Part I

Complex systems

Introduction

The tools for comprehension of High Performance Computing from theory, hardware and software give us the basis to go toward optimizations and benchmarking. We show that hybrid architectures seems to be the way to reach exascale in few years but many optimizations need to be done to fit the energy envelope. Our study focus on the behavior of accelerators compared to classical processor on irregular application. We showed that the downside of accelerators seems to be the synchronization and regularized work. In this part we propose our metric putting forward accelerators behavior on irregular problems.

This part is decomposed in three chapters: The first one details the mains bottlenecks, pitfalls and walls in HPC. We give explanations on our problems choices for the chapters two and three of this part.

The first problem for our metric characterizing the behavior of accelerators and more specifically GPUs focus on irregular computations. We choose the problem of Langford, known in our laboratory, and show how we can take advantage of the accelerator for this kind of problem.

The second problem we choose is focused on irregular memory accesses and communications. It is based on a benchmark we presented in the previous part, the Graph500 benchmark.

Chapter 1

Walls and pitfalls

1.1 Introduction

Domain scientists, chemists, physicists, meteorologists, etc. always need better and more accurate simulations. This is why HPC always need better architecture and faster computation models. As presented on the previous parts several methods allows the computational power to grow from processors optimization but also memories and communications. Those limitations to reach tomorrow supercomputers are called *walls*. We start this chapter by describing them and were they are faced and in which benchmark.

We then give details on realistic domain scientists problems and irregularity. From those observation we propose the metrics that will be presented in the two other chapters of this part.

1.2 Walls

In this section we describe the main walls in HPC. Starting from memory wall, communication wall, power wall and finally computational wall.

1.2.1 Memory Wall

THis problem is targeting for the first time in [WM95]. The author explains that:

> We all know that the rate of improvement in microprocessor speed exceeds the rate of improvement in DRAM memory speed, each is improving exponentially, but the exponent for microprocessors is substantially larger than that for DRAMs.

The new release of accelerators also have an impact on the memory wall, the memory of the host processors and devices accelerators cannot be accessed directly and copies from one to the other are requested.

1.2.2 Communication wall

The multiplicity of racks, nodes and

1.2.3 Power wall

The energy consumption of nowadays and future supercomputers is the main wall in HPC. Indeed, an exascale supercomputer could be construct using several petascale supercomputer but, with todays architectures, will require a nuclear plant to operate. In this objective low energy consumption architectures need to be find. Power wall can be of two kind: the energy to power the machine itself and the many nodes, processors and accelerators but also, and not the least, the energy requires to handle the heat generated by the machine. For the second part many new technologies arise with direct water cooling in the supercomputer racks.

1.2.4 Computational wall

The computational wall is a conjugation of all the wall cited before. By increasing the memory wall, the energy consumption and the communications we can increase the overall computation of the supercomputer.

1.3 Benchmark

1.3.1 Irregular behavior

1.3.2 Our choices

1.4 Conclusion

Chapter 2

Computational Wall: Langford Problem

2.1 Introduction

Our aim is to determine the behavior of accelerators compared to classical processor in case of irregular-computationally heavy problems. For this purpose we choose the Langford problem which is an academic problem of combinatorial counting. We show the optimizations made to the regular processor algorithm to efficiently implement this application on GPU. We compare the two approaches from classical processor to GPU implementation. The results are then presented to show the acceleration using the whole ROMEO supercomputer.

C. Dudley Langford gave his name to a classic permutation problem [Gar56, Sim83]. While observing his son manipulating blocks of different colors, he noticed that it was possible to arrange three pairs of different colored blocks (yellow, red, blue) in such a way that only one block separates the red pair - noted as pair 1 - , two blocks separate the blue pair - noted as pair 2 - and finally three blocks separate the yellow one - noted as pair 3 - , see Fig. 2.1.

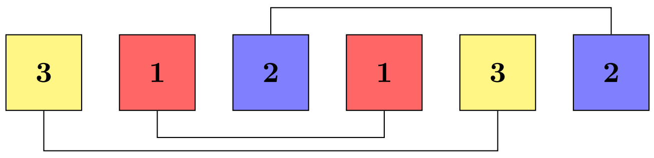

Figure 2.1: L(2,3) arrangement

This problem has been generalized to any number n of colors and any number s of blocks having the same color. $L(s, n)$ consists in searching for the number of solutions to the Langford problem, up to a symmetry. In November 1967, Martin Gardner presented $L(2, 4)$ (two cubes and four colors) as being part of a collection of small mathematical games and he stated that $L(2, n)$ has solutions for all n such that $n = 4k$ or $n = 4k - 1$ ($k \in \mathbb{N} \setminus \{0\}$). The central resolution method consists in placing the pairs of cubes, one after the other, on the free places and backtracking if no place is available (see Fig. 2.3 for detailed algorithm).

The Langford problem has been approached in different ways: discrete mathematics results, specific algorithms, specific encoding, constraint satisfaction problem (CSP), inclusion-exclusion ... [Mil99, Wal01, Smi00, Lar09]. In 2004, the last solved instance, $L(2, 24)$, was computed by our team[JK04] using a specific algorithm. (see Table 2.1); $L(2, 27)$ and $L(2, 28)$ have just been computed but no details were given.

The main efficient known algorithms are the following: the Miller backtrack method, the Godfrey algebraic method and the Larsen inclusion-exclusion method. The Miller one is based on backtracking and can be modeled as a CSP; it allowed us to move the limit of explicits

Instance	Solutions	Method	Computation time
L(2,3)	1	Miller algorithm	-
L(2,4)	1		-
...	...		...
L(2,16)	326,721,800		120 hours
L(2,19)	256,814,891,280		2.5 years (1999) DEC Alpha
L(2,20)	2,636,337,861,200	Godfrey algorithm	1 week
L(2,23)	3,799,455,942,515,488		4 days with CONFIIT
L(2,24)	46,845,158,056,515,936		3 months with CONFIIT
L(2,27)	111,683,611,098,764,903,232		-
L(2,28)	1,607,383,260,609,382,393,152		-

Table 2.1: Solutions and time for Langford problem using different methods

solutions building up to $L(2,21)$ but combinatorial explosion did not allow us to go further. Then, we use the Godfrey method to achieve $L(2,24)$ more quickly and then recompute $L(2,27)$ and $L(2,28)$, presently known as the last instances. The Larsen method is based on inclusion-exclusion [Lar09]; although this method is effective, practically the Godfrey one is better. The latest known work on the Langford Problem is a GPU implementation proposed in [ABL15] in 2015. Unfortunately this study does not provide any performance considerations but just gives the number of solution of $L(2,27)$ and $L(2,28)$.

2.2 Miller algorithm

In this part we present our multiGPU cluster implementation of the Miller's algorithm. First, we introduce the backtrack method. Then we present our implementation in order to fit the GPUs architecture. The last section presents our results.

2.2.1 CSP

Combinatorial problems are NP-complete[GJ79] and can be described as a SATISFIABILITY problems (SAT) using a polynomial transformation. They can be transformed into CSP formalism. A Constraint Satisfaction Problem (CSP), first introduced by Montanari[Mon74], is defined as a triple $< X, D, C >$ where: $X = \{X_1, ..., X_n\}$, a finite set of variables, $D = \{D_1, ..., D_n\}$, their finite domains of values et $C = \{C_1, ..., C_p\}$, a finite set of constraints.

