SGPU 2: a runtime system for using large applications on clusters of hybrid nodes

Matthieu Ospici[*] [‡] [§], Dimitri Komatitsch[†] [¶], Jean-François Méhaut[§], Thierry Deutsch[‡]
[*]*BULL SAS, 1 rue de Provence, 38432 Echirolles, France*
matthieu.ospici@imag.fr
[§] *Grenoble University, 38000 Grenoble, France*
jean-francois.mehaut@imag.fr
[‡] *Laboratoire L_Sim, INAC/SP2M CEA Grenoble, 17, rue des Martyrs 38054 Grenoble, France*
thierry.deutsch@cea.fr
[†] *Géosciences Environnement Toulouse, CNRS Université Paul Sabatier, Observatoire Midi-Pyrénées,*
14 avenue Édouard Belin, 31400 Toulouse, France.
dimitri.komatitsch@get.obs-mip.fr
[¶] *Institut Universitaire de France, 103 boulevard Saint-Michel, 75005 Paris, France.*

***Abstract*—In this article, we consider hybrid architectures that consist of standard CPU cores associated with accelerators (such as GPUs). These architectures are increasingly employed in large computing centers. We develop a strategy designed to deal with hybrid computing architectures from the computing performance and programmability points of view. We focus on hybrid computing clusters that consist of a potentially high number of standard CPU cores combined with some accelerators. Although such hardware is increasingly being used, because of related programming difficulties, only a small number of large applications use all the computing resources of such hybrid systems (i.e. both CPU cores and accelerators).**

To allow large applications to use both CPU cores and accelerators, we introduce SGPU 2, a runtime system that offers a programming model designed to express the fact that a computation can be performed on both CPU cores and accelerators. We first show how SGPU 2 manages the different computing resources of a hybrid architecture and how it can be used with a classical existing seismic wave propagation code; we then use this simulation code to benchmark SGPU 2.

***Keywords*-Hybrid architecture, GPU computing, High performance computing**

I. Introduction

The multicore concept has been applied with success to Graphics Processing Unit (GPU) processors. These kinds of processors contain hundreds of cores and were initially designed for video games or high performance 3D calculations. Seeing that GPUs can reach a single precision performance peak of 1 teraflops, and double precision peak of a half teraflops (for instance on the NVIDIA FERMI architecture) with a high memory bandwidth, it is particularly interesting to use the characteristics of such GPUs for general purpose calculations (GPGPU, General-Purpose computation on Graphics Processing Units). A GPGPU system is built with a standard CPU associated with one or more GPUs. Such a system that combines CPUs and GPUs is called a "hybrid system".

One faces several difficulties when developing software for hybrid systems. First, many programming environments exist to develop programs on GPUs (ATI Stream, NVIDIA CUDA, OpenCL), but these environments make development for GPUs very different from classic ones on CPUs; reaching GPU peak performance is challenging. Second, the GPU code must be integrated with a low-level software stack (MPI, OPEN_MP, etc.); and finally it has to be accessible by the existing code.

In this article, we focus on clusters consisting of hybrid nodes. To use this kind of hardware efficiently, one can consider several approaches. We focus on strategies that use both CPU cores and GPUs on a given machine. This approach can be difficult because programmers must first determine which computations can be executed concurrently on both CPU cores and GPUs. Secondly, they have to implement the computations. Finally, because GPUs and CPU cores do not have the same calculation power, the right calculation amount assigned to the computing units must be carefully determined in order to have good load balancing between CPU cores and GPUs.

To address these issues, we introduce a software runtime called SGPU 2 that offers programmers a way of expressing a computation that can be performed on both CPU cores and accelerators (such as GPUs). SGPU 2 also manages the execution in order to have good load balancing between CPU cores and accelerators. We have carefully designed SGPU 2 to be fully compatible with FORTRAN or C based MPI programs in order for it to be usable in large scientific simulations codes.

We will first summarize previous works on execution models for hybrid system. We will then describe the SGPU 2 system and state the work carried out to port the classical seismic wave propagation code SPECFEM3D to SGPU 2. Finally we will show some experimental results.

II. Related work on hybrid cluster use

The current trend in large computing centers is to install hybrid clusters based on general processor architecture and accelerators. The best known example is the Tianhe-1A (Intel processors + NVIDIA Fermi accelerators) machine, currently ranked second of the TOP500[1] with 2.5 PFlops in June 2011.

A. Technological context

A hybrid computing cluster with GPUs is a cluster in which nodes are hybrid i.e. they are equipped with one or

[1]TOP 500 www.top500.org

more GPUs. Typically, on a current hybrid cluster node, there are six or twelve standard CPU cores (Intel Xeon or AMD Opteron) associated with one or two GPUs.

Currently, using a GPU based cluster requires two kinds of communications: firstly, communication between nodes, set up through a high performance network interconnect such as Infiniband, and secondly, data transfers between the main memory and the GPU memory. Generally, applications initiate node-to-node communications via an MPI library and CPU to GPU transfers via a specific function call.

