Heterogeneous Cloud Computing: Design Methodology to Combine Hardware Accelerators

Bruno da Silva[1,2,3], Jan G. Cornelis[2], An Braeken[1], Erik H. D'Hollander[3], Jan Lemeire[1,2] and Abdellah Touhafi[1,2]
[1]*Industrial Department (INDI), Vrije Universiteit Brussel, Brussel, Belgium*
[2]*Informatics and Electronics Department (ETRO), Vrije Universiteit Brussel, Brussel, Belgium*
[3] *Electronics and Information Systems Department (ELIS), Ghent University Ghent, Belgium*

***Abstract*—Heterogeneous cloud computing servers provide access to different types of hardware accelerators in order to satisfy the computational demands that standalone general-purpose multi-processors can not deliver. The combination of different technologies provides more opportunities to accelerate the most compute-intensive applications by exploiting the key features of each type of hardware accelerator. Unfortunately, the design effort drastically increases when considering complex applications. We propose a methodology which provides an early speedup prediction when combining hardware accelerators and insights about potential performance leaks. In addition, we consider a couple of metrics to analize the computational use of the accelerators and the exploitation of the heterogeneous system. Our methodology is applied to a real-time surveillance system, which is composed of Field-Programmable Gate Arrays (FPGA) and Graphic-Processing Units (GPU). The methodology is used to provide an early speedup prediction, which is used to guide the combination of both accelerators. The achieved acceleration is only a 3.5% lower than estimated by our methodology.**

***Index Terms*—Heterogeneous Cloud Computing, High-Performance Computing, Hardware Accelerators, GPU, FPGA, Speedup**

I. Introduction

Heterogeneous cloud computing has evolved rapidly in the recent past, mostly due to applications in the fields of Big Data Analytics (BDA) or Artificial Intelligence (AI), which demand more computational power than multi-processors can deliver. Field-Programmable Gate Arrays (FPGAs), Graphic-Processing Units (GPUs), and even Application-Specific Integrated Circuits (ASICs), are different types of hardware accelerators currently available in cloud computing servers. GPUs have become successful thanks to their massive parallelism, unified design and programming model. These accelerators are most appropriate for vector calculations in general, but are less suited for irregular calculations. FPGAs are a different kind of hardware accelerators that provide a programmable and massively parallel architecture. They have an open architecture that is modelled by the control path and data path of the algorithm. The scope of applications that can be accelerated is enlarged when combining the power of GPUs with the flexibility of FPGAs. Heterogeneous clusters are presented in [1], [2] while the authors of [3] and [4] propose a hybrid desktop. These hardware accelerators have been recently included in the catalog of the major cloud services providers. For instance, Microsoft Azure offers FPGAs-in-the-cloud [8], which communicate to their hosts through dedicated Peripheral Component Interconnect Express ($PCIe$) interface. Google Cloud [5] offers Tensor Processing Unit (TPU), an ASIC designed from the ground up for machine learning, to accelerate deep learning models on their cloud computing services. Similarly, cloud services provided by Alibaba or Amazon, which originally only offered GPUs for their heterogeneous cloud computing services, have been recently extended to include FPGAs in their catalogs. Thus, Amazon [6], [7], Microsoft [8] or Alibaba [9] provide heterogeneous cloud computing servers, composed of different types of accelerators, to execute the most compute-intensive applications. The introduction of OpenCL as a common programming language for heterogeneous computing servers composed of FPGA and GPU accelerators, has increased the portability of the code between hardware accelerators. The performance degradation is, however, not negligible, demanding a design exploration before combining accelerators [10]. Thus, the selection of the most suitable technology to accelerate a particular application is left to the end-user. The designer must know the computational power of the accelerator, its key features, what type of I/O is used as interface and its available BW to make the best selection. This design effort drastically increases when searching for the most performance-efficient combination of different hardware accelerators. The achieved acceleration, when compared to general-purpose processors, might not be obtained due to a different accelerator or achieved as a result of the proper task scheduling between accelerators.

In this paper, we propose a design methodology to properly exploit the combination of different types of hardware accelerators. The use of different hardware accelerators demands insight knowledge about the key features, guiding the selection of the appropriated accelerator for the task's characteristics and demands. For instance, FPGAs are more suitable for low-latency demanding applications, while applications allowing data-independent parallel execution would benefit from GPUs' characteristics. Authors in [11] propose the use of *idioms*, which a re patterns of computation and memory access, to identify the most suitable hardware accelerator where to compute a certain algorithm. Our approach considers combination of hardware accelerators and bases the performance expectations on application's profiles in order to provide realistic estimations. Moreover, the proper task scheduling is critical due to the communication overhead as a result of the combination of hardware accelerators. This overhead might strongly decrease the speedup, becoming the main cause of performance degradation when scaling the tasks' execution. Some authors, like in [12] or in [13], have proposed extensions of the Amdahls Law for heterogeneous computing. Despite this simple analytical model provides insights about the achievable speedup when parallelizing the execution of an algorithm, it must be adapted to heterogeneous computing by considering the achievable acceleration of each different technology and the impact of the tasks scheduling. Our methodology does not only consider the impact of the additional communication when combining accelerators, but also provides insight into how efficiently are the accelerators used.

The main contributions of this work can be summarized as follows:

- A methodology to identify achievable performance when combining accelerators.
- The identification of performance boundaries.
- Metrics to evaluate the occupancy level of the accelerators and the efficiency of the system.