The goal in this formalism is to assign values in D to n-uple X respecting all the C p-uple constraints. This approach is a large field of research. [AC14] developed *local search* and compares GPU to CPU. This first work brings to light that GPU is a real contributor to the global computation speed. [CDPD$^+$14] proposes a solver using *propagator* on a GPU architecture to solve CSP problems. [JAO$^+$11] cares about GPU weak points, loading bandwidth and global memory latency.

Considering a basic approach, combinatorial problems formed into CSP can be represented as a tree search. Each level corresponds to a given variable, with values in its domain. Leaves of the tree correspond to a complete assignment (all variables are set). If it meets all the constraints this assignment is called an acceptor state. Depending on the constraints set, the satisfiability evaluation can be made either on complete or partial assignment.

2.2.2 Backtrack resolution

As presented above the Langford problem is known to be a highly irregular combinatorial problem. We first present here the general tree representation and the ways we regularize the computation for GPUs. Then we show how to parallelize the resolution over a multiGPU cluster.

Langford's problem tree representation

As explained, CSP formalized problems can be transformed into tree evaluations.

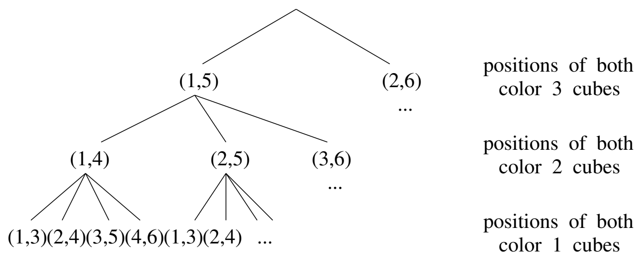
(1,5)
(2,6)
...
positions of both color 3 cubes
(1,4)
(2,5)
(3,6)
...
positions of both color 2 cubes
(1,3)(2,4)(3,5)(4,6)(1,3)(2,4) ...
positions of both color 1 cubes

Figure 2.2: Search tree for $L(2,3)$

In order to solve $L(2,n)$, we consider a tree of height n: see example of $L(2,3)$ in figure 2.2.

- Every level of the tree corresponds to a color.
- Each node of the tree corresponds to the placement of a pair of cubes without worrying about the other colors. Color p is represented at depth $n-p+1$, where the first node corresponds to the first possible placement (positions 1 and $p+2$) and i^{th} node corresponds to the placement of the first cube of color p in position i, $i \in [1,\ 2n-1-p]$.
- Solutions are leaves generated without any placement conflict.

There are many ways to browse the tree and find the solutions: *backtracking*, *forward-checking*, *backjumping*, etc [Pro93]. We limit our study to the naive *backtrack* resolution and choose to evaluate the variables and their values in a static order; in a depth-first manner, the solutions are built incrementally and if a partial assignment can be aborted, the branch is cut. A solution is found each time a leaf is reached.

The recommendation for performance on GPU accelerators is to use non test-based programs. Due to its irregularity, the basic *backtracking* algorithm, presented on figure 2.3, is not supposed to suit the GPU architecture. Thus a vectorized version is given when evaluating the assignments at the leaves' level, with one of the two following ways: assignments can be prepared on each tree node or totally set on final leaves before testing the satisfiability of the built solution (figure 2.4).

```
while not done do
 test pair            <- test
 if successful then
   if max depth then
     count solution
     higher pair
   else
     lower pair       <- remove
 else
   higher pair        <- add
```

Figure 2.3: Backtrack algorithm

```
for pair 1 positions
  assignment                    <- add
  for pair 2 positions
    assignment                  <- add
    for ...
      for pair n positions
        assignment              <- add
        if final test ok then
          count solution
```

Figure 2.4: Regularized algorithm

Data representation

In order to count every Langford problem solution, we first identify all possible combinations for one color without worrying about the other ones. Each possible combination is coded within an integer, one bit to 1 corresponding to a cube presence, a 0 to its absence. This is what we

	pair 1	pair 2	pair 3
1	0 0 0 1 0 1	0 0 1 0 0 1	0 1 0 0 0 1
2	0 0 1 0 1 0	0 1 0 0 1 0	1 0 0 0 1 0
3	0 1 0 1 0 0	1 0 0 1 0 0	
4	1 0 1 0 0 0		

Figure 2.5: Bitwise representation of pairs positions in $L(2, 3)$

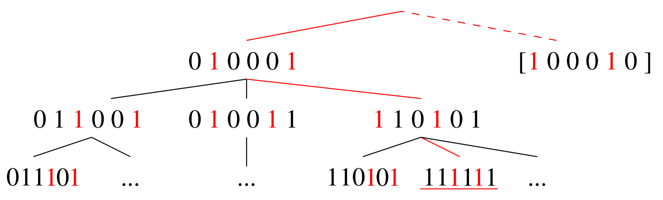

Figure 2.6: Bitwise representation of the Langford $L(2, 3)$ placement tree

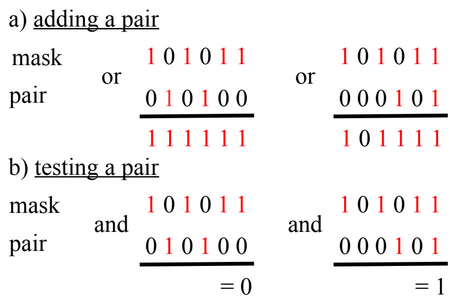

Figure 2.7: Testing and adding position

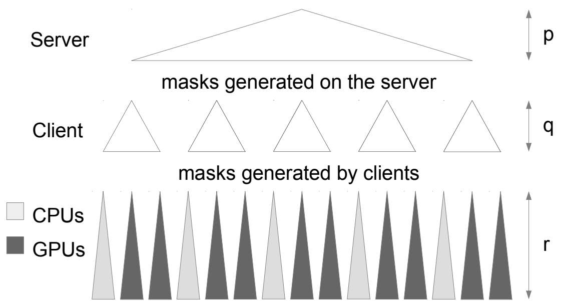

Figure 2.8: Server client distribution

called a *mask*. This way figure 2.5 presents the possible combinations to place the one, two and three weight cubes for the $L(2, 3)$ Langford instance.

Furthermore the masks can be used to evaluate the partial placements of a chosen set of colors: all the 1 correspond to occupied positions; the assignment is consistent *iff* there are as many 1 as the number of cubes set for the assignment.

With the aim to find solutions, we just have to go all over the tree and *sum* one combination of each of the colors: a solution is found *iff* all the bits of the sum are set to 1.

Each route on the tree can be evaluated individually and independently; then it can be evaluated as a thread on the GPU. This way the problem is massively parallel and can be, indeed, computed on GPU. figure 2.6 represents the tree masks' representation.

Specific operations and algorithms

Three main operations are required in order to perform the tree search. The first one, used for both backtrack and regularized methods, aims to add a pair to a given assignment. The second one, allowing to check if a pair can be added to a given partial assignment, is only necessary for the original backtrack scheme. The last one is used for testing if a global assignment is an available solution: it is involved in the regularized version of the Miller algorithm.

Add a pair: Top of figure 2.7 presents the way to add a pair to a given assignment. With a *binary or*, the new mask contains the combination of the original mask and of the added pair. This operation can be performed even if the position is not available for the pair (however the resulting mask is inconsistent).

Test a pair position: On the bottom part of the same figure, we test the positioning of a pair on a given mask. For this, it is necessary to perform a *binary and* between the mask and the pair.

