Chapter 3

Communication Wall: GRAPH500

3.1 Introduction

In order to face the communication wall in our metric for accelerators, we choose the Graph500 problem. In this part we present the benchmark itself, from the graph generation to the traversal. We focus on the BFS part, we do not work on the latest Graph500 using SSSP.

3.1.1 Breadth First Search

The most commonly used search algorithms for graphs are Breadth First Search (BFS) and Depth First Search (DFS). Many graph analysis methods, such as the finding of shortest path for unweighted graphs and centrality, are based on BFS.

As it is a standard approach method in graph theory, its implementation and optimization require extensive work. This algorithm can be seen as frontier expansion and exploration. At each step the frontier is expanded with the unvisited neighbors.

Algorithm 1 Sequential BFS

```
1: function COMPUTE_BFS(G = (V, E): graph representation, v_s: source vertex, In: current
   level input, Out: current level output, Vis: already visited vertices)
2:     In ← {v_s};
3:     Vis ← {v_s};
4:     P(v) ←⊥ ∀v ∈ V;
5:     while In ≠ ∅ do
6:         Out ← ∅
7:         for u ∈ In do
8:             for v|(u, v) ∈ E do
9:                 if v ∉ Vis then
10:                    Out ← Out ∪ {v};
11:                    Vis ← Vis ∪ {v};
12:                    P(v) ← u;
13:                end if
14:            end for
15:        end for
16:        In ← Out
17:    end while
18: end function
```

The sequential and basic algorithm is well known and is presented on Algorithm 1.

This algorithm is presented on figure 3.1. At each step from the current queue we search through the neighbors of nodes. All the new nodes, not yet traversed, are added in the queue for

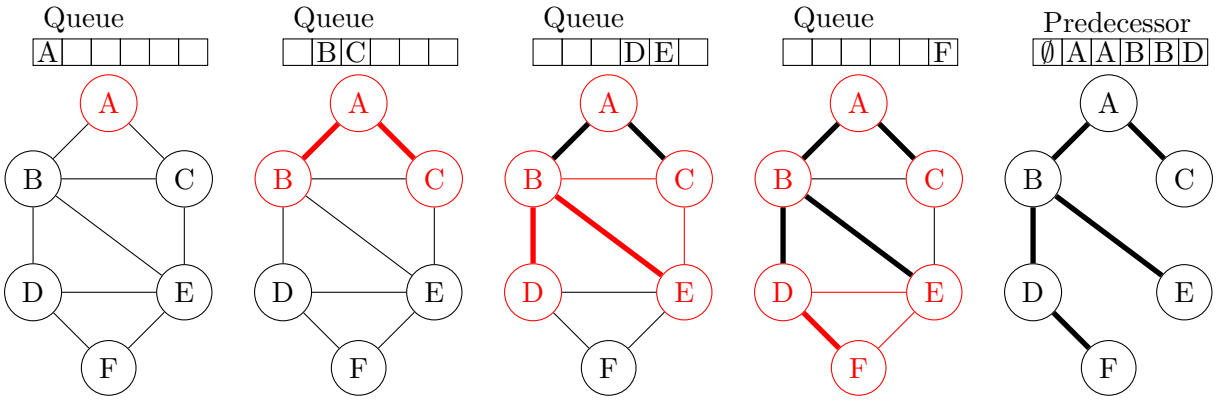

Figure 3.1: Example of Breadth First Search in an undirected graph

the next step. We keep the information of neighors exploration in order to recreate the precessor array.

This algorithm is very famous thanks to its use in many applications but also thanks to the world supercomputer ranking called Graph500[1]. This benchmark is designed to measure the performance on very irregular problems like BFS on a large scale randomized generated graph. The first Graph500 list was released in November 2010. The last list, issued in November 2017, is composed of 235 machines ranked using a specific metric: Traversed Edges Per Second, denoted as TEPS. The aim is to perform a succession of 64 BFS on a large scale graph in the fastest possible way. Then the ratio of edges traversed per the time of computation is used to rank the machines.