B. Strategies for using hybrid architectures

We analyze three main approaches to use hybrid architectures:

- **Approach 1:** GPUs are only used to execute kernels, which is the most straightforward approach.
- **Approach 2:** Some kernels are executed on the GPUs and some on CPU cores.
- **Approach 3:** Kernels are split into two sub-kernels intended to be executed simultaneously on GPUs and CPU cores. Our software library SGPU 2 is based on this approach.

In the two next sections, we study some examples of execution supports designed to use GPUs, and we then describe some scientific applications exploiting hybrid architectures.

C. Software stacks designed to deal with hybrid architectures

In this section we present several software environments that deal with hybrid architectures programming. We classify these environments into two categories.

1) Environments that only manage the accelerators: Each hardware manufacturer provides software environments to address its accelerators. For the CELL hybrid processor, IBM provides several software packages to use it efficiently [1]. GPUs also have their programming environments, such as CUDA [2] for NVIDIA based GPUs. Because these environments only deal with GPUs, they are well suited for the GPU usage depicted in approach 1. However, with efforts by the programmers, they can also be used for approaches 2 and 3.

2) Environments that manage CPU cores and accelerators: In this category, it is important to distinguish between software packages that provide scheduling to optimize the load balancing between both CPU cores and GPUs, and software packages without scheduling.

Without advanced scheduling: Companies such as CAPS Enterprise with HMPP [3], or PGI [4] provide solutions for automatically translating an existing CPU code (Fortran, C...) into an equivalent GPU code. With these tools, it can be easier to port existing programs on GPUs. We can also cite OpenCL [5] in this category. A program written in OpenCL should be executable on GPUs from AMD and NVIDIA and also on multicore CPUs. However, because the issue of computing simultaneously on CPU cores and GPUs efficiently is not addressed, these tools are generally used for approach 1.

With advanced scheduling: Many executive supports provide efficient scheduling for hybrid architectures. Many of them use a history-based time estimator designed to find the computing efficiency of GPUs and CPU cores. We can cite StarPU [6], a runtime system designed to efficiently schedule tasks on CPU cores and accelerators (GPU, Cell). The Harmony [7] runtime system provides a way of automatically extracting parallelism based on a speculative execution. StarPU and Harmony are designed to deal with the method depicted in approach 2. For an example of a runtime based on approach 3 we can cite Qilin [8], which manages kernels split into both a CPU part and a GPU part. To use Qilin, a problem has to be expressed with a CPU part (Intel TBB) and a GPU part (written in CUDA).

D. Hybrid applications

A significant amount of research has been devoted to analyzing how scientific applications need to be modified to be able to compute on GPUs. For instance, BigDFT [9] is an ab-initio simulation code built around the MPI library. It has been adapted to be able to use GPUs by using CUDA [10], and more recently OpenCL. As designed, when GPUs work, CPU cores are generally idle. SPECFEM3D [11], a seismic wave propagation simulation code, currently uses the same approach. The computing intensive part of the code is deported on the GPU. While GPUs compute, SPECFEM3D performs non blocking MPI communications simultaneously in order to overlap their cost. We can cite other applications [12], [13] for which the problem of simultaneously computing on both CPU cores and GPUs is not taken into consideration.

On the contrary, few large applications try to use the power of all hardware resources. DL_POLY [14], a dynamic molecular application, can split a computation between CPU and GPU. DL_POLY can automatically compute the right amount of data to send to GPUs and CPUs based on a calibration phase. However the implementation of the strategy is directly linked of using particular software, the authors do not provide ways to use it elsewhere. DL_POLY uses a hybrid computing strategy based on approach 3.

As we have recalled, a large number of codes only use CUDA or OpenCL to address GPU. Advanced runtime systems such as Qilin or StarPU seem to be difficult to build in large MPI applications. Consequently, with SGPU 2, our goal is to provide a general programming model, based on approach 3, that is usable for a large number of MPI based scientific applications in order to manage the complexity of hybrid computing programming.

III. The SGPU 2 runtime system

In hybrid architectures, the major challenge is to use all the computing resources efficiently. Because central CPU cores and accelerators are powerful, trying to use both should be the goal. However, accelerator power and CPU core power are different, depending on the types of computation and hardware. In a given hardware, some computations can be 10 times faster on GPUs than on CPU cores, whereas other kinds of computation might

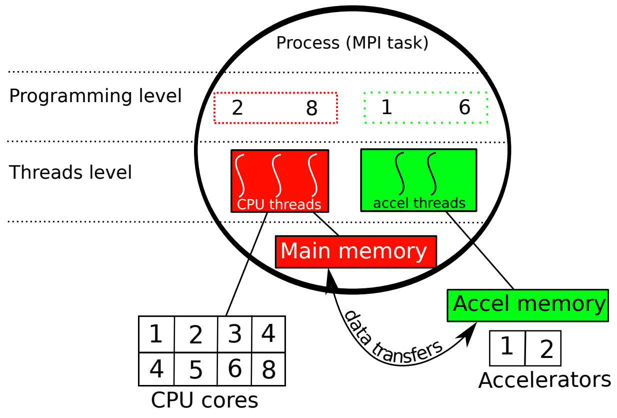


Figure 1: The SGPU 2 programming model.

only be twice as fast [15]. In SGPU 2 we thus address the problem of splitting a computation between CPU cores and accelerators by processing computations on both CPU and accelerators. Consequently, many data transfers have to be performed: transfers from the main memory to accelerators at the beginning of a computation, and then transfers from the accelerators to the main memory at the end of the calculation.

