Optimizing Memory Bandwidth in OpenVX Graph Execution on Embedded Many-Core Accelerators

Giuseppe Tagliavini
University of Bologna
Bologna, Italy
giuseppe.tagliavini@unibo.it

Germain Haugou
STMicroelectronics
Grenoble, France
germain.haugou@st.com

Luca Benini
University of Bologna / ETH Zurich
Bologna, Italy / Zurich, Switzerland
luca.benini@unibo.it / luca.benini@iis.ee.ethz.ch

***Abstract*—Computer vision and computational photography are hot applications areas for mobile and embedded computing platforms. As a consequence, many-core accelerators are being developed to efficiently execute highly-parallel image processing kernels. However, power and cost constraints impose hard limits on the main memory bandwidth available, and push for software optimizations which minimize the usage of large frame buffers to store the intermediate results of multi-kernel applications.**

In this work we propose a set of techniques, mainly based on graph analysis and image tiling, targeted to accelerate the execution on cluster-based many-core accelerators of image processing applications expressed as standard OpenVX graphs. We have developed a run-time framework which implements these techniques using a front-end compliant to the OpenVX standard, and based on an OpenCL extension that enables more explicit control and efficient reuse of on-chip memory and greatly reduces the recourse to off-chip memory for storing intermediate results. Experiments performed on the STHORM many-core accelerator prototype demonstrate that our approach leads to massive reductions of main memory related stall time even when the main memory bandwidth available to the accelerator is severely constrained.

I. Introduction

The evolution of imaging sensors and the growing requirements of applications to provide image understanding and processing functions are pushing hardware platform developers to incorporate advanced image processing capabilities into a wide range of embedded systems, ranging from smartphones to wearable devices. In particular, we consider instances of three classes of computationally intensive image processing tasks: Embedded Computer Vision (ECV) [1], brain-inspired visual processing computing [2], and computational photography [3].

Considering the actual market trend toward HD formats and real-time video analysis, these algorithms require hardware acceleration. Pushed by the need for extreme energy efficiency, embedded systems are embracing architectural heterogeneity, where a multi-core host processor is coupled with programmable many-core accelerators specialized for various application domains.

A heated debate is still ongoing on which accelerator architecture is best suited for speeding up advanced image-centric algorithms. Embedded GPUs are readily available and they do offer programming flexibility, but their use for computational imaging and computer vision in low-cost, energy-constrained mobile and embedded systems is not pervasive, and a few benchmarking experiments show that current embedded GPUs do not reach a fully satisfactory energy-efficiency sweet spot for advanced image processing and computer vision algorithms [4], [5]. Hence, several companies and research groups are looking for alternative solutions, ranging from using special functional units on the host CPU [6] to dedicated vector units [7], to many-core accelerators.

Many-core accelerators provide tens to hundreds of small processing elements (PEs) organized in clusters sharing on-chip L1 memory and communicating via low-latency, high-throughput on-chip interconnections. They differ from GPUs in two main traits. First, PEs are not restricted to run the same instruction on different data in an effort to improve execution efficiency of branch-rich computations and to support a more flexible workload-to-PE distribution. Second, DMA engines are used to program data transfers between main memory and on-chip memories without involving the cores, as opposed to using memory coalescing and multi-threading to hide main memory latency. Some examples of accelerators featuring this architectural paradigm are STM STHORM [8], Plurality HAL [9], KALRAY MMPA [10], and Adapteva Epiphany-IV [11]. In these architectures the PEs are simpler w.r.t. common multi-core architectures and offer a better trade-off between highly parallel computation and power consumption, so they are a promising target for running image processing workloads.

From the software viewpoint, tn the last years a number of programming models for many-core accelerators have been proposed [12]. Among others, the very successful Khronos OpenCL standard introduces platform and execution models which are particularly suitable for programming at the emerging intersection between multi-core CPUs and many-core programmable accelerators. OpenCL offers a conjunction of task parallel programming model, with a run-to-completion approach, and data parallelism, with global synchronization mechanisms. An OpenCL application runs on the host processor and distributes kernels on computing devices. Kernels are programmed in the OpenCL C language, which is based on the C99 standard. On the host side, applications are written in C/C++ language, and invoke standard API calls to orchestrate the distribution and execution of kernels on devices, using a mechanism based on command queues.[1]

A common issue of using OpenCL on embedded systems is related to the mandatory use of global memory space to share intermediate data between kernels. When increasing the number of interacting kernels, the main memory bandwidth required to fulfill data requests originated by PEs is much higher than the available one, causing a bottleneck. In addition, unlike accelerators for desktop computing environments (e.g. Intel Many Integrated Core architecture [13]), SoCs have unified host and global memory spaces, and have a common data path connecting host processor and accelerator with L3 memory. As a direct consequence, applications experience high contention for off-chip memory access, that may severely limit the final speed-up.

For instance, we consider a platform with an accelerator and a DDR3 memory (1.25GB/s per channel). If we reasonably assume that the accelerator has half of the available bandwidth (625MB/s), and our application need to process a 1280x800 video source at 30fps, then a full image access uses 30.72MB/s. Hence, after accessing 20 image buffers in one frame time the available bandwidth is saturated, but this could not be enough for a complex application that instantiates many kernels and requires many intermediate results.

Within a single OpenCL kernel this limit can be bypassed using the *async_work_group_copy* function, which provides asynchronous copies between global and local memory and vice versa. Nevertheless, this explicit handling of the data transfers turns into a complex task for programmers, since it requires a lot of programming effort and it is prone to errors. For basic kernels, the lines of code used for DMA orchestration and subsequent workload distribution are 90%

[1]The OpenCL 2.0 standard also enables dynamic parallelism on device side, but most programming environments do not support it yet.

of the total. Moreover, different kernels which share data are forced to use global memory.

