Efficient implementation of data flow graphs on multi-gpu clusters

Vincent Boulos, Sylvain Huet, Vincent Fristot, Luc Salvo, Dominique Houzet

Journal of Real-Time Image Processing manuscript No.
(will be inserted by the editor)

Vincent Boulos · Sylvain Huet · Vincent Fristot · Luc Salvo · Dominique Houzet

Efficient implementation of data flow graphs on multi-gpu clusters

Received: date / Revised: date

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 high level programming model based on a Data Flow Graph (DFG) allowing an efficient implementation of Digital Signal Processing (DSP) applications on a multi-GPU computer cluster. This DFG based design flow abstracts the underlying architecture. We focus particularly on the efficient implementation of communications by automating computation - communication overlap, which can lead to significant speedups as shown in the presented benchmark. The approach is validated on three experiments: a multi-host multi-gpu benchmark, a 3D granulometry application developed for research on materials and an application for computing visual saliency maps.

Keywords GPU cluster · Data Flow Graph (DFG) · computation communication overlap

Vincent Boulos
GIPSA-lab, Image-Signal Department
CNRS UMR 5216, University of Grenoble
38402 Saint Martin d'Heres
France
Tel.: +33-4-76826424 / Fax: +33-4-76574790
E-mail: vincent.boulos@gipsa-lab.grenoble-inp.fr

Sylvain Huet
GIPSA-lab, Image-Signal Department
CNRS UMR 5216, University of Grenoble
38402 Saint Martin d'Heres
France
Tel.: +33-4-76574362 / Fax: +33-4-76574790
E-mail: sylvain.huet@gipsa-lab.grenoble-inp.fr

Vincent Fristot
GIPSA-lab, Image-Signal Department
CNRS UMR 5216, University of Grenoble
38402 Saint Martin d'Heres
France
Tel.: +33-4-76574365 / Fax: +33-4-76574790
E-mail: vincent.fristot@gipsa-lab.grenoble-inp.fr

Luc Salvo
SIMAP, GPM2 group
CNRS UMR 5266, University of Grenoble
38402 Saint Martin d'Heres
France
Tel.: +33-4-76826379 / Fax: +33-4-76826382
E-mail: luc.salvo@simap.grenoble-inp.fr

Dominique Houzet
GIPSA-lab, Image-Signal Department
CNRS UMR 5216, University of Grenoble
38402 Saint Martin d'Heres
France
Tel.: +33-4-76574361 / Fax: +33-4-76574790
E-mail: dominique.houzet@gipsa-lab.grenoble-inp.fr

1 Introduction

Recent computers embed many CPU cores and at least one powerful commodity off-the-shelf 3D graphics card commonly called 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 build clusters for High Performance Computing (HPC). These machines provide highly parallel many-core 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 performance on muti-GPU architectures. The programmer has to design architecture-specific code including CPU-GPU communications, memory management, task scheduling and synchronization. Therefore, a high level programming paradigm is required in order to abstract all these important operations from the programmer.
In this paper, we propose a design flow allowing efficient implementation of a DSP application specified as a DFG on a multi-GPU computer cluster. Effective implementation of communications by automating the computation - communication overlap is also adressed.
The rest of the paper is organized as follows. Section 2

discusses some related work. Section 3 presents the benefits of communication - computation overlap on multi-GPU architectures. Section 4 details the efficient implementation through our design flow of an algorithm expressed as a DFG mapped on a GPU cluter. Section 5 shows experimental results for three examples: a multi-host multi-gpu benchmark, a 3D granulometry application developed for research on materials and an application for computing visual saliency maps.

2 Related Work

In the past years, there has been a trend to develop programming languages adapted to multi-core / many-core computing. Since the emergence of accelerators (such as GPU, Cell, etc), today's challenge is toward hybrid architectures.

The use of GPU clusters for scientific computing is not a recent topic. It was studied before any appropriate programming paradigm for GPU scientific computing even existed: the first exploited scalable GPU cluster was programmed using Cg language [10]. Now that adapted programming paradigms such as Compute Unified Device Architecture (CUDA) and Open Computing Library (OpenCL) exist, more focus is given to this kind of hybrid computing. The difficulty remains in the necessity to combine accelerator dedicated programming languages with traditional CPU programming languages in order to exploit an hybrid CPU/GPU cluster.

2.1 Parallelizers

Many studies have been conducted in order to simplify the code generation for users not accustomed to GPU programming (HMPP [9], HiCUDA [14], PGI Accelerator [30], StarSS, OpenACC, etc). An evaluation of five of these frameworks with a real world application from medical imaging is reported in [22]. The Compaan HotSpot Parallelizer [29] from Compaan Design transforms codes with parameterized nested loop to a process network which can then be implemented on multicores CPU, GPU or FPGAs.

Hybrid Multi-core Parallel Programming (HMPP) [9], proposed by the CAPS French company, helps implement incremental porting strategies by providing a set of compiler directives with tools and a also software runtime support for C and Fortran. HMPP allows to parallelize a sequential code and to distribute computation over CPU and multiple GPU cores. Several runtimes have also been proposed.

2.2 Runtimes

Several task scheduling strategies for a multi-GPU platform based on a pool of tasks have been proposed [19, 5, 6, 24]. However, the strategy of maintaining a pool of tasks that are randomly dispatched to idle processing units might perform poorly for applications with data dependencies and thus dependencies between tasks. In other words, applications with fine-grain parallelized independant tasks are not a generality. Some solutions propose a history-based time estimator for job scheduling in order to dispatch redundant tasks on the most suitable architecture. Previous tasks are distinguished based on their input data size and their computation time. However, this model is simplistic and implies exectued tasks to be launched several times in order to be efficient.

Moreover, communications between processing units introduce more complexity to the programmer since it will imply different programming APIs for each type of communication: for example, CPU-GPU transfers might use CUDA, multi-threads use OpenMP and multi-core use MPI. Some runtime system environments have been developped in order to abstract the communication layer to the programmer [3, 23, 20, 8].

StarPU [3] is a runtime system designed to dynamically schedule a pool of tasks on heterogeneous cores with various scheduling policies. It also avoids unecessary data transfers thanks to a Virtual Shared Memory (VSM) software. However, from our experience, it seems that the implementation of communication/computation overlapping, which can lead to significant speedups, requires that the designer express it in its code. This can be a tiedous task.

Most presented solutions focus on the exploitation of hybrid architectures for data parallelism purposes (using a Map-Reduce scheme) or dispatching of independant tasks. Our work focuses on a more specific scenario: the implementation of DSP applications on a multi-GPU cluster.

2.3 DFG based design flows

DFG is a formalism that expresses an application as a computation pipeline that highlights the potential parallelism of implementation. It has proved along years to be an adequate formalism to model DSP applications.

For example, the SynDex [12] CAD software generates an optimized implementation of on application specified with a directed acyclic hyper-graph on an architecture specified with a directed graph called multicomponent architecture which is composed of programmable components such as RISC, CISC, DSP processors and/or

of non programmable components such as ASIC, FPGA. Some heuristics are used to generate an optimized implementation of the application, e.g. minimization of the execution time of the application. It results in a distribution of the application on the architecture and a static schedule of the computations and communications. It can be noticed that the potential parallelism of the implementation depends on the application specification, i.e. SynDEx does not perform any analysis of atomic vertex to extract some parallelism. However, at our knowledge it is not possible to natively target GPU cluster with SynDEx.

