C. R. Mecanique 339 (2011) 78–89

Contents lists available at ScienceDirect

Comptes Rendus Mecanique

www.sciencedirect.com

High Performance Computing / Le Calcul Intensif

Introduction to GPGPU, a hardware and software background

Introduction au GPGPU, aspects matériels et logiciels

Guillaume Colin de Verdière

CEA, DAM, DIF, 91297 Arpajon, France

ARTICLE INFO

Article history:
Available online 30 December 2010

Keywords:
Computer science
GPGPU
Computer hardware
Programming languages

Mots-clés :
Informatique, algorithmique
GPGPU
Matériel
Logiciels

ABSTRACT

This article gives an introduction to GPU usage for High Performance Computing. After setting the context, we will describe the hardware and the programming languages currently available to programmers. From these explanations we will touch on the implications of these technologies for simulation codes and try to give trends for the future.

RÉSUMÉ

Cet article est une introduction au calcul sur GPU pour le monde du Calcul Hautes Performances. Après avoir resitué le contexte, nous décrirons les matériels et les logiciels à la disposition des programmeurs. Sur la base de nos explications, nous esquisserons les implications de ces technologies sur les codes de calcul. Enfin, nous essayerons de dégager des tendances pour le futur.

1. Context

During the past decade, industry and the academic worlds have seen an increasing influence of numerical simulations, due to a number of economical and technical reasons. Through numerical simulation, a company is able to do virtual prototyping rather than actually building expensive prototypes, therefore saving significant amounts in the development process. Scientists are able to reproduce *in silico* phenomena which are not within experimental reach. It is also an opportunity to investigate some fields in greater depth, such as material properties, drug design and so on. The more complex the computation is in terms of simulated phenomena, the finer the model, and more powerful machines are needed. This is why we have seen a constant increase in computer sizes, very well depicted by the TOP500[1] list's evolution. This trend is not ebbing as users are acknowledging the real benefits of ever finer numerical simulations. Recent projections of future computations forecast the emergence of exaflop[2] machines for the 2018 time frame. Yet the race to computing power has a significant consequence: the electrical power consumption of such large installations is also increasing. If the technology does not change, we shall discover soon that no one will be able to sustain the financial impacts, as well as the infrastructure consequences, of such energy hungry computers. Hundreds of megawatts are quoted today if one has to build an exaflop computer with current technology.

E-mail address: guillaume.colin-de-verdiere@cea.fr.

[1] See www.top500.org.

[2] Teraflops = 10^{12}, petaflops = 10^{15}, exaflops = 10^{18} floating point operations per second.

1631-0721/$ – see front matter
doi:10.1016/j.crme.2010.11.003

Table 1
Intel Nehalem-EX.

8 cores,[a] 2.266 GHz, 130 W, 72.5 GFlops DP.[b]

[a] A core is the simplest processing unit present on a die. It contains all the necessary logic to perform memory accesses through caches, computations both on integer and floating point values and branches (conditional and subroutine). Usually, in a multi core processor, the cores share one or several cache levels.

[b] A floating point value is either represented as a single precision (SP) entity (32 bits) or double precision (DP, 64 bits). More bits allow for a larger exponent range as well as a wider mantissa in the floating point number representation.

Table 2
NVIDIA Fermi C2050.

Attached to the PCI-Express 16X Gen 2, 448 cores, 1.15 GHz, GPU memory: 3 GB, 1.03 TFlops SP, 515 GFlops DP, ECC, 225 W.

A more reasonable goal, for 2018, is to try to keep the power consumption under the 80 MW range, with technology changes. The spectrum of possible optimizations is wide: it ranges from new cooling methods to new classes of interconnects or networks, new memory systems, new disk technology, new microprocessors or the use of accelerators such as graphic cards.

In this article, we will focus solely on the last item: the use of graphic cards (GPU[3]) in the field of numerical simulation, coined as GPGPU.[4] Many experiments have demonstrated that GPGPU can be a path to solve the exaflop conundrum. At first, we will describe the hardware, then, we will give an overview of the programming languages followed by a description of the challenges for applications. At last, we shall try to project ourselves in the future of GPGPU and its associated challenges yet to come.

2. Available hardware

The "gamer" community has always been the main customer of GPU designers. If many vendors were available some years ago, as of today, only three main actors are designing and producing GPUs. This is the result of the ever increasing costs of research and development required to meet gamers' expectations. The latest generations of computer games are so complex that they require highly programmable GPUs. It was then realized that a GPU could be devoted to non-graphic tasks, for it was able to deliver a substantial amount of computing power. With the addition of the proper programming language, a GPU can be used for numerical simulations, as has been demonstrated in numerous articles and conferences (for example see the proceedings of GTC 2009 in www.nvidia.com/object/gtc_2009_archive.html).

The three main GPU manufacturers are NVIDIA, AMD[5] and INTEL. The following sections will cover each vendor's latest offering in terms of GPUs. To somewhat ease the comparison, Table 1 summarizes the features of a standard processor, here the Intel Nehalem-EX.

2.1. NVIDIA

The latest architecture from NVIDIA, a 3 billion transistor chip named Fermi, has been specifically designed to address the GPGPU market along with the more classical graphic segment (both gamers and high end professionals using graphic hungry packages such as Catia from Dassault Systèmes). One of the noteworthy advances of the Fermi design is the appearance of a L2 cache, whose main purpose is to make access to memory faster.