Other programming approaches have been proposed to implement image processing applications on embedded systems, based on data-flow graphs [14] [15] or functional models [16]. Overall, these solutions have the advantage to greatly simplify the memory management, but in general they are not user-friendly for embedded system programmers, as they prefer C-based programming environments, which are much more open to the low-level optimizations that are fundamental for this domain. Moreover, in some cases these models are not suitable to build complex applications, that may include tens of kernels.

In this work we introduce a framework that implements a set of optimizations specifically targeted to accelerate the execution of graph-based image processing applications on many-core accelerators. The framework front-end is based on the current OpenVX standard proposal [17]. OpenVX is a cross-platform API standard which aims at enabling hardware vendors to implement and optimize low-level image processing and computer visions primitives, with a strong focus on mobile and embedded systems. OpenVX can be used directly by applications or to accelerate higher-level middleware, as it is a specification across multiple vendors and platforms. Image processing applications are commonly structured as a set of basic functions that interact with some data dependencies, hence the OpenVX model is based on a directed acyclic graph of nodes (i.e. kernels) with data as linkage. After its specification, the graph must be verified by the OpenVX run-time environment to make sure that the input and output data requirements are compliant to node interface on each parameter: at this stage, vendor specific optimization may be applied. Thereafter, the same graph can be executed multiple times, just changing the data inputs.

In our framework, most read/write operations are performed on local buffers in the L1 scratchpad memory of the reference architecture, that is what we call a *localized execution*. To satisfy this condition, we have defined a set of techniques to support automatic data tiling for input, intermediate and output data. We consider all the data access patterns that are most common in image processing and computer vision, and we provide support for tile overlap and graph partitioning. Using the double buffering mechanism, we can also achieve a good overlap between data communication and kernel execution on PEs, that guarantees a higher efficiency in terms of PE usage.

Our framework provides huge benefits in terms of speed-up when compared to execution of sequential code, for instance we get up to 31x for a complex benchmark in our experimental set. Experiments have also assessed high efficiency, over 95% of total accelerator time, and consistent bandwidth reduction, always over 50%.

The rest of this paper is organized as follows. In Section II we discuss the related works. In Section III we present the reference platform and its software run-time. In Section IV we describe our approach. We discuss the experimental results in Section V. Finally, we summarize our conclusions and discuss future research directions in Section VI.

II. Related Works

OpenCV [18] is an open-source and cross-platform library featuring high-level APIs for Computer Vision. OpenCV is the de-facto standard in desktop computing environment, its mainstream version is optimized for multi-core processors but it is not suitable for acceleration on embedded many-core systems. Some vendors provide accelerated versions of OpenCV which have been optimized for their hardware (e.g. OpenCV for Texas Instruments embedded platforms [19] or OpenCV for Tegra [20]). As an alternative to OpenCV, Qualcomm provides a specific library for ECV which includes the most frequently used vision processing function. This library is called FastCV [6], and it is optimized for ARM-based processors and tuned to take advantage of Qualcomm's Snapdragon processors. As a matter of fact, OpenCV needs a lower-level middleware for accelerating image processing primitives. This is precisely the goal of OpenVX, which aims at providing a standardized set of accelerated primitives, thereby enabling platform agnostic acceleration.

OpenCL [21] is a very widespread programming environment for both many-core accelerators and GPUs, and it is supported for an increasing number of heterogeneous architectures; for instance, Altera supports OpenCL on its FPGA architecture [22]. The OpenCL memory model is too constrained for SoC solutions, hence many extensions have been proposed by vendors. For instance, AMD provides a zero-copy mechanism to share data between host and GPU in Fusion APU products, also enabling the access to GPU local memory by host side through a unified north-bridge with full cache coherence [23]. In a many-core accelerator we need even more control on data allocation, because cores are not working in lock-step. In addition, we need the possibility to map the logical global space at different levels of the memory hierarchy, to efficiently maintain state between kernels.

There are well-known extensions for imperative programming languages which provide mechanisms to execute code on accelerators, and it is worth mentioning OpenMP [24], OpenACC [25] and Sequoia [26]. With these approaches, the compiler usually performs most optimization at loop level. Conversely, vision graphs assume the form of a sequence of loops nests, potentially located over different compile units. The connections among kernels are not taken into account, and accordingly applications may not be optimized at a global level, considering issues like data locality. Our focus is limited to the image processing domain, since in this context we can define an application with a component-based approach, and most details are implicit (including the memory transfers). Moreover, the above are general-purpose parallel programming environment, while in this work we take the route of domain-specific languages, since image processing is a large and very active domain that justifies this kind of approach [27].

Focusing on domain-specific approaches, Halide [16] is a programming language specifically designed to describe image processing pipelines with different architectural targets, including an OpenCL code generator. To implement an algorithm with Halide, the user must specify a functional description using a specific language. Halide defines a model based on stencil pipelines, with the aim to find a trade-off between locality, exploitation of parallelism, and redundant re-computation. Even if the basic principles are the same, our framework is based on specific architectural features of a class of many-core accelerators, so we can simplify some allocation problem constraints with minimum loss of generality. Moreover, the functional programming style is not fully suitable for complex programs, and embedded programmers are not used to it.

Graph-structured program abstractions have been studied for years in the context of streaming languages (e.g. StreamIt [14]). In these approaches, static graph analysis enables stream compilers to simultaneously optimize data locality by interleaving computation and communication between nodes. However, most research has focused on 1D streams, while image processing kernels can be modeled as programs on 2D and 3D streams. The model of computation required by image processing is also more constrained than general streams, because it is characterized by specific data access patterns. Some good results have been achieved with special-purpose data-flow processors targeted for vision algorithms (e.g. NeuFlow [15]). Our model takes into account the specific characteristics of image processing domain, but we target a general-purpose accelerator.

