Parallel Pipeline on Heterogeneous Multi-Processing Architectures

Andrés Rodríguez*, Angeles Navarro*, Rafael Asenjo*, Antonio Vilches*, Francisco Corbera*, María Garzarán†

* Universidad de Málaga, Andalucía Tech, Spain. e-mail: {andres, angeles, asenjo, avilches, corbera}@ac.uma.es
† Dept. Computer Science, UIUC, USA. e-mail: garzaran@illinois.edu

***Abstract*—We address the problem of providing support for executing single streaming applications implemented as a pipeline of stages that run on heterogeneous chips comprised of several cores and one on-chip GPU. In this paper, we present an API that allows the user to specify the type of parallelism exploited by each pipeline stage running on the CPU multicore, the mapping of the pipeline stages to the devices (GPU or CPU), and the number of active threads. Using as case of study a real streaming application, we evaluate how these parameters affect the performance and energy efficiency of a heterogeneous on-chip processor (Exynos 5 Octa) that has three different computational cores: a GPU, an A15 quad-core and an A7 quad-core. We also explore some memory optimizations and find that while their performance impact depends on the granularity type, they usually reduce energy consumption.**

Keywords*—*Pipeline pattern, Heterogeneous chips, On-chip GPU, Performance-Energy efficiency.

I. INTRODUCTION

In this paper, we focus on the problem of providing support for executing single streaming applications implemented as a pipeline of stages that run on heterogeneous chips comprised of several cores and one on-chip GPU. Streaming applications are very common in today's computing systems, in particular mobile devices [1] where heterogeneous chips are the dominant platforms. Our approach extends the parallel pipeline template that TBB provides [2] to allow its deployment on these heterogeneous on-chip architectures.

As study case to demonstrate the use of our template, we introduce ViVid[1], an application that implements an object (e.g., face) detection algorithm [3] using a "sliding window object detection" approach [4]. ViVid consists of 5 pipeline stages. The first and the last one are the Input and Output stages (serial), whereas the three middle stages are parallel (stateless[2]).

When applications like ViVid run on a heterogeneous on-chip architecture, many possible configurations are possible. For instance, one needs to consider the granularity or number of items that should be simultaneously processed on each stage, the device where each stage should be mapped, and the number of active CPU cores that minimize the execution time, the energy consumption, or both (depending on the metric of interest). The *granularity* determines the level at which the parallelism is exploited on the CPU. In our approach, the user can specify two levels of granularity: Coarse Grain (CG) and Medium Grain (MG). If different items can be processed simultaneously on the same stage (stateless) on the CPU, then CG granularity can be exploited. On the other hand, if the stage exhibits nested parallelism (which can be exploited by using OpenCL, OpenMP or TBB `parallel_for`), then a single item can be processed in parallel by several cores on the CPU, and MG granularity can be exploited. In our approach, CG granularity implies as many items in flight as number of threads the user sets, while MG granularity entails two items in flight at most (one on the GPU device and another one on the CPU multicore).

The *pipeline mapping* determines the device where the different stages of the pipeline can execute. Fig. 1 graphically depicts the possible mappings for the three middle parallel stages of ViVid. Let's assume that a pipeline consists of S_1, S_2, ... S_n parallel stages. We use a n-tuple to specify all possible stage mappings to the GPU and the CPU devices: $\{m_1, m_2, \ldots, m_n\}$. The i-th element of the tuple, m_i, specifies if stage S_i can be mapped to the GPU and CPU, ($m_i = 1$), or if it can only be mapped to the CPU, ($m_i = 0$). If $m_i = 1$, the item that enters stage S_i checks if the GPU is idle. If the GPU is idle, it executes on the GPU; otherwise, it executes on the CPU. For instance, the ViVid examples of Fig. 1 correspond to the tuples (row major order): {1,1,1}, {1,0,0}, {0,1,0}, {0,0,1}, {1,1,0}, {1,0,1}, {0,1,1}, {0,0,0}. In our implementation, mapping {1,1,1} represents a special case: if the GPU is available when a new item enters the pipeline, then all the stages are mapped to the GPU.