A. The SGPU 2 programming model

The SGPU 2 programming model is based on the classical process/thread organization found in a large number of operating systems. In classical applications, a process can create some threads to parallelize its execution. We add to this model the notion of “accelerator threads”: a process can create some CPU threads and some “accelerator threads” to parallelize an execution between CPU cores and accelerators. SGPU 2 defines two thread types:

- **CPU threads**, which are standard POSIX threads or OpenMP parallel sections holding code executable on CPU cores.
- **Accelerator threads**, which are threads defined by SGPU 2. Their programming interface is similar to Posix threads. They hold code executable on accelerators and can perform data transfers.

1) Thread sets: SGPU 2 threads are organized in collections called *thread sets*. A thread set can hold either a fixed number of threads, or a variable number of threads. In any case, when a thread set is created, it has to be tagged with the minimum($threads_{min}$) and maximum($threads_{max}$) numbers of computing resources to allocate when the thread set is executed. In other words, $threads_{min}$ and $threads_{max}$ correspond to the minimum and maximum hardware resources usable by a given thread set. Figure 1 shows a process that creates a CPU thread set with $threads_{min} = 2, threads_{max} = 8$ and a GPU thread set with $threads_{min} = 1, threads_{max} = 6$.

2) Hybrid computations: To allow for hybrid computations with SGPU 2, the idea is to parallelize a computation with a set of CPU threads and a set of accelerator threads. Parallelization can be set up by splitting data among CPU and accelerator memory or by distributing threads across CPU cores and accelerators. Because we do not have ways of automatically determining how to parallelize a given computation, SGPU 2 needs to be guided by programmers.

Consequently, a function designed to split computations, called a *split function*, has to be provided. In our programming model, the *split function* can be viewed as an operator designed to set up the CPU memory space and the accelerators memory space in order to prepare them for a hybrid computation. Thus, this function is the place to split data computation and initiate CPU to accelerator data transfers. Symmetrically a function designed to merge computation from accelerator threads and CPU threads must be provided. This function, called a *merge function*, can perform accelerator to CPU transfers. Both split and merge functions can be threaded to address more than one accelerator.

However, the problem of providing the right amount of calculations to the CPU threads and to the accelerator threads remains in order to balance the load between CPU cores and accelerators. The solution that we implement is for SGPU 2 to use a history-based time estimator. Every time a thread set is launched, SGPU 2 tracks its execution configuration and stores it in a database. A so-called execution configuration holds execution time, the size of the data processed and the number of threads allocated to the thread set. This strategy is efficient for iterative programs, when the same thread set is used many times with different data sizes. To estimate the execution time of an unknown execution configuration we have used an analytical regression algorithm whose implementation can be found in [16]. This strategy is similar to those employed by Qilin [8], Harmony [7] or StarPU [6]. Once the execution time of a thread set is known, SGPU 2 can compute how many computations it needs to send to the CPU cores and to the accelerators. This distribution is supplied to both the *split* and *merge* functions. The programmer needs to write a code that takes into consideration the distribution provided to these functions in order to send the right amount of computations to both CPU cores and accelerators.

3) Thread sets submission: To submit GPU or accelerator computations, corresponding thread sets must be submitted to SGPU 2. For hybrid computations, one CPU thread set, one GPU thread set and the split and merge functions must be submitted. When computations are submitted, the execution is asynchronous from the point of view of the programmer. SGPU 2 provides functions designed to wait for a computation to ensure that an execution is finished.

B. Execution model

In the previous section we described the notion of thread sets, which is the basic scheduling unit in SGPU 2. A thread set that contains CPU threads is called a CPU thread set and labeled TS_{CPU} in this section. An accelerator thread set is labeled TS_{Accel}.

1) Thread scheduling: SGPU 2 schedules thread sets to free resources. Figure 1 shows an example with 8 CPU cores and 2 GPUs to be used with SGPU 2.

