Productive interface to map streaming applications on heterogeneous processors

Andrés Rodríguez*, Antonio Vilches*, Angeles Navarro*, Rafael Asenjo*, Francisco Corbera*,
Rubén Gran†, María Garzarán‡
* University of Malaga, Spain. E-mail: {andres, avilches, angeles, asenjo, corbera}@ac.uma.es
† University of Zaragoza, Spain. E-mail: rgran@unizar.es
‡ Dept. Computer Science, UIUC, USA. E-mail: garzaran@illinois.edu

Abstract

Current commodity processors include several cores and hardware accelerators, such as Graphical Processing Units or GPUs. These architectures pose two main challenges: i) determining how to map the application tasks to the computational device that maximizes performance and minimizes energy consumption; and ii) easing the programming interface so that the user productively exploits the heterogeneous architecture without dealing with low level details.

In this paper, we consider the problem of efficiently executing streaming applications on commodity processors composed of several cores and an on-chip GPU. Streaming applications, such as those in vision and video analytic, consist of a pipeline of stages and are good candidates to take advantage of this type of platforms. We also consider that characteristics of the input may change while the application is running. Therefore, we propose a framework that adaptively finds the optimal mapping of the pipeline stages. We describe the proposed API based on C++ templates that hide the underlying TBB implementation. The core of the framework is an analytical model coupled with information collected at runtime used to dynamically map each pipeline stage to the most efficient device, taking into consideration both performance and energy. Our experimental results show that for the evaluated applications running on two different architectures, our model always predicts the best configuration among the evaluated alternatives, and significantly reduces the amount of information that needs to be collected at runtime. This best configuration has, on the average, 20% higher throughput than the configuration recommended by a baseline state of the art approach, while the ratio throughput/energy is 43% higher. We have measured improvements in throughput and throughput/energy of up-to 82% and 204%, respectively, when the model is used to adapt to a video that changes from low to high definition.

Index Terms

Heterogeneous CPU-GPU chips, pipeline pattern, adaptive mapping, analytical model, energy aware.

I. Motivation

Recently, we have seen a significant increase in the number of commodity multicore processors that include an on-chip GPU. Current desktops, ultrabooks, smartphones, tablets, and other embedded devices are powered by heterogeneous chips that comprise 2 to 8 CPU cores along with an integrated GPU. Examples of these are Intel Ivy Bridge and Haswell architectures, AMD APU, Qualcomm Snapdragon 800 and Samsung Exynos 5 Octa, to name a few. These heterogeneous chips can deliver significant speedups and low energy consumption compared to CPU-only systems on a large range of applications. However, issues such as the development of a suitable programming framework and runtime support for these architectures are in their infancy.

Most research in frameworks aimed at scheduling tasks on heterogeneous architectures, composed of CPU's and GPUs, has focussed on optimizing execution time without considering energy consumption [1], [2], [3], [4], [5]. However, a CPU core and a GPU exhibit different performance/energy trade-offs, this is, a workload can run faster on one device but consume less energy on the other one. Thus, in order to benefit from the potential energy efficiency that the accelerators can provide in these heterogeneous chips, the runtime scheduler also needs to consider the performance/energy asymmetry when making a scheduling decision [6].

In this paper, we focus on the problem of efficiently executing single streaming applications implemented as a pipeline of stages that run on heterogeneous chips comprised of several cores and one on-chip GPU, taking into consideration both performance and energy. Streaming applications are very common in today's computing systems, in particular mobile devices [7] where heterogeneous chips are the dominant platforms.

To tackle the aforementioned problem, we study different choices such as: i) the granularity level at which the parallelism of each stage can be exploited (coarse or medium grain), ii) the mapping of the pipeline stages to the different computational devices and iii) the number of cores for which the application scales up. Our aim is to find the optimal configuration that considers all the previously mentioned factors. The metric to optimize can be throughput, energy or a tradeoff metric such as the ratio throughput/energy. We also consider that the best configuration may change over time. This can happen because the number of operations performed by each pipeline stage changes. There are several reasons why this can take place. For instance, YouTube, Skype Video [8], or tele-operated robots [9] adjust the resolution of the video stream based on the bandwidth of the network connection. Also, the computation of a pipeline stage may depend on the characteristics of the input frame. In this situation, an off-line training may not be feasible, as the best configuration will depend on the runtime input.

As a motivating example to demonstrate the benefits of adapting the configuration of a pipeline we introduce ViVid[1], an application that implements an object (e.g., face) detection algorithm [10] using a "sliding window object detection" approach [11]. ViVid consists of 5 pipeline stages from which the first and the last one are the Input and Output stages. When applications like ViVid run on a heterogeneous on-chip architecture, many possible configurations are possible. To determine the best configuration, 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 CPU cores that minimize the execution time, or the energy consumption, or both. As we will discuss in Section VI, we have found that when ViVid runs on an Intel Ivy Bridge platform (also presented in Section VI), the best configuration for videos with Low Definition (LD) is different from the best one for videos with High Definition (HD) and not adapting to a input change, can have a significant impact in both, execution time and energy. For instance, when the video resolution changes from LD to HD, not changing from the best configuration for LD to the new optimal for HD results in 0.55x of throughput loss (and 1.7x of more energy used). On the other hand, if we are using the optimal configuration for HD, an input change from HD to LD will result in a 0.76x of throughput degradation (and 1.1x of more energy used) if we do not change to the new best configuration. These results indicate that an approach that can predict the best configuration, out of all the possible ones, is desirable. This approach should have low overhead, so that it can be used when an input change is detected.

In this paper, we propose an adaptive framework that can dynamically adjust the configuration of the pipeline (granularity, mapping and number of cores). This framework is based on an analytical model that, by collecting a small number of runtime experiments (only 8 on a quad-core), can predict the optimal pipeline configuration. Our framework can be targeted at optimizing performance, or energy or a tradeoff metric that considers the ratio throughput/energy. Our analytical model can provide knobs so that the user can specify a desired throughput or power budget. For instance, if the user specifies a throughput of 33 fps for realtime video streaming, the model can determine among the possible pipeline configurations, the one that minimizes the energy consumption and satisfies the user constraint. Similarly, given a power budget, the model can determine the fastest configuration. The information collected using runtime input data are used to dynamically adapt to input changes. Since this data collection phase can add some runtime overhead, our framework provides another knob so that the user can specify a threshold to limit the maximum overhead of this phase. We have evaluated our model using a set of streaming applications from vision and video analytic domain that are representative of the algorithms [12] that can benefit from the execution on these heterogeneous chips.

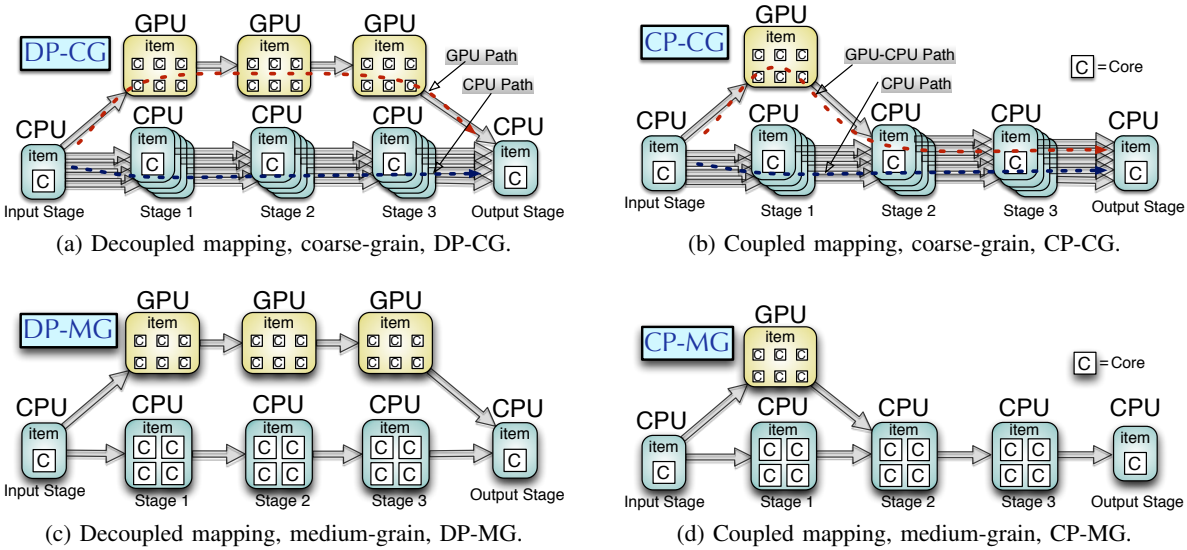
The contributions of this paper are the following ones:

- A taxonomy of the pipeline configurations for heterogeneous chips (section II).
- An adaptive framework that dynamically selects the best configuration while keeping the runtime overhead below an user-defined limit (section IV).
- An analytical model that quantifies how the different implementation factors interplay. This model can be used to predict the optimal granularity and mapping of the pipeline stages to the different computational devices, as well as the appropriate number of threads (section V).
- An evaluation of the accuracy of our analytical model. Our results demonstrate that the model accurately predicts, among the evaluated alternatives, the best pipeline configuration for all the applications and architectures studied (section VI).

II. Pipeline implementation alternatives

We use two axes to classify the different alternatives: (1) *granularity level*, that represents the level at which the parallelism is exploited on the CPU; and (2) *pipeline mapping*, that represents where the different

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



(a) Decoupled mapping, coarse-grain, DP-CG.

(b) Coupled mapping, coarse-grain, CP-CG.

(c) Decoupled mapping, medium-grain, DP-MG.

(d) Coupled mapping, medium-grain, CP-MG.

Fig. 1: Examples of the four different configurations for ViVid.

stages of the pipeline can execute. We call pipeline configuration to each possible combination of granularity and mapping. Fig. 1 graphically depicts examples of the 4 possible configurations for the ViVid pipeline on an Ivy Bridge-like architecture with a GPU (6 computing units) and a CPU multicore (4 CPU cores). The figure shows the paths that traverse the in-flight items being processed. The pipeline stages are represented as rounded rectangles, while the device (GPU or CPU) on which each stage is processed, is depicted with the number of computing resources (small squares with the letter 'C') that collaborate on the computation of each item.

Granularity level: The vertical axis in Fig. 1 classifies the approaches based on the granularity level used to exploit parallelism on the CPU. Two levels of granularity are considered: Coarse Grain (CG) and Medium Grain (MG). If different items can be processed simultaneously on the same stage (the stage is parallel or stateless[2]), 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 in the CPU, and MG granularity can be exploited. The CG granularity is shown in Figs. 1a and 1b, where, on the CPU multicore, each item is processed by a single core (one thread). MG granularity is shown in Figs. 1c and 1d. In this case, a single item is processed in the CPU multicore, with each core (or thread) processing a portion of the item.

GPUs are not as flexible as the multicores regarding the granularity level of parallelism they can exploit. They excel at exploiting SIMT (Single Instruction Multiple Threads) type of parallelism. Thus, stages mapped onto a GPU only process a single item, with all the GPU processing units computing a portion of the item (similar to MG granularity, but at a finer grain).

The MG granularity requires a barrier synchronization after each pipeline stage and before the next pipeline stage can start, to guarantee that all threads have finished processing the item. Therefore, MG can hurt performance when the load is imbalanced or there is not enough computational load per core. With MG, it is like having two devices, GPU and CPU, that can only work on two different items at a time. Thus, there is less pipeline parallelism when exploiting MG granularity. However, with the CG granularity, each CPU core (or thread) can process all the pipeline stages for a given item without intermediate synchronizations, i.e. each item traverses the pipeline at its own pace. Two drawbacks of the CG approach are that several items are in-flight at the same time, which increases the memory pressure, and that it only applies to parallel pipeline stages. Finally, notice that each CPU core can also exploit fine grain parallelism, due to the vector units of the processors. This can be exploited by both, CG or MG granularities.

Pipeline mapping: The horizontal axis in Fig. 1 classifies the configurations based on whether all the stages execute on the GPU or only a few do. The first pipeline mapping is called decoupled (DP), while the other one is called coupled (CP).

Disregarding the Input/Output stages, DP mappings are illustrated in Figs. 1a and 1c, where we depict two

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

"decoupled" paths: i) a *GPU path*, in which a thread (the GPU thread) offloads all stages to the GPU for processing one input item; and ii) a *CPU path*, in which a group of concurrent threads (the CPU threads) process all stages on the CPU. On the other hand, CP mappings are shown in Figs. 1b and 1d where we see two paths that are not independent: i) a *GPU-CPU path* in which a thread (the GPU-CPU thread) offloads some stages to the GPU (stage 1 in the figures) for processing one input item, while the remaining stages are executed on the CPU; and ii) a CPU path, in which a group of concurrent threads (the CPU threads) process all stages on the CPU multicore. The difference between the CP's GPU-CPU thread and the DP's GPU thread is the following. In a CP mapping, when an item reaches the stage for which it has been decided that it will be processed on the GPU (stage 1 for the ViVid example), we first check if the GPU is idle, and in that case the thread becomes a GPU-CPU thread that launches the item's kernel to the GPU and then waits for the GPU kernel to finish. Then, the same thread also processes the item through the remaining stages (in the example, stages 2 and 3 that are processed in the CPU). However, in DP, when an item reaches the first stage and finds the GPU is idle, the corresponding thread becomes a GPU thread that processes the item throughout all the stages on the GPU. Indeed, when we consider only 1 thread for the DP mapping, that thread becomes the GPU thread and therefore all the items traverse the GPU path. This is what we call a GPU homogeneous execution. In our example, for both CP and DP, if an item on the stage 1 finds that the GPU is already busy, then the item is directed through the CPU path. Although DP could be seen as a particular case of CP where all the stages happen to be mapped to the GPU, we distinguish both mappings because they have to be modelled differently as we will see in section V.

CP mappings can be a good alternative when not all the stages are suitable for the GPU, or because it's not advisable to divert the GPU computing power from the stages where it is faster and/or more energy-efficient. This approach also has the advantage that not all the stages have to be implemented for the CPU and GPU. However, in the CP mapping, the GPU-CPU thread must orchestrate the "coupling" of the GPU and the CPU devices and the host-to-device/device-to-host communications, which results in some data movement and synchronization overheads. Also, note that DP mappings can be implemented only if all stages are parallel pipeline stages (stateless). If, on the contrary, all stages are serial, heterogeneity can be exploited by mapping some stages on the GPU and the rest on the cores, which is a particular case of the CP mapping in which all items are directed through the GPU-CPU heterogeneous path.