This benchmark is more representative of communication and memory accesses than computation itself. Other benchmarks can be used to rank computational power such as LINPACK for the TOP500 list. Indeed the best supercomputers (K-Computer, Sequoia, Mira, ...) on the ladder have a very specific communication topology and sufficient memory, and are large enough to quickly visit all the nodes of the graph.

In this study we focus on GPU optimization. There are many CPU algorithms available, which are listed on the Graph500 website. In order to rank the ROMEO supercomputer we had to create a dedicated version of the Graph500 benchmark in order to fit the supercomputer architecture. As this supercomputer is accelerated by GPUs, three successive approaches had to be applied: first create an optimized CPU algorithm; second provide a GPU specific version and third take advantage of both CPU and GPU computation power.

This paper is organized as follows. The first section performs a survey of graph representation and analysis; it also describes some specific implementations. The second section describes the Graph500 protocol and focuses on the Kronecker graph generation method and the BFS validation. The third section presents the chosen methods to implement graph representation and work distribution over the supercomputer nodes. It particularly focuses on the interest of a hybrid CSR and CSC representation. We concude by examining the results for different graph scales and load distributions.

3.2 Related work

The most efficient algorithm to compute BFS traversal is used and detailed in [CPW+12]. It uses a 2D partition of the graph which will be detailed later. This algorithm is used on the BlueGene/P and BlueGene/Q architectures but can be easily adapted to any parallel cluster.

We use another key study in order to build our Graph500 CPU/GPU implementation. This paper [MGG15] proposes various effective methods on GPU for BFS. Merrill & al. explaine

[1] `http://www.graph500.org`

and teste a few efficient methods to optimize memory access and work sharing between threads on a large set of graphs. It focuses on Kronecker graphs in particular. First they propose several methods for neighbor-gathering with a serial code versus a warp-based and a CTA-based approach. They also use hybridization of these methods to reach the performance level. In a second part they describe the way to perform label-lookup, to check if a vertex is already visited or not. They propose to use a bitmap representation of the graph with texture memory on the GPU for fast random accesses. In the last phase, they propose methods to suppress duplicated vertices generated during the neighbor exploration phase. Then based on these operations they propose *expand-contract*, *contract-expand*, *two-phase* and finally *hybrid* algorithms to adapt the method with all the studied graph classes. The last part they propose a multi-GPU implementation. They use a 1D partition of the graph and each GPU works on its subset of vertices and edges.

In [FDB+14], a first work is proposed to implement a multi-GPU cluster version of the Graph500 benchmark. The scheme used in their approach is quite similar to the one in our study but with a more powerful communication network, namely FDR InfiniBand.

In our work we focus on the GPUDirect usage on the ROMEO supercomputer.

3.3 Environment

As previously mentioned, a CPU implementation is available on the official Graph500 website. A large range of software technology is covered with MPI, OpenMP, etc. All these versions use the same generator and the same validation pattern which is described in this part below.

The Graph500 benchmark is based on the following stages:

- *Graph generation.* The first step is to generate the Kronecker graph and mix the edges and vertices. The graph size is chosen by the user (represented as a based-2 number of vertices). The $EDGEFACTOR$, average ratio of edges by vertex, is always 16. Self-loop and multiple edges are possible with Kronecker graphs. Then 64 vertices for the BFS are randomly chosen. The only rule is that a chosen vertex must have at least one link with another vertex in the graph. *This stage is not timed*;

- *Structure generation.* The specific code part begins here. Based on the edge list and its structure the user is free to distribute the graph over the machines. In a following section we describe our choices for the graph representation. *This stage is timed*;

- *BFS iterations.* This is the key part of the ranking. Based on the graph representation, the user implements a specific optimized BFS. Starting with a root vertex the aim is to build the correct BFS tree (up to a race condition at every level), storing the result in a predecessor list for each vertex;

- *BFS verification.* The user-computed BFS is validated. The number of traversed edges is determined during this stage.

The process is fairly simple and sources can be found at `http://www.graph500.org`. The real problem is to find an optimized way to use parallelism at several levels: node distribution, CPU and GPU distribution and then massive parallelism on accelerators.

