Introduction

GPUs are being adopted by the world scientific community as hardware accelerators in various domains such as signal processing, bioinformatics, molecular dynamics, magnetic resonance imaging, tomography, reverse time migration, electrostatics, weather forecasting, and so on. They are designed with a massive number of small and dedicated compute cores running simultaneously. However, as with every specialized architecture, GPUs perform very well with some codes and can be inefficient with some others.

GPU-based Systems: A first step toward future of manycore architectures

Multicore and GPUs are an answer to the frequency wall that arose during last decade. Multiplying the number of cores is a way to increase performance of hardware without increasing their frequency. In the near future, convergence between CPUs and GPUs is to be expected as recently announced by the major processor vendors: Intel Sandy Bridge, AMD Fusion, NVIDIA project Denver, etc. Making applications that exploit accelerators such as GPUs should therefore be considered as a first step toward the future of manycore architectures.

Figure 1: multi-level parallelism system architecture.
Different levels of parallelism

As shown in Figure 1, applications running on large scale GPU machines exploit at least three levels of parallelism: message passing MPI between nodes, multithreading OpenMP/pthread within the node and CPU or GPU data vectorization at lower level. About what has happened is at your fingertips. In the screenshot is an application that is using tracepoints to record arguments to all the MPI function calls that are made.

No one single programming model

However, as GPU and CPU microarchitectures are very different, all these levels of parallelism cannot be exploited using a single native programming model. Multiple APIs need to be used in the same application source code: this is called hybrid parallel programming. The most common form of hybrid programming combines MP with OpenMP. But when dealing with GPUs, other programming APIs such as CUDA, OpenCL or HMPP complement them.

In the case of heterogeneous compute nodes (Figure 1), GPUs are used in a hostdevice configuration where all GPU operations are controlled by the host CPU. As each GPU device has its own memory and a different address space than the CPU, data need to be transferred over a PCI express bus.

Porting applications to heterogeneous parallel systems is therefore a complex task requiring many engineering skills to successfully achieve a performance goal in a reasonable time frame. No tools exist to automatically convert a sequential code to a massive parallel version that exploits a large number of different cores.

A methodology with enhanced development tools

To drive the development of manycore applications, a porting methodology that associates the right tools to use at the right time is key to economically succeed. New programming models emerge and software development tools with compilers and debuggers are being enhanced to embrace massive parallel processing capabilities.

Code migration methodology

Besides having a clear view of how to migrate applications onto new manycore processors, the main objective of a methodology is to reduce risks and to improve efficiency. It is indeed not
economically viable to start a project and to realize a few months later after having spent engineering resources and money that the project cannot succeed.

The code migration process defined here is nothing more than a common sense of delimiting a development cycle in steps associated to durations. Each step indicates what tools to use and ends up with a go/no go condition before starting the next phase.

The three steps are:

1) Parallel project definition: in this step a diagnosis of the application is performed in order to evaluate potential speedup and to determine the main porting operations associated to cost. As a prerequisite, a validation process is set up to ensure the validity of the numerical result.

2) Application porting: in a few weeks, a first functional GPU version of the code is developed and a GPU execution profile is performed to identify bottlenecks in order to improve code efficiency during the following step.

3) Application optimization: bottlenecks are analyzed and code optimization is performed to get a production code finely tuned. As risks of not succeeding have been raised, this step can last longer than previous ones.

The first two steps are part of the initial phase aiming at exhibiting heterogeneous parallelism. They are performed by programmers who have an intimate knowledge of the application algorithm and computing methods. The third step in the second phase requires more skills in code tuning techniques. These steps are defined with a control of the cost in mind. As the migration process goes on, the risk of failure decreases and more manpower can be spent on the final operations. The migration methodology is really oriented towards a “best effort” approach for a given period of time.

**Step 1:**

**Parallel project definition**

The top part in Figure 3 details the steps to perform in order to analyze the code and to define the main migration operations:

- Hotspot identification: using profiling tools, this first phase aims at finding the critical hotspots that might benefit from GPU acceleration. Code rewriting might be necessary to increase data parallelism. If hotspots are not computationally intensive enough, it might be necessary to pick up additional hotspots.
• CPU optimization: CPU optimization is required to ensure that the original code can serve as a fair performance comparison base. Tuning the CPU code also usually leads to an efficient migration starting point.

• Parallelism discovery: this step determines whether the kernels can be executed in parallel. If not, the accelerator will not be able to bring performance gain. Algorithms should then be reconsidered to exhibit parallelism.

