SignalPU: A programming model for DSP applications on parallel and heterogeneous clusters

Farouk Mansouri, Sylvain Huet, Dominique Houzet

SignalPU: A programming model for DSP applications on parallel and heterogeneous clusters

Farouk Mansouri, Sylvain Huet, 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*—The biomedical imagery, the numeric communications, the acoustic signal processing and many others digital signal processing (DSP) applications are present more and more in the numeric world. They process growing data volume which is represented with more and more accuracy, and use complex algorithms with time constraints to satisfying. Consequently, a high requirement of computing power characterize them.**

To satisfy this need, it's inevitable today to use parallel and heterogeneous architectures in order to speedup the processing, where the best examples are today's supercomputers like "Tianhe-2" and "Titan" of Top500 ranking. These architectures with their multi-core nodes supported by many-core accelerators offer a good response to this problem. However, they are still hard to program to make performance because of many reasons: Parallelism expression, task synchronization, memory management, hardware specifications handling, load balancing ... In the present work, we are characterizing DSP applications and propose a programming model based on their distinctiveness in order to implement them easily and efficiently on heterogeneous clusters.

I. Introduction

The digital signal processing applications like computer vision, medical imagery or acoustic signal processing take a real place in many domains. Also, cause of sensors technology evolution, they process increased data volume (data length) like the high definition imagery which reach ten or so of Go. Also the data unit is represented more and more precisely (data floating point encoding) from single precision to 32 digits to quadruple precision to 128 digits. From the other side, the DSP applications use complex algorithms (time complexity) in linear, quadratic or exponential time. And are usually constrained with latency or throughput. Because of all this reason, we can sayy that DSP applications need a big computing power.

To satisfy this need, it's recommended today to focus on parallel and heterogeneous hardware architectures which include generalist multi-core processors (Intel Xeon or AMD Opteron), supported by many-core accelerators (GPU, Xeon phi, Cell, FPGA, DSP) and structured in the form of a cluster of connected nodes with a high bandwidth network. The best examples are the supercomputers like "Tianhe-2" or "Titan" related in the Top500 list.

Certainly, these architectures can get high performance computing to satisfy DSP applications requirements. However, they present some difficulties of use. In fact, to make performance with that last, the programmer has to deal with heterogeneous computation units using different languages or API, he has to manage synchronization, memory allocation, data transfers and the load balance between the processors. Thus, programming models are necessary to hide all this hardware specifications, and produce easily and efficiently the desired performance.

In the present work, we present first (Section II) the state of the art of models of programming and their classification. Afterwards, we describe in the section III the digital signal processing applications and highlight their characteristics. Then, we give the features to efficiently implement them in the next section (Section IV). In the section V we propose a programming model and explain its conception parts. Finely in the section VI we present experimentations and results of applying our model of programming (MoP) on a real world application.

II. State of the art of programming models

A programming model is a programming concept providing some abstractions to mask (hide) hardware specificities of parallel and heterogeneous architectures in order to easily and efficiently express high level applications. In other words, it's an interface between the hardware complexity and the software features which represent a base for the high level program. Thus, they increase productivity, portability and performance. They exist in the form of languages, language extension or library. Below, we present their classification under three axis:

A. Classification according to abstraction level

One of the important purposes of MoP is to make easier the implementation of high level applications on a complex architecture. That is possible by bridging between the hardware functionalities like the memory management, the tasks scheduling, the data copy and the basis stand of the high level algorithm. Therefore, a good programming model has to abstract the architecture details for the programmer. Thus, we distinguish two abstraction levels for today's MoP :

Low level abstraction: In this kind, the programmer has to handle manually a lot the hardware functionalities like allocation and freeing of memory, creation, submission and synchronization of tasks, data transfer, load balance... Thus, this proximity of architecture gives him more control to tune the execution but reduce productivity.

***High level abstraction**:* By contrast, in this kind, the programmer is freed of several of these constraints because the programming model does this for him.

B. Classification according to the communication model

In this classification we focus on the communication sorts used by models of programming in order to synchronize or to copy data between the tasks executed on several computing units composing the cluster. Here, we distinguish also two kinds :

***Shared memory communication**:* In this case, the programming model assumes the data copy and the synchronization of tasks on parallel architectures which are based on shared memory like multi-core or many-core computing units.

***Message passing communication** (Distributed memory):* This kind of MoP assumes the data copy and the task synchronization onto the distributed architectures by exchange messages between the tasks through the network connecting nodes of the cluster.

