Task migration of DSP application specified with a DFG and implemented with the BSP computing model on a CPU-GPU cluster

Farouk Mansouri, Sylvain Huet, Vincent Fristot, Dominique Houzet

Task migration of DSP application specified with a DFG and implemented with the BSP computing model on a CPU-GPU cluster

Farouk Mansouri, Sylvain Huet, Vincent Fristot, Dominique Houzet
GIPSA-lab
UMR 5216 CNRS/INPG/UJF/Universite Stendhal
F-38402 GRENOBLE CEDEX, France
Email: firstname.lastname@gipsa-lab.grenoble-inp.fr

***Abstract*—Nowadays computer applications are becoming heavier and require, at the same time, real-time results. The Heterogeneous clusters with their computing power represent a good solution to this request. However, it is possible that during the execution, a computing element of the cluster becomes defaulting, needs maintenance, or that the load needs to be re-balanced...**

In this paper, we propose a migration strategy for relocating the execution of a task to another computing element. In particular, we are interested in remap nodes of Data Flow Graph (DFG), representing Digital Signal Processing (DSP) application, onto heterogeneous (CPU-GPU) clusters while keeping up the flow of data and minimizing the temporal perturbation. For our approach, we give a lower bound for the flow of data after the migration and, validate it by the real-time construction of visual saliency map from video input.

I. Introduction

With the evolution of technology, computer programs treat data increasingly large. In addition, many of them have real-time constraints and must ensure a response time less than a critical tolerated threshold. For example, the treatment of high definition video capture or the telecommunication applications for large public. To this demands, the heterogeneous cluster for High Performance Computing (HPC) represent a good response. In fact, this type of architecture develops high processing power, taking advantage of the parallelism and distribution of calculations on the many cluster nodes. The best example is Titan, the most powerful supercomputer today according to Top500 list. However, the real challenge for these machines is to achieve the high performance offered by them while facilitating their use. Indeed, during the execution of an application on a heterogeneous (CPU-GPU) cluster, computing elements may need to be stopped for reasons of failure, maintenance or to simply re-balance the load, therefore, it is necessary to relocate tasks that are executed to another computing element respecting the constraints of the application.

In this paper, we propose an approach for the migration of executing task from its original compute node to another, minimizing the impact on the response time of the executed program. More precisely, we are interested in remap tasks of DSP application specified as a DFG, executed on heterogeneous CPU-GPU cluster, minimizing the reduction of output data rate in order to satisfy the real-time constraints of the application. After presenting the related work in section II, we describe in section III the implementation context of our migration approach. In section IV, we present the principle of our approach, the elements of its implementation and analyze its impact on the output data rate. We validate it, in the last section V, on a real world case study of visual saliency map.

II. Related Work

Since the emergence of accelerators such as GPU, Cell or Xeon Phi, the HPC architectures become hybrid. The challenge has been to develop software tools allowing an efficient exploitation while easing their use.

While these architectures can be programmed using languages or Application programming interface (API) such as (1) Compute Unified Device Architecture (CUDA) or Open Computing Library (OpenCL) to program the GPUs (2) OpenCL, pthread, OpenMP to program the CPUs and (3) Message Passing Interface (MPI) to handle the communications and synchronizations in distributed memory architectures. Relying only on them is difficult. Indeed, in this case, the programmer has to handle the memory allocations and optimizations, to handle the inter-processing element's communications and synchronizations and to allocate the tasks on the processing elements. Programming with such an approach is complex and usually leads to unscalable programs. Several runtimes have been proposed to free the programmer of these tasks. Hereafter are some of them.

As stated before, communications between processing units introduce more complexity to the programmer since it implies 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 developed in order to abstract the communication layer to the programmer [1]–[4].

Some frameworks focus on the scheduling issues: [5]–[7] proposed strategies to schedule pool of tasks on a multi-GPU platform.

We also wish to highlight StarPU [1] which is a runtime system designed to dynamically schedule a pool of tasks on heterogeneous cores with various scheduling policies. It also avoids unnecessary data transfer thanks to a Virtual Shared Memory (VSM) software.

Most presented solutions focus on the exploitation of hybrid architectures for data parallelism purposes (using a Map-Reduce scheme) or dispatching of independent tasks. Our

work focuses on a more specific scenario: the implementation of DSP applications on a multi-GPU cluster. Our goal is to provide a runtime with an entry point adapted to the specification of DSP applications allowing on one hand to implement them on heterogeneous architectures (CPU-GPU) and on the other hand to migrate tasks between the processing elements for the previously mentioned reasons.
Our approach is in line with those proposed in SynDex [8] or PREESM [9]. These approaches generate an optimized implementation of an application specified with a directed acyclic hyper-graph on an architecture specified with a directed graph. But to our knowledge it is not possible to target clusters containing GPU and to do task migration with these tools.
We can also cite [10] which deals with task migration in the Bulk Synchronous Parallel (BSP) computing model context but, their work don't deal with accelerators (GPU) and has been validated only with a simulator.

