Algorithmic scheme for hybrid computing with CPU, Xeon-Phi/MIC and GPU devices on a single machine

Sylvain CONTASSOT-VIVIER [a] and Stephane VIALLE [b]
[a] *Loria - UMR 7503, Université de Lorraine, Nancy, France*
[b] *UMI 2958, Georgia Tech - CNRS, CentraleSupelec, University Paris-Saclay, 57070 METZ, France*

Abstract. In this paper, we address the problem of the efficient parallel exploitation of different types of computing devices inside a single machine, to solve a scientific problem. As a first step, we apply our scheme to the Jacobi relaxation. Despite its simplicity, it is a good example of iterative process for scientific simulation. Then, we evaluate and analyze the performance of our parallel implementation on two configurations of hybrid machine.

Keywords. Accelerator, Xeon-Phi/MIC, GPU, hybrid computing, heterogeneous computing, offload computing.

Introduction

According to the hardware evolution in the last decades, the architecture of parallel systems becomes more and more complex. In particular, the development of many-core devices such as GPU (Graphical Processing Unit) and MIC (Many Integrated Cores) have induced an additional hierarchical level of parallelism in supercomputers. Indeed, current parallel systems are typically organized as big clusters of nodes and the many-core devices provide much larger computational power at the node level. However, the efficient programming of all that gathered power is still a major difficulty in the domain of High Performance Computing, partly due to the hierarchy in the system and to the communications between the different levels. The main issue is to design and implement efficient parallel schemes, as general as possible, that allows an efficient cooperation between all the computing units in a parallel system. The study presented in this paper is, to the best of our knowledge, one of the first attempt to solve a scientific application by using three different types of computing units inside a single node: the CPU cores, a GPU and a MIC.

After a brief review of the previous works over the programming of *hybrid* machines (containing different kinds of computing devices) in Section 1, our application and hardware testsbeds are described in Section 2. Then, Section 3 details the different algorithms designed and implemented for each kind of device (CPU, GPU, MIC). Finally, a multiple devices solution is proposed in Section 4. An experimental performance comparison and analysis, proposed in Section 5, allows us to evaluate the efficiency of our scheme and to point out the major difficulties in such cooperation.

1. Related works

In the past, we have investigated some ways to efficiently design algorithms and codes for hybrid nodes (one PC with a GPU) and clusters of hybrid nodes (cluster of multi-core nodes with GPUs) [1,2]. Overlapping of computations with communications was a key point to achieve high performance on hybrid architectures. The processing speed of each node increases when using accelerators, while interconnection networks remains unchanged, and data transfer times between CPU main memory and accelerator memory introduce new overheads.

Today, scientific computing on GPU accelerators is common, but using Xeon Phi accelerators has still to be explored, although some comparison works have been achieved. In [3], authors point out the need to optimize data storage and data accesses in different ways on GPU and Xeon Phi, but no dot attempt to use both accelerators in the same program. Another strategy is to use a generic programming model and tool to program heterogeneous architectures, like OpenCL [4]. But usually it does not hide the different architectures requirements to achieve optimal performance, and it still requires an (important) algorithmic effort to design high performance codes running concurrently on different accelerators.

2. Benchmark application and testbeds

2.1. Jacobi relaxation application

According to the scope of this paper (hybrid computing with CPU, GPU and MIC), we have chosen an application with a regular domain, that is quite representative of the scientific problems adapted to the constraints of the studied devices (especially the GPU). Indeed, the objective of this work is not to propose parallel schemes for general numerical methods but to study the best ways to make the different internal devices of a hybrid computer work together.

The Jacobi relaxation is a classical iterative process providing a simple modeling of heat transfer or electrical potential diffusion in 2D or 3D discrete domain (regular grid). The objective of this application is to compute the stable state over the entire domain for some given fixed boundary conditions. An explicit iterative 2D scheme is performed as:

$$crt[l][c] = \frac{pre[l-1][c] + pre[l][c-1] + pre[l][c] + pre[l][c+1] + pre[l+1][c]}{5} \tag{1}$$

where $crt[l][c]$ is the value of the grid point at line l and column c at the current iteration, while $pre[l][c]$ gives the value of the same grid point at the previous iteration. The other four grid points involved are the direct neighbors (in 4-connexity) of the current point.