Fermi's main force, for the scientific community, lies in its speed in double precision[6] as well as a real support of ECC[7] (Table 2). NVIDIA also provides the CUDA programming environment which is easy to master and well suited to make the most of Fermi's architecture. OpenCL is also available, however, not with CUDA's maturity in terms of performance. CUDA and OpenCL are described in Section 3 on Programming Languages.

A NVIDIA GPU implements the SIMT paradigm. SIMT stands for Single Instruction Multiple Threads which means that multiple threads will execute the same instruction in a synchronous manner. This implies that the program must (be adapted to) fit to this model to fully exploit the hardware.

[3] Graphic Processing Unit.

[4] General Purpose computing on GPU.

[5] AMD has acquired ATI, a company dedicated to designing and building GPUs. Recently, AMD announced that the name ATI will disappear. Nonetheless, users are more familiar with the brand name of ATI for GPUs so we will keep it in this document.

[6] In the Fermi architecture, a core computes on single precision floats and cooperates with its neighbor to perform computations on double precision floats. Therefore the performance ratio between SP and DP peak performance is only around 2. All cores are synchronous, implementing the SIMT paradigm.

[7] ECC: Error Correcting Code. Additional bits and special functions are used by the hardware to detect and possibly correct errors which might appear in the memory sub system. The causes of errors are various, ranging from defective memory cells to incoming cosmic rays. A computer should be ECC protected to guarantee the correctness of a critical computation, provided that the program is correct (which is another debate).

Table 3
ATI Cypress HD5870. On this architecture, DP operations are slower by a factor of 5 than SP ones. Nonetheless, ATI designs deliver impressive SP performances far ahead of current CPU capabilities for the same amount of watts.

Attached to the PCI-Express 16X gen 2, 1880 cores, 2.72 GFlops SP, 544 GFlops DP, GPU memory: 2 GB, No ECC, 150 W.

Table 4
INTEL Larrabee. Exact numbers for performances and consumption vary greatly according to sources. Larrabee is a very promising design which should be analyzed in depth as soon as the Knights chips will be available for testing. A notable difference with NVIDIA and ATI is that a compute element of Larrabee relies on the x86 instruction set which makes it easy to integrate its usage in a compiler chain. There is also a vector unit on each processing elements which should make for the performance of the chip. Last, the number of processing elements is far below what ATI and NVIDIA offer.

Attached to the PCI-Express 16X gen 2, From 32 to 64 cores (x86), 2 TFlops SP (32 cores), 1 TFlops DP (tbc), No ECC, 125 W (tbc).

Threads are executed by the cores. On a 512 core Fermi, at least 512 threads are necessary to use all the available hardware. It implies that the minimum number of tasks required to solve a problem is 512. To hide latencies, 2 (if not 4) threads per core should be ready to run. Therefore, at least 2048 threads (tasks) should be running at any given point of time. This large number of tasks should be compared to the classical 64 to 512 MPI tasks seen in legacy codes. Thus, a GPU requires a very high level of fine grain parallelism in applications.

2.2. ATI

AMD/ATI was the latest to embrace the GPGPU cause, probably because the AMD company offers also efficient x86 solutions and that GPGPU has not been until now a mass market solution. However, they also have impressive GPU designs which are well appreciated by gamers. As a straggler, they have not had time to make ECC available to the scientific community, claiming that the reliability of their hardware does not make ECC a top priority. Nonetheless, they should have this feature, required by the scientific community in the next generations of GPU. As shown in Table 3, ATI is able to pack a lot of compute power in their chips.

ATI GPUs can be programmed by the OpenCL language. The software stack provided by AMD is young but maturing at a good pace.

ATI took a different approach than NVIDIA in their core design, heritage of the graphic era, favoring a vector view of data. This is why OpenCL program should make use of vector (float4 or double2) elements to get the maximum performance out of an HD5870 graphic card. As a consequence, algorithms should be thought both in terms of many threads (SIMD[8]) and vector elements, explaining why it will not be obvious to have the same program performing well on NVIDIA and ATI GPUs. Still, our remarks on the number of running threads hold for ATI GPUs too.

2.3. INTEL

INTEL is both a young and an old player in the GPU field. The company delivers millions of IGPs (Integrated Graphic Processor[9]) yearly, yet most of them are disabled in favor of a real GPU made either by NVIDIA or ATI. With Larrabee (see Table 4), INTEL tried to enter the full-fledge GPU world, but did not manage to transform it into a commercial product, although it allowed the company to make significant progress in this kind of design. The project following Larrabee[10] is Knights Corner, the first implementation of Intel's Many Integrated Core architecture chips (MIC). This new project will focus on high performance computing (HPC) and no longer on pure graphic capabilities. MIC will pave the way towards more advanced CPU designs (as said in www.intel.com/pressroom/archive/releases/2010/20100531comp.htm).

It will be possible to program the Knights Corner through INTEL's compilers using his usual language (C/C++ or FORTRAN). Even if OpenCL is in the radar scope of Intel, its implementation by this vendor is yet to be seen.

2.4. Synthesis

All three vendors provide (or will soon provide) a PCI-Express card which is able to compute in double precision. The ratio of single precision performance to double precision varies dramatically according to the design. ECC, a required feature

[8] SIMD: Single Instruction Multiple Data.