**Step 2:**
**Application porting**

The bottom box of Figure 3 gives the steps to develop and build a first functional GPU version of an application. They mainly consist in generating and calling the GPU kernels by annotating the previously identified hotspots with HMPP directives.

These steps are performed incrementally: kernels are ported and validated one by one; their performance is evaluated with regard to the original CPU performance so as to check they are suitable to GPU execution; basic code transformations as advised by the HMPP Wizard are applied to the kernel computations in order to make them GPU friendly; some data transfers are basically optimized so as to preload data before codelet execution and to suppress redundant transfers of constant data.

---

*Figure 3: parallel project definition and first application porting.*
This preliminary porting version serves to identify GPGPU issues as well as validate the parallel properties of the implementation. By tracking changes, and applying transformations incrementally, it is easier to find and correct bugs with Allinea DDT as the project progresses.

Step 3:
Application optimization

In this last step, the whole hybrid application is optimized by further reducing data transfers, by fine tuning GPU kernel performance and by moving GPU device allocation to application startup.

For instance, grouping codelets together in a HMPP group allows them to share data that is kept resident in the GPU memory. Also, HMPP performance analyzer synthesizes performance metrics based on the GPU execution profile of the application and gives code transformation advice.

Figure 4: manycore optimization steps.

**HMPP Wizard** performs static analysis of GPU codelet computations and gives developers diagnoses and advice on how to make them more GPU friendly.

**HMPP Performance Analyzer** synthesizes performance metrics based on GPU kernel execution profile and gives advice on how to improve code efficiency.
This white paper is not meant to cover all manycore programming models and tools.

In the two next parts, a specific focus is made on parallel APIs with CAPS HMPP and debugging heterogeneous HMPP code in desktops and massively parallel environments with Allinea DDT.
Manycore parallel APIs

The emergence of new massively parallel architectures also comes with new programming models. There are three different approaches: language-based, library APIs or directive-based. Existing models such as MPI for message passing between nodes or OpenMP for thread level parallelism have not been designed to exploit data stream parallelism exposed in GPUs.

There is no one single programming model that addresses all levels of parallelism in heterogeneous machines. Stream programming models such as NVIDIA CUDA and OpenCL have been introduced to exploit data parallelism available in GPUs. They are C extensions and as such do not easily integrate with Fortran applications. They enable very high performance but at the expense of portability.

On the other hand, by annotating application source code, directive-based models preserve legacy code, ease code maintenance and keep source compatibility between multicore and GPU versions of the code.

At a higher level, directive-based models abstract the programming of manycore systems. They offer an incremental way of migrating applications by first declaring and generating GPU kernels of critical computations, then by managing data transfers and finally by optimizing kernel performance and data synchronization. The compiler is in charge of detecting and extracting data parallelism in GPU kernels and minimizing data movement. Specific directives can be explicitly used to further help the compiler.

HMPP: an example of directive-based programming model

HMPP provides programmers with a simple, flexible and portable interface for developing parallel manycore applications whose most computation intensive sections are distributed at runtime over the available specialized and heterogeneous cores.

The HMPP approach is similar to OpenMP, a widely available standard, but designed to handle manycore hardware accelerators. Directives preserve legacy application source code and keep the original version executable using traditional compilers. Furthermore, if the hardware accelerator is not available for any reason, the sequential version of kernels can still be executed and the application behavior is unaffected.

Overview of HMPP directives

HMPP implements a Remote Procedure Call (RPC) of functions called codelets on GPU accelerators. A codelet takes several scalars and arrays as parameters, performs a computation on this data and returns the result in an argument passed as a reference of the codelet. The execution of a codelet is considered atomic: it does not have an identified intermediate state or data. The execution has no side effects. HMPP directives are safe meta-information added in the application source code that do not change the original code. They address the remote execution (RPC) of functions or regions of code as well as the transfers of data to and from the accelerator memory.
HMPP accelerated regions and functions

The simplest form of programming with HMPP consists of either using one single directive to declare a GPU version of a region or two directives for functions: one to declare the codelet function and another one to annotate its call site. This way all input data are uploaded in the GPU before the execution of the region/ function and the result is downloaded when the codelet has completed. Device allocation and data transfers can be dissociated from codelets’ calls using the appropriate directives. All directives are identified with a unique label indicating the codelet they are associated with. Directives of same label need to be used in the same compilation unit.