This iterative process is performed until the termination condition is reached. As the quantities in the grid are generally coded by real numbers, the strict stabilization may not be reachable in reasonable time. Among the different solutions to get around this problem, we have chosen to fix the number of iterations. This presents the advantage of providing a complete and accurate control over the amount of computation. In fact, this parameter is essential to study some aspects of parallel algorithms, such as the scalability.

2.2. Testbeds

The machine used at CentraleSupelec (CS) is a Dell R720 server with two 6-cores Intel(R) Xeon(R) CPU E5-2620 at 2.10GHz, and two accelerators on separate PCIe buses.

One accelerator is an Intel MIC *Xeon-Phi 3120* with 57 physical cores at 1.10 GHz, supporting 4 threads each. The other one is a Nvidia GPU *GeForce GTX Titan Black* (Kepler architecture) with 2880 CUDA cores. The machine used at Loria is a Dell R720 server with two 8-cores Intel(R) Xeon(R) CPU E5-2640 at 2.00GHz, and two accelerators on separate PCIe buses. One accelerator is an Intel MIC *Xeon-Phi 5100* with 60 physical cores at 1.05 GHz supporting 4 threads each. The other one is a Nvidia GPU *Tesla K40m* (Kepler architecture) with 2880 CUDA cores. The CS machine uses CentOs 7.1.1503 and the Intel compiler v15.0.0 whereas the Loria machine uses CentOs 6.6 and the Intel compiler v14.0.3.

In this paper, we study the behavior of our parallel scheme on these two different machines, taking into account the relative computing powers of CPU, GPU and MIC.

3. Optimized kernels for single architecture and device

3.1. Multi-core CPU with OpenMP

A first version of the multi-core CPU kernel to perform the Jacobi relaxation consists in a classical domain decomposition in horizontal strips through the cores. This is achieved by inserting the parallelism at the level of the loop over the lines of the domain inside the main iterative loop. That main loop updates the current version of the grid according to the previous one. The corresponding parallel scheme is given in Listing 1.

Listing 1: Simple OpenMP scheme for the Jacobi relaxation

```
#pragma omp parallel num_threads(nbT) // Threads creation
{
  ... // Local variables and array initializations
  for(iter=0; iter<nbIters; ++iter){  // Main iterative loop
    // Parallel parsing of horizontal strips of the domain
    #pragma omp for
    for(lig=1; lig<nLig-1; ++lig){   // Lines in each strip
      for(col=1; col<nCol-1; ++col){ // Columns in each line
        ind = lig * nCol + col;
        crt[ind] = 0.2 * (prec[ind - nCol] + prec[ind-1] + prec[ind]↩
             + prec[ind+1] + prec[ind+nCol]);
      }
    }
    #pragma omp single
    { ... // Arrays exchange for next iteration (avoids copy) }
  }
}
```

Although this simple version works quite well for small and medium sizes of grids, it is not fully scalable for grids with large lines, due to the L2 cache use that is not optimized. We remind the reader that one L2 cache is present in each core of a CPU. So, a second version has been implemented explicitly taking into account the cache management in each core. Due to the data dependencies in our application and to the cache mechanism, the modifications mainly consist in changing the update order of the elements in each horizontal strip.

In the first version, the updates are performed by following the order of the entire lines of the grid. In the second version, the horizontal strips are divided in blocks along their width and their updates are performed block by block. The height of the blocks in a given strip is the same as the height of the strip, but their width may be smaller as it is directly deduced from the cache size and the line width (lw), as illustrated in Figure 1. In fact, the optimal block width (obw) is deduced from the cache size. Then, the number of blocks (nbb) per strip is computed. Finally, the actual block width (abw) is computed in order to obtain blocks of the same width in the horizontal strip.

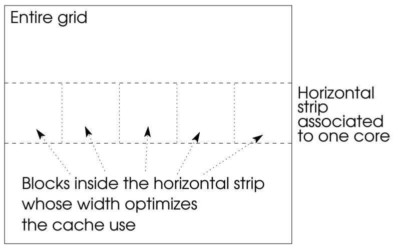
Entire grid
Horizontal strip associated to one core
Blocks inside the horizontal strip whose width optimizes the cache use

Figure 1.: Blocks in horizontal strips to optimize the cache use

$$nbb = \left\lceil \frac{lw}{obw} \right\rceil, \qquad abw = \frac{lw}{nbb} \tag{2}$$