Stencil kernels are a class of algorithms applied to multi-dimensional arrays, in which an output point is updated with weighted contributions from a subset of neighbor input points (called window or stencil). Our definition of tiles is equivalent to a 2D stencil. Many optimization techniques have been proposed to execute stencil kernels on multi-core platforms

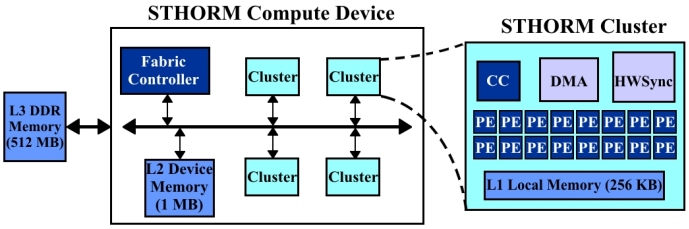

Fig. 1. STHORM Platform

[28], but an effective solution for many-core accelerators executing heterogeneous vision kernels has not been proposed yet. Such a solution has to consider all the data access pattern specific of this domain, handling the possible overlapping of input windows and providing a solution for the access patterns that are not properly describable in terms of stencil computation.

KernelGenius [29] is a tool that enables the high-level description of vision kernels using a custom programming language. It aims at generating an optimized OpenCL kernel targeting the STHORM platform, with a totally transparent management of the DMA data transfers. The structure of the tiling problem for a single kernel is analogous to the formulation we are using in this work. The final performance are equivalent to an optimized OpenCL solution, but there are also the same limitations that we have reported for OpenCL.

Many previous works have proposed specific optimizations for architectures with explicitly managed scratchpad memories, such as Cell BE [30]–[32]. In this paper we propose a totally transparent approach for the well-defined scope of image processing applications. We do not introduce any new programming model which programmers should learn in addition to the standard OpenVX API, and overall we get the major benefits from all the cases in which the localized execution is not directly possible.

III. Architecture

A. Hardware platform

Our approach targets many-core accelerator with the following characteristics: (i) a set of MPMD (Multiple Program, Multiple Data) processing elements organized into clusters, (ii) a scratchpad memory characterized by a low latency and shared between PEs at cluster level, and (iii) a DMA engine to transfer data. The platform used to validate our framework is the STHORM SoC by STMicroelectronics, previously known as P2012 project [8]. STHORM is a many-core computing accelerator based on clusters interconnected by an asynchronous network-on-chip (Figure 1). Each cluster features 16 dual-issue STxP70 cores, supporting MPMD instruction streams. Moreover, each cluster contains a multi-banked one-cycle access L1 scratchpad memory, connected by a multi-level logarithmic interconnect, and a dual-channel DMA engine that can handle both linear and rectangular transfers.

A L2 scratchpad memory is shared by all clusters, and it is currently used by software run-time to store accelerator binaries. The architecture has no data cache, as it is designed to minimize SoC size and energy consumption,

We use a STHORM evaluation board based on the Xilinx Zynq 7000 FPGA device, featuring a ARM Cortex A9 dual core host processor running at 667MHz, main (L3) DDR3 memory, plus programmable logic (FPGA). The STHORM chip is clocked at 430MHz. The ARM subsystem on the Zynq is connected to a AMBA AXI interconnection matrix, through which it accesses the DRAM controller. The latter is connected to the on-board DDR3. To allow transactions generated inside the STHORM chip to reach the L3 memory, and transactions generated inside the ARM system to reach internal STHORM L1 and L2 memories, part of the FPGA area is used to implement a bridge to/from the STHORM chip. It is important to stress that in this two-chip prototype, the main memory bandwidth available to the STHORM accelerator is severely limited by the low-speed link between the accelerator and FPGA. Clearly, in a full production SoC scenario accelerators and CPU host share the same silicon die and the accelerator gets a much larger share of the main memory bandwidth. The evaluation board represents a very challenging and interesting scenario for any software optimization focusing on reducing main memory bandwidth needs.

B. OpenCL run-time

The low-level programming environment for the STHORM platform, on which we build out our OpenVX run-time, is based on OpenCL 1.1 standard [33].In this environment, data must be transferred into shared local memory prior to computation in order to take advantage of low-latency accesses, and it has to be done explicitly using OpenCL built-in functions for asynchronous work-group copy. STHORM cores are hardware mono-threaded, hence the best performance is achieved by exactly matching the OpenCL ND-Range with the STHORM architectural parameters to avoid expensive context switches: that is, programmers should use as many work-groups as the number of clusters, and as many work-items as the number of processing elements.

In this work we use an *extended OpenCL run-time (CLE)*, which enables the creation of graphs containing nodes of different types:

- `CreateBuffer` – a node that allocates a buffer at the specified memory level (currently supported: L1, L3).
- `CopyBuffer` – a node that enqueues a DMA transfer to copy a portion of a source buffer to a portion of a destination buffer.
- `ExecKernel` – a node that enqueues a kernel execution on a single cluster, requesting a specified number of cores.
- `ReleaseBuffer` – a node that releases the specified buffer.
- `EndGraph` – a node that triggers the end of the graph execution.