III. The runtime context

In this section, we describe the runtime representing the implementation context of our task-migration approach. Developed by our team under BSP concept [11], which provides a conceptual bridge between the physical implementation of the parallel machines and the abstraction available to a programmer. This runtime allows the implementation of DSP applications on CPU-GPU cluster with computation communication overlap. First, we describe the DFG model featured DSP algorithm. Subsequently, we expose the design flow making the deployment of this algorithm on heterogeneous cluster. Finally, we explain the runtime execution.

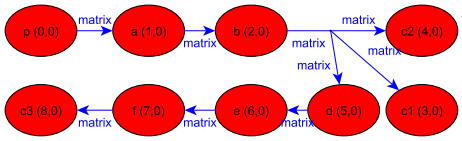

Fig. 1: Data Flow Graph of application

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. Also, 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. Thus, several studies have introduced the use of DFG for multi-GPU and/or multi-core workstations [12]–[16]. As shown in the Figure 1, the DSP application is specified with nodes (kernels), representing the computations, and edges, showing the data dependencies between nodes. The semantic of a DFG is as follows: a node can be executed if and only if all its inputs are available. When executed, it consumes all it inputs, calls the function code it is associated with, and produces all it outputs. A data type is associated to each edge and a function to each node. In the Figure 1, 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.

To deploy the DFG application on a cluster, our team proposed in their precedent's works the following design flow.

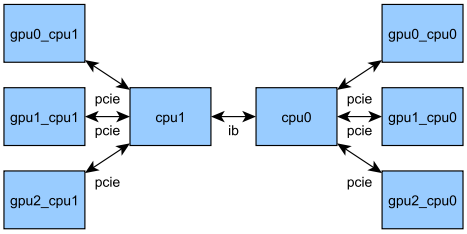

Fig. 2: Architecture Graph of cluster

First, the DFG application is mapped on the architecture graph (AG) shown in the Figure 2 which represents the heterogeneous computing elements linked by high throughput communication interfaces. Second, schedule the execution of kernels mapped on each computing element according to the user strategy. Finally, introduce buffers interface between kernel nodes to permit the communication through the architecture. This step produces the Implementation Graph (IG) shown in the Figure 3, that will be used by each computing element to determine what it has to do at runtime.

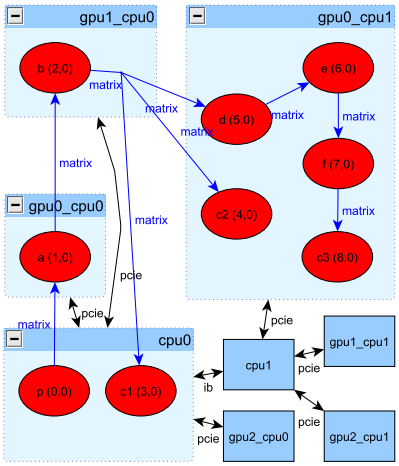

Fig. 3: Implementation Graph (IG)

At runtime, a CPU thread is associated to each processing element composing the Architecture Graph (AG). These threads manage the nodes mapped on the processing elements that are associated as inscribed in IG. Then, the runtime executes iteratively the following sequence, according to the communication-computation overlap principle: (1) launch the asynchronous transfers (2) execute each kernel with respect to the scheduling (3) wait until all the nodes mapped on all processing elements have finished their execution, and the asynchronous transfers have also completed.

IV. Task migration

Our goal is to relocate during the runtime of application, the execution of a kernel representing a node in a DFG model from its native computing element to another computing element, in a real-time conditions, i.e. in a deterministic time. In the first subsection, we present the migration strategy selected to minimize the impact of the migration on the output data flow. In the second subsection, we detail its implementation

through an example and we give the algorithm. The last subsection will be devoted to the mathematical modeling of the execution time and the throughput of data engendered by our migration approach.

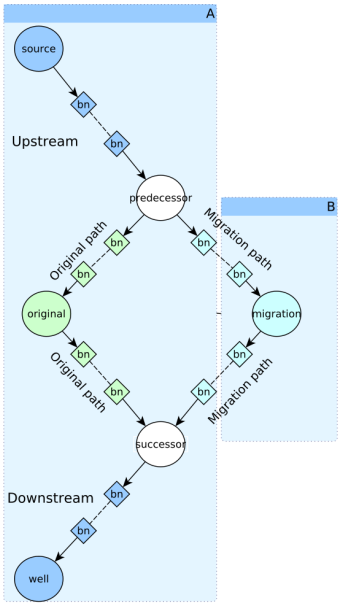

Fig. 4: Migration of the data flow graph (DFG) application mapped on cluster

A. The migration strategies

Several migration strategies can be proposed, but do not answer our goal that is to minimize the impact of the migration on the output data flow. For example, to proceed the migration in the Figure 4, we can stop the execution of all kernels of application, transfering the data contained in the buffers of original path to the buffers of destination path and resume the execution after all data was copied. The drawback of this approach is that the system will be halted while copying the data from original to the migrated path. This time depends on the data size to move and the data link speed. Therefore, this strategy can interrupt the flow during a long time. Another approach consists in leaving the data in original path and regenerates them once again on the destination path. This approach has many drawbacks: some computations are done twice and some data continuously generated data by sensors or video camera or intermediate data may not be available. For those reasons, this second strategy is not also a good response for our objective.

