DFG IMPLEMENTATION ON MULTI GPU CLUSTER WITH COMPUTATION-COMMUNICATION OVERLAP

Sylvain Huet, Vincent Boulos, Vincent Fristot

GIPSA-lab
UMR5216 CNRS/INPG/UJF/U.Stendhal
F-38402 GRENOBLE CEDEX, France
firstname.lastname@gipsa-lab.grenoble-inp.fr

Luc Salvo

SIMAP
UMR5266 CNRS/INPG/UJF
F-38402 GRENOBLE CEDEX, France
firstname.lastname@simap.grenoble-inp.fr

ABSTRACT

Nowadays, it is possible to build a multi-GPU supercomputer, well suited for implementation of digital signal processing algorithms, for a few thousand dollars. However, to achieve the highest performance with this kind of architecture, the programmer has to focus on inter-processor communications, tasks synchronization ...
In this paper, we propose a design flow allowing an efficient implementation of a Digital Signal Processing (DSP) application specified as a Data Flow Graph (DFG) on a multi GPU computer cluster. We focus particularly on the effective implementation of communications by automating the computation-communication overlap, which can lead to significant speedups as shown in the presented benchmark. The approach is validated on a 3D granulometry application developed for research on materials.

1. INTRODUCTION

Nowadays, computers embed many CPUs and a powerful graphics card based on Graphical Processing Unit (GPU). Workstations can host several GPU boards, which are well suited for scientific and engineering computations. Such computers are linked through high bandwidth networks to compose clusters for High Performance Computing (HPC). These machines provide highly parallel multicore architectures while being cost-effective. Moreover, they significantly reduce dissipated power, and space needs compared to classical HPC clusters. However, the real challenge is to achieve the highest performances on muti-GPU architectures. The programmer has to design architecture-specific code including GPU communications and memory management, task scheduling and synchronization. So, a high level programming abstract model is required to express all these important operations. In this paper, we propose a design flow allowing an efficient implementation of a DSP application specified as a DFG on a multi GPU computer cluster. We focus particularly on the effective implementation of communications by automating the computation-communication overlap.

After presenting the related work in section 2, we show in section 3 the interest of the implementation of communication-computation overlap on multi-GPU architectures. In section 4 we present our design flow that allows an efficient implementation of an algorithm expressed as DFG on a multi-GPU architecture. It is applied in section 5 on a real world application of 3D granulometry developed for research on materials.

2. RELATED WORK

Although multi-GPU architectures are recent, many studies have been conducted to raise the level of abstraction in GPU programming.
Some authors proposed directive-based programming, to enhance a C source code. HiCUDA, developed by University of Toronto [1] is a high-level directive-based language for NVIDIA-CUDA programming. A source-to-source compiler translates a sequential C program with hiCUDA pragmas to CUDA and CPU programs. The proposed directives allow the programmer to specify code regions which are executed on the GPU or on the host, to specify the memory location of the data (host, GPU global memory, GPU shared memory, GPU constant memory),... This tool simplifies GPU porting of C sequential program. Nevertheless, at our knowledge, it is does not support asynchronous transfers between CPU and GPU and thus does not allow to do communication-computation overlap and does not target multi-GPU clusters.
A Hybrid Multi-core Parallel Programming Environment HMPP [2] is proposed by the CAPS French company that provides a set of compiler directives with tools and a software runtime support multi-core processor parallel programming in C and Fortran. HMPP subroutines can be remotely executed on a hardware accelerator as GPU, FPGA.
StarPU [3] developed by INRIA and Bordeaux University, provides a high level, unified execution model for heterogeneous systems (CPU, GPU or CELL), including high level abstraction of tasks (codelets for multiple hardware implementations), with dynamic scheduling policy and transparent support for GPU pipeline (Virtual Shared Memory).

Our goal is to provide a design flow that has a suitable entry point to the application developer of signal processing application that allows him to do not focus on the expression of inter-processor communications, the synchronization of tasks and memory allocation and optimization on both the CPU and GPU. We choose to focus on DFG since it is a formalism that have been widely used to specify DSP applications.
Before presenting the design flow that we propose, the following section shows a benchmark that allows quantifying the performance gain that can be expected when computation and communication overlap on a GPU cluster.

