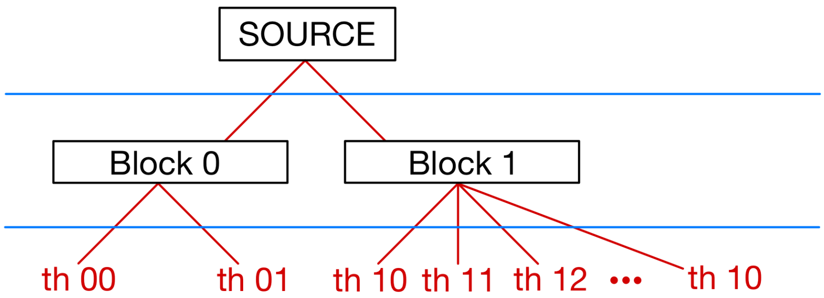
Starting from the given script, the goal is to change the function call ***BFS_sequential( )*** with a *GPU-kernell* call of a parallelised version of the *BFS Algorithm*, wich has been named ***BFS_gpu( )***.

1. First of all has to be modified the ***main( )*** function in order to satisfy the *CUDA* requirements:
   - have to be initialised some pointers for the variables to give to the kernell, paying attention to the double cases using the label *host_variable / gpu_variable*.
   - has to be allocated the *GPU* device memory via the function cudaMalloc( ).
   - have to be copied the data in the direction *HostToDevice* via the function cudaMemcpy( ).
   - only then can be done the *GPU-kernell* call to ***BFS_gpu( )*** with specific methodology.
   - when the kernell execution has come to an end, has to be free the *GPU* device memory via the function cudaFree( ).
   - everithing is checked via the macro CHECK(call) for the *CUDA* function calls, and via the macro CHECK_KERNELLCALL( ) after the *GPU-kernell* call.

```
// 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));
```

2. Since the definition of the *BFS* as a *Breadth Analysis Algorithm* of a Tree Structure, for each frontier have been made the following associations:
   - each vertex corrisponds to a *GPU* block.
   - in order to analyse one edge of a given vertex, is used a single thread of the corrispective block; a thread may have to analyse more edges, if there are more edges in a single vertex that threads per block
   - operating this way has a significantly less impact on the performance than operating in a completely sequential way.

This associations allow to obtain a complete parallelism, taking advantage of every level that the *GPU* architecture has to offer, being limited only by the *BFS Algorithm* definition of *Breadth Analysis* per layer.



3. The *GPU-kernell* call contains four variables:
   - a pointer to the root vertex from wich the Algorithm begins const int *source_ptr.
   - a pointer const int *rowPointers, which indicates the first edge of a vertex as rowPointers[vertex].
   - a pointer const int *destinations, which indicates the destination vertex of an edge as destinations[edge].
   - a pointer int *distances, wich contains the values of the distances between each vertex and the root; this array has been set at -1 by the ***main( )*** function.

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

4. For each block are initialised four *__shared__* variables:
   - an array wich represents the previous frontier of the Tree Structure int previousFrontier[ ].
   - an array wich represents the current frontier of the Tree Structure int currentFrontier[ ].
   - two integers used to indicate the size of the corrispective two frontiers, which are establish by the number of outgoing edges from the given frontier int previousFrontierSize and int currentFrontierSize.

This use of the *GPU* device shared memory allows a big memory optimisation, since all the threads of the block work on the same frontier; in this way less memory is used and the vertex analysis is way more fast and parallelised.

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

5. Given the pointer to the root const int *source_ptr, the next step is 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 Algorithm*: this loop will go on until the given frontier previousFrontier[ ] has at least a new child vertex to analyse, this is checked via the condition previousFrontierSize > 0.

```
// 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
{
```

6. The BFS mechanism has been implemented in the following way:

- at each iteration is analysed a single frontier, which is represented by the couple previousFrontier[ ] and previousFrontierSize.

- each frontier use only the necessary number of blocks in order to represent all the vertices, this is done via the condition if (blockIdx.x < previousFrontierSize).

- the threads are managed in a more organised way:
  - first is obteined the addres of the first edge of a given vertex, as int row_start = rowPointers[currentVertex], defining the vertex as int currentVertex = previousFrontier[blockIdx.x].
  - in order to establish the limit to the vertex edges, is obtained the addres of the first edge of the next vertex, as int row_end = rowPointers[currentVertex +1].
  - then is carried out the check row_i < row_end in order to establish if the thread associated to int row_i = row_start + threadIdx.x has to analyse an edge of the given vertex (a thread may have to analyse from zero to more than one int row_i, depending on the block size *DIM_BLOCK*).

- for each int row_i, is done a check on the distance between the root and the destination vertex, if it gives as a result that distances[destinations[row_i]] == -1, this means:
  - the destination vertex has not been explored yet, so the first thing to do is to update currentFrontier[ ] and currentFrontierSize, this is done via the *__device__* function ***insertIntoFrontier( )***.
  - the distance between the new vertex and the root vertex const int source is calculated as distances[destinations[row_i]] = distances[currentVertex] +1.

- once all the vertices of the frontier previousFrontier[ ], and all the relative edges rowPointers[blockIdx.x], have been analysed; the vertices distances results updated into the array currentFrontier[ ].

- when all the threads of each block have ended the analysis, the first thread per block calls the *__device__* function ***swap( )*** between currentFrontier[ ] and previousFrontier[ ]; after all the ***swap( )*** executions have come to an end, a new iteration of the while( ) loop is ready to start, always if the condition previousFrontierSize > 0 is satisfied.

```
// 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];

    // 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;
      }
    }
  }

  // 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

The *GPU-kernell* call of the ***BFS_gpu( )*** function give as a result the implementation of a completely pararellised and optimized, both at the thread and at the block levels, of the ***Breadth-First Search (BFS)*** *Algorithm*, all done in *CUDA* programming language, for the use of a *GPU* device.