The idea is to stop the flow at minimum as possible. The strategy that we propose is based on this principle. It consists in feeding progressively the flow to the destination path and, in the same time, to empty the data flow residue in the original path. However, the two paths may have different size, and therefore require different time to fill and empty. In this case, synchronization is necessary for maintaining the integrity of the data stream.

B. Strategy of emptying and filling in the same time

Concretely, the strategy of emptying and filling in the same time is to retrieve the data contained in the buffers of the original path and evacuate them to the downstream path, parallel and in the same time, redirect data produced by the upstream path to the buffers of migration path, so, the data stream is preserved. In what follows, we explain how this strategy is implemented in the context of our runtime.

At the time of migration, at the end of an iteration, the first step is the instantiation of a new node in the implementation graph (IG), identical to that we wish to migrate. It is mapped on the destination computing element, and connected by the edges to the predecessor and the successor of the original node as shown in the Figure 4. Subsequently, the buffers are introduced to complete the construction of the migration path. The second step is to program the original path and the migration path in the IG of application, to empty and fill gradually the data flow in parallel. To do this, we compute a couple of weights on the edges of each path that represents their start time and stop time. These two weights are checked and updated at each iteration for allowing or not the calculations and the data transfer on each edge. Thus, to gradually empty the original path simply set the weights representing the stop time increasing manner. And, to gradually fill the migration path, set the weights representing the start time of increasing manner as well. In this way, during the next iterations, both paths will empty and fill gradually and in parallel. However, the size of the paths may be different and can require a different number of execution's iterations to be crossed. Indeed, according to the processing element where the migrated node is instantiated, the number of interface buffers may increase or decrease compared to the number of buffers present in the original path. In this case, it is necessary to add one step to synchronize production and consumption of data in order to preserve the integrity of the stream. Concretely, it must reprogram the data transfer on the upstream path and the downstream path. We distinguish three scenarios that we discuss in what follows.

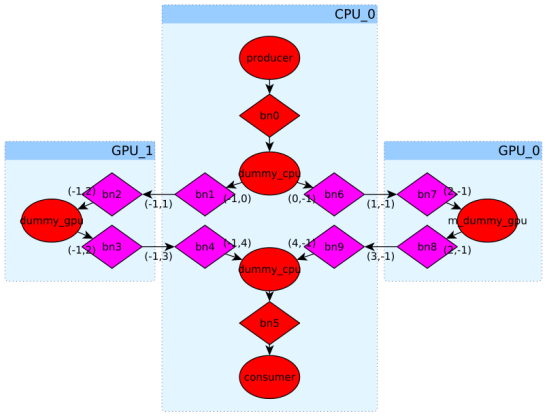

Fig. 5: Updated implementation graph (IG) with equal migration path

1) Original and destination paths are equal: This scenario, shown in Figure 5, is the simplest because the migration path is equal to the original path. In fact, a data unit requires 3 iterations for completely pass through

both paths. Therefore, to migrate the task ($dummy - gpu$) we just have to redirect the produced data stream in upstream path ($Producer, bn0, dummy - cpu$) toward the migration path ($dummy - cpu, bn6, bn7, m - dummy - gpu, bn8, bn9, dummy - cpu$), and then program its start-time weight increasingly, and the stop-time weight of the original path ($dummy - cpu, bn1, bn2, dummy - gpu, bn3, bn4, dummy - cpu$) increasingly as well. Finally, we compute the necessary number of iterations to switch between the data-flow of the original path and the migrated path, incoming to downstream path ($dummy - cpu, bn5, consumer$). In the example that is equal to 3.