3. COMMUNICATION-COMPUTATION OVERLAP BENCHMARKS

In this section, we detail communication/computation overlap, in a real world application. We suppose data transfer time is around the kernel execution duration. First, the principle of communication/computation overlap for multi-GPUs systems is given. Then this overlap is shown and measured. An assessment of data transfer rate can be derived for several hardware configurations. This study is done using a single multi CPUs host workstation of the GPU cluster. The programming interface used is the CUDA Nvidia API, a popular environment for GPGPU, dedicated to Nvidia's GPU devices. CUDA is generally more efficient than OpenCL programming standard [4], as the later is affected by its compatibility to program CPUs and GPUs from different vendors.

3.1. The method

A basic test model is proposed for one host PC with two GPU boards for the data flow algorithm presented on figure 1. The architectural target is a node equipped with 2 GPU accelerators (Kernel A running on GPU-0, Kernel B running on GPU-1). The host program is designed as follows. After initializing memory blocks on the host and devices, an infinite loop handles communications, computations and synchronization for each GPU device in four steps: (1) host to device data transfer, (2) kernel execution on GPU device, (3) device to host data transfer, (4) wait for the synchronization barrier.
The code is written in multi-threaded C, based on the Posix pthread library. We create one CPU thread for each GPU board plus one CPU thread for the synchronization barrier. Allocation of the host memory blocks in page-locked (pinned) memory (with the cudaHostAlloc() function) allows a 70% increase of transfer rate compared with pageable host memory allocated by the malloc() function.

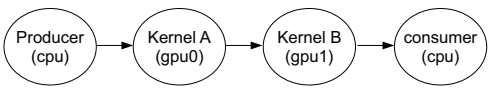


Fig. 1: data flow graph (DFG) of the test model

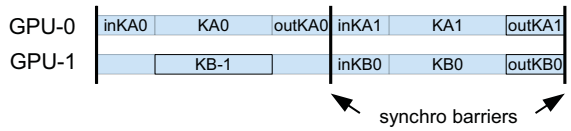


Fig. 2: serialized communication/computation execution

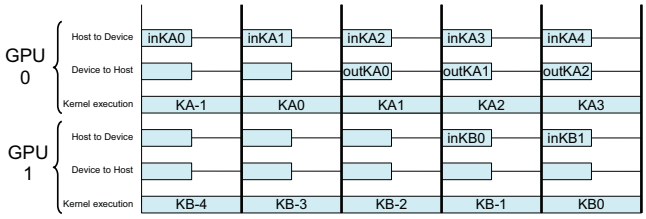


Fig. 3: concurrent execution of communications and computations

A first implementation concerns the synchronous mode with regular sequential data transfers and computations. Data transfers are launched by cudaMemcpy() functions, communications and kernel execute sequentially as shown in Figure 2. Both GPU devices run kernels concurrently. The latency stands to one cycle of synchronization, related to concurrent execution of the two kernels.
A second implementation is more efficient, running with communication/computation overlap. CPU threads launch kernels and transfers simultaneously, in asynchronous mode, which means that all data is stored in double buffers, each one allocated on the CPU and the GPU devices. Data transfers are launched by cudaMemcpyAsync() functions and concurrent execution of kernels and transfers is managed by CUDA streams. In the Figure 3 example, the latency stands to a total of seven cycles (two cycles are inserted by each kernel and retrieval of data plus one cycle to upload data to kernel B, and also two cycles are needed by host memory double-buffers for a producer out and consumer in).

3.2. Communication/computation overlap

We monitor CPU-GPU communication and GPU computation time. The overlap of data transfers and kernel execution is underscored by varying the time length of GPU kernels. We set kernel duration from 0 to twice the data transfer length. The test program saves loop's duration, with or without data transfers, in synchronous or asynchronous mode. Experimental conditions are transfers of 64 MiB blocks to achieve maximum bandwidth on the PCI express bus. We checked three motherboard configurations, recommended for GPU supercomputers
(1) an AsusTek G53JW notebook featuring an Intel I7 Q740 CPU and including a GTX460M board (mono GPU)
(2) a workstation based on AsRock X58 SuperComputer motherboard with an Intel I7 920 CPU and 3 GTX285 boards