In this work, we first introduce our framework API to specify the pipeline application and the parameters of interest to the user: i) the granularity at which the parallelism of each stage can be exploited (Coarse or Medium grain), ii) the mapping of the pipeline stages to the different devices; and iii) the number of threads to control the number of active computational cores (Section II). We then present results when executing ViVid on a heterogeneous on-chip architecture, analyzing how the granularity, the mapping, and the number of active threads affect the performance and energy consumption of our study case (Section III). Finally, we present some related works (Section IV) and conclusions (Section V).

II. INTERFACE FOR THE PIPELINE TEMPLATE

In this section we introduce the API of our pipeline library. It provides a programming environment for C++ that facilitates the configuration of a pipeline, hiding the underlying TBB implementation and automatically managing the necessary memory data management and transfers between devices.

Fig. 2 shows all the components involved in the pipeline template and the software stack. The `Item` is the object that traverses the pipeline. It contains the references to the data buffers that the different processing stages of the pipeline

[1]http://www.github.com/mertdikmen/vivid

[2]A stage is parallel or stateless when the computation of an item on a stage does not depend on other items.

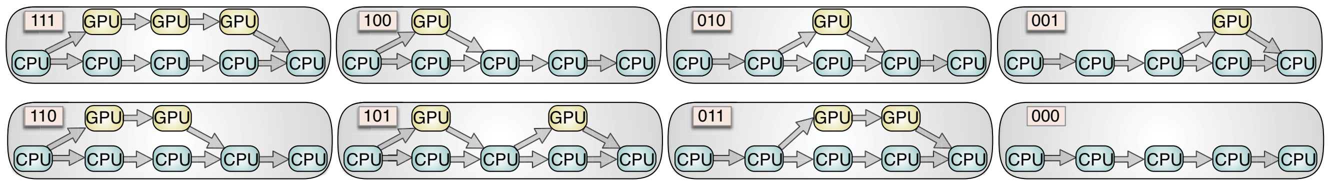


Fig. 1: All possible mappings of stages to GPU and/or CPU for ViVid.

use as input and output. To create a new pipeline instance, the user has to declare a new `Item` class (derived from a provided `Item` base class) that contains the references to data buffers used by the pipeline stages. For data buffer management, the template provides a data buffer class that hides all the important operations like allocation, deallocation, data movements, zero-copy buffer mappings, etc. The aim of this data buffer class is to offer an abstraction layer that manages and makes data accessible to the device (CPU or GPU) where the item has to be processed.

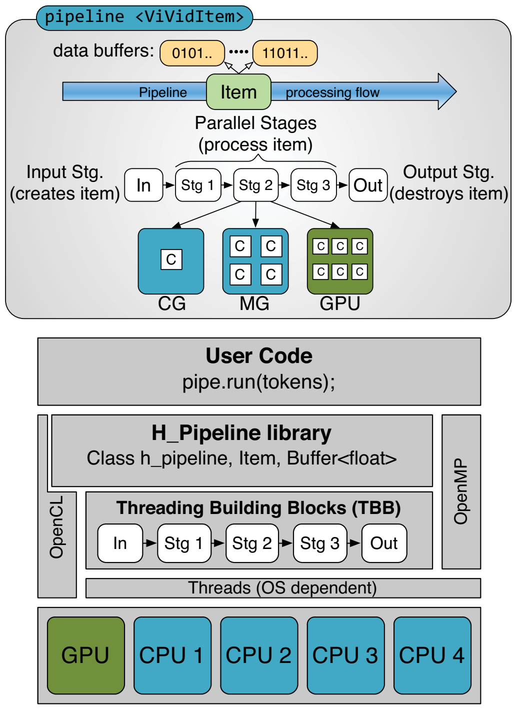


Fig. 2: ViVid application and software stack.

The programmer can provide up to three different functions for every processing stage of the pipeline: one to implement the processing stage on the GPU device using OpenCL, a second one to implement the stage on a single CPU core (CG granularity), and the third one to implement the stage on multiple CPU cores (MG granularity). The implementations not provided will not be considered when searching for the best pipeline configuration.