$= 0$: *success*, the pair can be placed here

$\neq 0$: *error*, try another position

Final validity test: The last operation is for *a posteriori* checking. For example the mask 101111, corresponding to a leaf of the tree, is inconsistent and should not be counted among the solutions. The final placement mask corresponds to a solution *iff* all the places are occupied, which can be tested as $\neg mask = 0$.

Using this data representation, we implemented both *backtrack* and *regularized* versions of the Miller algorithm, as presented in figure 2.3 and 2.4.
The next section presents the way we hybridize these two schemes in order to get an efficient parallel implementation of the Miller algorithm.

2.2.3 Hybrid parallel implementation

This part presents our methodology to implement Miller's method on a multiGPU cluster.

Tasks generation: In order to parallelize the resolution we have to generate tasks. Considering the tree representation, we construct tasks by fixing the different values of a first set of variables [pairs] up to a given level. Choosing the development level allows to generate as many tasks as necessary. This leads to a *Finite number of Irregular and Independent Tasks* (*FIIT* applications [Kra99]).

Cluster parallelization: The generated tasks are independent and we spread them in a client-server manner: a server generates them and makes them available for clients. As we consider the cluster as a set of CPU-GPU(s) machines, the clients are these machines. At the machines level, the role of the CPU is, first, to generate work for the GPU(s): it has to generate sub-tasks, by continuing the tree development as if it were a second-level server, and the GPU(s) can be considered as second-level client(s).
The sub-tasks generation, at the CPU level, can be made in parallel by the CPU cores. Depending on the GPUs number and their computation power the sub-tasks generation rhythm may be adapted, to maintain a regular workload both for the CPU cores and GPU threads: some CPU cores, not involved in the sub-tasks generation, could be made available for sub-tasks computing.
This leads to the 3-level parallelism scheme presented in figure 2.8, where p, q and r respectively correspond to: (p) the server-level tasks generation depth, (q) the client-level sub-tasks generation one, (r) the remaining depth in the tree evaluation, *i.e.* the number of remaining variables to be set before reaching the leaves.

***Backtrack* and *regularized* methods hybridization:** The Backtrack version of the Miller algorithm suits CPU execution and allows to cut branches during the tree evaluation, reducing the search space and limiting the combinatorial explosion effects. A regularized version had to be developed, since GPUs execution requires synchronous execution of the threads, with as few branching divergence as possible; however this method imposes to browse the entire search space and is too time-consuming.
We propose to hybridize the two methods in order to take advantage of both of them for the multiGPU parallel execution: for tasks and sub-tasks generated at sever and client levels, the tree development by the CPU cores is made using the backtrack method, cutting branches as soon as possible [and generating only possible tasks]; when computing the sub-tasks generated at client-level, the CPU cores involved in the sub-tasks resolution use the backtrack method and the GPU threads the regularized one.

2.2.4 Experiments tuning

In order to take advantage of all the computing power of the GPU we have to refine the way we use them: this section presents the experimental study required to choose optimal settings.

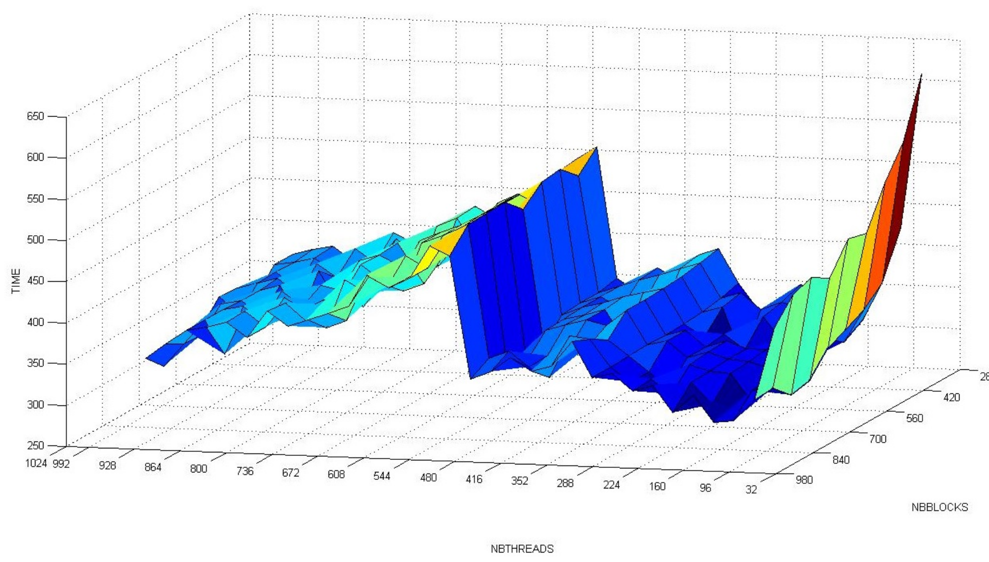

Figure 2.9: Time depending on grid and block size on $n = 15$

This tuning allowed us to prove our proposal on significant instances of the Langford problem.

Registers, blocks and grid: In order to use all GPUs capabilities, the first way was to fill the blocks and grid. To maximize occupancy (ratio between active warps and the total number of warps) NVIDIA suggests to use 1024 threads per block to improve GPU performances and proposes a CUDA occupancy calculator[1]. But, confirmed by the Volkov's results[Vol10], we experimented that better performances may be obtained using lower occupancy. Indeed, another critical criterion is the inner GPU registers occupation. The optimal number of registers (57 registers) is obtained by setting 9 pairs placed on the client for $L(2, 15)$, thus 6 pairs are remaining for GPU computation.

In order to tune the blocks and grid sizes, we performed tests on the ROMEO architecture. Figure 2.9 represents the time in relation with the number of blocks per grid and the number of threads per block. The most relevant result, observed as a local minimum on the 3D surface, is obtained near 64 or 96 threads per block; for the grid size, the limitation is relative to the GPU global memory size. It can be noted that we do not need shared memory because their are no data exchanges between threads. This allows us to use the total available memory for the L1 cache for each thread.

Streams: A client has to prepare work for GPU. There are four main steps: generate the tasks, load them into the device memory, process the task on the GPU and then get the results.

CPU-GPU memory transfers cause huge time penalties (about 400 cycles latency for transfers between CPU memory and GPU *device memory*). At first, we had no overlapping between memory transfer and kernel computation because the tasks generation on CPU was too long compared to the kernel computation. To reduce the tasks generation time we used OpenMP in order to use the eight available CPU cores. Thus CPU computation was totally hidden by memory transfers and GPU kernel computation. We tried using up to 7 streams; as shown by figure 2.10, using only two simultaneous streams did not improve efficiency because the four steps did not overlap completely; the best performances were obtained with three streams; the slow increase in the next values is caused by synchronization overhead and CUDA streams management.

[1]`http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

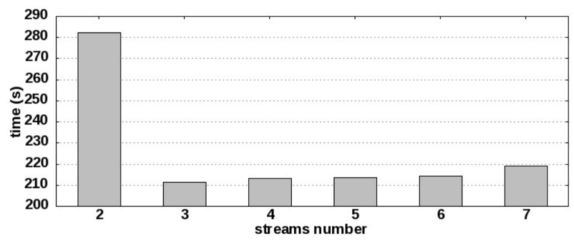

Figure 2.10: Computing time depending on streams number

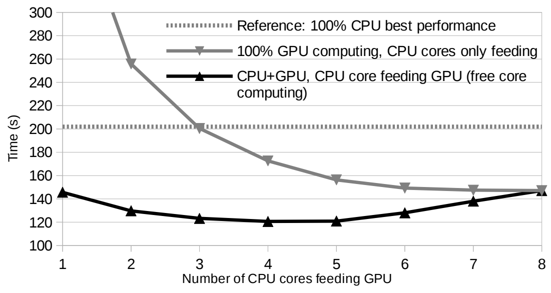

Figure 2.11: CPU cores optimal distribution for GPU feeding

Setting up the server, client and GPU depths: We now have to set the depths of each actor, server (p), client (q) and GPU (r) (see figure 2.8).