The scheduler is built around three queues in which thread sets are submitted:

- One "CPU queue" that handles calculations that use only the CPU cores (CPU thread set)
- One "accelerator queue" that handles calculations that use only the accelerators (accelerator thread sets)
- One "hybrid queue" that handles hybrid calculations (accelerator thread set and CPU thread set plus split and merge functions)

Procedure 1 getRes: find free resources

```
Input: $resource$ object (manage free and busy computing resources) and one thread set $TS$
Output: resources allocatable for the thread set
  $n \leftarrow max(resources.free, TS.max)$
  if $n \leq TS.min$ then
    $avail\_resources \leftarrow 0$
  else
    $avail\_resources \leftarrow n$
  end if
```

Procedure 2 thread set scheduling

```
Input: $CPU\_queue$, $Accel\_queue$ and $Hybrid\_queue$
Output: a thread set scheduling
  loop
    if $CPU\_queue.not\_empty$ then
      $TS_{CPU} \leftarrow$ bottom of the CPU queue
      getRes($TS_{CPU}, avail\_resources$)
      if $avail\_resources \neq 0$ then
        {allocate $avail\_resources$ CPU cores}
        {exec thread set}
      end if
    end if
    if $Accel\_queue.not\_empty$ then
      {...Like to CPU processing...}
    end if
    if $Hybrid\_queue.not\_empty$ then
      $Hyb\_elem \leftarrow$ bottom of the Hybrid queue
      getRes($Hyb\_elem.TS_{CPU}, avail\_resources_{CPU}$)
      getRes($Hyb\_elem.TS_{Accel}, avail\_resources_{Accel}$)
      if $avail\_resources_{CPU} + avail\_resources_{accel} \neq 0$ then
        if $avail\_resources_{CPU} \neq 0$ then
          $repartition \leftarrow 0\%$
        else if $avail\_resources_{Accel} \neq 0$ then
          $repartition \leftarrow 100\%$
        else
          $repartition \leftarrow$computeRep($avail\_resources_{CPU}$, $avail\_resources_{Accel}$, $Hyb\_elem.id$)
        end if
        {allocate $avail\_resources_{CPU}$ CPU cores}
        {allocate $avail\_resources_{Accel}$ accelerators}
        {exec cut_function($repartition$)}
        {exec both CPU and GPU thread sets}
        {exec merge_function($repartition$)}
      end if
    end if
  end loop
```

To schedule the thread sets submitted, SGPU 2 tries to allocate a number of resources closest to the $threads_{max}$ provided at the thread set definition. As an example, in Figure 1, SGPU 2 creates three CPU threads bound to three CPU cores (we assume there are only three free CPU cores). This policy is shown in Procedure 1.

Global scheduling is depicted in Procedure 2: when a thread set is submitted, one of the three queues is filled up. Each queue is successively analyzed to find if there are free resources on which thread sets can be executed. For the case of the hybrid queue, the time estimator is activated. Its role is to determine how many computations the CPU side and the GPU side have to treat in order to finish at the same time. Threads within a thread set are scheduled by the operating system in the case of a CPU thread. Conversely, GPU threads are directly scheduled by SGPU 2 on allocated accelerators.

C. Software architecture

SGPU 2 consists of a software library and a process running in each cluster node. Programs can perform MPI operations without changes and submit work to SGPU 2 at any time.

The SGPU 2 process interacts with the application through POSIX shared memory. CUDA needs to have pinned memory to perform asynchronous data transfers between CPU and GPU. To pin a region of POSIX shared memory, we use a CUDA 4.0 function called `cudaHostRegister`.

IV. Using SGPU 2 in a scientific application

SGPU 2 is designed to be used with large MPI applications. Nevertheless, the split and merge abstractions can be difficult to implement for some algorithms. Indeed, the computations have to be split between CPU cores and accelerators and problems that manage complex data (such as complex operations on a graph) can be very difficult to split and merge.

In this article, we have chosen to use a scientific code in which the main compute intensive part is well adapted to be split on both CPU cores and accelerators. Let us show how we have successfully used SGPU 2 in the context of that large existing MPI application in geophysics.

A. SPECFEM3D description

SPECFEM3D [11] is a seismic wave propagation simulation code based on the spectral-element method. It can simulate seismic wave propagation in the Earth at very high resolution. The high level of accuracy reached implies that amounts of data handled as well as computations can be very large. Computations are parallelized with MPI, thus, SPECFEM3D is suitable for large computer clusters.

1) GPU computations in SPECFEM3D: Details about the main principles behind porting SPECFEM3D to GPUs can be found in [17]. In what follows we focus on the main compute intensive part of the code, which consists of a loop that represents a time iteration algorithm. This loop consists of three parts: parts 1 and 3 contain a CUDA kernel while part 2 contains a succession of n calls to the same CUDA kernel (n depends on the numerical mesh used and is generally between 15 and 30). Data