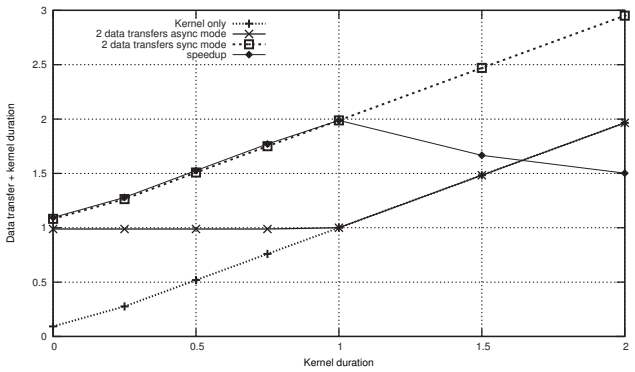


Fig. 4: Communication/kernel overlap on Asus Notebook

(3) a workstation based on Asus P6T7 WS Supercomputer motherboard with an Intel I7 920 CPU equipped and 3 GTX285 boards.
The results were measured using Nvidia's GPU CUDA SDK 3.2 (NVIDIA 260.19.26 driver) under linux Ubuntu 10.04. In asynchronous mode, there is a complete overlap if the kernel's duration exceeds transfer time. Otherwise, only the duration of the transfer remains. In synchronous mode, we find that transfer duration adds to the length of the kernel (Figure 4).

The speedup factor is defined as improvement of overlapped mode versus no overlap that reaches a factor of two if data transfer time is around kernel execution duration.

3.3. Data transfer bandwidth on PCIe

On the PC motherboards, data transfers between host memory and GPU devices go through the PCI express bus. PCIe is a high-speed point-to-point serial link connecting expansion boards to the chipset. For PCIe Gen2.0, the serial bus uses two low-voltage differential LVDS pairs, providing a 5 GT/s (giga transfers per second) in each direction. The AsusTek G53JW notebook has the 3400 Intel chipset, implementing PCIe 2.0 x8 lanes. Actual data transfer bandwidth performs 3.0 GB/s for host to device (h→d) transfer and 3.2GB/s d→h transfer. So in this case, there is no overlap between the h→d and the d→h data transfers, on the same GPU board. On Geforce boards, the PCIe interface does not support full-duplex transfers (h→d and d→h simultaneously). Half duplex transfers of Geforce boards is the bottleneck for high speed data transfers. It seems that Quadro and Tesla boards support full duplex transfers on PCIe, doubling the data transfer bandwidth. Both workstation motherboards have the X58 Intel chipset, with 2 PCIe 2.0 x16 links. The AsRock X58 motherboard (three GPU boards PCIe x16), seems to be less efficient for a single data transfer but both motherboards cap around 7.0 GB/s for six simultaneous data transfers, ie simultaneous h→d and d→h transfers for each GPU board. However, we demonstrated that communication and computation can overlap when asynchronous transfers in multi GPUs nodes are used. Thus, we improved the efficiency of multi-GPU parallelism by hiding the transfer time.

motherboard	1 transfer	2 transfers	6 transfers
AsRock X58	3.0	6.0	7.0
Asus P6T7	5.5	7.1	7.1

Table 1: PCIe 2.0 bandwidth of motherboards (in GB/s)

4. DESIGN FLOW

Our goal is to provide a design flow that allows implementing a DSP application on a computer cluster without taking care of implementation considerations. Particularly, we focus on automation of an efficient implementation of communications, i.e. with computation communication overlap, on computer cluster with multi GPU. In the first subsection, we present the steps of our design flow. In the second subsection, we detail the graph analysis and transformations we do to obtain the implementation graph.

4.1. Steps

The application is specified with a textual representation of a DFG. It is composed of nodes representing the computations and edges showing the data dependencies between them. The semantic of a DFG is as following: a node can be fired if and only if all its inputs are available. When fired, it consumes all its inputs and executes the function it is associated to and produces all its outputs. The designer associates in its DFG specification a data type to each edge and an object type to each node. Initially, we impose that each node has one input. This restriction is discussed at the end of subsection 4.2 and will be removed in a near future. Figure 5 presents an example of a DFG: node `p` produces data consumed by node `a` which produces data for node `b` that broadcasts it to nodes `c1`, `c2`, `d`, and so on. An iteration is the firing of all nodes of the DFG. The meaning of the couple between parentheses is given in subsection 4.2. The architecture is described with a textual representation of an Architecture Graph (AG). The nodes represent the processing elements, and the edges the communication channels between them. Figure 6 presents an AG of a cluster of two computers where each has a motherboard with one CPU and three GPUs. Here, the designer

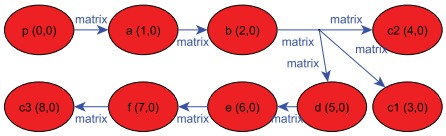


Fig. 5: Data Flow Graph example

also specifies the nature of each communication link. On our cluster, CPU and GPUs on the same computer communicate through PCIe links whereas the computers through Infiniband Network.