The interface has four main components:

- Items: objects that traverse the pipeline carrying the data buffers.
- Pipelines: the pipeline itself composed of n stages. The pipelines can be configured statically by the user or run in an auto-configuration mode. This auto-configuration model uses an analytical model [5] to dynamically compute and use the best configuration.
- Stage functions: each processing stage needs to be programmed to run on CPU and/or GPU. The pipeline uses the appropriate function for each stage.
- Buffers: n-dimensional arrays that can be used both in the host code and in the OpenCL kernels.

```
1  #include "h_pipeline.h"
2
3  class ViVidItem : h_pipeline::Item {
4    //Buffer Definitions
5    ...
6    ViVidItem(){...}   //Buffer allocation
7    ~ViVidItem(){...}  //Buffer deallocation
8  }
9
10 int main(int argc, char* argv[]){
11   int nTh = 4 ; //number of threads
12
13   h_pipeline::pipeline<ViVidItem> pipe(nTh);
14
15   //Set CG, MG and GPU functions for each stg.
16   pipe.add_stage(cg_f1, mg_f1, gpu_f1);
17   pipe.add_stage(cg_f2, mg_f2, gpu_f2);
18   pipe.add_stage(cg_f3, mg_f3, gpu_f3);
19
20   //Pipeline configuration: '101' and MG
21   pipe.set_conf({1,0,1},true);
22   pipe.run(numTokens);
23
24   //Dynamic dispatch of the pipeline
25   //pipe.run(numTokens, ENERGY, maxOverhead);
26 }
```

Fig. 3: Using the heterogeneous `pipeline` template

Due to space constraints, we will only show here some details of the pipeline specification and usage. The interested reader can find more details in [5].

Fig. 3 shows a pipeline definition and usage example. First, our pipeline interface is made available by including the `h_pipeline.h` header file (line 1) and is encapsulated inside the `h_pipeline` namespace. The programmer must define a class (`ViVidItem` in line 3) that must extend `h_pipeline::Item` and declare as many buffers as needed for the pipeline execution. The class' constructor and destructor methods hold the buffers creation or acquire, and deletion or release, respectively. The acquire and release methods can be used when a pool of buffers is created once and reused by different items (this optimization is discussed and evaluated in Section III-D). Line 13 shows the declaration of a new pipeline object using the ad-hoc previously defined `ViVidItem` class. The argument for the `pipe` constructor is the number of threads that will run the pipeline in parallel. The Input and Output stages automatically call the constructor and destructor of the `Item` class, respectively. Before using the pipeline, the programmer must specify the CPU (CG and/or MG granularity) and/or GPU functions to compute at the parallel stages (there are three in the ViVid example: lines 16 to 18). The pipeline will use the appropriate version of the function to run the stage on one CPU core (`cg_f`), several CPU cores (`mg_f`), or the GPU device (`gpu_f`). Each function internally receives as argument a pointer to the item to be processed. From such item, it is possible to obtain the pointers to the input/output data, that are automatically placed on the CPU or GPU memory space, depending on where the item is being processed.

The overloaded `run()` method can be used to run the pipeline with a static configuration, that can be previously set with the `pipe.set_conf()` method (lines 21-22). The latter method has to specify the stages that potentially could be mapped to the GPU and the granularity (MG or CG) that should be used in the CPU multicore. In our example, {`1,0,1`} represents the n-tuple defined in the previous section and the last argument in line 21, `true`, indicates that MG granularity will be exploited when an item is processed on the CPU. Once the pipeline is configured the user can run it (line 22) by setting the maximum number of items in flight allowed to simultaneously traverse the pipeline (`numTokens`). We use this approach in the experimental section. Alternatively, line 25 shows how to use the same `run()` method to dynamically find the best configuration relying on the analytical model described in [5]. In this case, the user has to select the optimization criterion (`THROUGHPUT, ENERGY, THROUGHPUT_ENERGY`) and the maximum overhead allowed (as a ratio between [0,1]) due to the dynamic search of the best configuration.

A. Pipeline Stages

One of the key components of the `pipeline<T>` class is the `parallel_stage` class. One object of this class is allocated for each `add_stage()` invocation (see Fig. 3 lines 16 to 18). This class holds important instance variables: three of them are function pointers which point to the CG (`cgFunc`), MG (`mgFunc`) and/or GPU (`gpuFunc`) functions. There are two other instance variables, `runOnGPU` and `grain`, that are used to decide whether the stage should execute on CPU or GPU (at runtime) and in the former case, if the MG or CG version should be used to execute the stage on the CPU. The `runOnGPU` variable is initialized according to the mapping tuple that can be specified using the `set_conf()` method (see line 21 in Fig. 3). An `operator()` function is automatically invoked when an item reaches the stage. This functor first receives a pointer to the item that needs to be processed so it can be passed down to the appropriate function. Then it is decided which function has to be called: if `runOnGPU` is true and the GPU is idle, the item is processed on the GPU (i.e. `gpuFunc` is called). Otherwise, depending on the `grain`, `mgFunc` or `cgFunc` will be invoked.