First we set the $r = 5$ for large instances because of the GPU limitation in terms of registers by threads, exacerbated by the use of numerous $64bits$ integers. For $r \geq 6$, we get too many registers (64) and for $r \leq 4$ the GPU computation is too fast compared to the memory load overhead.

Clients are the buffers between the server and the GPUs: $q = n - p - r$. So we have conducted tests by varying the server depth, p. The best result is obtained for $p = 3$ and performance decreases quickly for higher values. This can be explained since more levels on the server generates smaller tasks; thus GPU use is not long enough to overlap memory exchanges.

CPU: Feed the GPUs and compute The first work of CPU cores is to prepare tasks for GPU so that we can generate overlapping between memory load and kernel computation. In this configuration using eight cores to generate GPU tasks under-uses CPU computation power. It is the reason why we propose to use some of the CPU cores to take part of the sub-problems treatment. Figure 2.11 represents computation time in relation with different task distributions between CPU and GPU. We experimentally demonstrated that only 4 or 5 CPU cores are enough to feed GPU, the other ones can be used to perform backtrack resolution in competition with GPUs.

2.2.5 Results

Regularized method results We now can show the results obtained for our massively parallel scheme using the previous optimizations, comparing the computation times of successive instances of the Langford problem. These tests were performed on 20 nodes of the ROMEO supercomputer, hence 40 CPU/GPU machines.

The previous limit with Miller's algorithm was $L(2, 19)$, obtained in 1999 after 2.5 years of sequential effort and at the same time after 2 months with a distributed approach[Mil99]. Our computation scheme allowed us to obtain it in less than 4 hours (Table 2.2), this being not only due to Moore law progress.
Note that the computation is 1.6 faster with CPU+GPU together than using 8 CPU cores. In addition, the GPUs compute 200000× more nodes of the search tree than the CPUs, with a faster time.

The computation time between two different consecutive instances being multiplied by 10 approximately, this could allow us to obtain $L(2, 20)$ in a reasonable time.

Backtracking on GPUs It appears at first sight that using backtracking on GPUs without any regularization is a bad idea due to threads synchronization issues. But in order to compare CPU and GPU computation power in the same conditions we decide to implement the original backtrack method on GPU (see Fig. 2.3) with only minor modifications. In these conditions we

n	CPU (8c)	GPU (4c) + CPU (4c)
15	2.5	1.5
16	21.2	14.3
17	200.3	120.5
18	1971.0	1178.2
19	22594.2	13960.8

Table 2.2: Regularized method (seconds)

n	CPU (8c)	GPU (4c) + CPU (4c)
17	29.8	7.3
18	290.0	73.6
19	3197.5	803.5
20	–	9436.9
21	–	118512.4

Table 2.3: Backtrack (seconds)

observe very efficient work of the NVIDIA scheduler, which perfectly handles threads desynchronization. Thus we use the same server-client distribution as in 2.2.3, each client generates masks for both CPU and GPU cores. The workload is then statically distributed on GPU and CPU cores. Executing the backtrack algorithm on a randomly chosen set of sub-problems allowed us to set the GPU/CPU distribution ratio experimentally to 80/20%,

The experiments were performed on 129 nodes of the ROMEO supercomputer, hence 258 CPU/GPU machines and one node for the server. Table 2.3 shows the results with this configuration. This method first allowed us to perform the computation of $L(2, 19)$ in less than 15 minutes, 15× faster than with the regularized method; then, we pushed the limitations of the Miller algorithm up to $L(2, 20)$ in less than 3 hours and even $L(2, 21)$ in about 33 hours[2].

This exhibits the ability of the GPU scheduler to manage highly irregular tasks. It proves that GPUs are adapted even to solve combinatorial problems, which they were not supposed to be.

2.3 Godfrey's algebraic method

The previous part presents the Miller algorithm for the Langford problem, this method cannot achieve bigger instances than the $L(2, 21)$.
An algebraic representation of the Langford problem has been proposed by M. Godfrey in 2002. In order to break the limitation of $L(2, 24)$ we already used this very efficient problem specific method. In this part we describe this algorithm and optimizations, and then our implementation on multiGPU clusters.

2.3.1 Method description

Consider $L(2, 3)$ and $X = (X_1, X_2, X_3, X_4, X_5, X_6)$. It proposes to modelize $L(2, 3)$ by $F(X, 3) = (X_1X_3 + X_2X_4 + X_3X_5 + X_4X_6) \times (X_1X_4 + X_2X_5 + X_3X_6) \times (X_1X_5 + X_2X_6)$

In this approach each term represents a position of both cubes of a given color and a solution to the problem corresponds to a term developed as $(X_1X_2X_3X_4X_5X_6)$; thus the number of solutions is equal to the coefficient of this monomial in the development. More generally, the solutions to $L(2, n)$ can be deduced from $(X_1X_2X_3X_4X_5...X_{2n})$ terms in the development of $F(X, n)$.

If $G(X, n) = X_1...X_{2n}F(X, n)$ then it has been shown that:

$$\sum_{(x_1,...,x_{2n})\in\{-1,1\}^{2n}} G(X, n)_{(x_1,...,x_{2n})} = 2^{2n+1}L(2, n)$$

So $$\sum_{(x_1,...,x_{2n})\in\{-1,1\}^{2n}} (\prod_{i=1}^{2n} x_i) \prod_{i=1}^{n} \sum_{k=1}^{2n-i-1} x_k x_{k+i+1} = 2^{2n+1}L(2, n)$$

That allows to get $L(2, n)$ from polynomial evaluations. The computational complexity of $L(2, n)$ is of $O(4^n \times n^2)$ and an efficient big integer arithmetic is necessary. This principle can be

[2]Even if this instance has no interest since it is known to have no solution

optimized by taking into account the symmetries of the problem and using the Gray code: these optimizations are described below.

2.3.2 Optimizations

Some works focused on finding optimizations for this arithmetic method[Jai05]. Here we explain the symmetric and computation optimizations used in our algorithm.

Evaluation parity:

As $[F(-X,n)=F(X,n)]$, G is not affected by a global sign change. In the same way the global sign does not change if we change the sign of each pair or impair variable.

Using these optimizations we can set the value of two variables and accordingly divide the computation time and result size by four.

Symmetry summing:

In this problem we have to count each solution up to a symmetry; thus for the first pair of cubes we can stop the computation at half of the available positions considering $S'_1(x)=\sum_{k=1}^{n-1} x_k x_{k+2}$ instead of $S_1(x)=\sum_{k=1}^{2n-2} x_k x_{k+2}$. The result is divided by 2.

Sums order:

Each evaluation of $S_i(x)=\sum_{k=1}^{2n-i-1} x_k x_{k+i+1}$, before multiplying might be very important regarding to the computation time for this sum. Changing only one value of x_i at a time, we can recompute the sum using the previous one without global recomputation. Indeed, we order the evaluations of the outer sum using Gray code sequence. Then the computation time is considerably reduced.