3.2. *Many-core MIC with offloaded OpenMP*

In order to use the MIC Xeon-Phi, Intel proposes an extension of the OpenMP library in its C/C++ compiler. It mainly consists in additional directives that allows the programmer to control the MIC directly from the CPU. It must be noticed that any classical OpenMP program can be run directly on a MIC. However, in this context, the MIC acts as an autonomous multi-core machine but it cannot cooperate (via OpenMP) with the central CPU cores. So, in the perspective of making the MIC cooperate with other devices (CPU, GPU,...), it is required to use the MIC as a co-processor of the central CPU (see [5] for an introduction to *offload* programming paradigm). One great advantage of the MIC, compared to other devices such as GPU, is that the same OpenMP code that runs on the CPU can be executed on the MIC without modification. Hence, Listing 1 can be directly executed on a MIC. However, the MIC has its own memory and can only process data in its memory. This implies the need of explicit data transfers between the central memory of the node and the memory on the MIC board.

The execution and data transfers can be expressed with the same directive, called `offload`, as depicted in Listing 2.

Listing 2: Offloading of the Jacobi relaxation on a MIC with synchronous data transfers

```
#pragma offload target(mic:0) inout(tabM:length(nLig*nCol) align(64))
{ // Computes nbIters iterations of Jacobi over array tabM
  // with nbTMic cores on the MIC
  jacobi(tabM, nLig, nCol, nbIters, nbTMic);
}
```

In this example, `target` gives the identifier of the MIC device to use, and the `inout` parameter specifies that the array `tabM` (whose size must be given in number of elements) is an `in`put as well as an `out`put of the offload. That means that before the start of the computation on the MIC, the array is copied from central RAM to the MIC RAM. And once the computation is over, the array is copied back from the MIC RAM to the central RAM (at the same location). The scalar variables passed as parameters of the `jacobi` function are implicitly copied from the central RAM to the MIC RAM. Finally, the `align` parameter is optional and forces the memory allocations for the data on the MIC to be aligned at boundaries greater or equal to the specified number of bytes. Such memory alignments improve the performance of data transfers.

It is worth noticing that the offload presented in Listing 2 is blocking. So, the core CPU that executes this offload will wait for the end of the execution of `jacobi` on the MIC and for the completion of the output data transfer from the MIC memory to the central one, before resuming its execution. When the MIC is used alone, without cooperating with the CPU, this synchronous scheme is pertinent. Nevertheless, it prevents any computation by the CPU while the MIC is running. We will see in Section 4 how to perform asynchronous (non-blocking) offloads, in order to allow the CPU to work during the MIC execution. Also, we will point out the need to replace blocking data transfers by asynchronous ones, in order to overlap communication with computation.

3.3. Many-core GPU with CUDA

We designed a single CUDA kernel to process the GPU part of the Jacobi relaxation. It is called two times per iteration: to quickly and early compute the boundary of the GPU part of the Jacobi grid, and to compute the (large) rest of this grid part. We optimized our algorithm and code to make fast *coalescent* memory accesses, to use the *shared memory* of each vectorial multiprocessor of the GPU, and to *limit the divergence* of the thread of a same block (when not executing exactly the same instructions). See [6] for efficient CUDA programming rules.

Each thread of this kernel updates one point of the Jacobi Grid during one cycle, and threads are grouped in 2-dimensional blocks of a 2-dimensional grid. This kernel has been optimized using the *shared memory* of each multiprocessor of the GPU, allowing each thread to read only one data from the GPU global memory, to share this data with others threads of its block, and efficiently access the 5 input data it requires to update its Jacobi grid point. Global memory read and write are achieved in a coalescent way. Considering a block of size $BSY \times BSX$, all the threads (in the range $[0; BSY-1] \times [0; BSX-1]$) load data from the global memory into the shared memory, and $(BSY-2) \times (BSX-2)$ threads in the range $[0; BSY-3] \times [0; BSX-3]$ achieve computations, limiting the divergence of the threads inside a block.