3.3.1 Generator

The *Kronecker graphs*, based on Kronecker products, represent a specific graph class imposed by the Graph500 benchmark. These graphs represent realistic networks and are very useful in our case due to their irregular aspect [LCK+10]. The main generation method uses the Kronecker matrix product. Based on an initiator adjacency matrix K_1, we can generate a Kronecker graph of order $K_1^{[k]}$ by multiplying K_1 by itself k times. The Graph500 generator uses Stochastic

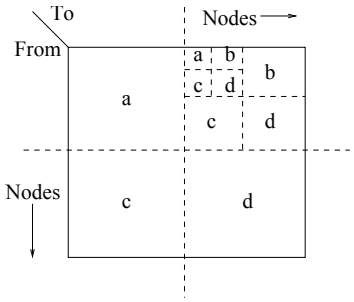

Figure 3.2: Kronecker generation scheme based on edge probability

Kronecker graphs, avoiding large scale matrix multiplying, to generate an edge list which is utterly mixed (vertex number and edge position) to avoid locality.

As presented on figure 3.2, the generation is based on edge presence probability on a part of the adjacency matrix. For the Graph500 the probabilities are $a = 0.57$, $b = c = 0.19$ and $d = 0.05$. The generator handle can be stored in a file or directly split in the RAM memory of each process. The first option is not very efficient and imposes a lot of I/O for the generation and verification stage but can be very useful for large scale problems. The second option is faster but uses a part of the RAM thus less ressources are available for the current BFS execution.

3.3.2 Validation

The validation stage is completed after the end of each BFS. The aim is to check if the tree is valid and if the edges are in the original graph. This is why we must keep a copy of the original graph in memory, file or RAM. This validation is based on the following stages, presented on the official Graph500 website. First, the BFS tree is a tree and does not contain cycles. Second, each tree edge connects vertices whose BFS levels differ by exactly one. Third, every edge in the input list has vertices with levels that differ by at most one or that both are not in the BFS tree. Finally, he BFS tree spans an entire connected component's vertices, and a node and its parent are joined by an edge of the original graph.

In order to meet the Graph500 requirements we use the proposed verification function provided in the official code.

3.4 BFS traversal

In this section we present the actual algorithm we used to perform the BFS on a multi-GPU cluster. In a first part we introduce the data structure; then we present the algorithm and the optimizations used.

3.4.1 Data structure

We performed tests of several data structures. In a first work we tried to work with bitmap. Indeed the regularity of computation can fit very well with the GPU architecture. But this representation imposes a significant limitation on the graph size. This representation is used on the BlueGene/Q architecture. Indeed they have some specific hardware bit-wise operations implemented in their processors and have a large amount of memory, allowing them to perform very large scale graph analysis.

In a second time we used common graph representations, Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) representation, which fit very well with sparse graphs

such as the Graph500 ones. The following example illustrates the CSR representation:

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$R = \{0, 2, 4, 7, 8\}$$

$$C = \{1, 2, 0, 2, 0, 1, 3, 2\}$$

The M adjacency matrix represents the graph. R vector contains the cumulative number of neighbors for each vertex, of size $(\#vertices + 1)$. C, of size $(\#edges)$, is, for each index of R, the edges of a vertex. This representation is very compact and very efficient to work with sparse graphs.

3.4.2 General algorithm

When looking at the latest Graph500 list we see that the best machines are the BlueGene ones. We count about 26 BlueGene/Q and BlueGene/P machines in the first 50 machines. This is due to a quite specific version of the BFS algorithm proposed in [CPW+12]. It proposes a very specific 2D distribution for parallelism and massive use of the 5D torus interconnect.

In the BFS algorithm, like other graph algorithms, parallelism can take several shapes. We can split the vertices into partitions using 1D partition. Each thread/machine can then work on a subset of vertices. The main issue with this method is that the partitions are not equal since the number of edges per vertex can be very different; moreover in graphs like Kronecker ones where some vertices have a very high degree compared to other ones. Thus we are confronted with a major load balancing problem.

