A domain-specific high-level programming model

Farouk Mansouri, Sylvain Huet, Dominique Houzet

2015; **00**:1–16

A domain-specific high-level programming model

Farouk. Mansouri, Sylvain. Huet and Dominque. Houzet. *

Gipsa-Lab, 11 rue des Mathmatiques, Grenoble Campus BP46, F-38402 SAINT MARTIN D'HERES CEDEX, France
Email: firstname.lastname@gipsa-lab.grenoble-inp.fr

SUMMARY

Nowadays, computing hardware continues to move toward more parallelism and more heterogeneity, to obtain more computing power. From personal computers to supercomputers, we can find several levels of parallelism expressed by the interconnections of multi-core and many-core accelerators. On the other hand, computing software needs to adapt to this trend, and programmers can use parallel programming models (PPM) to fulfil this difficult task. There are different PPMs available that are based on tasks, directives, or low level languages or library. These offer higher or lower abstraction levels from the architecture by handling their own syntax. However, to offer an efficient PPM with a greater (additional) high-level abstraction level while saving on performance, one idea is to restrict this to a specific domain and to adapt it to a family of applications. In the present study, we propose a high-level PPM specific to digital signal processing applications. It is based on data-flow graph models of computation, and a dynamic run-time model of execution (StarPU). We show how the user can easily express this digital signal processing application, and can take advantage of task, data and graph parallelism in the implementation, to enhance the performances of targeted heterogeneous clusters composed of CPUs and different accelerators (e.g., GPU, Xeon Phi).

Received . . .

KEY WORDS: Programming model, Heterogeneous cluster, Data-flow graph, Digital signal processing, Unfolding techniques, Auto-tunning

1. INTRODUCTION

Since about 2005, the hardware trend has moved toward the era of multi-cores and many-cores architecture to offer improved performance. However, these architectures are still difficult to handle. Programmers need to express parallelism of tasks and data, and deal with their constraints, such as communication, load balancing, memory management, and synchronization. In addition, programmers have to face heterogeneity problems with the use of different softwares for each accelerator. For all of these reasons, parallel programming models (PPM)s have emerged. These are a programming concept that is used to abstract the cited hardware specificities in order to decrease implementation needs on parallel and heterogeneous clusters, while increasing performance, which is not a simple problem.

In the present study, we exploit the idea that a domain-specific parallel programming model (PPM) can offer high abstraction, and at the same time, allow efficient implementation of a family of applications on a heterogeneous cluster. Following this idea, we focus on the digital signal processing (digital signal processing (DSP)) domain, where the major challenge is to more easily express these applications in a high-level mode while efficiently exploiting the performance of the clusters. Thus, we propose a design flow that is based on a dynamic run-time (StarPU), to efficiently implement DSP applications by simply specifying them graphically as a data flow graph (Data Flow Graph (DFG)) model of computation (model of computation (MoC)).

The organization of this paper will be as follow: first, in section 2, we present PPMs and their classification according to their abstraction level. In section 3, we present related studies on the implementation of DSP applications on heterogeneous clusters, and we study and extract their characteristics that are necessary to optimize their implementation on a cluster. Finally, we position

our approach by comparing it to related studies. In section 4, we describe our programming model and detail its functionalities. Finally, in section 5, we present a comparison of the implementation of two applications, to validate our approach.

2. PARALLEL-PROGRAMMING MODELS

A PPM is a bridge between a system developer's natural model of an application and its implementation on parallel architectures. It is a programming concept to provide connections between the user applications and the functionality of clusters. A PPM has to achieve the best trade-off between designer productivity and implementation efficiency. Indeed, as presented in [1], a good PPM should have some specific proprieties, such as: (1) simplification of the programming effort; (2) portability and architecture undependability; (3) efficiency of implementation.

To achieve all of these requirements in a PPM might be a difficult task, knowing that several of them are in opposition to each other. The abstraction levels of PPMs is an important basis to distinguish them, because they affect several evaluation metrics, as cited above. In [1], they proposed to classify PPMs according to whether or not they offer some inherent functionality of abstraction: (1) Decomposition of a program into parallel threads or processes; (2) Mapping of threads to processing elements (processing element (PE)s); (3) Communication among the threads; and (4) Synchronization among the threads. Figure 1 shows a classification of several PPMs according to the abstraction level of their hardware.

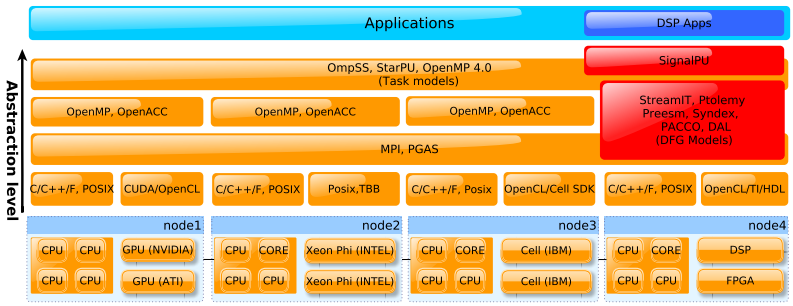

Figure 1. Classification of parallel programming models according to their abstraction level. SignalPU is shown as a combination between a task-based run-time (StarPU) and a DFG model, with an additional abstraction level.

3. RELATED STUDIES ON IMPLEMENTATION OF DIGITAL SIGNAL PROCESSING APPLICATIONS ON PARALLEL AND HETEROGENEOUS ARCHITECTURES

Throughout the history of computing, DSP applications have pushed the limits of compute power. Calculations have been performed on different hardware, like the FPGA, DSP, and SMP platforms. Recently, DSP applications have seen rapid advance in multimedia computing and high-speed communications. They process large data volumes using complex algorithms. In response to these advances, research is focusing on the implementation of several of these applications on parallel architectures, including heterogeneous accelerators like GPU, Xeon Phi and others. Next, we present the state of the art of the implementation of DSP applications on these architectures.

Low-level parallel-programming models (mostly explicit implementation) Traditional implementations are based on low-level tools or PPMs. In [2], they implemented a Ray-tracing application on a Cray XD-1 many-core architecture with distributed memory architecture. They used a message-passing interface (MPI)[3] to decompose the algorithm over hundreds of MPI processes that iteratively performed a number of rays, while communicating over the network. In [4], a synthetic aperture sequential beamforming GPU implementation of ultrasound images was performed. The algorithm processed two-dimensional data sampling. The authors used both OpenCL[5] and OpenGL[6] to decompose the algorithm into thread blocks. Each block of threads processed a two-dimensional subset of input image. In [7], they used CUDA[8] PPM to implement a medical ultrasound algorithm to generate B-mode images. They decomposed the algorithm into four kernels that were sequentially processed on the GPU to execute one frame. The algorithm was

iteratively re-executed for each frame of the input dataset. In [9], they implemented a least mean square algorithm on different GPUs by also using a CUDA model. They optimized their algorithms for a least mean square algorithm family using shared memory. These implementations using low-level models are efficient and the authors achieved good performance in comparison to sequential implementation. However, the programming effort is significant. Indeed, for all of the cited PPMs, the programmer has to carry out the following manually: (1) decompose the algorithm into threads by unrolling loops of data samples to exploit data parallelism, or loops of algorithm steps to exploit task parallelism; (2) manage the memory allocation for each thread; (3) synchronize the threads to save data coherence, using barriers, Mutex or Semaphore; (4) manage communications among threads over shared memory or distributed memory architecture; and (5) map the parts that result from the decomposition over threads or processes. In addition, the low-level PPMs used are specific to a kind of PE, so CUDA, OpenCL, and OpenGL for GPU (accelerators), and Pthread and MPI for multi-core and many-core. Finally, these are also specific to one memory architecture. MPI for distributed memory, and the others for shared memory architecture.