The paper is organized as follows. Different heterogeneous cloud computing solutions have been proposed combining different hardware accelerators. In Section II the proposed methodology is introduced. The metrics used to determine parameters like the accelerator's occupancy or the overall efficiency of the systems are explained. The performance bounds and the factors related to performance degradation are also identified. An explanation about how the methodology can be applied to a heterogeneous platform is given in Section III. A surveillance system is used as case study to demonstrate how our methodology can be applied. Finally, conclusions are drawn in Section IV.

II. METHODOLOGY

Combining different hardware accelerators on a heterogeneous platform is challenging. The performance benefits of this combination must go beyond exploiting the key features of each standalone accelerator. On the one hand, a speedup is expected when running a compute-intensive task on a different type of hardware accelerator. The use of different hardware accelerators provides more opportunities for acceleration by exploiting the accelerators based on their characteristics. On the other hand, the performance increases due to the use of an additional accelerator. Nonetheless, this acceleration is strongly linked to an adequate scheduling and balance of the tasks executed on each accelerator. The overall performance results from both speedups. Although the presented methodology can be generalized for n accelerators, from here on we only refer to two types of hardware accelerators for the sake of simplicity.

A. Finding a candidate

The combination of several types of hardware accelerators means that at least one task is executed on each accelerator. However, this combination demands a significant effort reallocating the computational tasks. A first step is a profiling of the application to identify the most compute intensive tasks, which are usually the best candidates to be accelerated. The second step is an evaluation of the level of concurrency of the tasks since a certain speedup can be obtained if the application allows concurrent execution of the tasks. Once the compute intensive candidates are identified, a preliminary performance estimation must be done in order to avoid unexpected degradation.

An overlap speedup ($S_{overlap}$) is the acceleration resulting from concurrency when the execution of the tasks are completely overlapped on the accelerators. It demands a perfect balance on the scheduling of the tasks between accelerators exists, beyond the achieved speedup of the candidate task on the accelerator. Eq. 1 defines the achieved acceleration due to executing a candidate task of the application in another accelerator.

$$S_{overlap}(task) = \frac{t_{exec}^{acc1}}{max(t_{exec}^{acc1} - t^{acc1}(task), t^{acc1}(task))} \tag{1}$$

where t_{exec}^{acc1} is the execution time of the application and $t^{acc1}(task)$ is the time demanded by the *candidate* task.

The information obtained by profiling provides an idea about the complexity of the application and insights about achievable acceleration. For instance, let's consider an application composed of 2 main tasks (A and B). If we consider the tasks A and B as the loop-body of a loop, with independent iterations and without data dependencies within an iteration, both tasks can be executed concurrently. We can distinguish two potential accelerations:

- If $t^{acc1}(A) >= t^{acc1}(B)$, any acceleration of task B would have no significant impact. The unique acceleration results from the overlap speedup of both accelerators. Benefit is due to concurrency.
- If $t^{acc1}(A) < t^{acc1}(B)$, task B is an interesting candidate since its acceleration will report benefits together with the overlap speedup of both accelerators, thanks to concurrency and to the speedup running on the second accelerator.

Considering the case where task B is the most intensive one, and therefore the candidate to be accelerated, Eq. 1 results:

$$S_{overlap}(B) = \frac{t^{acc1}(A) + t^{acc1}(B)}{t^{acc1}(B)} \tag{2}$$

B. Accelerating the candidate

Heterogeneous computing provides more opportunities to accelerate the execution of our *candidate*. The attainable speedup of a task depends directly on the hardware features and the adequacy of the algorithm to the architecture. The selection of the most appropriate type of tasks for a given hardware accelerator is not a trivial task. Several authors have proposed algorithm classifications, based on patterns of computations and memory access [14], [15], which facilitate the identification of the most suitable hardware accelerator. Alternatively, existing performance models, such as the roofline model [16], can be used to predict the peak performance for different technologies. For the particular case of the FPGAs, a new class of programming tool called High-Level Synthesis (HLS) tools can help to estimate performance [17]. Most of the HLS tools provide reports with estimations of the FPGA resource utilization, latency, and throughput of the resulting RTL module. This information allows a fast design space exploration and an early performance estimation or even performance prediction [18]. Both features drastically reduce the migration effort as will be seen in our case study.

Independently of the selection method, the overall impact of accelerating a task is limited. For instance, when the execution time of the task on the accelerator is lower than the remaining computations on the original accelerator, the last one dominates the total execution time.

$$S_{th}(task) = \frac{t_{exec}^{acc1}}{t_{exec}^{acc1} - t^{acc1}(task)} \tag{3}$$

For our previous example, this will occur when $t^{acc1}(A) > t^{acc2}(B)$, where $t^{acc2}(B)$ is the execution time of task B on a second accelerator. This is the basis of Amdahl's law [12], the sequential part dominates the overall time execution and the achievable speedup. For the previous case, the speedup of the application is given by:

$$S_{th}(B) = \frac{t^{acc1}(A) + t^{acc1}(B)}{t^{acc1}(A)} \tag{4}$$

This theoretical speedup is only reached when the execution part that has not been accelerated dominates the overall execution time. Notice that any further acceleration of task B will have no impact when $t^{acc1}(A) > t^{acc2}(B)$. Therefore, a maximum speedup of task B exists above which any further acceleration has no impact. This limit can be obtained applying Eq. 5:

$$S_{ideal}^{task} = \frac{t^{acc1}(task)}{t_{exec}^{acc1} - t^{acc1}(task)} \tag{5}$$

when assuming independent tasks. Considering the previous example, the timing of task B running on the second accelerator ($t^{acc2}(B)$) must equal $t^{acc1}(A)$. In such a case, the achievable speedup by accelerating task B becomes:

$$S_{ideal}^{B} = \frac{t^{acc1}(B)}{t^{acc1}(A)} = \frac{t^{acc1}(B)}{t^{acc2}(B)} \tag{6}$$