A. Alternatives not considered

Some additional alternatives not considered in this classification are the following:

- Splitting an item to be simultaneously computed on the CPU and GPU. As it was demonstrated by Totoni et al. [13], this possibility is not beneficial for our vision applications due to additional synchronization overheads between both devices.
- Having one stage exploiting both MG and CG granularities on the CPU. For example, a quad-core can be splited into two CPU devices with two cores each. This approach would combine CG and MG on the same stage: two CPU devices processing two items in parallel (CG), and each item running on two cores (MG). For that, we explored the OpenCL Device Fission function (`cl_ext_device_fission`) that can divide the CPU device into several subdevices with lower core count. However, we discarded this alternative for the following two reasons. On the first hand, we have measured 14% of overhead (for ViVid on Ivy Bridge) if the `device_fission` is called to change the subdevices configuration from one pipeline stage to the next one. Thus, this approach is beneficial only if the optimum number of cores per device coincides for all the pipeline stages. On the second hand, the current OpenCL driver for Windows (there is no such a driver for Linux), relies on a busy waiting mechanism where the host thread has to wait for the kernel to finish on the corresponding subdevice. This means that for two CPU subdevices, two additional threads doing busy-wait will occupy two CPU cores, which results in further degradation of the execution time.
- Exploiting stages with both MG and CG granularities on the GPU. The OpenCL fission feature is currently not able to split the GPU on Intel or AMD heterogeneous chips. Therefore, we do not consider this feature to evaluate additional pipeline configurations.
- Hybrid mappings in which some CPU stages exploit MG and the rest CG granularity. This is left for future work.

B. Accounting for all pipeline alternatives

Let's assume we have nC CPU cores (4 in Fig. 1), and 1 GPU in an heterogeneous chip (current commercial heterogeneous chips only contain a single GPU, so we overlook configurations with two or more GPUs in

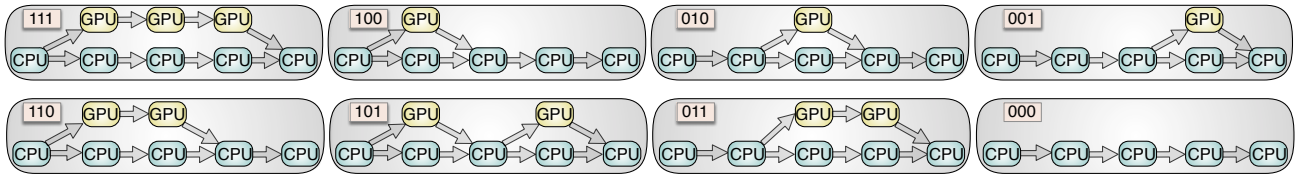


Fig. 2: All considered mappings of pipeline stages to CPU and/or GPU for ViVid.

this work). We've seen we have 4 pipeline configurations, DP-CG, DP-MG, CP-CG and CP-MG, but there are two additional factors to consider: i) for CP mappings we have to find out the stages for which the GPU is more profitable; and ii) for the CG granularity we also have to explore the optimal number of threads. For this CG granularity the number of threads in the CPU multicore can go from 0 to nC. Additionally, since the GPU thread in DP-CG, or the GPU-CPU thread in CP-CG, will be mainly hosting the GPU (waiting for the GPU task to complete), the total number of threads, n, we explore goes from 1 to $nC + 1$. This means that we allow oversubscription of one thread when $n = nC + 1$, and therefore, the GPU (or GPU-CPU) thread eventually interferes with the other nC CPU threads. For the MG granularity, we always configure $nC + 1$ threads because the constructors used to exploit nested parallelism (OpenCL or TBB `parallel_for`) by default use all the threads available in the multicore, nC, plus the GPU (or GPU-CPU) thread.

With all that, assuming that the pipeline consists of s parallel stages, there would be 2^s possible GPU/CPU mappings (this is illustrated in Fig. 2 for ViVid with $s = 3$). These mappings can be combined with $nC + 1$ different CG options, depending on the number of threads used and 1 MG option, i.e., $nC + 2$ options. Thus, in total we have $2^s \cdot (nC + 2)$. That results in 48 alternatives for ViVid with $s = 3$ and $nC = 4$.

Our goal is to be able to predict the optimal pipeline configuration specifying the granularity, mapping (identifying the stages that should be mapped on the GPU), and the optimum number of threads for a given stream input. But first, the general framework is presented.

III. Programming interface

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

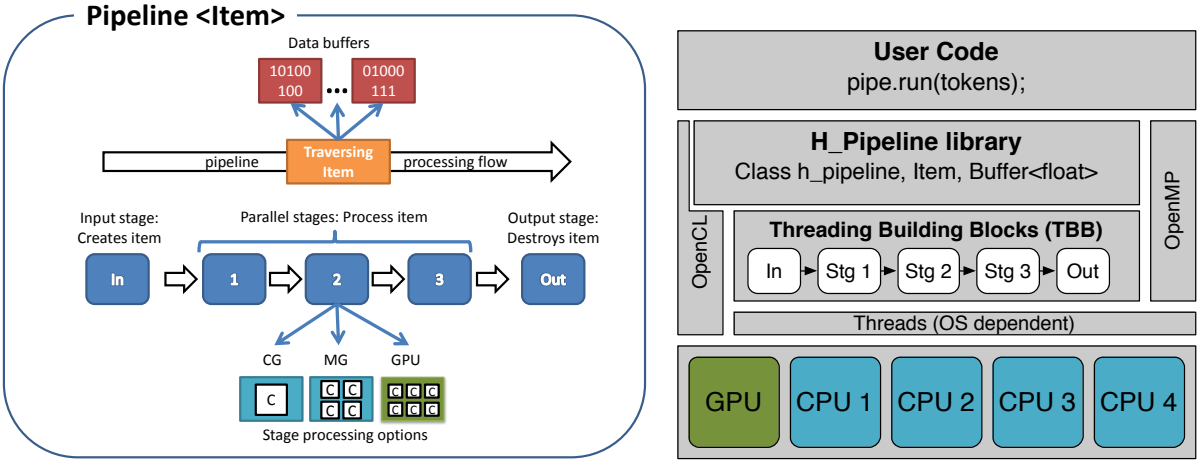
The interface has four main components:

- Items: objects that traverse the pipeline carrying pointers to the data buffers.
- Pipelines: the pipeline itself is composed of $s + 2$ stages. We assume that it contains stages S_{in}, S_1, S_2, ... S_s, S_{out}, being S_{in} and S_{out} the serial Input and Output stages. Pipelines can be configured statically or run in an adaptive-configuration mode. This adaptive-configuration mode is the one that uses our proposed model to dynamically compute the best configuration and to adapt to input changes.
- Stage functions: each processing stage needs to be programmed to run on CPU and/or GPU. The pipeline uses the appropriate function for every stage.
- Buffers: n-dimensional arrays that can be used by both, the host code and the OpenCL kernels.

Fig. 3a shows all components involved in the pipeline operation. 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 use as input and output. To create a new pipeline instance, the user needs to declare a new `Item` subclass (it must extend from a provided `Item` class) that should contain the references to data buffers used by the pipeline stages. For data buffer management, there is a `DataBuffer<T>` template class already defined, 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 make the data accessible to the device (CPU or GPU) where the item has to be processed. Fig. 3b shows the hardware layer with the heterogeneous devices at bottom. On top of that, the middle layers (TBB, OpenCL, OpenMP) provide different programming models to exploit parallelism on the heterogeneous chip. However, we partially hide the details required by these low level libraries by providing a simpler interface based on C++ clases and templates.

The programmer can provide up to three different functions for every stage of the created pipeline: one to implement the 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 (CP, MG, and/or GPU) will not be considered when searching for the best pipeline configuration.

Let's see the components in more detail.



(a) Pipeline building blocks (b) Software stack

Fig. 3: Pipeline building blocks and software stack.

A. *Item class*

Fig. 4 shows a code snippet for the `Item` class declaration used in the ViVid pipeline. First, our pipeline interface is made available by including the `h_pipeline.h` header file (line 1) . It also defines the `h_pipeline` namespace which contains all the classes of the interface. As previously mentioned, before creating the pipeline, the programmer must define an `Item` subclass that will traverse the pipeline (line 7). The item class must extend from `h_pipeline::Item` and declare as many `DataBuffers<T>` members as needed for the pipeline execution. The class constructor and destructor methods must hold the buffers creation and deletion, respectively. Alternatively and to avoid too many operations of buffer allocation/deallocation, a pool of buffers can be used. In that case, acquire and release methods can be invoked so that the same buffers are reused by different items. The Input and Output stages (i.e. the first and last serial stages of the pipeline, S_{in} and S_{out}) automatically call the constructor and destructor of the `Item` class, respectively. In the last part of this section we will show an example of buffer usage (Fig. 7).

```
1  #include "h_pipeline.h"          // Required clases defined here
2  using namespace h_pipeline;      // New namespace
3
4  /*********************************************************************
5  * 1.- ITEM Class (holds the data that traverse the pipeline stages
6  *********************************************************************/
7  class ViVidItem : public h_pipeline::Item {
8  public:
9      //Buffer definitions
10     DataBuffer<float> *frame;    // Input buffer
11     ...
12     DataBuffer<float> *out;   // Output buffer
13
14    //Constructor definition. Allocation or buffer acquire
15    ViVidItem() {
16         //Data Buffer allocation
17         ...
18     }
19     //Destructor definition. Deallocation or buffer release
20     ~ViVidItem() { ... }
21 };
```

Fig. 4: Using the Item Class.

B. *Pipeline*

Fig. 5 shows a pipeline definition and usage example. After declaring the `ViVidItem` as shown in Fig. 4, we can create a new pipeline using that class (line 6) passing as constructors arguments the number of threads, `numThreads`, that will run the pipeline in parallel. In this study we set as maximun number of threads $nC+1$,

being nC the number of CPU cores (see section II-B). Notice also that the CPU (CG and MG granularity) and the GPU functions of the three stages for the ViVid example need to be set up before the pipeline can run (see lines 9 to 11). In case we want to run the pipeline using a static configuration, we use specific methods to configure some aspects of the pipeline, such as the stages that should map to the GPU or the granularity (MG or CG) that should be used in the CPU (line 14). In our example, in line 14, parameter {`1,1,1`} represents the s-tuple that express the mapping of stages to the devices (we will define it next) and the last argument, `USE_MG`, indicates that MG granularity will be exploited when an item is processed on the CPU.

As mentioned before, we assume that a pipeline consists of S_{in}, S_1, S_2, ... S_s, S_{out} stages (s+2). S_{in} and S_{out}, the serial Input and Output stages will be always mapped to the CPU. For any other stage S_i such that $1 \leq i \leq s$, we use a s-tuple to specify all posible stage mappings to the GPU and the CPU devices: {m_1, m_2, ..., m_s}. The i-th element of the tupla, m_i, will specify 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 will check if the GPU is available, in which case it will execute on the GPU; otherwise, it will execute on the CPU. For instance, for the ViVid example of Fig. 2 we represent 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}.

Once the pipeline is configured for a static configuration, it can be run (line 15) by setting the maximum number of items that are allowed to be simultaneously in flight traversing the pipeline. Another option to run the pipeline is to use the adaptive configuration mode (line 18) presented in section IV. Under this mode, our framework dynamically finds the best configuration. In this case, the user has to select the optimization criterion (`THROUGHPUT, ENERGY, THROUGHPUT_ENERGY`) and the maximum overhead allowed due to the training step required when running in the adaptive mode explained in section IV.

```
1  /*************************************************
2  * 2.- Pipeline declaration and usage
3  *************************************************/
4  int main(int argc, char* argv[]){
5    int numThreads = nC+1;   // number of threads = nC+1
6    h_pipeline::pipeline<ViVidItem> pipe(numThreads);
7
8    // Set CG, MG and GPU functions for each stage
9    pipe.add_stage(cg_f1, mg_f1, gpu_f1);
10   pipe.add_stage(cg_f2, mg_f2, gpu_f2);
11   pipe.add_stage(cg_f3, mg_f3, gpu_f3);
12
13   //Setting a static pipeline configuration: mapping '111' and MG
14   pipe.set_configuration({1,1,1}, h_pipeline::USE_MG);
15   pipe.run(numTokens);   // maximum number of items in flight
16
17   //Dispatch of the adaptive configuration mode for the pipeline
18   //pipe.run(numTokens, ENERGY, maxoverhead);
19 }
```

Fig. 5: Pipeline declaration and usage.

C. Pipeline stage functions

An important part of the pipeline definition is the set up of the pipeline stage functionalities. In the interface, the `add_stage()` method (Fig 5, lines 9 to 11) is used to add each one of the stages while identifying the possible functions that may be called to process the items.

Fig. 6 shows an example of these stage functions definition. The programmer can provide three different versions of the same function. The pipeline will use the appropriate version of the function to map the stage to one CPU core, several CPU cores, or the GPU device. Each function receives as argument a pointer to the item to be processed. From such item we can obtain the pointers to the input/output data buffers by using the method `get_HOST_PTR()` to obtain a host pointer, or `get_CL_BUFFER()` to obtain an OpenCL buffer object usable at the GPU device. In both cases, the access type to that buffer inside the function must be indicated by the programmer (options are: `BUF_READ, BUF_WRITE, BUF_READWRITE`).

Fig. 6 shows the definition of two functions that can be invoked on the CPU and a third one to process an item on the GPU. First, in line 5 we have the CPU function for CG granularity, that is basically a serial code to process an item on the CPU. For this granularity, parallelism is exploited at the task level since several cores may be running this function at the same time for different items. Next in line 13, we have the definition for MG granularity, were all the cores will collaborate in processing a single item. Now, data parallelism is

```
1  /*****************************************
2  * 3.- Functions definition example
3  *****************************************/
4  // Example for filter 3 of ViVid
5  void cg_f3(ViVidItem *item) // Coarse grain CPU version
6  {
7      float * out, cla, his;
8      out = item->out->get_HOST_PTR(BUF_WRITE);   // get buffer on host for writing
9      cla = item->cla->get_HOST_PTR(BUF_READ);    // get buffer on host for reading
10     his = item->his->get_HOST_PTR(BUF_READ);    // get buffer on host for reading
11    // do cpu things like  out[XXX] = his[XXX] + cla[XXX];
12 }
13 void mg_f3(ViVidItem *item) // Medium grain CPU version
14 {
15     float * out, cla, his;
16     out = item->out->get_HOST_PTR(BUF_WRITE);   // get buffer on host for writing
17     cla = item->cla->get_HOST_PTR(BUF_READ);    // get buffer on host for reading
18     his = item->his->get_HOST_PTR(BUF_READ);    // get buffer on host for reading
19
20     tbb::parallel_for( 0, aheight, 1, [&] (size_t i) {
21        // do cpu things like  out[i] = his[i] + cla[i];
22     });
23
24     // #pragma omp parallel for
25     // for (size_t i=0; i<aheight; i++) {
26           // do cpu things like  out[i] = his[i] + cla[i];
27     // }
28 }
29 void gpu_f3(ViVidItem *item)    // GPU OpenCL version
30 {
31     cl_mem out,cla,his;
32     out = item->out->get_CL_BUFFER(BUF_WRITE);   // get buffer on device for writing
33     cla = item->cla->get_CL_BUFFER(BUF_READ);    // get buffer on device for reading
34     his = item->his->get_CL_BUFFER(BUF_READ);   // get buffer on device for reading
35
36     // Setting kernel parameters
37     // Launching kernel
38     //...
39 }
```

Fig. 6: Functions for pipeline stages operations (CG, MG, GPU).

exploited, and to that end in this example we rely on `tbb::parallel_for()` (line 20). MG granularity can be also exploited using OpenMP as shown in commented line 24. Finally we have the GPU code defined in the `gpu_f3` function (line 29). Note also that pipeline parallelism is exploited because concurrent items traverse the stages of a pipeline at their own pace.