III. Experimental results

A. Experimental Settings

We run our experiments on an Odroid XU3 bare-board. The Odroid has a Samsung Exynos 5 Octa (5422) featuring a Cortex-A15 2.0 Ghz quad-core and a Cortex-A7 quad-core CPUs (ARM's big.LITTLE architecture). The Cortex-A15 quad-core contains 32 KB (Instruction)/32 KB (Data) private caches and a 2 MB L2 shared cache, while the Cortex-A7 quad-core has 32 KB (Instruction)/32 KB (Data) private caches and a 512 KB L2 shared cache. This platform features current/power monitors based on TI INA231 chips that enable readouts of instant power consumed on the A15 cores, A7 cores, main memory and GPU. The Exynos 5 includes the GPU Mali-T628 MP6. We have chosen this heterogeneous on-chip processor because it presents one interesting feature that we consider in our study. In particular, it allows two levels of heterogeneity: two devices with different ISA: a GPU and a CPU multicore, plus two different processor technologies inside the CPU multicore. Thus, in this system we have three type of cores with different computational capabilities sharing the main memory.

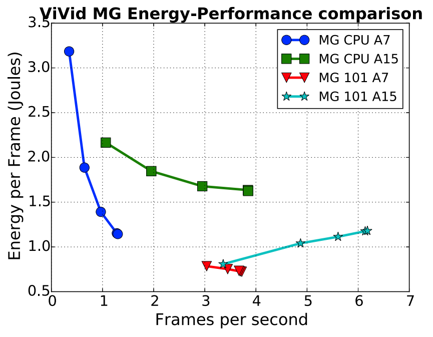
The board runs Linux Ubuntu 14.04.1 LTS and an in-house library measures energy consumption [6]. We use gcc 4.8 with -O3 flag and the OpenCL SDK v1.1.0 for the Mali GPU. The CPU codes are vectorized using NEON intrinsics. For the MG granularity versions, the `omp parallel for` pragma was used on each stage. Intel TBB 4.2 provides the core template of the `pipeline` pattern. For each experiment, we measured time and energy in 5 runs of our application and report the median value. The input data consists of 100 frames (SD resolution: 600 x 416).

In the next section, we explore how the level of granularity of the stages (MG and CG), the mapping of stages to the different devices, and the number of threads affects the energy-performance efficiency. Then, we explore the impact of several memory optimizations.

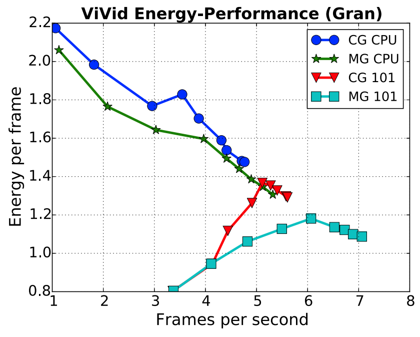
B. Study of the Granularity

We first study the performance and energy efficiency of homogenous executions (only one type of CPU cores) vs. heterogeneous executions (in which we incorporate the GPU) when running the previously mentioned application, ViVid. For the heterogeneous executions we have explored all the pipeline mappings and both granularities, Coarse-Grain (CG) and Medium-Grain (MG). We have found that, for ViVid, the pipeline mapping that achieves the highest frame rate (pipeline throughput) is {1,0,1} for both granularities. This means that stages 1 and 3 can potentially be mapped to the GPU and CPU, but stage 2 is only mapped to the CPU. This is not surprising because stage 2 performs poorly in this GPU. Thus, the next figures show ViVid results for this mapping, labelled as `101`. For reference, the figures also show the experiments in which only the CPU cores are used, labelled as `CPU`.