Listing 3: Optimized CUDA kernel

```
__global__ void update(double *gpuPrec, double *gpuCrt, int gpuLigs,↩
      int cols)
{
  int idx, lig, col;
  __shared__ double buf[BLOCKSIZEY][BLOCKSIZEX];

  // Coordinates of the Jacobi grid to load in shared memory
  col = blockIdx.x * (BLOCKSIZEX - 2) + threadIdx.x;
  lig = blockIdx.y * (BLOCKSIZEY - 2) + threadIdx.y;
  // If valid coordinates: load data and compute
  if(col < cols + 2 && lig < gpuLigs + 2){
    idx = lig * (cols + 2) + col;
    buf[threadIdx.y][threadIdx.x] = gpuPrec[idx];
    __syncthreads();
    lig++; col++; // shift coordinates to point out element to compute
    // if new coordinates are valid: achieve computation
    if(col <= cols && lig <= gpuLigs && threadIdx.x < BLOCKSIZEX-2 ↩
        && threadIdx.y < BLOCKSIZEY-2){
      idx = lig * (cols + 2) + col;
      gpuCrt[idx] = 0.2 * (buf[threadIdx.y][threadIdx.x+1] +
                    buf[threadIdx.y+1][threadIdx.x] +
                    buf[threadIdx.y+1][threadIdx.x+1] +
                    buf[threadIdx.y+1][threadIdx.x+2] +
                    buf[threadIdx.y+2][threadIdx.x+1]);
    }
  }
}
```

Moreover, CPU memory arrays involved in the CPU-GPU data transfers have been locked in memory, using `cudaHostAlloc(...)` routine, in order to support asynchronous and faster data transfers. Finally, we used some CUDA *streams* to efficiently manage and run concurrent data transfers and kernel computations, so that we obtain a maximal overlapping.

4. Multiple architectures and devices solution

4.1. General asynchronous scheme and data distribution

In our context, the GPU is used as a scientific co-processor, and we use the MIC in *offload* mode. So, our hybrid CPU+MIC+GPU solution still uses the CPU to run the `main` function, to launch all computation steps on the GPU, on the MIC and on its own cores, and to launch the data transfers between the CPU and the accelerators. The CPU memory hosts the entire current (`crt`) and previous (`prev`) Jacobi grids, but the top part is transferred on the GPU and the bottom part on the MIC (see Figure 2). We name CPU boundaries the first and last lines computed by the CPU, GPU boundary the last line computed by the GPU, and MIC boundary the first line computed by the MIC. We name *corpus* the other lines computed by a computing device. So, each *computing device* (CPU, MIC and GPU) stores its parts of the Jacobi grids and the adjacent boundary(ies) of other device(s). In order to save memory and optimize the transfers, our parallel algorithm is designed to allow direct transfers of the frontiers in their right place in the local arrays on the CPU and GPU. So, no intermediate array is required for the frontiers between CPU and GPU. A similar scheme would be also possible for the MIC device. However, due to a particularly slow memory allocation of offloaded data and the impossibility to transfer data from the CPU to an array that has been locally allocated on the MIC, the use of an intermediate buffer for the CPU/MIC frontiers has been necessary on the MIC side. Save for this difference, the CPU algorithm uses as symmetric

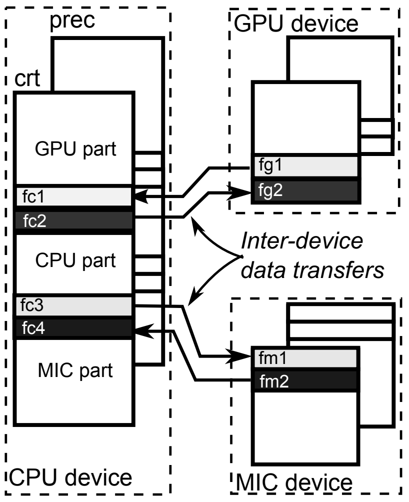

Figure 2.: Data management scheme on the three devices (CPU, MIC and GPU)