Thus DFG is well suited to the specification of applications that must be implemented with time rate constraints, such as streaming applications, on parallel architectures, such as GPU clusters. Also, several studies have introduced the use of DFG for multi-GPU and/or multi-core workstations [11, 15, 4, 27, 1, 2].

Sbirlea et al. [27] present a design flow relying on a DFG specification of the application on a multi-architecture (CPU, GPU and FPGA) platform with work-stealing capabilities. Their framework has a dispatching model of tasks among processing units based on their best performance affinity provided manually (hard-coded) by an expert. However, from a more practical point of vue, this solution implies the development of as many implementations as there are architectures: CPU,GPU and FPGA. Therefore, this work is mostly interesting for the use of applications with relatively high common tasks. Their goal is to allow medical imaging domain experts to program without worrying about parallelism and the details of underlying C implementation of the model. Also, the amount of time spent developping and optimizing the different implementations is an investment compensated by the fact that they will be part of a task database to be used several times along time.

Aldinucci et al. [1] present a two phase process targeting heterogeneous architectures built of multi-cores and GPUs. The first step is aimed at translating high level languages into macro data flow graphs. These graphs are then executed by means of a parallel macro data flow interpreter specialized to run data parallel computations on GPUs without programmer intervention. Also, all the details relative to data movement to and from GPU as well as to memory allocation and to thread scheduling on the GPU are managed by a modified interpreter loop. The preliminary experimental results show that the approach is feasible and efficiently implements different kinds of applications on a heterogeneous, single node architecture.

2.4 Our design flow

Our solution also relies on a DFG specification of the application but our goal is different. It is to simplify the porting of an already GPU-implemented application onto a GPU cluster and, most importantly, to free the designer from expressing all the extra coding glue needed by the communications and synchronizations between the processing elements. Indeed, DFG's nodes may be dispatched on different processing units of different kinds (CPU/GPU) and its edges can represent different data types which can be implemented on different kinds of communication channels (PCIe, Inifiniband, Ethernet)... Also, while the DFG formalism expressed within each processing element (CPU or GPU) is sequentially executed, the processing elements run in parallel. Thus, the data flow between them and the computations need to be synchronized to ensure the data coherency. Therefore, a representation of the model of implementation presented later in this article would be a set of communicating sub-DFG, one per processing element, with a global synchronization mecanism ensuring the data coherency.

To our knowledge, there is no work that completely abstracts all these routines from the programmer. In this paper, we provide a design flow that has a suitable entry point to the application developer of DSP applications. It abstracts several programming portions of code by automating their production in order to lighten the programmer's work. These abstractions concern:

- the memory allocation and optimization on both CPU and GPU
- the expression of inter-processor communications
- the synchronization of tasks

Another feature brought by our design flow is the possibility to overlap computation and communication which can significantly reduce the amount of time spent per iteration in a pipeline.

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-GPU 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-CPU 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 [17] [7], as the later is affected by its ability to produce versatile CPU/GPU-compatible code.

3.1 Methodology

A basic test model is proposed for one host PC with two GPU boards for the following data flow algorithm:

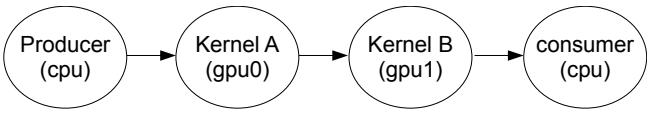

Fig. 1: data flow graph (DFG) of the test model

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:

- host to device data transfer
- kernel execution on GPU device
- device to host data transfer
- 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. The reader can find a detailed comparison between non pinned and pinned memory here[18].

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 two cycles of synchronization due to concurrent execution of the two kernels.

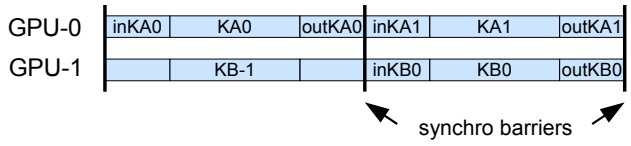

Fig. 2: serialized communication - computation execution

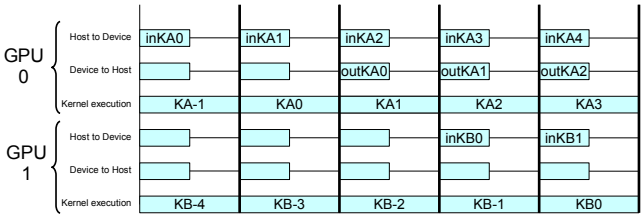

Fig. 3: concurrent execution of communications and computations

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 six cycles (two more cycles are inserted on each GPU for host → device and device → host communications).

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:

- an AsusTek G53JW notebook featuring an Intel I7 Q740 CPU and including a GTX460M board (mono GPU)
- a workstation based on AsRock X58 SuperComputer motherboard with an Intel I7 920 CPU and 3 GTX285 boards
- 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.

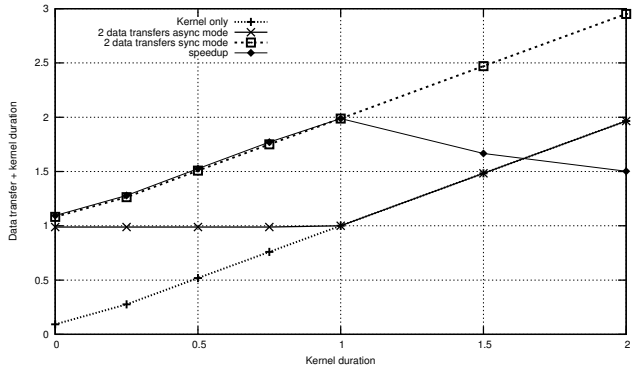

Fig. 4: Communication/kernel overlap on Asus Notebook

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.

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)

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. Quadro and Tesla boards support full duplex transfers on PCIe which allows at best 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.

Thus 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.

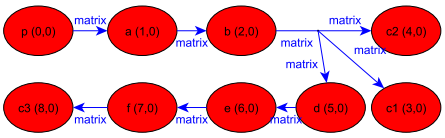

Fig. 5: Data Flow Graph example

4 Design Flow

Our goal is to provide a design flow that allows the implementation of a DSP application on a computer cluster without considering the cluster's inter-processing elements communication - specific implementations. Particularly, we focus on an automated and efficient implementation of communications. This includes buffer allocation, inter-processor communication and computation - communication overlap management.

The first subsection presents the efforts the designer produces in our design flow to express the work distribution on a cluster's processing elements of a DFG application. The second subsection discusses the runtime execution process. Finally, in the last subsection, we present the two runtime communication - computation strategies offered to the user.

4.1 Programmer's Contribution

The Algorithm-Architecture Adaptation (AAA) is expressed thanks to the combination of two graphs: the application DFG and the Architecture Graph (AG).

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 nodes. The semantic of a DFG is as follows: 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, producing all its outputs. A data type is associated to each edge and a function to each node. 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. Each node's label is composed of its name and a 2-tuple representing its scheduling and latency ; it is discussed more in detail in subsection 4.2. Notice that the DFG decomposition is manual. It is naive because the goal of the tool is not to help in the parallelization process but rather help to deploy an application on a cluster. Thus, the programmer already has a good understanding of the bottlenecks of the application and decides which computation node is best suited for each step in the process. Parallelizing compilers/tools can be used beforehand.

The architecture is described with a textual representation of an AG. The nodes represent the processing

elements, and the edges the communication channels between them. Edges also specify the nature of each communication link. Figure 6 presents the AG of a cluster of two computers with one CPU and three GPUs each. On our cluster, CPU and GPUs on the same computer communicate through PCIe whereas computers communicate through Infiniband Network.