Directive-based parallel-programming models (partly implicit implementation) Common tools and models that are used to implement DSP applications on parallel and heterogeneous architecture are directive-based high-level PPMs. The users explicitly annotate the sequential code to decompose the algorithm into parallel parts only at compilation times that can increase the productivity and portability of the codes. The user does not care about communication, synchronization, and mapping in default mode, but has to explicitly specify them in performance mode. In [10], they used OpenMP[11] to implement geodesic applications on multi-core architecture. The authors inserted directives to data parallelize the execution of two-dimensional images, where each OpenMP thread processed a row at a time. The algorithm was iteratively repeated until convergence. OpenMP is a commonly used standard and targets both multi-core and many-core architecture, and accelerators in its last version (OpenMP 4.0). This allows the user to unroll loops over CPU or accelerator threads, or to construct sets of dependent tasks. However, it is a structured (fork-join) execution model. It allows parts of parallel regions, called Chuncks, to be dynamically scheduled, but their sizes are manually fixed by the user, and there is no task scheduling across heterogeneous PEs. OpenMP implementation and more details are discussed in section 5.1.1. Another PPM very close to OpenMP is OpenACC[12]. This is also a directive-based PPM, but it is more oriented to GPU accelerators. In [13], they presented an OpenACC implementation of a three-dimensional elastic wave simulator on a multi-core plus GPUs architecture. They manually decomposed the algorithm over two MPI processes, and parallel regions to off-load onto GPUs. They obtained good performance in comparison to the programming effort. However, it is necessary for the user to manually tune the data allocation and transfer, and the scheduling of parallel regions on GPU architecture. In addition, the user has to use MPI to target the distributed memory of the two GPUs. Finally, the OpenACC implementation required a commercial compiler, such as PGI or Cap's. Other directive-based tools and models, like HMPP[14] or OmpSS[15], follow the same model of OpenMP, with some extensions. However, they include the same inconvenience in their implementation of DSP applications. They do not manage data-flow dependencies, except for OmpSS and OpenMP 4.0. They are restricted to shared memory architecture. In addition, in performance mode, the user has to deal with the architecture to map parallel work over PEs.

Task-based parallel-programming models (mostly implicit implementation) Other models and run-times to implement DSP applications are task-based PPMs. Using these, the user explicitly decomposes the algorithm in the form of a graph of the task, in general in the form of directional acyclic graphs (Directional Acyclic Graph (DAG)s) of tasks that are managed according to data-flow dependencies. The user is saved from explicitly managing the communication, synchronization, and scheduling of tasks over the architecture. In [16], they used StarPU[17] to implement applications of coin and contour detection in a large-volume database of images on heterogeneous cluster. They pre-loaded all of the images into memory, and then executed the algorithm that was decomposed into DAGs of tasks, where each task partly processed one image on the CPU or GPU. The authors obtained good performances in comparison to sequential implementation, and exploited all of the PEs of the cluster using dynamic scheduling, like work-stealing or heterogeneous earliest

finish time (HEFT). However, they needed to manually construct the DAGs of the tasks. They had to manipulate the API to create codelets, tasks and buffers. They had to link tasks between these, using some functions and data-struct. Finally, they had to submit the DAG to the run-time. Cilk[18], X-Kaapi[19], TBB[20], Ptask[21], are run-times and models based on DAGs of tasks, but these all include the same cited disadvantages.

Data flow graph model-of-computation-based parallel-programming models (DSP-specific implementation) There are models and tools based on DFG MoCs that can be used to simulate and implement DSP applications on different parallel platforms. As discussed in [22, 23], the DFG MoC models algorithms as nodes, known as operators, that represent sub-functions of the system. These are connected with directional edges that represent FIFO buffers and that transport sub-result data, known as Tokens, to synchronize operations. An operation runs as soon as all of its inputs become valid. Thus, this model is inherently parallel and allows the user to easily express task parallelism on the application. In [24], they presented StreamIT, which is a language with specific syntax to textually model the algorithm as a DFG. This includes an execution model to implement the algorithm that exploits different levels of parallelism, and targets multi-core architecture, but not accelerators. Another major tool to model, simulate and execute DSP applications is Ptolemy [22]. This offers the expression of several DFG MoCs using a graphical interface; however, this execution model is restricted to a mono-core DSP. PREESM [25] is another tool to implement DSP applications on multi-cores using DFG MoCs. However, the user has to manually express the application and the architecture in the form of a graph, and to statically map these according to specific scenarios. All of these tools are specific for simulation and execution of DSP applications. They take in account the characteristics, such as data-flow synchronization, granularity, iterative forms, and scaling. These are implicit models for communications and synchronization. However, they are restricted to mono-core or multi-core accelerators. In [26], they presented an extension of StreamIT that can automatically map streaming applications represented as DFGs onto GPUs. However, the mapping is static and it is performed at compilation time. In [27], they proposed a compiler-based PPM to implement streaming DSP applications on multi-GPU architecture. The tool they proposed is in the form of a source-to-source compiler that generates GPU and CPU codes from the SystemC [28] description of the application as a DFG. They also used the SynDEX tool [29] to map application graphs onto architecture graphs. Thus, also here, the user has to manually represent the architecture as a graph, and the scheduling generated from SynDEX is static. The same inconvenience characterizes the PACCO [30, 31] tool. This is manual and static scheduling, and the user has to describe the architecture graph. In addition, it is a BSP tool [32] that implies a synchronization barrier for each iteration. In[33], they introduced a framework based on OpenCL to execute DSP applications specified as DFGs on a heterogeneous cluster. However, the user has to manually map the operators onto OpenCL workers in charge of PEs.

Summary and contribution For all of the above-cited implementations of DSP applications, we can characterize them as follows: they use repetitive (iterative) processing of an input set of digital signal samples to produce an output set; e.g., a set of images. DSP algorithms are usually and preferably modeled with inherently parallel DFG MoCs [22, 23], where the application is designed with an oriented graph. The nodes represent the operators (functions of an algorithm) and the edges represent the data exchange between these, as sub-results or variables. In the parallel implementation of DSP applications, programmers must exploit these specificities to take advantage of targeted heterogeneous and parallel architectures. First, to highlight the operators that can be executed in parallel (task parallelism), they have to express their algorithm with a set of tasks, using threads or process. They have to manage thread communication and synchronization according to the data-flow application dependencies (edges of graphs) on both shared and distributed memory architectures. In addition, to benefit from the accelerator capacity to speed up the SIMD processing (data parallelism), the user has to off-load a part of their tasks toward PEs, like GPU or Xeon Phi. The user has to deal with memory allocation on the accelerators, copy-in the input data, launch the execution, copy-out the results, and finally free the allocated memory space. DSP applications are also mostly iterative, so in some cases, the designer has to unroll the main loop of the application to increase the task parallelism, to increase the occupancy of the

computing units. So, the designer must duplicate the process (thread) in charge of executing the main loop, while taking care of the data coherency. The programmer must also cope with other difficulties, like overlapping the communication with the computation, the load balancing between PEs that assumes a scheduling algorithm that takes into account the communication cost, the efficient memory management to reduce the bottleneck due to allocating or freeing buffers, and the unnecessary synchronizations that can delay the execution, plus more.

Taking into account all of these implementation constraints carried out by hand, this is particularly bad and leads to application-specific implementation. Using low-level tools and models like CUDA and MPI presented in section 3, the programmer has to combine the handling of some API, language or extension of language to explicitly decompose the algorithm, and to manage memory, communications, synchronizations, and scheduling. These models are very close to hardware language and restricted to a specific architecture, which can decrease the productivity. Otherwise, users can use directive-based tools like OpenMP and OpenACC presented in section 3 to implement DSP applications on heterogeneous architecture. However, it can be hard for them to exploit the above-cited characteristics of this application family such as DFG decomposition and synchronization. In addition, to obtain good performance, users have to focus on allocation and communication of data, and scheduling of parallel region onto threads. Another solution the users can use to implement DSP applications on heterogeneous architecture is the task-based tools like StarPU discussed in paragraph 3. These tools are more adapted for decomposition and synchronization of DSP applications. However, programmers have to manipulate their API to create, delete, and submit tasks. They have to create buffers and link them to synchronize tasks. They also have to manually unroll loops of input data to exploit data parallelism, to create a DAG of tasks. In addition, users have to take care of the overhead cost due to run-time management. Finally, users can use the DFG MoC-based models cited in section 3, which are the most adapted tools for implementing DSP applications. However, according to our discussion in section 3, part of these do not target heterogeneous architecture. Other parts of these are static tools; i.e., at compilation time construction and mapping of tasks. Finally, the others are manual scheduling of applications, and are not architecture aware. So, in our opinion, there are no tools that can implement DSP applications using DFG MoCs on heterogeneous clusters that use an architecture-aware run-time, and that offer dynamic scheduling. In [34, 35], they discussed the advantage of dynamic versus static scheduling for DFG applications. Indeed, even if several DFG MoCs are decidable and predictable at compilation time, there are some dynamic DFG MoCs that need to be scheduled at run-time. In addition, some applications are data dependent, where the processing time is not predictable and changes according to the input data. Finally, dynamic scheduling is more adapted for clusters to optimally take advantage of all of the heterogeneous PEs by load balancing the work. Thus, for all of these reasons, in the next section we propose to enrich the StarPU programming model with a novel design flow, to adapt it for implementing DSP applications represented with DFG models of computation that simplify their expression and computation, and at the same time increase their performance. On this basis, our major contributions are:

Conceptual contribution: A novel design flow to implement DSP applications on **heterogeneous** clusters based on **DFG** models of computation in high-level abstraction, and the architecture-aware and **dynamic** run-time (StarPU).

Functional contribution: Based on DSP domain characteristics, we propose the additional functionalities of:

1. Dynamic and implicit construction of the DAG of the task.
2. Automatic unfolding of the DFG of the application.
3. Implicit allocation and reuse of buffers.
4. Automatic execution and saving of the initialization part of repetitive tasks.
5. Dynamic auto-tunning of GPU tasks.

4. THE PROPOSED PROGRAMMING MODEL

In this section, we propose the use of SignalPU, a novel parallel programming model based on DFG models of computation, and the dynamic run-time StarPU [17] to implement DSP applications on heterogeneous clusters. With SignalPU, programmers do not have to manipulate StarPU API to deal with the algorithm expression and the architecture specificities, like memory management, task creation and synchronization, execution placement, and others. Using the DFG model of computation, they can implicitly express tasks, data, and graph parallelisms in their implementations, to optimally take advantage of the hardware. In addition, because it is based on StarPU, our PPM makes use of shared and distributed memory architectures. It deals with many-node clusters using the MPI, and it can target several accelerators: GPU, Xeon Phi, Cell..etc

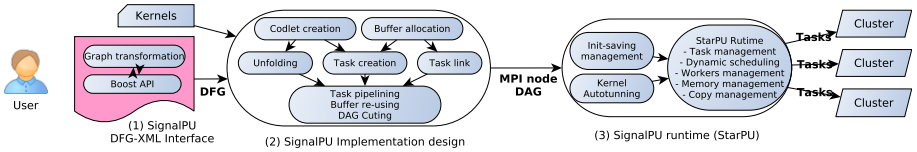

Figure 2. SignalPU design: Three levels of processing

We present our proposed PPM in the form of three levels of processing, as shown in Figure 2. First, the user can easily express the application with a DFG using an XML interface. Thus, the user is freed to manipulate the StarPU API for: tasks and codelet creating, buffer management between pairs of tasks, and jobs submitted onto the corresponding PE. Second, at run-time, the DFG-XML is analyzed and transformed using some DSP-adapted functionalities (e.g., graph-unfolding techniques [36], pipelining of tasks, buffer re-use, MPI multi-node distributions) to produce a DAG of the independent tasks distributed over MPI nodes. The goal is to express all levels of parallelism, while limiting the overhead due to memory and task management. Also, at this level of processing, the user does not have to deal with any API to unroll the main loop, manage necessary memory buffers, submit and synchronize tasks, or distribute the processing over MPI nodes. Finally, at the third level, the StarPU run-time is in charge of task management, scheduling and load balancing, data-flow dependency, and PE management. However, we have added two optimizations: initialization saving, and auto-tuning, to enhance the performance of each task. Next, we illustrate these levels through the synthetic example application presented in Figure 3.

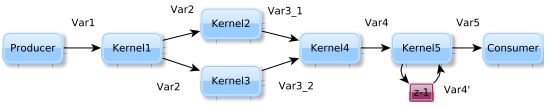

Figure 3. DFG model of an example of a DSP application. Z-1 is a delayed auto-dependence.

4.1. *Level 1: SignalPU DFG-XML interface*

In this step, we propose an interface that is based on the DFG model of computation and the XML description, where the programmer can easily express the application. First, the programmer has to describe each operator (function) of the algorithm in the form of a node (vertex) using an XML structure. The programmer has to input the name of the functions, which will be called in the code, the number of input and output arguments of these functions, and the architecture kind that corresponds to each of them (e.g., CPU, GPU,..). Second, the programmer has to describe in the same manner all of the data flows in the form of graph edges, with a structure that includes information about the type and size of the data that exchange between operators. After this, a DFG-XML of the application is produced. Finally, the programmer has to include its functions in the code, written in the form of a combination of two sub-functions: an initialization one, and an execution one (Init(..), Exec(..)). The XML code below (Table I) gives an example of how a node and an edge of the DFG can be described.

```
<!-- ************************************
          NODES
*************************************-->
<node id="n0">
  <data key="k_node_id">n0</data>
  <data key="k_node_name">producer</data>
  <data key="k_kernel">producer(out0)</data>
  <data key="k_kind">CPU/GPU</data>
  <data key="k_time">7</data>
</node>
```

```
<!-- ***********************************
          EDGES
************************************ -->
<edge source="n0" target="n1">
  <data key="k_edge_name">n0_to_n1</data>
  <data key="k_port_from">out0</data>
  <data key="k_port_to">in0</data>
  <data key="k_type">mat</data>
  <data key="k_size">262144</data>
</edge>
```

Table I. A portion of the XML code that represents a description of one node and one edge.

4.2. Level 2: SignalPU implementation

As illustrated in Figure 4, at run-time, the XML description of the DFG is parsed with the Boost Graphml Reader [37], and the StarPU API is invoked to generate an implementation that can highlight all of the levels of parallelism (i.e., task, data, graph parallelism) to be efficiently executed on the cluster. Several instances of the graph will be submitted to StarPU run-time using the unfolding techniques presented in the following section. The later sections describe first the buffer reuse strategy to avoid unnecessary memory allocation and release, second, the pipelining functionality to limit the number of submitted tasks, and finally, the decomposition of the DAG of tasks over MPI nodes.

Graph unfolding Unfolding is a transformation technique to duplicate the functional blocks to reveal hidden parallelism of the DSP program in such a way that preserves its functional behavior at its outputs. It is usually used for low-level implementation on FPGA, DSP, and ASIC hardware [36]. We propose to use it to unroll the main loop of the application and to increase task parallelism, to achieve better occupancy and load balancing across processing elements of the cluster. Next, we exemplify that technique through the z transformation of a digital signal: $Z_n = a * X_n + b * Y_n$

If k is the kth iteration of the J unfolding. So, the J-unfolded application becomes:

$$\begin{cases} Z_{j*k} = a * X_{j*k} + b * Y_{j*k} \\ Z_{j*k+1} = a * X_{j*k+1} + b * Y_{j*k+1} \\ .... \\ Z_{j*k+(j-1)} = a * X_{j*k+(j-1)} + b * Y_{j*k+(j-1)} \end{cases}$$

Thus, as illustrated in Figure 4, at run-time, we express first each operator of the DFG-XML with a StarPU structure, called a "codelet", which denotes tasks and contains all of the information about the corresponding operator (e.g., number of input arguments, number of output arguments, function identifiers, architecture kind). We then iteratively J-unfold the DFG to create a DAG of tasks. The unfolding degree can be adjusted dynamically according to the measured occupancy, the load balancing, and the available resources.

Buffer reuse Many DSP applications have static communication patterns. So, to avoid unnecessary overhead due to buffer allocations and freeing, a fixed number of buffers is allocated according to the available resources (i.e., the available memory on the node), the DFG-XML description of the application (i.e., data type, data size, number of dependencies), and the unfolding degree. Each time an iteration (graph level) of the original submitted DAG is finished, its buffers are used by the next submitted iteration (graph level). To implement this mechanism and to control the unfolding degree, we use a semaphore initialized with J. Each time an iteration of the original graph is finished, a semaphore is given to the main loop of the control thread that is waiting for a semaphore before launching a new iteration.