[9] An IGP offers only basic graphic performances and is well suited for non-demanding laptops. It is a good option to save battery life. IGPs are not easily programmable for scientific purposes.

[10] Larrabee is now called Knights Ferry.

Table 5
Reduction algorithm in C. This very basic algorithm is heavily used in scientific codes. Here the reduction is performed using an addition. Other operators can be useful. Another example is the max operator which is used to find the global time step of hydrodynamics codes ($dt = \frac{dx}{\max_{cells}(CFL)}$). This algorithm is not parallel and needs to be adapted to meet the GPUs particularities.

```
1  sum = 0.0;
2  for (I = 0; I < N; I++) {
3      sum += array[I];
4  }
```

for a data center, is considered important by NVIDIA and delayed for future implementations by both INTEL and AMD. The amount of device memory is small compared to the central memory of the machine hosting the GPU. The data transfer rate on the PCI-Express is low (8 GB/s peak) and will be a bottleneck in many applications. All solutions are programmable, yet portability across platforms is not important to the vendors.

The most important feature of all those GPUs is the impressive peak performance for a low electricity budget if compared to a regular CPU.[11] To reach such a performance level, programs will have to saturate the functional units. This will be achieved by having a huge number of threads ready to execute, larger than the number of functional units available, allowing for latency masking. Performance is also reached by increasing the number of floating point operations compared to the number of memory accesses. Experiments show that a ratio of 2 operations per memory access is a minimum to get the most out of a GPU.

3. Programming languages

When the first programmable GPUs appeared, many languages were offered to the users. Most were either assembly language oriented (CTM, CAL/IL) or coming from the graphic world (Cg, OpenGL ARB extensions) just to mention a few of them. All in all, they were useless for the broad mass of scientific programmers since they were not meant for scientific programming.

NVIDIA was the first able to offer a real solution, easy to understand and program: CUDA. Then OpenCL came as a response to a proprietary solution. In the mean time, directive based languages were investigated. In order to help the programmer of large simulation codes, tools are needed, but they started appearing only after the programming languages were mature enough. In this section, we will try to analyze each of those four items. As an illustration of the impact of GPU usage, we will use the same small example throughout this section: the “reduction” which is typically non-parallel algorithm (see Table 5).

3.1. CUDA

CUDA stands for Compute Unified Device Architecture. Created by NVIDIA to allow for more general programming of its GPUs, it exists in a free version for the C/C++ language or as a commercial product from PGI for FORTRAN. It is a very mature solution which allows for production of quality code. The existence of CUDA explains why GPGPU has so many enthusiasts now: the language is easy to learn, being similar to classical C (the CUDA C designers were careful to limit the number of extensions to the language). CUDA has also been adapted to scripting languages (for example, PyCUDA is a Python interface to CUDA) thus enlarging the community of potential users.

CUDA relies on a memory model and an execution model which differ significantly from the CPU side. Therefore, when programming in CUDA, a developer cannot immediately reuse all programs. Work has to be done to match the constraints depicted in Fig. 1. A shift in the programmer’s habits has to occur: the problem must now be decomposed in thousands of threads (if not millions) to ensure the full utilization of the GPU and the data placement (as well as movements) carefully planned to limit the usage of slow paths (see Fig. 3).

Programming efficiently in CUDA requires a good understanding of the hardware, as is also illustrated in Table 6. This hinders the portability of a code across many NVIDIA GPUs: if the hardware evolves, then constants (unrolling factors, threads numbers, warp size, …) have to be evaluated again.

Some of the burden is alleviated if the programmer is used to implementing most of his algorithms using libraries (such as BLAS or FFT). In this case, CUDA comes along with highly tuned libraries (e.g. CUBLAS, CUFFT) matching one to one with the CPU version. Having access to those special versions makes for an easy integration of the GPU power in the programs.

[11] Comparing a CPU to a GPU can be quite challenging: should we take all CPU cores into account (implying that the code is fully parallel even on a CPU – which is a pretty strong statement) or should we compare a single CPU core to the GPU thus wasting quite a bit of potential performance of the CPU side. A comparison metric has not been yet defined and numbers are interpreted by each party in order to make their solutions look better than the others.

