A Visual Programming Model to Implement Coarse-Grained DSP Applications on Parallel and Heterogeneous Clusters

Farouk Mansouri, Sylvain Huet, and Dominique Houzet

GIPSA-LAB, 11 rue des Mathmatiques
Grenoble Campus BP46, France
firstname.lastname@gipsa-lab.grenoble-inp.fr

Abstract. The digital signal processing (DSP) applications are one of the biggest consumers of computing. They process a big data volume which is represented with a high accuracy. They use complex algorithms, and must satisfy a time constraints in most of cases. In the other hand, it's necessary today to use parallel and heterogeneous architectures in order to speedup the processing, where the best examples are the supercomputers "Tianhe-2" and "Titan" from the top500 ranking. These architectures could contain several connected nodes, where each node includes a number of generalist processor (multi-core) and a number of accelerators (many-core) to finally allows several levels of parallelism. However, for DSP programmers, it's still complicated to exploit all these parallelism levels to reach good performance for their applications. They have to design their implementation to take advantage of all heterogeneous computing units, taking into account the architecture specificities of each of them: communication model, memory management, data management, jobs scheduling and synchronization ... etc. In the present work, we characterize DSP applications, and based on their distinctiveness, we propose a high level visual programming model and an execution model in order to drop down their implementations and in the same time make desirable performances.

1 Introduction

The DSP applications require a high computing power. They process increased data volume (data length) which reach ten or so of Go. Also, data units are represented more and more precisely (data floating point encoding), from single precision (32 digits) to quadruple precision (128 digits). They use complex algorithms (time complexity) in linear, quadratic or exponential time, and are usually constrained in execution time (latency or throughput).

To satisfy this need, it's possible today to get a Tera-flop computing power with a thousand dollars price by using 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) and structured in the form of a cluster of connected nodes with a high bandwidth network.

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 141–152, 2014.

Certainly, these architectures can be the best response for computing power requirements of DSP applications. However, they present some difficulties of use. In fact, to produce performance with that last, the programmer has to deal with heterogeneous computing units using different languages or API. He has to manage synchronization, memory allocation, data transfer and the load balance between the processes. Consequently, programming models are necessary to hide all this hardware specifications, and produce easily and efficiently the desired performance.

In the present work, we propose a visual programming model based on data flow graph (DFG) model which allows to users to easily express their applications. It includes an execution model (Runtime) based on StarPU, and adapted for DSP implementing on heterogeneous platform (CPU, GPU, Cell) and dynamically scheduling of tasks on computational units. First, we present in section 2 the DSP applications and highlight their characteristics and distinctiveness. In the section 3, we give the features to efficiently implement them and discuss the existing programming models to do that. In the section 4, we describe our programming model and explain its conception parts. Finely, in the section 5, we present experimentations and results of applying our model of programming (MoP) on a real world application.

2 Distinctiveness of DSP Applications

DSP applications are in form of repetitive (iterative) processing of data set of input digital signals, which produce an output signals or a results (Fig.1). In the classic algorithmic aspect as shown in the example Algorithm 1, it represents the main loop which iteratively process all data units using several functions.

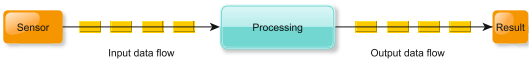

Fig. 1. Illustration of DSP applications

Theses functions, also called operators or kernels, fire each data unit of input signal, where each of them represents an independent processing with some input and output arguments. So, in almost cases of DSP applications, it's possible to model them in form of DFG [9,2], where nodes of the graph represent the kernels and the edges represent the data trading between operators as a flow. The figure (Fig.2) illustrates this DFG model representing the given example in Algorithm 1. That model emphasizes the movement of data and models programs as a series of connections. Explicitly defined input and output arguments synchronize operations. Where an operation runs as soon as all of its inputs become valid. Thus, that model are inherently parallel and allows to user to easily express task parallelism on his application. In addition, in DSP applications, all input data units are processed using the same actors (kernels), for example in video processing, each input image is processed in the same manner using the same

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: $Var_1 \leftarrow Producer()$
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**

algorithm. So, the idea is to overlap the execution of multiple DFG, where each DFG processes one date unit. Also, according to the data kind and the algorithm of each kernel, it's interesting in most of cases to offload the execution of certain of kernels on a massively parallel computation unit (accelerators) like GPU, Xeon Phi or Cell.