The mapping of an application on the architecture is specified in the textual representation of the application DFG. Figure 7 shows a possible mapping between the application DFG and the AG we presented earlier.

The designer's code is then 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 capability. Since we rely on MPI [13] for inter-computer communications, we use mpirun to distribute then launch the application on the cluster. At runtime, the DFG with its mapping annotations and the AG are analyzed. Some graph transformations are done to allow communication - computation overlap and to obtain an optimized memory allocation. We then create the threads that will manage the CPU and GPU tasks: DFG nodes firing, memory transfers and synchronizations. This code needs to be recompiled only when the designer introduces a new data type (resolution functions) in the DFG or associates a new function (C++ class) to a DFG node.

To summarize, the designer role is:

1. provide an application DFG
2. provide an AG
3. provide a c++ class description for each kind of function associated to the DFG nodes
4. complete resolution functions that are called at runtime to do the memory allocations

Figure 8 summarizes all the steps of the design flow and its execution process. The designer provides the information within the dark rounded rectangles.

4.2 At Runtime

The objective of graph analysis is to obtain an Implementation Graph (IG) that contains information used by each CPU and GPU thread to determine the data flow path, an optimised buffer allocation, functions scheduling and their repartition among the processing elements. There are five steps the execution process goes through and which are detailed in the following subsubsections: node scheduling, buffer insertion, buffer size, optimizing memory allocation and computing node's latency.

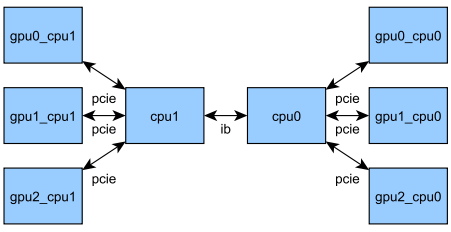

Fig. 6: Architecture Graph example

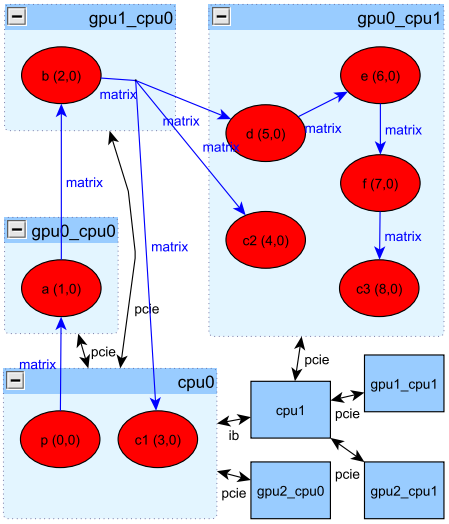

Fig. 7: DFG mapped on AG

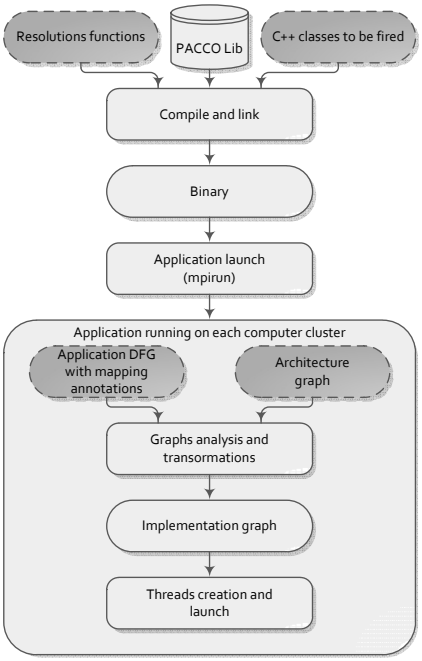

Fig. 8: Design flow

4.2.1 Scheduling

First step consists in finding a schedule, i.e. the nodes' firing order for each iteration of the DFG. For this purpose, a recursive algorithm is used which principle is as follows: a node can be scheduled if and only if all its predecessors are scheduled, otherwise schedule the predecessors. Each time a node is scheduled, it takes the value of a counter which is then incremented. The schedule is specified in the first element of the couple appearing on each DFG node label.

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 between nodes. This is easily done on the application DFG by interleaving buffers between nodes. However, while mapping an application DFG to an AG we introduce some architecture specific constraints which makes the data flow path evolve. When mapped on the same processing element, buffer insertion between nodes is the same as working directly on a DFG ; when mapped on different processing elements, the data flow path becomes longer and its length may vary. Figure 9 illustrates the result of buffer node insertion of the application DFG presented in Figure 5 on the AG shown on Figure 6 with the mapping proposed 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: (1) `bn_2` allocated on the memory of GPU1 of CPU0 computer (2) `bn_9` allocated on the CPU0 computer main memory (3) `bn_10` allocated on the CPU1 computer main memory (4) `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.
To insert the buffer node, we implement the algorithm which behaves as follows: a buffer node is inserted for all DFG nodes outputs. The inserted buffer nodes are mapped on the same processing element as the DFG node it is connected to (case of buffer nodes `bn_0..5`, in Figure 9). For each of these buffer nodes, we search an architectural path going from the considered buffer node to the DFG node it targets. When travelling along this path, with every processing element crossed, we insert a new buffer node if there exists none with the same source, cf. the two paths of previous example.

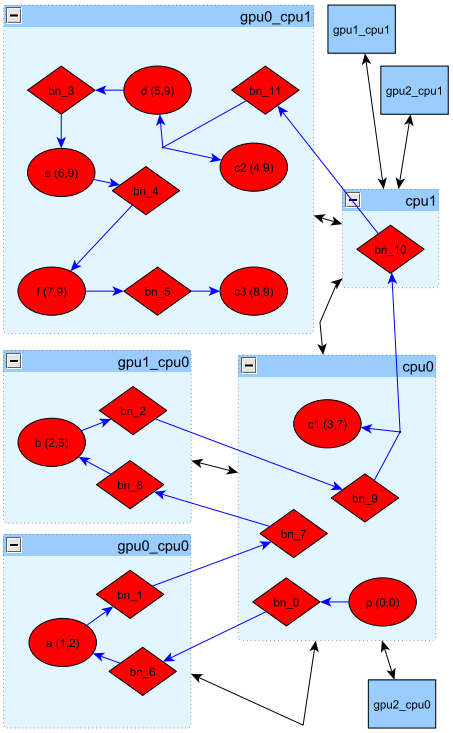

Fig. 9: Buffer nodes insertion

4.2.3 Buffer Depth

This step consists in determining the depth of each buffer node: simple or double. The choice of the depth of a buffer node depends on the location of its surrounding computation nodes (same/different processing elements) and on the communication strategy (with/without computation overlapping): if a buffer receives/sends data to a processing element different from its own in asynchronuous run, it has to be allocated as a double buffer. The communication strategy also impacts the IG model of execution, cf. sub section 4.3.

4.2.4 Optimizing Buffer Memory Usage

Memory allocation optimization is possible thanks to an optimization pass that allows to share buffers inside a processing element. This is done thanks to a graph coloring algorithm which colors the mergeable buffer nodes in the same color ; the goal is to reduce the number of colors in the graph. The coloring algorithm takes one input: the buffer nodes' incompatibility graph. This incompatibility graph, containing information about buffer nodes' ability to be merged, is produced following a single rule: two buffer nodes are incompatible (cannot be merged) if they are connected to the same computation node. It makes sense since a computation node may read/write to the same memory location otherwise.