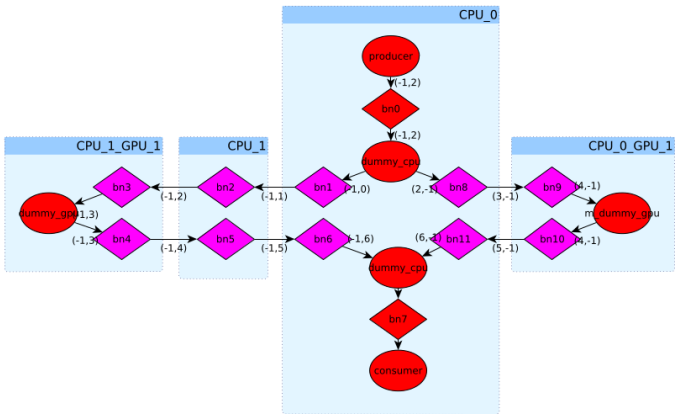

Fig. 6: Updated implementation graph (IG) with shorter migration path

2) Original path is longer than destination path: In this case, shown in Figure 6, the original path ($dummy - cpu, bn1, bn2, bn3, dummy - gpu, bn4, bn5, bn6, dummy - cpu$) is longer than the migration path ($dummy - cpu, bn8, bn9, m - dummy - gpu, bn10, bn11, dummy - cpu$). Indeed, a given data unit requires 5 iterations to cross the original path, while on the migration path, it requires only 3 iterations. Therefore, the flow in the upstream path ($Producer, bn0, dummy - cpu$) must stop during this difference of number of iterations to allow synchronization between the arrival of the data on the migration path and the emptying of the data flow from the original path, which corresponds to 2 iterations in the example. To perform this, we set the value of the difference in length as stop-time weight on the upstream path, i.e. 2 in the example. The original path should also be gradually emptied that is done by setting its stop-time weights increasingly, $(0, 1, 2, 3, 3, 4, 5, 6)$ in the example and by setting the start-time weights in the migration path increasingly, which corresponds to $(2, 3, 4, 4, 5, 6)$ in the example. Finally, the flow switching between the original path and the migrated path will occur when all the data on the original path are consumed, and in the same time, when the first data arrives at the end of the migration path. In the example, it is 5 iterations after the beginning of the migration. That way, the downstream path ($dummy - cpu, bn7, consumer$) is continuously fed.

3) Original path is shorter than destination path: In the latter case, shown in Figure 7, we present the application of our approach if the migration path is longer than the original path, which means that data unit requires more iterations to circulate on the migration path ($dummy - cpu, bn6, bn7, bn8, dummy - gpu, bn9, bn10, bn11, dummy - cpu$) than on the original path ($dummy - cpu, bn1, bn2, dummy - gpu, bn3, bn4, dummy - cpu$), respectively 5 iterations for the migration path and 3 for the original path. Consequently, it is necessary to stop the flow in the downstream path ($dummy - cpu, bn7, consumer$) for a number of iterations representing the difference in length between the two paths, 2 in the example, but only after the migration path is filled, i.e. 3 iterations in the example. Concretely, it is necessary to initialize the stop time weights of the original path increasingly $(0, 1, 2, 2, 3, 4)$ and the start time weight of the migration path in the same way $(0, 1, 2, 3, 3, 4, 5, 6)$. The start time weight of the downstream path has also to be increasingly initialized from the number of iterations necessary for the flow to come to the end of the migration path $(6, ...)$ and the weight of the stop time from the number of iteration necessary for the flow come to the end of the migration path $(4, ...)$, which will in the example, stop the flow in the downstream path during 2 iterations at 4th iteration time, then restarting after.

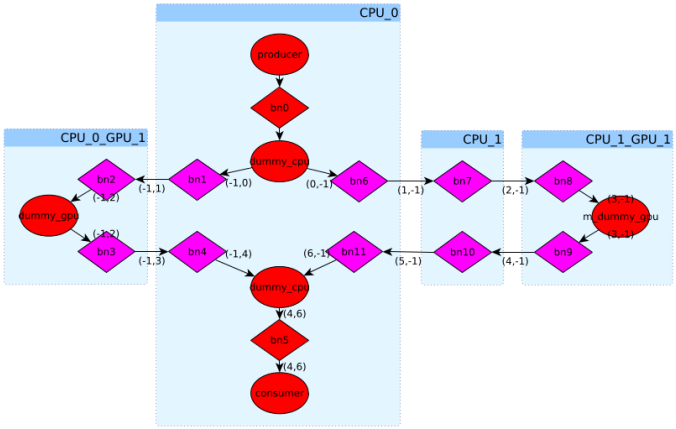

Fig. 7: Updated implementation graph (IG) with longer migration path

All these steps of our task-migration approach implementing are resumed in the algorithm 1.

Algorithm 1 Task migration

Input: Implementation graph (IG). Destination computing element (CE_dest). Kernel to migrate (Ker).
Output: Updated implementation graph (IG')

$org \leftarrow Research(IG, Ker)$
$mig \leftarrow Copy(org)$
$IG \leftarrow Mapping(mig, CE\_dest)$
$IG \leftarrow Buffers\_introduce(Pred(org), mig)$
$IG \leftarrow Buffers\_introduce(mig, Succ(org))$
$IG \leftarrow Update\_weight(Original\_path)$
$IG \leftarrow Update\_weight(Migration\_path)$
if $Length(Orig\_path) > Length(Mig\_path)$ **then**
    $IG \leftarrow Update\_weight(Upstream\_path)$
end if
if $Length(Orig\_path) < Length(Migr\_path)$ **then**
    $IG \leftarrow Update\_weight(Downstream\_path)$
end if
$IG' \leftarrow IG$

C. Modeling and analysis

In this section, we propose a mathematical model to describe our migration approach and predict its impact on the outflow data. As described in the section III, our runtime runs in the form of repetition of iterations. In one iteration, all computing elements execute in parallel the tasks of the DFG application mapped on it, and transfer the results to others computing elements through the buffers with communication computation overlap. We describe the execution of our runtime in time as follows:

$$Time = \sum_{i=1}^{n} (IT(i))$$

- $IT(i)$: Time of executing the iteration number i.
- n : Total number of iterations.