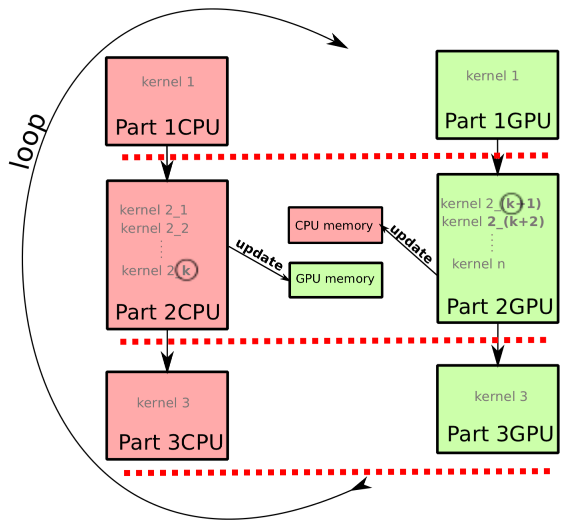


Figure 2: SGPU 2 port: both CPU cores and GPUs are used, extra data transfers have to be performed to update both CPU and GPU memories. After each dotted line, CPU and GPU memories should contain exactly the same data.

dependencies require that parts 1, 2 and 3 be executed serially, and all n calls to the kernel from part 2 have also to be done serially.

B. Porting SPECFEM3D to SGPU 2

Figure 2 depicts how we have ported SPECFEM3D to SGPU 2. To use SGPU 2, the most important work to do is to decide how to split computations between CPU cores and GPUs. Because CPU cores and GPUs process only part of a given computation, extra data transfers must be implemented to update the different memories with the updated data.

After parts 1, 2 and 3, CPU and GPU memories must hold exactly the same data, as depicted by dotted lines in Figure 2. Because computations are split across CPU cores and GPUs, extra data transfers must be performed in both CPU to GPU and GPU to CPU directions to update the main memory and the GPUs memory.

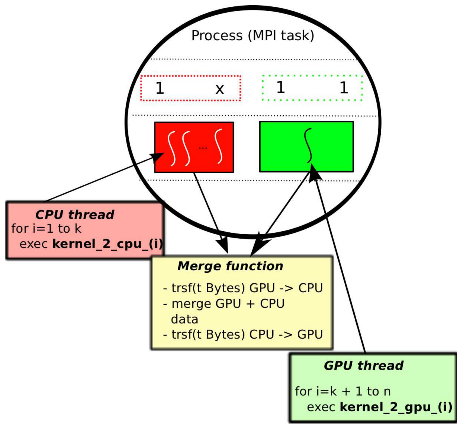
1) Parts 1 and 3 in SGPU 2 : For parts 1 and 3, which consist of very simple computations (basically, a sum of two large vectors multiplied by constant coefficients), splitting the computation in two parts is very costly due to data transfers to update memories. Consequently, we have chosen to duplicate parts 1 and 3 on both GPU and CPU to avoid extra data transfers. The CPU implementation of parts 1 and 3 is parallelized with OpenMP.

2) Part 2 in SGPU 2: For the succession of kernel_2_1...n calls, which is the most intensive part of the loop, hybriding the execution is a crucial step. Each kernel reads and writes non-consecutive elements in memory, consequently, it is very difficult to split data. Thus, we prefer to split the *computations*. This means that we simultaneously execute k kernels on the CPU and $(n - k)$ kernels on the GPU, which is possible because CPU memory and GPU memory are separated.

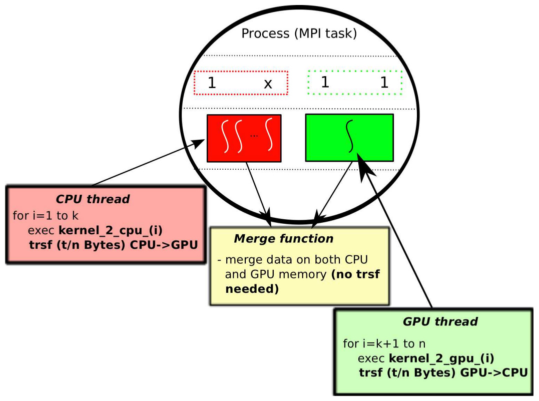
Figure 3 depicts two possible implementations of part 2: contrary to parts 1 and 3, hybrid computations are used, therefore both split and merge functions are mandatory. For the CPU thread set, we have kept the original serial loop of the SPECFEM3D code. Nevertheless we have parallelized the implementation of kernel_2 with OpenMP in order to use more than one CPU core. Thus, a CPU thread set that allocates a variable number of CPU core is created; one can choose the number of threads to create for each thread set (labeled with a “x” in Figure 3). The CPU thread set holds a loop that executes the k kernels sequentially. For the GPU side, 1 GPU thread is created by the thread set, which allocates one GPU. Creating more than one MPI task then allows one to address a multi-GPU system.

3) Updating CPUs and GPUs memory: To deal with the extra data transfers induced by memory updates, we present the two strategies (called $S1$ and $S2$) depicted in Figure 3. Let us consider that the kernel_2_1...n succession modifies an array of t bytes. In the $S1$ strategy (Figure 3a), when both CPU and GPU threads finish their computations, a data transfer of t bytes is performed from the GPU memory to the CPU memory by the merge function. The main memory now contains both GPU and CPU contributions. A merge between these two contributions must be performed; it consists of a sum of the two contributions. Finally, the merged data (t bytes) must be transferred to the GPU memory. After these two transfers, both GPU memory and CPU memory have up-to-date data, so part 3 can be executed.