Any further acceleration of task B will be hidden by the execution of task A, which becomes the dominated one. The contribution of the task's acceleration and the concurrency speedup is given by:

$$S_{combined}(task) = \frac{t_{exec}^{acc1}}{max(t_{exec}^{acc1} - t^{acc1}(task), t^{acc2}(task))} \tag{7}$$

where $S_{combined}(task)$ is the overall acceleration due to concurrency and the task's acceleration. For the previous example, it becomes:

$$S_{combined}(B) = \frac{t^{acc1}(A) + t^{acc1}(B)}{max(t^{acc1}(A), t^{acc2}(B))} \tag{8}$$

C. Communication

Despite the performance increases due to the additional hardware accelerator, communication is a critical factor. When splitting an application, all the function context must be transferred in runtime, and probably adapted at design time, for another architecture. Parameters like the amount of data to be transferred and available BW of the I/O interface must be taken in consideration. The communication overhead ($C_{overhead}$) might be so high that it reduces any potential speedup. The total time exclusively dedicated to communicate decreases the final performance and must be subtracted from the achievable speedup.

$$C_{overhead} = \frac{t_{comm}}{t_{exec}^{acc1}} \tag{9}$$

where t_{comm} is the additional time exclusively dedicated to communicate with the new accelerator. During t_{comm} there is no overlap with any computation.

The performance degradation due to communication impacts on $S_{overlap}(task)$ and $S_{combined}(task)$, and can be defined as:

$$D_{overlap}(task) = \frac{max(t_{exec}^{acc1} - t^{acc1}(task), t^{acc1}(task))}{max(t_{exec}^{acc1} - t^{acc1}(task), t^{acc1}(task)) + t_{comm}} \tag{10}$$

$$D_{combined}(task) = \frac{max(t_{exec}^{acc1} - t^{acc1}(task), t^{acc2}(task))}{max(t_{exec}^{acc1} - t^{acc1}(task), t^{acc2}(task)) + t_{comm}} \tag{11}$$

where $D_{overlap}(task)$ and $D_{combined}(task)$ corresponds to $S_{overlap}(task)$ and $S_{combined}(task)$ respectively.

D. Schedule

The communication has a direct dependency with the schedule of the tasks on the accelerators. If both accelerators are well-balanced it is possible to hide the communication between accelerators. In order to measure how efficiently the heterogeneous platform is used, we introduce the concepts of *occupancy* and *efficiency*. The total occupancy of the accelerators, together with the transfer time, provide useful information about the level of the balancing of the tasks. The occupancy of the accelerator reflects what percentage of the runtime the accelerator is computing:

$$Occupancy_i = \frac{t_{comp}^{acci}}{t_{exec}} \times 100 \tag{12}$$

The occupancy of the accelerators together with the communication time provide an idea about the level of parallelization.

- If the sum of the occupancy of all accelerators and the communication cost is 100%, there is no overlap. Therefore, the effort should be dedicated to balance the tasks.
- If the occupancy of one accelerator is significantly low, and the data transfer is not high enough to justify it, that could be a symptom that a semi-complete balance has been achieved between accelerators. In such a case, the *candidate* can be further accelerated.
- If the occupancy is high on both accelerators, the data transfer would be hidden by the computations. This is the goal of combining accelerators.

The efficiency of the system is defined in Eq. 13 as the product of the occupancy of the accelerators. This parameter reflects if the platform is completely exploited by the application due to the occupancy and the data transfer relation.

$$Efficiency = \prod_{i=1}^{n} Occupancy(i) \tag{13}$$

Parameter	Equation
$S_{overlap}(task)$	$\frac{t_{exec}^{acc1}}{max(t_{exec}^{acc1} - t^{acc1}(task), t^{acc1}(task))}$
$S_{th}(task)$	$\frac{t_{exec}^{acc1}}{t_{exec}^{acc1} - t^{acc1}(task)}$
S_{ideal}^{task}	$\frac{t^{acc1}(task)}{t_{exec}^{acc1} - t^{acc1}(task)}$
$S_{combined}(task)$	$\frac{t_{exec}^{acc1}}{max(t_{exec}^{acc1} - t^{acc1}(task), t^{acc2}(task))}$
$C_{overhead}$	$\frac{t_{comm}}{t_{exec}^{acc1}}$
$D_{overlap}(task)$	$\frac{max(t_{exec}^{acc1} - t^{acc1}(task), t^{acc1}(task))}{max(t_{exec}^{acc1} - t^{acc1}(task), t^{acc1}(task)) + t_{comm}}$
$D_{combined}(task)$	$\frac{max(t_{exec}^{acc1} - t^{acc1}(task), t^{acc2}(task))}{max(t_{exec}^{acc1} - t^{acc1}(task), t^{acc2}(task)) + t_{comm}}$
$Occupancy_i$	$\frac{t_{comp}^{acci}}{t_{exec}} \times 100$
Efficiency	$\prod_{i=1}^{n} Occupancy(i)$
$S_{overlap_{real}}(task)$	$S_{overlap}(task) \cdot D_{overlap}(task)$
$S_{combined_{real}}(task)$	$S_{combined}(task) \cdot D_{combined}(task)$

TABLE I
Relevant parameters involved on combining hardware accelerators. The first three parameters are known a priori while the rest are obtained a posteriori.