When creating a CLE graph, the user has to specify the kernel actual parameters and a set of dependencies for each node: if there is a dependency edge between node A and node B, node A must terminate its execution before node B can start. The memory management is totally explicit, including the allocation of stack area used by cores for kernel execution. The `cleGraphSetBuffer` primitive associates an OpenCL classic buffer to a graph input/output, and using `clEnqueueGraph` we can add a graph execution request in a standard OpenCL queue. When the `EndGraph` node terminates, a notification is sent to the host side, and the concerned cluster is made available for a subsequent graph execution.

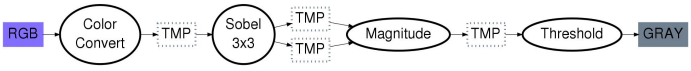

Fig. 2. OpenVX sample application (graph diagram)

IV. Execution model and memory-centric optimization

A. OpenVX front-end

To introduce OpenVX programming, we consider the example application depicted in Figure 2, that is an edge detector based on Sobel filter. All the kernels are primitives contained in the OpenVX API proposal, for further reference see [17]. The OpenVX code snippet for this application is reported in Listing 1. Note that an OpenVX program is much more abstract and concise than an OpenCL program, and with our approach the underlying OpenCL runtime is completely

hidden to the programmer. Moreover, we suppose to have no additional information from the programmer.

```
1 vx_context ctx = vxCreateContext();
2
3 vx_image imgs[] = {
4   vxCreateImage(ctx, width, height, FOURCC_RGB),
5   vxCreateVirtualImageWithFormat(ctx, FOURCC_U8),
6   vxCreateVirtualImageWithFormat(ctx, FOURCC_U8),
7   vxCreateVirtualImageWithFormat(ctx, FOURCC_U8),
8   vxCreateVirtualImageWithFormat(ctx, FOURCC_U8),
9   vxCreateImage(ctx, width, height, FOURCC_U8),
10 };
11 vx_node nodes[] = {
12   vxColorConvertNode(graph, imgs[0], imgs[1]),
13   vxSobel3x3Node(graph, imgs[1], imgs[2], imgs[3]),
14   vxMagnitudeNode(graph, imgs[2], imgs[3], imgs[4]),
15   vxThresholdNode(graph, imgs[4], thresh, imgs[5]),
16 };
17
18 status = vxVerifyGraph(graph);
19
20 while (/* input images? */) {
21   /* capture data into imgs[0] */
22   status = vxProcessGraph(graph);
23   /* use data from imgs[5] */
24 }
```

Listing 1. OpenVX code snippet (C code)

In OpenVX programs some intermediate images are declared as virtual (lines 5-8). As specified by the OpenVX reference, virtual data are not guaranteed to actually reside in main memory, and can not be read or written using the API. Basically, virtual data just define a dependency between adjacent kernel nodes, and is not associated with any accessible memory area. These intermediate virtual images are the target of our optimization efforts. In contrast, the images defined in line 4 and 9 are not virtual, as they represent graph input and output data.

Before execution, an OpenVX graph must be verified to guarantee some mandatory properties, that are:

- Input and output requirements must be compliant to the node interface (data direction, data type, required/optional flags).
- No cycles are allowed in the graph.
- Only a single writer node to any data object is allowed.
- Writes have higher priorities than reads.
- Virtual data can be made into either real data or optimized away during verification.

After these topological verification steps, we integrate our algorithms for graph partitioning and scheduling, that provide buffer allocation, buffer sizing and CLE run-time graph creation. Our main goal is the maximization of execution efficiency, which is defined as the amount of time that PEs spend to execute kernel code over the total graph execution time spent on accelerator side. This goal implies to minimize the total waiting time due to memory transfers to/from L3 memory. For this purpose, virtual images are not allocated in L3 memory, but they are partly allocated in a set of L1 buffers managed by the framework. In the next section we initially consider data access patterns for each kernel, and then we provide a global approach at graph level.

Since the verification stage of an OpenVX graph is performed at run-time, that gives the capabilities of changing the application graph with an adaptive approach and supporting dynamic hardware resources, we have preferred a heuristic approach to the optimization steps performed by the run-time when preparing a graph to execute.

B. Data access patterns

In an unoptimized OpenVX run-time, images reside by default in L3 memory; in out framework, they are partitioned into smaller blocks, called *tiles*, to fit L1 buffers. Intermediate (virtual) images are not allocated to L3 memory whenever it is possible w.r.t. other constraints. The allowed size for tiles strictly depends on the data access patterns used by kernels. To describe these patterns, we associate a tiling descriptor to each input/output port of OpenVX kernels. For input data, this structure specifies the minimum set of points necessary to

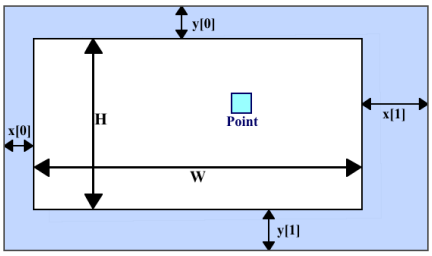

Fig. 3. Structure of a tiling descriptor

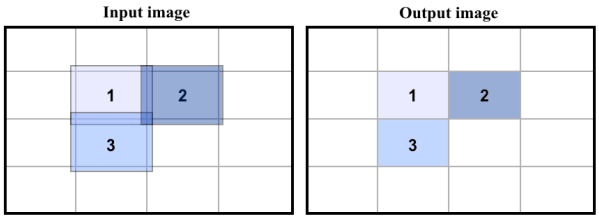

Fig. 4. Image tiling schema for a single kernel

compute an output value, in terms of both *computing area* and *neighboring area*. Hereinafter we consider the common case of a single point (i.e. image pixel) per output value, but in some cases it could comprise more adjacent points (e.g. scaling operator), and this condition is managed by the framework. Figure 3 describes the structure of a tiling descriptor. W and H are the dimensions of the computing area, that is the set of points used to compute a single output value; for output tiles, these values represent the minimum number of output points generated by a single computation. x and y values describe the neighboring area, that is the set of additional points contributing to the computation of a single output value, but they may belong to other computing or neighboring areas.