D. Buffers

As shown previously in Fig. 4, thanks to our `Databuffer<T>` template class, the programmer does not need to manage memory buffers explicitly. The supplied buffer class hides all the data buffer management and the programmer just need to ask for the references to the buffers he wants to use, indicating whether the buffers will be read or/and written. Fig. 7 shows an example of buffer declaration and access. In line 5 a data buffer is declared. In the next line a pointer, `*frame`, is declared to access the former data buffer from the CPU.

The `DataBuffer` class offers a way to set up the type of access to a certain OpenCL buffer, so a method to set it up must be used (line 9) and we can choose to use a Zero-Copy Buffer approach (line 10) or copy data from host CPU to device (and viceversa) when required.

On the creation of the buffer (line 9) we need to indicate the kind of access that this buffer will take from the OpenCL kernel (read or/and write). The actual allocation of the buffer (on host or device memory) will be delayed until the first use. To use the buffer the programmer will invoke the right method: `get_HOST_PTR()` to obtain a host pointer, or `get_CL_BUFFER()` to obtain a device memory object. In the example of Fig. 7 (line 14), the host pointer, `frame`, is initialized using `get_HOST_PTR()` with write intent. In the next line the buffer is initialized with the appropriate data.

```
/*************************************************
* 4.- BUFFER usage
*************************************************/
int main(int argc, char* argv[]){
  DataBuffer<float> * global_frame;   // data buffer
  float *frame;                       // pointer to access the buffer from CPU

 //Specify access mode for OpenCL kernel: BUF_READ  BUF_WRITE  BUF_READWRITE
  global_frame = new DataBuffer<float>(size, BUF_READ);
  global_frame->set_ZCB(true);  //Set Zero Copy Buffer usage
  global_frame->use_Pool(true);  //Use a pool of buffers

  //acquire the buffer reference to write it on the CPU
  frame = global_frame->get_HOST_PTR(BUF_WRITE);
  frame[XXX] = XXX;   // fill the buffer on the CPU
  //Pipeline definition and usage
  ...
}
```

Fig. 7: Example of DataBuffer class usage.

E. Implementation details

In this section we dive into the internal details of the framework that we have implemented. First, we describe the parallel stage class and finally we elaborate on the implementation of the pipeline class.

```
/*************************************************
* Parallel Stage Internal Details
*************************************************/
template <class Item_T>
class parallel_stage : public tbb::filter{
  //members
  int runOnGPU;  //1 runs on GPU, 0 runs on CPU
  bool grain;    //True is MG, False is CG
  void (*cgFunc)(Item_T*);
  void (*mgFunc)(Item_T*);
  void (*gpuFunc)(Item_T*);

  //Constructor
  parallel_stage(void (*cg_f)(Item_T*), void (*mg_f)(Item_T*), void (*gpu_f)(Item_T*)){
    cgFunc=cg_f; mgFunc=mg_f; gpuFunc=gpu_f;
  }
  //Methods
  void setConfiguration(int mapping, bool granularity){
    runOnGPU = mapping; grain = granularity;
  }
  ...
  void * operator()(void * item){
    Item_T *it = (Item_T *) item;
    if(runOnGPU && h_pipeline::isGPUidle()){
      gpuFunc(it);
    }else if(grain){
      mgFunc(it);
    }else{
      cgFunc(it);
    }
    return it;
  }
};
```

Fig. 8: Internal details of `parallel_stage` class.

1) 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. 5 lines 9 to 11). This class holds important instance variables: three of them are function pointers (see Fig. 8 lines 9-11) which point to the functions declared in Fig. 6 (they are initialized in the class constructor at line 14). The other two instance variables, `runOnGPU` and `grain`, 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 `operator()` function (line 22) 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.

```
1  /***********************************************
2  * pipeline class inner details
3  ***********************************************/
4  template <class Item_T>
5  class pipeline : public tbb::pipeline {
6    //members
7    atomic<int> gpuStatus;  //0 GPU is idle, 1 GPU is busy
8    list<parallel_stage> l_stage;
9    int num_stages, nthreads;
10
11   //Constructor
12   pipeline(int numthreads){
13     //Initialize TBB scheduler and OpenCL boilerplate
14     num_stages=0; nthreads=numthreads;
15   }
16
17   //Adding Stages
18   void add_stage(void (*cg_f)(Item_T*), void (*mg_f)(Item_T*), void (*gpu_f)(Item_T*)){
19     parallel_stage<Item_T> * iStage(cg_f, mg_f, gpu_f);
20     l_stage.add(iStage, ++num_stages);
21   }
22
23   //Setting Configuration for all stages
24   void set_configuration(int mappings[], bool granularity){
25     for(int i=0; i<num_stages; i++){
26       l_stage.get(i).setConfiguration(mapping[i], granularity);
27     }
28   }
29
30   //Overloaded pipeline.run(): static configuration mode
31   void run(int tokens){
32     //Build the TBB pipeline and run it
33   }
34
35   //Overloaded pipeline.run(): adaptive configuration mode
36   void run(int tokens, const int criteria, float overhead){
37     while(/*there are more items*/){
38       //Training Phase:
39         setConfiguration({1,1,1}, USE_MG); pipe.run_training(1);
40         setConfiguration({0,0,0}, USE_MG); pipe.run_training(1);
41         setConfiguration({0,0,0}, USE_CG);
42         for(int i=1; i<=(nthreads+1);i++){
43           pipe.run_training(i);
44         }
45         computeModelAndSetBestConfiguration(criteria);
46
47       //Running Phase: It can abort if a change in throughput is detected
48       pipe.run_monitoring(tokens);
49     }
50   }
51   ...
52 };
```

Fig. 9: Internal details of pipeline class.

2) Pipeline class: In this section we introduce some internal details regarding the `pipeline<T>` class. This class is at the top of the software stack described in Fig. 3b, so it is designed to work on top of TBB and OpenCL libraries. Fig. 9 sketches the main implementation decisions we have considered for this `pipeline<T>` class.

The `pipeline` class is the main component of the library, it is responsible to glue the set of stages and schedule the items in flight to maximize the optimization criteria. In order to make this possible the `pipeline` class defines a few member variables (lines 7-9). The `gpuStatus` variable shows the current status of the GPU at runtime: it can be ($0 = Idle$ and $1 = Busy$). This class also defines an ordered list of stages, (`l_stage`), that represents the stages of the pipeline. Additionally, there are two integer variables: `nthreads` which is used to set the number of logical threads to be created and `num_stages` which has

the current count of parallel stages in the list. The class constructor (line 12) initializes the TBB library with the number of threads passed as argument. Then the constructor creates the OpenCL environment (context, command_queues, ...) and selects the GPU as the target device.

The `pipeline` class defines two methods (`add_stage` and `set_configuration`) to configure the topology of the pipeline. As mentioned before, this class keeps a list of stages, where the Input and Output stages (first and last ones) are serial. For each one of the middle stages (parallel or stateless) the function `add_stage` is called with three function pointers passed as arguments (CG, MG, GPU), so a new `parallel_stage` instance is created and inserted in the list (line 18). The `set_configuration` (line 24) method allows the user to set a specific pipeline configuration. This method receives two parameters: an array of zeros and ones, where the i-th element specifies whether the i-th stage can use the GPU (1) or not (0). The second parameter is an enumerated type that sets the type of CPU functor (`USE_MG` for Medium Grain, MG, or `USE_CG` for Coarse Grain, CG) that should be used for all stages.

In order to execute the pipeline, the `run` method must be invoked. Notice that it is possible to invoke two versions of the `pipeline.run()` method (lines 31 and 36), thus two types of modes are available: a static configuration or an adaptive configuration mode. The former (line 31) has a static behavior. This means that just one pipeline configuration is used during the whole execution of the pipeline. In this case, the user is responsible to set the particular pipeline configuration by calling the function `set_configuration` (i.e. `pipe.set_configuration({1,1,1}, USE_MG)`) (see line 14 in Fig. 5). The adaptive configuration mode of the `run` method (line 36) takes three arguments: the maximum number of items in flight, the optimization criteria (THROUGHPUT, ENERGY, THROUGHPUT_ENERGY) and a float number between 0 and 1 that represents the allowed overhead ratio (see section IV-B).

The adaptive configuration `run` method has 2 phases: the training phase and the running phase (see section IV). The training phase carries out three experiments. The first one executes one item on the GPU through all stages (line 39). The second one executes one item on the CPU with MG granularity (line 40) on all stages. Finally, the third experiment launches nthreads (from 1 to nthreads) executions on the CPU with CG granularity (line 43). More details in section IV-A. Notice that the `run_training` method is used here. Then we compute our analytical model (line 45) with the time and energy collected in the previous experiments and the desired optimization criteria. This analytical model returns the best configuration that maximizes the optimization criteria passed as argument. This model is explained with deeper details in section V.

Once the desired configuration is found, the pipeline enters in the second phase (running phase). In this phase, the `run_monitoring` method (line 48) always monitors the throughput and energy. Whenever a change (drop/rise) in throughput is detected, the pipeline checks the overhead parameter and the number of items processed in this phase. If the ratio between the total time taken by the previous training phase and the execution time of the current phase is less than the overhead threshold, then the current running phase is aborted and the training phase is executed again. Otherwise the algorithm continues in the running phase until the overhead ratio is less than the overhead. More details about the computation of the overhead in section IV-B.

IV. Framework

Our framework is particularly suitable for streaming applications that may exhibit a variation in the streaming characteristics. In these cases, we can adjust the pipeline configuration to optimize the desired metric (throughput, or energy, or a tradeoff). Our framework is designed as a two phase engine: first, a *training phase* followed by a *running phase*. The training phase carries out two steps: i) *a measurement collection step*, where some measurements of time and energy are performed on the GPU and CPU; and ii) *an analytical evaluation step*, where our model (see next section) finds the optimal pipeline configuration using the collected measurements. Once the optimal configuration has been found, the framework enters the running phase. In order to adapt to variations in the behavior of the applications, throughput is monitored during the running phase, so that any significant change can return the framework to the training phase. However, the training phase is only performed if the associated overhead is less than a threshold value provided by the user (more details in subsection IV-B).

During the training phase, runtime items are used, so no off-line training is necessary. Also, the runs to collect the measurements are conducted only on the CPU or on the GPU (homogeneous runs). Then, using the proposed model, the framework can predict the behavior of all possible heterogeneous pipeline configurations and select the best.

We assume that the s parallel stages of our streaming application are S_1, S_2, .., S_s and that each item will be executed through all these stages. Additionally, we consider that both the Input and Output stages are serial (S_i, S_o), although this is not a pre-requisite in our model. Our model is based on a set of equations that allow

us to estimate the throughput and energy consumption per item for all possible pipeline alternatives. Let's suppose that our system consists of nC CPU cores and 1 on-chip GPU. Then, our framework traverses the $2^s \cdot (nC+2)$ possible pipeline configurations, and for each one computes the effective throughput, λ_e, and the effective energy per item, E_e. From the estimations, it selects the pipeline configuration for which the optimal is found: highest λ_e or lowest E_e, depending on the metric considered. We can also use any combination of these metrics to define a tradeoff metric and look for the configuration which obtains the optimal value.

A. Measurement Collection step

As mentioned, the equations of our model use the data recorded in the measurement collection step. In this step, we carry out $nC+3$ experiments to obtain all the values needed. Note that this number of experiments is usually much smaller than the $2^s \cdot (nC+2)$ possible alternatives, that thanks to the model we do not need to experimentally assess. For time measurements we use the clock ticks hardware counter, while for the energy measurements, we use the energy counters available on the Ivy Bridge and the Haswell architectures [14], [15]. These counters measure three domains: P, C and G. P or Package means the consumption of the whole chip, including CPU, GPU, memory hierarchy, etc. C is CPU domain and G is the GPU domain. In our model we consider C, G and $U = P - C - G$. Therefore, this last component represents the Uncore energy consumption. For other architectures, energy information can be estimated by either relying on performance counters that can be read by using a library, such as PAPI [16], or by using a power sensor, like the INA231 power monitor integrated with the Exynos 5 on the Odroid XU3 platform [17]. Current trends point out that energy counters will be more widely available in the near future. Anyway, even if energy information is not accesible, our framework is still useful to minimize execution times.

The experiments and measurements we collect are always from homogeneous runs (only GPU or CPU execution). These experiments are:

- CG experiments: we perform 1 experiment in which all stages are executed by one thread in one CPU core. We collect time and energy per stage (see T_k^{CG} and $\left(E_{C_k}^{CG}, E_{G_k}^{CG}, E_{U_k}^{CG}\right)$, $k = 1:s$, in Table I). For energy measurements we collect three components (C, G, U) as explained before. Next, we carry out nC additional experiments in the CPU multicore: on each one, n threads (with n changing from 2 to $nC+1$) process n items (each thread processes one item) throughout all the pipeline stages, i.e. homogeneous CG executions. We collect the total time and energy per item (see $T^{CG}(n)$ and $\left(E_C^{CG}(n), E_G^{CG}(n), E_U^{CG}\right)$, $n = 2 : nC+1$, Table I). Note that the case for one thread was already considered in the first experiment described above. Actually, $T^{CG}(1) = \sum_1^s T_k^{CG}$ and $E_*^{CG}(1) = \sum_1^s E_{*_k}^{CG}$, where $*$ takes the value C, G and U. With these measurements we implicitly incorporate to the model the impact that n threads processing n items have in the memory traffic as well as the scalability behavior in the CPU. To carry out these $nC+1$ experiments, $(nC+2)\cdot(nC+1)/2$ items of the stream are processed.
- MG experiments: we conduct 2 additional experiments in which all stages are executed first by one thread in the GPU, and next by nC threads in the CPU multicore, i.e. homogeneous MG execution, where nC is the number of CPU cores. We collect time and energy per stage (see T_k^G and $\left(E_{C_k}^G, E_{G_k}^G, E_{U_k}^G\right)$, $k = 1:s$, for the GPU and T_k^{MG} and $\left(E_{C_k}^{MG}, E_{G_k}^{MG}, E_{U_k}^{MG}\right)$, $k = 1:s$, for MG on the CPU, in Table II). Two additional items of the stream are processed to carry out these two MG experiments.