Based on all these improvements and optimizations we can use the Godfrey method in order to solve huge instances of the Langford problem. The next section develops the main issues of our multiGPU architecture implementation.

2.3.3 Implementation details

In this part we present the specific adaptations required to implement the Godfrey method on a multiGPU architecture.

Optimized big integer arithmetic:

In each step of computation, the value of each S_i can reach $2n-i-1$ in absolute value, and their product can reach $\frac{(2n-2)!}{(n-2)!}$. As we have to sum the S_i product on 2^{2n} values, in the worst case we have to store a value up to $2^{2n}\frac{(2n-2)!}{(n-2)!}$, which corresponds to 10^{61} for $n=28$, with about 200 bits.

So we need few big integer arithmetic functions. After testing existing libraries like GMP for CPU or CUMP for GPU, we came to the conclusion that they implement a huge number of functionalities and are not really optimized for our specific problem implementation: product of "small" values and sum of "huge" values.

Finally, we developed a light CPU and GPU library adapted to our needs. In the sum for example, as maintaining carries has an important time penalty, we have chosen to delay the spread of carries by using buffers: carries are accumulated and spread only when useful (for example when the buffer is full). Fig. 2.12 represents this big integer handling.

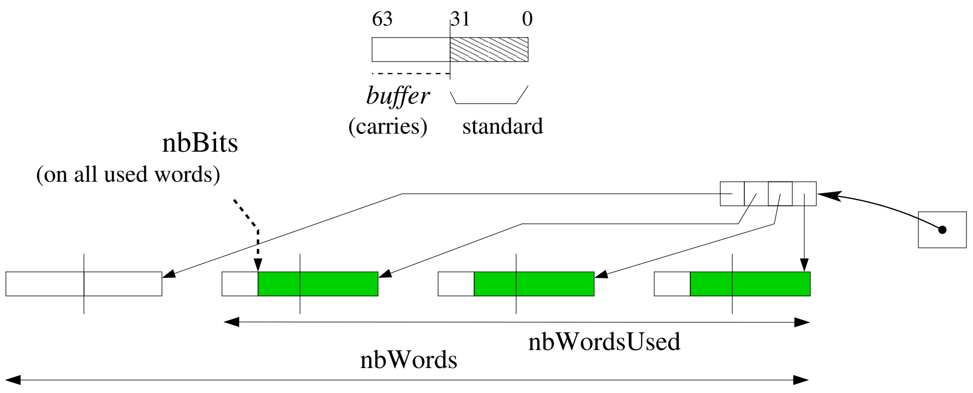

Figure 2.12: Big integer representation, 64 bits words

Gray sequence in memory:

The Gray sequence cannot be stored in an array because it would be too large (it would contain 2^{2n} byte values). This is the reason why only one part of the Gray code sequence is stored in memory and the missing terms are directly computed from the known ones using arithmetic considerations. The size of the stored part of the Gray code sequence is chosen to be as large as possible to be contained in the processor's cache memory, the L1 cache for the GPUs threads: so the accesses are fastened and the computation of the Gray code is optimized. For an efficient use of the E5-2650 v2 ROMEO's CPUs, which disposes of 20 MB of level-3 cache, the CPU Gray code sequence is developed recursively up to depth 25. For the K20Xm ROMEO's GPUs, which dispose of 8 KB of constant memory, the sequence is developed up to depth 15. The rest of the memory is used for the computation itself.

Tasks generation and computation:

In order to perform the computation of the polynomial, two variables can be set among the $2n$ available. For the tasks generation we choose a number p of variables to generate 2^p tasks by developing the evaluation tree to depth p.

The tasks are spread over the cluster, either synchronously or asynchronously.

Synchronous computation: A first experiment was carried out with an MPI distribution of the tasks of the previous model. Each MPI process finds its tasks list based on its process id; then converting each task number into binary gives the task's initialization. The processes work independently; finally the root process ($id = 0$) gathers all the computed numbers of solutions and sums them.

Asynchronous computation: In this case the tasks can be computed independently. As with the synchronous computation, the tasks' initializations are retrieved from their number. Each machine can get a task, compute it, and then store its result; then when all the tasks have been computed, the partial sums are added together and the total result is provided.

2.3.4 Experimental settings

This part presents the experimental context and methodology, and the way the experiments were carried out. This study has similar goals as for the Miller's resolution experiments.

Experimental methodology:

We present here the way the experimental settings were chosen. Firstly we define the tasks distribution, secondly we set the number of threads per GPU block; finally, we set the CPU/GPU distribution.

Tasks distribution depth: This value being set it is important to get a high number of blocks to maintain sufficient GPU load. Thus we have to determine the best number of tasks for the distribution. As presented in part 2.3.3 the number p of bits determines 2^p tasks. On the one hand, too many tasks are a limitation for the GPU that cannot store all the tasks in its 6GB memory. On the other hand, not enough tasks means longer tasks and too few blocks to fill the GPU grid. Fig. 2.14 shows that for the $L(2, 23)$ instance the best task number is with generation depth 28.

Number of threads per block: In order to take advantage of the GPU computation power, we have to determine the threads/block distribution. Inspired by our experiments with Miller's algorithm we know that the best value may appear at lower occupancy. We perform tests on a given tasks set varying the threads/block number and grid size associated. Fig. 2.13 presents the tests performed on the $n = 20$ problem: the best distribution is around 128 threads per block.

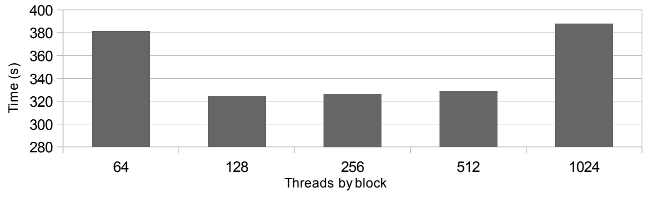

Figure 2.13: $L(2, 20)$, number of threads per block

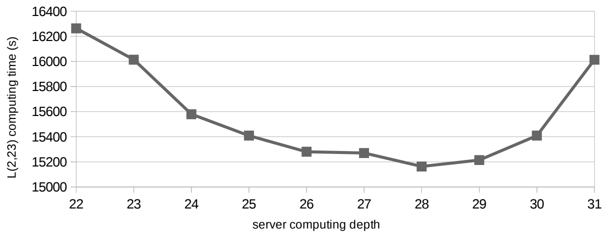

Figure 2.14: Influence on server generation depth

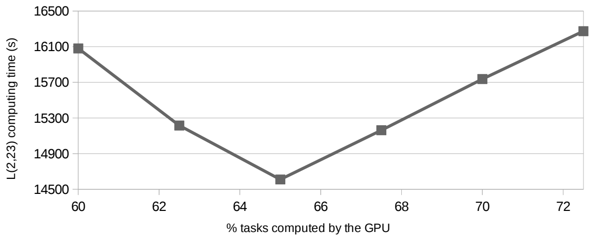

Figure 2.15: Influence of tasks repartition

CPU vs GPU distribution:

The GPU and CPU computation algorithm will approximately be the same. In order to take advantage of all the computational power of both components we have to balance tasks between CPU and GPU. We performed tests by changing the CPU/GPU distribution based on simulations on a chosen set of tasks. Fig. 2.15 shows that the best distribution is obtained when the GPU handles 65% of the tasks. This optimal load repartition directly results from the intrinsics computational power of each component; this repartition should be adapted if using a more powerful GPU like Tesla K40 or K80.