The mapping of an application on the architecture is specified in the textual representation of a DFG. Figure 7 presents the mapping we choose to illustrate the design flow.
Finally, the designer has to provide a c++ class description for each kind of object associated to the DFG nodes and to complete resolution functions that are called at runtime to do the memory allocations, the instantiation of the c++ class associated to each nodes ...

At this time, the designer's code is compiled and linked with a library we developed, called Parallel Computations with Communications Overlap (PACCO). This library allows to obtain a binary which can execute the application with computation communication overlap. Since we rely on MPI [5] for inter-computer communications, we use mpirun to launch and distribute the application on the cluster. At runtime, the DFG with its mapping annotations and the AG are analyzed. We do some graph transformations to allow communication-computation overlap and to obtain an optimized Implementation Graph (IG). We then create the threads that will manage the CPU and GPU computations, i.e. DFG nodes firing, the communications among them and the synchronizations. This code needs to be recompiled only when the designer introduces a new data type in the DFG or associates a new object to a DFG node.
Figure 8 summarizes all the steps of the design flow. Here, the designer only provides the information within the dark rounded rectangles.

4.2. Graphs analysis and transformations

The objective of the analysis and transformations is to obtain an IG which will be used by each CPU and GPU thread to determine what it has to do. The IG specifies the scheduling of the DFG nodes and the buffers that allow them to communicate through the architecture.

4.2.1 Scheduling

A first step consists in finding a schedule, i.e. a firing order of the nodes for an iteration of the DFG. For this purpose we used a recursive algorithm called on sink nodes. The schedule

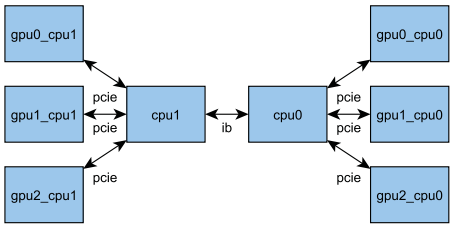


Fig. 6: Architecture Graph example

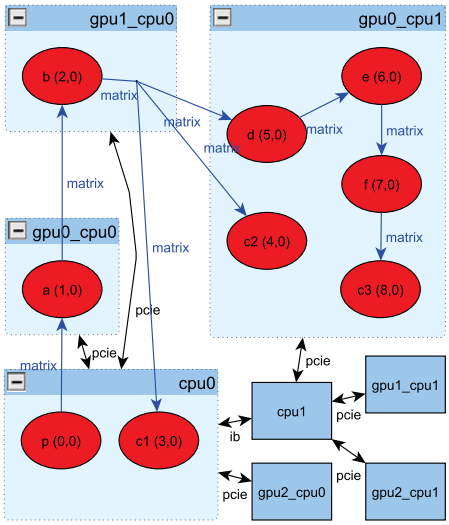


Fig. 7: DFG mapped on AG

of our application example is specified in the first element of the couple between parentheses in the DFG nodes.

4.2.2 Buffer node insertion

The DFG of an application specifies the data dependencies (edges) between computations (nodes). From the implementation point of view, we need buffers to store the data that is produced and consumed by the nodes. This means that buffers are required all along an architecture's path connecting the nodes which with producer/consumer relationship. Figure 9 illustrates the result of buffer node insertion of the applica-