TABLE I: Measured time per item, T, and energy per item, E, for CG. Also time to collect them. Note that in practice, $T^{CG}(1) = \sum_1^s T_k^{CG}$, $E_*^{CG}(1) = \sum_1^s E_{*_k}^{CG}$, where $*$ takes the value C, G and U, and $t^{CG}(1) = t^{CG}$, so no additional experiment need to be conducted for 1 thread.

Parameter	Device	time col.	Description
$T_1^{CG}, \ldots, T_s^{CG}$	CPU	t^{CG}	time per item (and stage) on the CG exec. (1 thread)
$\left(E_{C_1}^{CG}, E_{G_1}^{CG}, E_{U_1}^{CG}\right)$ $\cdots$ $\left(E_{C_s}^{CG}, E_{G_s}^{CG}, E_{U_s}^{CG}\right)$	CPU		(C,G,U) components of the energy per item (and stage) on the CG exec. (1 thread)
$T^{CG}(1), \ldots T^{CG}(n_m)$	CPU	$t^{CG}(1), \ldots\ t^{CG}(n_m)$	total time per item on the CG exec. (1, 2 ... $n_m = nC+1$ threads)
$\left(E_C^{CG}(1), E_G^{CG}(1), E_U^{CG}(1)\right)$ $\cdots$ $\left(E_C^{CG}(n_m), E_G^{CG}(n_m), E_U^{CG}(n_m)\right)$	CPU		(C,G,U) comp. of the total energy per item on the CG exec. (1, 2 ... $n_m = nC+1$ threads)

Notice that the model can also be used when hyperthreading is enabled. To consider the use of hyperthreading, the collecting measurement step needs to run $nC*2+3$ (instead of $nC+3$) experiments.

TABLE II: Measured time per item, T, and energy per item, E, and per stage for GPU and for MG. Also time to collect them.

Parameter	Device	time col.	Description
$T_1^G, \ldots, T_s^G$	GPU	t^G	time per item (and stage) on the GPU exec. (1 thread)
$(E_{C_1}^G, E_{G_1}^G, E_{U_1}^G)$ $\cdots$ $(E_{C_s}^G, E_{G_s}^G, E_{U_s}^G)$	GPU		(C,G,U) components of the energy per item (and stage) on the GPU exec.
$T_1^{MG}, \ldots, T_s^{MG}$	CPU	t^{MG}	time per item (and stage) on the MG exec. (nC threads)
$(E_{C_1}^{MG}, E_{G_1}^{MG}, E_{U_1}^{MG})$ $\cdots$ $(E_{C_s}^{MG}, E_{G_s}^{MG}, E_{U_s}^{MG})$	CPU		(C,G,U) components of the energy per item (and stage) on the MG exec.

B. Controlling the overhead of the measurement collection step

The cost of the training phase is mainly due to the measurement collection step, where items are processed inefficiently due to the homogeneous runs (only one device is used at a time) carried out during this step. After the measurement collection step and the subsequent model instantiation, we can control the period of time at which a new training can be performed to guarantee that the overhead due to the training is bounded. Let's suppose that λ_c is the throughput to process items in the current configuration (after performing the last training) and that $N_s = (nC+2)\cdot(nC+1)/2+2$ is the number of items processed during the measurement collection step (see Tables I and II). We define Δt as the overhead or time increment incurred during that last training. It is computed as the time needed to carry out the collection step minus the time the N_s items would have taken at the current λ_c throughput:

$$\Delta t = \left(t^{CG} + \left(\sum_{n=2}^{nC+1} t^{CG}(n) \right) + t^{MG} + t^{G} \right) - N_s/\lambda_c \tag{1}$$

The *overhead* that the last training has supposed with respect to the current throughput can be computed as,

$$ov = \frac{\Delta t}{t + \Delta t}$$

We can keep this overhead below a threshold value, ov_{thl}, if $\Delta t/(\Delta t + t) < ov_{thl}$, or in other words:

$$t > \frac{(1 - ov_{thl})}{ov_{thl}} \cdot \Delta t \tag{2}$$

This expression gives us a lower bound for controlling the time for entering in a new training phase, t, with a maximum allowed overhead. For example, for the ViVid application on the Ivy Bridge chip presented in section VI, 5% of overhead is paid when the training takes place every 3.7 sec. for low resolution input video. However, in general, the training phase leads to pipeline configuration changes that deliver a better throughput which result in that the training overhead is amortized after processing a few items with the new recommended pipeline configuration, as we discuss in section VI.

V. Analytical model: finding the optimal

We model the heterogeneous pipeline configurations as a closed network of logical queues where items arrive following a Poisson process [18]. This is pertinent in the context of streaming applications where item arrivals can be considered independent and inter-arrival time can be viewed as following an exponential distribution [19]. In these closed systems, items can be viewed as circulating continuously and never leaving the network of queues, because a new item can not enter until a previous one leaves. Fig. 10 shows our models for the Decoupled and Coupled configurations, where we can see that an item can follow one of two alternative paths before entering again in the system. In our models, we can find one or more queues on each path. In particular, any sequence of consecutive stages mapped to one device (the GPU or the CPU) is represented as a M/M/1 queue. This stands for a logical queue where a single server serves items that arrive according to

a Poisson process and have exponentially distributed service times. Although there can be several concurrent threads on a device processing the sequence of stages represented by the queue, we have found that assuming one logical server on each queue still provides accuracy while keeping the equations of the model simple. In a closed network of queues, the following expressions define the *flow balance conditions* [20] at equilibrium,

$$\lambda_e = \sum_{path_j} \lambda_j \tag{3}$$

$$\sum_{path_j} p_j = 1 \tag{4}$$

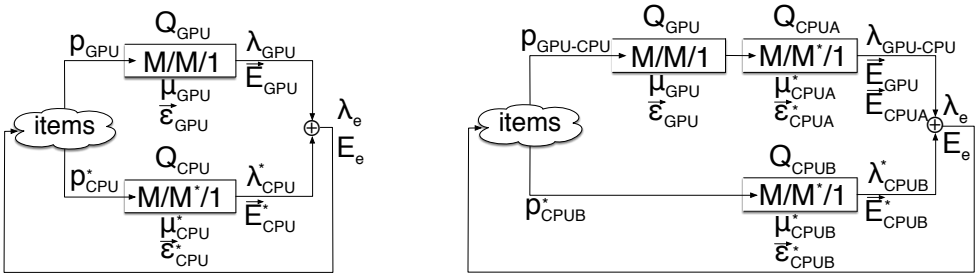
$$p_j \cdot \lambda_e = \lambda_j \tag{5}$$

These equations allow us to relate the relative throughput of each path in a configuration with the effective throughput in that configuration. A path $path_j$ refers to one of the two possible paths defined in section II for each configuration: for DP configurations, it is either the GPU path or the CPU path (note the subindices for each path (GPU, CPU) in the model in Fig. 10a); For CP configurations, it is either the GPU-CPU path or the CPU path (note the subindices for each path (GPU-CPU, CPUB) in the model in Fig. 10b). In particular, equation 3 establishes that given the relative throughputs of the paths in the system, λ_j, then the effective throughput, λ_e, may be obtained as a sum (i.e. combining independent Poisson processes leads to a Poisson process). Equation 4 states that splitting a Poisson process probabilistically leads to Poisson processes, being p_j the probability of taking $path_j$. Equation 5 states that, in a M/M/1 queue at equilibrium, the average flow rate leaving the queue will also be the same as the average flow rate entering the queue.

We define two parameters for each queue Q_i: the *service rate*, μ_i, or average rate at which an item is processed, and the *energy rate*, $\overrightarrow{\varepsilon}_i$, or average energy consumed by an item in the corresponding device (GPU or CPU) where the queue works. This parameter represents a vector with three components, one for each energy domain: $(\varepsilon_{i_C}, \varepsilon_{i_G}, \varepsilon_{i_U})$. They can be seen as the components of the average energy consumed by an item on a device due to the stages represented by the queue Q_i, when the device is the only one working in the system (homogeneous execution).

In any case, as the network is in equilibrium, each individual queue must be in equilibrium. This means that the utilization of the queue's server, ρ_i, is less than 100%, that is, the ratio between the relative throughput of the corresponding path, λ_j, and the queue's service rate, μ_i, is at most 1 [18],

$$\rho_i = \frac{\lambda_j}{\mu_i} \leq 1 \tag{6}$$



(a) Model for DP configurations

(b) Model for CP configurations

Fig. 10: Closed network of queues.

Regarding the energy, as each individual queue Q_i is in equilibrium, we assume that the energy utilization on the corresponding device, $\rho_i^{\overrightarrow{E}}$, is proportional to the probability of items serviced on the corresponding queue, p_j, or in other words,

$$\rho_i^{\overrightarrow{E}} = p_j \leq 1, \quad \overrightarrow{E}_i = \rho_i^{\overrightarrow{E}} \cdot \overrightarrow{\varepsilon}_i \tag{7}$$

This expression allows us to estimate the *relative energy per item* consumed by queue Q_i on the corresponding device (GPU or CPU), $\overrightarrow{E}_i$. This parameter is also a vector that consists of three components: $(E_{i_C}, E_{i_G}, E_{i_U})$. In the case there were several logical queues mapped on a device, from Q_1 to Q_d, then the relative energy per item consumed by the queues in the device is the sum of the relative energy per item for all the queues working in the device: $\sum_{i=1}^{d} \overrightarrow{E}_i = \left(\sum_{i=1}^{d} E_{i_C}, \sum_{i=1}^{d} E_{i_G}, \sum_{i=1}^{d} E_{i_U}\right)$. These components can be seen as the components of the energy consumed by the items that a device processes when the device is the only one working in the system (homogeneous execution).

However, the effective energy consumed by the GPU and CPU when both devices are working in the system (heterogeneous execution), is not the sum of the relative energies of the queues on each device. Let's define the *effective energy per item* consumed in the system, E_e, as,

$$E_e = min\left(\frac{TDP}{\lambda_e}, E_{e_C} + E_{e_G} + E_{e_U}\right) \tag{8}$$

where TDP is the power budget of the chip, λ_e the effective throughput, and $E_{e_C} + E_{e_G} + E_{e_U}$ the effective energy consumed when the TDP is not reached. For the heterogeneous chips studied, we have found that in case the TDP is not reached, then each component of the effective energy is given by the dominant component of the relative energy computed for each device. This is what we call the *energy balance condition*. Let's suppose that the relative energy per item for all the queues in the GPU device is given by $\overrightarrow{E}_{GPU} = (E_{GPU_C}, E_{GPU_G}, E_{GPU_U})$, and the relative energy per item for all the queues in the CPU device is given by $\overrightarrow{E}_{CPU} = (E_{CPU_C}, E_{CPU_G}, E_{CPU_U})$. The rationale for the energy balance condition is that the C-component of the effective energy is typically dominated by the C-component of the relative energy of the CPU device, E_{CPU_C}, while the C-component of the GPU device, E_{GPU_C} is just a "residual" or standby consumption when the CPU is idle. Remember that this C-component of the relative energy of the GPU device is obtained with homogeneous runs (runs on the GPU where the CPU is idle) during the measurement collection step. On an heterogeneous run, however, the CPU is not idle, and so the standby consumption measured during the homogeneous run is already included in the C-component of the relative energy of the CPU, E_{CPU_C}, and does not need to be included again. A similar argument can be made for the G-component of the effective energy. With respect to the U-component, we have observed that the effective energy consumed is determined by the device (CPU or GPU) that processes a higher load.

Next, Sections V-A and V-B explain how we model Decoupled and Coupled configurations, respectively, and how we incorporate the granularity to the models.

A. Model for Decoupled configurations

These configurations are shown in Figs. 1a and 1c (DP-CG and DP-MG, respectively). Fig. 10a depicts our model for them. As explained in section II, in these configurations there is a GPU path in which a thread processes an item through all stages in the GPU, and also there is a CPU path in which n concurrent threads process other item/s through all the stages in the CPU device.

The GPU device is modeled with Q_{GPU} which is the M/M/1 queue that services all the stages for the items that go through the GPU path. This queue is characterized with two parameters: μ_{GPU}, the service rate of the GPU server, and $\overrightarrow{\varepsilon}_{GPU}$, the energy rate consumed by the queue in the GPU device. These parameters are computed from the time and energy measurements taken in the collection step, as we show in Table III. For both parameters we consider the time and the energy per item of all the stages S_k that are mapped to the GPU (k from 1 to s, see Table II).

The CPU device is modeled with Q_{CPU} which is the M/M*/1 queue that services all the stages for the items that go through the CPU path. The * stands for the different instantiations of the queue, depending on the granularity exploited. For the CG granularity, the queue is characterized with two parameters: $\mu_{CPU}^{CG}(n)$, the service rate of the CPU server under CG granularity, and $\overrightarrow{\varepsilon}_{CPU}^{CG}(n)$, the energy rate consumed by the queue in the CPU device under CG granularity. Note that under the CG granularity the CPU device can run from 0 to nC concurrent threads. The $n = 0$ case represents in fact the GPU homogeneous execution, while the $n = nC$ represents the maximun number of threads in the CPU path. Therefore, for the CG granularity, both the service rate and the energy rate are computed for each possible number of threads. Table III shows

how these parameters are computed, where we see that time and energy are taken from the measurements in Table I. Regarding the MG granularity, the queue is defined by μ^{MG}_{CPU} and $\overrightarrow{\varepsilon}^{MG}_{CPU}$. In Table III we show these parameters, where we notice that time and energy are taken from the measurements in Table II.

TABLE III: Parameters of the DP-* configurations. * stands for CG or MG. s is the number of stages.