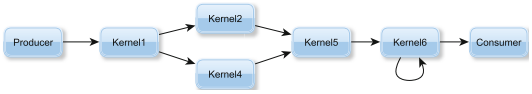

Fig. 2. Data flow graph (DFG) model of DSP application (the example)

Taking into account these characteristics, we set up some rules to apply in implementation and execution of DSP applications on parallel and heterogeneous architectures: First, express task parallelism using the DFG modeling. It allows to highlight the dependencies between the tasks and detect the tasks able to be executed in parallel. Second, optimize data parallelism. Identify the tasks according to their Flynn taxonomy [6]. The MISD (Memory bounds) tasks are oriented to generalist processors and the SIMD tasks (Compute bounds) towards the accelerators. Third, implement graph parallelism. In fact, in order to optimize the occupancy of computing units composing the cluster, deal with several graphs to process several input data unit in the same time.

In the next section (Section 3), 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.

3 Implementing DSP Applications on Clusters

As presented in the preceding section (Section 2), the DSP applications have some characteristics that 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, launch 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 time, 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 [13] or OpenMP [4] to generate threads and express task parallelism on each node of the cluster (shared memory architecture), but also the MPI [11] or PGAS [5] model to manage them by creating processes onto many nodes (distributed memory architectures). He has to use CUDA [14] or OpenCL [10] 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 [8], OmpSS [3] or StarPU [1] 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 MoP, 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 (speci-

fied) to DSP applications as characterized in the section 2 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 DFG model to make easier the application modelling and automatize the generation of the directional acyclic graph (DAG) of tasks in order to adapt StarPU to the implementing of DSP applications on heterogeneous cluster.

4 Proposed Programming Model

We propose, in this section, a visual programming model as an extension of StarPU programming model [1] which we enrich by giving some functionalities specified to DSP applications, in order to allow for programmers to implement easily and efficiently their programs on parallel and heterogeneous clusters. Our MoP is a high level abstraction concept. The programmers don't have to worry about several architecture specificities, like memory management, task creation and synchronization, load balancing etc... They can implicitly express task, data and graph parallelisms in their implementations to optimally take advantage of hardware. Also, because it's based on StarPU, our MoP take in charge shared and distributed memory architectures, and deal with many-node cluster using the messages passing interface (MPI [11]).

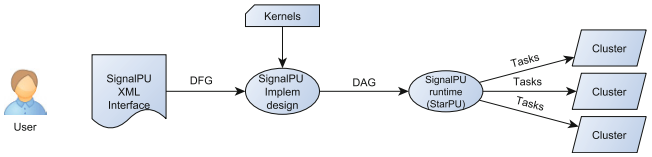

Fig. 3. SignalPU design: Three levels of processing

Bellow, we present our proposed MoP in form of 3 levels of processing as shown in the figure (Fig.3). First, the user can easily express his application, by using an XML interface, in the form of DFG. Thus, he is saved to manipulate the StarPU's API for creating tasks, for managing the buffers between each couple of tasks, or for submitting jobs onto the corresponding computation unit. Second, the implementation of application is designed by using some functionalities like: graphs unfolding techniques [12], pipelining of tasks, buffers re-use, initialization saving ... etc. The aim is to produce a DAG of independent tasks. Also here, the user doesn't have to deal with the API to unroll the main loop or to manage necessary memory buffers for that.Finally, in the third level, the StarPU runtime is used to physically manage the set of tasks and execute it on the cluster according to a dynamic scheduling to balance the load and favour the locality.

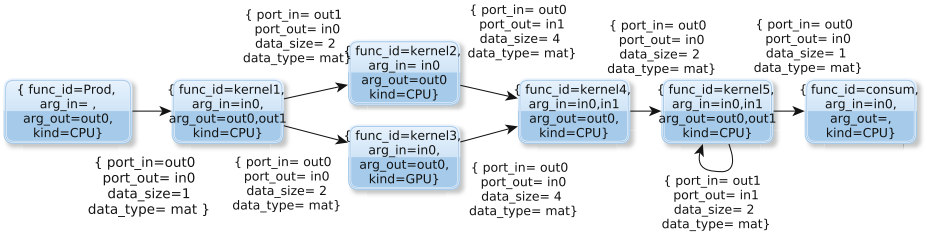

Fig. 4. The DFG-XML of the synthetic DSP application

Next, we describe all these steps of our proposed MoP with more details through a synthetic example of DSP application:

4.1 Level 1: SignalPU DFG-XML Interface

In this step, programmer has to express his application using the DFG-XML interface. First, he has to describe each kernel (operator) of his algorithm in the form of a node (vertex) using an XML structure. He has to put the name of functions which will be called in the code, the number of input and output arguments of these functions, and the architecture kind corresponding to each of them (CPU, GPU, Cell, Xeon Phi ...). Second, he has to describe in the same manner all data flows in the form of graph edges with a structure including information about type and size of data which is traded between kernels. After that, a DFG of application is produced as exemplified in the figure Fig.4, which represents DFG-XML modelling of Algorithm 1.