C. Classification according to the parallelism and heterogeneity expression

To deal with the parallel and heterogeneous architecture, the programmer has to express the part of the code which will be executed in parallel or on the accelerator by programming it inside the application. However, thanks to programming models, he can do that with more easier manners. For example, by function calls in the API, by directive insertion for language extension or by compilator's options added for some languages. We propose to classify this kind of expressions under two sorts:

***Explicit expression**:* In this case, the programmer has to explicitly express the part of the program to be executed in parallel by adding at the portion of code inside. For example, insert some directives or call functions from an API.

***Implicit expression**:* Unlike the previous case, here, the expression of the parallelism or heterogeneity is implicitly made by the programmer. For example, by giving the task graph of the application or by adding some compilation options.

In the figure Fig.1, we present some programming languages, extension of language and API, representing software aspects of programming models and their characteristics classified according to two axis:the communication model and the abstraction level.

1) Languages:

CUDA [1](Compute Unified Device Architecture): Is a proprietary programming model invented by NVIDIA to implement programs in their graphical processing unit (GPU). It's proposed in the form of a language inspired from C that the programmers can insert on their C/C++ code to offload in generally the SIMD portion. It's a low level abstraction MoP, and support only the shared memory architecture.

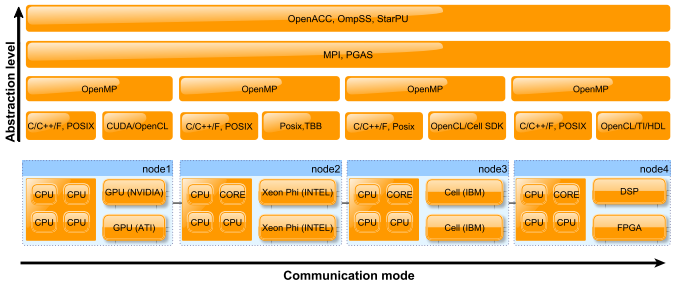

Fig. 1: Architecture and models of programming

OpenCL [2](Open Computing Language): Unlike CUDA, it's an open standard language based on C99 with some limitations and additions. It allows for the programmers to address heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGA, and others processors. Also, this MoP offers a low level abstraction and is designed for shared memory architectures but allows the both task and data parallelism.

PGAS [3](partitioned global address space): Is a parallel programming model which assumes a global memory address space for distributed architectures that is logically partitioned. Each portion of it is locally affected to each process or thread. It is a base for many programming languages like Chapel [4] developed by Cray, X10 by IBM [5], Fortress [6] by Sun or Unified parallel C [7] by UPC consortium.

StreamIt [8]: Is a programming language and a compilation infrastructure, specifically engineered for streaming systems. It is designed to facilitate the programming of large streaming applications, as well as their efficient mapping to a wide variety of target architectures like CPUs, GPUs, DSPs and FPGA.

2) Extension of language:

OpenMP [9](Open Multi-Processing): Is a high level abstraction MoP in the form of a set of directives and environment variables that allow for programmers to model his algorithms for data parallelism or task parallelism on a shared memory architecture. The programmer has to annotate his code by inserting some extensions (directives) which are identified at compilation time to generate supplementary code and executed on both hardware, CPUs and accelerators.

OpenAcc [10]: Like OpenMP, it's a programming standard for parallel computing based on PRAGMA directive but more oriented to data parallelism on shared memory architectures containing accelerators.

OmpSS [11]: Is another variant of OpenMP extended to support asynchrony, heterogeneity and data movement for task parallelism. As OpenMP, it is based on decorating an existing serial version with compiler directives which are translated into calls to a runtime system in order to manage the parallelism extraction and the coherence and the movement of data.

3) Application interface programming (API):

MPI [12] (Message Passing Interface): Is the most used MoP for distributed memory architecture. It consist

of a specific set of routines (i.e., an API) directly callable from supported languages (C/C++/Fortran ...). Using MPI, the programmers can distribute execution of their code on many nodes connected through a TCP/IP network.

TBB [13] (Threading Building Blocks): Is a C++ template library developed by Intel for writing software programs under task parallelism model and executing it on multi-core processor according to the dependency graph. It also schedules the processing to balance the load.

StarPU [14]: In the same manner, StarPU is a task programming library but for hybrid architectures CPU-GPU, and allows also for programmers to express the data parallelism. Using some routines and data structures, the programmer constructs a graph of tasks which are optimally scheduled and executed on heterogeneous and distributed memory cluster.

III. Digital signal processing applications

Nowadays the digital signal processing applications know a great boom and are present in a lot of domains. They are a repetitive (iterative) processing of data set of input digital signal for producing an output signal or a result, as shown in the figure Fig.2. In the algorithmic aspect, it represents a main loop which iteratively process all input data units.