For instance, in the code example 1 below, a codelet directive with a ‘cuda_kernel’ label is inserted line 2 to declare a CUDA version of the ‘kernel’ function to be generated by HMPP. A call to this codelet is indicated with a callsite directive of same label inserted just before the call to kernel line 31.

```
1 #pragma hmpp cuda_kernel codelet, target=cuda, arg=[vout].in=[inout]
2 static void kernel(unsigned int N, unsigned int M,
3            float vout[N][M], float vin[N][M]){
4    int i, j;
5    for(i = 2; i < (N-2); i++) {
6            for(j = 2; j < (M-2); j++) {
7                float temp;
8                temp = vin[i][j]
9                 + 0.3f *(vin[i-1][j-1] + vin[i+1][j+1])
10                - 0.506f *(vin[i-2][j-2] + vin[i+2][j+2]);
11                vout[i][j] = temp * (vout[i][j]);
12            }
13        }
14    }
15 int main(int argc, char **argv){
16    unsigned int n = 100;
17    unsigned int m = 20;
18    int i, j;
19    float resultat = 0.0f;
20    float out[n][m];
21    float in[n][m];
22    ....
23    // init
24    for(i = 0 ; i < n ; i++){
25            for(j = 0 ; j < m ; j++){
26                in[i][j] = (COEFF) * (-1.0f);
27                out[i][j] = (COEFF) + (j * 0.01f) ;
28            }
29        }
30    #pragma hmpp cuda_kernel callsite
31    kernel(n,m,out,in);    
32    ....
33    printf("result : %f\n",resultat);
34    }
```

Code example 1: basic HMPP programming.
Grouping codelets with HMPP

Codelets can be part of a group executing on the same GPUs. They benefit from a single device allocation for the whole group and data can be shared between codelets without the need to send them back and forth. Parameters of two codelets of the same group can also be mapped in the same GPU memory location as shown in code example 2.

Optimizing data movement with HMPP

Many techniques can be used to optimize data transfers. The first one to consider consists in asynchronously preloading data before any call to codelets. To avoid useless transfers, data can be kept resident so that they will not be uploaded between several executions of the same codelet.

In the case of domain decomposition, partial transfers of border zones between sub-domains save time. Finally, overlapping data transfers with computation of kernels should be considered when there are no dependencies between the data that is being uploaded and the kernel that is currently executing.

Code example 2: (below)

Gouping codelets and mapping their parameters.

```c
#pragma hmpp <mygp> group, target=CUDA
#pragma hmpp <mygp> map, args[f1::inm; f2::inm]

#pragma hmpp <mygp> f1 codelet, args[outv].io=inout
static void kernel1(int sn, int sm, float inm[sn][sm], float outv[sm])
{
...
}

#pragma hmpp <mygp> f2 codelet, args[v2].io=inout
static void kernel2(int sn, int sm, float v2[sn], float inm[sn][sm])
{
...
}
```
HMPP Multi-GPU partitioning

As the number of cores keeps growing, it is essential to help developers easily distribute data and computations over multiple CPUs and GPUs. The HMPP programming model version 3.0 supports multi-GPU programming by enabling developers to either perform array distribution or spread out a collection of data on multiple devices.

In the C example below, two CUDA devices are attached to a group of codelets. The loop in the main function is indicated parallel with a clause expression defining how data and computations in the region are to be distributed between the two devices. All the directive operations in the loop inherit

HMPP Kernel optimizations

While HMPP abstracts the programming of GPU devices in CUDA, it is key to let developers access specific features of the architecture such as memory, grid of computations, etc. They can this way achieve very high performance, but at the price of portability! HMPP tuning directives enable developers to apply code transformations so as to increase data parallelism and to also add properties in order to just finely map computations on the underlying target architecture. from the distribution expression.

Code example 3: (below) distributing codelet execution and data over multiple GPUs.
Debuggers

During the porting effort there will be the possibility of introducing bugs into the software. For this reason, it is essential that a debugger is used to solve the problems quickly.

The multiple levels of parallelism in today’s systems require a debugger that is able to handle this complexity. Allinea DDT is a graphical parallel debugger – used by many scientific computing centres, universities and corporations to help in the everyday task of finding and fixing bugs, from single process workstations through to the very largest supercomputers. It has many features not present in ordinary debuggers – such as memory debugging, data visualization and support for the many MPI and OpenMP implementations that are used by parallel software developers - and it has an interface that makes debugging easy, at any scale.