The distinction between computing and neighboring area is really important, since it has a major impact on data partitioning. The partition of input images into tiles is correct when the juxtaposition of the output tile produces a complete output image, equivalent to execute the graph in one step, that is when the tiles have the same size of the images. In this context, x and y values exactly correspond to the horizontal/vertical overlap between adjacent tiles which is required to compute all the points in the output tile. Taking into account a single kernel, the size of output tiles is implied by input tiles, and output tiles do not require any overlap (see Figure 4).

Referring to the literature on image processing functions [34], we can identify five different classes of operators (see Figure 5):

A) *Point operators* (e.g. color conversion, threshold) compute the value of each output point from the corresponding input point. These operators do not require any tile overlap by construction ($W = 1, H = 1, \forall ix[i] = 0$).
B) *Local neighbor operators* (e.g. linear operators, morphological operators) compute the value of a point in the output image that corresponds to the input tile. Local neighbor operators require a complete handling of the tile overlap, based on the parameters of the kernel neighboring area ($W = 1, H = 1, \exists ix[i] >= 1$).
C) *Recursive neighbor operators* (e.g. integral image) are similar to the previous ones, but in addition they also consider the previously computed values in the output tile. The managing of tiling is equivalent to local neighbor operators, but we also need to save state data between tiles (usually the borders of previous output tiles).
D) *Global operators* (e.g. DFT) compute the value of a point

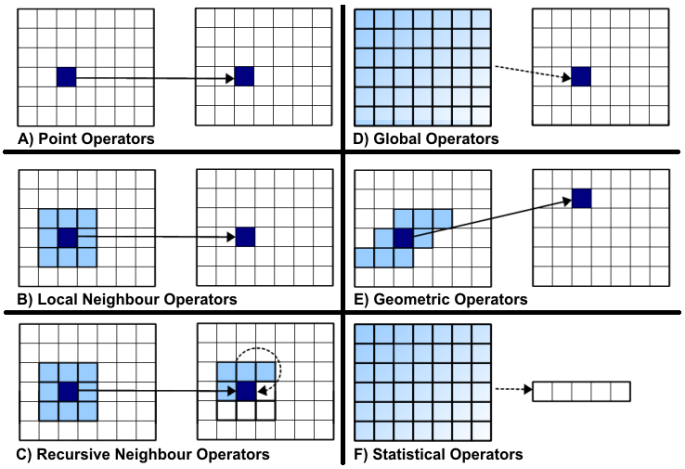

Fig. 5. Classes of image processing kernels

in the output image using the whole input image. In this case it is impossible to apply tiling to input data.

E) *Geometric operators* (e.g. affine transforms) compute the value of a point in the output image using a non-rectangular input area. In the most general case, we cannot apply a classical input tiling due to the generic shape of the neighboring area. For some transformations we can specify a tile defining a bounding box, even if this causes an overhead in terms of data that are transferred and not used, and we can derive an equivalent local neighbor operator.

F) *Statistical operators* (e.g. mean, histogram) compute statistical functions of image points. Tiling can be activated on input images, and we can use a persistent buffer to implement a reduction pattern "walking" through the tiles.

C. Tile size propagation

When we execute a complete graph, we must take into account an additional effect that we call *execution overlap*. Considering the code of Listing 1 and an output tile size of 160x120, we get the results depicted in Figure 6. `Color Convert` is a point operator, and so it does not need any overlap by itself. However the `Sobel 3x3` operator specifies a neighboring areas of one point ($W = 1, H = 1, \forall ix[i] = 1$). This connection adds a constraint on the intermediate tile B1: overall, we must consider an output with no overlap for the first kernel and an input with a fixed overlap for the second one. To satisfy this inter-kernel constraint, we enlarge the tile B0 of a fixed overlapping area and compute the color conversion kernel multiple times for the points on the borders (exactly one per including tile).

Execution overlap enforces data locality for intermediate buffers at the cost of transferring and computing the tile borders multiple times, but we have observed during experiments that this aspect does not affect the general benefits of data locality. To guarantee a correct execution of the entire graph, the computation of the tile size is computed into two passes: (i) the first pass analyzes the graph forward, simulating a graph execution and simultaneously collecting the tiling constraints for each kernel, which are associated to physical buffers; (ii) the second pass performs a backward analysis, starting from the last simulated node, and sets the buffer final overlap according to all collected constraints.

This model directly applies to operators of classes A, B, and C (see Section IV-E for a discussion on the other classes).

D. State buffers

To handle the computation of recursive neighbor operators, the framework supports the use of *state buffers*. Each kernel implementation must specify the amount of bytes needed to

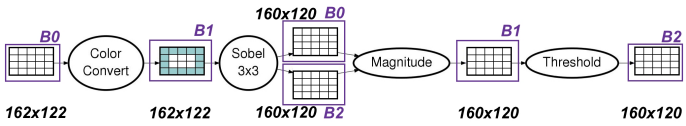

Fig. 6. Example of tile size propagation

maintain its state across tiles, as a constant value or as a linear combination of buffer size (width and height). The framework computes the final size of state buffers and allocates them, then each kernel is responsible to handle the content of its buffer. State buffers are persistent for a specific kernel instance over multiple tile executions.

State buffers are also used to handle reduction patterns when executing statistical operators. For example, the mean kernel can use different accumulator variables (sum of values, number of points) for each executing core, and after the last tile a single core is responsible to compute the reduction and perform a division. At the current stage of development, the framework provides a pointer to the state buffer and some flags (first tile/last tile), and the kernel programmer must implement the reduction patterns directly in the kernel code. Overall, this mechanism is totally transparent to the user of OpenVX interface.