Moreover, when two or more buffers are merged, it is the buffer with the biggest need in memory size that is allocated. Thus, all merged buffers are contained in the most voluminous one. Figure 10 shows the results of this optimization applied to our presented 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 has freed the allocated memory from two buffer nodes.

4.2.5 Latency Computation

Transfers between processing elements introduce delay cycles in either with/without computation - communication overlap cases.

- with computation - communication overlap: each double buffer introduces a delay of one cycle. The latency specified in the second element of each node's couple in figure 9 were computed in this case
- without, each pair-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.

4.3 Runtime Strategies

At runtime, a POSIX CPU thread is associated to each processing element (a CPU thread is dedicated to each GPU device for transfers/computations management). Each thread knows its attributed nodes by checking the IG for nodes mapped to its processing element. Two types of execution exist. Each one iteratively executes in cycles as follows:

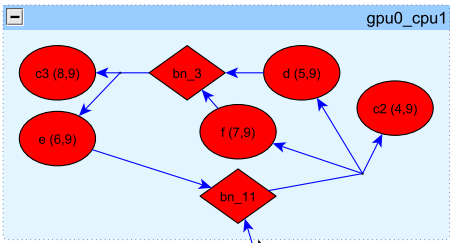

Fig. 10: Buffer nodes optimization

1. without computation - communication overlap
   - launch CPU-CPU transfers
   - wait for all cluster's transfers completion
   - launch CPU-GPU transfers
   - wait for all cluster's transfers completion
   - in each processing element, sequentially fire nodes according to their scheduling order (processing elements run simultaneously)
   - wait for all processing elements' computation to be finnished
2. with computation - communication overlap
   - launch asynchronous transfers and fire nodes simultaneously
   - wait until all previous tasks are finnished

5 Case Studies

In this section we present three experiments that were made with our tool. The first one shows its ability to implement an application on a multi-host multi-gpu cluster with computation - communication overlap. The second one deals with the implementation of a 3D ganulometry application used to study a material's properties. At last, the third one presents the implementation of an application for computing visual saliency maps.

5.1 Multi-Host Experiment

This section presents an experiment which shows the ability of our design flow to implement an application on a multi-host multi-gpu cluster with computation - communication overlap.
The cluster that was used for this experiment is composed of two hosts connected by Quad Data Rate (QDR) x4 Infiniband adapters connected on each host on a PCIe V2 X8 slot. QDR maximum throughput is 10 Gb/s per line in each direction. Thus with x4 (four lanes) adapters, the maximum throughput is 40 Gb/s (5 GB/s), in each direction. However, we measured useful data transmission rate of 2.6 GB/s in each direction, which is in accordance with the manufacturer's data. Each host has also a GTX 285 GPU board connected to a PCIe v2 x16 slot. The PCIe v2 bus offers a maximum throughput of 500 MB/s per line in each direction, leading to a maximum bandwidth of 8GB/s in each direction with 16 lanes (x16). In practice we measured useful data transmission rate of 5 GB/s in each direction.
The application that was used for this experiment consists of a four nodes DGF that exchange matrices, cf. figure 11. The P (Producer) node generates matrices that it sends to a first I (Incrementer) node that increments each element of the matrix it receives. This first incrementer node sends its result to a second incrementer node I. The matrix produced by this last node is then sent to

the C (Consumer) node that can be used to make functional checks on the results produced by the two I nodes.

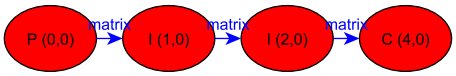

Fig. 11: Application graph of the multi-host experiment

We mapped this application on the cluster as shown in figure 12. The application, although similar to that presented in section 3, has this time been distributed over a multi-host architecture (the design flow we presented in section 4 is able to switch from these two configurations only by changing a few lines of the architecture and mapping description files).

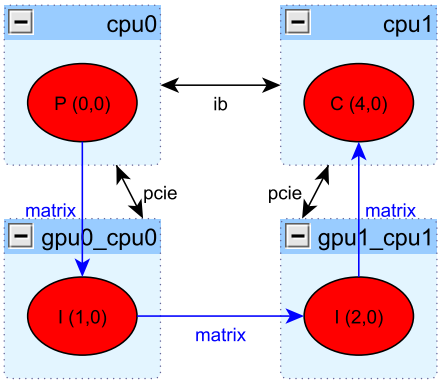

Fig. 12: Mapping used in the multi-host experiment

Recall that our goal is to demonstrate the ability of our tool to implement an application with and without computation - communication overlapping. So our approach was to define experimental conditions under which the computations time are of the same order of magnitude as those of communication. As the production time of a matrix (which can be done by the P node) and the check time of a matrix (which can be done by the C node) on a CPU is roughly one order of magnitude greater than the communication between hosts or between CPUs and their associated GPU, we have chosen not to make any processing in the P and C nodes. We therefore varied the computational load with the I nodes which are executed by the GPUs. Each incrementer node runs the GPU kernel shown in figure 13. This kernel takes as input the matrix `in` and produces as output the matrix `out`. The `width` and `height` parameters are used to specify the matrix size. The argument `nb_loop` is used to vary the kernels computation load by varying the upper bound of the following series

$$\sum_{i=2}^{i \leq nb\_loop+3} \frac{1}{i^2}$$

which converges to 0.5 and which we used to compute 1. It's not very interesting from a functional point of view, but the goal was to have a simple way to vary the computational loads on the GPUs.

```
__global__ void kernel_matrix_inc(
    float* in,
    float* out,
    int width,
    int height,
    int nb_loop) {

    unsigned int x =
        blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y =
        blockIdx.y*blockDim.y + threadIdx.y;

    float one = 0.0f;
    int i;
    for( i = 2; i < nb_loop+3; i++ ) {
        one += (1.0f/i)*(1.0f/i);
    }
    one = (float)( (int)(one + 0.5f) );

    int index = y*width+x;
    out[index] = in[index]+one;
}
```

Fig. 13: GPU kernel exucuted by the incrementers

We compared the performances of this application with and without computation - communication overlap for matrix size of 16 MB for values of `nb_loop` within the range $[0; 400]$. Figure 14 shows the allocation of buffers made by the tool in both cases with and without computation - communication overlap. In the without case, the tool instantiates 2 * 16 MB = 32 MB of memory on both the hosts (main memory) and the GPUs. In the with case the sizes of the buffers are doubled.

The "without overlapping" curve in figure 15 shows the time in ms (left ordinate scale) required to complete an iteration of the algorithm depending on the value of `nb_loop` without computation - communication overlap. The "with overlapping" curve shows the same results with computation - communication overlap. The "speedup" curve shows the speedup (right ordinate scale) which is calculated as the ratio of the curve without computation - communication overlap and with computation - communication overlap.