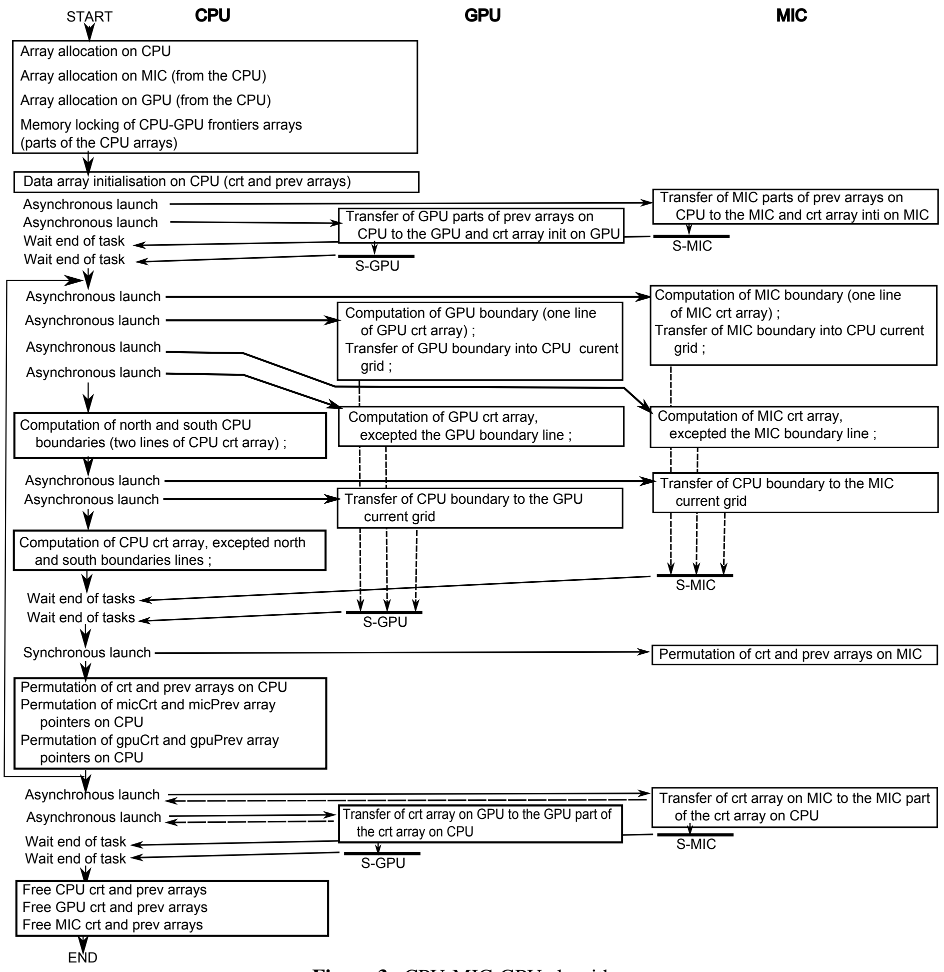
START
CPU
GPU
MIC
Array allocation on CPU
Array allocation on MIC (from the CPU)
Array allocation on GPU (from the CPU)
Memory locking of CPU-GPU frontiers arrays (parts of the CPU arrays)
Data array initialisation on CPU (crt and prev arrays)
Asynchronous launch
Asynchronous launch
Wait end of task
Wait end of task
Transfer of MIC parts of prev arrays on CPU to the MIC and crt array inti on MIC
Transfer of GPU parts of prev arrays on CPU to the GPU and crt array init on GPU
S-MIC
S-GPU
Asynchronous launch
Asynchronous launch
Asynchronous launch
Asynchronous launch
Computation of MIC boundary (one line of MIC crt array) ; Transfer of MIC boundary into CPU current grid ;
Computation of GPU boundary (one line of GPU crt array) ; Transfer of GPU boundary into CPU curent grid ;
Computation of north and south CPU boundaries (two lines of CPU crt array) ;
Computation of GPU crt array, excepted the GPU boundary line ;
Computation of MIC crt array, excepted the MIC boundary line ;
Asynchronous launch
Asynchronous launch
Transfer of CPU boundary to the MIC current grid
Transfer of CPU boundary to the GPU current grid
Computation of CPU crt array, excepted north and south boundaries lines ;
Wait end of tasks
Wait end of tasks
S-MIC
S-GPU
Synchronous launch
Permutation of crt and prev arrays on MIC
Permutation of crt and prev arrays on CPU
Permutation of micCrt and micPrev array pointers on CPU
Permutation of gpuCrt and gpuPrev array pointers on CPU
Asynchronous launch
Asynchronous launch
Transfer of crt array on MIC to the MIC part of the crt array on CPU
Transfer of crt array on GPU to the GPU part of the crt array on CPU
Wait end of task
Wait end of task
S-MIC
S-GPU
Free CPU crt and prev arrays
Free GPU crt and prev arrays
Free MIC crt and prev arrays
END

Figure 3. CPU-MIC-GPU algorithm

as possible data structures and interactions for both accelerators. Figure 3 introduces our multi-device algorithm, based on the following principles:

- Before to enter a new computation step, a processor has its previous Jacobi grid entirely updated, including the boundary of the adjacent processor. So it can compute all its part of its current grid.
- A processor sends its newly updated boundary to the adjacent processor while it receives the updated boundary of this adjacent processor.
- Boundary(ies) computation and transfer of a processor are sequentially linked, but achieved in parallel of its corpus computation. The objective is to overlap as much as possible the data transfers with large computations, as well as to avoid that a processor is underused by processing only its boundary(ies).
- The CPU launch asynchronous computations on accelerators and asynchronous data transfers from and to the accelerators. So, the two accelerators and the CPU can compute in parallel, and the different data transfers can exploit the two PCI express buses in parallel.