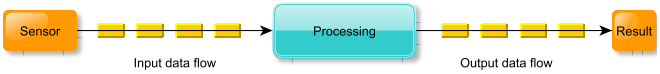

Fig. 2: DSP application's form illustration

The processing part represents the firing of each data unit composing the input signal using some operators (kernels), where each operator represents an independent function with some input and output arguments. In the algorithm, that's the function calls inside the main loop. We can model this part with a data flow graph, where the vertex of the graph represents the kernels and the edges represent the data trading between operators in the form of flow. The figure (Fig.3) illustrates this model. In this work, we limit ourselves to the treatment of synchronous data flow graph [15], [16].

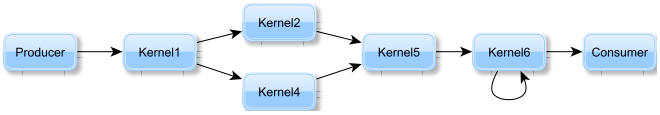

Fig. 3: Data flow graph (DFG) model of DSP application

In majority of DSP applications, multiple input data units are processed with the same set of instructions. For example, in video processing, each input image is processed in the same manner using the same algorithm. So, it's possible to run multiple data flow graph (DFG) in the same time on a parallel architecture. Also, each unit of data goes through the operators as input arguments and comes out as output arguments. Thus, depending on the DFG organization, some operators (kernels) must be executed in order, but some of them may be executed separately and in parallel. On the other hand, according to the data kind and the kernel algorithm, it could be interesting in some cases to offload execution of certain of kernels on a massively parallel computation unit (accelerators) like the GPU, Xeon Phi or Cell in order to enhance performances.

Taking into account these characteristics, it's possible to deduce some rules to apply in implementation and execution of DSP applications on parallel and heterogeneous architectures:

Task parallelism: Using the DFG model, highlight the dependencies between the tasks and detect the tasks able to be executed in parallel.

Data parallelism: Identify the tasks according to their Flynn taxonomy [17]. The MISD (Memory bounds) tasks are oriented to generalist processors and the SIMD tasks (Compute bounds) towards the accelerators.

Graph parallelism: In order to optimize the occupancy of computing units composing the cluster, deal with several graphs where each of them processes one input data unit.

In the next section (Section IV), based on extracted distinctiveness and the rules cited above, we focus on the implementation side of the DSP applications, and we discuss on which programming model is more suitable for programmers to easily and efficiently porting these applications on parallel and heterogeneous architectures.

IV. Implementation of DSP applications

As presented in the preceding section (Section III), the DSP applications have some characteristics, that the programmers must exploit in order to take advantage of targeted heterogeneous and parallel architectures. First: To highlight the kernels able to be executed in parallel (task parallelism), they have to express their algorithm in the form of a set of tasks using threads or process technologies. They have to manage these threads for communicating between them or to be synchronized according to the application's dependencies on the both shared and distributed memory architectures. In addition, to profit from the accelerator's capacity to speedup the SIMD processing (data parallelism), the user has to offload a part of their task towards these compute units. To do this, he has to deal with memory allocation on accelerators, copy-in the input data, lunch the execution, copy-out the results and finally freedom the used memory zone. Also, the DSP applications are mostly iterative, so it's a good idea to unroll the main loop of the application and therefore process a number of data units in the same time (graph parallelism) in order to increase the occupancy of computing units. To do that, the code writers must duplicate the process (thread) in charge of executing the main loop taking care to guarantee the data coherence by restricting some variables or sharing others. All this implementation features are necessary for porting DSP applications on heterogeneous clusters but not enough to optimize productivity of the hardware. In fact, the programmer must cope with others difficulties like communication cost which must be masked by overlap it with the computation, or the load balancing between the computational units which must be assumed by a good scheduling of tasks.

Applying all these implementation rules is very hard. The programmer has to combine the handling of some API, language or extension of language which are low level for certain or restricted to specific hardware for others. For example, the programmer has to use Pthread , TBB or OpenMP to

generate threads and express task parallelism on each node of the cluster (shared memory architecture), but also the MPI or PGAS model to manage them by creating processes onto many nodes (distributed memory architectures). He has to use CUDA or OpenCL to address accelerators like the GPU, Cell or Xeon-Phi and offload a part of a SIMD work on it. In the other case, the higher level tools like OpenACC, OmpSS or StarPU which are based on the low level tools, must be the solution. They offer more abstraction of the hardware and can target the complete cluster. But some of them are restricted to a particular model of programming, for example OpenAcc express only the data parallelism. Others of them like OmpSS are rather oriented to decorating an existing sequential code by inserting some PRAGMA directive and transforming it at compilation time into a parallel code. The rest, based on API like StarPU is, in our opinion, the most adapted programming models to implement high level applications on heterogeneous cluster. It offers an interface based on a large routines and structures which the programmers can use to design their applications, and in addition proposes a runtime which manages the tasks, their dependencies and dynamically schedule their executions on the architecture. However, it's not adapted (specified) to DSP applications as characterized in the section III with their iterative and repetitive form, and also it's still complicated to handle because of the number of routines and data structures proposed to the user as interface to implement their applications. Because of these reasons, we propose in the next section, a programming model based on a data flow graph model to make easier the application modeling and automatize the generation of the directional acyclic graph of tasks (DAG) in order to adapt StarPU to the implementing of DSP applications on heterogeneous cluster.

