Generic algorithmic scheme for 2D stencil applications on hybrid machines

Stephane Vialle[2], Sylvain Contassot-Vivier[1], and Patrick Mercier[2]

[1] Loria - UMR 7503, Université de Lorraine, Nancy, France
[2] UMI 2958, Georgia Tech - CNRS, CentraleSupelec, University Paris-Saclay, Metz, France

Abstract. Hardware accelerators are classic scientific coprocessors in HPC machines. However, the number of CPU cores on the mother board is increasing and constitutes a non negligible part of the total computing power of the machine. So, running an application both on an accelerator (like a GPU or a Xeon-Phi device) and on the CPU cores can provide the highest performance. Moreover, it is now possible to include different accelerators in a machine, in order to support and to speedup a larger set of applications. Then, running an application part on the most suitable device allows to reach high performance, but using all unused devices in the machine should permit to improve even more the performance of that part. However, the overlapping of computations with inter-device data transfers is mandatory to limit the overhead of this approach, leading to complex asynchronous algorithms and multi-paradigm optimized codes. This article introduces our research and experiments on cooperation between several CPU and both a GPU and a Xeon-Phi accelerators, all included in a same machine.

1 Introduction and objectives

Hardware accelerators have become classical scientific coprocessors but they still need a CPU to control the entire machine. Also, current standard CPU have a significant number of cores and computing power. Using both CPU and accelerator cores seems an interesting way to achieve the maximal computing speed on a computing node. But using both the CPU and accelerators leads to frequently transfer intermediate results between all those devices. So, asynchronous data transfers, overlapped with computations, are required to obtain significant speedups compared to mono-device executions.

In this study, we consider hybrid computing nodes: each node hosting one or more CPU, one GPU and one Xeon-Phi, allowing one to choose the most adapted architecture for each application, or application module. However, once the most adapted device has been identified for a computation in an algorithm, it may be interesting to cooperatively use any other kind of computing device available in the machine to concurrently process that computation, and then to increase the overall performance. In order to ease the use of our multi-devices and multi-architectures machines, we developed a generic parallel algorithmic scheme achieving an overlapping of data transfers with computations between two or three devices (simultaneously), for 2D stencil applications. This generic scheme has been implemented and optimized for GPU and Xeon-Phi devices, considering the specific features of their programming models and API.

Our approach has been validated in [3] with a Jacobi relaxation application. In this paper, we extend it to a more complex stencil application, more representative of real scientific applications: a shortest path computation. As in [3], that application has been tested on two different hybrid machines. We were able to experiment our different kernels (on CPU, GPU and Xeon-Phi), using one, two or three devices simultaneously, and to identify the most suited combination for each application.

2 Application examples

Our parallel scheme (see Sec.4) aims at running *2D stencil* applications, on hybrid machines. Two different applications have been used to test and validate our approach: a very classical *2D Jacobi relaxation* algorithm, and a *shortest path calculation on 2D+ ground* (taking the elevations into account). They both work on 2D regular grids of points. The reader is invited to see [3] for a detailed description of our *Jacobi* algorithm. Our *shortest path* algorithm computes *minimal paths costs* from a given position in the grid to all other positions. It is slightly different from the Jacobi algorithm as it is an adaptation of the Dijkstra's algorithm [5]. The path cost of each point $P_{i,j}$ is updated according to the previous costs of its eight neighbors in the 2D grid (denoted $N_8(P_{i,j})$). Indeed, for each neighbor, the 3D euclidian distance to the current point is computed and added to the current minimal cost of the neighbor. Then, the new cost of the current point is chosen as the minimal cost among the eight computations:

$$C^{n+1}(i,j) = \min_{P_{a,b} \in N_8(P_{i,j})} (C^n(a,b) + d(P_{i,j}, P_{a,b}))$$

Considering h, the side length of a grid cell (between two successive points in the grid), the distance between $P_{i,j}$ and $P_{a,b}$ is:

$$d(P_{i,j}, P_{a,b}) = \sqrt{((i-a).h)^2 + ((j-b).h)^2 + (z(i,j) - z(a,b))^2}$$

In fact, $N_8(P_{i,j})$, the neighboring of point $P_{i,j}$ in the 2D grid (see Fig. 1), contains less than eight neighbors when $P_{i,j}$ is located on a grid edge. This algorithm requires a 2D array of costs and a 2D array of elevations (z). Initially, a target point $T_{\alpha,\beta}$ is chosen and its cost is set to 0 while all others costs are set to $+\infty$. Then the algorithm starts to iterate until no cost changes during an entire iteration (convergence is reached). Finally, we get a 2D array of minimal costs (distances) and paths from every point $P_{i,j}$ to the target point $T_{\alpha,\beta}$.