4.2 Level 2: SignalPU Implementation Design

In this step, illustrated in the figure Fig.5, a DAG of tasks is iteratively produced from the result of the previous processing level (the DFG-XML interface) using some functionalities adapted to DSP applications. This DAG represents a set of independent tasks linked by several kinds of data dependencies (Fork-join, producer-consumer, inter-graph producer-consumer), witch are ready to be concurrently executed on the cluster. The aim of this step is to design the execution in order to optimally take advantage of all levels of parallelism (task, data, graph parallelism) by overcoming overheads due to the execution management, like memory management (Allocation, affectation and free of buffers), data management (Copy-in, copy-out), tasks management (Creation, dependencies management, scheduling, status updating, destruction) ... etc. Next, we describe used functionalities to do that:

First, from the DFG-XML model of application, we create a set of "codlet" which is a StarPU structure and represents a mould of tasks. So, for each node in the DFG we match a "codlet" which contains all informations about the corresponding kernel (Number of input arguments, the number of output arguments,

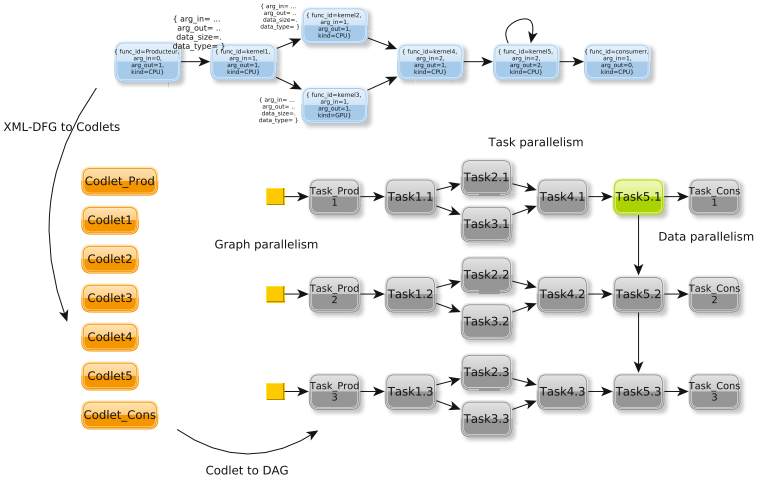

Fig. 5. The creation process of DAG of tasks of synthetic DSP application

function identifiers, architecture kinds . . .), which will characterise his tasks children. After that, based on unfolding techniques [12] which allow to unroll the main loop of the application to unravel hidden concurrency, we iteratively create tasks corresponding to all codlets, and progressively we connect them according to the dependency informations contained in the DFG-XML structure (Data type, data size, input argument, output argument, input node, output node). Also, during that process, we affect in turn to tasks the corresponding buffers according to a "First available-First affected" rule. So a buffer is affected to a new task, if and only if it's not still used by the old "sister" task. So, by using that technique named "Buffers re-use", we reduce the overhead due to memory management by the allocation and the freeing. In addition, in order to reduce overhead due to tasks management, we limit the number of submitted tasks by using pipelining functionality. So, at runtime, only a fixed levels of task is managed (Dependencies, scheduling, task status updating, . . . etc), where each pipeline level corresponds to a graph level in the DAG. Finally, we use a functionality which we call "Initialisation saving" to preserve the initialisation part of each task. So, each task leaves his initialisation data to his sister task on each device. Thus, the production or the copy of that initialisation data is made only one time per kernel per device.

4.3 Level 3: SignalPU Runtime (StarPU)

In this step, the submitted DAG of tasks generated in the previous level is physically processed by StarPU runtime. So, he manages tasks by creating the

submitted one, he updates their status according to data dependencies synchronization, he schedules them thanks to different algorithms like the work stealing (ws) or the heterogeneous earliest finish time (heft) [1] in order to balance the load and to highlight the locality, and finally, he executes them on the corresponding devices. The figure Fig.6 illustrates that process made by the StarPU runtime.