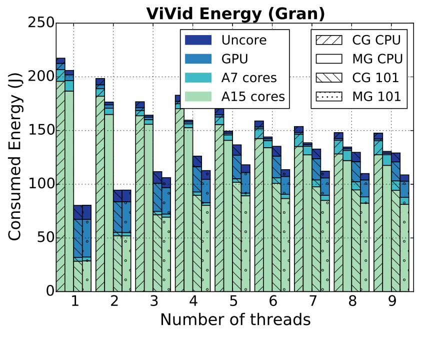
Fig. 4(a) shows how the different devices behave under MG granularity for the homogeneous (only CPU cores) and the heterogeneous (GPU+CPU cores) mappings in ViVid. For each experiment, Fig. 4(a) shows the energy per frame (Joules) vs. the frame rate (i.e. throughput measured in frames per sec. or fps). In other words, they depict energy efficiency vs. performance efficiency. Each line exhibits 5 points: each one representing energy per frame vs. fps when the experiment runs from 1 thread (the leftmost point) to 5 threads (the last point in the line). For the A7 (A15) experiments, the threads are always



(a)



(b)



(c)

Fig. 4: (a) Study of the impact of MG granularity on ViVid for A7 vs A15 vs GPU+A7 vs GPU+A15; (b) and (c) Study of the impact of the CG and MG granularity on ViVid for A15+A7 vs GPU+A15+A7

confined on the A7 (A15) multicore using the `taskset` command. In the homogeneous experiments (`MG CPU A7`, and `MG CPU A15`) the first 4 points represent the contribution due to 1 to 4 threads running on the A7 or on the A15 multicores. The last point represents a situation of oversubscription on each multicore. When the GPU is incorporated, i.e. the `MG 101 A7` and `MG 101 A15` experiments, the first point represents the contribution due to only the GPU (1 thread) and the next 4 points (from 2 to 5 threads) the contribution due to the A7 or A15 multicores.

Fig. 4(a) shows that when increasing the number of threads, and therefore the number of active cores, the performance increases. In the case of homogeneous experiments, the energy efficiency improves also as the number of threads increases in both granularities. For these experiments, clearly the CPU A7 executions are more energy efficient (when number of threads is higher than 1) while the CPU A15 ones are more performance efficient. The figure also shows that when incorporating the GPU (`MG 101 A7` and `MG 101 A15`) the energy consumed decreases. In particular, using only the GPU (heterogeneous runs with 1 thread) we observe 4.7x and 3.1x more energy efficiency with respect to using just one A7 core (first point of MG CPU A7 curve) and one A15 core (first point of MG CPU A15 curve), respectively. The minimum energy consumption is achieved in the heterogeneous 101 A7 execution with 5 threads, which is 1.5x more energy efficient than the 5 threads `MG CPU A7` homogenous one. Interestingly, while the heterogeneous `MG 101 A7` executions tend to slightly increase the performance while marginally decreasing the energy consumption when incorporating more threads, the `MG 101 A15` executions increase the performance but at the cost of increasing the energy consumed. One other interesting observation is that similar performance is achieved for `MG CPU A15` with 4 threads and `MG 101 A7` with 5 threads (i.e. A15 quad-core vs. GPU+A7 quad-core), but the latter is 2.7x more energy efficient than the former.

We notice a similar trend for the CG heterogeneous executions. In any case, given a number of threads, MG heterogeneous executions provide higher performance and consume less energy than the CG equivalent ones, both for the A7 and the A15 experiments.

Next, we explore the impact of incorporating all the different devices (A15, A7 and GPU) under CG and MG granularities. Since taskset command is not used in the following experiments, the default scheduler first occupy the A15 cores, and when all A15 are already busy, A7 cores are also exploited. Figs. 4(b) and (c) shows the study for CG and MG granularities when we incorporate to the A15 executions the A7 cores. The study also includes the only CPU experiments (`CG CPU` and `MG CPU`) vs. the heterogeneous mappings with the GPU (`CG 101` and `MG 101`). Our goal now is to understand how the performance and energy efficiency get affected by incorporating cores of different characteristics.

Fig. 4(b) represents the energy per frame vs. the frame rate for each experiment. Each line shows the results from 1 thread (the leftmost point) to 9 threads (the last point in the line). In the only CPU experiments, the first 4 points (from 1 to 4 threads) represent the contribution due to the A15 cores, the next 4 points (from 5 to 8 threads) the contribution due to the A7 cores, and the last point represents the situation of oversubscription. When the GPU is incorporated, i.e. the `MG 101` and `CG 101` experiments, the first point represents the contribution due to only the GPU (1 thread), the next 4 points (from 2 to 5 threads) the contribution due to the A15 cores, and the last 4 points (from 6 to 9 threads) the contribution due to the A7 cores.