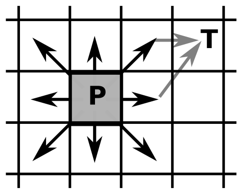


Fig. 1. Shortest path point neighboring

3 Related work

Today, scientific computing on GPU accelerators is common, while using Xeon-Phi accelerators starts to be deeply investigated and some comparisons have been achieved. In [6], authors point out the need to optimize data storage and data accesses in different ways on GPU and Xeon Phi. In [2], authors optimize data storage for stencil applications on different CPU, GPU and APU (processors with both CPU and GPU cores). Several studies like [8, 11] address the deep optimization of parallel stencil applications or other lattice based applications [12], for different types of devices. However, their programs use only one device at a time. Some authors use a generic programming model and tool to develop on different architectures, like OpenCL [7, 10]. Nevetheless, such tool does not hide to the programmer the devices hardware specificities to be taken into account to attain optimal performance. Thus, an important algorithmic effort is still required to design codes efficiently running on different architectures.

Another approach consists in using the CPU memory to store data too large for the accelerator memory, and to efficiently send sub-parts of the problem to the accelerator [9]. Here again, there is no simultaneous use of the CPU and the accelerators.

Finally, some works propose general frameworks to use different devices by performing dynamic scheduling of tasks graphs or data flows [4]. However, the generality of such frameworks induces additional costs compared to application specific schemes.

In [3], we introduced our first algorithmic scheme of heterogeneous computing, successfully running a Jacobi relaxation on CPU, GPU and Xeon-Phi. In the current study, we generalize our approach to a more realistic scientific problem involving more complex and irregular computation.

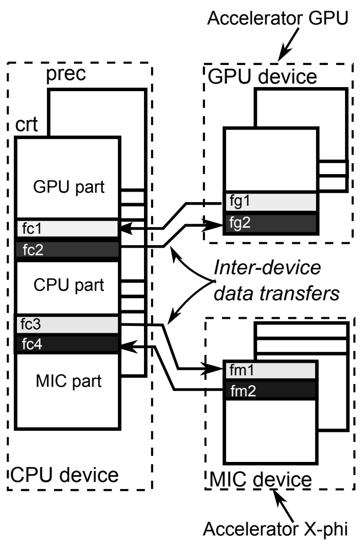
4 Algorithmic scheme

In our context, the GPU and the Xeon-Phi are used as scientific co-processors in *offload* mode. The Xeon-Phi could have been used via MPI but we chose the offload mode to get similar programming paradigms between the GPU and the Phi. So, our hybrid solution uses the CPU to launch and control the computation steps on the GPU, the Xeon-Phi and its own cores, and to manage data transfers between CPU and devices.

4.1 Global strategy of the parallel scheme

Our global data distribution strategy is to partition the 2D grid in three horizontal strips (potentially of different heights), so that the first one is processed on the GPU, the second one, on the CPU, and the third one, on the Phi. Placing the CPU between the GPU and Phi in the logical organization is a strategic choice as direct data transfers between GPU and Phi are not currently possible. 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 Xeon-Phi. We name *corpus* the other lines computed by a computing device. According to data dependencies in the application, each computing device must store, in addition to its strip of the grid, the adjacent boundary(ies) of its neighboring device(s). So, the top part of the grid is transferred to the GPU and the bottom part to the Xeon-Phi (see Fig. 2 left). Although the CPU memory may host only its associated part of the grid (and the required boundaries), for the sake of simplicity in the overall management of the system, we have chosen to store the entire current (`crt`) and previous (`prev`) relaxation grids. However, this could be easily optimized if the CPU memory were too small to store the entire grids.

At each iteration during the computation loop, the GPU boundary may be transferred to the CPU while the adjacent CPU boundary may be transferred to the GPU. Symmetrically, the CPU/Xeon-Phi boundaries may be transferred too. Indeed, as a frontier may remain unchanged during one iteration, our algorithm transfers a frontier to/from an accelerator only if it has undergone at least one modification. Also, in order to optimize the transfers, our algorithm is designed to allow direct transfers of the frontiers in their place in the local arrays on the destination device. So, no intermediate array is required to store the received frontiers coming from another device, and the CPU algorithm uses symmetric data structures and interactions for both accelerators.



```
**Input**: one initialized array on the CPU
                    ↓
Urgent allocations and initialization on CPU
2x Async {Data allocation from CPU on Accelerator
              Data transfer from CPU to Accelerator
              Complementary initialization on Accelerator}
Complementary allocations and initialization on CPU
2x Sync. barrier: waiting for end of all Accelerator ops
                    ↓
**Computation loop**:
   2x Async {Frontier comput. on Accelerator}
   2x Async {Corpus comput. on Accelerator}
   2x {Frontier comput. on CPU}
   2x {If CPU frontier modified then
              Async frontier transfer from CPU to Accelerator}
   2x {Sync. barrier on the end of Accel. frontier comput.
         If Accel. frontier modified, then
            Async frontier transfer from Accelerator to CPU}
   Corpus comput. on CPU
   4x {If Async. frontier transfer launched, then
            Sync. barrier on the end of Async. transfer}
                    ↓
Async {Result transfer from Accelerator 1 to CPU}
Result transfer from Accelerator 2 to CPU
Sync. barrier: waiting for end of transfer from Accelerator 1
2x Free data structures on Accelerator
Free data structures on CPU
                    ↓
**Output**: one result array on the CPU
```

Fig. 2. Data structures (left) and asynchronous and overlapping algorithm (right) to obtain efficient simultaneous computations on the three devices (CPU, GPU and Xeon-phi)

4.2 Multi-device algorithm

Figure 2 (right) details the three main parts of our generic multi-device algorithm (see [3] for a simplified version, with straightforward array allocation on accelerators and systematic frontier transfers):

Initialization step: Memory allocation on accelerators and initial data transfers from CPU to accelerators can be long, so the initialization step is parallel. Firstly, input data are prepared on the CPU. Then, the other devices are initialized and they simultaneously receive their respective part of input data. Complementary allocations and initialization are performed concurrently between all devices. At last, synchronization barriers are used to ensure that each device is ready to enter the computation loop.