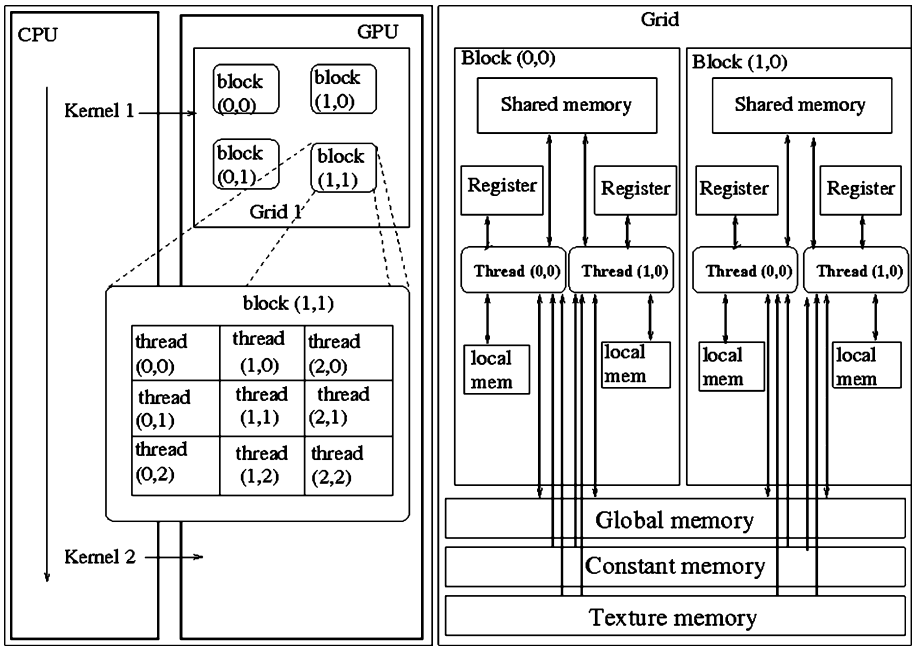

Fig. 1. CUDA execution and memory models. The left side (execution model) explains that the problem must be mapped onto a grid of blocks which are themselves made of a grid of threads. A block is handled by a symmetric multiprocessor (SM) of the Fermi GPU and the threads are executed on the cores of this SM. The right side shows the hierarchy of memory available in the GPU. A block has a private and blazingly fast memory shown in the block $(1, 1)$ box. This memory is not visible from others blocks. On the other hand, the global memory is available to all blocks. This implies that the programmer has to be very careful about the problem decomposition to adapt it to the grid view of blocks and threads. He must also check the data placement to insure the right visibility of his objects. Finally, he has to take into account the performance of the resources in action e.g. shared memory is two orders of magnitude faster to access than global memory.

3.2. OpenCL

OpenCL is a new language promoted by the Khronos Group,[12] a consortium of many vendors, some of which are major actors either in the GPU field (NVIDIA or ATI), in the consumer world (Apple), or in the HPC market (IBM, INTEL). Its goal is to provide a seamless view of future parallel machines (a single node), yet at a somewhat low level (Fig. 2). All partners claim their interest in pushing OpenCL, even if some do not have a clear politic on the subject; it is clearly the case of INTEL to whom OpenCL is a challenger to its home grown compiler centric solutions.

OpenCL can be programmed using the C/C++ programming language (as for CUDA, PyOpenCL exists). This is a problem for the FORTRAN community who cannot directly use OpenCL (as of now). OpenCL borrows its memory and execution models from CUDA with only minimal variations (see Table 7 for a version of the reduction in OpenCL). OpenCL is more flexible than CUDA since the CPU is handled as well as the GPU. This means that a programmer can easily switch from CUDA to OpenCL to reach portability. Maintaining the same performance level is not guaranteed though, for programs have to use idiosyncrasies of each GPU to get the maximum performance. However, this portability has already been demonstrated by running the very same openCL program on ATI and NVIDIA cards.

OpenCL is rather verbose and cumbersome to program. Therefore, it is a great tool for automatically produced programs, such as in the directive based languages that we will describe now.

3.3. Directive based languages

Rather than inventing yet another language, or language variation, PGI[13] and CAPS Entreprise,[14] along with several academic research projects, have taken the path of inserting directives in the existing codes so that the compiler can produce adequate code to use the GPU. (See Table 9.) The actual language of the GPU is hidden in the compiler backend. It is the responsibility of the compiler to match the produced code to the requirements of the GPU. Since the compiler has a global view of the program (usually a subroutine) it can implement many tricks to boost performances, many of which would be rather cumbersome if programmed by hand by the end user. Most of the time, the user needs a minimal (but not null) understanding of the GPU characteristics.

[12] The Khronos Group (www.khronos.org) is also in charge of OpenGL, a 3D graphic library heavily used in professional packages.
[13] www.pgroup.com.
[14] www.caps-entreprise.com.

Table 6
Reduction algorithm in CUDA. This implementation is discussed in details in www.developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf. Here, a detailed knowledge of the underlying hardware is required to get performances: loops have been unrolled and the thread execution structure exploited to limit the need of synchronization (on a NVIDIA GPU, all threads of a warp [a group of 32 threads] execute in a SIMD fashion so there is not need to synchronize them). We also see how lengthy a GPU version of a very simple program could be.

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;
    while (i < n) {
        sdata[tid] += g_idata[i] + g_idata[i+blockSize];
        i += gridSize;
    }
    __syncthreads();
    if (blockSize >= 512) {
        if (tid < 256) {
            sdata[tid] += sdata[tid + 256];
        }
        __syncthreads();
    }
    if (blockSize >= 256) {
        if (tid < 128) {
            sdata[tid] += sdata[tid + 128];
        }
        __syncthreads();
    }
    if (blockSize >= 128) {
        if (tid < 64) {
            sdata[tid] += sdata[tid + 64];
        }
        __syncthreads();
    }
    if (tid < 32) {
        if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
        if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
        if (blockSize >= 16) sdata[tid] += sdata[tid +  8];
        if (blockSize >=  8) sdata[tid] += sdata[tid +  4];
        if (blockSize >=  4) sdata[tid] += sdata[tid +  2];
        if (blockSize >=  2) sdata[tid] += sdata[tid +  1];
    }
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

This does not mean that porting a legacy code is a straightforward process using PGI Accelerator or CAPS HMPP. Some cleanup and code reorganization have to be done so that the compiler will be able to process the sections which are supposed to run on the GPU[15] nicely.