Fig. 4(b) shows that for the only CPU experiments, the energy efficiency and the frame rate improve as the number of threads increases. It also shows that incorporating the GPU (`CG 101` and `MG 101`) reduces the energy consumption. The minimum energy consumption is achieved with 1 thread (just the GPU), which is 2.7x and 2.6x more energy efficient than the 1 thread `CPU CG` and `CPU MG` granularities, respectively. However, both `CG 101` and `MG 101` experiments tend to increase the energy consumption when adding more threads (specially when the A15 cores are incorporated, as shown for the 2nd to the 5th point in both lines). Both CG and MG granularities improve the frame rate as the number of threads increases, but CG consumes more energy and achieves lower performance than MG. Interestingly, when we add the A7 cores in the `CG 101` and `MG 101` experiments (threads 6 to 9, or 6th to 9th point), both granularities improve the performance while slightly reducing the energy consumption. In any case, for this application, `MG 101` delivers the highest frame rate (for 9 threads) with just a degradation of 32% of the energy consumption when compared to the minimum consumption (1 thread, just the GPU).

Fig. 4(c) shows the breakdown of the energy consumption as we increase the number of threads: it shows the contribution due to the A15 and A7 cores, the GPU, and the Uncore units of the chip. The first bar represents the only `CPU CG` experiment, the second one the only `CPU MG` experiment, while the third and fourth the `CG 101` and `MG 101` experiments, respectively. The figure shows that, for the CPU-only experiments, the A15 energy component reduces significantly (while slightly increasing the A7 component) as the number of threads increases. Also, for any number of threads, the A15 energy component is always smaller in `CPU MG` than in `CPU CG`: this is due to a better locality exploitation under this granularity, thanks to the L1/L2 caches. On the other hand, for the `CG 101` and `MG 101` experiments, the A15 energy component increases from 1 to 5 threads. However, it reduces the GPU energy component because the cores are now taking some of the iterations that were previously processed on the GPU. The overall result is an increment in the total energy consumption because the A15 are less energy efficient than the GPU. Adding more threads in these experiments means that we incorporate the A7 cores, which slightly reduces or stabilizes the A15 energy component, while marginally increasing the A7 component. The figure also shows that the GPU and Uncore energy components are correlated: both decrease from 1 to 5 threads (A15 cores), and from there they estabilize. The `MG 101` experiment shows smaller consumption in the A15 energy component, due to a better cache exploitation under this granularity, as commented before. In any case, for 9 threads, `MG 101` is 16.2% more energy efficient than `CG 101`.

C. Study of the Mapping

Since we have identified that the best granularity for this code is MG, we study the impact that mapping has for this granularity. We compare the case that obtains the best throughput (the mapping 101) with the case in which we optimize communications among the GPU and the CPU (mapping $\{1,1,1\}$ where all the stages can be mapped to the GPU if the item entering the pipeline finds the GPU available). Fig. 5(a) shows a comparison between `MG 101` and `MG 111`. Again, for reference, we show the results when only the CPU is exploited (`CPU MG`).

In this case, both mappings have a similar behavior: adding the A15 cores (from 2 to 5 threads) increases the throughput at the cost of increasing the energy consumption (due to the A15 energy component. Adding the A7 cores improves the frame rate but reduces (slightly) the energy consumption. Anyway, for 9 threads the `MG 101` mapping gives a slightly higher frame rate (not apparent in the figure) and lower energy consumption than `MG 111`. The differences between both mappings are small because stage 2 (the stage with a different mapping on each experiment) has a small computational load: it takes less than 5% of the computation of an item in the pipeline.

D. Memory optimizations

As the Odroid does not have a Last Level Cache shared among the CPU cores and the GPU, memory operations compete for the memory bus (AHB) when both devices work concurrently. In this section, we explore a memory optimization that tries to minimize the amount of buffer copying in the shared physical memory system to reduce the traffic on the memory bus. We also explore other optimizations to reduce the amount of memory management operations, which reduces system overheads. We study the impact of the optimizations on the CG and MG granularities, for the mappings in which each one achieves the highest frame rate: `CG 101` and `MG 101`.