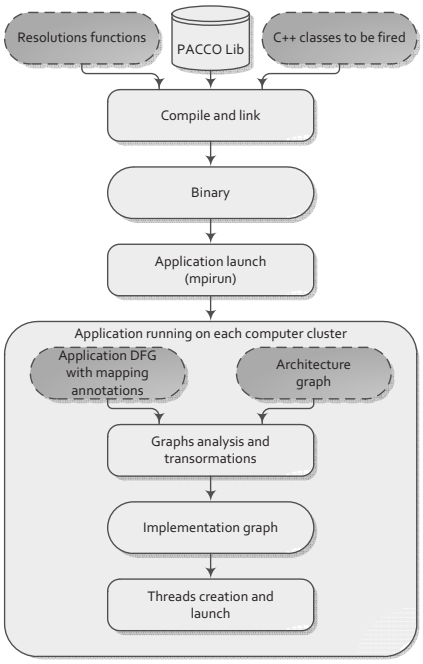


Fig. 8: Design flow

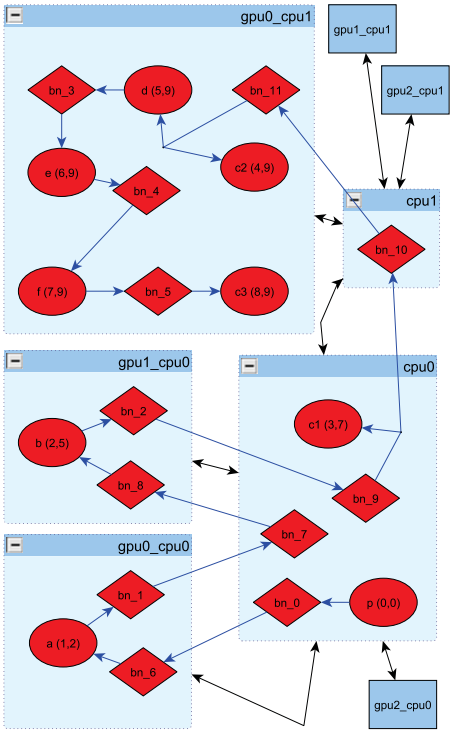


Fig. 9: Buffer nodes insertion

tion presented in Figure 5 on the architecture Figure 6 with the mapping presented in Figure 7.

For example we consider the case of nodes `b` And `c2` that have a producer-consumer relationship. Node `b` is mapped on GPU1 of CPU0, whereas `c2` is mapped on GPU0 of CPU1. To go from GPU1 of CPU0 to GPU0 of CPU1, you have to pass by CPU0 and then by GPU1. Thus, four buffers are required: `bn_2` allocated on the memory of GPU1 of CPU0 computer, `bn_9` allocated on the CPU0 computer main memory, `bn_10` allocated on the CPU1 computer main memory and `bn_11` allocated on the memory of GPU0 of CPU1 computer. We notice that the data produced by `b` is also consumed by `c1`. As the architectural path going from `b` to `c1` is included in the path from `b` tp `c2`, no other buffer node is required.

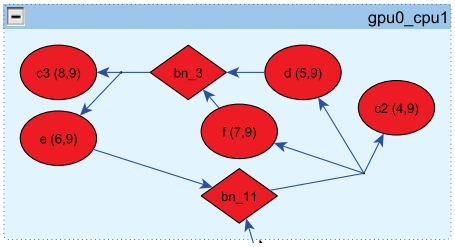


Fig. 10: Buffer nodes optimization

4.2.3 Buffer node depth and optimization

This step consists in determining the depth of each buffer node, simple or double, and to optimize their allocation through sharing. The choice of the depth of a buffer node depends on location of the processing elements of the nodes it connects and the communication strategy, i.e. with or without computation overlapping. The communication strategy also impacts the IG model of execution

a) Inter processing elements communication with computation communication overlap In this case, each buffer node that receives or send data to a processing element other than the one it is mapped to must be a double buffer. Indeed, in this case the two following situations have to be considered. (1) The case of a buffer node which receives a data from another processing element. To implement computation communication overlap, the buffer node can be written by an asynchronous data transfer, whereas it can be read by an internal DFG node. It is the case of `bn_7`, in Figure 9, which is read by an asynchronous data transfer between `bn_7` and `bn_8` whereas it can be written by another asynchronous data transfer between `bn_1` and `bn_8`. (2) The case of a buffer node that sends data to another processing element. With computation communication overlapping, the buffer node with the data sends asynchronously to the other processing element is written by an internal node. It is the case of `bn_1` in figure 9 which can be asynchronously sent to `bn_7` while it is written by node `a`.

b) Inter processing elements communication without computation communication overlap In this case, the transfers between processing elements are done when all the kernels of each processing elements have been fired. Thus only single buffers are required.

c) Intra processing elements communication As the DFG nodes are fired with respect to the scheduling, only single buffer nodes are required as internal storage. We develop an optimization pass, which allows sharing the internal buffer nodes that transport the same data type. Even the double buffer node's part which is not used by an asynchronous transfer between two processing elements can be shared. Figure 10 shows the results of this optimization applied to our example application. In the context of computation communication overlap, the part of `bn_1` double buffer which is not used by the asynchronous transfer between CPU1 and GPU0 of CPU1 is read by `c2,d,f` and written by `e` and buffer node `bn_3` is read by `e,c3` and written by `d,f`. To ensure data consistency, all these transfers are of course done in the scheduling order. This optimization avoids the instantiation of two buffer nodes. Currently, we are working to determine a scheduling specific to each processing element that minimizes the buffer nodes number.

4.2.4 Implementation graph model of execution

At runtime, a CPU thread is associated to each processing element (GPU computations and transfers are managed through a CPU thread which launches the GPU kernels and transfers).