V. Our model of programming

In this section we propose a programming model as an extension of StarPU's application programming interface (API) [14], which allows for programmers to express easily and efficiently DSP applications on parallel and heterogeneous cluster. In comparison with the state of the art classification described in section II, our MoP is a high level abstraction concept. The programmers don't have to worry about several architecture specificities, like memory management, task creation, computing units profiling etc... They can design their algorithms for tasks and data parallelisms. Also, because it's based on StarPU, our MoP take in charge the both shared and distributed memory architectures, and deal with many-node cluster using the messages passing interface (MPI). Finally, it's implicit parallelism and heterogeneity expression. In fact, thanks to the DFG application design, the programmer is free to specify in the code which tasks will be executed in parallel and on accelerators.

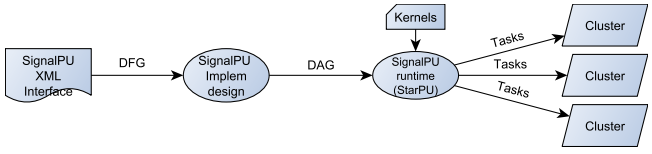

Fig. 4: SignalPU design : Three levels of processing

We present our MoP in 3 steps of treatment as shown in the figure (Fig.4). First level: the DFG-XML interface with which the programmer can describe his application in the form of DFG, he can specify the operators composing the processing (function, arguments, architecture kind) and the data flow information (size, type, link). Thus, he is saved to manipulate the StarPU's API for creating tasks, for managing buffers between each couple of task, or for submitting jobs onto the corresponding computation unit. Second level: the application design. In this step, the DFG is transformed into a directional acyclic graph (DAG) of task using graphs unfolding techniques [18], where tasks are linked according to data dependencies. Also here, the user doesn't have to deal with the API to unroll the main loop. Third level: the StarPU runtime is used to physically manage the set of tasks and execute them on the cluster according to an optimal dynamic scheduling.

Below, we describe with more details and through a synthetic application example the 3 steps of our programming model :

A. Level 1: SignalPU DFG-XML interface

Let's take the synthetic application described in the algorithm 1

Algorithm 1 Synthetic DSP application

Input: Number of iterations (Nbr). Input data set ($Data_{in}$).
Output: Output data set ($Data_{out}$).

1: **for each** $Data_{unit}$ in $Data_{in}$ **do**
2: $Var_1 \leftarrow Producer(Data_{unit})$
3: $Var_2 \leftarrow kernel_1(Var_1)$
4: $Var_{3_1} \leftarrow Kernel_2(Var_2)$
5: $Var_{3_2} \leftarrow Kernel_3(Var_2)$
6: $Var_4 \leftarrow Kernel_4(Var_{3_1}, Var_{3_2})$
7: $Var_5 \leftarrow Kernel_5(Var_4, Var'_4)$
8: $Var'_4 \leftarrow Var_4$
9: $Consumer(Var_5)$
10: **end for**
/// $Data_{unit}$ may be an image or a sample ///

In this step, the programmer has to express his application using the DFG-XML interface description. First, he has to describe each kernel (functions) in the algorithm in the form of graph node (vertex) using the given XML structure. He has to put the number of input and output arguments, the architecture kind corresponding to the kernel (CPU, GPU, Cell, Xeon Phi ...). Second, he has to describe, using an XML structure, the data flows between the kernels in the form of graph edges with a weight corresponding to the type and size of the data which is traded between kernels. As result, a data flow graph of application is produced as shown in the figure Fig.5.