Obviously, two synchronization points, S-MIC and S-GPU, are mandatory to ensure that data transfers and computations are finished respectively on MIC and GPU, before to switch the arrays (current and previous grids) and to enter the next iteration.

A slight asymmetry appears between the MIC and GPU concerning the arrays switching management (pointers switching). In fact, pointers on GPU arrays are stored in the CPU memory and sent to the GPU computing kernel as parameters when launching the kernel. So, these pointers can be switched directly in the CPU memory by the CPU process. On the contrary, array pointers on MIC are stored in the MIC memory and managed by the MIC. So, the CPU needs to launch a short task on the MIC to make it switch its local array pointers.

Finally, we obtain an efficient and rather generic and symmetric parallel scheme that make cooperate CPU, MIC and GPU devices to solve the same problem.

4.2. Implementation details

To achieve asynchronous transfers between CPU and GPU, three CUDA streams are used together with two CUDA registrations of the memory banks concerned by the transfers to lock them and avoid their swapping. We recall that CUDA streams run concurrently but in each stream, data transfers and kernels are serialized. One stream is used to compute and send the *FG1* line (cf. Fig.2) to the CPU (*FC1*), another one is used to receive the *FG2* line from the CPU (*FC2)*, and the last one is used to control the asynchronous computation of the GPU part. The two registrations concern the two frontier lines (*FG1* and *FG2*). The `cudaMemcpyAsync` and `cudaStreamSynchronize` functions are used to perform the asynchronous transfers and to ensure their completion before to proceed to the following computations.

Concerning the asynchronous data transfers between CPU and MIC, the `signal` clause is used in the `offload` directive computing and sending (with a `out` clause) the *FM2* line to the CPU (*FC4*). It is also used in the `offload_transfer` directive related to the reception of *FM1* from the CPU (*FC3*). There is also a signaled `offload` to asynchronously perform the computation of the MIC part. Then, the `offload_wait` directive is used to ensure the transfer completions before performing the following computations.

5. Experiments

5.1. Individual performances of the devices

Table 1 shows the absolute and relative performances of the three devices (computing units) on each testbed machine, during 5000 iterations on a grid of 20000×10000 points. The results are averages of 5 executions. On both machines, the CPU part (cores on the motherboard) is the less powerful, the MIC device is medium, and the GPU is the most powerful for this problem. The CPU and MIC units in the Loria machine are faster than the ones in the CS machine, whereas the CS machine GPU is faster than in the Loria one. This implies that the two machines have different behaviors when running the multi-device algorithm. This is detailed in the following part.

20000 × 10000 pts, 5000 iterations				
Testbed	Measure	CPU	MIC	GPU
CentraleSupelec machine	Computation speed	1.19E+009	3.50E+009	9.84E+009
	Global speed	1.19E+009	3.48E+009	9.78E+009
	Standard deviation (%)	0.51	6.18	0.04
	Speedup	1.0	2.95	8.31
Loria machine	Computation speed	1.60E+009	4.83E+009	7.93E+009
	Global speed	1.60E+009	4.69E+009	7.87E+009
	Standard deviation (%)	5.14	4.72	0.09
	Speedup	1.0	3.02	4.96
Computation speed = number of updated points / second				
Global speed includes computations, allocations and transfers				

Table 1. Absolute and relative performance of the three devices (averages of 5 executions)

20000 × 10000 pts, 5000 iterations						
Absolute speeds (in updated points / s)						
	CS machine C = 13500, M = 15000			Loria machine C = 12500, M = 14000		
C / M cutting lines	M - 500	M	M + 500	M - 500	M	M + 500
C - 500	1.15E+010	1.11E+010	8.79E+009	8.99E+009	1.01E+010	1.02E+010
C	1.12E+010	**1.22E+010**	1.12E+010	9.76E+009	**1.06E+010**	1.03E+010
C + 500	1.12E+010	1.12E+010	1.13E+010	9.64E+009	1.00E+010	9.98E+009
Speedups from GPU alone						
C - 500	1.17	1.13	0.89	1.13	1.27	1.28
C	1.14	**1.24**	1.14	1.23	**1.34**	1.29
C + 500	1.14	1.14	1.15	1.21	1.26	1.26