Computing context:

As presented in part **??**, we used the ROMEO supercomputer to perform our tests and computations. On this supercomputer SLURM[JG03] is used as a reservation and job queue manager. This software allows two reservation modes: a static one-job limited reservation or the opportunity to dynamically submit several jobs in a Best-Effort manner.

Static distribution: In this case we used the synchronous distribution presented in 2.3.3. We submited a reservation with the number of MPI processes and the number of cores per process. This method is useful to get the results quickly if we can get at once a large amount of computation resources. It was used to perform the computation of small problems, and even for $L(2,23)$ and $L(2,24)$.
As an issue, it has to be noted that it is difficult to quickly obtain a very large reservation on such a shared cluster, since many projects are currently running.

Best effort: SLURM allows to submit tasks in the specific Best-Effort queue, which does not count in the user *fair-share*. In this queue, if a node is free and nobody is using it, the reservation is set for a job in the best effort queue for a minimum time reservation. If another user asks for a reservation and requests this node, the best effort job is killed (with, for example, a SIGTERM signal). This method, based on asynchronous computation, enables a maximal use of the computational resources without blocking for a long time the entire cluster.

For $L(2,27)$ and even more for $L(2,28)$ the total time required is too important to use the whole machine off a challenge period, thus we chose to compute in a Best-Effort manner. In order to fit with this submission method we chose a reasonable time-per-task, sufficient to optimize the treatments with low loading overhead, but not too long so that killed tasks are not too penalizing for the global computation time. We empirically chose to run 15-20 minute tasks and thus we considered $p = 15$ for $n = 27$ and $p = 17$ for $n = 28$.

The best effort based algorithm is presented on Fig. 2.16. The task handler maintains a maximum of 256 tasks in the queue; in addition the entire process is designed to be fault-tolerant since killed tasks have to be launched again. When finished, the tasks generate an ouput containing the number of solutions and computation time, that is stored as a file or database entry. At the end the outputs of the different tasks are merged and the global result can be provided.

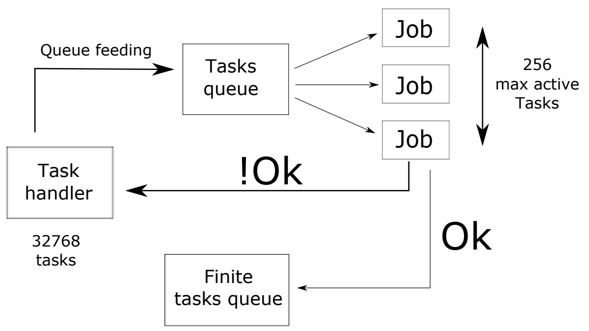

Figure 2.16: Best-effort distribution

2.3.5 Results

After these optimizations and implementation tuning steps, we conducted tests on the ROMEO supercomputer using best-effort queue to solve $L(2,27)$ and $L(2,28)$. We started the experiment after an update of the supercomputer, that implied a cluster shutdown. Then the machine was restarted and was about 50% idle for the duration of our challenge. The computation lasted less than 2 days for $L(2,27)$ and 23 days for $L(2,28)$. The following describes performances considerations.

Computing effort - For $L(2,27)$, the effective computation time of the 32,768 tasks was about 30 million seconds (345.4 days), and 165,000" elapsed time (1.9 days); the average time of the tasks was 911", with a standard deviation of 20%. For the $L(2,28)$ 131,072 tasks the total computation time was about 1365 days (117 million seconds), as 23 day elapsed time; the tasks lasted 1321" on average with a 12% standard deviation.

Best-effort overhead - With $L(2,27)$ we used a specific database to maintain information concerning the tasks: 617 tasks were aborted [by regular user jobs] before finishing (1.9%), with an average computing time of 766" (43% of the maximum requested time for a task). This consumed 472873", which overhead represents 1.6% of the effective computing effort.

Cluster occupancy - Fig. 2.17 presents the tasks resolution over the two computation days for $L(2,27)$. The experiment elapse time was 164700" (1.9 days). Compared to the effective computation time, we used an average of 181.2 machines (CPU-GPU couples): this represents 69.7% of the entire cluster.

Fig. 2.18 presents the tasks resolution flow during the 23 days computation for $L(2,28)$. We used about 99 machines, which represents 38% of the 230 available nodes.

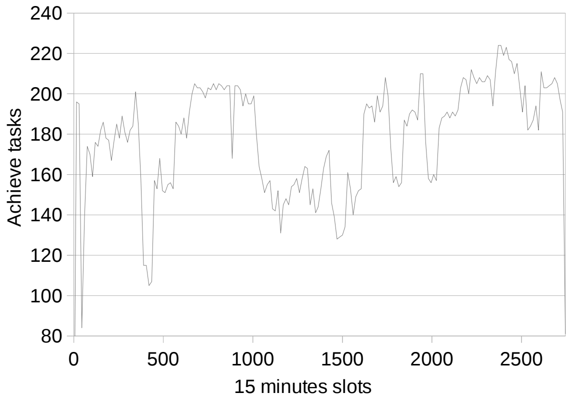

Figure 2.17: $L(2,27)$ tasks grouped by 15" slots

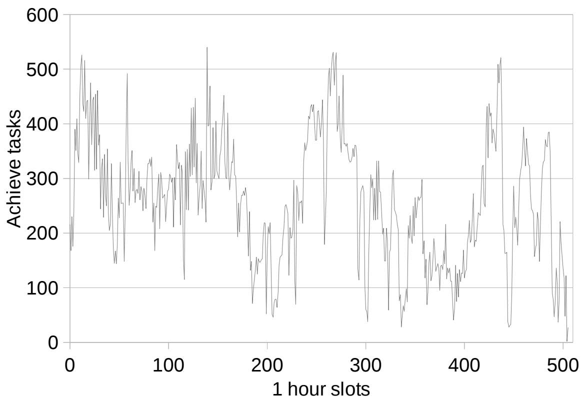

Figure 2.18: $L(2,28)$ tasks grouped by 1 hour slots

For $L(2,27)$, these results confirm that the computation took great advantage of the low occupancy of the cluster during the experiment. This allowed us to obtain a weak best-effort overhead, and an important cluster occupancy. Unfortunately for $L(2,28)$ on such a long period we got a lower part of the supercomputer dedicated to our computational project. Thus we are confident in good perspectives for the $L(2,31)$ instance if computed on an even larger cluster or several distributed clusters.

2.4 Conclusion

This study presents two methods to solve the Langford pairing problem on multiGPU clusters. In its first part the Miller's algorithm is presented. Then to break the problem limitations we show optimizations and implementation of Godfrey's algorithm.

2.4.1 CSP resolution method

As any combinatorial problem can be represented as a CSP, the Miller algorithm can be seen as general resolution scheme based on the backtrack tree browsing. A three-level tasks generation allows to fit the multiGPU architecture. MPI or Best-Effort are used to spread tasks over the cluster, OpenMP for the CPU cores distribution and then CUDA to take advantage of the GPU computation power. We were able to compute $L(2,20)$ with this regularized method and to get an even better time with the basic backtrack. This proves the proposed approach and also exhibits that the GPU scheduler is very efficient at managing highly divergent threads.

2.4.2 MultiGPU clusters and best-effort -

In addition and with the aim to beat the Langford limit we present a new implementation of the Godfrey method using GPUs as accelerators. In order to use the supercomputer ROMEO, which is shared by a large scientific community, we have implemented a distribution that does not affect the machine load, using a best-effort queue. The computation is fault-tolerant and totally asynchronous.