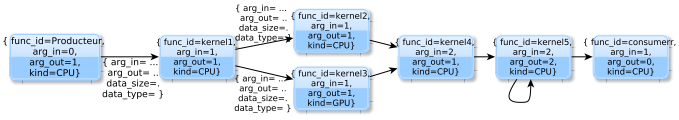

Fig. 5: The DFG-XML of the synthetic DSP application

B. Level 2: SignalPU application design

In this step, using the graph unfolding techniques, a DFG of task is iteratively produced from the result of previous processing level (DFG-XML model of application). This DFG represents a set of independent tasks linked by several kinds of data dependencies (Fork-join, producer-consumer, inter-graph producer-consumer).

a) Tasks generation: First, the set of tasks is iteratively created when each task represents the execution of each kernel of DFG with one data unit (One iteration). The characteristics of task (Number of input arguments, the number of output arguments, function identifier, architecture kind) are imported from the XML structure defining the corresponding kernel in the DFG-XML model.

b) Tasks linking: Second, the links between each couple of tasks are putted according to the edges defined in the XML data flow graph. A link represents a buffer of memory which serves to stock the data produced by output argument from predecessor task and consumed as input argument in the successor task. Also here, the characteristics of buffers (Data type, data size, input argument , output argument, input vertex, output vertex) are found in the XML structure of the corresponding edge.

This level's part of our programming model is described by the algorithm 2. We present in the figure Fig.6 the DSP of task of the synthetic DSP application obtained by applying the algorithm 2 over 4 iterations.

Algorithm 2 DAG of tasks creation

Input: XML data flow graph (DFG_{xml}). Number of iterations (Nbr)
Output: DAG of tasks (DAG)

```
for each iteration in (Nbr) do
    for each node in (DFG_xml) do
        T_set ← Create_task()
    end for
    for each edge in (DFG_xml) do
        T_set ← Link_task()
    end for
    DAG ← Submit_tasks(T_set)
end for
```

C. Level 3: StarPu runtime

In this step, we use StarPU runtime to manage the DAG of tasks generated in the previous level and to dynamically schedule them taking into account the dependencies between the tasks in order to guarantee the data coherence. The figure below (Fig.7) show how StarPU runtime does that:

At runtime, all the submitted tasks are stored on a stack, each task has an affected status (ready to be executed, waiting some things, in firing, terminated ...). StarPU runtime piles up the "ready to execute" tasks on another task list which represents the input set of tasks for scheduler, which dynamically affects them to be executed on the appropriate computation unit using some heuristic algorithms in order to determine the best time for each task. One of these algorithms is the Work Stealing (WS) [14] is based on estimated transfer times

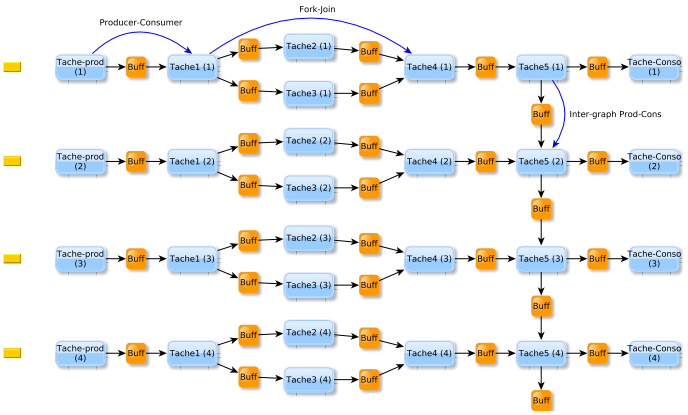

Fig. 6: The DAG of tasks of synthetic DSP application

of task's data to predict which compute unit will be the best for each task. Another policy like the Heterogeneous Earliest Finish Time (HEFT) [19] uses in addition, the estimation of execution times of each task on each device in order to produce more efficient scheduling.

With these three steps of our proposed MOP, the programmer can easily implement his DSP application and applies the optimizations: Tasks parallelism (TP) by extracting the tasks in the DFG which be able to be executed in the same time. Data parallelism (DP) by off-loading some tasks on (SIMD) accelerators. Graph parallelism (GP) by overlapping the processing of some graphs. And the load balance tanks to dynamic scheduling using StarPU runtime.

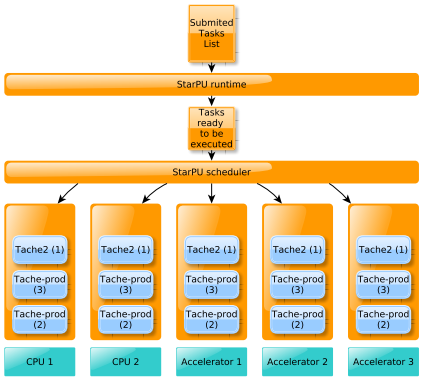

Fig. 7: StarPU runtime's levels

VI. Experimentations and Results

In this section we present you the real world experimentations in order to validate our approach and demonstrate the interest of its usage. We use the saliency application to process a set of images on the heterogeneous CPU-GPU architecture. First, we describe the saliency application and give its algorithm. Then, we explain its implementation using our programming model. And finally, we give the results and discuss their impacts.