As of the writing of this article, no agreement on the directive set exists today. HMPP and Accelerator use a different approach with a different set of features offered to the user. It would be pertinent to see a standardized set being adopted, such as it has been done with OpenMP, to ensure code portability in the coming years. The PathScale company has recently announced (see www.pathscale.com/node/232) that its compiler will adopt the HMPP directives, which they describe as the most mature to date. This is a first step towards standardization. One could dream that the OpenMP committee will extend their specifications towards GPU, based on the HMPP proposal. The future will tell us what directive set will survive.

Working with directives in a compiler has many positive side effects, one being that the compiler has a view of the user's intents being exposed to a high level of semantics. Therefore, it is able to pinpoint structural problems (section of code inefficient on a GPU) or suggest code reorganization to boost the code performances (such as loop reordering as shown

[15] The programmers must avoid global variables, recursion, dynamic arrays, which are common pitfalls hindering a smooth port to a GPU.

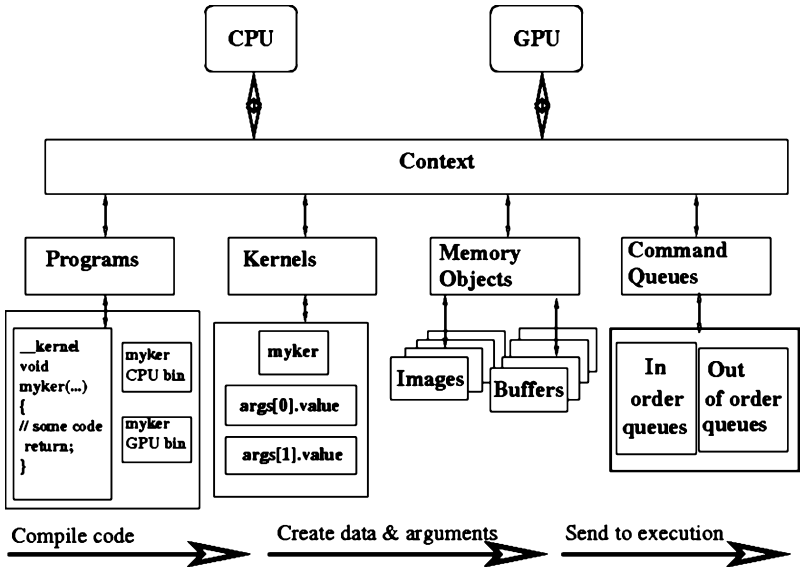

Fig. 2. OpenCL. This picture shows, in a synthetic manner, the objects manipulated by OpenCL, how to create and use them and finally the targeted hardware. Contrary to CUDA, there is no differentiation of the CPU and the GPU in the OpenCL world. This allows for new classes of programs which could dynamically adapt their work on different resources, depending on the load or the complexity of the algorithm.

in Table 8). Many well known compiler techniques can be applied, classical sequences of code can be recognized, enabling the compiler writers to suggest modifications to the user. The CAPS HMPP compiler has two experimental tools "HMPP feedback" and "HMPP wizard" whose goals are to help the programmer in his migration to the GPGPU world. It would be more challenging to have such tools using CUDA or OpenCL since part of the modifications are already done and a compiler is not in the right position to help the programmer further in his optimizations, being at too low level of an abstraction.

3.4. Development tools

When developing a large code, developers have to use a set of tools to fix the inevitable bugs or to understand why the code is not running at the expected speed. In the early days of GPGPU programming, the programmer had to rely solely on trial and error. This situation made for a reputation of difficulty to get a program right on a GPU. Fortunately those days are gone and some tools are emerging as essential to serious code development. Such tools are NVIDIA Nexus, currently available for the Windows environment, Totalview [16] or Allinea DDT[17] which have been extended to debug CUDA programs. The case of those debuggers is interesting, for a user can work with the very same tool in different situations: the program can be multithreaded on a single node (e.g. OpenMP) or based on MPI running on the largest cluster or, as we said it, using CUDA on a NVIDIA GPU.

Once the first port has been made, the running version of a program is often not optimal. To push further performance, profilers are an important set of tools. While available in the CPU world for quite a while, profilers for GPUs are still a very active field of research and development. The complexity of the hardware, as well as the execution model, leads to difficult GUI representations or to reliable descriptions of the program behavior if, for example, the results are based on a thread behavior taken randomly in a block.

3.5. Synthesis

Two years ago the situation was quite awkward: the scientist had only crude compilers to experiment with GPGPU. Today, the landscape is changing rapidly: compilers and tools are maturing and allow for in depth mastering of the hardware with ever increasing usage of the peak performance. The coming years will tell us if CUDA will survive the OpenCL pressure and if the users will rather use high level languages (HMPP like) than lower but more powerful and less portable ones.

[16] www.totalviewtech.com.
[17] www.allinea.com.