Computation loop: We focused on maximizing the overlap of boundary(ies) computations, boundary(ies) transfers, and corpus computations on the different devices. Boundary transfers are performed simultaneously between devices that compute adjacent strips of the grid when boundaries are modified. On each device, boundaries computation and potential transfer are performed concurrently with the corpus computation in order to maximize the overlap of boundaries transfers with corpus computations. All the devices work concurrently thanks to asynchronous operations. Data transfers from/to the accelerators are fully efficient when two PCI express buses are present. Each iteration is ended by synchronization barriers to ensure that every device has all its newly updated data.

Final step: Results retrieving from devices to the CPU can be long and have been also overlapped. The results from one accelerator are transferred asynchronously during the synchronous transfer from the other one. Thus, transfers overlap and only one synchronization barrier is required to ensure the complete reception on the CPU. Finally, the CPU cleans up the devices and its own memory and releases them. As this last operation never appeared to be time consuming, it is done sequentially.

5 Computing kernels

The design of optimized multi-core or many-core kernels is out of scope of this article as we focus on the efficient interactions between cooperating kernels. However, as we developed the computing kernels for the three types of devices by adapting the Jacobi kernels to the shortests path problem, the reader should see [3] for further details.

On GPU, both for *Jacobi* and *Shortest Path* applications, we designed CUDA kernels using the fast (but small) *shared memory* of each stream-processor, in order to load and access sub-parts of the data arrays. This is a kind of cache memory explicitly managed by the developer, which is a classical optimization on NVIDIA GPU. For the *Shortest Path* program, we designed a first solution that pre-compute distances between all neighbors in the grid, in order to avoid recomputing $d(P_{i,j}, P_{a,b})$ (see Sec. 2). However, this version showed to be penalized by the induced larger data transfers between GPU memories. The adopted solution, less memory consuming, consists in taking advantage of the regular structure of the grid cells by pre-computing only X and Y axis contributions to the distances between neighbors, which represents at most 8 values instead of four times the number of grid points. This implies an important reduction of data copies into the *shared memory* at each iteration. In compensation, distances have to be recomputed at each iteration, but only in a partial form (thanks to X and Y pre-computed contributions) that does not increase the computational load so much. This version has been used in all the experiments presented in Sec. 8.

Finally, we deployed 2D grids of 2D blocks of CUDA threads, and experimented different sizes of 2D blocks. The optimal solution depended on the GPU used: blocks of 16×16 ($Y \times X$) threads run faster on the *GeForce GTX Titan Black*, while blocks of 16×32 threads run faster on the *Tesla K40m* (both GPU have a Kepler architecture).

On CPU and Xeon-Phi, we designed basic and cache-blocking algorithms. When blocked in cache, the grid update was processed per small sub-grids filling only the amount of L2 cache available per thread. Moreover, we attempted to guide the compiler to vectorize computational loops by using the AVX units (but we did not implement *intrinsics* routine calls). On the basic calculations of the *Jacobi* algorithm, our cache blocking mechanism has been efficient. On the opposite, on the *Shortest Path* algorithm our cache blocking implementation had no effect but vectorization has been efficient. Using `#pragma ivdep` and `#pragma simd` directives and AVX compiler options, and achieving minor code modifications we succeeded to significantly improve performance on Xeon CPU and on Xeon-Phi accelerator (3120 and 5100 series).

6 Optimized solution for Xeon-Phi and GPU accelerators

We present below the optimization of the initialization step and the computation loop.

6.1 Initial datastructure allocations and settings

Figure 3 introduces the actually implemented data-structures, not so different from the generic ones (see Fig. 2 right). Listing 1.1 illustrates how we implemented data-structures allocations and initialization by mixing GPU CUDA, CPU OpenMP and Xeon-Phi Offload semantics and syntax. This code implements and respects the initial step of our generic algorithm (see Sec. 4).

Main changes in the data-structures concern the Xeon-Phi (see Fig. 2 (right) and 3). In fact, we could observe during our developments that allocations of large arrays on the

Xeon-Phi are very long when achieved from the CPU through a `#pragma offload in(...)` directive. However, this allocation mechanism is required to be able to transfer data between the CPU and the Xeon-Phi. So, we allocate the `micPrev` array and transfer CPU data with a `#pragma offload` directive (lines 4-5), but the `_micCrt` array is allocated as a pure Xeon-Phi variable (line 12), which is a fast mechanism. Then, it is initialized with a Xeon-Phi internal `memcpy` call (line 13). Obviously, the final results will have to be transferred from the Xeon-Phi to the CPU only through the `micPrev` array. This strategy significantly reduces the initialization step, but leads to allocate two additional small buffers to transfer the CPU and Xeon-Phi frontiers during each computation step, as shown in Fig. 3 (`fm1` and `fm2`). This allocation of a transferable 2-lines array is done at line 6. All this sequence of operations is run asynchronously due to the `signal` clause in line 9.