Similarly to communication, a certain performance degradation occurs when the computations of tasks on the accelerators are not perfectly overlapped. Although scheduling strategies can then be applied to improve the computations' balance, they are directly determined by the data dependencies between the tasks.

E. Summary

In order to clarify our approach, we want to summarize the most relevant points. Firstly, in order to find a good candidate to be accelerated, the overlap speed up expressed by Eq. 1 provides an idea about the speedup obtained by combining accelerators. The acceleration of a task does not need to be extremely high. The maximal effective accelerator speedup (Eq. 5) gives an idea about the effort required to achieve the maximum overall speedup (Eq. 3). The communication has a high cost, which cannot be omitted since it decreases the overall speedup. Our definitions of occupancy (Eq. 12) and efficiency (Eq. 13) help to identify where to focus in order to achieve the highest acceleration. Nevertheless, the communication and the non-overlapped execution of the tasks introduce performance degradation. Table I summarizes the parameters and their formulation.

III. Case Study

Cloud video surveillance systems are used to record video, detect events or for simple monitoring, all performed in remote servers. This type of systems are used to preserve video recording that can not be damaged or removed by intruders. Recent smart systems also offer motion detection

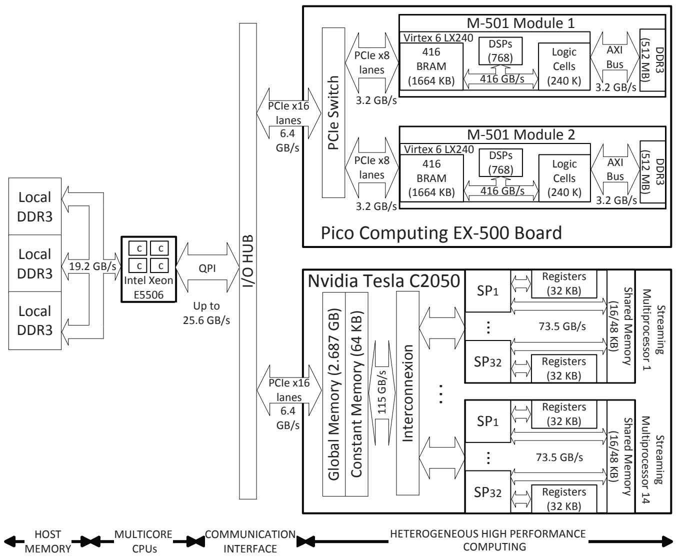

Fig. 1. *The heterogeneous cloud server is composed of a multi-core CPU, one GPU and 2 FPGAs.*

and person detection [19], all performed in the cloud. This type of application does not only demand a high BW but also the execution of several compute-intensive tasks on the remote server. The performance demands drastically increase when considering real-time person detection using multiple cameras. Our case study consists of the implementation of an algorithm, called fastHOG [20], capable to perform real-time person detection. This algorithm is implemented on a heterogeneous system composed of two different hardware accelerators. The proposed methodology is used to exploit the combination of both accelerators to reduce the execution time of the person detection algorithm.

A. Heterogeneous Cloud Server

Figure 1 details the heterogeneous system, which is composed of an Nvidia TESLA 2050 GPU and two Xilinx Virtex-6 LX240 FPGAs. Both hardware accelerators share a $PCIe$ interface to provide high-BW communication. The data transfer between both accelerators has to pass, however, through the host memory.

OpenStack cloud computing can be used to support this heterogeneous system, like proposed in [21], and in [22]. The stack description to support this heterogeneous clod server is out of the scope of this paper, therefore, we do not provide further technical details about the virtualization of this heterogeneous cloud server. The benefits of combining accelerators are explained in the following section.

B. Person Detection: fastHOG

The fastHOG is an application to detect pedestrians originally proposed to be executed on a GPU [20]. It is based on the Histogram Oriented Gradient (HOG) detector, which is a sliding window algorithm, and on a pre-trained linear Support Vector Machine (SVM) classifier, which is used to

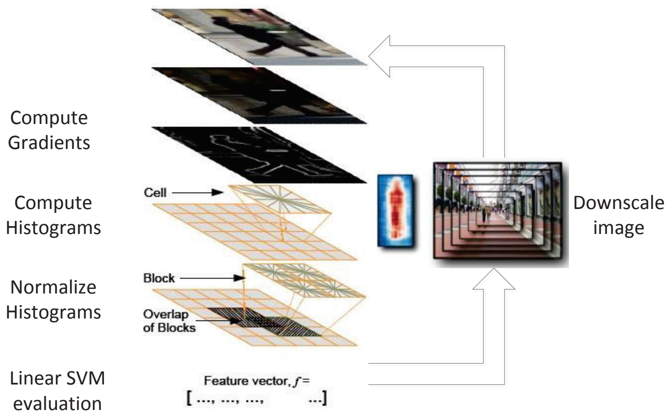

Fig. 2. *Different stages of the fastHOG [20].*

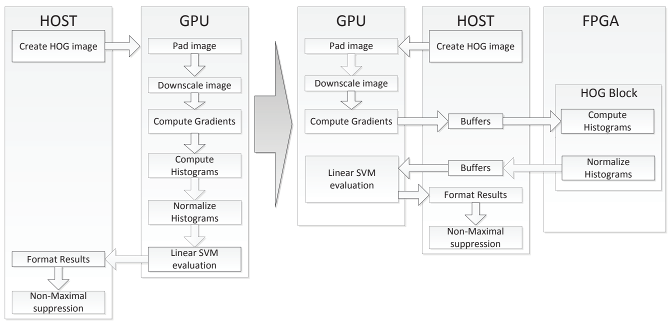

Fig. 3. *New dataflow of fastHOG combining GPU and FPGA board. The communications between both hardware accelerators is do through the $PCIe$ ports and the host memory.*