Allinea DDT allows developers to debug GPU and CPU code simultaneously and to debug this scenario across multiple compute nodes. Using a debugger allows developers, for example, to examine the state of an application at the time of a crash, to “step” the program and observe how an application behaves or to stop and examine state at particular locations in the source code on every passing. This sort of capability is extremely useful for hybrid and parallel applications, where other techniques such as inserting print statements in the code are usually ineffective.

Debugging of CAPS HMPP applications is a straightforward task with Allinea DDT, on one node, or on thousands. Allinea DDT is fully compatible with the CAPS HMPP compiler and runtime libraries - allowing debugging access to codelets and CPU code.

Allinea DDT allows developers to debug GPU and CPU code simultaneously and to debug this scenario across multiple compute nodes. Using a debugger allows developers, for example, to examine the state of an application at the time of a crash, to “step” the program and observe how an application behaves or to stop and examine state at particular locations in the source code on every passing. This sort of capability is extremely useful for hybrid and parallel applications, where other techniques such as inserting print statements in the code are usually ineffective.

Debugging of CAPS HMPP applications is a straightforward task with Allinea DDT, on one node, or on thousands. Allinea DDT is fully compatible with the CAPS HMPP compiler and runtime libraries - allowing debugging access to codelets and CPU code.
Debugging inside HMPP Codelets is, at first glance, similar to debugging an ordinary CPU function. The source code is displayed, and colour highlighted to show that threads are on particular lines, as shown above. Double clicking on a line of code will cause threads to stop when that line is reached, and the usual buttons used to control processes, such as step over, play or pause, can still be used when debugging inside codelets.

The features of Allinea DDT that are used to manage and display many CPU threads and processes now help debug GPU code. Inside a codelet, many threads will be active at the same time. In the screenshot below we can see the parallel stack view - this shows the stack traces of threads (and processes), merged together to simplify dealing with large numbers of threads. A small kernel (hmppcg_loop0) with a total of 224 GPU threads active - the threads are spread over a number of lines in the codelet, which is quite normal when working with GPUs. By clicking in the stack view on one of the branches of the view (such as basic.f90:29) - DDT will select a thread that is on that line and evaluate variables in that thread’s context.

It’s easy to see variable values in the CPU code with debugging tools, and Allinea DDT is able to also show variable values inside HMPP codelets on the GPU. Large data structures, such as F90 arrays can be explored easily using Allinea DDT’s array viewing feature - which also supports filtering to allow searching for particular values in large data sets. By examining the data on the GPU device, and using the same tools to explore data on the host CPU, it is often simple to discover errors.
Conclusion

In the last decade, as processors hit a power wall, the microprocessor industry has started to deliver parallel chips, going from unicore to multicore and now manycore with about hundreds of cores per chip.

By featuring a very large number of simplicores, GPUs have already entered the manycore era and large systems now exhibit massive parallelism at different granularities: coarse grain between nodes, threads within a node and core fine grain parallelization with vectors.

Exploiting those levels of parallelism is a key challenge developers have to face in order to write efficient and scalable applications. They first need to extract sufficient parallelism from their existing algorithm and then be able to transpose it in the manycore computer productively, efficiently and correctly.

A porting methodology combining steps to migrate applications with tools to use each step increases the chance to succeed and improves productivity.

Software tools are also part of the revolution, they are evolving alongside the manycore trend by handling massive number of cores.

Directive-based approaches abstract the programming of a single core. Developers focus on the parallelization of their algorithm and how computations are spread over the numerous compute units. They enable to control communication so as to maximize effective use of bandwidth. All the fine grain parallelization is kept to the compiler.

Debuggers such as Allinea DDT handle multi-level parallelism going from single CPU or GPU core to a large number of nodes running simultaneously. Data visualization and memory layout have evolved to scale to complex memory hierarchies and ease their debugging. It is now time to get with manycore programming!

References

• Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. Patterns for Parallel Programming (First ed.). Addison-Wesley Professional.
• Performance Tuning of Scientific Applications, David H. Bailey, Robert F. Lucas, Samuel Williams

Online Resources

• GPGPU.org is a central resource for GPGPU news and information:
  • http://gpgpu.org
• ACM Parallel Computing Tech Pack:
  • http://techpack.acm.org/parallel/JourneymanTour.pdf
• Multicore association: http://www.multicore-association.org/workgroup/mpp.php
• Introduction to parallel programming:
  • https://computing.llnl.gov/tutorials/parallel_comp/