In [CPW+12] they propose a new vision of graph traversal, here BFS, on distributed-memory machines. Instead of using standard 1D distribution their BFS is based on a 2D distribution. The adjacency matrix is split into blocks of same number of vertices. If we consider $l \times l$ blocks $A_{i,j}$ we can split the matrix as follows:

$$M = \begin{bmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,l-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,l-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{l-1,0} & A_{l-1,1} & \cdots & A_{l-1,l-1} \end{bmatrix}$$

Each bloc $A_{x,y}$ is a subset of edges. We notice that blocks $A_{0,l-1}$ and $A_{l-1,0}$ have the same edges but in a reverse direction for undirected graphs. Based on this distribution they use *virtual processors*, which are either machines or nodes, each associated with a block. This has several advantages. First we reduce the load balancing overhead and a communication pattern can be set up. Indeed each column shares the same *in_queue* and each row will generate an *out_queue* in the same range. Thus for all the exploration stages, communications are only on line and we just need a column communication phase to exchange the queues for the next BFS iteration. Algorithm 2 presents the BlueGene/Q and BlueGene/P parallel BFS.

This algorithm is based on the exploration phase, denoted by *ExploreFrontier()*. It performs the exploration phase independently on all the machines. Then several communication phases follow. The first two phases are performed on the same processes line. The last one is performed on a processes column.

- On line 15, an exclusive scan is performed for each process on the same line, all the $A_{i,x}$ with $i \in [0, l-1]$. This operation allows us to know which vertices have been discovered in this iteration.

Algorithm 2 Parallel BFS on BlueGene

1: $Vis_{i,j} \leftarrow In_{i,j}$
2: $P(N_{i,j}, v) \leftarrow \perp$ **for all** $v \in R^{1D}_{i,j}$
3: **if** $v_s \in R^{1D}_{i,j}$ **then**
4: $\quad P(N_{i,j}, v_s) \leftarrow v_s$
5: **end if**
6: **while** *true* **do**
7: $\quad (Out_{i,j}, Marks_{i,j} \leftarrow$ ExploreFrontier();
8: $\quad done \leftarrow \bigwedge_{0 \le k,l \le n} (Out_{k,l} = \emptyset)$
9: $\quad$ **if** *done* **then**
10: $\quad\quad$ **exit loop**
11: $\quad$ **end if**
12: $\quad$ **if** $j = 0$ **then**
13: $\quad\quad prefix_{i,j} = \emptyset$
14: $\quad$ **else**
15: $\quad\quad$ **receive** $prefix_{i,j}$ from $N_{i,j-1}$
16: $\quad$ **end if**
17: $\quad assigned_{i,j} \leftarrow Out_{i,j} \setminus prefix_{i,j}$
18: $\quad$ **if** $j \neq n - 1$ **then**
19: $\quad\quad$ **send** $prefix_{i,j} \cup Out_{i,j}$ **to** $N_{i,j+1}$
20: $\quad$ **end if**
21: $\quad Out_{i,j} \leftarrow \bigcup_{0 \le k \le n} Out_{i,k}$
22: $\quad$ WritePred()
23: $\quad Vis_{i,j} \leftarrow Vis_{i,j} \cup Out_{i,j}$
24: $\quad In_{i,j} \leftarrow Out_{j,i}$
25: **end while**

- On line 19, a broadcast of the current *out_queue* is sent to the processes on the same line. With this information they would be able to update the predecessor list only if they are the first parent of a vertex.
- On line 24, a global communication on each column is needed to prepare the next iteration. The aim is to replace the previous *in_queue* by the newly computed *out_queue*.

Two functions are not specified: *ExploreFrontier()* converts the *in_queue* into *out_queue* taking account of the previously visited vertices; *WritePred()* aims to generate the BFS tree and therefore store the predecessor list. In this algorithm the predecessor distribution is still in 1D to avoid vertex duplication. This part can be done using RDMA communication to update predecessor value or with traditional MPI all-to-all exchanges. It can be done during each iteration stage or at the end of the BFS but this requires using a part of the memory to store this data.

This algorithm, which is the basis of many implementations, is the main structure of our distribution.

3.4.3 Direction optimization

In order to get an optimized computation in terms of TEPS we decided to sacrifice a small part of the memory for storing both the CSC and CSR representations. Indeed during the different BFS iterations the *in_queue* size varies a lot and, taking this into account, it is wiser to perform exploration from *top-down* or *bottom-up*. So, as proposed in [BAP13], we perform a direction-optimized BFS.