Knowing that each runtime iteration consumes one unit of input data and produces one unit of output data; we can describe the rate of outflow through time as follows:

$$D(i) = \frac{1}{IT(i)}$$

Therefore, the minimum throughput of outflow is :

$$D_{min}(i) \geq \frac{1}{\max\limits_{i \in 1,n}(IT(i))}$$

In the case of migration, the description of the overall runtime execution time is represented by the equation below in three parts. A first part before the migration where the runtime executes the initial graph DFG. A part during migration: the runtime empties the original path and fills the migration path in parallel. And third part after the migration, where the runtime executes the updated DFG graph.

$$Time_{mig} = \sum_{i=1}^{a-1} (IT_{org}(i)) + \sum_{i=a}^{a+b} \max(IT_{org}(i), IT_{mig}(i)) + \sum_{i=a+b+1}^{n} (IT_{mig}(i))$$

- $IT_{mig}(i)$: Time of iteration on migration graph.
- $IT_{org}(i)$: Time of iteration on original graph.
- a : Iteration of starting the migration process.
- b : Number of iterations representing the biggest length between the migration path and the original path.

However, to describe the outflow for our migration approach, we distinguish two cases according to their repercussion on the output data rate: The first case, where the output stream is not stopped during the iterations, i.e. each iteration produces output data unit. It is the case of a task-migration with a destination path equal to or shorter than the original path. Indeed, since the downstream path is not stopped, the flow of output data is given as one data unit by iteration. Consequently, the minimum rate of outflow is described by the following equation:

$$D_{min} \geq \frac{1}{\max\limits_{i \in 1,n}(IT_{org}(i), IT_{mig}(i))}$$

The second case, where the output stream is delaying during a few iterations in task-migration process with the destination path longer than the original path. Here, the flow of output data is not always equal to a unit by iteration because in the time of migration, the downstream path is stopped during a number of iterations equal to the difference in length between the original path and the migration path. Consequently, the minimum rate of outflow is described in the following equation:

$$R_{min} \geq \frac{1}{\sum_{i=a+e}^{a+e+k} \max(IT_{org}(i), IT_{mig}(i))} \geq \frac{1}{k \max\limits_{i \in 1,n}(IT_{org}(i), IT_{mig}(i))}$$

- e : Length of the original path in iterations.
- k : Number of iteration representing the length difference between the original path and the migration path.

Based on the proposed model, we predict in any case the impact of our migration process on the response time of the execution. Indeed, the lower bounds presented in the previous equations allowing to measure the worst case of reduction of output data flow.

V. Case study: saliency map

The goal of our work is to migrate the execution of a task of a DFG application between nodes of cluster. In the precedents subsections, we proposed an approach of migration, described its implementation and gave a mathematical model which can be used to predict its impact. In what follows, we apply our approach for the construction of visual saliency map in order to validate our work under real world case. First, we describe the visual saliency map application and we present its algorithm. Second, we show the DFG modeling of this algorithm and its implementation on a heterogeneous CPU-GPU cluster. Third, we present the results of task-migration with our approach, and analyze its impact on the output data flow.

A. The visual saliency model

Based on the primate's retina, the visual saliency model is used to locate regions of interest, i.e. the capability of human vision to focus on particular places in a visual scene. The implementation that we use is the one proposed by [17] as shown in Figure 8. His algorithm (Algorithm 2) is: 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

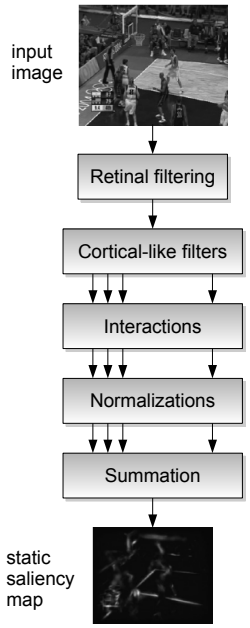

Fig. 8: The static pathway of the visual saliency application

Algorithm 2 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: $saliency\_map \leftarrow Summation(r\_normaps[i,j])$

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.