Parameter	Device / Gr.	Value	Description
μ_{GPU}	GPU	$\frac{1}{\sum_{k=1}^{s} T^G_k}$	service rate for the stages mapped to the GPU
$\overrightarrow{\varepsilon}_{GPU}$	GPU	$\left(\sum_{k=1}^{s} E^G_{C_k}, \sum_{k=1}^{s} E^G_{G_k}, \sum_{k=1}^{s} E^G_{U_k}\right)$	energy rate consumed by the stages mapped to the GPU
λ_{GPU}	GPU	μ_{GPU}	relative throughput of the GPU path
$\overrightarrow{E}_{GPU}$	GPU	$p_{GPU} \cdot \overrightarrow{\varepsilon}_{GPU}$	relative energy per item consumed by Q_{GPU}
$\mu^{CG}_{CPU}(n)$	CPU / CG	$\frac{1}{T^{CG}(n)},\ n = 0 : nC$	service rate for the stages mapped to the CPU under CG granularity and n threads
$\overrightarrow{\varepsilon}^{CG}_{CPU}(n)$	CPU / CG	$\left(E^{CG}_C(n), E^{CG}_G(n), E^{CG}_U(n)\right),\ n = 0 : nC$	energy rate consumed by the stages mapped to the CPU under CG and n threads
μ^{MG}_{CPU}	CPU / MG	$\frac{1}{\sum_{k=1}^{s} T^{MG}_k}$	service rate for the stages mapped to the CPU under MG granularity
$\overrightarrow{\varepsilon}^{MG}_{CPU}$	CPU / MG	$\left(\sum_{k=1}^{s} E^{MG}_{C_k}, \sum_{k=1}^{s} E^{MG}_{G_k}, \sum_{k=1}^{s} E^{MG}_{U_k}\right)$	energy rate consumed by the stages mapped to the CPU under MG granularity
λ^*_{CPU}	CPU / *	μ^*_{CPU}	relative throughput of the CPU path
$\overrightarrow{E}^*_{CPU}$	CPU / *	$p^*_{CPU} \cdot \overrightarrow{\varepsilon}^*_{CPU}$	relative energy per item consumed by Q_{CPU}
λ_e	GPU + CPU	$\lambda_{GPU} + \lambda^*_{CPU}$	effective throughput of the system
E_e	GPU + CPU	$min\left(\frac{TDP}{\lambda_e}, E_{e_C} + E_{e_G} + E_{e_U}\right)$	effective energy per item consumed in the system. See eq. 9

Since our queues are in equilibrium, and we assume maximum utilization of the servers on each queue, by applying equation 6 we get $\rho_{GPU} = 1$ and $\rho^*_{CPU} = 1$. From this assumption, we find that the relative throughput for each path is given by the corresponding queue's service rate, that is, $\lambda_{GPU} = \mu_{GPU}$ and $\lambda^*_{CPU} = \mu^*_{CPU}$. Also, the flow balance conditions at equilibrium (equations 3-5) allow us to compute the effective throughput of the system, $\lambda_e = \lambda_{GPU} + \lambda^*_{CPU}$, and the probability that an item goes through the GPU path, $p_{GPU} = \lambda_{GPU}/\lambda_e$, or the probability that it goes through the CPU path, $p^*_{CPU} = \lambda^*_{CPU}/\lambda_e$.

On the other hand, by applying equation 7 we get that the energy utilization of each queue on the corresponding device is proportional to the probability of items serviced on the queue, or in other words, $\rho^{\overrightarrow{E}}_{GPU} = p_{GPU}$ and $\rho^{\overrightarrow{E}^*}_{CPU} = p^*_{CPU}$. This assumption allows us to estimate the relative energy per item consumed by Q_{GPU} in the GPU device, $\overrightarrow{E}_{GPU} = p_{GPU} \cdot \overrightarrow{\varepsilon}_{GPU}$ and by Q_{CPU} in the CPU device, $\overrightarrow{E}^*_{CPU} = p^*_{CPU} \cdot \overrightarrow{\varepsilon}^*_{CPU}$ (for CG or MG granularities), respectively.

The effective energy per item consumed in the system, E_e, can be computed as the minimum of TDP/λ_e and the sum of three components, as defined in equation 8. If the TDP is not reached, then each component can be computed by the energy balance condition that establishes that each component of the effective energy is given by the dominant component of the relative energy computed for each device. In particular, this condition in the DP-* configurations means,

$$\begin{aligned}(E_{e_C}, E_{e_G}, E_{e_U}) &= max\left(\overrightarrow{E}_{GPU}, \overrightarrow{E}^*_{CPU}\right) = \\ &= \left(max(E_{GPU_C}, E^*_{CPU_C}), max(E_{GPU_G}, E^*_{CPU_G}), max(E_{GPU_U}, E^*_{CPU_U})\right) \end{aligned} \quad (9)$$

B. Model for Coupled configurations

These configurations are shown in Figs. 1b and 1d (CP-CG and CP-MG, respectively). Fig. 10b depicts our model for them. In these configurations there is a GPU-CPU path in which a thread processes a item through some stages in the GPU and other stages in a CPU core, and there can also be a CPU path in which other concurrent threads process items through all the stages in the remaining CPU cores. To model the service

provided by a sequence of stages mapped to each device on each path, we use a logical queue. Thus, in the GPU-CPU path we can find at least a Q_{GPU} which is the M/M/1 queue that represents the sequence of consecutive stages that service an item in the GPU device, and at least a Q_{CPUA} which is a M/M*/1 queue that represents the rest of stages that service the item in the CPU device (* stands for the granularity studied). For simplicity, in the figure we have represented the case in which the item is first processed by some consecutive stages in the GPU, and later by the rest of stages in the CPU. In case of a mapping where the item is first processed by consecutive stages mapped to the CPU, then to the GPU, then to the CPU, and so on, the model would include more logical queues in the GPU-CPU path: first a Q_{CPUA}, followed by a Q_{GPU}, then another Q_{CPUA}, and so on.

Each Q_{GPU} queue is characterized with two parameters: μ_{GPU}, the service rate due to the consecutive stages mapped to the GPU server, and $\overrightarrow{\varepsilon}_{GPU}$, the energy rate consumed by those stages in the GPU device. These parameters are computed as we show in Table IV. For both parameters we just consider the time and the energy per item of the corresponding consecutive stages S_k that are mapped to the GPU ($S_k \in Q_{GPU}$). Also, each Q_{CPUA} queue is characterized with two parameters, depending on the granularity. For the CG granularity, the parameters are: μ^{CG}_{CPUA}, the service rate due to the consecutive stages mapped to the CPU under CG granularity, and $\overrightarrow{\varepsilon}^{CG}_{CPUA}$, the energy rate consumed by those stages in the CPU device under CG granularity. Table IV shows how these parameters are computed, where time and energy come from measurements in Table I. Regarding the MG granularity, the Q_{CPUA} queue is defined by μ^{MG}_{CPUA} and $\overrightarrow{\varepsilon}^{MG}_{CPUA}$. In Table IV we show these parameters, where we notice that time and energy are taken from measurements in Table II.

On the other hand, the stages mapped to the CPU in the CPU path, are modeled with Q_{CPUB} which is a M/M*/1 queue. Again, * stands for the different instantiations of the queue, depending on the granularity. For the CG granularity, the queue is characterized with: $\mu^{CG}_{CPUB}(n)$, the service rate of the CPU under CG granularity, and $\overrightarrow{\varepsilon}^{CG}_{CPUB}(n)$, the energy rate consumed by the server in the CPU device under CG granularity. With CG, the CPU can run from 0 to nC concurrent threads, in addition to the coupled GPU-CPU thread that serves the GPU-CPU path. Therefore, for CG, the service rate is computed taking into account this additional coupled thread and we model it assuming that the coupled thread is interfering with the threads that are working concurrently on the CPU. We model this interference by subtracting to the service rate of $n+1$ concurrent threads running in the CPU (because the CPU consists of Q_{CPUA} and Q_{CPUB}), a virtual service rate of 1 thread that is executing in the GPU-CPU path (Q_{CPUA}, the coupled thread). The energy rate is computed for the n concurrent threads working on the queue. In any case, for CG, both the service rate and the energy rate are computed for each number of threads. Table IV shows how these parameters are computed, where time and energy come from measurements in Table I. Regarding the MG granularity, the queue is defined by μ^{MG}_{CPUB} and $\overrightarrow{\varepsilon}^{MG}_{CPUB}$. Under this granularity, all the CPU threads will be serving the Q_{CPUA}. Therefore, we assume that Q_{CPUB} will have a very low probability of serving new items, and so, $\mu^{MG}_{CPUB} = 0$ and $\overrightarrow{\varepsilon}^{MG}_{CPUB} = 0$.

In this configuration, we assume optimistic utilization of servers on each queue. By applying equation 6 we get $\rho_{GPU} \leq 1$, $\rho^*_{CPUA} \leq 1$ and $\rho^*_{CPUB} \leq 1$. From these expressions we find that a solution for the relative throughput for each path is given by, $\lambda_{GPU-CPU} = min(\mu_{GPU}, \mu^*_{CPUA})$ and $\lambda^*_{CPUB} = \mu^*_{CPUB}$. In general, if there were more logical queues in the GPU-CPU path, then a solution for $\lambda_{GPU-CPU}$ could be the minimum of the corresponding service rates in the path. Again, the flow balance conditions at equilibrium (equations 3-5) lead to computing the effective throughput of the system as $\lambda_e = \lambda_{GPU-CPU} + \lambda^*_{CPUB}$, and the probability that an item goes through the GPU-CPU path as $p_{GPU-CPU} = \lambda_{GPU-CPU}/\lambda_e$, or through the CPU path as $p^*_{CPUB} = \lambda^*_{CPUB}/\lambda_e$.

Similar to the DP-* configurations, we assume that the energy utilization of each queue on each device is proportional to the probability of items serviced on the corresponding queue, as defined in equation 7. This means $\rho^{\overrightarrow{E}}_{GPU} = p_{GPU-CPU}$, $\rho^{\overrightarrow{E}^*}_{CPUA} = p_{GPU-CPU}$ and $\rho^{\overrightarrow{E}^*}_{CPUB} = p^*_{CPUB}$. These expressions allow us to estimate the relative energy per item consumed on the GPU device, $\overrightarrow{E}_{GPU} = p_{GPU-CPU} \cdot \overrightarrow{\varepsilon}_{GPU}$ and on the CPU device, $\overrightarrow{E}^*_{CPUA} + \overrightarrow{E}^*_{CPUB} = p_{GPU-CPU} \cdot \overrightarrow{\varepsilon}^*_{CPUA} + p^*_{CPUB} \cdot \overrightarrow{\varepsilon}^*_{CPUB}$ (for CG or MG granularities), respectively. As we see, in the CP-* configurations we estimate the relative energy per item consumed in the CPU from the activity in Q_{CPUA} and in Q_{CPUB}. In general, if there were more logical queues in the GPU-CPU path, then all the resultant $\overrightarrow{E}_{GPU}$ for the different Q_{GPU} should be added to estimate the relative

TABLE IV: Parameters of the CP-* configurations.* stands for CG or MG granularities.

Parameter	Device / Gr.	Value	Description
μ_{GPU}	GPU	$\frac{1}{\sum_{S_k \in Q_{GPU}} T_k^G}$	service rate of stages mapped to Q_{GPU} in the GPU-CPU path
$\overrightarrow{\varepsilon}_{GPU}$	GPU	$\left(\sum_{S_k \in Q_{GPU}} E_{C_k}^G, \sum_{S_k \in Q_{GPU}} E_{G_k}^G, \sum_{S_k \in Q_{GPU}} E_{U_k}^G\right)$	energy rate consumed by stages mapped to Q_{GPU} in the GPU-CPU path
μ_{CPUA}^{CG}	CPU / CG	$\frac{1}{\sum_{S_k \in Q_{CPUA}} T_k^{CG}}$	service rate of stages mapped to Q_{CPUA} in the GPU-CPU path under CG granularity
$\overrightarrow{\varepsilon}_{CPUA}^{CG}$	CPU / CG	$\left(\sum_{S_k \in Q_{CPUA}} E_{C_k}^{CG}, \sum_{S_k \in Q_{CPUA}} E_{G_k}^{CG}, \sum_{S_k \in Q_{CPUA}} E_{U_k}^{CG}\right)$	energy rate consumed by stages mapped to Q_{CPUA} in the GPU-CPU path under CG
μ_{CPUA}^{MG}	CPU / MG	$\frac{1}{\sum_{S_k \in Q_{CPUA}} T_k^{MG}}$	service rate of stages mapped to Q_{CPUA} in the GPU-CPU path under MG granularity
$\overrightarrow{\varepsilon}_{CPUA}^{MG}$	CPU / MG	$\left(\sum_{S_k \in Q_{CPUA}} E_{C_k}^{MG}, \sum_{S_k \in Q_{CPUA}} E_{G_k}^{MG}, \sum_{S_k \in Q_{CPUA}} E_{U_k}^{MG}\right)$	energy rate consumed by stages mapped to Q_{CPUA} in the GPU-CPU path under MG
$\lambda_{GPU-CPU}^*$	GPU-CPU / *	$\min(\mu_{GPU}, \mu_{CPUA}^*)$	relative throughput of the GPU-CPU path
$\overrightarrow{E}_{GPU}$	GPU	$p_{GPU-CPU} \cdot \overrightarrow{\varepsilon}_{GPU}$	relative energy per item consumed by stages mapped to the Q_{GPU} in the GPU-CPU path
$\overrightarrow{E}_{CPUA}^*$	CPU / *	$p_{GPU-CPU} \cdot \overrightarrow{\varepsilon}_{CPUA}^*$	relative energy per item consumed by stages mapped to Q_{CPUA} in the GPU-CPU path
$\mu_{CPUB}^{CG}(n)$	CPU / CG	$\frac{1}{T^{CG}(n+1)} - \frac{1}{T^{CG}(1)}, \ n = 0 : nC$	service rate for the stages mapped to Q_{CPUB} in the CPU path under CG and n threads
$\overrightarrow{\varepsilon}_{CPUB}^{CG}(n)$	CPU / CG	$\left(E_C^{CG}(n), E_G^{CG}(n), E_U^{CG}(n)\right), n = 0 : nC$	energy rate consumed by the stages mapped in CPU path under CG and n threads
μ_{CPUB}^{MG}	CPU / MG	0	service rate for the stages mapped to Q_{CPUB} in the CPU path under MG granularity
$\overrightarrow{\varepsilon}_{CPUB}^{MG}$	CPU / MG	0	energy per item rate consumed by the stages mapped in the CPU path under MG granularity
λ_{CPUB}^*	CPU / *	μ_{CPUB}^*	relative throughput of the CPU path
$\overrightarrow{E}_{CPUB}^*$	CPU / *	$p_{CPUB}^* \cdot \overrightarrow{\varepsilon}_{CPUB}^*$	relative energy per item consumed by the stages mapped in the CPU path
λ_e	GPU + CPU	$\lambda_{GPU-CPU} + \lambda_{CPUB}^*$	effective throughput of the system
E_e	GPU + CPU	$min\left(\frac{TDP}{\lambda_e}, E_{e_C} + E_{e_G} + E_{e_U}\right)$	effective energy per item consumed in the system. See eq. 10

energy per item consumed in the GPU device. Similarly, the $\overrightarrow{E}_{CPUA}$ terms should be added to estimate the relative energy per item consumed in the CPU in that path. Thus, as in DP-* configurations, the effective energy consumed in the system, E_e, can be computed as the minimum of TDP/λ_e and the sum of three components (see eq. 8). If the TDP is not reached, then following the energy balance condition we get for CP-* configurations that,

$$(E_{e_C}, E_{e_G}, E_{e_U}) = max\left(\overrightarrow{E}_{GPU}, \overrightarrow{E}_{CPUA}^* + \overrightarrow{E}_{CPUB}^*\right) =$$
$$= \left(max(E_{GPU_C}, E_{CPUA_C}^* + E_{CPUB_C}^*), max(E_{GPU_G}, E_{CPUA_G}^* + E_{CPUB_G}^*), max(E_{GPU_U}, E_{CPUA_U}^* + E_{CPUB_U}^*)\right) \quad (10)$$

C. Model extensions

Notice that in our throughput and energy estimations, the transfer time and energy between GPU and CPU devices have not been explicitly stated for the sake of readability. However, they can be easily incorporated into our models: the measurement collection step can collect the host-to-device and device-to-host time and energy for each stage mapped to the GPU during the GPU homogeneous run[3]. Then, the service rate of each Q_{GPU} queue, μ_{GPU}, would need to add the host-to-device time of the first stage and the device-to-host time of the last consecutive stage mapped to the corresponding Q_{GPU}. Similarly, the energy rate of the queue

[3]If Zero-Copy-Buffer approach is used then this information can not be measured easily, but in this case time and energy due to communication operations can be disregarded.