E. Graph partitioning

There are some applications for which the resultant graph cannot be executed allocating all the intermediate tiles in L1 buffers, basically for two reasons: (i) the buffer sizing algorithm fails to fit all buffers in L1 memory (see Section IV-G), or (ii) the graph contains a kernel of classes D, E, or F. In these cases. the graph is automatically partitioned into multiple sub-graphs, each one corresponding to a CLE run-time graph, and the intermediate images that connect different graphs are saved into L3 memory. Hence, the execution of an OpenVX graph is divided into multiple stages at run-time level, and the tiling is applied at each stage independently. This process is totally transparent to the programmer

Kernels of classes D and E are executed separately, as they do not support any tiling scheme, while a graph boundary is automatically added after a kernel of class F. As a further optimization, all the statistical operators are postponed as long as possible by the partitioning algorithm, and eventually they are grouped together at the end of the current sub-graph schedule.

F. Node scheduling

For each graph extracted at the previous stage, a node scheduling is determined through a breadth-first visit, starting from the kernels connected to graph input data (head nodes). The current version of the framework considers a single cluster, and so a single kernel node is selected for execution, hence all the processing elements are always allocated to the running kernel and the schedule is an ordered list.

Experimental results on STHORM show that the contention for L1 memory is very limited when all the cores are active, due to the low-latency of the logarithmic interconnect and the address interleaving across a large number of memory banks. In this scenario, having all the PEs executing the same kernel guarantees that precedence constraints bound to active kernel can be faster satisfied, and the time gaps in the schedule are minimized. At the same time, the number of output buffers that are currently active is the lowest one, accordingly to the policy of Section IV-G.

G. Buffer allocation and sizing

The buffer allocation policy specifies the maximum number of buffers that are allocated in L1 memory and their association to input/output kernel image parameters.The visit order used by the scheduling algorithm guarantees that allocated buffers

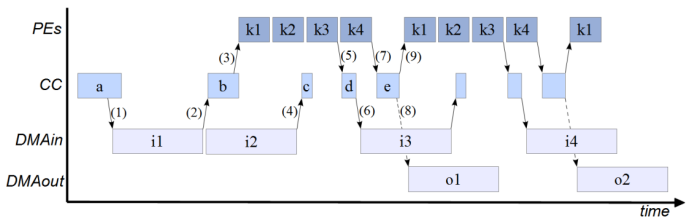

Fig. 7. Example of CLE runtime schedule

are used as soon as possible, and nodes that release more data references are executed first, so that we can promote buffer reuse to save L1 memory space.

1) The number of L1 buffers that are initially allocated is equal to the number of input images to the graph.
2) When a kernel is added to the schedule list, we allocate output images to buffers. If there is a buffer that is no more used, we reuse it, otherwise we increment the buffer count; Due to double buffering policy, buffers that have been use for inputs cannot be reused for outputs.
3) Using a reference counter, we verify whether images are used by other nodes; if there is no further reference, we can reuse it, hence we add the buffer to the free list. Buffers associated with graph outputs can be reused no more, so they are never added to the free list.

As a subsequent step, the buffer sizing algorithm computes the maximum size for allocated buffers in L1 memory. This is a different context w.r.t. a buffer sizing problem for data-flow graphs (such as [35]): at a given time each buffer contains the tile of a specific input/output/intermediate image, but the referred image will change due to the buffer reuse policy. We size buffers with a heuristic approach. We initially set the size of each buffer equal to its upper bound (maximum image size), then we alternatively halve width and height for all buffers until we have a total memory footprint that fits in L1 available memory and is greater than the related lower bound (maximum tile size, including neighborhood). To implement double buffering, we consider a double size for input buffers. If the buffers do not fit in memory, the algorithm backtracks to graph partitioning stage.

H. Run-time graph

The graph generation algorithm interprets all the decision made in previous steps to build a complete graph for the CLE run-time, using double buffering to achieve a good overlap between data transfer and execution stages. Figure 7 depicts the execution schema of an OpenVX application at run-time level, with a graph structure equivalent to the example of Listing 1. Note that in a target architecture without a cluster controller processor, the same tasks could be performed by a thread running on the host side.

The Cluster Controller (CC) initializes the execution environment (a), in particular the allocation of the L1 buffers, and then program the DMA engine to transfer the first two input tiles tiles from L3 memory to L1 buffers (1). When the first transfer is completed (2), the CC is notified, and then the computation of the kernels is triggered (3). The CC is notified one more time when the second transfer is completed (4), but it does not take any immediate action at this time (c), because the PEs are still executing the kernels for the first tile. When the input buffer is no more occupied by any intermediate results, the CC (d) is notified (5), and a new input data transfer is triggered (6). When the last kernel terminates its execution, the CC is notified again (7), and the DMA engine is programmed to transfer an output tile from L1 buffer to L3 memory (8); in the most general case, each kernel can produce an output image, and so this block could be triggered at the end of any kernel. If a DMA input transfer is completed, that is our case, a new tile computation is triggered (9); however, the next kernel execution is triggered when both events have occurred.

TABLE I
DETAILS ON OPENVX BENCHMARKS

Benchmark	Nodes	Images (in/out/virtual)
Random graph	10	1 / 1 / 10
Edge detector	4	1 / 1 / 4
Object detection	4	2 / 1 / 3
Super resolution	8	3 / 1 / 5
Retina preproc.	165	1 / 4 / 120

I. User-defined nodes

Current OpenVX proposal enables the programmer to specify new custom nodes, and our framework supports this feature. On the host side, programmmers have to specify two functions, that are input and output validator callbacks for the verification stage, and a data descriptor specifying image parameters, tiling behavior and kernel state. On the accelerator side, programmers must provide a standard OpenCL kernel that accesses image parameters directly in global memory space, without using any intermediate local buffer.