assign a matching score. Figure 2 shows the main stages. The application starts with a downscaling and a gamma normalization of the input image. Next, gradient orientation and magnitude are determined for each pixel of the preprocessed image. The resulting gradients are grouped in cells of 8x8 gradients. Furthermore, the cells are grouped in blocks of 2x2. Each block corresponds to the position of a sliding window and because the stride of this window is 8 gradients, each cell pertains to 4 blocks. For each block four histograms of gradient orientations are computed whereby each histogram corresponds to a cell of the block. It should, however, be noted that gradients of one cell may contribute to the histograms of another cell also. The four histograms of a block are stored in a 2-dimensional matrix of histograms where the position of each histogram corresponds to the position of the corresponding cell, i.e. the histograms of the top cells are on the top row and the ones of the bottom cells on the bottom row. The result is normalized using the L2-normalization and used as input for the pre-trained linear SVM classifier. Here after, the input image is downscaled and the whole process is repeated. The total computation of an image requires 31 iterations, each processing a scaled version of the original image.

C. Profiling fastHOG

The first step implementing an algorithm on different architectures is a runtime analysis. This profiling helps to identify the most interesting candidate to be accelerated. The runtime of fastHOG executed on the Nvidia TESLA 2050 GPU is breakdown in Table II. This timing analysis makes clear that HOG is the most intensive part, and therefore, the best candidate to be accelerated. The fact of being the most intensive component favours that even a minimal acceleration has a measurable impact on the application. Moreover, the HOG computation is a good candidate to be implemented on an FPGA [23], where it can perform faster than other machine learning approaches [24].

Operation	Execution time [ms]	%
Downscale image	1.237	1.59
Compute Gradients	13.461	17.33
HOG	47.851	61.61
Normalization	5.506	7.09
SVM	9.615	12.38
Total	77.67	100

TABLE II
Detailed execution time of fastHOG on the Tesla C2050.

A deeper analysis of the fastHOG scheduling shows that the L2-normalization is done over each individual block of 2x2 cells (or equivalently 16x16 gradients). However, the linear SVM classifier requires several blocks to be computed since its detection window has 64×128 pixels, equivalent to 7×15 blocks. The L2-normalization is a good breakpoint to reduce the communication impact since it reduces the output data. Therefore, the HOG computation and the L2-normalization are the best candidates to be allocated on the FPGA. Figure 3 shows the new dataflow of fastHOG combining the GPU and the FPGA boards, as well as their communication through the host. Once HOG is loaded on the FPGA, the fastHOG execution starts and ends on the GPU while the most intensive part is executed on the FPGA. The communication must always pass via the host memory.

Table III summarizes how the formulas introduced in Section II are used. Firstly, the expected $S_{overlap}(\text{HOG,L2-Norm})$ due to the FPGA can be obtained applying Eq. (1). Secondly, the maximum speedup, which occurs when the execution of HOG on the FPGA is lower than the execution time of the HOG on the GPU, is obtained using Eq. (3). Notice that while the $S_{overlap}(\text{HOG,L2-Norm})$ is achieved only thanks to use an additional accelerator, the $S_{th}(\text{HOG,L2-Norm})$ is the upper performance bound which assumes an additional speedup by running HOG on the FPGA. Finally, the desirable acceleration of HOG and L2-Normalization running on the FPGA is defined in Eq. 5.

D. Accelerating HOG on the FPGA

Adapting the HOG computation to FPGA required to consider some features of this technology. On FPGAs, floating point (FP) operations have high resource and performance costs. A comparison of the initial design is done using FP and fixed point (FxP), giving a latency of 1.28 s and 0.48 s

Speedup	Equation	Value
$S_{overlap}(\text{HOG,L2-Norm})$	$\frac{t^{GPU}(fastHOG)}{t^{GPU}(\text{HOG,L2-Norm})}$	1.45
$S_{th}(\text{HOG,L2-Norm})$	$\frac{t^{GPU}(fastHOG)}{t^{GPU}(fastHOG)-t^{GPU}(\text{HOG,L2-Norm})}$	3.198
$S_{ideal}^{\text{HOG,L2-Norm}}$	$\frac{t^{GPU}(\text{HOG,L2-Norm})}{t^{GPU}(fastHOG)-t^{GPU}(\text{HOG,L2-Norm})}$	2.194

TABLE III
Expected accelerations by considering HOG and the L2-normalization as candidates.