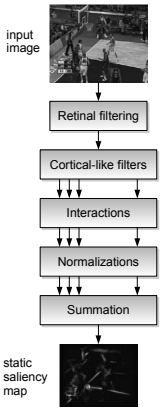

Fig. 8: The static pathway of the visual saliency application

A. The saliency application

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 [20] as shown in figure Fig.8. His algorithm (Algorithm 3) is: First, the input image ($r\text{-}im$) is filtered by a Hanning function to reduce intensity at the edges. In the frequency domain,($cf\text{-}fim$) is processed with a 2-D Gabor filter bank using six orientations and four frequency bands. The 24 partial maps ($cf\text{-}maps[i;j]$) are moved in the spatial domain ($c\text{-}maps[i;j]$). 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 3 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: $saliency\_map \leftarrow Summation(r\_normaps[i,j])$

B. The SignalPU implementation

To implement the application with our programming model, the first step is to model its algorithm (Algorithm 3) given before in the form of DFG-XML using the SignalPU interface. For this, we represent each of all functions ($Hanningfilter()$, $FFT()$, $GaborFilter()$, $IFFT()$, $Interactions()$, $Normalizations()$, $Summation()$) with a node in the graph including the characteristics of each of them (architecture kind, input arguments, output arguments). Then, we represent the data flow between each twice kernels with an edge in the graph including its characteristics (data type, data size). In the figure (Fig.9) we present the DFG resulting of this step.

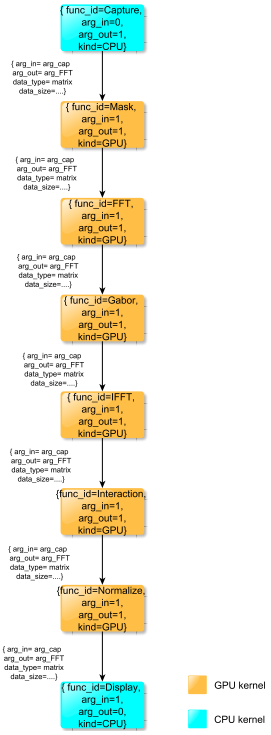

Fig. 9: The DFG-XML model of the visual saliency application

At runtime, the XML-DFG description of the saliency application is analyzed and a DAG of independent tasks is iteratively generated, where each task represents the execution of each kernel's code (function's code) for each image on the corresponding computation unit (CPU,GPU). Thus, we haven't to use the StarPU's API for describing the application's tasks. Also, we have not to manage the buffer's allocating and freeing. We haven't written the main loop which processes the set of input images, and don't have to unroll it. The StarPU's API is almost entirely masked.

C. The results

In this subsection, we present results of two experimentations where we show the performance provided by the implementation with our programming model using the optimizations which we propose (graph unfolding + dynamic scheduling), compared to the same implementation with a static scheduling, and without graph unfolding. The aim is to give you an indication of the performance gained using our model of execution compared to another classical implementation. The architecture used for experimentations is a heterogeneous CPU-GPU node composed of a 4 cores CPU (intel-i7 core) and 3 GPU (1 NVIDIA GeForce GT 610, 1 NVIDIA Quadro 4000, 1 NVIDIA GeForce GTX TITAN). Note that the GPU devices are ordered from the lowest to the highest powerful. For all experimentations, we decide to put 2

tasks per device as a static scheduling. That's not the optimally placement, but the goal is to compare a dynamic scheduling to a naive user placement without profiling device's power.

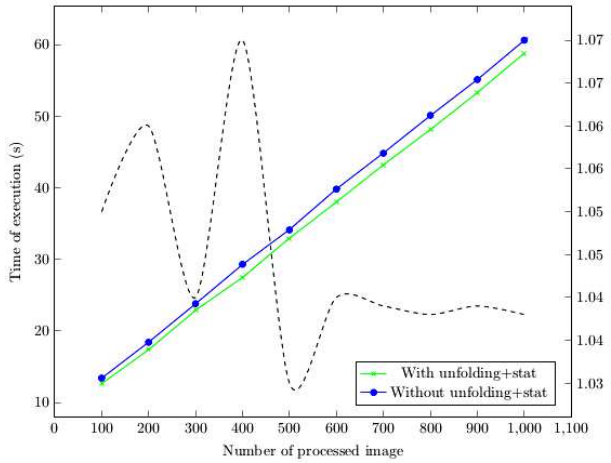

Fig. 10: The performance as a function of number of processed images with static scheduling