Figs. 5(b) and (c) compares the speedup and the energy consumption between `CG 101` and `MG 101` (the baselines) and the following optimization alternatives (which can be also activated in our framework using a knob in the buffers initialization [5]):

- `CG 101 ZCB` and `MG 101 ZCB`: Uses the Zero-Copy Buffer feature available in OpenCL to allocate the data that are shared between the GPU and the CPU in the same buffer. This optimization reduces the memory footprint and the amount of buffer copying in the shared physical memory.
- `CG 101 BUF` and `MG 101 BUF`: Uses a pool of buffers that are created (and destroyed) once before starting (exiting) the pipeline run. These buffers are used by the pipeline template to pass the data between the pipeline stages. Our optimization avoids the dynamic allocation/deallocation of memory for each item that is the default policy in the TBB `pipeline` template. This optimization removes overheads due to system calls.
- `CG 101 BUF ZCB` and `MG 101 BUF ZCB`: Use the two optimizations.

Fig. 5(b) shows that applying only the ZCB feature degrades performance for both CG and MG granularities. The degradation of performance for the MG granularity, compared with the MG baseline, is significant: it degrades 34% for 9 threads. For CG granularity, the degradation with respect to the CG baseline is smaller: less than 4.2% for 9 threads. This loss of performance is due to a locality problem: we think that the ZCB functionality employs a remote direct memory access protocol that prevents the data from being cached (we have observed that the CPU-only performances also degrade when ZCB is used). This affects greatly the MG granularity, that loses the locality advantage over the CG granularity.

However, the BUF optimization has a positive impact in the CG granularity: it improves the performance of the CG baseline by 28.5%. In the MG case, this optimization brings up a load imbalance problem between the A15 and A7 cores with 6 and 7 threads. Although it starts to recover performance for 8 and 9 threads, this is not enough to compensate the loss due to the imbalance. For instance, with 9 threads, the performance loss of BUF over the MG baseline is 6.2%.

Interestingly, the BUF optimization combined with the ZCB feature is the one that achieves the highest performance in CG: for 9 threads, it improves speedup by 32.9% when compared to the baseline. For MG, the saving of system call overheads due to BUF does not compensate for the loss of locality due to ZCB: as a result, the speedup of BUF+ZCB for 9 threads degrades 7.9% when compared to the baseline.

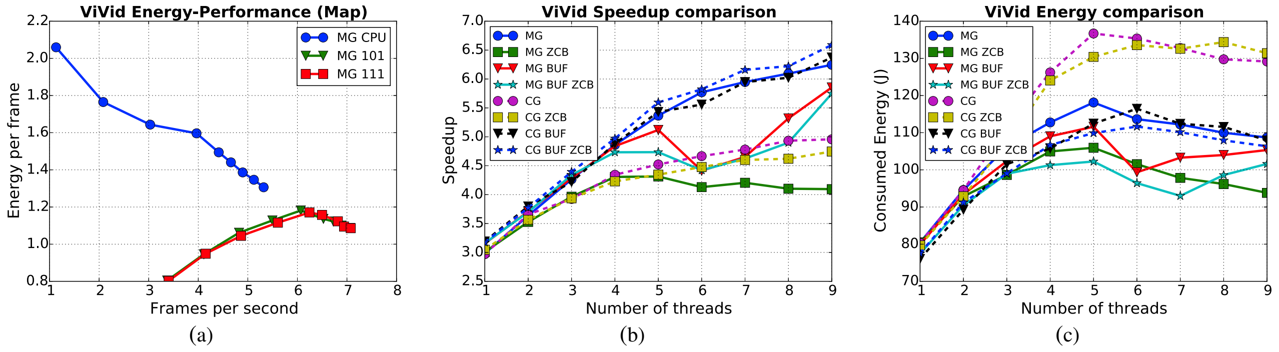


Fig. 5: (a) Study of the impact of the mapping; (b) and (c) Study of the impact of different memory optimizations on ViVid (101 mapping). Baseline for speedup is execution time for one A15 CPU core

Fig. 5(c) shows the impact of the optimizations over the energy consumption. All the optimizations, except ZCB in CG, reduce energy consumption. For instance, for the MG granularity and 9 threads ZCB, BUF and BUF+ZCF reduce energy by 13.7%, 3.1% and 6.5% when compared to the MG baseline. In particular, the ZCB optimization greatly reduces the A15 energy component and increases the GPU energy component when compared to the baseline (it trades A15 for the more energetically efficient GPU, what explains the improvement in spite of the performance degradation discussed before). For the CG granularity and 9 threads ZCB slightly increases energy consumption by 1.7%, while BUF and BUF+ZCF reduce energy by 16% and 17.6%, when compared to the CG baseline.