In the first case, *top-down*, we start from the vertices in the *in_queue* and check all the neighbors verifying each time if this neighbor has ever been visited. Then if not, it is added to the *out_queue*. When the *in_queue* is sparse, like for the first and latest iterations, this method is very efficient. In the second case, *bottom-up*, we start the exploration by the not-yet-visited vertices and verify if there is a link between those vertices and the *in_queue* ones. If yes, the not-yet-visited vertex is added to the *out_queue*. figure 3.3 presents the two approches, whith the time visiting all the edges, and the benefits of their hybridization.

3.4.4 GPU optimization

In algorithm 2, two parts are not developed. namely *ExploreFrontier()* and *WritePred()*. Indeed these phases are optimized using the GPU. Based on the Merill et al. implementation, the algorithm is optimized to use the shared memory and the texture memory of the GPU. For our version we decided to keep the bitmap implementation for the communications and the queues. So we have to fit the CSR and CSC implementations. On algorithm 3 we present the CSR algorithm; CSC is based on the same approach but starting from the *visited* queue.

In the CSR version each warp is attached to a 32 bit word of the *in_queue* bitmap. Then if this word is empty the whole warp is released; if it contains some vertices, the threads collaborate to load the entire neighbor list. Then they access the coalescent area in the main memory to load the neighbor list. A texture memory is used to accelerate the verification concerning this vertex. Indeed this memory is optimized to be randomly accessed. Then the vertex is added in the bitmap *out_queue*.

3.4.5 Communications

Based on the algorithm 2 communications pattern, we first used MPI with the CPU transferring the data. But the host-device transfer time between the CPU and the GPU was too time-consuming. In order to accelerate the transfers between the GPUs, we used a specific GPU MPI-aware library. This library allows direct MPI operations from the memory of one GPU to another and also implements direct GPU collective operations. GPUDirect can be used coupled with this library. In the last version we used this optimization with GDRCopy.

Algorithm 3 Exploration kernel based on CSR

Constants:
$NWARP$: number of WARPS per block

Variables:
pos_word: position of the word in in_queue
$word$: value of the word in in_queue
$lane\_id$: thread ID in the WARP
$warp\_id$: WARP number if this block
$comm[NWARP][3]$: shared memory array
$shared\_vertex[NWARP]$: vertex in shared memory

Begin
if $word = 0$ **then**
    free this WARP
end if
if $word$ &1 $<<$ $lane\_id$ **then**
    $id\_sommet \leftarrow= pos\_word * 32 + lane\_id$
    $range[0] \leftarrow C[id\_sommet]$
    $range[1] \leftarrow C[id\_sommet + 1]$
    $range[2] \leftarrow range[1] - range[0]$
end if
while $\_\_any(range[2])$ **do**
    if $range[2]$ **then**
        $comm[warp\_id][0] \leftarrow lane\_id$
    end if
    if $comm[warp\_id][0] \leftarrow lane\_id$ **then**
        $comm[warp\_id][0] \leftarrow range[0]$
        $comm[warp\_id][0] \leftarrow range[1]$
        $range[2] \leftarrow 0$
        $share\_vertex[warp\_id] = id\_sommet$
    end if
    $r\_gather \leftarrow comm[warp\_id][0] + lane\_id$
    $r\_gather\_end \leftarrow comm[warp\_id][2]$
    while $r\_gather < r\_gather\_end$ **do**
        $voisin \leftarrow R[r\_gather]$
        if $not \in tex\_visited$ **then**
            Adding in $tex\_visited$
            AtomicOr(out_queue,$voisin$)
        end if
        $r\_gather \leftarrow r\_gather + 32$
    end while
end while