The implementation of this strategy is very simple but the necessity to transfer $2t$ bytes is a major drawback. To reduce the cost of data transfers, we thus present the $S2$ strategy based on overlapping between computations and data transfers (Figure 3b). We have implemented a function f_ker2_{update} that provides, for each kernel_2_k call, a buffer in which there is exactly the data updated by this kernel call. f_ker2_{update} selects non contiguous data and packs them in a buffer. This function selects only the data not modified by the next kernels to be called, thus, we can transfer data and simultaneously compute the next kernels: issues such as read-after-write are totally avoided. After each kernel, the GPU thread sends the updated memory (by a call to f_ker2_{update}) to the CPU memory (transfer of t/n bytes on average). Simultaneously, the second kernel can start its execution: kernel_2_k transfer and kernel_2_k+1 computation are overlapped. This overlapping is automatically managed by SGPU 2 without any intervention from the programmer. Symmetrically, on the CPU side, after each kernel execution, f_ker2_{update} is called and the buffer is sent to the GPU memory, resulting in t/n bytes for each transfer on average. Consequently, the merge function f_ker2_{merge} does not perform any data transfer: it only merges the GPU and CPU contributions. With this approach, we have n data transfers but the total size of these transfers is equal to the size of one transfer presented in the first strategy ($n.\frac{t}{n} = t$ bytes). Thus, the total amount of data transferred is reduced by a factor of two.



(a) $S1$ strategy: two data transfers are performed at the end of the CPU and GPU computations. The merge function is simple but large data transfers are necessary.



(b) $S2$ strategy: each thread performs a partial data transfer to reduce the total amount of transfers and to overlap it with computations. The merge function is then less straightforward.

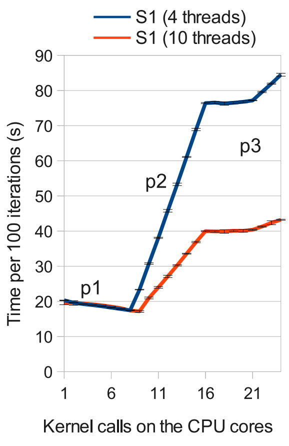
Figure 3: Two possible strategies for performing memory updates.

There are several advantages to using this approach: overlapping of the transfers and getting smaller total size. However, f_ker2_{merge} and f_{update} are non-trivial functions to write. On the GPU, reading non contiguous data is not optimal. On the CPU, data manipulations can lead to many cache misses. Thus, execution time of these functions may take a long time to process and reduce the gain obtained by the memory transfers optimization.

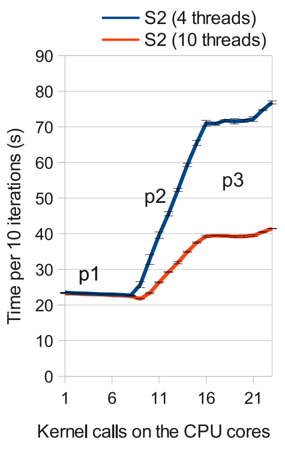
We have checked that the $S1$ and $S2$ strategies produce correct numerical results by comparing them to the output of the original pure MPI version of the SPECFEM3D code. As expected, the combination of CPU and GPU computations does not affect the numerical results.

V. Performance Evaluation

In this section, we want to evaluate the advantage of hybrid computations, i.e. simultaneously using the computing power of CPU cores and GPUs for computations. To do this, we perform several experiments consisting of varying the amount of calculations executed by the CPU cores and by the GPUs. After describing the experiments performed, we present the performance level of the $S1$ and $S2$ strategies and analyze the weight of data transfers and



(a) Performance of $S1$ strategy.



(b) Performance of $S2$ strategy.

Figure 4: Performance level of $S1$ and $S2$ strategies for 5 and 10 threads when the number of kernels processed by the CPU increases.

merge operations. We complete this section by a scalability comparison between our two hybrid implementations and the original SPECFEM3D pure GPU code on a hybrid cluster described below.

A. *Experimental context*

Experiments are carried out on a hybrid cluster consisting of 4 nodes with 24 GB of memory and two Intel Xeon X5650 processors operating at 2.5 GHz. Each processor has 6 cores and each node is connected to NVIDIA TESLA T10 GPUs with 4 GB of memory each. Nodes are connected by an Infiniband DDR network.

In this section, we use a mesh for SPECFEM3D that can be computed with 2, 4 or 8 MPI tasks.