These threads manage the nodes mapped on the processing elements that are associated.
In the case with computation communication overlap they iteratively execute the following sequence: (1) launch the asynchronous transfers (2) fire each node with respect to the scheduling (3) wait until all the nodes all processing elements have finished for their execution and asynchronous transfers have completed.
In the case without computation communication overlap they iteratively execute the following sequence: (1) fire each node with respect to the scheduling (2) wait until all the nodes all processing elements have finished their execution (3) lauch the transfers (4) wait until all the transfers finished.

4.2.5 Latency computation

Transfers between processing elements introduce delay cycles in either case with computation communication overlap or without.
With computation communication overlap: each double buffer introduces a delay of one cycle. The latencies specified in the second element of the couple between parentheses in figure 9 were computed in this case.
Without computation communication overlap, each pair of connected buffer nodes located on different processing elements introduces a delay of one cycle.
So, whatever the case, at a given time, DFG nodes can be working on different iterations of the application DFG. Moreover, to avoid firing DFG nodes before valid data is present, and thus to avoid transients, we compute the latency of the input of each DFG node. The threads only fire a DFG node after a number of cycles equals to this latency.

4.2.6 Multiple input problem

Initially, we impose that a DFG node only has one input. Indeed when a DFG node has more than one input, it can be necessary, depending on its predecessors mapping, to resynchronize its inputs since their latency can be different. We are working on this problem through two means: (1) delaying the execution of shortest paths (2) adding delay lines.

5. CASE STUDY: GRANULOMETRY

The goal of our work is to provide a design flow that simplifies CPU/GPU and GPU/GPU inter-communication and allows computation/transfer overlapping. In order to apply this design flow concretely, we intend to use the granulometry application. Firstly, we will present the algorithm. Secondly, we will discuss its practical interest and the reason why this algorithm is adapted to parallel programming and more specifically GPUs. Lastly, we will present the speedup we get thanks to our design flow implementation of the algorithm.

5.1. Algorithm description

Granulometry is the study of the statistical distribution of the sizes of a population of finite elements. In other words, it is the study of an image's object sizes. In physics, that would resemble sieving (grain sorting): the image would be filtered with a series of 'sieves' of different mesh sizes.

The algorithm is based on the opening mathematical morphology operator which consists in an erosion followed by a dilation with the same structuring element. It takes an image to process as an entry then computes openings with an increasing structuring element size, until all objects in the volume disappear ; meaning the volume is empty. After each opening, we collect the number of positive pixels still present in the image. We then plot the results on a curve: the granulometric curve. The abscissa of this curve represents the number of openings, and ordinate represents the corresponding number of positive pixels remaining in the image. The discrete derivative of the granulometric curve is called the pattern spectrum and the abscissa of its peak is the predominant size of objects in the image (see Fig. 11).

5.2. Practical interest

In order to study a material's characteristics, tomographic reconstruction is widely used as an efficient method. Many implementations of these algorithms have been ported to GPUs considerably accelerating computations [6, 7]. However, less attention has been paid to post-tomographic computations such as granulometry. Yet, their execution time is not negligible, can be easily reduced, and hence discredits the efforts made on tomographic reconstruction only.
Ganulometry can be efficiently implemented on GPU since:
(1) there aren't many flow control instructions needed
(2) it is the same operations done on each pixel of the image
(3) one pixel's result is independent from other pixels' results
(4) as we work on binary images, it uses basic operations such as logic AND and logic OR (maximal instruction throughput according to the CUDA programming guide [8]) and few intermediate variables meaning the occupancy should be maximal [9]

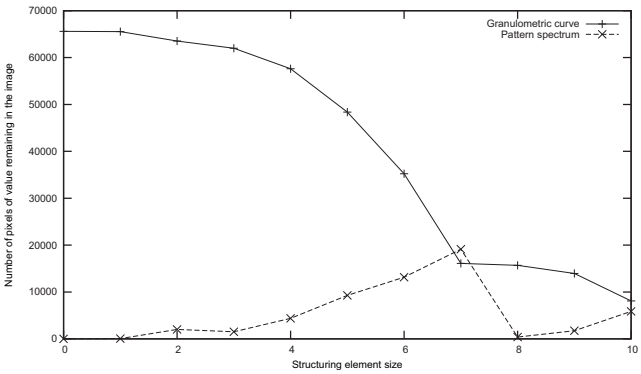


Fig. 11: Example of appearance of a granulometric curve. We can see on the pattern spectrum an extrema indicating the predominant size of the objects in the image.