B. The DFG visual saliency mapping on CPU-GPU cluster

As described in last subsection, the visual saliency model is a DSP application represented as sequence of successive processing kernels implemented by nodes in the DFG model ($Capture, Hanningfilter, FFT, GaborFilter, IFFT, Interactions, Normalizations, Display$). Thus, the kernel is a part of code of application that processes input data and produces output data, a frame in our case. In our application, the $Capture$ and $Display$ kernels are mapped on CPUs, while the others are mapped on GPUs. We previously show that this application can be implemented in real-time on our CPU-GPU cluster (Figure 9) for a video resolution of 512×512 pixels. Based on this work, we experiment our migration approach to remap the execution of this saliency application kernels on different computing elements of the cluster, and we analyze its impact on the video outflow. The cluster that we use consists of two nodes, A and B shown in the Figure 9 connected by Infiniband link. Each node contains a 8-core CPU Intel(R) i7 and two Graphics Processing Unit (GPU), GeForce GTX 285 and Quadro 4000.

To estimate the real impact of our approach, we propose three scenarios of kernel-migration experimented on the saliency application described before. In these scenarios, we maintain all the application kernels as originally mapped in Figure 9, and move only the kernel $(gpu - normalize)$. In the first scenario, we experiment the case where the original and the migration paths are equals, i.e. data needs the same number of iterations for running through them. We migrate at iteration 25 the kernel $(gpu - normalize)$ mapped initially on computing element (B-GPU0) to (B-GPU1). In the second scenario we experiment the case where the migration path is shorter than the original path, i.e. data need fewer number of iterations for running through the destination path. For this, we migrate the kernel $(gpu - normalize)$ from (B-GPU0) to (A-GPU0). In the last scenario, to obtain a migration path longer than the original path we migrate the kernel $(gpu - normalize)$ from (A-GPU0) to (B-GPU0). In what follows, we present the obtained results for the output flow and the execution time of iterations. We also explain and analyze the impact of migration on them in each scenario.

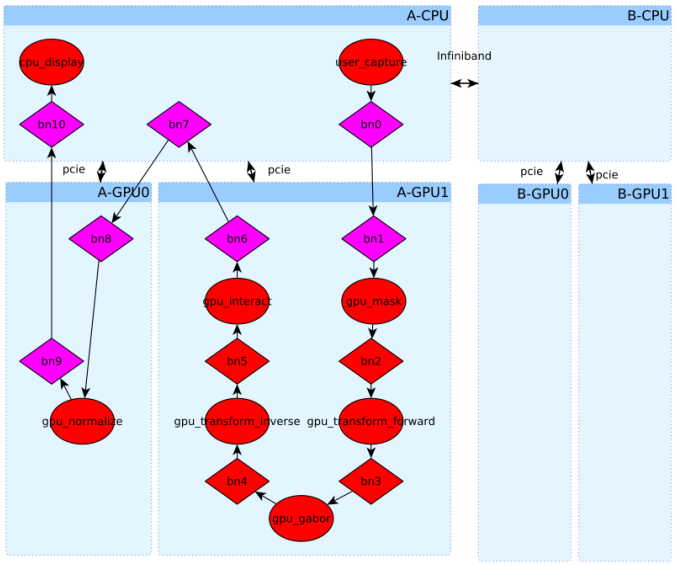

Fig. 9: Implementation graph of visual saliency application

C. Results

In this section, we present for each scenario two measures in one Figure: The evolution in each iteration of its execution time in millisecond represented by green curve and the throughput of output video (FPS) with the blue curve. In the same figure, we show also for each iteration if an output frame is produced or not. The production of a new frame is represented by a green star on the green curve. The output FPS depends on the production or not by each iteration of an output frame, and of iteration executing time. In that

follow, we analyze and explain the variations induced by our migration process on these measures and compare them with the predictions given by equations in our model (section IV).

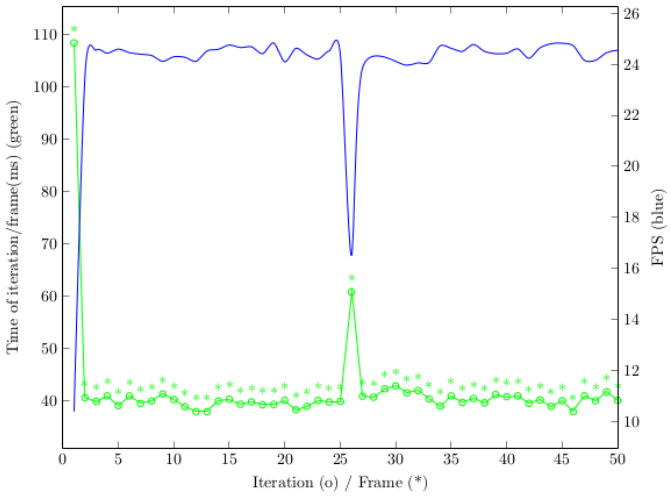

Fig. 10: Throughput and time of execution per iteration

1) Equal migration path: In this case, the results are shown in Figure 10 where the migration begins at iteration 26 and takes 7 iterations. We distinguish three parts:

Before the migration until iterations 25, the runtime executes iteratively the original implementation graph (IG) in constant time nearly of 40 ms per iteration (Green curve). Except in the first iteration, where it constructs the graph model and allocates the buffers. Also, in this step of execution, each iteration consumes one frame from input video and produces also one frame. Thus, the throughput of outflow video (Blue curve) depends only on executing time of the iteration, and it is relatively constant (25 FPS) that corresponds to our prediction.