Tasks pipelining In addition, all of the DSP applications process a high number of iterations, so to reduce overhead due to task management (e.g., dependency management, scheduling, task status updating), we limit the number of submitted tasks using pipelining functionality. So, at run-time, by using semaphores, only a fixed number of pipeline levels are submitted, where this number is J, the unfolding degree, and each pipeline level corresponds to one graph level in the DAG. As illustrated in Figure 4, the pipeline depth (length) is four, and a semaphore locks all of the buffers of the four graph levels. So, to add (submit) a new graph level to the pipeline, it is necessary to wait for the termination of the processing of one iteration (graph level) of the submitted graph.

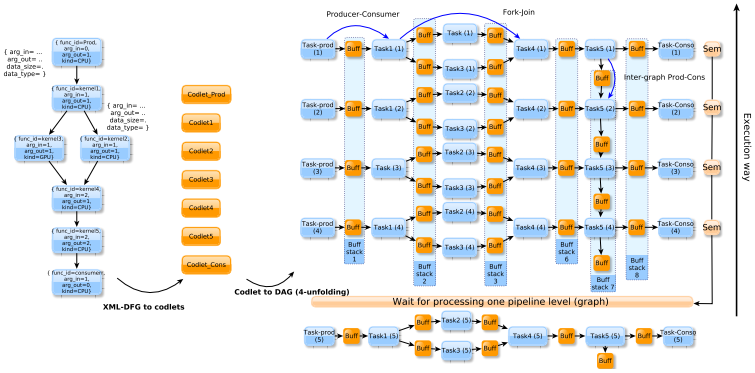

Figure 4. Illustration of the implementation process of the DSP application shown in Figure 3

MPI multi-node distribution Finally, our design flow allows the user to deploy the application over multi-nodes using the StarPU-MPI [38] library. Indeed, the user can distribute the processing costs over the MPI nodes in a SIMD way by assigning to each node the unit of data needed to be processed. So, using a predefined function called $MPI\_data\_schedule()$, the user simply has to specify the number of the iterations that the node has to process. Of note, the DAG of tasks is dynamically divided across nodes. Below we give a simplistic example of how the user can process this function:

```
1  int MPI_data_schedule(int loop, int rank) { return( loop % rank ); }
```

At run-time, each MPI node processes its part of the DAG composed of graph levels corresponding to the number of iterations, as specified by the user. All of the MPI nodes follow the same design described by the functionality cited above. So each node dynamically unfolds its part of the DAG of tasks, and it automatically re-uses buffers and pipiline tasks. Also, each MPI node uses the functionality we cite next.

4.3. Level 3: SignalPU run-time (StarPU)

In this step, as illustrated in Figure 5, we focus on the efficient execution of the tasks. On each MPI node, StarPU run-time manages the submitted sub-DAG of the tasks generated in the previous level. StarPU dynamically schedules them by taking into account the data-flow dependencies between the tasks, to guarantee data coherence and to limit superfluous synchronizations. At run-time and on each MPI node, the submitted unfolded part of the DAG of tasks relative to this node are stored on a pile and scheduled. StarPU dynamically assures that they are executed on the ”best” computation unit using some heuristic algorithms to balance the load, such as work-stealing or HEFT [17], while taking into account the location of the data. Also, to mask communication cost, we enable asynchronous data copy between accelerators and the host, and we activate the streaming processing to increase the occupancy of the computation units. Finally, to reduce the execution time of each task, we propose two DSP-specific functionalities, as described below:

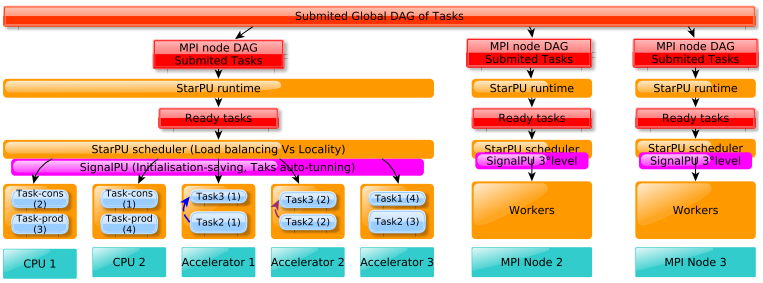

Figure 5. SignalPU runtime levels.

Initialization saving Many DSP operators include an initialization part to initialize the data structures used for the computation. This implementation phase can be long relative to the

computation time. For example, the Gabor filter used in the Saliency application presented in section 5 has an initialization phase of about 380 milliseconds, compared to the computation phase that has a duration of about 2.5 milliseconds on a Quadro4000 GPU. Thus, it is obvious that this phase has to be executed only one time for each processing element on which the operator is mapped. However, no such native mechanism exists in StarPU. Therefore, we propose that the designer specifies its operators with two sub-functions: an initialization one, named "Init", which is executed only once on each processing element, and an execution one, named "Exec", which contains the computations. With this mechanism, the Init sub-function is executed only once on each processing element and the initialized data are preserved.

Kernels auto-tuning As stated above, clusters today are heterogeneous in terms of the processing element types (e.g., CPU, GPU, Xeon Phi), but they can also be heterogeneous for one processing element type. For example, a cluster can contain GPUs of different generations, with different computation capabilities. The GPUs can be different in terms of, for example, the number of cores and registers, the size of the memory levels, the throughput, and the latency. The GPU performances are particularly affected by the threads per block parameter. Moreover, the optimal value of this parameter can depend on the GPU characteristics. Thus, we propose an iterative-specific functionality to determine the best thread per block parameter for GPU tasks. We propose a function interface, which iteratively explores a three-dimensional space of parameters (threads per block (x,y,z)) limited by the designer, which saves the best ones for each kernel for each device. Thus, due to the auto-determined most efficient parameter, the processing time of tasks is enhanced, while the user is freed to adapt the code of each kernel according to each processing element.

5. EVALUATION

In this section, we present some comparisons, experimentation, and results used to validate our approach. We use two DSP applications that we implement using: the SignalPU, MPI+OpenMP+CUDA, and PACCO tools, to provide expressiveness and performance comparisons. First, we present these applications and compare their implementation according to the time effort and abstraction level. Second, we show and discuss the results of each experiment by comparing the performances.

5.1. Applications and implementations

Here, we present the two DSP applications and describe their implementation. First, a synthetic application as a computing intensive case with inherent task parallelism is presented and implemented using SignalPU and MPI+OpenMP+CUDA. Second, we show a relevant real-world application (image processing), which is a relatively communication-bound case without inherent task parallelism, with the implementation of the SignalPU and PACCO tools.

Algorithm 1: Synthetic DSP application.

Require: A image r_im of size $w \cdot l$,
Number of images (Nbr)
Ensure: Processed image
for each $image_i$ in Nbr **do**
  $(Var_{1_1}, Var_{1_2}) \leftarrow Producer()$
  $Var_{2_1} \leftarrow PixelProcesSimu(Var_{1_1}, k_1)$
  $Var_{2_2} \leftarrow PixelProcesSimu(Var_{2_1}, k_2)$
  $(Var_{3_1}, Var_{3_2}, Var_{3_3}) \leftarrow JoinFork(Var_{2_1}, Var_{2_2})$
  $Var_{4_1}, \leftarrow PixelProcesSimu(Var_{3_1}, k_3)$
  $Var_{4_2}, \leftarrow PixelProcesSimu(Var_{3_2}, k_4)$
  $Var_{4_3}, \leftarrow PixelProcesSimu(Var_{3_3}, k_5)$
  $Processed\_image \leftarrow Consumer(Var_{4_1}, Var_{4_2}, Var_{4_3})$
end for

Algorithm 2: Static pathway of the visual model.