Table 2. Performance (speed and speedup) of heterogeneous computing with CPU, MIC and GPU

5.2. Performance of heterogeneous computing on the three devices

Table 2 shows the speeds and speedups of our heterogeneous algorithm for the same problem parameters (iterations and grid size), but with different cutting lines. As shown above, the two machines having different devices speeds, their optimal cutting lines are not the same. So, for each machine, nine measures are performed around the theoretical optimal cutting lines (based on single-device performances), using variations of ±500 lines.

First of all, we observe that our heterogeneous algorithm obtains gains with both machines, according to the fastest device alone (the GPU). However, those gains are different according to the relative powers of the devices. In the CS machine, the GPU is much more powerful than the two other devices, implying a limited potential gain of the heterogeneous version. It is worth noticing that the ideal speedup, estimated without any communication/allocation overhead is 1.43. So, with a speedup of 1.24, our algorithm obtains 87% of the ideal case. With the Loria machine, the powers of the devices are a bit closer, implying larger potential gains. However, in this case, although the ideal speedup is 1.74, the obtained one is 1.34 and the efficiency is only 77%. Moreover, it must be noticed that the cutting lines reported in Table 2 for this machine are a bit larger than the ones indicated by theory (11500 and 13500). This shifting may come from a slight overestimation of the CPU and MIC, itself due to the higher variations of performance on this machine (see *Standard deviation* in Table 1). Another possible reason of the smaller efficiency may come from the older compiler available on that machine.

Finally, those experiments confirm that our algorithmic scheme can achieve significant gains by performing a quite efficient cooperation of different kinds of computing devices inside a same machine.

6. Conclusion

A parallel scheme has been described that allows the cooperation of different computing devices inside a single hybrid machine. The major difficulty in exploiting such devices together comes from the data transfers between the central memory of the system and the local memory on each device. To obtain good efficiency, it is required to make extensive use of asynchronism between the devices as well as overlapping computations with communications by asynchronous data transfers.

Our experiments have validated our multiple-devices parallel scheme: results were qualitatively identical using one, two or three devices. Two series of asynchronous data transfers (CPU $\leftrightarrow$ GPU and CPU $\leftrightarrow$ MIC) have been implemented with different mechanisms and the most efficient combination has been presented.

The results show the possibility to achieve significant gains with quite good efficiencies (from 77 to 87%) according to the ideal gains without the management cost of heterogeneous devices. Our algorithm shows how to achieve them. Moreover, as it can be used inside nodes of a cluster, it represents an additional step towards a better exploitation of large parallel systems (clusters with heterogeneous accelerators).

Among many possibilities, interesting extensions of this work consist in adapting that parallel scheme to more complex scientific applications, extending it to the coupling of different solvers running on different devices, adapting it to clusters, and considering large problems that do not fit in the memory of one device alone.

References

[1] S. Vialle and S. Contassot-Vivier. *Patterns for parallel programming on GPUs*, chapter Optimization methodology for Parallel Programming of Homogeneous or Hybrid Clusters. Saxe-Coburg Publications, 2014. ISBN: 978-1-874672-57-9.

[2] S. Contassot-Vivier, S. Vialle, and J. Gustedt. *Designing Scientific Applications on GPUs*, chapter Development Methodologies for GPU and Cluster of GPUs. Chapman & Hall/CRC Numerical Analysis and Scientific Computing series. Chapman & Hall/CRC, 2013. ISBN 978-1-466571-64-8.

[3] J. Fang, A. L. Varbanescu, B. Imbernon, J. M. Cecilia, and H. Perez-Sanchez. Parallel computation of non-bonded interactions in drug discovery: Nvidia GPUs vs. Intel Xeon Phi. In *2nd International Work-Conference on Bioinformatics and Biomedical Engineering (IWBBIO 2014)*, Granada, Spain, 2014.

[4] B. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL.* Morgan Kaufmann, 2nd edition, 2012. ISBN 9780124058941.

[5] J. Jeffers and J. Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Elsevier Waltham (Mass.), 2013. ISBN 978-0-12-410414-3.

[6] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010. ISBN-10 0131387685, ISBN-13 9780131387683.