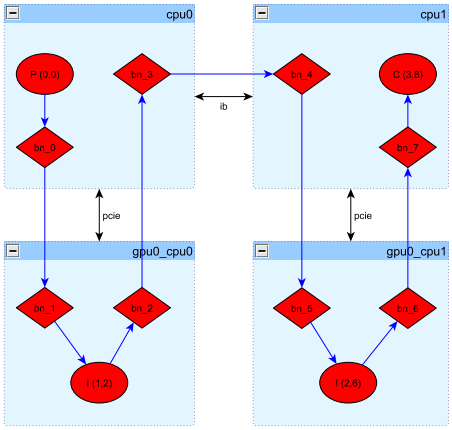

Fig. 14: Buffer allocations generated for the multi-host experiment

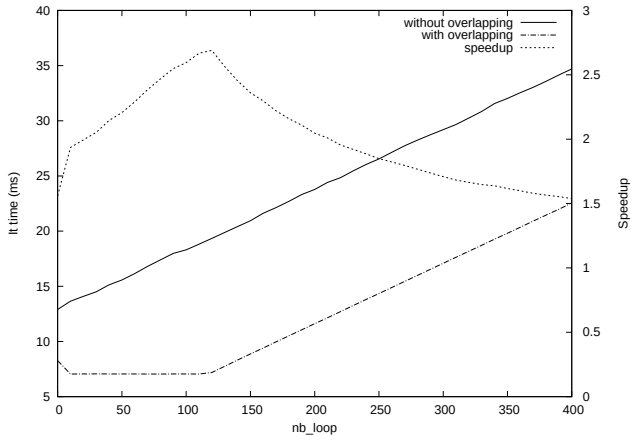

Fig. 15: Results of the multi-hosts experiment

The "without overlapping" curve is a straight line. This is what we expect since the computation and communication are serialized and since the complexity of the algorithm executed by the GPU is in $o(nb\_loop)$. More formally, if we note:

- T_{ib} the transfer time of a matrix on infiniband,
- T_h the transfers times of a matrix from CPU to GPU and from GPU to CPU (which are done in parallel),
- α the slope of the line,
- T the time of an iteration

then we can write, assuming that transfers times are greater than the time used by the tool to initiate the transfers and to launch the kernels:

$$T = T_h + T_{ib} + \alpha * nb\_loop$$

As expected the "with overlapping" curve starts with a horizontal line, which corresponds to the values of `nb_loop` for which the calculation time is shorter than the communication time. Then, when the calculation time is greater than the communication time, the "with overlapping" curve is a line parallel to the "without overlapping" curve. As the transfers and communications are parallelized, the expression of this "with overlapping" curve is:

1. if $\alpha * nb\_loop < \max(T_{ib}, T_h)$
$$T = \max(T_{ib}, T_h)$$
2. if $\alpha * nb\_loop \geq \max(T_{ib}, T_h)$
$$T = \alpha * nb\_loop$$

The expression of the speedup S, which corresponds to the "speedup" curve, is therefore:

1. if $\alpha * nb\_loop < \max(T_{ib}, T_h)$
$$S = \frac{T_{ib} + T_h + \alpha * nb\_loop}{\max(T_{ib}, T_h)}$$
It is a linear equation which has a maximum value when $\alpha * nb\_loop = \max(T_{ib}, T_h)$ equals to:
$$\frac{T_{ib} + T_h}{\max(T_{ib}, T_h)} + 1$$
We can see that the speedup is therefore at most equals to 3 if $T_{ib} = T_h$, which occurs when the computation time equals the communication time on infiniband which also equals the communication time between CPUs and GPUs.
In our case, if we note:
   - R_{ib} the data rate on the infiniband,
   - R_h the data rate in each direction between CPUs and GPUs,
   - d the matrix size

   the expression of S becomes (knowing that the data rate on the infinband is the lowest):
$$S = \frac{\frac{d}{R_{ib}} + \frac{d}{R_h}}{\frac{d}{R_{ib}}} + 1 = 2 + \frac{R_{ib}}{R_h}$$
which equals to $2 + \frac{2.6}{5} = 2.5$ with the figures given in the beginning of this subsection. This speedup is in line with the speedup curve of figure 15 which has a maximum equals to 2.7 at $nb\_loop = 120$.
2. if $\alpha * nb\_loop > \max(T_{ib}, T_h)$
$$S = \frac{T_{ib} + T_h}{\alpha * nb\_loop} + 1$$
which has a $+\infty$ asymptote equals to 1.

To conclude the experiment, we obtained results consistent with those expected. In addition, our tool allows switching from configurations with computation - communication overlap to configuration without only by changing a parameter of the program call. Similarly, it is as simple to change the mapping of the application nodes. The tool therefore simplifies and encourages the

architectural exploration. From the standpoint of performance, we have seen that the computation - communication overlapping requires a memory usage 2 times greater than without, at constant-grained computations, and that it is only interesting when the communication time is the same order of magnitude as the computation time.

5.2 Granulometry

In the previous section, we validated the portage of a functionnal application to a multi-GPU cluster which permited to validate our design flow. This section will discuss the implementation of a real-life application used in the study of materials: granulometry [28].

5.2.1 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 objects sizes. In physics, that would resemble sieving (grain sorting): the image is filtered with a series of sieves with decreasing hole sizes. A more specific goal is to define the predominant size of objects in the image.

This is done using the morphological opening operation. An opening is the combination of two mathematical morphology operators: it is an erosion followed by a dilation. These operations are filters and the mask used by these filters is called a structuring element. When performing an opening on an image, all finite element that is smaller than the structuring element disappears. Thus, the granulometry application processes an input image by computing openings with an increasing structuring element size until all objects in the volume disappear *i.e.* the volume is empty. One optimization consists in performing morphological operations with a constant structuring element size: an erosion/dilation with a structuring element of size $n \times x$ is the same as performing n successive erosions/dilations with a structuring element of size x. In the end, our granulometry application looks like Figure 16.

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 the ordinate shows the number of positive pixels left 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 (Fig. 17).

5.2.2 Implementation Graph

Based on the granulometry decomposition of Fig. 16, computing entities (nodes) are easily identified: (1) the

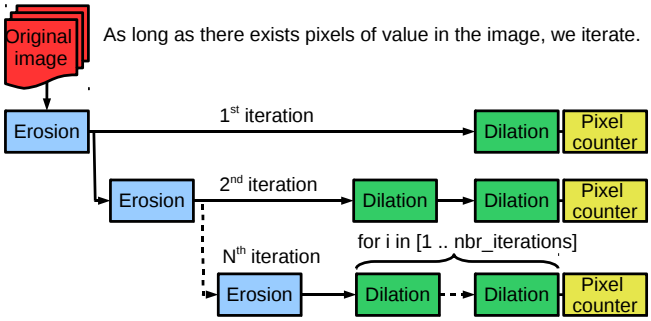

Fig. 16: Granulometry application decomposed.

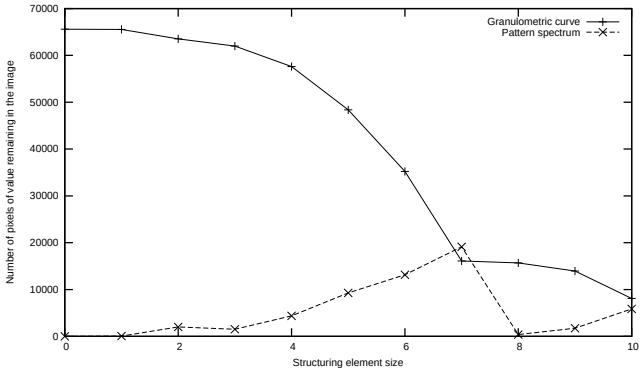

Fig. 17: Example of a granulometric curve. The pattern spectrum's extrema indicates the predominant size of the objects in the image.