Require: An image r_im of size $w \cdot l$
Ensure: The saliency map
$r\_fim \leftarrow Hanningfilter(r\_im)$
$cf\_fim \leftarrow FFT(r\_fim)$
for $i \leftarrow 1$ **to** orientations **do**
  for $j \leftarrow 1$ **to** frequencies **do**
    $cf\_maps[i,j] \leftarrow GaborFilter(cf\_fim, i, j)$
    $c\_maps[i,j] \leftarrow IFFT(cf\_maps[i,j])$
    $r\_maps[i,j] \leftarrow Interactions(c\_maps[i,j])$
    $r\_normaps[i,j] \leftarrow Normalizations(r\_maps[i,j])$
  end for
end for
$saliency\_map \leftarrow Summation(r\_normaps[i,j])$

5.1.1. Synthetic digital signal-processing applications To allow users to easily test our PPMs, we designed a library of operators for several PE types with a parameterized workload. We use this library to build test cases with different graph structures, communication, and computation loads. In this paper, we present the experimentation based on the synthetic application described in Algorithm 1, and this application simulates a real-word computation-intensive application that includes different granularities of operators that can be executed on both CPUs and GPUs, and which

```
1  void pixPrSimuCuda(float* H_in,float* D_in,
2  float*H_out,float*D_out,k,int size,int dev);{
3  cudaMemcpyAsync(D_in,H_in,size,HtoD,stream_in);
4  Auto_tune(nblocks,threadsB);
5  kernelCuda<<<nblocks,threadsB,stream_exec>>>
6  (D_in,D_out,k,n);
7  cudaMemcpyAsync
8  (H_out,D_out,size,DtoH,stream_out); }
9
10 int main(int argc, char **argv)
11 {
12  int high = 512 ; int with=512;
13  int size_img = high*with;
14  const int size_loop = (int) atoi((argv[1]));
15  const int unfolding = (int) atoi((argv[2]));
16  int rank; int n=0; int dev;
17
18 MPI_Init(&argc,&argv);
19 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20
21 #pragma omp parallel num_threads(unfolding)
22 { float * H[nb_buf]; float * D[nb_buf];
23
24 Host_alloc(H); Device_alloc(D);
25
26 #pragma omp for private(n) private(dev)
27 if (rank==0) for(n=0; n<size_loop_1; ++n)
28 else for(n=size_loop_1+1; n<size_loop_2; ++n)
```

```
1  {
2  dev=load_balance(n,GPUs_number);
3
4  #pragma omp task depend(out:H[0],H[1])
5  producer( H[0], H[1] , size_img );
6
7  #pragma omp task depend(in:H[0]) depend(out:H[2])
8  PPSimuCuda(H[0],D[0],H[2],D[2],7,size,dev);
9
10 #pragma omp task depend(in:H[1]) depend(out:H[3])
11 PPSimuCuda(H[1],D[1],H[3],D[3],3,size,dev);
12
13 #pragma omp task depend(in:H[2],H[3])
14  depend(out:H[4],H[5],H[6])
15 ForkJoin(H[2],H[3],H[4],H[5],H[6],size);
16
17 #pragma omp task depend(in:H[4]) depend(out:H[7])
18 PPSimuCuda(H[4],D[4],H[7],D[7],6,size,dev);
19
20 #pragma omp task depend(in:H[5]) depend(out:H[8])
21 PPSimuCuda(H[5],D[5],H[8],D[8],2,size,dev);
22
23 #pragma omp task depend(in:H[6]) depend(out:H[9])
24 PPSimuCuda(H[6],D[6],H[9],D[9],2,size,dev);
25
26 #pragma omp task depend(in:H[7],H[8],H[9])
27 Consumer(H[7],H[8],BufH[9]);
28   }
29 }
```

Table II. An MPI+OpenMP+CUDA pseudo code of a synthetic application. Left the table a. Right the table b.

contains multiple dependencies (inherent task parallelism). The k parameters are used to adjust the workload of each operator. In Figure 6, we show the DFG-XML description of the synthetic application. This DFG-XML description of the application is the major part of the expression that the user has to do. The user only has to model the algorithm in the form of a DFG of operators, by giving information about edges and nodes. The user focuses only on the expression of the algorithm without manipulating any particular API or Pragma syntax. On the contrary, in an MPI+OpenMP+CUDA implementation represented in Table II, the user has first to express the functions in the form of tasks using the **omp task** directive (Table II-b, lines 4, 7, 10, ...). The user has to link these using the **omp depend** directives (Table II-b, lines 4, 7, 10, ...) to data-synchronize the functions. The user has to unfold the main loop over the parallel region of the threads using the **omp parallel for** directive (Table II-a, lines 21, 26). In addition, to target the GPU, the user has to call the CUDA function (Table II-b, lines 8, 11, 21,...), and has to put allocations out of the loop to avoid overhead due to allocation and free (Table II-a, line 24). The user has to asynchronously copy-in and copy-out, and call the CUDA functions to overlap the communication and computation (Table II-a, lines 3, 5, 7), and has to manually and statically load the balance operators over the GPUs (Table II-b, line 2). The user also has to tune the CUDA parameters of each operator according to each GPU (Table II-a, line 4). Finally, to distribute the computation over the nodes, the user has to use the MPI API to create the main process in each node by specifying which data each MPI process has to treat (Table II-a, lines 27, 28). Thus, in comparison to the SignalPU implementation, the user has to handle three PPMs. The user has to take care of the memory allocation, and the data communication over discrete memory. The user has to manipulate the directive pragma to create the tasks and to link them. The user has to manage the load balancing and some of the other functionalities close to the architecture characteristics. Thus, it is easier for the user to use SignalPU to implement DSP applications on heterogeneous clusters.

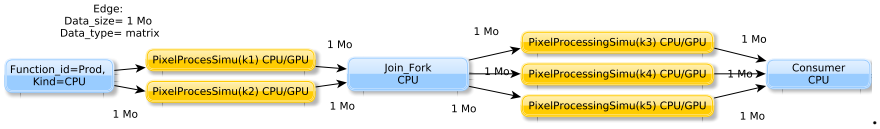

Figure 6. The DFG-XML model of the synthetic application.

5.1.2. Saliency application We also experiment with our approach on a real-world application based on the primate retina, with the visual saliency model used to locate the regions of interest; i.e., the capability of human vision to focus on particular places in a visual scene. For implementation, we use Algorithm 2 that was described in the preliminary work of our team [39, 40]. To

Node	CPU	GPU
Archi1	Intel-i7 core (4 Cores)	2 NVIDIA Quadro 4000 + NVIDIA Quadro 2000
Archi2	Intel-i7 core (4 Cores)	NVIDIA GTX TITAN + NVIDIA GTX 780
Archi3	Intel-i7 core (4 Cores)	NVIDIA GTX 680 + NVIDIA GTX 680
Archi4	Intel-i7 core (4 Cores)	NVIDIA GTX 480

Table III. Architecture cluster used for experimentation.

implement the application with our programming models, the first step is to model Algorithm 2 with a DFG-XML description using the SignalPU interface. For this, we represent each function ($Hanningfilter()$,...) with a node in the graph, which includes their own characteristics (e.g., architecture kind, input arguments, output arguments, function identifier). Then, we represent the data flow between each operator with an edge in the graph, which includes its characteristics (e.g., data type, data size, input/ output arguments). In Figure 7, we show the DFG-XML result of this step. The second step is to include the function code in the program, where each function is in the form of initialization and execution sub-functions (Init(..), Exec(..)). Thus, we do not have to use the StarPU API to describe the application tasks. In comparison, PACCO implementation is the same, regarding what the user has to do. However, the execution model is different. Indeed, the PACCO execution model is an MPI+Pthread+CUDA implementation based on an iterative BSP model [32], where the execution of the operators and the transfer of the data are overlapped inside each synchronized iteration. Thus, in contrast, SignalPU is more adapted to DSP implementations because it is a data-flow synchronization.

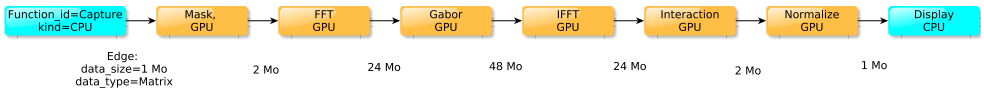

Figure 7. The DFG-XML model of the visual saliency application.