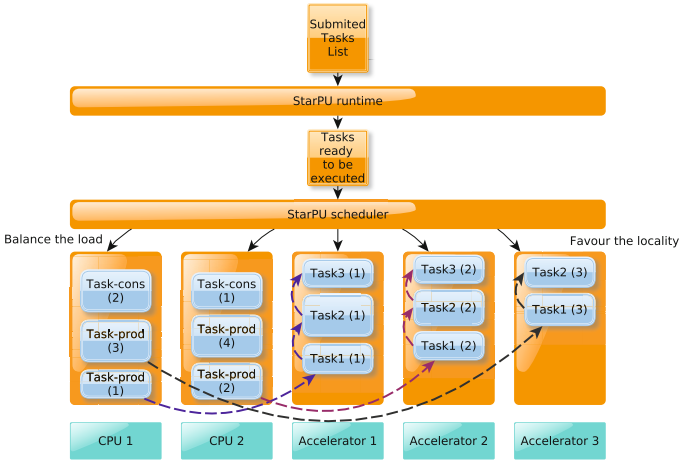

Fig. 6. StarPU runtime's levels

Thus, according to these three steps of our proposed MoP, the programmer can easily implement his DSP applications in high abstraction level, and efficiently take advantage of the optimizations: Task parallelism (TP) by extracting tasks in the DFG which are 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 optimally scheduling tanks to load balance versus data locality using StarPU runtime.

5 Validation

In this section we present a real world experimentation 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.

5.1 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 [7]. His algorithm (Algorithm 2) 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 2. Static pathway of visual model

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

5.2 The SignalPU Implementation

To implement the application with our programming model, the first step is to model its algorithm (Algorithm 2) 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.7) we present the DFG result of this step.

At runtime, the DFG-XML description of the saliency application is analyzed and 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

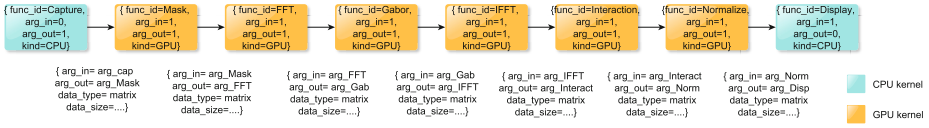

Fig. 7. The DFG-XML model of the visual saliency application

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.

5.3 The Results

In this subsection, we present the results of experimentations where we show the performance provided by the implementation based on our programming model using the proposed optimizations (Graph unfolding, Dynamic scheduling, Buffers reuse, Tasks pipelining, Initialization saving). The aim is to highlight the performance gain by using these optimizations to take advantage of parallelism and heterogeneity of clusters. The architecture used for the experimentations is a CPU-GPU node composed of a 4 cores CPU (intel-i7 core) and 3 GPU (NVIDIA Quadro 400, NVIDIA Quadro 400, NVIDIA GeForce GTX TITAN).

In the figure Fig.8, we present the total time necessary to process 1000 images (512x512 pixels) of 3 executions using different processing units. This total time is composed of the effective execution time on the processing units, plus the sleeping time which represents the time went without doing anything on the device, plus the overhead time which represents additional time consumed by managing the work (Tasks management, Scheduling, Buffer management, ...). For the first execution, we use 1 CPU core and 1 GPU (Quadro) to process images. In the GPU bar chart, we can show that execution time is higher compared to overhead time and sleeping time thanks to graph unfolding optimization which reduce waiting time. In the CPU bar chart, execution time is lower because the application is more GPU need. In the second experimentation, we use 1 CPU core and 2 GPU (Quadro). Here, we can show that total time necessary decreases compared to the first execution, so we note a speedup equal to 1.7 x. Also, thanks to dynamic scheduling, we can see the result of load balance between the 2 GPUs processing unit, thus the execution time is the same in the both GPU bar chart. In the third execution, we use 1 CPU core and 3 GPUs to enhance performance. And, we obtain a speedup equal to 2.2x compared to the first execution. Also here, we can note the advantage of using dynamic scheduling to reduce execution time and balance the load, and the graph unfolding to reduce sleeping time. But also the advantage of using the buffers reuse technique, the pipelining of tasks and the initialization data saving to stabilize overhead time.