CPU thread reads the image to process from hardware and (2) the GPU thread computes the openings until the image is empty. However, for a certain image size and from a certain number of openings the GPU computation time becomes preponderant to other CPU and communication times. Therefore, in order to obtain an equally distributed workload per Processing Element, we chose to use two GPUs. This will introduce three more synchronization cycles of latency in the 'without computation - communication overlap' case and one more cycle of synchronization in the 'with computation - communication overlap' case. Also, this application is not adapted for pipeline usage. Yet, it will leverage the load balancing capability of our design flow. Thus, the process can be divided into three computations / five tasks (Fig. 18 and Fig. 19):

1. CPU thread reads the image to process from hardware,
2. $CPU \rightarrow GPU$ 0 image transfer (automated by the flow),
3. GPU 0 computes the N first openings (N is fixed by the user),
4. GPU 0 $\rightarrow CPU \rightarrow GPU$ 1 intermediate result transfer (automated by the flow),
5. GPU 1 computes the M next openings.

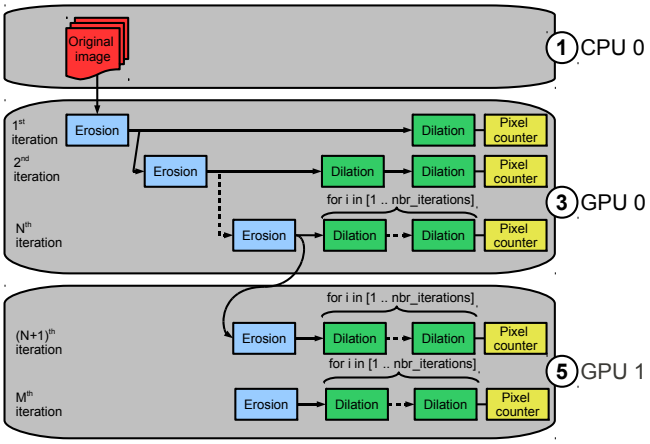

Fig. 18: Granulometry application divided in three different tasks.

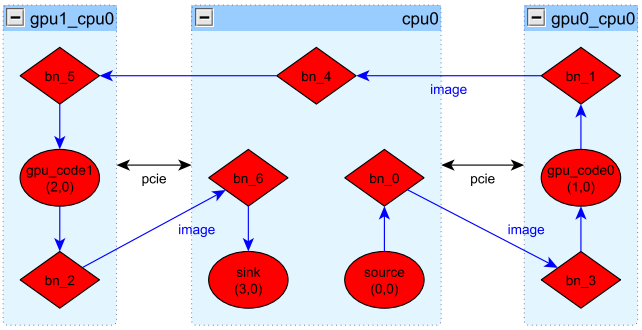

Fig. 19: Granulometry implementation graph.

5.2.3 Results

As earlier mentionned, the application has been ported on two GPUs to allow an optimal workload distribution. That is possible by tuning the workload on both GPUs by fixing the number of openings. The number of openings, the image size, its location and some other user code parameters are read at runtime from a file. Hereafter, the results for different GPU workloads with computation - communication overlap (Fig. 20). In this example, the image needs 40 openings to sieve all finite elments. The distribution of these openings on both GPUs is optimal when openings 1-28 are performed on the first gpu while the 29-40 others are performed on the second one : it shows a speedup of $\times 2.7$ thanks to an overlap between CPU and GPUs computations compared to a sequential computation.

5.3 Visual Saliency Model

This section describes the implementation of a compute intensive application related to a biologically-inspired model. Based on the primate's retina, the visual saliency

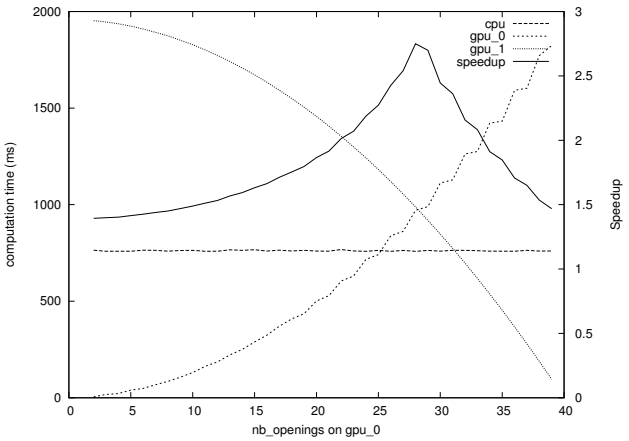

Fig. 20: Load balancing the granulometry application on two GPUs while CPU read time is constant.

model is used to locate regions of interest i.e. the capability of human vision to focus on particular places in a visual scene. We propose an implementation of this model on a multi-gpu node with computation- communication overlap.

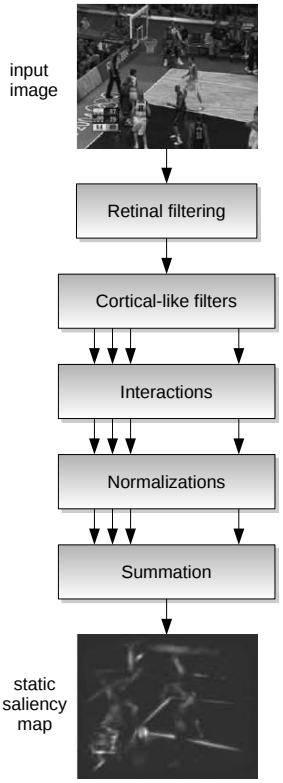

Fig. 21: Static pathway of the saliency model

5.3.1 The Model

The saliency model implemented is the one proposed by [16] as shown in figure 21. Visual information is decomposed into different spatial frequencies and orientations in the primary visual cortex using Gabor filters. Each filter G_{ij} at orientation i and at frequency j, is determined by its central radial frequency f_j and its standard deviation σ_{ij} (figure 22).

$$G_{i,j}(u,v) = exp(-\frac{(u'-f_j)^2 + v'^2}{2{\sigma_{ij}}^2})$$

$$\begin{cases} u' = u \cdot cos(\theta_i) + v \cdot sin(\theta_i) \\ v' = v \cdot cos(\theta_i) - u \cdot sin(\theta_i) \end{cases}$$

Neuron responses in the primary cortex are influenced by other neurons. Interactions occur with the same pixel in different partial maps, that reinforce objects lying in the same orientation (but different frequencies) while inhibiting those of different orientations and the similar frequency. A region is salient if it is different from its neighbors. Then, after a first normalization, each map is multiplied by a factor $(max(m_{ij}) - \overline{m_{ij}})^2$, maximum of the map minus its average squared. Finaly, all partial maps are added to obtain the static saliency map. This bottom-up approach of the static pathway is detailled in [21].

5.3.2 Algorithm Implementation

First, the input image r_im is filtered by a Hanning function to reduce intensity at the edges. In the frequency domain, cf_fim is processed with a 2-D Gabor filter bank using six orientations and four frequency bands. The 24 partial maps $cf\_maps[i,j]$ are moved in the spatial domain $c\_maps[i,j]$. Short interactions inhibit or excite the pixels, depending on the orientation and frequency band of partial maps. The resulting values are normalized between a dynamic range before applying Itti's method for normalization, and suppressing values lower than a certain threshold. Finally, all the partial maps are accumulated into a single map that is the saliency map of the static pathway (algorithm 1) [25], [26].