After launching the allocation and initialization sequence on the Xeon-Phi, the CPU immediately enters the GPU sequence at line 19. It allocates several *streams* to manage concurrent operations on the GPU, and several arrays on the GPU (lines 21 and 22). According to CUDA paradigm, the CPU allocates memory on the GPU calling a `cudaMalloc` function. At lines 24-26 the CPU runs another CUDA library function to lock in its memory the arrays that will be transferred to/from the GPU, in order to speedup the transfers and to do them asynchronously. All these memory allocations and locking operations are achieved synchronously but are fast. Then, the CPU launches long asynchronous data transfers to fill the allocated GPU arrays (lines 28-30), and an asynchronous GPU-to-GPU memory copy at lines 32-33. All these long operations are run asynchronously on stream 0 so that the CPU can concurrently perform its own initialization at lines 35-38 (running several threads to speedup a large memory copy).

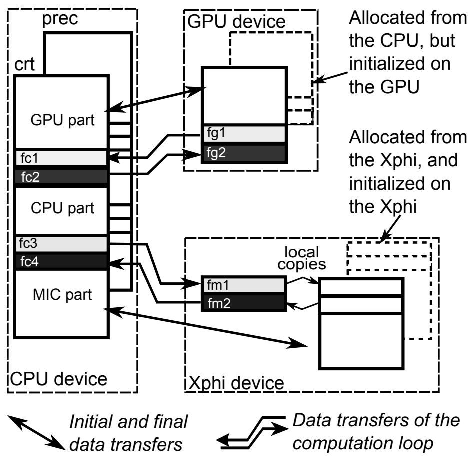

prec
crt
GPU part
fc1
fc2
CPU part
fc3
fc4
MIC part
CPU device
GPU device
fg1
fg2
Allocated from the CPU, but initialized on the GPU
Allocated from the Xphi, and initialized on the Xphi
local copies
fm1
fm2
Xphi device
Initial and final data transfers
Data transfers of the computation loop

Fig. 3. Optimized data structures implemented on the three devices (CPU, Xphi and GPU)

Finally, we implemented the two synchronization barriers of our algorithm (Fig. 2 right). At line 40, the CPU waits for the end of all asynchronous GPU operations on stream 0, and at line 41, it waits for the end of all asynchronous Xeon-Phi operations. This implementation is very close to the initial step of our generic algorithm, save for some fast GPU memory allocation and CPU memory locking that remain synchronous.

Listing 1.1. Initial step implementation

```
1 // CPU allocation of data array
2 posix_memalign(&crt, ALIGN, TotCol * TotLin * sizeof(double));
3 // ASYNCHRONOUS XEON-PHI ALLOCATIONS AND INITIALIZATION
4 #pragma offload target(mic:0)                                \
5   in(micPrev:length(micTotLin*TotCol) free_if(0) align(ALIGN)) \
6   in(micCrt:length(2*TotCol) free_if(0) align(ALIGN))          \
7   ...                                                          \
  nocopy(micRes, _micCrt, micTopFront, micBotFront)            \
  signal(&micPrev)
{
 // Alloc and init of _micCrt array on Xeon-Phi (matching micPrev array)
 posix_memalign(&_micCrt, ALIGN, micTotLin*TotCol*sizeof(double));
 memcpy(_micCrt, micPrev, micTotLin*TotCol*sizeof(double));
 // Init pointers on data arrays on Xeon-Phi
 micRes = micPrev + TotCol;
 ...
}
// Synchronous GPU Allocations
for (i=0; i<NBSTREAMS; ++i)   // 3 streams
   cudaStreamCreate(&(streamTab[i]));
cudaMalloc(&gpuPrev, gpuTotLin*TotCol*sizeof(double));
...
// Synchronous CPU memory lock (to speedup CPU-GPU data transfers)
cudaHostRegister(prev, gpuTotLin*TotCol*sizeof(double), // transferred
                 cudaHostRegisterPortable);             // to gpuPrev
...
// ASYNCHRONOUS GPU INIT: data transfer from CPU to GPU
cudaMemcpyAsync(GpuPrev, prev, gpuTotLin*TotCol*sizeof(double),
                cudaMemcpyHostToDevice, streamTab[0]);
...
// ASYNCHRONOUS GPU INIT: data copy from GPU to GPU
cudaMemcpyAsync(GpuCrt, GpuPrev, gpuTotLin*TotCol*sizeof(double),
                cudaMemcpyDeviceToDevice, streamTab[0]);
// Complementary CPU init. (parallelized with OpenMP)
#pragma omp parallel num_threads(nbTHet) {
  size_t idxFirstValTh = ...; size_t nbValTh = ...;
  memcpy(crt+idxFirstValTh, prev+idxFirstValTh, nbValTh*sizeof(double));
}
// SYNCHRONIZATION BARRIERS on the end of the GPU and Xeon-Phi async. init.
cudaStreamSynchronize(streamTab[0]);               // GPU
#pragma offload_wait target(mic:0) wait(&micPrev) // Xeon-Phi
```