5.2. *Experience and results*

In this subsection, we describe the experimentation we carried out to evaluate the performance of our approach, and we present and discuss the results obtained. The architecture used for the experimentation is a heterogeneous CPU-GPU cluster that includes four nodes, as described in Table III, connected via an infiniband network. The dataset used for the experimentation is a set of images with 512x512 pixels.

5.2.1. Global performance The first experiment we present is the processing of a number of images on hardware configurations. The aim is to measure the scalability of our implementation and how much it can take advantage of the parallelism levels (e.g., task, data and graph parallelism) and the capability of the architecture.

	1 Core (1CPU)	2 Cores (1CPU)	3 Cores (1CPU)	4 Cores (1CPU)	8 Cores (2CPUs)	12 Cores (3CPUs)	16 Cores (4CPUs)	1Core +1Quadro 4000	1Core +2Quadro 4000	1Core +2Quadro4000 +1Quadro 2000	2CPU+4GPUs (Archi1+Archi2)	3CPU+6GPUs (Archi1+Archi2 +Archi3)	4CPU+7GPUs (Archi1+Archi2 +Archi3+Archi4)
MPI+OpenMP+CUDA (Time)	197,00	99,67	68,17	51,17	32,59	21,52	15,34	3,58	2,27	1,93	0,95	0,77	0,72
MPI+OpenMP+CUDA (Speedup)	1,00	1,98	2,89	3,85	6,04	9,15	12,84	54,98	86,91	101,90	207,37	255,84	273,61
SignalPU (Time)	198,33	100,67	68,83	51,83	36,50	25,73	19,35	3,25	2,12	1,58	0,65	0,45	0,39
SignalPU (Speedup)	1,00	1,97	2,88	3,83	5,43	7,71	10,25	61,03	93,70	125,26	305,13	440,74	508,55

Figure 8. Global performance comparison of the SignalPU versus MPI+OpenMP+CUDA implementations of the synthetic application. Time in minute for processing 1000 images

SignalPU versus MPI+OpenMP+CUDA First, we process 1,000 images with the computation-intensive synthetic application on several CPU-GPU configurations using these two implementations: a SignalPU with an unfolding of 10, compared to the MPI+OpenMP+CUDA low-level implementation presented above in section 5.1.2. The results of the comparison are shown in Figure 8. For the CPU only configurations, we can note that the speed-up is proportional to the number of cores in the two implementations, due to the task, data and graph parallelism. Task parallelism is achieved through the expression of task dependencies. Data and graph parallelism are exploited by unfolding the graph. On CPUs-only cluster, OpenMP implementation get slightly better results than SignalPU. That is due to the rutime overhead and the dynamic management of tasks. For the CPU-GPU configurations over multi-nodes architeture, performances are enhanced

by exploiting data parallelism on GPUs. So, for SignalPU implementation, we obtain about 61x speed-up if we use 1 CPU core + 1 Quadro4000 GPU, compared to a 1-core configuration. For MPI+OpenMP+CUDA, we only reach about 55x. The difference in these results is due to the model of execution. Indeed, with MPI+OpenMP+CUDA, the execution model is the Fork-Join model, where threads of unfolding parallel regions concurrently invoke CPU and GPU system drivers to execute tasks. However, in SignalPU implementation, the model of execution is the scheduling of DAG of tasks over the CPU and GPU workers, thus the invocation is done only one time for each device. For one CPU core + 2 Quadro4000, we get 93x speed-up of performance with SignalPU, and only about 69x with MPI+OpenMP+CUDA. By using 1 CPU core + 2 Quadro4000 + Quadro2000, we get a speed-up of about 125x with SignalPU, and only about 89x with MPI+OpenMP+CUDA. These differences in the result are for the same reasons as the execution models. In addition, the dynamic scheduler of SignalPU is more efficient because it uses some optimizations, such as out-of-order and data prefetching, which increase device occupancy. Finally, the performance still increases with the number of nodes: we obtain a speed-up of 508x with SignalPU on 4 nodes exploiting a total of 4CPUs+7GPUs. However, we obtain a speed-up of about 274x with the MPI+OpenMP+CUDA implementation on the same hardware configuration. We can explain the differences in scalability by the weakness of the static scheduling of the MPI+OpenMP+CUDA solution compared to the dynamic one used in SignalPU implementation. Thus, in comparison to an equivalently tuned MPI+OpenMP+CUDA implementation, SignalPU produces the best performance on the heterogeneous cluster experiments, due to the dynamic run-time and the DAG of task scheduling model of execution.

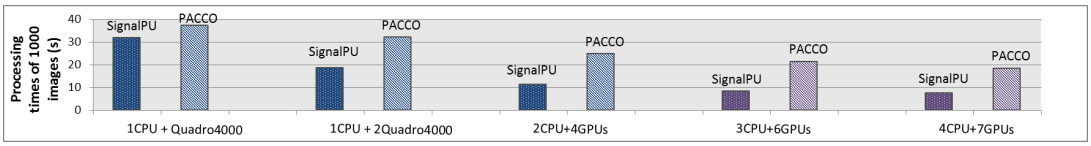

Figure 9. Comparison of global performance of the visual saliency application implementations.

SignalPU versus PACCO(MPI+Pthread+CUDA) In the same manner, we process 1,000 images with the communication bounded saliency application, and we compare our SignalPU with an unfolding degree of 10 (J=10) to the PACCO optimized BSP implementation based on MPI+Pthread+CUDA cited in subsection 5.1.2. The comparisons of both of these implementations are shown on Figure 9. For the first comparison of the results illustrated in the first and the second groups of bars in Figure 9, we use a Quadro 4000 + 1 CPU core, and we obtain a better performance for the SignalPU implementation (32 s) compared to the performance of the PACCO implementation (38 s). This is due to the synchronization of the task. Indeed, in the PACCO implementation, the synchronizations between iterations (graphs) are carried out using a global synchronization barrier, which leads to an imbalanced load. However, in our SignalPU implementation, we use only data dependencies to synchronize the processing. So, as soon as the data is available, new tasks are launched. The second hardware configuration (2 Quadro 4000 + 1 CPU core) enhances the performance of both of the implementations: it reduces the processing time for the SignalPU implementation, and reaches a speed-up of 1.7x, due to the graph and data parallelism performed by unfolding the DFG. However, for the PACCO implementation, we only reach 1.2x speed-up compared to the previous implementation, by only exploiting data parallelism over the pipelining of the tasks. This difference is explained by the scheduling of the tasks. Indeed, the PACCO implementation does not scale well because the scheduling is static, while the DFG of the application has few imbalanced execution-time coarse-grained operators. Thus, the most loaded device delays the iteration. In contrast, in the SignalPU implementation, the dynamic scheduling of several unfolded graphs (iterations) allows the optimal balance of the load between the GPUs (as discussed for the next experiment; section 5.2.2). For the multi-nodes configurations, the performance of both implementations is enhanced by balancing the data processing over the nodes. In the SignaPU implementation it is achieved throught the MPI distribute functionality. For the PACCO implementation, we manually unfold the application graph of and map each actors

on the architecture. Thus, we reduce the global processing time of each implementation on scalable cluster. Finally, we get about 4.2x speed-up on 4CPUs+7GPUs compared to the first implementation executed on 1 CPU Core+1 Quadro 4000. However, for the PACCO implementation, we get only an about 2.1x speed-up with the same experimental setups. The differences can be explained by the previously mentioned reasons: synchronization and load balancing over processing elements and nodes. Indeed, with SignalPU, we balance the load in a SIMD way over the MPI nodes using the $MPI\_data\_schedule$ function described in 4.2. whereas in the PACCO implementation, the iteration is manually and statically unbalanced over the nodes, which delays its processing time. In addition the global synchronization of at the end of the super step of BSP model of execution blocks the scheduling of future iterations.