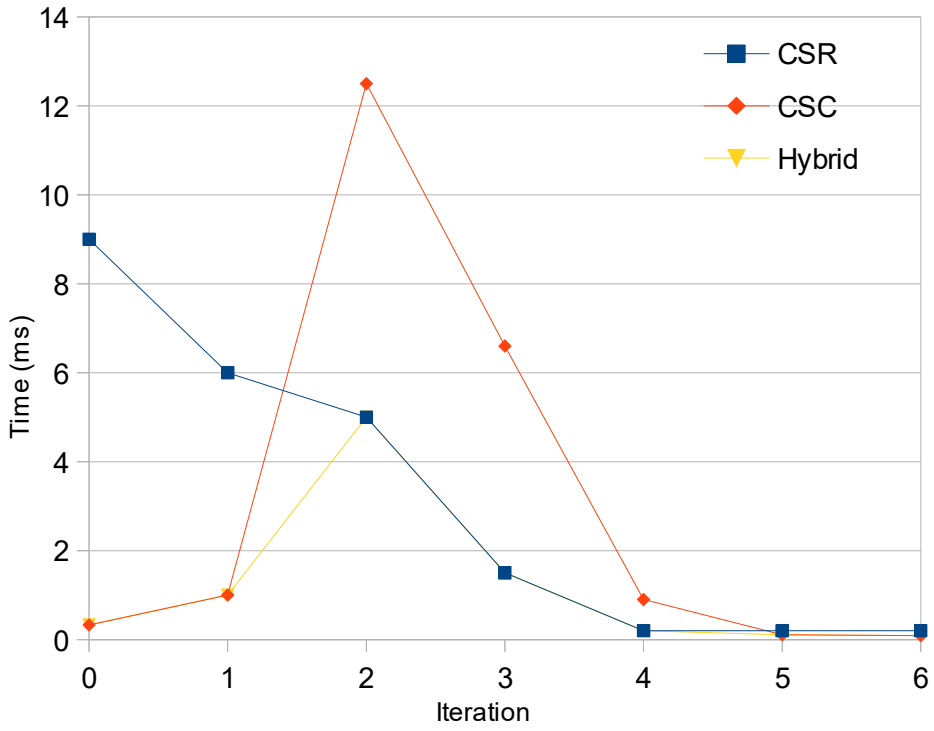

Figure 3.3: CSR and CSC approach comparison. On a 6 iterations BFS, the time with the two method is compared. The hybridization just takes the best time of each method

3.5 Results

3.5.1 CPU and GPU comparison

On figure 3.4 we present the single node implementation. Here we compare the best CPU implementation proposed by the Graph500 benchmark with our GPU implementation. On our cluster we worked with K20Xm GPUs. The GPU result is twice times better than the CPU one. We also carried out tests on some "general public" GPUs like GTX980 and GTX780Ti. The result is better on these GPUs because they do not implement the ECC memory and do not provide double precision CUDA cores. Indeed all the cores can be used for the Exploration phase.

3.5.2 Strong and weak scaling

On figure 3.6 and figure 3.5 we see the result of strong and weak scaling. In the strong scaling we used a $SCALE$ of 21 for different numbers of GPUs. The application scales up to 16 GPUs but then the data exchanges are too penalizing; performance for 64 GPUs is lower. Indeed as the problem scale does not change, the computational part is reduced compared to the communication one. Using 16 GPUs we were able to perform up to 4.80 GTEPS.

For the weak scaling, the $SCALE$ evolves with the number of GPUs. So the computation part grows and the limitation of communications is reduced. On figure 3.5, the problem SCALE is presented on each point. With our method we were able to reach up to 12 GTEPS using this scaling.

3.5.3 Communications and GPUDirect

Each node of the ROMEO supercomputer is composed of two CPU sockets and two GPUs, named GPU 0 and GPU 1. Yet the node just has one HCA (Host Channel Adapters), linked with CPU 0 and GPU 0. In order to use this link GPU 1 has to pass through a Quick Path