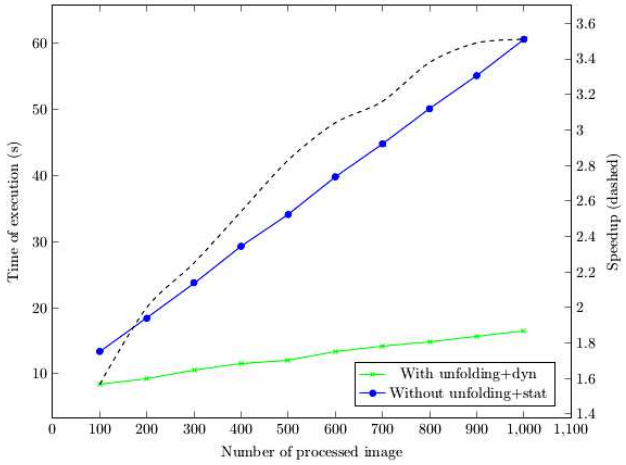

Fig. 11: The performance as a function of number of processed images with dynamic scheduling

For the first experimentation, we present first in the figure Fig.10 the evolution of the performance (Time of execution) as a function of number of processed images (iterations) in a static scheduling context. In the green curve (marked with 'x'), we show the evolution of execution time (in second) necessary to process each set of images using our programming model with graph unfolding which allow to process several graphs in the same times. In the blue curve (marked with '*'), we show the same result of implementation but without graph unfolding techniques. The comparison between them is represented on the dotted black curve as speedup obtained between both results which reach and is stabilized on 1.04 x. In the figure Fig.11, we present the same experimentation (evolution of time of execution) as a function of number of processed images (iterations)) but here, we combine the dynamic scheduling and the graph unfolding optimizations. So, here the speedup obtained between both results reach 3.5 x.

We explain this variance of performance gains in both parts of this first experimentation by the increasing of the number of tasks executed in parallel. In fact, the present application is completely sequential as shown in the figure Fig.10, but thanks for the graph unfolding, it's possible to process a number of data flow graphs and several images are executed at the same time. However, in the first part of this experiment (Fig.11), the tasks are statically putted on the architecture and each kernel of the DFG model of the application is associated with a computing unit. Therefore, each compute unit processes all the time the same size of the task, consequently, one of them (which processes the biggest tasks) creates a bottleneck, and it delays the processing. In contrast, with using the dynamic scheduling, it's possible to more take advantage of vacancy of all computation units and enhance the processing.

In the second experiment, we process a fixed number of images (100 images) but we vary its size (height and width). The figure Fig.12 shows the evolution of execution time in tow implementations: with dynamic scheduling, and with static scheduling. Also here, the green curve (marked with 'x') represents the evolution of execution time of the set of images processed by the application with dynamic scheduling but without graph unfolding, as function of image size. And in the blue curve (marked with '*'), we present the same result obtained by applying static scheduling. The difference between both results is shown in the black dotted line as a speedup that we get by using our programming model which reach only 1.19 x. The figure Fig.13 represent the second part of experimentation, shows the same result where we add the graph unfolding optimization to dynamic scheduling in the the green curve (marked with 'x'), and keep the second implementation (marked with '*') in static scheduling and without graph unfolding. Also here, the difference between both results is shown in the black dotted line as a speedup which grows-up to 2 x in this case.

The explanation of the performance increasing (from 1.19x to 2x) we can give is: In the first part of the experimentation (Fig.12), the use of dynamic scheduling without graph unfolding restrict the performance because there is not enough tasks to process inside each iteration of the main loop, in order to optimally occupy all computing units. In fact, by using the graph unfolding in the second part of experimentation (Fig.13), it's possible to overlap the processing of several iterations (graphs), so the runtime continually distribute works, and optimally takes advantage of hardware availability. In addition, we note that the dynamic scheduling is more efficient when data is bigger. That's due to the reduction of the overhead time compared to execution and communication times of processed tasks.

From this result, we can say that it's it's profitable to use graph unfolding techniques combined with a dynamic scheduling to process the DSP applications as we propose in our programming model in order to better take advantage of computation unit availability.