5.2.2. Unfolding versus scheduling In this second experiment, we are interested in indicating the performance gain obtained by combining the unfolding techniques and the dynamic scheduling. We experiment with the processing of 1,000 images on a CPU-GPU hardware configuration, with both applications using the following SignalPU implementation:

1. **Static scheduling**: This is an iterative implementation without unfolding (J=1), where we statically map each operator (node of DFG-XML) to a PE. An exhaustive exploration of all of the possible static placements was carried out to select the best placement.
2. **Dynamic scheduling**: In this implementation without unfolding (J=1), we use in addition a dynamic scheduler of StarPU to balance the load inside the loop.
3. **Static scheduling with unfolding**: In this implementation we 10x unfold the main loop, but the scheduling is the same as the first implementation.
4. **Dynamic scheduling with unfolding**: In this last implementation, the test includes both functionalities, the dynamic scheduling (work stealing), and the graph unfolding (J=10).

In Figure 10, which is related to the synthetic application, and Figure 11, which is related to the saliency application, we show the performance of the four processed implementations of each application on a CPU+GPU hardware configuration. The results are presented in the form of four groups of bars, where each group of bars represents the global execution time of an implementation. Each bar composing a group represents the global processing time on each processing element, and its decomposition is: execution time, labelled as ”Execution”, which represents the effective processing time on the computing unit; sleeping time, labelled as ”Sleep”, which represents the time when any computation is done on the device; and the overhead time, which is labelled as ”Overhead” and represents the necessary time to manage the work and the device.

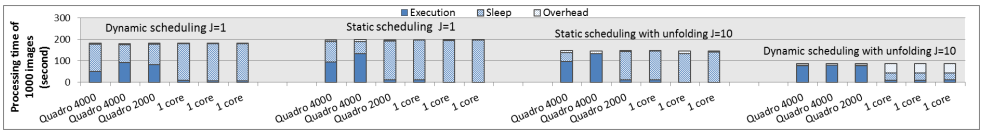

Figure 10. Scheduling versus unfolding of the synthetic application.

The first group bars in Figure 10 (left) illustrates the global processing time of 1,000 images with the dynamic scheduling implementation. This represents a bad performance for two reasons. First, the load balancing between the GPUs is done, but it is poor, as shown in the ”Execution” section. The scheduler cannot optimally share the jobs between the GPUs because of the granularity of the tasks (coarse grained). Second, the sleeping time of each GPU is high, as shown in the ”Sleep” section, because of the iterative form of the implementation, which forces available devices to wait for the slowest one.

In the second group of bars in Figure 10, labelled ”Static scheduling”, the performance is enhanced in the saliency implementation (Figure 11), because the optimal manual scheduling is more efficient than the previous dynamic scheduling, so the iteration time is reduced, and thus the execution times (”Execution” section) and sleeping times (”Sleep” section) on the GPUs decrease. Note in the saliency application case that only 1 GPU (Quadro400) is used to process

all of the operators. The task parallelism is not exploited because the application design does not contain inherent parallelism. However, for the synthetic application, the performances decrease in comparison to the first implementation. This is due to the imbalance of the load inside the iteration obtained by statically mapping the operators on GPUs.

The third group of bars in Figure 10, labeled "Static scheduling with unfolding", shows more performance improvement compared to the second one. Indeed, even if the execution times ("Exec" section) do not change, the operators are processed with the same placement, and the sleeping times are consequently reduced due to the unfolding techniques, which allow the overlapping of the processing of several iterations. Thus, the devices do not wait for the last task of the iteration before processing their next tasks.

The last group of bars in Figure 10, labeled "Dynamic scheduling with unfolding" represents the best performance compared to the other implementations. The execution time is enhanced due to the load balancing that is done by the dynamic scheduler, dealing with enough numbers of tasks made available by the unfolding techniques allowing it to more efficiently distribute jobs between processing elements. Thus, as proposed in our PPM, it is interesting to combine the use of unfolding techniques and dynamic scheduling to efficiently take advantage of the device availability.

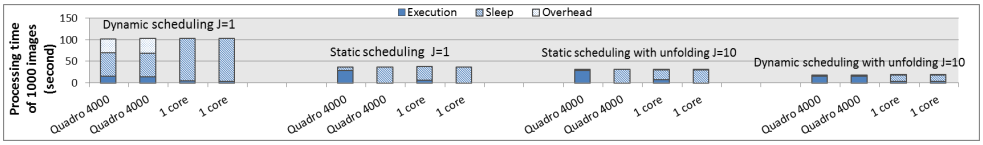

Figure 11. Scheduling versus unfolding of the saliency application.

5.2.3. Overhead In that experiment, we focused on the estimation of the overhead due to the run-time management (e.g., management of device, task, memory, dependencies) and the impact of using the proposed functionalities (e.g., pipelining of the task, buffer reusing) described in sub-section 4.2. So, for that, we use the SignalPU implementation of the synthetic and saliency applications to process an increasing dataset of images (from 1,000 to 10,000), and we measure the percentage of overhead on the global execution time. In Figure 12, we show the evolution of this percentage for three implementations: a PACCO implementation of saliency application shown in the blue line marked with diamonds, which converges to 2% of the overhead rate because it uses static scheduling and does not manage data synchronization (static run-time). A SignalPU implementation of the synthetic and saliency applications, which is illustrated in red and green lines in Figure 12, indicated with squares and triangles, respectively, which stabilizes to 7% of the overhead rate. It should be noted that even if the overhead of the SignalPU implementation is greater than the overhead of the PACCO implementation, the SignalPU implementation exploits the hardware better and leads to better performance.

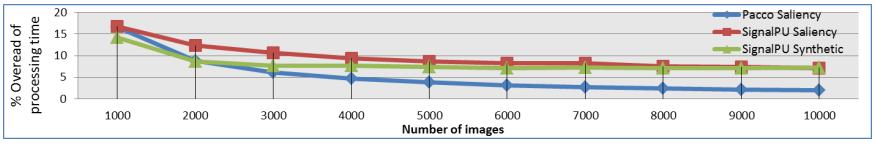

Figure 12. Percentage of overhead in the global processing time of the SignalPU implementation.

5.2.4. Task performance The last experiment we present is the evolution of the processing time of the tasks during the iterations. The goal is to show the impact of the proposed functionality (e.g., initialization saving, operator auto-tuning). For each application, we measure the processing time of two operators: $PixelProcesingSimu()$ for the synthetic application, and $Interaction()$ for the saliency application, during the first 15 iterations on two GPUs. The results are shown in Figure 13 for the saliency and the saliency applications. First, we note that the initialization part of each operator that is represented with a dashed line at the first iteration of each line is done only once for each device, due to the proposed initialization saving optimization. Second, we show that the computation times can vary from single up to double, according to the threads per block parameter,

and that the exploration converges to an optimum which depends on the GPU architecture. The tuning is carried out iteratively, and the best parameters are saved. So, we finally obtain (128,1,1) and (256,1,1) for the $PixelProcessingSimu$ operator on the GTX780 and Quadro4000 and (32,8,1) and (32,16,1) for the $Interaction$ operator on the GTX780 and Quadro4000. Thus, due to the proposed execution model functionality of initialization saving and task auto-tuning, we take advantage of the hardware capability to enhance the execution time of the tasks, while the designer does not have to worry about this work.

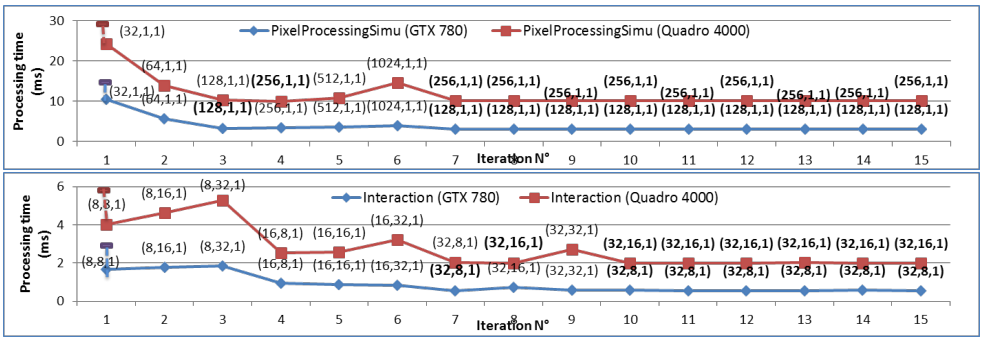

Figure 13. Evolution of the execution time of the task according to the thread per block parameter: $PixelProcessigtime$ of synthetic application (top), and $Interaction$ of Saliency application (bottom).