Table 7
Reduction algorithm in OpenCL. The discussion about this implementation of the reduction on a sum using OpenCL can be found at www.developer.amd.com/documentation/articles/Pages/OpenCL-Optimization-Case-Study-Simple-Reductions_6.aspx. Here we notice that, if the global algorithm is not far from the CUDA implementation, slight variations are present for the synchronization of the thread during the parallel phase. Notice also that the knowledge on the thread execution mechanism is not exposed here.

```
__kernel
void reduce(__global float* buffer,
            __local float* scratch,
            __const int length,
            __global float* result) {

  int global_index = get_global_id(0);
  float accumulator = INFINITY;
  // Loop sequentially over chunks of input vector
  while (global_index < length) {
    float element = buffer[global_index];
    accumulator = (accumulator < element)? accumulator: element;
    global_index += get_global_size(0);
  }

  // Perform parallel reduction
  int local_index = get_local_id(0);
  scratch[local_index] = accumulator;
  barrier(CLK_LOCAL_MEM_FENCE);
  for(int offset = get_local_size(0) / 2;
      offset > 0;
      offset = offset / 2) {
    if (local_index < offset) {
      float other = scratch[local_index + offset];
      float mine = scratch[local_index];
      scratch[local_index] = (mine < other) ? mine : other;
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (local_index == 0) {
    result[get_group_id(0)] = scratch[0];
  }
}
```

Table 8
This example illustrates simple code reordering using HMPP directives. The code in the "before" column is efficient on a CPU and slow on a GPU, while the generated code "after" is well suited for a GPU. Doing code reordering by hand is error prone and kills the CPU efficiency.

Before

```
!$hmppcg permute k, i, j
DO I = 1, M
 DO J = 1, N
  DO K = 1, L
   A(I, J, K) = B(I, J, K) * 1.2
  ENDDO
 ENDDO
ENDDO
```

After

```
DO K = 1, L
 DO I = 1, M
  DO J = 1, N
   A(I, J, K) = B(I, J, K) * 1.2
  ENDDO
 ENDDO
ENDDO
```

4. Simulation codes

4.1. Amdahl's law

It is impossible to discuss the pros and cons of GPGPU in simulation codes without a reminder of the limits of parallelism. Amdahl's law is too often forgotten (on purpose?) by the vendors to promote their newest piece of hardware or by the programmers who focus their efforts on unworthy sections of their codes.

The limit of the potential speedup, for a given application ported onto a parallel machine, has been stated by Gene Amdahl as early as 1967 in [1]. The law is given by the following formula $S = \frac{1}{(1-p)+\frac{p}{N}}$, where S is the measured speedup,

Table 9
The reduction algorithm is here expressed in HMPP syntax. In HMPP, the programmer just has to let the compiler know about what is the reduction variable and operator and let it automatically generate the best code for the targeted GPU. This small piece of code illustrates the benefits of working at a higher level: the user specifies his intention and needn't to worry about the implementation thus increasing portability of his code.

```
1  sum = 0.0;
2  #pragma hmppcg Parallel Reduce(+: sum)
3  for (I = 0; I < N; I++) {
4      sum += array[I];
5  }
```

Table 10
Here is a practical explanation why codes need to be deeply adapted to take benefit of GPGPU.

A small example, taken from the author's experiments shown in Table 11, illustrates that changes in codes are required. Let us consider the well known algorithm of alternate directions used in two-dimentionnal hydrodynamics Eulerian codes. The typical coding practice for such an algorithm is to iterate in the Y direction along all X values and then in the X direction along all Y values. It means that a whole line (resp. column) can be processed in parallel. Thus a small adaptation of the program runs on a GPU but at a fraction of the peak performance. One must use very large grids (e.g. $10\,000 \times 10\,000$) to see a significant speedup compared to the CPU version.
The problem of such algorithm is that, for a medium size grid (e.g. 1000×1000), the functional units are not saturated, as explained in the hardware section (the 1000×1000 test case would only use 1000 threads at a time instead of the expected 2048). To reach saturation (and thus larger speedups), the full grid should be processed in a single pass. As a consequence, each routine has to be adapted to process 2D planes instead of 1D lines (columns), modifying the memory footprint as a side effect.

p is the fraction of the code which has been parallelized and N is the number of processors used. The maximum theoretical speedup can be found for $N = \infty$ which is $S_{\max} = \frac{1}{(1-p)}$. According to this last equation, if a programmer wants a $10\times$ acceleration of his code, then 90% of the code should be strictly parallel with an infinite number of CPU. In reality, with 100 processors and 90% of the code being parallel, the speedup is at most 9.17 (and only 9.83 for 512 processors).

This law has two consequences:

1. A programmer must drastically reduce the sequential time to get performance speedups;
2. The possible speedup has a known limit, whatever are the programming skills of the developer.

The next section takes these remarks into account.

4.2. Simulation code development

Simulation code types are as varied as the subjects they are simulating. The fields covered could be as different as asset simulations, drug design, automobile crash, material properties, study of the universe expansion, photorealistic games or movies, ... In all cases, specific numerical methods have been used or invented, many (if not most) of them being sequential. Since the year 2000, a tremendous effort has been made to transform some of those codes to parallel versions, yet using only a modest number of processing units. Moving codes to the GPU is a challenge which is under heavy investigation, worldwide. While some problems are embarrassingly parallel (e.g. ray-tracing) hence somewhat easy to port to GPUs, most exhibit a high level of sequential time. It requires quite a bit of engineering time to work on the slowest part. (See Table 10.)