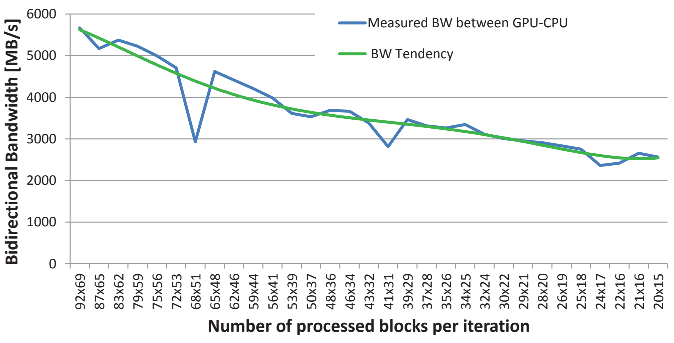

Fig. 4. *Measured bidirectional bandwidth for each iteration using the x16 $PCIe$ between the GPU and the CPU. The decrement of the achieved bandwidth is more critical between the GPU and the host since the communication between the host and the FPGA is hidden by the computation due to operating in pipeline.*

respectively, almost three times as fast as the FP version. The translation of the FP to FxP is the first adaptation of the algorithm. In addition, the data bit-width adjustment of the internal operations of the HOG algorithm not only reduces the logic consumption but also increases performance. This is a consequence of the latency reduction, since low bit-width operations are faster, although less accurate.

Computing histograms in parallel usually results in memory access conflicts that need to be managed. Our solution is to split histogram memories in individual memory banks, allowing simultaneous access. Once one block is computed, the contents of those memories are added in parallel with the computation of the next block. The use of HLS tools like the Vivado HLS tool alleviates the design effort to accelerate HOG. Moreover, it is not only possible to realize a faster design exploration but also to obtain a valid solution by operating in streaming mode and pipelining the operations. The available set of directives for memory treatment and supported I/O interfaces facilitates the memory partitioning. Due to the operation in streaming, the histogram computation starts as soon as the first data is available. The outputs of the same block need to be normalized with the L2-normalization, which is a low-latency operation to be executed in parallel with the computation of the new block. Thus, the I/O data transfer and the normalization are hidden by the HOG latency. The final FPGA design demands around 47 ms to communicate to the host and to compute HOG and L2-normalization, which is over 27x faster than the initial adapted code for FPGAs. This has been possible due to Vivado HLS directives and minor adjustments on the sequential code. Finally, in order to exploit the available resources on the FPGA, up to 16 HOG blocks can be placed in parallel. Because each block processed during the HOG computation is independent of each other, each HOG component on the FPGA processes one block at the time. Further details are available in [25]. The effort accelerating the HOG computation can be summarized as follows:

- Migrate from FP to FxP in most HOG operations.
- Avoid memory conflicts by using internal memory.
- Pipeline the execution of the HOG computation and the L2-normalization.
- Operate in streaming mode.
- Replicate the HOG cores as much as possible in order to exploit the FPGA's resources and the available I/O BW.

In spite of the modifications, the performance on the FPGA is only slightly faster than on the GPU. There exist several implementations of HOG on FPGAs that significantly offer higher performance [24], but implementing them would demand a complete redesign. Since our purpose is to present a methodology and not to implement the fastest possible HOG computation on the FPGA, we consider this implementation, which has been achieved using Vivado HLS in a short time, as good enough to help us on our purpose. Moreover, a second FPGA is used in our analysis to further accelerate HOG.

E. Analysing the communication

Once the application has been profiled, the data transfer must be analysed. This analysis identifies potential bottlenecks due to the available BW of the communication interfaces on the hybrid platform. The communications between stages of the applications is done by transferring a certain amount of data as a result of the finalization of the previous stage.

As mentioned above, the fastHOG application consists of a number of iterations, each of which tries to detect a pattern in a downscaled version of the input image. The number of iterations depends on the size of the original input image; in our case there are as many as 31 iterations. Because the input image is downscaled for each iteration, the amount of data transferred through the $PCIe$ port decreases. This fact has an additional negative impact on the available $PCIe\ BW$. As explained in [26], the attainable BW decreases when smaller amounts of data are transferred due to the $PCIe$ overhead. Figure 4 shows the measured BW between the GPU and the host for the particular execution of fastHOG. Despite our experimental measurements of $\times 16$ $PCIe\ BW$ rounds 5.5 GB/s, the small amount of data transmitted leads to a low BW consumption for most of the iterations. This communication cost must be considered as an overhead which reduces the final speedup. As a result, the total additional time dedicated to transfer data between the GPU and the CPU rounds to 18 ms, which represents a 14.68% of the total execution time based on Eq. 9.