V. EXPERIMENTAL RESULTS

We have implemented the framework described in the previous section targeting the STHORM evaluation board (see Section III-A). The framework supports the data access patterns described in Section IV-B, enabling the execution of all the 40 kernels included in the OpenVX standard (first provisional version).

To assess the benefits of our approach on real applications, we have selected a significant set of benchmarks from the main fields of image processing domain:

- *Random graph* is a synthetic benchmark including 10 morphological nodes, it exposes a wider branching schema compared to real applications, with the specific aim to stress allocation and scheduling algorithms.
- *Edge detector* is an edge detector based on Sobel filter [36];
- *Object detection* is an abandoned/removed object detection algorithm based on NCC background subtraction, as described in [37];
- *Super resolution* includes the final recombination phase of a computation photography algorithm, which is used to increase the quality of an image using multiple overlapping pictures of a scene [38];
- *Retina preprocessing* implements the retina preprocessing filter, as described in [2].

Table I reports some relevant statistics related to the OpenVX implementation of these benchmarks.

The *Retina preprocessing* algorithm described in [2] is composed by 24 building blocks. In our implementation, some of these blocks (e.g., complementary, sum, subtraction) are directly mapped on OpenVX nodes, while the other ones have been implemented as the composition of multiple nodes. We have chosen this methodology for three main aspects: (i) it respects the typical approach promoted by OpenVX, first using standard nodes and then defining a new set of reusable user-defined nodes; (ii) the complex blocks may be implemented using multiple OpenVX nodes, defining a sub-graph which is instantiated multiple times; (iii) the user-define kernels are members of the first three classes (see Section IV-B), and the more complex patterns are handled by standard kernels already provided by the framework.

A. Comparison with sequential code

Figure 8 shows the speed-up of the OpenVX accelerated versions w.r.t. the OpenVX sequential version executed on the host processor (a single core of ARM Cortex A-9 on STHORM board). Table II reports the benchmark execution times and the amount of accessed L3 bytes for both versions.

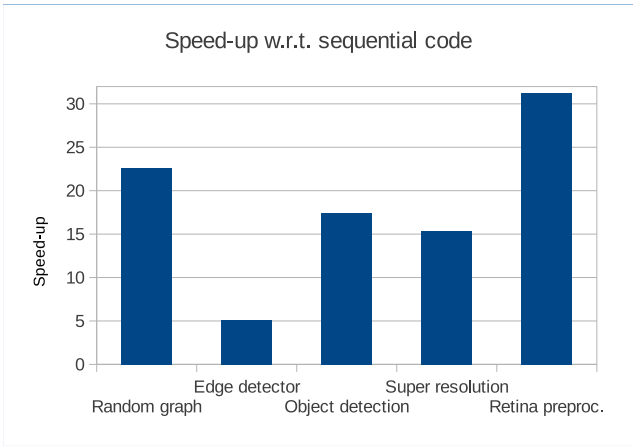

Fig. 8. Speed-up of OpenVX CLE (accelerator) w.r.t. sequential code (ARM)

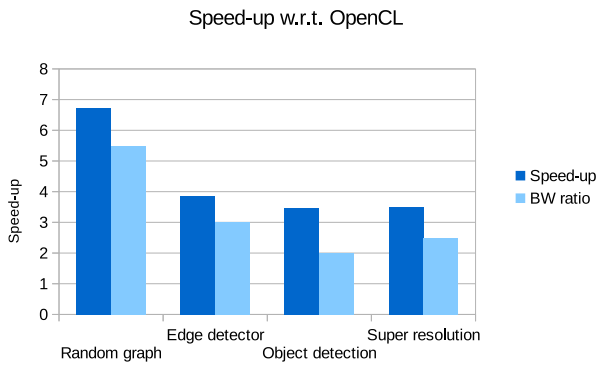

Fig. 9. Speed-up of OpenVX CLE w.r.t. standard OpenCL code

It also summarizes the speed-up (S) and the L3 bandwidth reduction (BWR) for the accelerated version. "OVX CLE" denotes the version executed using our framework. while "OVX seq" is the sequential version, compiled for target ARM, with O3 optimization level and NEON acceleration enabled (just for automatic vectorization, NEON intrinsics are not used). The reference image size is 640×480 pixels.

The speed-up for *Edge detector* is limited, as it is composed by kernels with a low computational workload, and so the benefits of using any accelerator are very limited. Nevertheless, the execution of *Edge detector* takes 11ms with our framework, against a value of 22ms resulting from KernelGenius [29]. The code produced by KernelGenius is equivalent to hand-tuned OpenCL code, while in Section V-B we assemble our benchmarks using a library of stand-alone OpenCL kernels. In this case our OpenVX-based solution shows better performance than hand-tuned code, and also promotes code reusability by maintaining stand-alone kernels.

Assuming an ideal benchmark that contains just ALU instructions, and considering the different clock frequencies set for host and accelerator, the maximum speed-up w.r.t. a single Cortex-A9 core would correspond to 10.8x. In practical cases, this value could be significantly conditioned by memory latency effects. Using our localized approach, we limit the accesses in L3 memory regions (latency $\approx$ 450 cycles) and maximize the localized accesses (latency = 1 cycle). In this context, the speed-up values of the benchmarks are justified considering the number of intermediate results (i.e. virtual images), which are always allocated to L3 memory in the sequential version. Table II reports the amount of total accesses to L3 memory, and it shows the L3 bandwidth reduction (BWR) using our approach.