Langford problem results: This study enabled us to compute $L(2, 27)$ and $L(2, 28)$ in respectively less than 2 days and 23 days on the University of Reims ROMEO supercomputer. The total number of solutions is:

L(2,27) = 111,683,611,098,764,903,232
L(2,28) = 1,607,383,260,609,382,393,152

GPU benefit: This study shows the benefit of using GPUs as accelerators for combinatorial problems. In Miller's algorithm they handle 80% of the computation effort and 65% in Godfrey's. As a near-term prospect, we want to scale and show that it is possible to use the order of 1000 or more GPUs for pure combinatorial problems.
The next step of this work is to generalize the method to optimization problems. This adds an order of complexity since shared information has to be maintained over a multiGPU cluster.

We can conclude that even on irregular problems accelerators like GPU can be use and show better results than classical processors.

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:     $In \leftarrow \{v_s\}$;
3:     $Vis \leftarrow \{v_s\}$;
4:     $P(v) \leftarrow \perp \forall v \in V$;
5:     while $In \neq \emptyset$ do
6:         $Out \leftarrow \emptyset$
7:         for $u \in In$ do
8:             for $v | (u, v) \in E$ do
9:                 if $v \notin Vis$ then
10:                    $Out \leftarrow Out \cup \{v\}$;
11:                    $Vis \leftarrow Vis \cup \{v\}$;
12:                    $P(v) \leftarrow u$;
13:                end if
14:            end for
15:        end for
16:        $In \leftarrow Out$
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

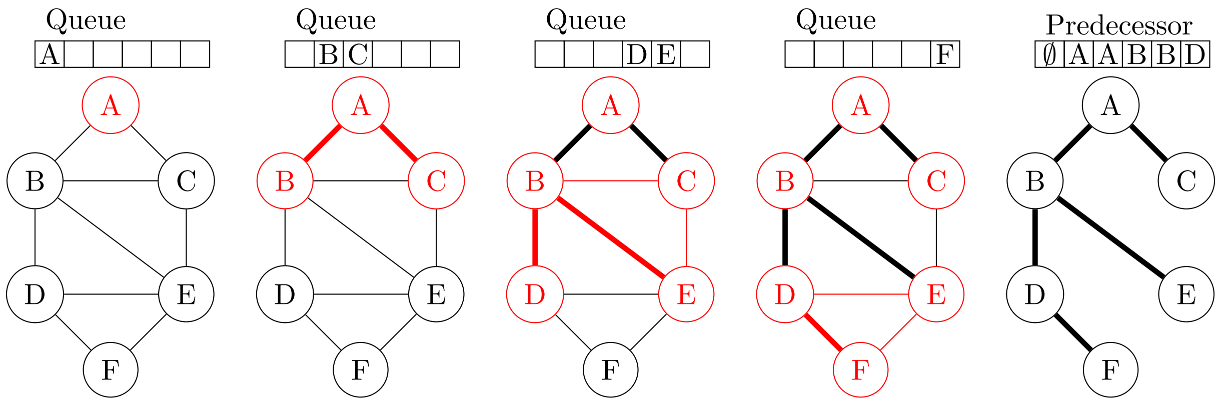

Figure 3.1: Example of Breadth First Search in an undirected graph

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

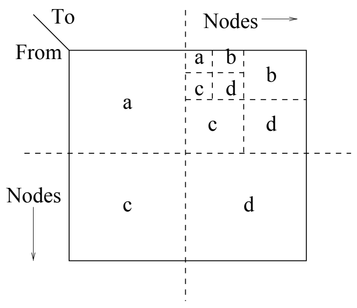

Figure 3.2: Kronecker generation scheme based on edge probability

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

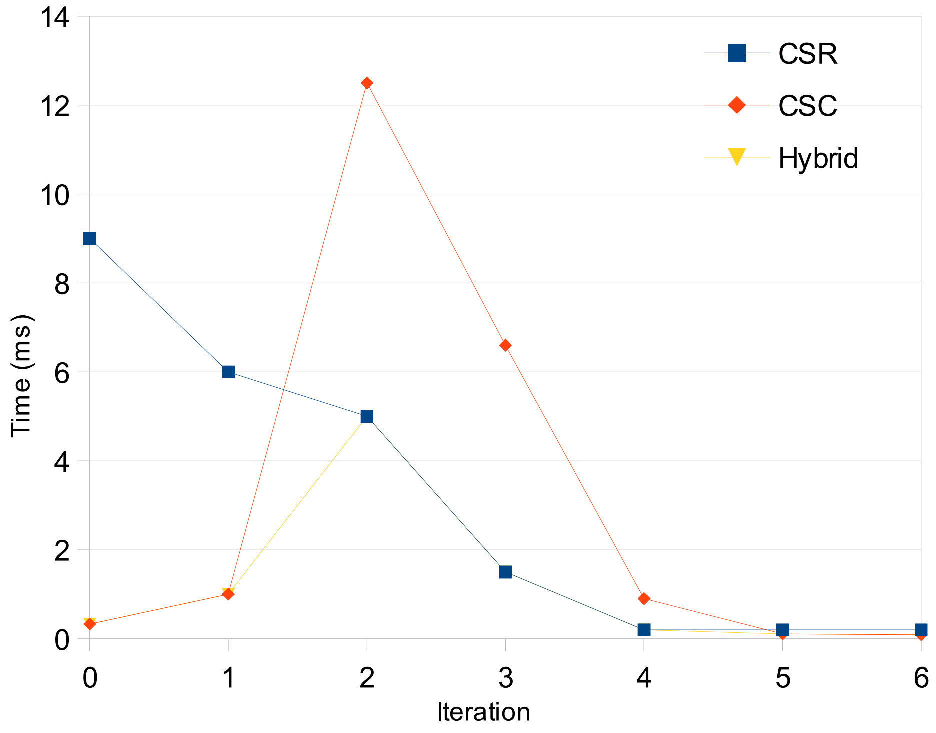

Figure 3.3: CSR and CSC approach comparison. On a 6 iterations BFS, the time with the two method is compared. The hybridization just takes the best time of each method