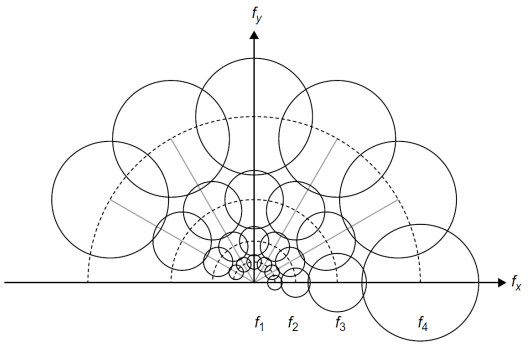

Fig. 22: Cortical like Gabor filters, set of 24 G_{ij} filters with 4 frequencies and 6 orientations

Algorithm 1 Static pathway of visual model

Input: An image r_im of size $w \cdot l$
Output: The saliency map
1: $r\_fim \leftarrow Hanningfilter(r\_im)$
2: $cf\_fim \leftarrow FFT(r\_fim)$
3: **for** $i \leftarrow 1$ **to** orientations **do**
4: **for** $j \leftarrow 1$ **to** frequencies **do**
5: $cf\_maps[i,j] \leftarrow GaborFilter(cf\_fim, i, j)$
6: $c\_maps[i,j] \leftarrow IFFT(cf\_maps[i,j])$
7: $r\_maps[i,j] \leftarrow Interactions(c\_maps[i,j])$
8: $r\_normaps[i,j] \leftarrow Normalizations(r\_maps[i,j])$
9: **end for**
10: **end for**
11: $saliency\_map \leftarrow Summation(r\_normaps[i,j])$

5.3.3 Results

Porting this application on a single gpu gives acceptable results for real-time exploitation. For a 512×512 data input, the output rate is 44 fps. However, for a 720×576 data input, it drops to 21 fps. In order to obtain a real-time (30 fps) rendering with input data dimensions bigger than 512×512 , the application has been pipelined on two GPUs. An exhaustive exploration of all possible work repartition between both GPUs brings us to chose the following optimal configuration: the first five steps (Hanning filter, FFT, Gabor filter, IFFT, Interactions) are ported on the first gpu while the last step (Normalizations) is ported onto the second gpu.

Figure 23 shows the trace look of the resulting configuration for one output (one frame). By using two GPUs, the output rate is equal to the lowest of both GPUs: here GPU 0 has the lowest one and the output rate of the application is 33 fps. Note that while GPU 0's occupation time is 100%, GPU 1's computations occupy only 55% of its time. That is the cost needed in order to implement a real-time application. The workload could have been better balanced between both GPUs if the interactions step could be devided in smaller grain. Also, it is interesting to note that the $CPU \leftrightarrow GPU$ transfer time is higher with this configuration than with one single GPU. That is because 24 partial maps are exchanged between GPU 0 and GPU 1 instead of a single map on a single GPU but since there is communication - computation overlap that doesn't affect performance. Note that the amount of allocated buffers on GPU 0 (Figure 24) is divided by two after the buffer optimization pass.

5.3.4 Discussion on data and temporal (pipeline) parallelisms

One might wonder why we implemented the visual saliency application as a pipeline: instead of transferring inter-

step	time (μs)	time (%total)
Hanning filter	136	0.60
FFT	358	1.57
Gabor filter	1446	6.33
IFFT	6645	29.09
Interactions	4127	18.07
Normalizations		
-normalize	3269	14.31
-Itti's method	3453	15.12
-normalize and fuse	3405	14.91
Total	22839	100

(a) 512×512

step	time (μs)	time (%total)
Hanning filter	225	0.47
FFT	738	1.55
Gabor filter	2276	4.77
IFFT	14908	31.23
Interactions	12510	26.21
Normalizations		
-normalize	5554	11.64
-Itti's method	5839	12.23
-normalize and fuse	5681	11.90
Total	47731	100

(b) 720×576

Table 2: Time consumption by each step of the saliency application for different input dimensions.

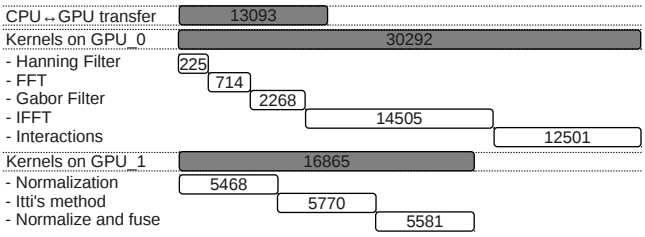

Fig. 23: Trace look (ms) for a pipelined implementation of the saliency application on a video with 720×576 pixels resolution. Transfers between devices and kernels launched on different devices occur simultaneously.

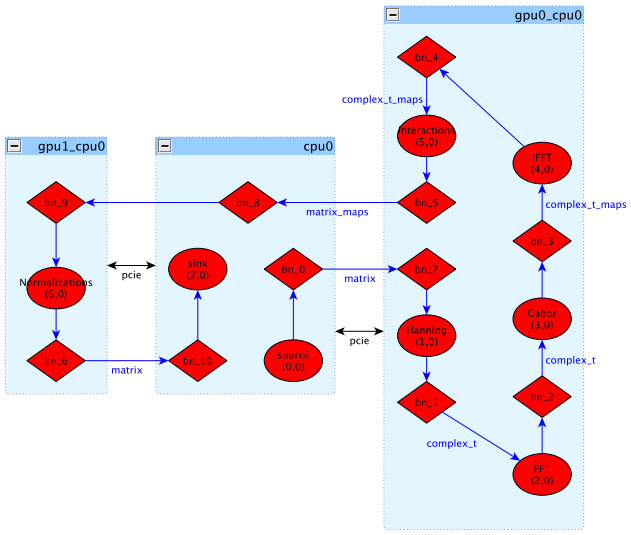

Fig. 24: Saliency implementation graph.

mediate images from one to the other GPU in the visual saliency model, we could also distribute different images to the GPUs and do the complete processing there.
This question arises when implementing any DFG application on a multi-processor architecture: is it preferable to allocate the application's nodes on the different processing elements, or to run a whole DFG instance on each GPU, each of them working on a different image?
However, pipelining the application has some more advantages. It ensures:

- the output is delivered at a constant time rate (in the case of data independent computations, which is the case of the presented visual saliency application),
- the output is delivered in the right order,
- no further code development or modifications is needed. It only needs to be split into steps dispatched onto processing elements (CPU or GPU). Intermediate transfers are hidden with computation-communication overlap.

The only disadvantage is when workload on GPUs is poorly distributed. Then, there is a loss compared to peak performance.

If data were to be split among GPUs and the complete process done there:

- the designer has to think how to split the input data into as many data sets as there are GPUs executing the application. It can't just be split in three random chunks of data. It has to be split in some kind of Round Robin manner among the GPUs,
- the amount of data per GPU may need to vary depending on the throughput of each GPU
- the frames delivered by all GPUs data sets will most probably not be delivered in the same order as the original input data,
- the output Frames Per Second won't be constant.

Thus the advantage is that the designer does not have to deal with input data sets repartition, output data reordering and eventually with output data rate balancing. In addition, some applications require a pipelined implementation, typically applications with latency constraints and which have computations with inter iterations dependencies, i.e. computation nodes which have an internal state.

6 Conclusion

This paper presents our analysis of task parallelism implementation on a multi-gpu cluster, dealing with communication - computation overlap. Using our design flow,

the programmer doesn't have to deal with inter - component communication and memory management. He doesn't waste time on basic, rudimentary but sometimes complex coding (MPI, POSIX threads, task synchronization ...) but rather focuses on the development of computation code. Thus, an application developed for a certain cluster configuration is easily portable on another platform. As shown in the previous case studies, the proposed design flow permits an easy deployment of an application onto the multi-GPU cluster thus leaving more time for exploitation of the best mapping configuration. Also, since the efforts undertaken by the GPU manufacturers to make their hardware more and more adapted to scientific computations, inter - processing elements' communication is to become the most important programming bottleneck after kernel coding. Therefore, we believe that hybrid architecture exploitation needs to be adressed with deeper interest.