B. *Description of the experiments*

We vary the distribution of kernel_2 execution between the CPU cores and the GPU. In our system, we have a total of 25 kernel_2 calls, so we first launch one call on the CPU cores and 24 on the GPU, then two on CPU cores and 23 on the GPU and so on until reaching 24 on the CPU cores and one on the GPU. To evaluate the influence of the CPU computing power, we first run the experiments on four CPU cores (i.e. creating two threads per MPI task) and then on ten CPU cores (i.e. creating five threads per MPI task). The same experiments are performed for the $S1$ and $S2$ strategies. For these experiments, we use only one node. Because there are two NUMA (Non Uniform Memory Access) banks per node on our machine, we have chosen to launch two MPI tasks, each of them bound to one NUMA bank. Consequently, the created threads process local data only and we avoid remote data transfers from one NUMA bank to another. Each thread is bound to one CPU core.

C. *Performance of the $S1$ and $S2$ strategies*

Results are depicted in Figure 4; the small error bars show a very stable execution time. Values in the X axis

	$S1$	$S2$
Transfers (s)	5.23	2.85
Merge operations (s)	0.65	6.52

Table I: Memory transfers and merge operations costs for the best $S1$ and $S2$ configurations (10 threads, 9 kernels on the CPU).

correspond to the number of kernel_2 executions performed on the CPU cores. For each CPU/GPU distribution the time to perform 100 time steps is shown on the Y axis. In Figures 4a and 4b, both curves represent the execution times when two different numbers of CPU cores are used. When ten threads are used the CPU computing power is the highest. With four threads, the computing power is reduced.

The different curves have basically three phases:

Phase 1 (p1) GPU computations are dominant and take more time than CPU computations. Thus, switching kernel calls from GPU to CPU decreases the execution time.

Phase 2 (p2) is a transition phase, CPU computations begin to take more time than GPU computations. Execution time increases dramatically.

Phase 3 (p3) CPU computations are now dominant and conversely to phase 1, switching kernel calls from GPU to CPU increases the execution time.

Because the GPU computations are dominant in phase 1, the same execution time is achieved with ten or four threads. Nevertheless, with ten threads, phase 1 stops when nine kernels are performed on the CPU while it stops for eight kernels with four threads. Adding CPU power increases phase 1 and thus increases global performance for both strategies. As expected, because phase 2 and 3 are dominated by CPU computations, the execution time is smaller when a large number of threads is used for the CPU computations.

D. Memory transfers and merge operations

The values represented in Figures 4a and 4b include the data transfers and merge operations. A detailed analysis of the execution of $S1$ and $S2$ allows us to compute the cost of the data transfers and merge operations.

Table I shows the cost of data transfers and merge operations for the best $S1$ and $S2$ configurations, i.e. with nine kernels calls on the CPU and ten threads. For $S1$, the large amount of data transferred and the simple code for the merge operation lead to a data transfer time considerably higher than for the merge time. For $S2$, the many small data transfers designed to overlap memory updates allow one to reduce the cost of memory transfers by a factor of 1.8 compared to $S1$. Nevertheless, the complexity of the code designed to merge the GPU and CPU data very significantly increases the merge time.

E. Scalability on a hybrid cluster

We now increase the number of nodes in order to analyze the scalability of the SGPU 2 port of SPECFEM3D. We use a fixed problem size per MPI task (weak scalability). In each node we launch two MPI tasks managed by

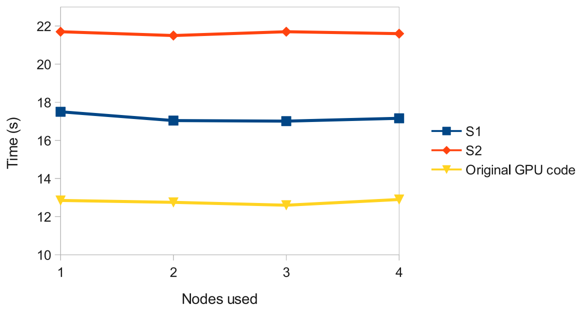


Figure 5: Scalability of the original SPECFEM3D code and S1 and S2 strategies of SGPU 2. On each node, there are two MPI tasks.

SGPU 2. In Figure 5 we show only the best performance levels achieved for both $S1$ and $S2$ (i.e. with nine kernel executions on the CPU and ten OpenMP threads). Scalability is very good, as for the original SPECFEM3D code. Although $S2$ reduces the cost of data transfers, the cost of the merge operations leads to a total execution time longer than $S1$.

F. Comparison of both strategies

Strategy $S1$ is a relatively non-intrusive solution: source code modification is limited and consists of two data transfers and basic operations on arrays. Nevertheless, the two large data transfers performed after each iteration are costly. We have thus suggested another strategy, $S2$, that requires more extensive changes in the code, with significant algorithm modifications.