Tools are needed to help the transition, for programmers are facing a new level of complexity. As we have discussed, they have to think in terms of thousands, if not millions, of threads. They also have to take into account an adverse memory layout inside the GPU. On top of all this, the topology of the parallel machines increases the virtual distance between two memory locations (as depicted in Fig. 3).

Campaigns of porting code to GPU showed that:

- It is often difficult to isolate parallel sections or subroutines to port to the GPU, especially if the code has been written in a very object oriented manner and if the program is demonstrating a flat profile of execution (many routines with a small fraction of the execution task). It will often be the case with old FORTRAN codes.
- A huge work must be done to understand and master the data layout throughout the code as well as in its internal representation. Some codes have opted for arrays of structures which are well suited to describe many problem. On a

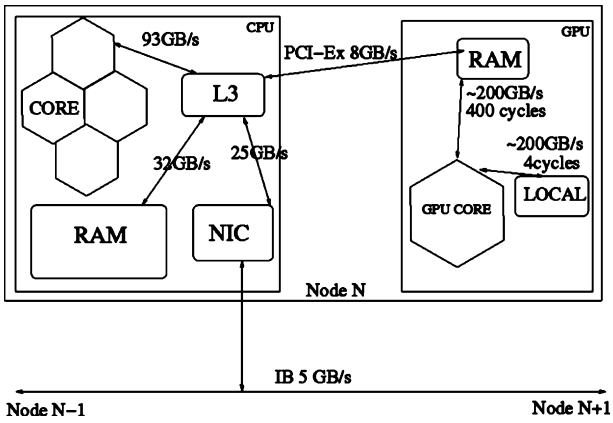

Fig. 3. This diagram illustrates the different speeds present in a hybrid machine (a cluster of multicore nodes with GPU). To move a block of data from one node's GPU memory of to the next node's GPU memory, the programmer must plan a transfer across the PCI-Express, on the network and again on the PCI-Express (in a very optimized implementation).

GPU, it will lead to sub performing memory access hence a sub optimal program (in NVIDIA terminology it's a non-coalescent access). Therefore this kind of program has to be changed to use structure of arrays (or more complex hybrid structures), allowing for linear accesses to the memory. This can have a major impact on a project in terms of man power as well as code revalidation.

- Data dependencies have to be limited, if not suppressed, to allow independent thread executions. Since compute units of a different processor group do not share the fastest memory (shared memory in CUDA, local memory in OpenCL), programs cannot make the assumption that they can have access to every memory location. Furthermore, the order of execution cannot be predicted between blocks of threads (both in CUDA and OpenCL). Therefore, it will be important to work on dependencies which would break massive parallelism.
- In the same line, indirections will lead to implementations which are difficult to parallelize because of the random access to memory location. If the algorithm could be expressed in terms of vector, the port to a GPU would be much easier.
- Explicit methods are easier to port to GPU than implicit ones, for they limit the data movements required. This is a very active field of research which, hopefully, will see some breakthrough in the near future. The good news is that linear algebra can benefit from GPU acceleration because this family of hardware is well suited for matrix operations.
- Numerical methods need to be revisited in depth (as done by Tim Warburton in [2]). In the past, floating point operations were considered as expensive compared to memory access. This is not true anymore as we have seen in a previous section. High order compute intensive methods can be considered again as they will exhibit a good compute to memory access ratio. A side effect of the relatively low cost of computation is that GPU programs should consider re-computing some values rather than storing them.

  The reader must be warned when reading speedups numbers. The quoted $100\times$ values usually apply to sections of programs and not to the complete application's run time. When comparing a sequential run to a GPU port, using a NVIDIA C1070, a simulation speedup of $10\times$ should be considered a good achievement. Nonetheless, GPGPU has already shown that interactive speeds are possible nowadays compared to when the batch mode was the only option, not too long ago. It will bring new practices to the industry: parametric studies will be possible even on the engineers' workstation; validation of concepts can become a routine procedure.

On going efforts are too many to give here a comprehensive list of current projects. Mike Giles quotes some of these,[18] noting that all the seven dwarfs presented in *Defining Software Requirements for Scientific Computing*, a 2004 presentation of P. Collela, have more a less a hybrid version. One notable work is the implementation for GPUs of the famous LAPACK library made by the Magma project (http://icl.cs.utk.edu/magma/, www.netlib.org/lapack/lawnspdf/lawn223.pdf).

Porting some applications to the GPU promoted new ways of programming. The S_GPU[19] library is one of those new approaches which allow for impressive speedups.

4.3. Performance

The performance achieved, when porting to a GPU, is vastly problem dependent (algorithm and data-set size). It is also a function of the hardware,[20] the compiler's version, the program's optimization level and so on.

[18] In https://ktn.innovateuk.org/c/document_library/get_file?p_l_id=111051&folderId=1050771&name=DLFE-10346.pdf.
[19] Described in www.teratec.eu/actu/teratec/forum_2009/S1%2010.pdf.
[20] As depicted in www-ccrt.cea.fr/fr/collaborations/fichiers/journee_thematique_110408/Matthieu_DSM.pdf.