$\overrightarrow{\varepsilon}_{GPU}$, would also consider the energy consumed during the transfers on those stages. In the integrated GPUs that we use in our experiments and for our benchmarks, the transfer times are negligible and can be ignored without affecting the accuracy of the model. For discrete GPUs, we expect transfer times to have a higher impact, though.

Our model can also be extended to include the alternatives not considered in section II-A. For instance, the splitting of an item on each stage would be modelled with a GPU-CPU path, where each stage i would be represented by a Q_{CPUi} and a Q_{GPUi} queue, and the service rates of the corresponding queues should be the time to process the portion of the item in the corresponding device and stage (similarly for the energy rate). The model for the alternative in which one stage can have items exploiting both MG and CG granularities in the CPU multicore can be modelled as two independent paths, with a CPU queue on each one: one path with a Q_{CPUA} queue should consider in its service rate the time to compute the item on the stage under one type of granularity (for example the MG granularity) and the other path with a Q_{CPUB} queue should consider the time to compute the item on the stage under the other type of granularity (the CG granularity in the example). Similar considerations should be taken for computing the energy rate on each queue. For this alternative, the GPU could be incorporated as a Q_{GPU} queue to one of the paths in the case of a Coupled Configuration (the GPU-CPU path as shown in Figure 10b), or in the case of a Decoupled Configuration the GPU would be incorporated to one independent path with a Q_{GPU} queue (the GPU path as shown in Figure 10a). The model for the alternative in with some stages exploit MG while others exploit CG granularity would be similar to the ones studied in this paper, but in these cases the service rates of the Q_{CPU} queues should consider the time to process the item under MG or CG granularities in the corresponding stages (similarly for the energy rate). In any case, due to the constraints commented in section II-A we do not explore these alternatives further.

In the next section, we study the accuracy of our models in two different heterogenous chips by using a set of real applications as well as the benefits of adapting to changes in the input stream.

VI. Experimental results

In this section we present our experimental results. Section VI-A discusses our evaluation methodology; Section VI-B shows the benefit of the analytical model by comparing its performance with a state-of the art baseline approach as well as a study of the overhead due to the training phase and the profit due to the adaptive nature of our framework; Section VI-C discusses our experimental results in detail and compares the throughput and energy predicted by the model with the values measured.

A. Evaluation methodology

Two Intel Quad-Core processors have been used in our experiments: a Core i5-3450, 3.1GHz, 77W TDP based on the Ivy Bridge architecture, and a Core i7-4770, 3.4GHz, 84W TDP based on the Haswell one. Both processors feature Advance Vector Extensions (AVX) and have an on-chip GPU, the HD-2500 and HD-4600, respectively. Although the Core i7 supports hyperthreading, we found that hyperthreading was not beneficial for our applications, maybe because our benchmarks implementations use the AVX vector units and they fully utilize the computational resources. Thus, only one thread per core was considered for all experiments, and so the upper value for n is 5 threads (4 cores plus 1 GPU). We rely on Intel Performance Counter Monitor (PCM) tool [15] to access the HW counters (energy, clock ticks, L2 and L3 misses, etc). Intel TBB 4.1 provides the core template to implement the pipeline [21]. Inside each pipeline stage, we use Intel OpenCL SDK 3.0 for the stages that can be executed on the GPU, or AVX intrinsics for the computations conducted on the cores. For the MG results, we implemented nested parallelism on each stage either using TBB `parallel_for` or OpenCL (it depends on the benchmark, as we will note for each code in the next sections). All versions have been compiled using Intel C++ Compiler 14.0 with -O3 optimization flag. We measured time and energy in 10 executions of the applications and compute the average. The reported metrics are throughput, λ, energy per item, E, and as a tradeoff metric, throughput/energy, λ/E. Therefore, λ is the number of frames per second, fps, E stands for the Joules per frame, and λ/E is the fps/Joule.

We validate our framework on Ivy Bridge and Haswell heterogeneous chips using four real applications: ViVid [10], with Low Definition (LD) videos (600×416 pixels) and High Definition (HD) videos (1920×1080 pixels), SRAD [1], Tracking [22] and Scene Recognition [23]. For all the benchmarks and the heterogeneous

TABLE V: Comparison of alternatives. For both λ and λ/E the higher the better.

			Homogenous Results			Heterog. Results			
Bench.	Architect.	Metric	CPU MG	CPU CG	GPU	Baseline	Best	Improv.	Best conf.
ViVid LD	Ivy Bridge	λ (fps)	40	62	10	51	65	27%	CP-CG (5)
		λ/E (fps/J)	46	83	8	66	92	40%	CP-CG (5)
	Haswell	λ (fps)	59	47	22	80	91	13%	CP-MG
		λ/E (fps/J)	61	43	24	116	134	15%	CP-MG
ViVid HD	Ivy Bridge	λ (fps)	3.7	3.1	1.1	5.6	5.9	5%	CP-MG
		λ/E (fps/J)	0.3	0.2	0.1	0.6	0.7	15%	CP-MG
	Haswell	λ (fps)	5.4	2.8	2.7	6.5	7.2	10%	CP-MG
		λ/E (fps/J)	0.5	0.1	0.3	0.78	0.9	12%	CP-MG
SRAD	Ivy Bridge	λ (fps)	82	62	72	114	132	16%	DP-MG
		λ/E (fps/J)	212	100	362	403	523	30%	DP-MG
	Haswell	λ (fps)	95	64	93	147	170	15%	DP-MG
		λ/E (fps/J)	182	79	673	499	673	34%	DP-CG (1)
Tracking	Ivy Bridge	λ(fps)	6.2	10	6.8	13	16	23%	CP-CG (4)
		λ/E (fps/J)	1.3	3.2	2.8	4.0	6.7	67%	CP-CG (4)
	Haswell	λ (fps)	6.3	11	9.2	13	19	46%	DP-CG (5)
		λ/E (fps/J)	1.1	2.8	4.0	3.7	8.4	127%	DP-CG (5)

chips evaluated, the transfer times between GPU and CPU are negligible. Thus, we did not use them in our model equations.

B. Baseline comparison and impact of adaptation

To assess the benefit of using our framework, we compare the pipeline configuration that our model predicts as best with the baseline configuration recommended by a previous work [13] that recommends a configuration based on the intuition that pipeline stages should be mapped to the device where they run more efficiently. This work also recommends exploiting parallelism using an approach similar to software pipelining where two frames are computed at the same time, one on the GPU and another one on the CPU. Therefore only MG granularity is exploited on the CPU cores by this baseline approach.

Table V shows the throughput in terms of frames per second, fps, and throughput/energy, fps/Joule, for homogenous runs, where only the CPU (with MG and CG granularities) or only the GPU is used: "CPU MG", "CPU CG" and "GPU"; and for two heterogenous executions, where CPU and GPU are used: "Baseline" that identifies the results of the aforementioned baseline configuration [13], and "Best" which correspond to the best configuration found by our framework. For both performance metrics, fps and fps/Joule, the higher the value, the better. The "Improv." column shows the percentage of improvement of "Best" with respect to "Baseline" (computed as (Best-Baseline)/Baseline). The last column shows the best pipeline configuration and the optimum number of threads, between parenthesis, for the CG cases.

The table shows that the best configuration obtained using our model significantly outperforms the baseline, specially when energy is also considered. These data show that the intuition can result in the selection of a suboptimal configuration, whereas the model can evaluate all configurations and select the best. Also, the baseline only considers "CP-MG"-like mappings, whereas the model considers more alternatives. As the table shows, in 10 out of 16 cases, the best configuration is not CP-MG. In 6 cases (all appear in ViVid) the baseline uses the same mapping as the best (Stage 1 is mapped on the GPU). In these 6 cases, λ and λ/E of baseline and best differ because in the baseline the stages mapped to the GPU can only run in the GPU (Stage 1 only runs in the GPU), while in our implementation the stages mapped to the GPU can also execute on the CPU (Stage 1 runs on both GPU and CPU). Notice that even if we only consider CP-MG mappings, the approach we use as baseline may not find the best mapping of stages to CPU and GPU. This is the case in SRAD and Tracking. For instance, for Tracking the best CP-MG mapping would be to map filters 1 and 3 to the GPU, whereas the baseline approach would map filters 1 and 2 to the GPU. In addition, as the number of possible configurations increases, relying on the intuition to find the best one becomes increasingly difficult. In this situation, our model can reduce the number of runtime tests that are needed to determine what is the best configuration.

The table shows that, overall, throughput improvement ranges from 5% to 46% (20% on average), whereas the improvement in throughput/energy ranges from 12% to 127% (43% on average). Energy improvement

ranges from 1% to 55% (18% on average). Interestingly, for ViVid on Ivy Bridge, the best pipeline configuration depends on the resolution. CP-CG is the best configuration for LD, while CP-MG is the best for HD. Also, the best configuration can change based on whether the metric to be optimized is λ or λ/E. For instance, SRAD on Haswell obtains maximum throughput with a DP-MG configuration, whereas the maximum throughput/energy is obtained using DP-CG with a single thread. We will discuss each benchmark in more detail in the next subsection.

The previous work that we have considered as a baseline can not adapt to changes in the input stream. We experimented with changes in the video stream feeding the ViVid application, from Low Definition to High Definition and viceversa. For instance, on Ivy Bridge, when changing from LD to HD, a change in the pipeline configuration from CP-CG to CP-MG results in an improvement of 81% in λ (204% in λ/E). Also, when changing from HD to LD, reconfiguring the pipeline from CP-MG back to CP-CG results in 30% improvement in λ (40% in λ/E). Since the training time for LD is 0.46 sec, and for HD is 6.47 sec, we can determine that the training is amortized (from the throughput point of view) when changes from LD to HD happen at most every 7.8 sec ($\sim$25 HD frames) and when changes from HD to LD happen at most every 0.85sec ($\sim$42 LD frames). Next we describe how these numbers have been computed:

1) Suppose the system is in the best configuration for LD (CP-CG with 5 threads). This configuration delivers 65fps. If the input changes to HD, there are two options:
   - Keep the same configuration (CP-CG, 5 threads). In this case, the new λ is 3.2 fps (0.31 sec/frame) with no sampling overhead.
   - Carry out the training procedure: 6.47 sec to conduct nC+3 experiments to process 17 frames (1+2+3+4+5+1(gpu)+1(MG) frames). The average throughput during sampling is 17/6.47=2.62fps (sub-optimal λ), but, in return, the model finds out that now the best configuration is CP-MG, that delivers 5.9 fps (81% better), i.e. 0.17sec/frame (wrt 0.31sec/frame, 0.14 fewers seconds per frame).

   To determine if the sampling was beneficial we need to find how many frames x, at 5.9fps, have to be processed to compensate the sampling of the 17 frames at 2.62fps, given that the system was processing frames with a throughput of 3.2 fps. By solving the equation $17/2.62 + x/5.9 = (x + 17)/3.2$, it can be found that $x = 8.22$ and that after processing 25.22 (17+8.22) frames, the sampling overhead has been amortized. This corresponds to 7.8 seconds (17/2.62+8.22/5.9).
2) Similarly, suppose the system is in the best configuration for HD (CP-MG). This configuration delivers a throughput os 5.9fps (CP-MG). If the input changes to LD, there are two options:
   - Keep the same configuration (CP-MG for LD). In this case, the new λ is 50 fps (0.02 sec/frame) with no sampling overhead.
   - Carry out the training procedure: 0.46 sec to conduct nC+3 experiments to process 17 frames (1+2+3+4+5+1(gpu)+1(MG) frames). The average throughput during sampling is 17/0.46=37fps (sub-optimal λ), but, in return, the model finds out that now the best configuration is CP-CG with 5 threads, that delivers 65 fps (30% better), i.e. 0.015sec/frame (wrt 0.02sec/frame, 0.05 fewer seconds per frame).

   In this case, we find out that x=25.8 frames, and that after processing 42.8 (17+25.8) frames, the sampling overhead has been amortized. This corresponds to 0.85 seconds (17/37+25.8/65).

Now, in hindsight it is possible to perform these computations because we know the time it took to perform the sampling and that the change will provide 81% improvement from LD to HD and 30% improvement from HD to LD. However, in a real scenario, the system will detect a throughput change and start the sampling procedure without knowing these numbers or even being positive of the improvement that can be obtained. Because of that, our system limits the frequency of the sampling based on the overhead the user is willing to pay. In the worst case, after paying the sampling overhead we may end up with no configuration change and therefore no extra improvement. In that case, eq. 2 provides the maximum frequency ($1/t$) at which sampling can be carried out for a given overhead ov_{thl} (Section IV-B).

Notice that the scene recognition benchmark was not used on this section, as only one pipeline mapping was possible for this benchmark. Nevertheless, it is used in the following section to validate the accuracy of the model.