During the migration between iterations 26 and 32, the migration process initiates first at the iteration 26, the construction of migration path. It allocates its buffers and programs the filling and emptying of data. Therefore, the time of this iteration is extended, shown by a peak in the green curve up to 60 ms. The iterations (26,27,..,32) fill and empty in parallel the data without extra time. So the time of iteration equals the longer execution between the original path and the migration path as predicted in our model. In this step, one frame is produced at each iteration. Thus, the throughput of outflow depends only on iteration time. It is relatively constant (25 FPS), except at iteration 26 where it decreases to 17 FPS that is the minimum given by model.

After the migration from the iteration 33, the runtime executes the updated implementation graph (IG) in constant time nearly of 40 ms per iteration (Green curve). Each iteration produces one frame. Therefore, the throughput of output flow depends only on executing time.

2) Shorter migration path: In this scenario represented by the Figure 11, the migration process occurs also at iteration 26 and takes 7 iterations. the difference in length between original and migration path is 2 iterations. We distinguish three parts of execution:

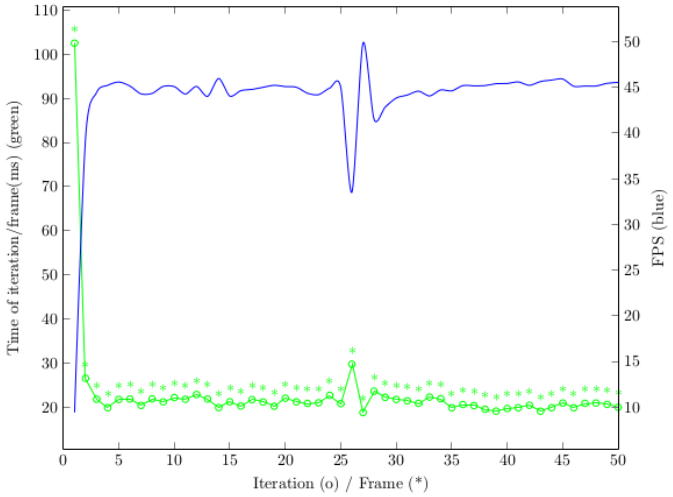

Fig. 11: Throughput and time of execution per iteration

Before the migration ($it < 26$): In this step, we see the same behavior as in the first scenario. The time of iteration (green curve) is nearly of 20 ms except for the first iteration. The throughput of the output flow depends on it and is also constant with a value of nearly 45 FPS.

During the migration ($25 < it < 33$): At iterations 26 and 27, the migration process initiates first the update of the implementation graph (IG) and empties the original path without filling the migration path because it is shorter. It extends the time of iteration 26 to 30 ms, but reduced the time of iteration 27 to 18 ms. In the rest of iterations (28,29,30,31,32), the runtime fills and empties in the same time, which stabilizes progressively the green curve at 20 ms. In this scenario, the output flow doesn't stop during iterations. The throughput of output flow depends only of the time of iteration. The minimum is 35 FPS as expected by proposed model in section IV.

After the migration ($it > 32$): In this step, the runtime executes the updated implementation graph (IG) with constant time in each iteration (20 ms in Green curve). Also, all iterations produce a frame as shown in Figure (Green stars). Therefore, the throughput of output flow is constant at 45 FPS.

3) Longer migration path: The last scenario shown in the Figure 12 represents the extension of migration path of 3 iterations. The migration occurs at iteration 26 and lasts 5 iterations. The evolution of results is also characterized by the three parts:

Before the migration ($it < 26$): The runtime executes the original implementation graph (IG) in constant time nearly of 30 ms per iteration (Green curve). In the first iteration, it constructs the graph model and allocates the buffers which extend time of execution. Also, in this step, each iteration consumes one frame from input video and produces also one frame (Green stars). Thus, the throughput of outflow video (Blue curve) depends only of executing time of the iteration, and it is relatively constant (50 FPS).