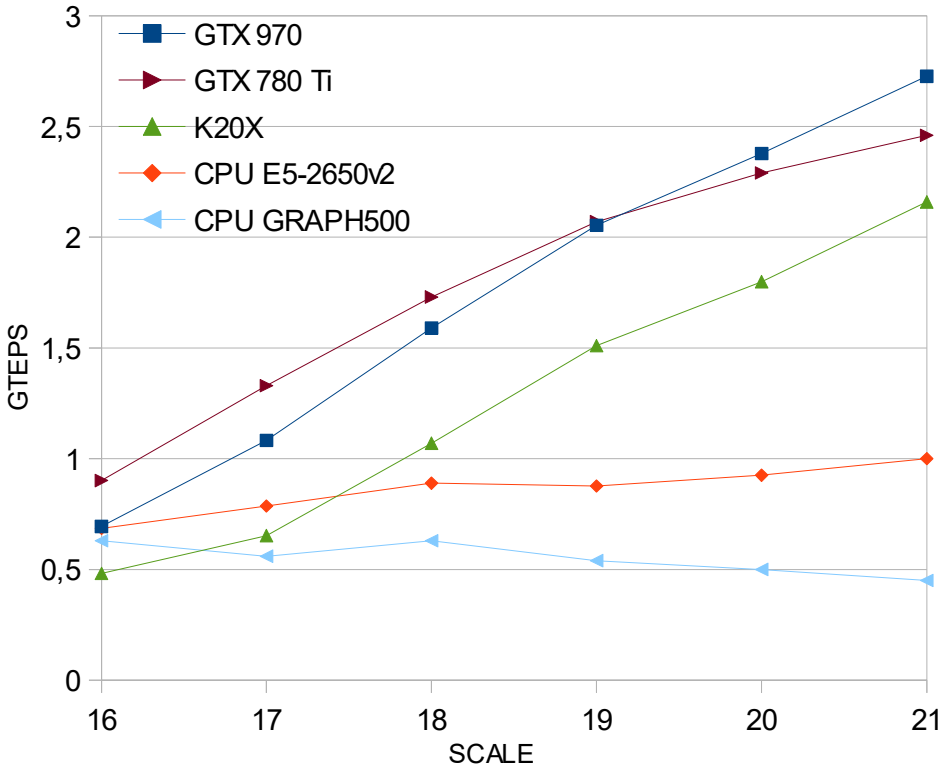

Figure 3.4: Single CPU and accelerators comparison. CPU Graph500 represent the best implementation proposed by the Graph500 website.

Interconnect link (QPI) between the two CPU sockets. This link considerably reduces the bandwidth available for node-to-node communication. Another problem is that the two GPUs have to share the same HCA for their communication.

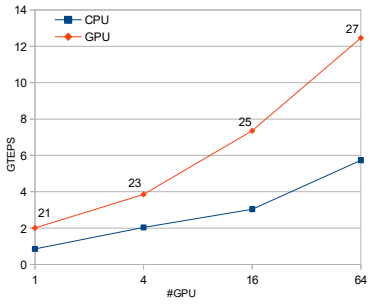

Figure 3.5: CPU *vs* GPU weak scaling. The number of CPUs is the same as the number of GPUs.

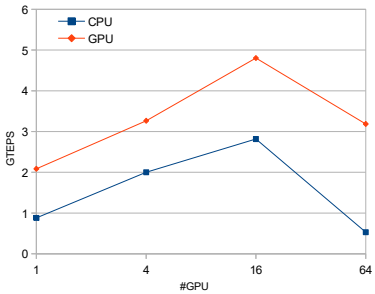

Figure 3.6: CPU *vs* GPU strong scaling. The *SCALE* is showed on the GPU line. The number of CPUs is the same as the number of GPUs.

On figure 3.7, the tests are based on the GPU-only implementation. First we worked with the two GPUs of the nodes. We were able to perform up to to *SCALE* 29 with 12 GTEPS. The GPUDirect implementation does not allow the communication with a QPI link. So in order to compare the results, we used only the GPU 0 of each node of the supercomputer. Based on our algorithm implementation we need to use a number 2^{2n} of GPUs. Then the tests on figure 3.7 are for 256 GPUs (with GPU 0 and GPU 1) and with 64 GPUs (using just GPU 0 only). Thus we were able to reach a better value of GTEPS. As the major limitation is the communications stage, using only GPU 0 allowed us to obtain about 13.70 GTEPS on the ROMEO supercomputer.