C. Discussion of the results

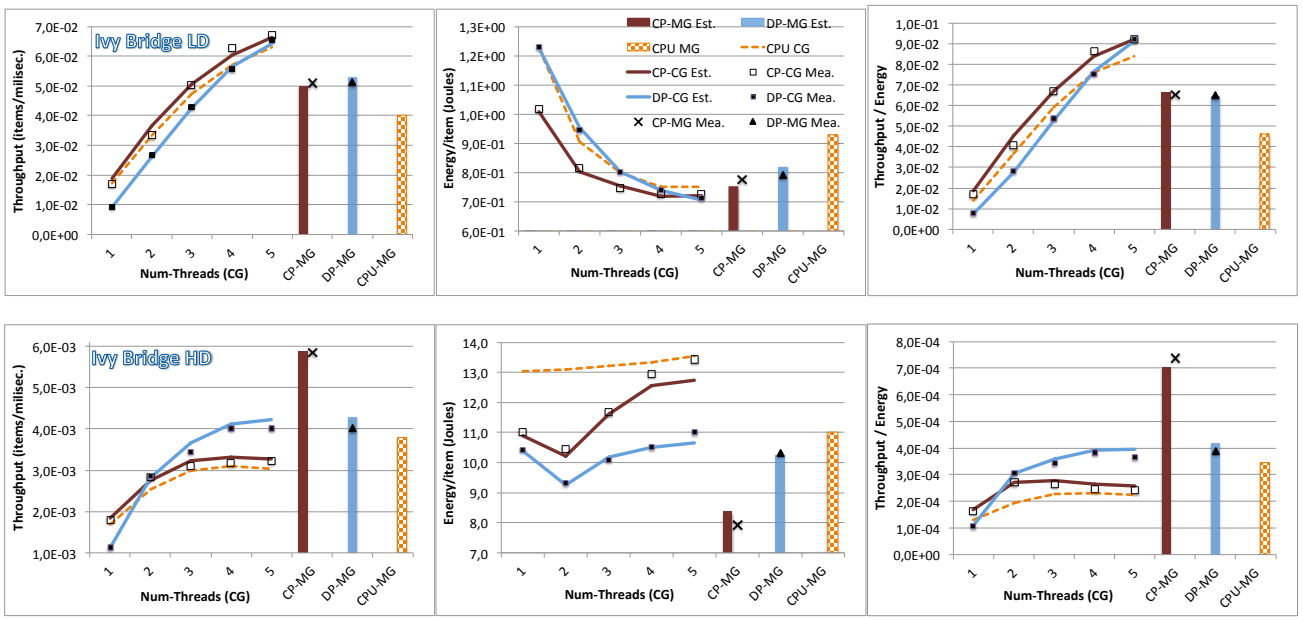
Figs. 11 to 15 show the results for all applications. In all of them we follow the same convention. On the left of each figure we see the CG evaluation (lines and marks) as the number of threads changes from 1 to 5, as shown on the x-axis. On the right of each figure we have the MG evaluation (three bars and two marks). The homogeneous CPU measurements collected in the training phase are represented by a dashed orange line for the CG execution (see Table I) and by a patterned orange bar for the MG execution (see Table II). Solid lines and bars represent model estimations for heterogeneous runs and marks represent experimental results. For CG granularities predictions we use solid lines: in light-blue for the DP-CG configuration and in dark-brown for the CP-CG one. The square marks are the measurements obtained for both CG mappings: solid for DP-CG and hollow for CP-CG. The solid bars represent the model prediction for MG granularities for heterogeneous runs: in light-blue for the DP-MG configuration and in dark-brown for the CP-MG one. The x marks are the experimental results obtained for the CP-MG configuration whereas the solid triangles are the results for the DP-MG one. By comparing the measurements with the model estimates we can asses the accuracy of the model.

We have experimentally assessed all the evaluated pipeline configurations (48, 384, 48, and 5 for Vivid, SRAD, Tracking, and Scene Recognition, respectively). To facilitate readability, instead of cramming the results of all these experiments on a single chart, *amongst all the possible CP mappings, Figs. 11 to 15 only show the configuration that achieves the highest λ/E result*. As we discuss in the next sections, for all the applications and architectures studied, the estimations of the model reasonably match the measured metrics. For all the cases, the model needs less than 10 microseconds to instantiate the equations for all the possibilities and determine the optimal granularity, mapping and number of threads.

1) ViVid: This application was introduced in section I. ViVid is comprised of 5 stages, being the first and last ones the serial Input and Output, while the three middle ones are parallel. These middle stages are: i) Stage 1 that finds the maximum response of 100 filters; ii) Stage 2 that summarizes the low level information collected by the previous stage; and iii) Stage 3 that computes the actual detection step. In this benchmark, the MG results were obtained exploiting nested parallelism with TBB `parallel_for` in the stages executed in the multicore. Fig. 11 and Fig. 12 depict the estimated and measured Throughput (λ), Energy per item (E) and Throughput/Energy (λ/E) for LD and HD on Ivy Bridge and Haswell, respectively. Amongst all the CP mappings we only show the most performing one: when stage S_1 is the only one mapped on the GPU (as illustrated in Fig. 1 b) and d)), both for the CG and MG granularities.

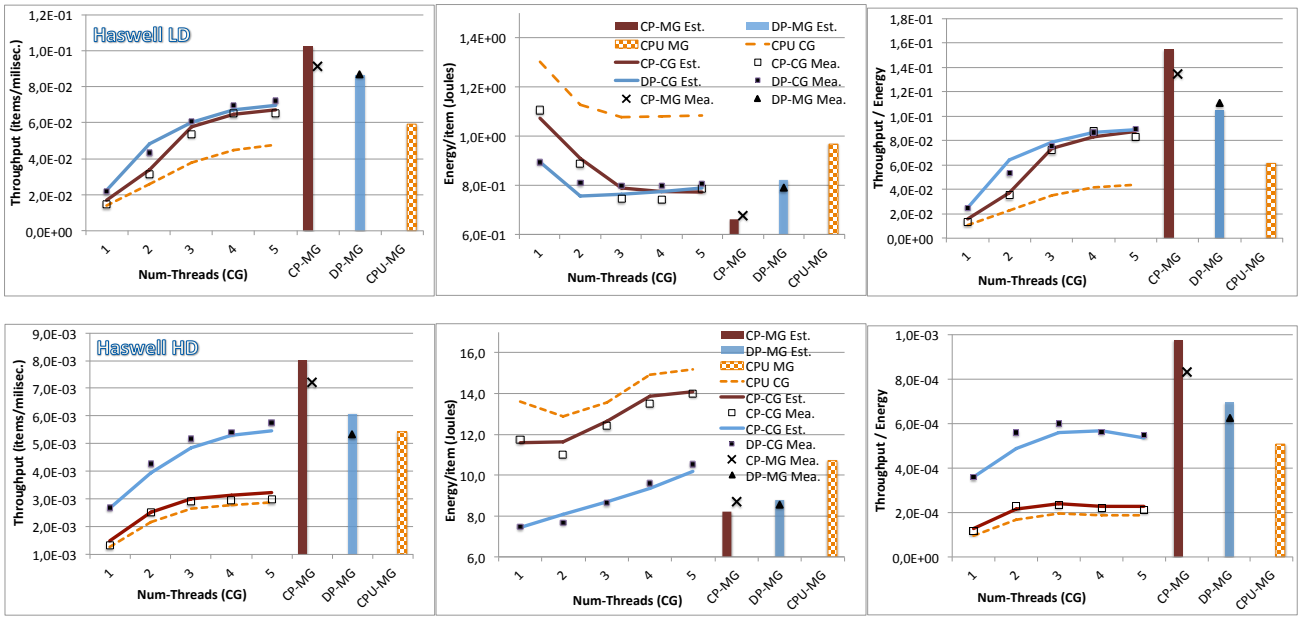
As Figs. 11 and 12 show, our model is able to give a good estimation of λ, E and λ/E. In general, it tends to slightly overestimate the throughput in the CP configurations, because for CP we always consider the ideal contribution of all the threads without considering the overheads. These overheads account for the synchronization costs of the GPU-CPU threads in the coupled GPU-CPU path, that we do not consider in our equations. The results show that our model fits the measured throughput reasonably well, specially on the Ivy Bridge architecture for which the estimated values are within 2% of the measured ones. On Haswell, our overestimation of the throughput is within 9%. Regarding the energy results, we can also see that, in general, our equations tend to slightly underestimate the energy, although deviation is always within 5% of the measured values. The deviation is more noticeable for Haswell, where our model predicts that CP-MG is better than DM-MG, though measures tell the contrary. Anyway, this imprecision is not significative because the differences between CP-MG and DP-MG are small, so there is not a big penalty to be paid by this error. In any case, the best configuration for energy optimization is DP-CG with 1 thread that our model correctly predicts. In general, these results validate our initial assumption when deriving the simplified model for the energy consumption, which we introduced in section V. Also, we can mention that the accuracy of the predicted values for our other metric of interest, Throughput/Energy (λ/E), are within -5% to 10% with respect to the measured values.

Overall, and despite these small inaccuracies, the model successfully predicted the best pipeline configuration (granularity, mapping) as well as the appropriate number of threads, for each type of input and architecture. On Ivy Bridge, for LD videos the optimal is found with the CP-CG configuration and 5 threads (although DP-CG is very close), whereas for the HD input the optimal is provided by the CP-MG configuration. However, on Haswell, the best option for LD and HD is always CP-MG.



(a) Throughput (higher the better) (b) Energy per item (lower the better) (c) Thr./energy (higher the better)

Fig. 11: Performance metrics for ViVid when processing LD (up) or HD (bottom) video on Ivy Bridge. Solid lines/bars represent model predictions. Marks are the experimental results.

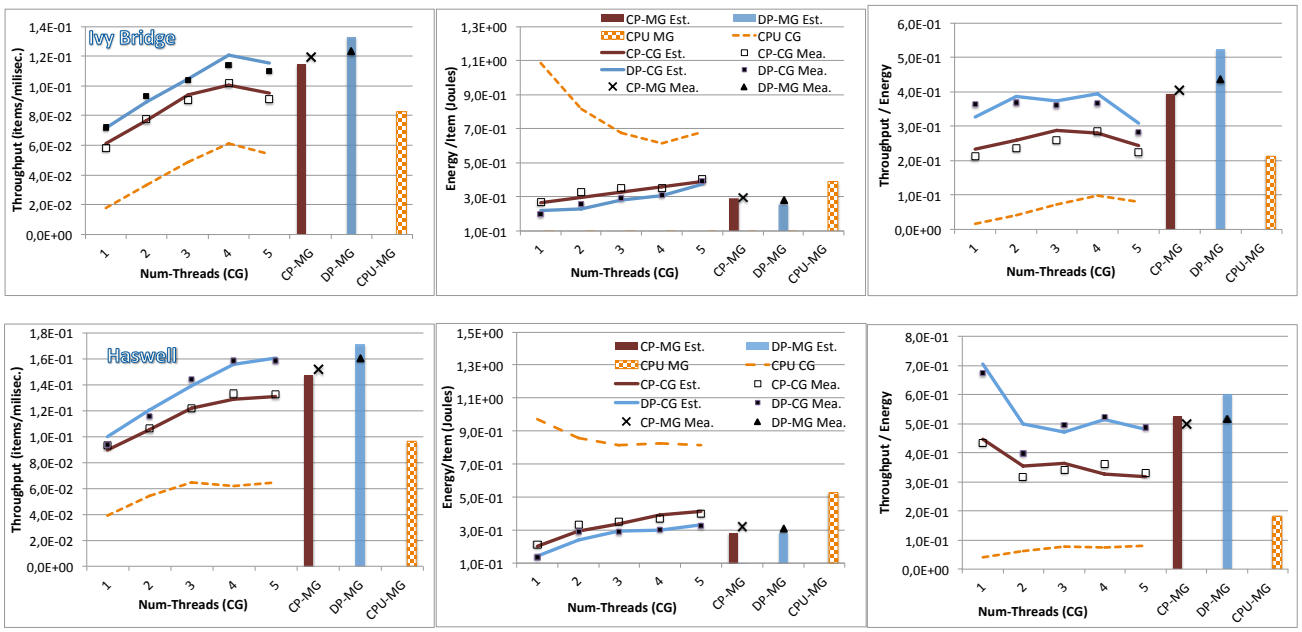


(a) Throughput (higher the better) (b) Energy per item (lower the better) (c) Thr./energy (higher the better)

Fig. 12: Performance metrics for ViVid when processing LD (up) or HD (bottom) video on Haswell.

The figures also show an important result: a configuration with a higher throughput can consume more energy than a lower throughput one. This can be observed on Haswell, in Figs. 12a and 12b, for the HD input and the CP-CG configuration, for which the highest throughput is obtained with $n = 5$ threads. However, the energy consumption is also higher for that number of threads. In fact, for this configuration the optimal λ/E for HD is found for $n = 3$, solution that our model correctly predicts as we see in Fig. 12c.

2) SRAD: The SRAD (Speckle Reducing Anisotropic Diffusion) benchmark is part of the Rodinia benchmark suite [1]. This benchmark implements a diffusion method for ultrasonic and radar imaging applications



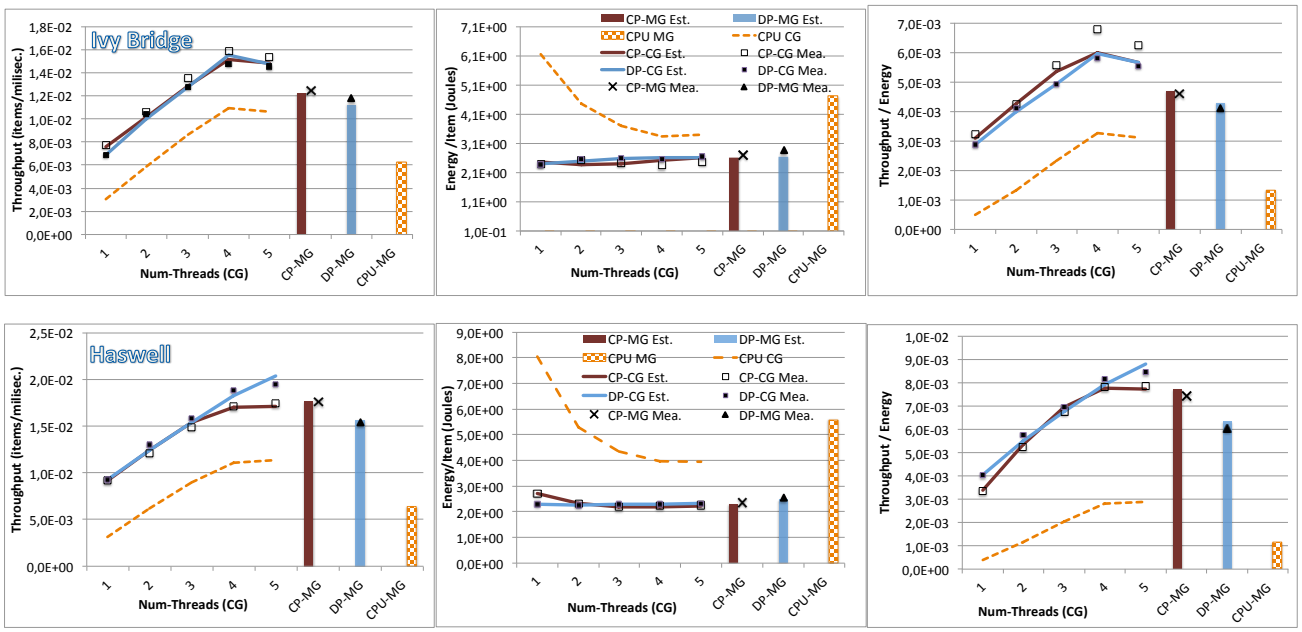
(a) Throughput (higher the better) (b) Energy per item (lower the better) (c) Thr./energy (higher the better)

Fig. 13: λ, E and λ/E for SRAD on Ivy Bridge (up) and Haswell (bottom).

based on partial differential equations (PDEs) [24]. SRAD has 8 pipeline stages: serial Input, image extraction, preparation, reduction, statistics, computation 1 and computation 2, image compression and serial Output. In our experiments we ran these stages over a stream of images (200). Each stage can implement a CG or MG granularity (in this benchmark, the MG granularity was exploited using OpenCL in the stages executed in the multicore) so there are $2^6 \cdot (4+2) = 384$ pipeline alternatives. However, just instantiating our model equations for all these possibilities we can find out the best configuration on each architecture. Figure 13 shows all metrics for Ivy Bridge and Haswell. From all the CP mappings we only show the most efficient one, that happens to be when the GPU is mapped on all but the second stage, for both the CP-CG and CP-MG configurations, in both machines.