IV. Related work

One approach for coding streaming applications is to use a programming language with support for streams, such as StreamIt [7]. However, these languages do not provide support for heterogenous CPU-GPU executions. Concurrent execution on the CPU and GPU of the heterogenous platforms delivers higher performance than CPU-only or GPU-only executions [8], but programming frameworks that provide support for computing in a heterogeneous architecture such as Qilin [9], OmpSs [10] or StarPU [11] just consider performance when deciding task distribution among CPU cores and GPU accelerators. The difference between these related works and ours is that they focus on data parallel patterns, while we focus on streaming applications. We also study energy efficiency. FastFlow [12] provides a good framework for programming pipelines. Our proposal offers added values as the dynamic mapping of stages to processing devices taking into account not only performance but also energy efficiency or the data buffer self-management. We could have used FastFlow instead of TBB as the base to implement our pipeline framework.

V. Conclusions

In this work, we present a framework API to specify pipeline applications that can run on heterogeneous on-chip processors. Our template allows the configuration of three parameters: the granularity exploited by each stage in the CPU multicore (CG or MG), the mapping of the stages to the devices (GPU or CPU cores), and the number of active threads. We use this framework to study how these parameters affect performance and energy efficiency on a heterogeneous on-chip processor (Exynos 5 Octa) that has three different computational cores: a GPU, an A15 quad-core and an A7 quad-core. Our results show that although heterogeneous executions are the ones that achieve higher performance, it does so by increasing the energy consumption in the system when the A15 cores are incorporated. By incorporating the energy efficient A7 cores we can still increase the performance a little while slightly reducing the energy consumption. Our experiments also show that MG granularity tends to demand less consumption in the A15 energy component due to a better locality exploitation. We also explore some memory optimizations finding that their performance impact depends on the type of granularity, but they usually improve the energy efficiency.

Acknowledgments

This work has been supported by the following Spanish projects: TIN2010-16144 and TIN 2013-42253-P from Ministerio de Ciencia e Innovación, and P11-TIC-08144 from Junta de Andalucía.

References

[1] J. Clemons, H. Zhu, S. Savarese, and T. Austin, "Mevbench: A mobile computer vision benchmarking suite," in *IISWC*, Nov 2011, pp. 91–102.

[2] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism.* O'Reilly Media, 2007.

[3] M. Dikmen, D. Hoiem, and T. S. Huang, "A data driven method for feature transformation," in *Proc. of CVPR*, 2012, pp. 3314–3321.

[4] M. Jones and P. Viola, "Fast multi-view face detection," *Mitsubishi Electric Research Lab TR-20003-96*, vol. 3, 2003.

[5] A. Vilches, A. Navarro, R. Asenjo, F. Corbera, R. Gran, and M. Garzaran, "Mapping streaming applications on commodity multi-CPU and GPU on-chip processors," *IEEE Tran. on Parallel and Distributed Systems*, 2015, doi:10.1109/TPDS.2015.2432809.

[6] F. Corbera, A. Rodriguez, R. Asenjo, A. Navarro, A. Vilches, and M. J. Garzaran, "Reducing overheads of dynamic scheduling on heterogeneous chips," in *HIP3ES*, 2015.

[7] R. Soule, M. I. Gordon, S. Amarasinghe, R. Grimm, and M. Hirzel, "Dynamic expressivity with static optimization for streaming languages," in *Intl Conf on Distributed Event-Based Systems*, June 2013.

[8] C. Yang *et al.*, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *CLUSTER*, 2010, pp. 19–28.

[9] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO 42*, 2009, pp. 45–55.

[10] J. Planas *et al.*, "Self-adaptive OmpSs tasks in heterogeneous environments," in *Proc. of IPDPS*, 2013.

[11] C. Augonnet *et al.*, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, Feb. 2011.

[12] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, Oct. 2014.