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

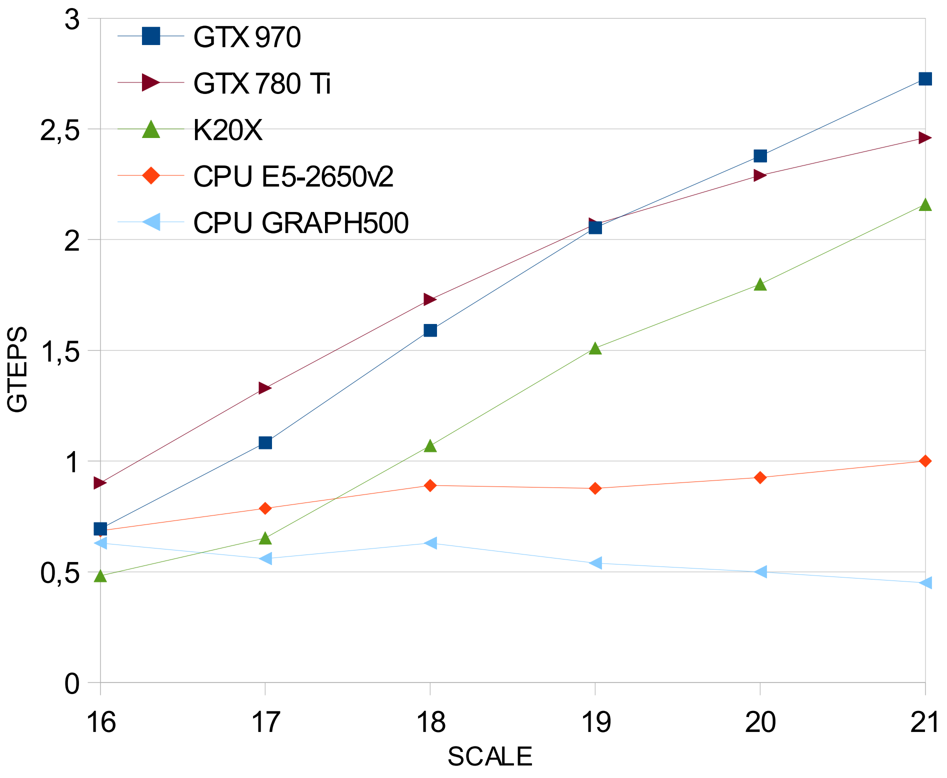

Figure 3.4: Single CPU and accelerators comparison. CPU Graph500 represent the best implementation proposed by the Graph500 website.

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.

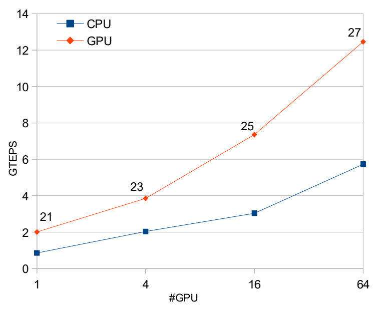

Figure 3.5: CPU *vs* GPU weak scaling. The number of CPUs is the same as the number of GPUs.

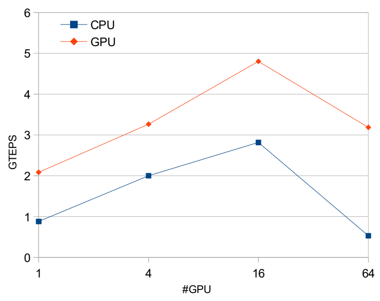

Figure 3.6: CPU *vs* GPU strong scaling. The *SCALE* is showed on the GPU line. The number of CPUs is the same as the number of GPUs.

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.

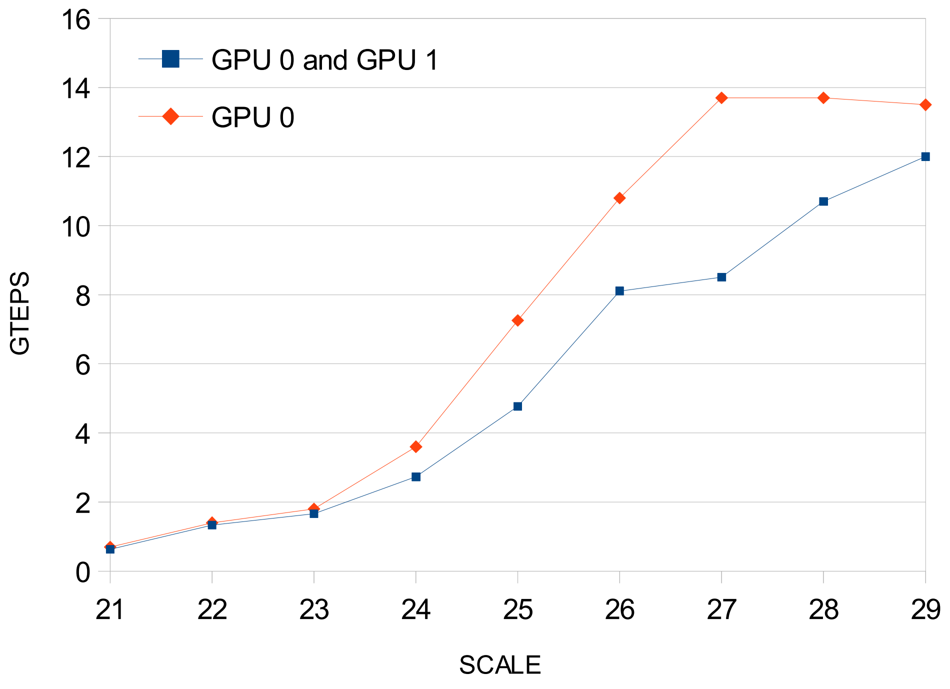

Figure 3.7: Full node GPUs *vs* GPU 0. The GPU 0 implementation not use the QPI link of the two CPU socket.

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.

Conclusion

In this part we show the real advantage of accelerators toward classical processor utilization. In the Langford problem the GPU have an acceleration of ***** regarding processors. It allows us to beat a new world record in term of computational speed in 2015 solving the last instance $L(2,28)$.

In the Graph500

Bibliography

[ABL15] Ali Assarpour, Amotz Barnoy, and Ou Liu. Counting the number of langford skolem pairings. 2015.

[AC14] Alejandro Arbelaez and Philippe Codognet. A gpu implementation of parallel constraint-based local search. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 648–655. IEEE, 2014.

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

[CDPD$^+$14] Federico Campeotto, Alessandro Dal Palu, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli. Exploring the use of gpus in constraint solving. In *Practical Aspects of Declarative Languages*, pages 152–167. Springer, 2014.

[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.

[Gar56] Martin Gardner. *Mathematics, magic and mystery.* Dover publication, 1956.

[GJ79] M. R. Garey and D. S. Johnson. *Computer and Intractability.* Freeman, San Francisco, CA, USA, 1979.

[Jai05] Christophe Jaillet. *In french: Résolution parallèle des problèmes combinatoires.* Phd, Université de Reims Champagne-Ardenne, France, December 2005.

[JAO$^+$11] John Jenkins, Isha Arkatkar, John D Owens, Alok Choudhary, and Nagiza F Samatova. Lessons learned from exploring the backtracking paradigm on the gpu. In *Euro-Par 2011 Parallel Processing*, pages 425–437. Springer, 2011.

[JG03] Morris Jette and Mark Grondona. *SLURM : Simple Linux Utility for Resource Management.* U.S. Departement of Energy, June 23, 2003.

[JK04] Christophe Jaillet and Michaël Krajecki. Solving the langford problem in parallel. In *International Symposium on Parallel and Distributed Computing*, pages 83–90, Cork, Ireland, July 2004. IEEE Computer Society.

[Kra99] Michaël Krajecki. An object oriented environment to manage the parallelism of the fiit applications. In *Parallel Computing Technologies*, pages 229–235. Springer, 1999.

[Lar09] J Larsen. Counting the number of skolem sequences using inclusion exclusion. 2009.

[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.

[Mil99] J.E. Miller. Langford's problem: http://dialectrix.com/langford.html, 1999.

[Mon74] U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Pictures Processing. *Information Sciences*, 7:95–132, 1974.

[Pro93] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993.

[Sim83] James E. Simpson. Langford sequences: perfect and hooked. *Discrete Math*, 44(1):97–104, 1983.

[Smi00] B. Smith. Modelling a Permutation Problem. In *Proceedings of ECAI'2000, Workshop on Modelling and Solving Problems with Constraints, RR 2000.18*, Berlin, 2000.

[Vol10] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

[Wal01] T. Walsh. Permutation problems and channelling constraints. Technical Report APES-26-2001, APES Research Group, January 2001.

[WM95] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.