Strategies $S1$ and $S2$ depict the two main approaches to handle memory updates in SGPU 2. We can classify these approaches into two categories:

- The first category contains solutions to minimize the amount of extra code to be added in applications. Consequently, operations designed to select the updated data to transfer and operations designed to merge both accelerator and CPU contributions should be very simple. In a complex code, these solutions can imply selecting a very large memory zone, and thus transferring unmodified data. Strategy $S1$ belongs to this category and implies transferring entire arrays while only a part of these arrays has been updated.
- The second category comprises solutions designed to reduce data transfers; partial data updated by a computation must be found, transferred and merged. This implies adding extra code to the application in order to select, transfer and merge an amount of memory closest to that modified by the application. The additional code can take a long time to be processed and is more intrusive in applications. Strategy $S2$ belongs to this category and a relatively complex code must be written in order to select data to transfer and to merge them.

Choosing a strategy from the first or the second category mainly depends of the application in which SGPU 2 is used; the size of data updated and transferred as well as the complexity of the code to add to deal with memory updates must be considered.

VI. Conclusions and future work

We have presented SGPU 2, a runtime system designed to deal with hybrid architectures. A programming model based on threads allows for a computation to be processed on different computing resources (CPU cores and accelerators). With thread notions for both CPU cores and accelerators, we can consider multi-core and multi-accelerator systems. A history-based time estimator is connected to the SGPU 2 scheduler to balance the load between CPU cores and accelerators. We have described how a large existing scientific simulation code called SPECFEM3D can be ported to SGPU 2 and we have suggested categorizing the solutions to deal with memory updates.

Our experiments show that the programming model provided by SGPU 2 is well adapted to an existing MPI based scientific application. Furthermore, even with an application highly optimized for the GPUs, using both CPU and accelerator computing power has an impact on the performance level and one can observe an improvement of its total execution time. Nevertheless, operations designed to update data in the different memories reduce final performance in our tests.

Novel hardware architectures, such as AMD Fusion, in which accelerator and CPU memory are shared can help to reduce the cost of memory update operations. Thus we plan to evaluate this kind of hardware with SGPU 2 in future work.

So far we have experimented with the SGPU 2 approach only in the context of the SPECFEM3D code. We thus plan to use it for other applications, for instance BIGDFT [9].

Acknowledgments

This work has been supported in part by the French National Agency for Research through the ProHMPT project (ANR-08-COSI-013). We thank Pieyre Le Loher for help in implementing SGPU 2 in the SPECFEM3D code.

References

[1] *IBM SDK for Multicore Acceleration v.3.1*, IBM, 2008.

[2] *NVIDIA CUDA Programming Guide 4.0*, NVIDIA, 2011. [Online]. Available: http://www.nvidia.com/object/cuda_home.html

[3] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: a hybrid multicore parallel programming environment," in *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[4] *PGI Fortran & C Accelerator Programming Model ver. 1.3*, The Portland Group, 2010.

[5] *The OpenCL Specification, version 1.1*, Khronos OpenCL Working Group Std., 2011. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877

[7] G. Diamos and S. Yalamanchili, "Speculative execution on Multi-GPU systems," in *24th IEEE International Parallel & Distributed Processing Symposium*, Atlanta, Georgia, USA, 4 2010.

[8] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 45–55. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669121

[9] L. Genovese, A. Neelov, S. Goedecker, T. Deutsch, S. Ghasemi, A. Willand, D. Caliste, O. Zilberberg, M. Rayson, A. Bergman, and R. Schneider, "Daubechies wavelets as a basis set for density functional pseudopotential calculations," *Journal of Chemical Physics*, vol. 129, 2008. [Online]. Available: http://link.aip.org/link/?JCP/129/014109/1

[10] L. Genovese, M. Ospici, T. Deutsch, J.-F. Méhaut, A. Neelov, and S. Goedecker, "Density Functional Theory Calculation on many-cores Hybrid CPU-GPU architectures," *Journal of Chemical Physics*, vol. 131, no. 3, p. 034103, jul 2009.

[11] J. Tromp, D. Komatitsch, and Q. Liu, "Spectral-element and adjoint methods in seismology," *Communications in Computational Physics*, vol. 3, no. 1, pp. 1–32, 2008.

[12] S. Filippone, D. Barbieri, and V. Cardellini, "Generalized GEMM kernels on GPGPUs: experiments and applications," in *PARCO*, 2009.

[13] R. Wayth and L. Greenhill, "A GPU-based real-time software correlation system for the murchison widefield array prototype," in *Publications of the Astronomical Society of the Pacific*, 2009.

[14] C. Kartsaklis, I. Todorov, and W. Smith, "DL POLY 3: Hybrid CUDA/OpenMP porting of the non-bonded force-field for two-body systems," in *Symposium on Chemical Computations on GPGPUs, 240th ACS National Meeting and Exposition*, 2010.

[15] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 451–460. [Online]. Available: http://doi.acm.org/10.1145/1815961.1816021

[16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3rd ed. New York, NY, USA: Cambridge University Press, 2007.

[17] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa, "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster," *Journal of Computational Physics*, vol. 229, no. 20, pp. 7692–7714, 2010.