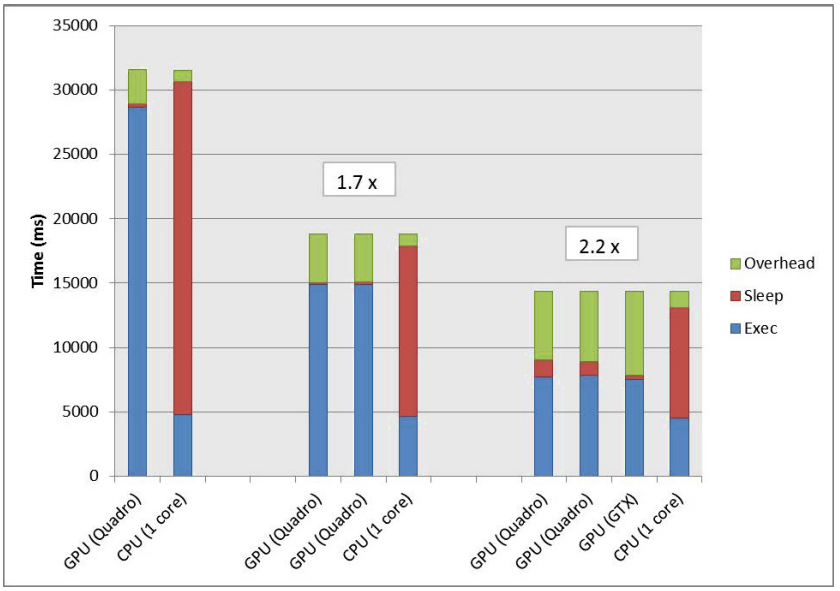

Fig. 8. Processing time of 1000 images using different processing units

6 Conclusion

In this paper we presented our proposed programming model used to implement DSP applications, allowing a high level abstraction from the hardware specificities thanks to its visual data-flow programming capabilities, and in the same time, producing a good performance of application's implementation through exploiting task parallelism, data parallelism, graph parallelism (graph unfolding), and dynamic scheduling. First, we described DSP applications and specified their characteristics in order to implement them in an optimal way. Then, we proposed an XML interface to easily describe DSP applications in the form of a DFG model. In addition, we proposed an execution model based on StarPU runtime and exploiting some techniques. We used unfolding techniques to construct DAG of independent tasks, which we submit in pipeline mode and configured to reuse a static buffers and to save the initializations data on devices in order to reduce overhead time, after that we dynamically schedule and process them on heterogeneous and parallel architecture. 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 efficiently take advantage of architecture's power to speed up the execution.

References

1. Augonnet, C., Thibault, S., Namyst, R.: StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Research Report RR-7240. INRIA (2010)

2. Bhattacharya, B., Bhattacharyya, S.: Parameterized dataflow modeling for dsp systems. Trans. Sig. Proc. 49(10), 2408–2421 (2001), http://dx.doi.org/10.1109/78.950795
3. Bueno, J., Martinell, L., Duran, A., Farreras, M., Martorell, X., Badia, R.M., Ayguade, E., Labarta, J.: Productive cluster programming with ompss. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 555–566. Springer, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2033345.2033405
4. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers Inc., San Francisco (2001)
5. Chen, W.Y.: Optimizing Partitioned Global Address Space Programs for Cluster Architectures. Ph.D. thesis, EECS Department, University of California, Berkeley (December 2007), http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-140.html
6. Flynn, M.: Some computer organizations and their effectiveness. IEEE Transactions on Computers C-21(9), 948–960 (1972)
7. Itti, L., Koch, C., Niebur, E.: A model of saliency-based visual attention for rapid scene analysis. IEEE Trans. Pattern Anal. Mach. Intell. 20(11), 1254–1259 (1998), http://dx.doi.org/10.1109/34.730558
8. Kirk, D.B., Hwu, W.M.W.: Programming Massively Parallel Processors: A Hands-on Approach, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2010)
9. Lee, E., Messerschmitt, D.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Transactions on Computers C-36(1), 24–35 (1987)
10. Munshi, A., Gaster, B., Mattson, T., Ginsburg, D.: OpenCL Programming Guide. OpenGL, Pearson Education (2011), http://books.google.fr/books?id=M-Sve_KItQwC
11. Pacheco, P.S.: Parallel Programming with MPI. Morgan Kaufmann Publishers Inc., San Francisco (1996)
12. Parhi, K., Messerschmitt, D.: Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. IEEE Transactions on Computers 40(2), 178–195 (1991)
13. Reinders, J.: Intel threading building blocks - outfitting C++ for multi-core processor parallelism. O'Reilly (2007)
14. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming, 1st edn. Addison-Wesley Professional (2010)