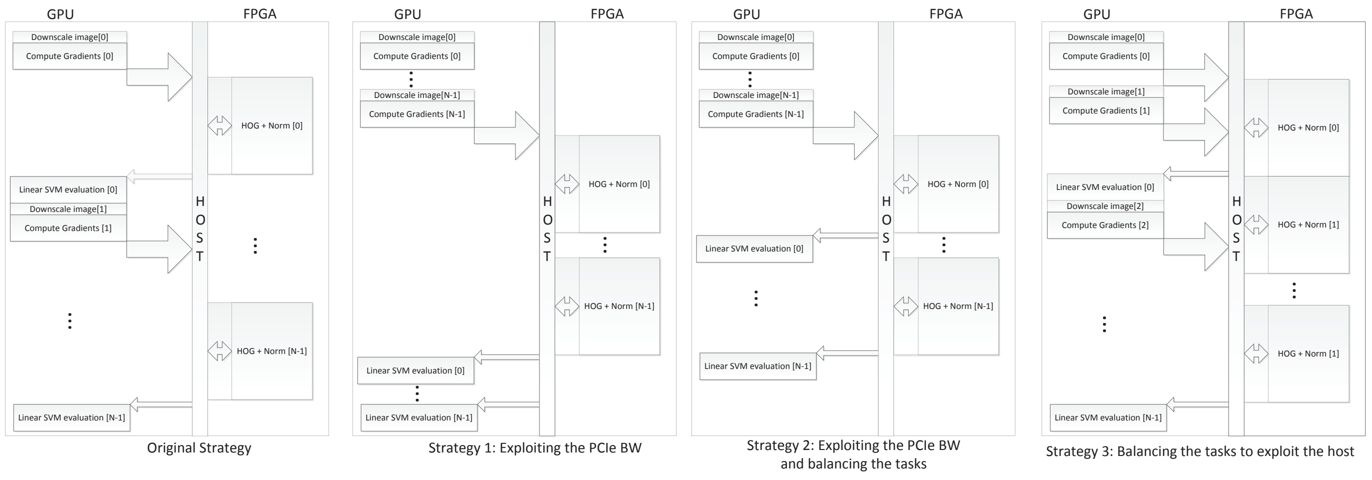

Fig. 5. *Different strategies to increase the final performance by reducing the* $C_{overhead}$.

F. Scheduling of the accelerators

In the original implementation, the input image is first transferred to the GPU where the gradients are computed. For the first iteration 6.5 MB must be transferred from the GPU to the host and from the host to the FPGA. There, the HOG computation and the L-2 normalization are executed. Finally, the results of the FPGA must be sent back via the host to the GPU where the linear SVM is computed. The whole process is repeated for each iteration processing downscaled versions of the original image. Following this strategy, the speedup obtained computing the HOG and the L2-normalization on the FPGA is $\times 0.85$ of the performance of the Tesla GPU standalone version. The main performance decreases due to the additional data transfers of data between the GPU and the host over the $PCIe$ bus. The performance becomes increasingly worse for the smaller amounts of data transferred because of the overhead involved with such a transfer:

- The *first strategy*, then, to achieve a higher BW is to carry out all downscale and gradient computation operations on the GPU together. Then, all 31 results can be transferred to the host at once. The host can store these results and send them one after the other to the FPGA until all HOG computations are completed. Finally, all results are sent to the GPU to complete all SVM computations. The main benefit of this approach is the higher $PCIe\ BW$ that can be achieved for the data communication between the GPU and the host.
- The *second strategy* is an improvement of the previous because the amount of data transferred from the host to the GPU is about 3 MB, consuming less than 5% of the total execution time. Therefore, if the GPU and the FPGA are computing in parallel, the total execution time decreases even when the maximum $PCIe\ BW$ is not reached for the host to GPU data transfer.
- One of the most important details of the combination of both accelerators is the way to communicate. There is no direct communication between the GPU and the FPGA since all data is transferred through the Northbridge and sent to the host memory. Besides this performance bottleneck, due to the complexity of the application and the use of two hardware accelerators, it is possible to properly balance the tasks between the GPU and the FPGA by using the host memory. The *third strategy* increases the number of tasks executed on the GPU while the FPGA is computing. The data transfers occur while the HOG computation and the L2-normalization are computed on the FPGA. Therefore, the FPGA is busy most of the time while the GPU is generating all the data that the FPGA is going to require for the next iterations.

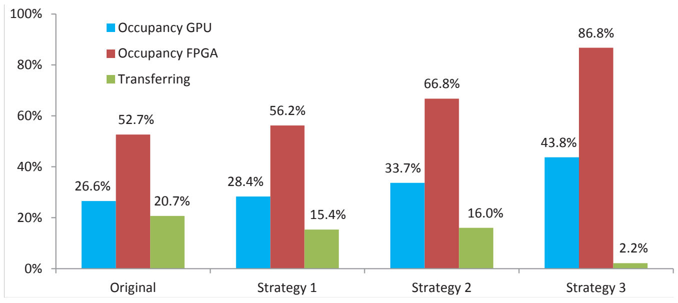

Fig. 6. *The occupancy of the accelerators helps to identify the level of task's balancing between accelerators. While the percentages sum 100% there will not be overlap and the communication would have a significant impact.*

Figure 5 details the schedule of the four strategies. For every strategy, the occupancy of each accelerator is calculated. Figure 6 shows the evolution while the strategies improve the balance of the tasks between accelerators. On the first two strategies, where there is no overlap between accelerators, the percentages sum 100%. As soon as the tasks on the accelerators start to be overlapped, the impact of the communication is hidden and the occupancy increases on both accelerators. In our case, as there is no real speedup on the HOG FPGA's implementation, the execution time on the FPGA dominates

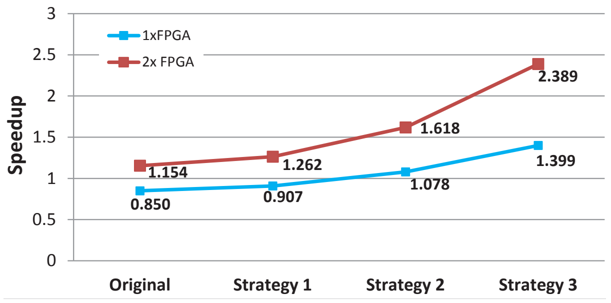

Fig. 7. *Relative speedup of the presented strategies compared to the original GPU standalone version.*

the execution of the application. Thus, it is not possible to increase the occupancy on the GPU side further.

Finally, Figure 7 shows the evolution of the $S_{combined_{real}}$ respect to the original fastHOG computed on the GPU standalone. Our third scheduling strategy attains a speedup of 1.39, slightly below the expected $S_{overlap}(\text{HOG,L2-Norm})$ detailed in Table III due to a residual 2.2% related to $C_{overhead}$. The use of the second FPGA available on the described platform enables further acceleration when distributing the computation of of HOG and L2-Normalization between both FPGAs. Notice that this additional accelerator only accelerates the computation of of HOG and L2-Normalization, by almost reaching $S_{ideal}^{\text{HOG,L2-Norm}}$. Moreover, the use of an additional FPGA does not increases $C_{overhead}$ since the additional communication is only related to the data transfer between the GPU and the CPU.