6.2 Asynchronous and overlapped computation loop

Code snippet in Listing 1.2 summarizes our implementation of the computation loop of the generic algorithm (Fig. 2 right). Line 1 shows that the global loop is stopped as soon as no modification occurs during one iteration. Line 4 is an asynchronous CUDA kernel launch, running frontier computation on the GPU. Indeed, a CUDA kernel launch has a specific CUDA syntax, and requires to be implemented in a `.cu` source file, and to be compiled with the Nvidia CUDA compiler (`nvcc`), that cannot compile Intel offload or OpenMP directives. So, CUDA kernels launching operations have to be encapsulated into functions located in a `.cu` file. Line 5 transfers a *modification flag* of the GPU frontier onto the CPU. This asynchronous transfer uses the same CUDA *top frontier* stream (`TF`) as the previous kernel execution, so these two asynchronous operations are sequenced: frontier computation is achieved and the modification flag set before it is transferred to the CPU. Lines 7 and 8 reproduce this kernel launch and modification flag transfer, focusing on the rest of the problem processed on the GPU, using the CUDA *corpus* stream.

Lines 11-16 perform a similar frontier and corpus computations on the Xeon-Phi (like on the GPU). Here, the modification flags are scalar boolean values automatically transferred between the CPU and the Xeon-Phi at the beginning and at the end of each `offload` directive. Moreover, when declared (at top of the source file) the `heterMIC` routine has been labeled to be a Xeon-Phi routine, in order the Intel compiler applies all the suited optimizations. In particular, it can take into account the vectorization directives embedded in the source code of this routine.

Lines 18-25 are the synchronous CPU computation of the two CPU frontiers (CPU/GPU and CPU/Xeon-Phi), using OpenMP multithreading and Intel compiler vectorization on SSE or AVX units. The `CellUpdate` routine (lines 22-23) is compiled only for CPU, and Intel compiler can achieve vectorization optimization adapted to CPU cores. Lines 27-35 perform the asynchronous transfers of the CPU frontiers to the GPU, using a new CUDA *bottom frontier* stream (`BF`), and to the Xeon-Phi, using a new signal clause (`dummyTransfer`). According to our generic algorithm (see Sec. 4) these transfers are done only if the frontiers have been modified on the CPU.

At lines 37-40, the CPU waits for the end of the GPU frontier computation and modification flag transfer (`TF` CUDA stream) before to asynchronously transfer the GPU frontier to the CPU if it has been modified on the GPU. Lines 42-46 execute the same operations for the Xeon-Phi. Lines 48-61 correspond to the computation of the CPU part of the problem excepted its frontiers (the CPU *corpus*). Again, it is a multithreaded and vectorized computation, and a modification flag is raised when a modification occurs (line 60).

Lines 63-65 contain synchronization barriers for the end of the CPU/GPU frontier transfers and the end of the GPU corpus computation. Lines 67-71 perform the same operations on the Xeon-Phi. Lines 73-77 permute current and previous array pointers on the Xeon-Phi, the GPU, and the CPU. These permutations are not complex but their implementations differ: pointers on GPU memory are CPU variables managed by the CPU, while pointers on Xeon-Phi memory are stored on the Xeon-Phi and must be permuted by the Xeon-Phi. Finally, pointers on the Xeon-Phi memory have *mirror pointers* on the CPU (due to the offload semantics) that must be permuted too (line 77). Lines 78-81 compute global modification flags corresponding to each computing device.