During the migration ($25 < it < 32$): At iterations 26 and 27, the migration process initiates in the first one the update of

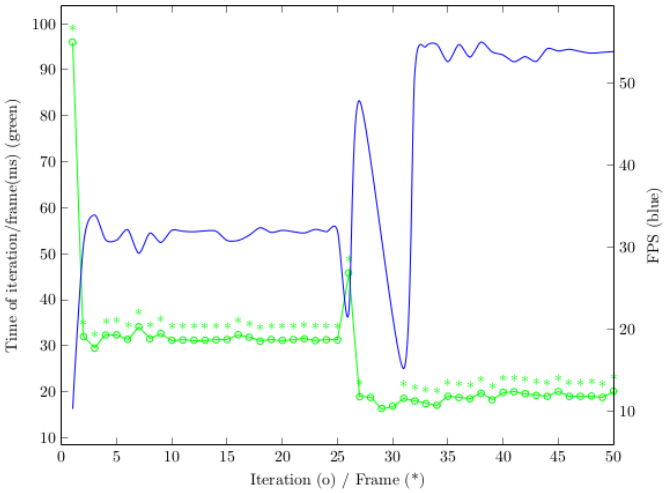

Fig. 12: Throughput and time of execution per iteration

the IG, and it fills the migration path and empty the original path. Thus, time of iteration 26 is extended to 47 ms, while in iteration 27 is preserved to 20 ms. The output flow doesn't stop during this two iterations and the throughput depends only of time. However, in the rest of iterations (28,29,30) the downstream path is stopped during 3 to synchronize the stream. The output frames are not produced during this time, and generate an outflow throughput reduction to minimum 15 FPS, even if the times of iterations are in 20 ms. This comportment is also predicted by our proposed model in section IV.

After the migration ($it > 31$): In this step, the runtime executes the updated implementation graph IG. The time (green curve) is stabilized at nearly 20 ms which produces a throughput (Blue curve) with 50 FPS because it depends only on it.

VI. Conclusion

This paper presents our approach of task migration between nodes of CPU-GPU cluster under real-time constraints. We provided the following contributions. We proposed a strategy to migrate a kernel of DFG application that minimizes the reduction of the data flow rate. Its implementation permits its achievement for parallel and distributed execution of DFG on CPU-GPU cluster. We also proposed a mathematical model to predict the impact of our migration approach on the output data stream. Finally, the migration process was experimented on a real-time saliency map construction of an input video and we shown the repercussion of the migration on the output video rate.

References

[1] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-Aware task scheduling on multi-accelerator based platforms," in *2010 IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, Dec. 2010, pp. 291–298.

[2] M. Ospici, D. Komatitsch, J.-F. Mehaut, and T. Deutsch, "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, dec 2011.

[3] M. Linderman, J. Collins, H. Wang, and T. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ACM SIGOPS Operating Systems Review*, vol. 42, 2008, pp. 287–296.

[4] G. F. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *Proceedings of the 17th international symposium on High performance distributed computing*, ser. HPDC '08. New York, NY, USA: ACM, 2008, pp. 197–200. [Online]. Available: http://doi.acm.org/10.1145/1383422.1383447

[5] A. P. Binotto, B. M. Pedras, M. Gotz, A. Kuijper, C. E. Pereira, A. Stork, and D. W. Fellner, "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, Oct. 2010, pp. 37–42.

[6] L. Chen, O. Villa, and G. R. Gao, "Exploring Fine-Grained Task-Based execution on Multi-GPU systems," in *2011 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Sep. 2011, pp. 386–394.

[7] Y. Ou, H. Chen, and L. Lai, "A dynamic load balance on GPU cluster for fork-join search," in *2011 IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS)*. IEEE, Sep. 2011, pp. 592–596.

[8] T. Grandpierre, C. Lavarenne, and Y. Sorel, "Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors," in *Proceedings of the seventh international workshop on Hardware/software codesign*, ser. CODES '99. New York, NY, USA: ACM, 1999, pp. 74–78. [Online]. Available: http://doi.acm.org/10.1145/301177.301489

[9] M. Pelcat, J. Piat, M. Wipliez, S. Aridhi, and J.-F. Nezan, "An open framework for rapid prototyping of signal processing applications," *EURASIP J. Embedded Syst.*, vol. 2009, pp. 11:3–11:3, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1155/2009/598529

[10] R. da Rosa Righi, L. Pilla, A. Carissimi, P. Navaux, and H.-U. Heiss, "Migbsp: A novel migration model for bulk-synchronous parallel processes rescheduling," in *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, 2009, pp. 585–590.

[11] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[12] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, "Targeting distributed systems in fastflow," in *Euro-Par Workshops*, 2012, pp. 47–56.

[13] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *in Programming Multi-core and Many-core Computing Systems, ser. Parallel and Distributed Computing, S. Pllana*, 2012, p. 13.

[14] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the starss programming model for platforms with multiple gpus," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 851–862. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03869-3_79

[15] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multiprocessors," in *2007 international workshop on Parallel symbolic computation*. Waterloo, Canada: ACM, 2007, pp. 15–23. [Online]. Available: http://hal.inria.fr/hal-00684843

[16] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms," *SIGPLAN Not.*, vol. 47, no. 5, pp. 61–70, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2345141.2248428

[17] L. Itti, C. Koch, and E. Niebur, "A model of saliency-based visual attention for rapid scene analysis," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 11, pp. 1254–1259, Nov. 1998. [Online]. Available: http://dx.doi.org/10.1109/34.730558