G. Summary

Our methodology helped us to early predict the achievable speedups and, through scheduling strategies, to achieve the highest reachable performance. Although there is no real acceleration executing the HOG and the L2-normalization on the FPGA, the expected $S_{overlap}$ is achievable by exploiting the task's balance between the GPU and the FPGA. Finally, our scalable methodology shows how additional accelerators can be used to obtain closer speedups to the ones predicted in Table III.

IV. Conclusions

Heterogeneous cloud computing servers are able to satisfy the most computationally demanding applications at the cost of additional design effort to exploit the advantages of each type of hardware accelerator. The proposed methodology enables an early estimation of the achievable performance acceleration due to combining hardware accelerators. The combination of heterogeneous accelerators provides more opportunities to accelerate compute-intensive tasks by exploiting the key features of each technology. However, the impact of the communication must be considered in order to avoid performance degradation. Nevertheless, we believe that tasks with high data traffic demands can better exploit the BW of $PCIe$ more efficiently than tasks with lower data demand. Our methodology shows how an adequate rescheduling of the tasks executed on the hardware accelerators significantly increases the performance acceleration thanks to hiding the communication latency by the accelerator's computation.

References

[1] Tsoi, Kuen Hung, et al. "Axel: a heterogeneous cluster with FPGAs and GPUs." Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays. 2010.

[2] Showerman, Michael, et al. "QP: A heterogeneous multi-accelerator cluster." Proc. 10th LCI International Conference on High-Performance Clustered Computing. 2009.

[3] Inta, Ra, et al.. "The Chimera: an off-the-shelf CPU/GPGPU/FPGA hybrid computing platform." International Journal of Reconfigurable Computing. 2012.

[4] da Silva, Bruno, et al. *"Comparing and combining GPU and FPGA accelerators in an image processing context."* Field Programmable Logic and Applications (FPL), IEEE 23rd International Conference on. 2013.

[5] Google Cloud, https://cloud.google.com/tpu/

[6] Amazon EC2 Elastic GPUs, https://aws.amazon.com/ec2/elastic-gpus/

[7] Amazon EC2 F1, https://aws.amazon.com/ec2/instance-types/f1/

[8] Microsoft Azure, https://azure.microsoft.com/en-us/resources/videos/build-2017-inside-the-microsoft-fpga-based-configurable-cloud/

[9] Alibaba E-HPC, https://www.alibabacloud.com/product/ehpc

[10] Minhas, Umar Ibrahim, et al. *"Exploring Functional Acceleration of OpenCL on FPGAs and GPUs Through Platform-Independent Optimizations."* Applied Reconfigurable Computing. Architectures, Tools, and Applications: 14th International Symposium on. Springer. 2018.

[11] Meswani, Mitesh R., et al. *"Modeling and predicting performance of high performance computing applications on hardware accelerators."* Journal of High Performance Computing Applications, 2013

[12] Moncrieff, David, et al. *"Heterogeneous computing machines and Amdahl's law."* Parallel Computing 22.3. 1996

[13] Marowka, Ami. *"Extending Amdahl's law for heterogeneous computing."* Parallel and Distributed Processing with Applications (ISPA), IEEE 10th International Symposium on. IEEE, 2012.

[14] Colella, Phillip. *"Defining software requirements for scientific computing."* (2004).

[15] Asanovic, Krste, et al. *"The landscape of parallel computing research: A view from berkeley."* Vol. 2. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[16] Williams, Samuel, et al. *"Roofline: an insightful visual performance model for multicore architectures."* Communications of the ACM. 2009.

[17] da Silva, Bruno, et al. *"Performance modeling for FPGAs: extending the roofline model with high-level synthesis tools."* International Journal of Reconfigurable Computing. 2013

[18] da Silva, Bruno, et al. *"A Lost Cycles Analysis for Performance Prediction using High-Level Synthesis."* International Symposium on Applied Reconfigurable Computing. Springer, Cham, 2016.

[19] Amazon Cloud Camera Security, https://cloudcam.amazon.com/

[20] Prisacariu, Victor, et al. *"fastHOG-a real-time GPU implementation of HOG."* Department of Engineering Science 2310.9. 2009.

[21] Crago, Steve, et al. *"Heterogeneous cloud computing."* Cluster Computing (CLUSTER), IEEE International Conference on. IEEE, 2011.

[22] Byma, Stuart, et al. *"FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack."* Field-Programmable Custom Computing Machines (FCCM), IEEE 22nd Annual International Symposium on. 2014.

[23] Kadota, Ryoji, et al. *"Hardware architecture for HOG feature extraction."* Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP'09). Fifth International Conference on. IEEE, 2009.

[24] Drre, Jan,et al. *"A HOG-based Real-time and Multi-scale Pedestrian Detector Demonstration System on FPGA."* Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2018.

[25] da Silva, Bruno, et al. *"Study of combining GPU/FPGA accelerators for High-Performance Computing."* High-Level Synthesis for High Performance Computing workshop (HLS4HPC). HiPEAC, 2013.

[26] Coleman, James, et al. *"Hardware level IO benchmarking of PCI express."* Intel White Paper (2008).