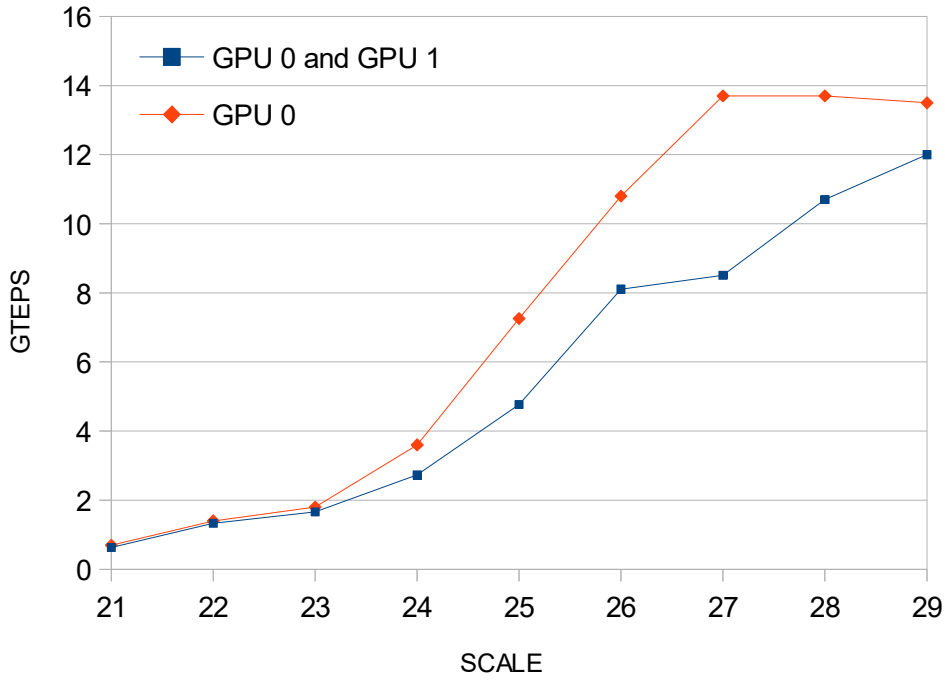

Figure 3.7: Full node GPUs *vs* GPU 0. The GPU 0 implementation not use the QPI link of the two CPU socket.

3.6 Conclusions

In this study we present an optimized implementation of the Graph500 benchmark for the ROMEO multi-GPU cluster. It is based on the BlueGene/Q algorithm and GPU optimization for BFS traversal by Merrill et al. This work highlights different key points. First, we have chosen a hybrid memory representation of graphs using both CSR and CSC. Although this representation requires more memoriy, it significantly reduces the computation workload and allows us to achieve outstanding performance. Second, the inter-node and intra-node communication is a critical bottleneck. Each compute node has two GPUs, however only one shares the same PCIe bridge with the Infiniband HCA that allows to take advantage of the GPUDirect technology. Third, due to the low compute power needed for BFS traversal, we get better performance by fully loading GPUs. Otherwise communication time cannot be overlapped with computation time. Thus to achieve the best performance we had to use only half of each node. Finally, using all these optimizations, we achieved satisfactory results. Indeed, by using GPUDirect on 64 GPUs, we are able to achieve 13,70 GTEPS. In this configuration CPUs are only used to synchronize GPUs kernels. All the communications are directly GPU to GPU using a CUDA-aware MPI library and GPUDirect.

These results will be published in the next Graph500 list. With a total of 13.70 GTEPS the ROMEO supercomputer could be ranked at the 91th position.

Today we can identify some interesting perspectives to carry on the study. Communication cost is the major limitation and a better control of load distribution is needed between communication and computation in order to obtain even better performance. Part of the solution might come from new technologies developed by Nvidia, such as the new PASCAL architecture or NVlink buses.

Bibliography

[BAP13] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[CPW+12] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12, Nov 2012.

[FDB+14] Zhisong Fu, Harish Kumar Dasari, Bradley Bebee, Martin Berzins, and Bradley Thompson. Parallel breadth first search on gpu clusters. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 110–118. IEEE, 2014.

[LCK+10] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *The Journal of Machine Learning Research*, 11:985–1042, 2010.

[MGG15] Duane Merrill, Michael Garland, and Andrew Grimshaw. High-performance and scalable gpu graph traversal. *ACM Transactions on Parallel Computing*, 1(2):14, 2015.