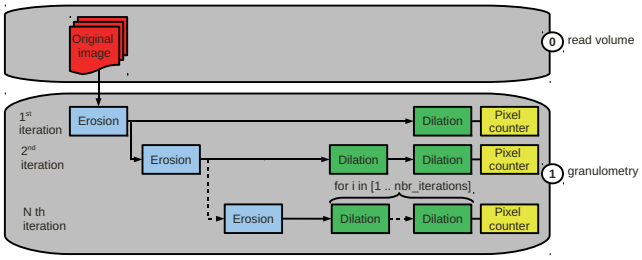


Fig. 12: The Data-Flow Graph modelization.

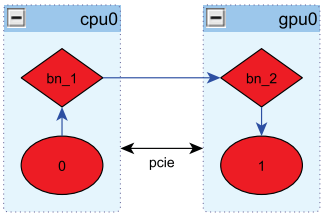


Fig. 13: The mapping of the dataflow graph on the architecture.

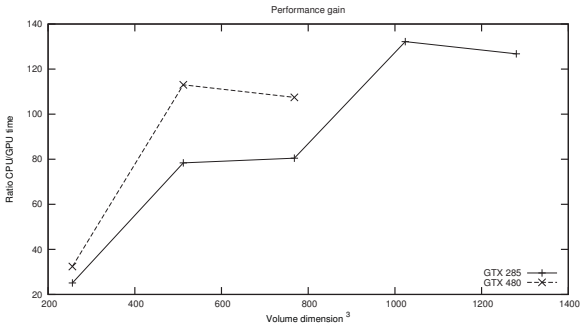


Fig. 14: Performance gain after porting the granulometry algorithm to GPU.

(5) processing doesn't need to follow a special data order, so we can do it in the simplest way, sequentially, to have coalesced memory accesses

5.3. The DFG implementation of the granulometry application

The granulometry GPU implementation comes in handy when processing huge volumes of data. However, the bigger the volume to process is, the more time it takes to be read from the hard disk: our SATA disk evaluated bit rate is 120 MB/s. Commonly used 3D volumes' sizes for post-tomographic processing are of at least 1024^3 voxels. Therefore, an improvement could be to overlap the CPU "read and binarize" task with the GPU "granulometry" task. However, this implies the use of double buffers and introduces a latency. Thus, this solution is mostly useful when processing the granulometry application on a set of volumes. In this subsection, we will illustrate the steps to follow in order to implement this solution by using our design flow. First, we decompose our application in computing entities (nodes). Then, we do the mapping between these nodes and the architecture. Finally, we conclude with the results obtained, the benefits and the limitations to deal with in near future.

5.3.1 DFG application modelization

Our computing entities (nodes) are easily identified. We can divide the process into two tasks (see Fig. 12): (1) read volume from hard drive and binarize it (on CPU) (2) process the granulometry application on the loaded volume (on GPU).

5.3.2 Mapping DFG on architecture

The mapping is then processed, automatically adding buffers, according to the architecture we wish to port the algorithm onto. In our case, it would look like Fig. 13

5.3.3 GPU versus CPU implemented application results

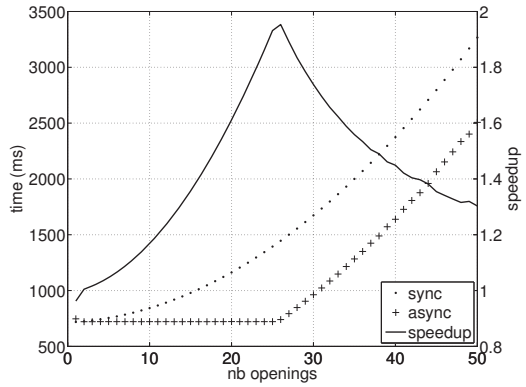
Firstly, let's observe the acceleration brought just by porting this specific application to the GPU (without task and transfer overlapping). Fig. 14 shows the speedup of the GPU implemented granulometry application compared to the CPU single threaded implementation.

Sizes of the processed images on Fermi GTX 480 and GTX 285 differ because of the difference in available device memory: 1.5 GB for GTX 480 and 2 GB for GTX 285. Also, three buffers (four buffers in asynchronous mode) are needed on the GPU. Moreover, pitched memory is used which significantly increases (up to 8 times) the amount of allocated memory for 2D memory alignment reasons. That limits the size of the images that can be processed on the GPUs. Let us note that for a fixed image size, the performance gain is constant whatever number of openings are performed. Thus, performance gain is independant from the maximal object size in an image but depends only on the number of pixels simultaneously processed by the GPU compared to the CPU.