References

1. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M (2012) Targeting heterogeneous architectures via macro data flow. In: Intl. Workshop on High-level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU), HiPEAC, pp 1–6
2. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M (2013) Fastflow: high-level and efficient streaming on multi-core. In: Pllana S, Xhafa F (eds) Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing, Wiley, chap 13
3. Augonnet C, Clet-Ortega J, Thibault S, Namyst R (2010) Data-Aware task scheduling on multi-accelerator based platforms. In: 2010 IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS), IEEE, pp 291–298, DOI 10.1109/ICPADS.2010.129
4. Ayguadé E, Badia RM, Igual FD, Labarta J, Mayo R, Quintana-Ortí ES (2009) An extension of the starss programming model for platforms with multiple gpus. In: Euro-Par, pp 851–862
5. Binotto AP, Pedras BM, Gotz M, Kuijper A, Pereira CE, Stork A, Fellner DW (2010) Effective dynamic scheduling on heterogeneous Multi/Manycore desktop platforms. In: 2010 22nd International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), IEEE, pp 37–42, DOI 10.1109/SBAC-PADW.2010.6
6. Chen L, Villa O, Gao GR (2011) Exploring Fine-Grained Task-Based execution on Multi-GPU systems. In: 2011 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, pp 386–394, DOI 10.1109/CLUSTER.2011.50
7. Danalis A, Marin G, McCurdy C, Meredith JS, Roth PC, Spafford K, Tipparaju V, Vetter JS (2010) The scalable heterogeneous computing (shoc) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units
8. Diamos GF, Yalamanchili S (2008) Harmony: an execution model and runtime for heterogeneous many core systems. In: Proceedings of the 17th international symposium on High performance distributed computing, ACM, New York, NY, USA, HPDC '08, pp 197–200, DOI 10.1145/1383422.1383447, URL http://doi.acm.org/10.1145/1383422.1383447
9. Dolbeau R (2007) Hmpp : A hybrid multi-core parallel. First Workshop on General Purpose Processing on Graphics Processing Units pp 1–5, URL http://www.caps-entreprise.com/upload/ckfinder/userfiles/files/caps-hmpp-gpgpu-Boston-Workshop-Oct-2007.pdf
10. Fan Z, Qiu F, Kaufman A, Yoakum-Stover S (2004) GPU cluster for high performance computing. In: Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference, IEEE, pp 47– 47, DOI 10.1109/SC.2004.26
11. Gautier T, Besseron X, Pigeon L (2007) KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: 2007 international workshop on Parallel symbolic computation, ACM, Waterloo, Canada, pp 15–23, DOI 10.1145/1278177.1278182, URL http://hal.inria.fr/hal-00684843
12. Grandpierre T, Lavarenne C, Sorel Y (1999) Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In: Proceedings of the seventh international workshop on Hardware/software codesign, ACM, New York, NY, USA, CODES '99, pp 74–78, DOI 10.1145/301177.301489, URL http://doi.acm.org/10.1145/301177.301489
13. Gropp W, Lusk E, Skjellum A (1999) Using MPI (2nd ed.): portable parallel programming with the message-passing interface. MIT Press
14. Han TD, Abdelrahman TS (2009) hicuda: a high-level directive-based language for gpu programming. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, ACM, New York, NY, USA, GPGPU-2, pp 52–61, DOI 10.1145/1513895.1513902, URL http://doi.acm.org/10.1145/1513895.1513902
15. Hsu CJ, Pino JL, Bhattacharyya SS (2008) Multithreaded simulation for synchronous dataflow graphs. In: Proceedings of the 45th annual Design Automation Conference, ACM, New York, NY, USA, DAC '08, pp 331–336, DOI 10.1145/1391469.1391553, URL http://doi.acm.org/10.1145/1391469.1391553
16. Itti L, Koch C, Niebur E (1998) A model of saliency-based visual attention for rapid scene analysis. IEEE Trans Pattern Anal Mach Intell 20:1254–1259, DOI 10.1109/34.730558

17. Karimi K, Dickson NG, Hamze F (2010) A performance comparison of cuda and opencl. CoRR abs/1005.2581
18. LAVALab (2008) Choosing between pinned and non-pinned memory. Tech. rep., URL http://www.cs.virginia.edu/∼mwb7w/cuda_support/pinned_tradeoff.html
19. Li T, Narayana VK, El-Ghazawi T (2011) A static task scheduling framework for independent tasks accelerated using a shared graphics processing unit. In: 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), IEEE, pp 88–95, DOI 10.1109/ICPADS.2011.13
20. Linderman M, Collins J, Wang H, Meng T (2008) Merge: a programming model for heterogeneous multi-core systems. In: ACM SIGOPS Operating Systems Review, vol 42, pp 287–296
21. Marat S, Ho Phuoc T, Granjon L, Guyader N, Pellerin D, Guérin-Dugué A (2009) Modelling spatio-temporal saliency to predict gaze direction for short videos. Int J Comput Vision 82:231–243, DOI 10.1007/s11263-009-0215-3
22. Membarth R, Hannig F, Teich J, Korner M, Eckert W (2011) Frameworks for GPU accelerators: A comprehensive evaluation using 2D/3D image registration. In: 2011 IEEE 9th Symposium on Application Specific Processors (SASP), IEEE, pp 78–81, DOI 10.1109/SASP.2011.5941083
23. Ospici M, Komatitsch D, Mehaut JF, Deutsch T (2011) SGPU 2: a runtime system for using of large applications on clusters of hybrid nodes. In: Second Workshop on Hybrid Multi-core Computing, held in conjunction with HiPC 2011, Bangalore, India
24. Ou Y, Chen H, Lai L (2011) A dynamic load balance on GPU cluster for fork-join search. In: 2011 IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS), IEEE, pp 592–596, DOI 10.1109/CCIS.2011.6045138
25. Rahman A, Houzet D, Pellerin D, Marat S, Guyader N (2010) Parallel implementation of a spatio-temporal visual saliency model. Journal of Real-Time Image Processing 6 special issue(1):3–14, DOI 10.1007/s11554-010-0164-7, département Images et Signal
26. Rahman A, Houzet D, Pellerin D (2011) Visual Saliency Model on Multi-GPU. In: GPU Computing Gems Emerald Edition, Elsevier, pp 451–472
27. Sbîrlea A, Zou Y, Budimlíc Z, Cong J, Sarkar V (2012) Mapping a data-flow programming model onto heterogeneous platforms. In: Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems, ACM, New York, NY, USA, LCTES '12, pp 61–70, DOI 10.1145/2248418.2248428, URL http://doi.acm.org/10.1145/2248418.2248428
28. Serra J (1983) Image Analysis and Mathematical Morphology. Academic Press, Inc., Orlando, FL, USA
29. Stefanov T, Zissulescu C, Turjan A, Kienhuis B, Deprettere E (2004) System design using kahn process networks: The compaan/laura approach. In: Proceedings of the conference on Design, automation and test in Europe - Volume 1, IEEE Computer Society, Washington, DC, USA, DATE '04, pp 10,340–, URL http://www.compaandesign.com/
30. Wolfe M (2010) Implementing the pgi accelerator model. In: Kaeli DR, Leeser M (eds) Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010, Pittsburgh, Pennsylvania, USA, March 14, 2010, ACM, ACM International Conference Proceeding Series, vol 425, pp 43–50, DOI http://doi.acm.org/10.1145/1735688.1735697