Table 11
Accelerating a code with a GPU. The test case is the explosion of a supernovae simulated on a 9000×9000 grid with a two-dimensional hydrodynamics Eulerian code. The two first columns refers to the OpenMP version of the program, while the three last ones show the impact of the port on a GPU using different approaches (CUDA, HMPP, OpenCL). From this table we conclude that GPUs, even with a moderately aggressive port, yield better performances than a CPU. We notice also that the OpenCL compiler is less mature than CUDA (the final assembly language should, in theory, be the same on the same GPU) and that a directive-based language can yield as good results as a hand tuned CUDA code, demonstrating the value of this approach for large scientific codes.

	Intel Nehalem		NVIDIA Fermi		
	1 task	16 tasks	CUDA 3.1	HMPP 2.3.3	OpenCL
	57.704 s	25.542 s	11.406 s	11.999 s	13.404 s
Speedup	1X	2.3X	5X	4.8X	4.4X

As an illustration of the potential benefits of GPUS, Table 11 gives the observed results of a moderate reorganization of a two-dimensional hydrodynamics Eulerian code for hybrid computing (as explained in Table 10). The goal of this work was to demonstrate if the execution time could be decreased using a GPU and to measure the development costs of moving to different languages. Most of the aforementioned remarks of the previous section were confirmed by this work.

4.4. Synthesis

Moving a significant percentage of an old numerical simulation code to GPU will require a considerable amount of rewriting (at least for the sake of Amdahl's law). It will also put much pressure on the tool developers. At the same time, work on numerical schemes and algorithms will be required to better fit the hardware. Without such investments, it will be impossible to benefit from the potential of GPUs.

New codes have already demonstrated the immense potential of those powerful extensions of today's computers.

5. Perspectives

There should be an increase in GPU usage, both on the hardware and the software side in the coming years.

All vendors new GPUs of will be ever more powerful in terms of FLOPS. The double precision performance will be closer to the single precision performance, following the NVIDIA trend. Memory will be larger and ECC will be available in all chips. Programming will be easier and more flexible (C++ support in Fermi is a good example of this trend). This is the natural evolution of graphic cards we can observe right now.

A deeper evolution can also be predicted: the GPU will be closer to the CPU. This is no new trend. In the past, the 8086 processor had an 8087 companion to perform floating point computation. It was very inconvenient to program floating point operations. Eventually the 8087 merged into the 8086 to give the CPU architecture we know today. At the same time, the compiler took charge of the floating point calls, relieving the user of a burden. On the same token, even if separate graphic cards are used today, AMD Fusion[21] shows that the CPU–GPU merger will occur. NVIDIA tries this also with its Tegra,[22] Intel might do it starting with the Sandy Bridge[23] processor and beyond the Knights Corner project.

At the same time CPU performances will (have to) increase since the sequential parts of codes must be accelerated as well.

On the software side, it is believed that a convergence of views will arise on OpenCL along with more integrated solution relying on higher level descriptions such as CAPS HMPP, PGI or what is offered by Intel compilers. The code generation strategy will be less "users assisted" (i.e. be more automatic) as compilers writers get more familiar with various hybrid architectures. This will go along with an active development of highly optimized libraries, especially tailored for hybrid architectures.

As far as simulation codes are concerned, a massive re-writing will be contemplated for those which will have to run on the largest machines. Migrating legacy codes to new architectures will be costly in terms of manpower required. To prepare this phase, studies on algorithms and numerical schemes, focused on hybrid architectures, should be actively promoted. This evolution should be seen as an opportunity to imagine new ways to do computations.

[21] Fusion, the new design from AMD, will bring the GPU very close to the CPU (here against its cores). While the full details of the chip are not public, the Llano processor will provide a HyperTransport link from the CPU and the GPU to the main memory system, thus hopefully removing one of the bottlenecks present with a real graphic card using the PCI-Express. The Llano APU (Accelerated Processing Unit), as it is called by AMD, is not intended for HPC but for the laptop market. A future version will be more HPC oriented.

[22] In Tegra, NVIDIA is using an ARM design to provide a single chip (SOC – System On Chip) featuring both a CPU and a GPU. The market targeted is the cell phone. One can easily imagine an extension to this design more oriented towards servers and programmable in CUDA or OpenCL, provided DP and ECC are available. This also shows that HPC has a lot to learn from the embedded world.

[23] The compute capability of the GPU, present on the desktop version, is unknown yet but this design paves the way to more compact solutions. The challenge for Intel will be to pack more features and performances in an energy efficient design.

6. Conclusion

As a summary, using GPU (and its follow-on designs) cannot be avoided, pushed both by users' needs and the evolution of vendors' designs. Production machines using GPUs do exist today (see for example www-ccrt.cea.fr for the description of the Titane machine). Advanced users have already shifted from experimenting with a single GPU in a desktop machine to the usage of large hybrid clusters. Vendors have also acknowledged it and a wealth of hybrid designs and solutions are readily available.

As a conclusion your position should not be "*will I join the GPGPU trend?*" but rather be "*how soon can I get ready to master GPGPU?*".

References

[1] Gene Amdahl, Validity of the single processor approach to achieving large-scale computing capabilities, in: AFIPS Conference Proceedings, 1967.
[2] Andreas Klöckner, Tim Warburton, Jeff Bridge, Jan S. Hesthaven, Nodal discontinuous Galerkin methods on graphics processors, Journal of Computational Physics 228 (21) (November 2009) 7863–7882.