Listing 1.2. Computation loop implementation

```
 1 for(iter=0; iter<nbIters && (modifCPU || modifGPU || modifMIC); ++iter) {
 2   ...
 3   // ASYNCHRONOUS GPU computations and frontier transfer to the CPU ------------
 4   gpuOneIterationLast(gpuPrec+..., gpuCrt+..., 1 /*1 line*/, nbCol, ..., TF);
 5   cudaMemcpyFromSymbolAsync(&modifGPULast, pt_gpu_modif_hf, ...,
 6                             cudaMemcpyDeviceToHost, streamTab[TF]);
 7   gpuOneIteration(gpuPrev, gpuCrt, ..., nbLinsGPU-1, nbCol, ..., CORPUS);
 8   cudaMemcpyFromSymbolAsync(&modifGPUCorpus, pt_gpu_modif_corpus, ...,
 9                             cudaMemcpyDeviceToHost, streamTab[CORPUS]);
10   // ASYNCHRONOUS Xeon-Phi computations and frontier transfer to the CPU -------
11   #pragma offload target(mic:0) nocopy(botFrontMIC, micPrev, _micCrt, ...) \
12                                 signal(&botFrontMIC)
13   { heterMIC(micPrev, _micCrt, ..., botFrontMIC, &modifMICFirst); }
14   #pragma offload target(mic:0)                                             \
15        nocopy(micPrev, _micCrt, micTer, ..., modifMICCorpus) signal(&_micCrt)
16   { heterMIC(micPrev, _micCrt, ..., &modifMICCorpus); }
17   // Synchronous CPU frontier computations ------------------------------------
18   #pragma omp parallel num_threads(nbTHet)
19   { #pragma omp for
20     #pragma simd
21     for (col=1; col <= nbCol; ++col) {
22        CellUpdate(prev, crt, ..., firstLine + col, &modifCPUFirst);
23        CellUpdate(prev, crt, ..., lastLine  + col, &modifCPULast);
24     }
25   }
26   // ASYNCHRONOUS CPU frontiers transfers to accelerators ----------------------
27   if (modifCPUFirst)
28     cudaMemcpyAsync(gpuCrt+..., cpuCrt+..., nbCol*sizeof(double),
29                     cudaMemcpyHostToDevice, streamTab[BF]);
30   if (modifCPULast) {
    #pragma offload target(mic:0)                                             \
         in(topFrontMIC:length(nbCol+1) alloc_if(0) free_if(0) align(ALIGN)) \
         in(dummyTransfert) nocopy(_micCrt) signal(&dummyTransfer)
    { memcpy(_micCrt+1, topFrontMIC+1, nbCol * sizeof(double)); }
  }
  // ASYNCHRONOUS GPU frontier transfer to the CPU ----------------------------
  cudaStreamSynchronize(streamTab[TF]);
  if (modifGPULast)
    cudaMemcpyAsync(topFrontCPU, gpuCrt+..., nbCol*sizeof(double),
                    cudaMemcpyDeviceToHost, streamTab[TF]);
  // ASYNCHRONOUS Xeon-Phi frontier transfer to the CPU -----------------------
  #pragma offload_wait target(mic:0) wait(&botFrontMIC)
  if (modifMICFirst)
    #pragma offload_transfer target(mic:0)                                     \
         out(botFrontMIC:length(nbCol+1) alloc_if(0) free_if(0) align(ALIGN)) \
         signal(&botFrontMIC)
  // Synchronous CPU corpus computation ---------------------------------------
  #pragma omp parallel for schedule(dynamic) num_threads(nbTHet)
  for (lin=2; lin < nbLinesCPU; ++lin)            // multithreaded loop
    #pragma simd                                  // guided vectorization
    for (col=1; col <= nbCol; ++col) {
      size_t ind = lin*TotCol + col;
      #pragma ivdep                               // guided vectorization
      for (dl=-1; dl <= 1; ++dl)
        for (dc=-1; dc <= 1; ++dc) {
          cost = cpuPrev[ind + ...] + sqrt(...); // Compute the new distance
          minCost = min(cost, minCost);           // Store current min distance
        }
      cpuCrt[ind] = minCost;                      // Store the min distance
      if(minCost < cpuPrev[ind]) modifCPUCorpus = true; // Rise modif flag if
    }                                             // min distance has changed
  // SYNCHRONIZATION BARRIER on the end of GPU operations ---------------------
  if (modifGPULast)  cudaStreamSynchronize(streamTab[TF]);
  if (modifCPUFirst) cudaStreamSynchronize(streamTab[BF]);
  cudaStreamSynchronize(streamTab[CORPUS]);
  // SYNCHRONIZATION BARRIER on the end of Xeon-Phi operations ----------------
  #pragma offload_wait target(mic:0) wait(&_micCrt)
  if (modifMICFirst)
    #pragma offload_wait target(mic:0) wait(&botFrontMIC)
  if (modifCPULast)
    #pragma offload_wait target(mic:0) wait(&dummyTransfer)
  // SYNCHRONOUS PERMUTATION of current and previous arrays on each device -----
  #pragma offload target(mic:0) nocopy(micPrev, _micCrt)
  { double *tmp = _micCrt; _micCrt = micPrev; micPrev = tmp; } // Phi ptr on Phi
  tmp = gpuPrev; gpuPrev = gpuCrt; gpuCrt = tmp;                 // GPU ptr on CPU
  tmp = prev; prev = crt; crt = tmp;                             // CPU ptr on CPU
  micCrt = crt + ...; micPrev = prev + ...; ...         // Mirror Phi ptr on CPU
  // Modification flags computations
  modifGPU = modifGPULast  | modifGPUCorpus;
  modifMIC = modifMICPre   | modifMICCorpus;
  modifCPU = modifCPUFirst | modifCPULast | modifCPUCorpus;
}
```

Finaly, mixing the Nvidia CUDA and Intel offload programming paradigm leads to a complex syntax, but we succeeded to implement our generic asynchronous parallel algorithm, including many *computations / data transfers* overlapping. Vectorization can be used on the CPU and Xeon-Phi kernels, but must be implemented in separated routines so that the compiler can easily generate efficient code for each architecture. Some different CUDA *streams* must be associated with the GPU kernel launches and data transfers, in order to concurrently run sequences of operations.

The final step of our generic algorithm mainly consists in retrieving the results from the accelerators. The mechanisms used to implement this are similar to the initialization step and consists in overlapping the results transfers from GPU and Xeon-Phi by using an asynchronous transfer from the GPU. So, we do not give code sample for that part. Instead, we mention that on the Xeon-Phi, it is necessary to copy the results into the transferable array (the one initialized with the *in* offload directive, line 5 in Listing 1.1) when the number of iterations is odd.