VII. Conclusion

This paper presents our proposed programming model for parallel and heterogeneous architecture, which allows a high level abstraction from the hardware specificities, and at the

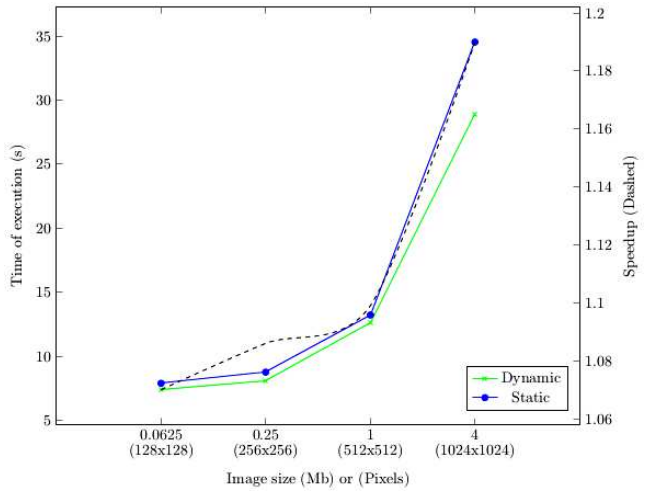

Fig. 12: The performance as a function of size of processed images without graph unfolding

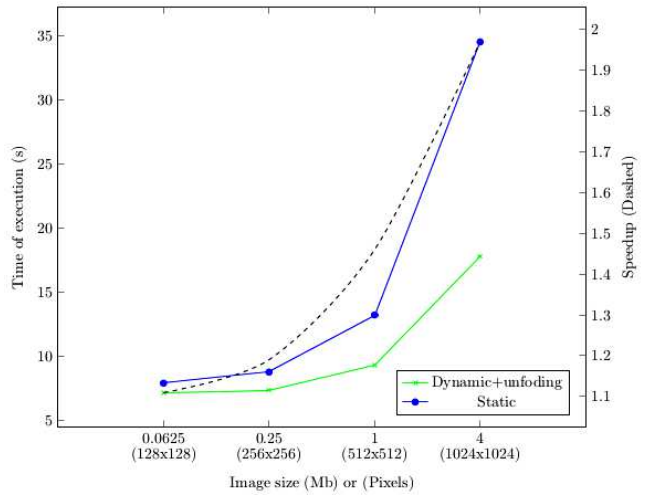

Fig. 13: The performance as a function of size of processed images with graph unfolding

same time increases the performance of the implemented application. First, we presented the state of the art of MoP and classified them according to different axis. We described the DSP applications and specified their characteristics in order to implement their with optimal manner. For that, we proposed an Xml interface to easily describe DSP applications in the form of a DFG model. Also, we proposed a strategy based on unfolding graph techniques to construct a directional acyclic graph (DAG) of tasks which we process using StarPU on a heterogeneous and parallel architecture with a dynamic scheduling. Finally, we experimented our MoP on the real-world saliency application and shown that's easier to use our programming model to design it, but at the same time, it's possible to optimally take advantage of architecture's power to speed up the execution using the optimizations: Task parallelism, data parallelism, graph parallelism, and dynamic scheduling.

References

[1] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[2] A. Munshi, B. Gaster, T. Mattson, and D. Ginsburg, *OpenCL Programming Guide*, ser. OpenGL. Pearson Education, 2011. [Online]. Available: http://books.google.fr/books?id=M-Sve_KItQwC

[3] W.-Y. Chen, "Optimizing partitioned global address space programs for cluster architectures," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2007. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-140.html

[4] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442

[5] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, "X10 language specification," IBM, Tech. Rep., January 2012. [Online]. Available: http://x10.sourceforge.net/documentation/languagespec/x10-222.pdf

[6] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt, "The Fortress Language Specification," Sun Microsystems, Inc., Tech. Rep., March 2008, version 1.0.

[7] UPC Consortium, "Upc language specifications, v1.2," Lawrence Berkeley National Lab, Tech Report LBNL-59208, 2005. [Online]. Available: http://www.gwu.edu/ upc/publications/LBNL-59208.pdf

[8] M. I. Gordon and S. Adviser-Amarasinghe, "Compiler techniques for scalable performance of stream programs on multicore architectures," 2010.

[9] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[10] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[11] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with ompss," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 555–566. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033345.2033405

[12] P. S. Pacheco, *Parallel Programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[13] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[14] C. Augonnet, S. Thibault, and R. Namyst, "StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines," INRIA, Research Report RR-7240, 2010.

[15] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *Computers, IEEE Transactions on*, vol. C-36, no. 1, pp. 24–35, Jan 1987.

[16] B. Bhattacharya and S. Bhattacharyya, "Parameterized dataflow modeling for dsp systems," *Trans. Sig. Proc.*, vol. 49, no. 10, pp. 2408–2421, Oct. 2001. [Online]. Available: http://dx.doi.org/10.1109/78.950795

[17] M. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, no. 9, pp. 948–960, Sept 1972.

[18] K. Parhi and D. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *Computers, IEEE Transactions on*, vol. 40, no. 2, pp. 178–195, Feb 1991.

[19] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," vol. 13, no. 3. IEEE, 2002, pp. 260–274.

[20] 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