6. CONCLUSION

In this paper, we have presented SignalPU, a DSP-domain specific PPM based on a DFG model of computation and a dynamic run-time (StarPU) that allows programmers to easily and efficiently implement their DSP applications on parallel and heterogeneous architecture. Thus, we have shown that it is possible to construct a domain-specific PPM to achieve both productivity and performance using a dynamic task-based run-time. In this study, we have presented PPMs and have classified them according to the abstraction level. After that, we presented related studies on the implementation of DSP applications on parallel and heterogeneous clusters, and we classified and discussed these. Thus, according to the extracted characteristics of the DSP implementation, we have proposed a novel high-level abstraction approach of PPM, named as SignalPU, which is structured at the following three levels: (1) DFG-XML interface: The application is easily modeled in the form of a DFG model using an XML interface. The user does not need to deal with the StarPU library to express an application. (2) Implementation: Using some functionalities (e.g., graph unfolding, buffer reuse, task pipelining, MPI multi-node distribution), the implementation of the application is optimized to optimally exploit all of the levels of parallelism, while overcoming run-time overhead. The user does not need to worry about architecture specificity to implement an application. (3) Run-time: The execution is effectively managed to take advantage of the availability and the capability of the devices. The user gets an efficient sequence of execution due to the StarPU scheduler, and each task is enhanced on each device due to the proposed functionalities of task auto-tunning and initialization saving. Finally, the implementations of two applications were presented. We have shown how easy it is to implement these in comparison to an MPI+OpenMP+CUDA expression, while getting enhanced performances. We have shown that this approach offers better performance than PACCO, a BSP tool based on MPI+Pthread+CUDA that we previously developed. In future work, we would like to generalize our approach on more complex DSP applications, like dynamic ones represented with Boolean data flow, and on a data-dependent application. Also, we would like to study how to offer to the user the choice of the unfolding degree, taking into account the targeted clusters. We also want to adapt the proposed auto-tunning technique for other parameters and other kinds of operators. Finally, we will work on automatic multi-node load balancing based on the unfolding.

REFERENCES

1. Skillicorn DB, Talia D. Models and languages for parallel computation. *ACM Comput. Surv.* 1998; **30**:123–169.
2. Cameron C. Parallel ray tracing using the message passing interface. *IEEE Transactions* 2008; .
3. Pacheco PS. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1996.
4. Kjeldsen T, Lassen L, Hemmsen M. Synthetic aperture sequential beamforming implemented on multi-core platforms. *Ultrasonics Symposium (IUS), 2014 IEEE International*, 2014; 2181–2184.
5. Munshi A, Gaster B, Mattson T, Ginsburg D. *OpenCL Programming Guide*. OpenGL, Pearson Education, 2011.
6. Board OAR, Shreiner D, Woo M, Neider J, Davis T. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1*. 6th edn., Addison-Wesley Professional, 2007.
7. Kim S, yeol Sohn H, Chang JH, kyoung Song T, Yoo Y. A pc-based fully-programmable medical ultrasound imaging system using a graphics processing unit. *Ultrasonics Symposium (IUS), 2010 IEEE*, 2010; 314–317.
8. Sanders J, Kandrot E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st edn., Addison-Wesley Professional, 2010.
9. Bozejko W, Dobrucki A, Walczynski M. Parallelizing of digital signal processing with using gpu. *Signal Processing Algorithms, Architectures, Arrangements, and Applications Conference Proceedings (SPA), 2010*, 2010; 29–33.
10. Pham T. Parallel implementation of geodesic distance transform with application in superpixel segmentation. *Digital Image Computing: Techniques and Applications (DICTA), 2013 International Conference on*, 2013; 1–8.
11. Chandra R, Dagum. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2001.
12. Kirk DB, Hwu WmW. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st edn., Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2010.
13. Colgrove M. 5x in 5 hours: Porting a 3d elastic wave simulator to gpus using pgi accelerator. Technical News from The Portland Group 2012.
14. Hiram E. *Openhmpp*. Placpublishing, 2012.
15. Bueno J, Martinell L. Productive cluster programming with ompss. *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, 2011.
16. Sidi Mahmoudi CAST Pierre Manneback. Dtection optimale des coins et contours dans des bases d'images volumineuses sur architectures multicurs htrognes. Rencontres francophones du paralllisme, 2011.
17. Augonnet C, Thibault S, Namyst R. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. *Research Report RR-7240*, INRIA 2010.
18. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.* Aug 1995; **30**(8):207–216.
19. Gautier T, Besseron X, Pigeon L. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. *2007 international workshop on Parallel symbolic computation*, ACM, 2007.
20. Reinders J. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism.* O'Reilly, 2007.
21. Rossbach CJ, Currey J, Silberstein M, Ray B, Witchel E. Ptask: Operating system abstractions to manage gpus as compute devices. *Proceedings of SOSP '11*, SOSP '11, ACM: New York, NY, USA, 2011; 233–248.
22. Lee E, Messerschmitt D. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on* Jan 1987; **C-36**(1):24–35.
23. Bhattacharya B, Bhattacharyya S. Parameterized dataflow modeling for dsp systems. *Trans. Sig. Proc.* 2001; .
24. Gordon MI, Adviser-Amarasinghe S. Compiler techniques for scalable performance of stream programs on multicore architectures 2010; .
25. Pelcat M, Desnos K, Heulot J, Guy C, Nezan JF, Aridhi S. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. *EDERC, 2014 6th European Embedded Design in*, 2014; 36–40.
26. Huynh HP, Hagiescu A, Wong WF, Goh RSM, Ray A. Poster: Automated mapping streaming applications onto gpus. *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, 2012; 1490.
27. Houzet D, Huet S, Rahman A. SysCellC: a data-flow programming model on multi-GPU. *procedia computer science* May 2010; **1**(1):1029–1038.
28. Grotker T. *System Design with SystemC*. Kluwer Academic Publishers: Norwell, MA, USA, 2002.
29. Ghezal N, Matiatos S, Piovesan P, Sorel Y, Sorine M. SYNDEX un environnement de programmation pour multi-processeur de traitement du signal. Mecanismes de communication. *Research Report RR-1236* 1990. Projet SOSSO.
30. Boulos V, Huet S, Fristot V, Salvo L, Houzet D. Efficient implementation of data flow graphs on multi-gpu clusters. *Journal of Real-Time Image Processing* Oct 2012; :n/c.
31. Mansouri F, Huet S, Fristot V, Houzet D. Task migration of DSP application specified with a DFG and implemented with the BSP computing model on a CPU-GPU cluster. *DASIP 2013*, Cagliari, Italy, 2013; 326–333.
32. Valiant LG. A bridging model for parallel computation. *Commun. ACM* Aug 1990; **33**(8):103–111.
33. Schor L, Tretter A, Scherer T, Thiele L. Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of opencl. *ESTIMedia, 2013 IEEE 11th Symposium*, 2013; 41–50.
34. Lee EA. Scheduling strategies for multiprocessor real-time dsp. 1989; 1279–1283.
35. Balarin F, Lavagno L, Murthy P, Sangiovanni-vincentelli A. Scheduling for embedded real-time systems. *IEEE Design & Test of Computers* 1998; **15**(1):71–82.
36. Parhi K, Messerschmitt D. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *Computers, IEEE Transactions on* Feb 1991; **40**(2):178–195.
37. Polukhin A. *Boost C++ application development cookbook*. Packt Publ.: Birmingham, 2013.
38. Augonnet C, Aumage O. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. *EuroMPI 2012*, vol. 7490, Springer.
39. Mansouri F, Huet S, Houzet D. SignalPU: A programming model for DSP applications on parallel and heterogeneous clusters. *HPCC IEEE International Conference*, IEEE Computer Society: Paris, France, 2014; 8.
40. Mansouri F, Huet S, Houzet D. A visual programming model to implement coarse-grained dsp applications on parallel and heterogeneous clusters. *Euro-Par 2014: Parallel Processing Workshops*, Springer, 2014; 141–152.