Results in Fig. 13 show that for SRAD our model also provides a reasonable estimation of the throughput and energy on each pipeline alternative. For this application, our equations tend to overestimate the throughput of the DP configurations, especially for the DP-MG configuration, where 7% of deviation over the measured throughput was found. Also, as pointed out for ViVid, a slight underestimation of the energy consumption was registered, in this case always below 8%. These inaccuracies are the reason of the 16% of overestimation for λ/E in the DP-MG configuration. All in all, our model correctly predicts that the optimal configuration for Ivy Bridge is DP-MG, whereas for Haswell is DP-CG with $n = 1$ if we optimize λ/E. Notice, that DP-CG with $n = 1$ implies that the only thread is the GPU one, which corresponds to an homogeneous execution on the GPU. This is another example of a case in which the highest throughput did not result in the lowest energy. For instance, here we find the maximum λ with DP-CG for $n = 5$ (see Fig. 13a for Haswell). However, since the minimum energy consumption is found for $n = 1$ (Figure 13b), the optimal λ/E is also for DP-CG $n = 1$ (Fig. 13c for Haswell). Our model correctly captures this fact.

3) Tracking: Tracking calculates the movement of a set of features over the image-flow of a video stream. The implementation is based on the Kanade Lucas Tomasi (KLT) [22] algorithm of the San Diego Visual Benchmark Suite [12]. This implementation comprises three phases: image processing, feature extraction and feature tracking. The pipeline of this application has 5-stages: the first and fifth ones are Input and Output, whereas the middle ones are parallel and can be mapped on both CPU and GPU. Each parallel stage can implement a CG or MG granularity. In this benchmark, the MG granularity was exploited using TBB `parallel_for` in the stage executed on the multicore. So there are $2^3 \cdot (4+2) = 48$ pipeline alternatives. Again, from all the CP mappings we only show the most efficient one: stages 1 and 3 on the GPU for both



(a) Throughput (higher the better) (b) Energy per item (lower the better) (c) Thr./energy (higher the better)

Fig. 14: λ, E and λ/E for Tracking on Ivy Bridge (up) and Haswell (bottom).

the CP-CG and CP-MG configurations for Ivy Bridge and Haswell. In the experiments for tracking, we have used a video stream with 200 frames (1080x1920).

Fig. 14 shows the computed and estimated λ, E and λ/E on Ivy Bridge and Haswell. The model's deviation for both platforms is always below 5%, 7% and 11% of measured throughput, energy and throughput/energy, respectively. Again, the model predictions are accurate enough to guess that the appropriate configuration is CP-CG with 4 threads with the GPU used on stages 1 and 3 for Ivy Bridge and DP-CG with 5 threads for Haswell.

4) Scene Recognition: This application performs generic visual categorization, ie., it identifies the object content of natural images while generalizing across variations inherent to the object class (view, imaging, lighting, occlusion, etc). This code is based in the algorithm proposed in [23]. The code is implemented as a 4-stages pipeline. The first and last stages take care of the sequential Input, and Output. The two middle stages are parallel. The input to this code are 200 images of 256×256 pixels from a database containing images from 8 different classes (forest, street, coast, etc). In this benchmark, only the CP-CG configuration is feasible. DP mapping is not an option because the branchy nature of the second parallel stage makes it not suitable for the GPU. Besides, MG granularity does not scale. The first parallel stage can execute on both the CPU and GPU. However, this stage just represents the 18% of the pipeline execution (on both architectures).

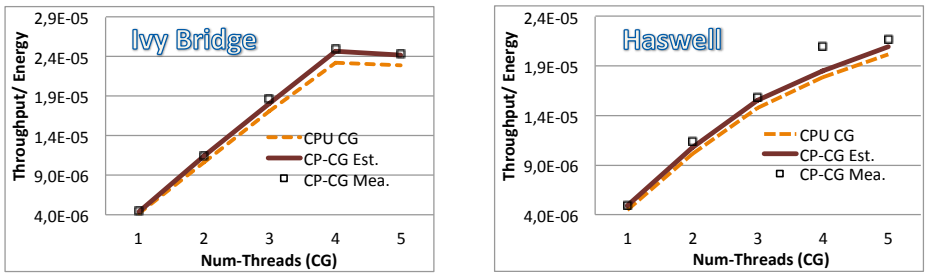


Fig. 15: λ/E for Scene Recognition (higher the better).

Fig. 15 shows the computed and estimated λ/E for the CP-CG configuration when executing the application with 1 to 5 threads on Ivy Bridge and Haswell. Our model accurately predicts the measured values. The

higher difference between predicted and measured values is found in Haswell with 4 threads. In this case, λ is understimated 7% whereas the energy is overestimated 5.5% which turns into 11% of underestimation for λ/E. We can note that the improvement of the CP-CG execution with respect to the homogeneous multicore is small, as we are just affecting the 18% of the application, being the improvement factors (the ratio of the stage's throughput/energy on the CPU vs the GPU) also small. Anyway, one interesting finding is that the Ivy Bridge CP version reaches the point of diminishing returns with 4 threads (although the throughput is slightly higher with 5 threads, the energy is also higher). On Haswell, the optimal solution is for $n = 5$, and our model finds it.

D. Lessons learned

One relevant result of our model is that it helps us to identify the appropriate granularity for each problem. In the quest of choosing the right granularity for each problem, we have found one important piece of experimental evidence that helps us to understand how the granularities affect performance: the throughput and energy values reported by the multicore homogeneous execution for the two type of granularities (in the figures, the dashed orange line for the CG execution –CPU-CG – and the patterned orange bar for the MG execution–CPU-MG–) are key to predict when one type of granularity will perform better than the other. For instance, for ViVid on Ivy Bridge-LD (see Fig. 11), the CPU-MG throughput (and energy) is outperformed by the CPU-CG one when $n > 3$. However, on ViVid for Ivy Bridge-HD, the CPU-MG throughput (and energy) outperforms the CPU-CG for any n. Therefore, configurations that exploit CG granularity seem more suitable for LD, while configurations exploiting MG will be the ones for HD, as finally the heterogeneous executions prove. This is also valid for all the other benchmarks and architectures, as we can also see in Table V by comparing the "CPU MG" and "CPU CG" columns of the homogeneous results: if λ or λ/E is larger for "CPU MG" than for "CPU CG", then the recommended granularity for the best configuration (see "Best conf." column in that table) is MG, and viceversa.

In addition to granularity, the mappings also play an important role. DP mappings work well if the GPU thread obtains better values for the metric of interest in all the stages of the pipeline than a CPU thread for CG granularities (or better efficiencies in all stages than nC threads for MG granularities). If this is not the case, then CP can potentially exploit better the heterogeneity of the system, as long as the CP mapping ensures that the pipeline stages are mapped to the device where they execute most efficiently.

Other interesting result is that higher throughput does not always imply a lower energy consumption. This is most noticeable in the CG plots of previous figures, mainly for SRAD and ViVid HD. For example, in Fig. 13, the CG configurations for SRAD have the highest throughput for 4-5 threads, whereas the minimum energy consumption is achieved for one thread.

As summary, we have discussed some of the main trade-offs that affect throughput and energy in the DP and CP mappings under different granularities, and how our reasonable simple model is able to correctly predict all these trade-offs.

VII. Related works

One approach for coding streaming applications is to use a programming language with support for streams, as for example StreamIt [25]. But currently these approaches do not provide support for heterogenous CPU-GPU executions. By using both CPU cores and GPUs, simultaneous computation on heterogenous platforms delivers higher performance than CPU-only or GPU-only executions [26]. However, programming frameworks that provide support for computing in heterogeneous architecture such as Qilin [2], OmpSs [3], XKaapi [4] or StarPU [5] 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 center on streaming applications.The work by Totoni et al. [13] is perhaps the closest to ours. In this paper, we propose several other pipeline configurations they do not consider. In Section VI-B we have used their approach as a baseline and compared with the configuration that our model finds to be the best.

Research on power and energy aware heterogeneous computing started to draw attention several years ago. Most works try to model power or energy specifically for the GPU: some works analytically model GPU power with architecture level instructions [27], [28], or hardware performance events [29], [30]. However, they model

the execution of applications in a GPU or a cluster of GPUs, without considering the simultaneous execution on the CPU multicores, which is central in our approach. Those works try to model the power consumption of specific GPU micro-architecture components (such as global memory accesses, texture cache accesses, bank conflicts, etc.) to identify the power bottlenecks in a kernel and suggest power aware optimization strategies. We are concerned in how the different computational resources (CPU cores and GPUs) interact when working in parallel and how to dynamically select energy and performance aware mapping configurations in streaming applications.

There have been other research efforts, such as [31], [32], that have tried to define analytical models to optimize the scheduling of pipeline applications, considering energy and throughput as an objetive or a constraint of the problem. However, these works focus on optimizing the concurrent execution of multi-programed workloads that consist of independent pipeline applications, whereas we are interested in optimizing single streaming execution. In addition, they model energy as a sum of system level components (processor, network, disk, ...) where the energy consumed on each component is the product of the execution time in that component and the dynamic power in the component (measured or estimated using microbenchmarks and supposed constant for the benchmarks evaluated). Our approach, on the other hand, uses the accurate hardware energy counters available on the architectures we study, that allow us to measure at runtime the consumption on the CPU, GPU and Uncore components of the system. In contrast with previous static approaches, we use this information to guide the scheduler at runtime to find the optimal granularity, mapping and number of threads that optimize the throughput or the energy (or a trade-off metric) of our application.

VIII. Conclusions

To the best of our knowledge, this is the first work proposing an analytical model that can be used to efficiently map the different stages of a pipeline application onto an heterogeneous chip (integrated CPU-GPU processor). The model can use throughput, energy, or a tradeoff such as throughput/energy to predict the best pipeline setting. The model was validated with four applications, finding that the accuracy of our estimations are within 2% to 16%, that suffices to find out the optimal pipeline configuration.

We have also compared the best configuration predicted by the model with a state of the art approach. Our results show that the configurations selected by the model produce, on the average, 20% higher λ and 43% higher λ/E. We have measured improvements in λ and λ/E of up-to 82% and 204%, respectively, when the model is used to adapt to an input video that changes its resolution. Our framework guarantees that the runtime overhead due to the training required to adapt to a changing input is always kept below a user-defined limit.

References

[1] S. Che *et al.*, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *IISWC*, 2010, pp. 1–11.

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

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

[4] T. Gautier, J. Lima, N. Maillard, and B. Raffin, "XKaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *IPDPS 2013*, 2013, pp. 1299–1308.

[5] 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.

[6] J. Cong and B. Yuan, "Energy-efficient scheduling on heterogeneous multi-core architectures," in *ACM/IEEE Intl. Symp. on low power electronics and design*, 2012, pp. 345–350.

[7] J. Clemons, H. Zhu, S. Savarese, and T. Austin, "Mevbench: A mobile computer vision benchmarking suite," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, Nov 2011, pp. 91–102.

[8] L. De Cicco, S. Mascolo, and V. Palmisano, "Skype video responsiveness to bandwidth variations," in *18th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2008, pp. 81–86.

[9] J. Fernandez-Madrigal, E. Cruz-Martin, A. Cruz-Martin, J. Gonzalez, and C. Galindo, "Adaptable web interfaces for networked robots," in *Intelligent Robots and Systems, (IROS 2005)*, Aug 2005, pp. 3441–3446.

[10] M. Dikmen, D. Hoiem, and T. S. Huang, "A data driven method for feature transformation," in *Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2012, pp. 3314–3321.

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

[12] S. K. Venkata and et al., "SD-VBS: The San Diego Vision Benchmark Suite," in *IISWC*, 2009, pp. 55–64.

[13] E. Totoni *et al.*, "Easy, fast and energy efficient object detection on heterogeneous on-chip architectures," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, 2013.

[14] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *ISPLED*, 2010.
[15] R. Dementiev, T. Willhalm, O. Bruggeman, P. Fay, P. Ungerer, A. Ott, P. Lu, J. Harris, P. Kerly, and P. Konsor, *Intel Performance Counter Monitor*, 2012. [Online]. Available: www.intel.com/software/pcm
[16] H. McCraw, J. Ralph, A. Danalis, and J. Dongarra, "Power monitoring with PAPI for extreme scale architectures and dataflow-based programming models," in *Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications, IEEE Cluster 2014*, sep 2014.
[17] "Odroid-XU3," Aug. 2014, http://www.hardkernel.com/main/.
[18] W. J. Gordon and G. F. Newell, "Closed queuing systems with exponential servers," *Operations Research*, vol. 15, no. 2, pp. 254–265, 1967.
[19] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism," in *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, Sept 2009, pp. 281–290.
[20] S. K. Bose, "Open and closed networks of M/M/m type queues," Indian Inst. of Technology Guwahati, Tech. Rep., 2002.
[21] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.
[22] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *IJCAI*, 1981, pp. 674–679.
[23] G. Csurka *et al.*, "Visual categorization with bags of keypoints," in *ECCV*, 2004, pp. 1–22.
[24] Y. Yu and S. Acton, "Speckle reducing anisotropic diffusion," *Image Processing, IEEE Tran. on*, vol. 11, no. 11, pp. 1260–1270, 2002.
[25] 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*, Jun. 2013.
[26] C. Yang *et al.*, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *CLUSTER*, 2010, pp. 19–28.
[27] J. Pool, A. Lastra, and M. Singh, "An energy model for graphics processing units," in *ICCD*, 2010, pp. 409–416.
[28] J. Leng *et al.*, "GPUWattch: Enabling energy optimizations in GPGPUs," in *ISCA*, 2013, pp. 487–498.
[29] K. Kasichayanula *et al.*, "Power aware computing on GPUs," in *SAAHPC*, 2012, pp. 64–73.
[30] S. Song, C. Su, B. Rountree, and K. Cameron, "A simplified and accurate model of power-performance efficiency on emergent GPU architectures," in *IPDPS*, 2013, pp. 673–686.
[31] A. Benoit, P. Renaud-Goud, and Y. Robert, "Performance and energy optimization of concurrent pipelined applications," in *IPDPS*, 2010, pp. 1–12.
[32] C.-S. Lin *et al.*, "Multi-objective exploitation of pipeline parallelism using clustering, replication and duplication in embedded multi-core systems," *Journal of Systems Architecture*, vol. 59, no. 10, Part C, pp. 1083 – 1094, 2013.