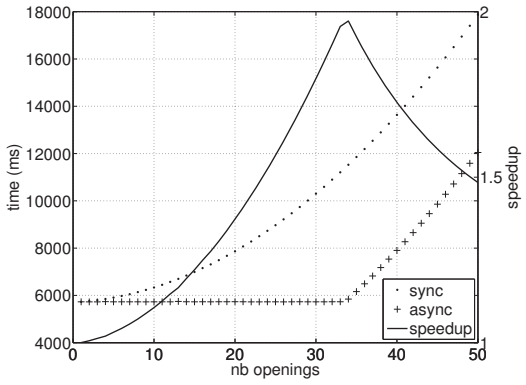
5.3.4 Synchronous versus asynchronous results

Asynchronous mode differs from synchronous in the fact that, the CPU task (hard drive read and binarize) and the GPU task (granulometry) are processed simultaneously. Also, the CPU $\rightarrow$ GPU PCI-e transfer is overlapped with the GPU kernels execution. The results obtained for our CPU/GPU task overlapping granulometry implementation appear on Fig. 15.

For a small number of openings, meaning when objects' sizes in the processed volume are small, the GPU granulometry computations are faster than the CPU "read and binarize" task. This is clearly demonstrated on Fig. 15: as long as the "CPU read and binarize" task is slower than the "GPU computations", you can observe that the processing time remains constant on the asynchronous curve while increasing by the amount of time spent by the granulometry application on the synchronous one. On the asynchronous curve, when the number of openings becomes big enough, the granulometry application time exceeds this constant value and the processing



(a) $512^3 voxels$



(b) $1024^3 voxels$

Fig. 15: Synchronous and asynchronous timings for different volume sizes.

time increases equally to the time spent by the GPU computations. In other words, when GPU time reaches CPU time the speedup is at its peak value: x2 since the both tasks are perfectly overlapped. The speedup then decreases asymptotically to 1. In conclusion, our design flow automatically overlapped CPU and GPU tasks successfully with a gain which can attain a x2 speedup. In a less impacting manner, GPU kernels and the PCI-e CPU → GPU memory transfer were also overlapped automatically. Since the granulometry application is not suited for multi-GPU pipeline implementation, we didn't have the opportunity to develop a multi-host multi-GPU application thanks to our design flow but that would certainly be of higher interest for people aiming at automatically dispatching their workload onto parallel processing plateforms.

6. CONCLUSION

This paper presents our analysis of task parallelism implementation on a GPU cluster's node, dealing with communication and computation optimization.

We made the following contributions: We detailed communication and computation overlap measurement method. Our microbenchmarks revealed a speedup factor of two when data transfer time is around the kernel execution duration. Using our design flow, the programmer doesn't have to deal with inter-component communication and allocation of buffers. He doesn't waste time on basic, rudimentary and sometimes complex coding (MPI, POSIX threads, etc.) but rather focus on the computation code development. Thus, an application developed for a certain configuration might be easily portable on another platform. Also, with the efforts undertaken by the GPU manufacturers to make the hardware even more adapted to scientific computations, components' inter-communication will be the most important programming bottleneck after kernel coding. The proposed design flow automates tasks overlap. It allows to hide the CPU-GPU transfer time and lead to optimal use of the hardware.

7. REFERENCES

[1] Tianyi David Han and Tarek S. Abdelrahman, "hicuda: a high-level directive-based language for gpu programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, 2009.

[2] R. Dolbeau, S. Bihan, and F. Bodin, "Hmpp: A hybrid multi-core parallel programming environment," in *Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[3] Cédric Augonnet, Samuel Thibault, and Raymond Namyst, "StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines," Research Report RR-7240, INRIA, 2010.

[4] Kamran Karimi, Neil G. Dickson, and Firas Hamze, "A performance comparison of cuda and opencl," *CoRR*, vol. abs/1005.2581, 2010.

[5] William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*, MIT Press, 1999.

[6] Damien Vintache, Bernard Humbert, and David Brasse, "Iterative reconstruction for transmission tomography on gpu using nvidia cuda," *Tsinghua Science & Technology*, vol. 15, pp. 11 – 16, 2010.

[7] Byunghyun Jang, D. Kaeli, Synho Do, and H. Pien, "Multi gpu implementation of iterative tomographic reconstruction algorithms," in *Biomedical Imaging: From Nano to Macro, 2009. IEEE International Symposium on*, 2009.

[8] NVidia, "Nvidia cuda c programming guide 3.2," .

[9] NVidia, "Cuda occupancy calculator," .