7 Testbeds and benchmark data for shortest path computation

The machine used at CentraleSupelec is a Dell R720 server containing two 6-cores Intel(R) Xeon(R) CPU E5-2620 at 2.10GHz, with 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 second accelerator is a Nvidia GPU *GeForce GTX Titan Black* (Kepler architecture) with 2880 CUDA cores. The machine used at Loria is a Dell R720 server containing two 8-cores Intel(R) Xeon(R) CPU E5-2640 at 2.00GHz, with 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 second accelerator is a Nvidia GPU *Tesla K40m* (Kepler architecture) with 2880 CUDA cores.

In order to get representative results with the shortest path problem, we used elevation data of a large area in the French and Italian Alps from the NASA SRTM project [1]. The 2D grid resolution is 30m, with 10803 columns and 18005 lines ($\approx$324Km$\times$540Km), and the elevations range from -87m to 4797m. The overall calculation takes 9005 iterations with a target point located at the center of the area. In this study, we did not use larger datasets to be able to compare executions on single devices.

8 Benchmark results with different hybrid machines

In the following paragraphs, we present the performance results of our algorithm in several steps. Firstly, we present the performance of each device alone in order to give an idea of the relative computing powers of the devices inside each machine. These relative powers are useful to deduce the theoretical optimal partitioning of the 2D grid by using a simple static load balancing scheme. Then, we consider double device mixing, where two devices cooperate (GPU-CPU, CPU-PHI, GPU-PHI). Finally, the results with the three cooperating devices are presented. All the following experiments are averages of several executions although we observed a very limited standard deviation.

Standalone performance In Tab. 1 are presented the performance results for each device in both machines. Speeds are expressed in points per second instead of GFlops as the computation amount varies among the points. Relative computing powers are estimated according to the total power of each machine, deduced from the aggregation of the devices speeds. As expected, the CPU is the slowest device inside each machine and the GPU is the fastest one. However, strong differences between the two machines are observed, which is interesting as it provides significantly different hardware configurations. It allows us to see how our algorithm behaves in either cases. Absolute speeds are useful here to deduce the relative computing powers of the devices but they are omitted in the following tables as they are redundant information with gains.

	Loria machine			CentralSupelec machine		
	CPU	PHI	GPU	CPU	PHI	GPU
Speed (pts/s)	6.35E+008	1.56E+009	3.00E+009	4.46E+008	1.74E+009	1.74E+009
Relative powers	12.20%	29.97%	57.83%	11.36%	44.32%	44.32%

Table 1. Performance results of each device alone and relative computing powers

	Loria machine						CentraleSupelec machine					
	GPU-CPU		CPU-PHI		GPU-PHI		GPU-CPU		CPU-PHI		GPU-PHI	
	Cut line	gain	Cut line	gain	Cut line	gain	Cut line	gain	Cut line	gain	Cut line	gain
-1000	13900	-47.74%	4200	29.14%	10900	-14.80%	13300	-48.14%	1700	7.40%	8000	19.99%
-500	14400	-45.88%	4700	33.32%	11400	-13.27%	13800	-46.64%	2200	10.38%	8500	22.31%
opt. cut	14900	-42.92%	5200	37.19%	11900	-14.58%	14300	-44.66%	2700	13.97%	9000	23.01%
+500	15400	-37.03%	5700	28.37%	12400	-15.91%	14800	-41.61%	3200	13.94%	9500	22.32%
+1000	15900	-33.24%	6200	15.69%	12900	-17.59%	15300	-37.58%	3700	-0.92%	10000	19.29%

Table 2. Performance results and gains around theoretical optimal cutting lines for both machines.

Double device mixing According to the single devices results, the relative computing powers of the devices in each machine are used to perform static load balancing. The optimal grid partitioning into two horizontal strips is computed for every couple of devices. For example, GPU is 4.7 times faster than CPU in the Loria machine, then the strip associated to the GPU will be 4.7 times larger than the CPU one, leading to a cutting line (opt. cut in Tab. 2) of 14900. The results are presented in Tab. 2 where the gains are computed relatively to the speed of the fastest device involved in the mixing.

First of all, we can see that some gains are negative and others are positive. Negative gains indicate that the mixing is less efficient than the fastest device alone. With both machines, the GPU-CPU mixing does not provide any gain. Indeed, even extremely imbalanced partitions with most of the grid on the GPU (17500 lines) provided a loss of 19.42% on the Loria machine and 8.20% on the CentralSupelec one. Those results show that the mixing overheads are never compensated by the computation gains with that grid size. Concerning the CPU-PHI mixing, we obtain significant gains on both machines, showing that our scheme is efficient for mixing those two devices. Finally, we obtain diverging results between the two machines for the GPU-PHI mixing. This mainly comes from the difference of powers between the two GPU devices. As the powers of the PHI and GPU are similar on the CS machine, their mixing is actually efficient whereas on the Loria machine the two devices have too different powers to compensate the mixing overheads. Also, it is surprising to get so different gains between GPU-CPU and CPU-PHI on the CS machine and this will require a deeper analyze.

CPU	Loria machine					CPU	CentraleSupelec machine					
PHI	GPU-CPU cut line					PHI	GPU-CPU cut line					
cut line	9400	9900	10400	10900	11400	cut line	7000	7500	8000	8500	9000	9500
11600	1.28%	-3.38%	-7.07%	-11.66%	-13.60%	9000	24.87%	28.94%	27.08%	25.63%	23.47%	
12100	-2.13%	1.55%	-2.31%	-5.70%	-9.57%	9500	3.05%	23.26%	32.60%	30.65%	28.41%	22.53%
12600	-12.73%	-5.19%	3.00%	-1.03%	-5.78%	10000	-9.33%	1.61%	21.92%	34.83%	32.30%	26.24%
13100	-21.84%	-16.00%	-9.32%	-4.51%	-2.01%	10500	-19.32%	-10.69%	0.07%	20.51%	37.01%	30.83%
13600	-30.68%	-23.83%	-20.22%	-12.83%	-7.13%	11000	-27.32%	-20.27%	-11.68%	-1.08%	19.08%	32.01%

Table 3. Performance gains for the triple mixing on both machines.

Triple device mixing For the sake of concision, we present in Tab. 3 only the percentage gains for both machine around their respective optimal cutting lines between GPU-CPU and CPU-PHI. For the CS machine, we added an extra column of GPU-CPU cutting line as the maximal performance was obtained at the limit of the initial set. It can be observed that our heterogeneous algorithm hardly obtains positive gains on the Loria machine whereas it obtains up to 37% gain over the GPU on the CS machine (better than double mixing). The bad performance with the Loria machine is quite unexpected. It probably comes from the different hardware configuration as well as the older Intel compiler used. However, results with the CS machine show that our algorithmic scheme can provide significant gains. All those results show that the efficiency of our approach is sensitive to the hardware configuration as well as to the problem size. This will deserve a more complete investigation over the behavior of our scheme according to larger grid sizes.

9 Conclusion

An algorithmic scheme has been presented that enables the cooperation of several computing devices of different types (CPU, GPU, PHI) to process a stencil application on a single machine. Our scheme makes an intensive use of asynchronous computations and data transfers in order to obtain an efficient overlapping of both operations.

The scheme has been evaluated with different combinations of devices cooperation on a representative elevation map of the Alps region. Results show that our algorithm can provide significant gains either with two or three devices. It has been observed that better gains are obtained when the relative computing powers of the devices are closer.

Several extensions should be interesting such as studying the behavior of our scheme with larger grid sizes, including a dynamic load balancing by re-partitioning the grid during the algorithm execution, as well as extending our scheme to cluster systems by adding explicit inter-machine communications.

References

1. Shuttle Radar Topography Mission (2000), https://lta.cr.usgs.gov/SRTM1Arc
2. Calandra, H., Dolbeau, R., Fortin, P., Lamotte, J.L., Said, I.: Evaluation of Successive CPUs/APUs/GPUs Based on an OpenCL Finite Difference Stencil. In: 21st Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP) (Feb 2013)
3. Contassot-Vivier, S., Vialle, S.: Algorithmic scheme for hybrid computing with CPU, Xeon-Phi/MIC and GPU devices on a single machine. In: ParCo'2015. Edinburgh, UK (Sep 2015)
4. Courtès, L.: C language extensions for hybrid CPU/GPU programming with StarPU. Tech. Rep. 8278, INRIA (2013)
5. Dijkstra, E.: A note on two problems in connexion with graphs. Numerische Mathematik 1(1), 269–271 (1959), http://dx.doi.org/10.1007/BF01386390
6. Fang, J., Varbanescu, A.L., Imbernon, B., Cecilia, J.M., Perez-Sanchez, H.: Parallel computation of non-bonded interactions in drug discovery: Nvidia GPUs vs. Intel Xeon Phi. In: 2nd Intl Work-Conf. on Bioinformatics and Biomedical Engineering. Granada, Spain (2014)
7. Gaster, B., Howes, L., Kaeli, D., Mistry, P., Schaa, D.: Heterogeneous Computing with OpenCL. Morgan Kaufmann, 2nd edn. (2012), ISBN 9780124058941
8. Gysi, T., Grosser, T., Hoefler, T.: MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In: Proceedings of the 29th ACM on Intl Conf on Supercomputing. pp. 177–186. ICS '15, ACM, New York, USA (2015)
9. Jin, G., Lin, J., Endo, T.: Efficient utilization of memory hierarchy to enable the computation on bigger domains for stencil computation in CPU-GPU based systems. In: 2014 Intl Conf on High Performance Computing and Applications (ICHPCA) (Dec 2014)
10. Su, H., Wu, N., Wen, M., Zhang, C., Cai, X.: On the GPU-CPU Performance Portability of OpenCL for 3D Stencil Computations. In: Proceedings of the 2013 Intl Conf on Parallel and Distributed Systems. ICPADS'13, Washington, DC, USA (2013)
11. Szustak, L., Rojek, K., Olas, T., Kuczynski, L., Halbiniak, K., Gepner, P.: Adaptation of MPDATA heterogeneous stencil computation to Intel Xeon Phi coprocessor. Scientific Programming (2015)
12. Wende, F., Steinke, T.: Swendsen-Wang Multi-cluster Algorithm for the 2D/3D Ising Model on Xeon Phi and GPU. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '13, ACM, New York, NY, USA (2013)