TABLE II
COMPARISON BETWEEN SEQUENTIAL AND ACCELERATED OPENVX (TIME/L3 DATA)

Benchmark	OVX seq	OVX CLE	S	BWR
Random	427ms / 6.76MB	18ms / 1.23MB	**23.7**	**82%**
Edge det.	65ms / 3.69MB	11ms / 1.23MB	**5.9**	**67%**
Object det.	299ms / 3.69MB	17ms / 1.84MB	**17.5**	**50%**
Super res.	613ms / 6.14MB	40ms / 2.46MB	**15.33**	**60%**
Retina pre.	31284ms / 63.11MB	1001ms / 14.35MB	**31.25**	**77%**

B. Comparison with OpenCL

Figure 9 shows the speed-up of the OpenVX accelerated versions w.r.t. the same applications implemented on the standard OpenCL run-time. Each application is built using a library of image processing kernels, with the aim to mimic the component-like approach promoted by OpenVX. In this chart we have not included *Retina preprocessing* due to the high number of specific kernels required by this benchmark which would have lead to a very slow OpenCL execution.

Since each OpenCL kernel copies its outputs to L3 memory in order to pass data to the next one, the speed-up is closely related to the L3 bandwidth reduction (see Table II). To highlight this aspect, Figure 9 also reports a value labeled "BW ratio", that is the ratio between the data transferred by the accelerated version and the data transferred by the sequential version.

These measures do not include the initialization time for the programming framework. In particular, the algorithms which verify the graph are polynomial and the computational complexity is $O(n*e)$, where n and e are the number of nodes and edges in the CLE run-time graph.

C. Execution efficiency

Figure 10 depicts the accelerator efficiency related to benchmarks execution, computed as the percentage of total graph execution time. At *execution time*, the cores are actually executing kernel instructions, while the *wait time* is the time to complete DMA transfers that precede node execution in the CLE run-time graph.

Edge detector and *Retina preprocessing* are characterized by a significant level of inefficiency. In both cases the wait time is very high, and the execution on the accelerator is dominated by the memory latency. In *Edge detector* this is due to the low computational intensity of the algorithms. In *Retina preprocessing* it is related to the high number of statistical kernels that imply multiple splits in the graph, and consequently the traffic to/from L3 memory is increased. These effects are not limitations of our framework, since in both cases it gets the maximum overlap for data transfers and computation, but

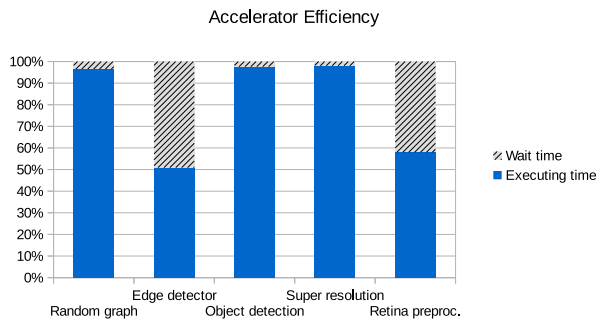

Fig. 10. Breakdown analysis of accelerator efficiency

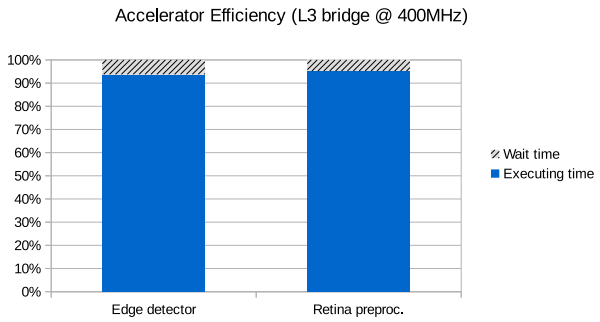

Fig. 11. Efficiency of Edge detector and Retina preprocessing with memory brdge at 400MHz (simulation)

they are due to the limits of the STHORM evaluation board. The FPGA bridge is clocked very conservatively at 40MHz; consequently, the main memory bandwidth available to the STHORM chip is limited to 250MB/s for the read channel and 125MB/s for the write channel, with an access latency of about 450 cycles. Compared with ARM processor, which uses all the DDR3 bandwidth (1.25GBps per channel) with a limited latency (50 cycles), STHORM chip is severely penalized. This is a worst case w.r.t. to a fully integrated SoC, where the accelerator will have a much greater share of L3 bandwidth. Nevertheless, we can simulate a more realistic scenario using the STHORM simulation tools. Figure 11 shows the efficiency of *Edge detector* and *Retina preprocessing* when the memory bridge is clocked at 400MHz. In both cases the wait time is drastically reduced.

VI. Conclusions and future works

In this paper we have proposed a set of techniques to improve the execution efficiency of image processing algorithms on many-core accelerators, and we have implemented a framework with a standard OpenVX front-end that implements these features on the STHORM platform. Experimental results show that our approach to localized execution provides huge benefits in terms of speed-up, considering both a sequential version and an accelerated OpenCL version, and also in terms of execution efficiency and bandwidth reduction.

Our future work will be focused on the study of new techniques to split and execute a vision graph over multiple accelerators, also supporting GPU execution.

Acknowledgment

Work supported by the EU-funded research projects PHIDIAS (g.a. 318013) and P-SOCRATES (g.a. 611016).

References

[1] Embedded Vision Alliance, http://www.embedded-vision.com/.
[2] S. Park, A. A. Maashri, K. M. Irick, A. Chandrashekhar, M. Cotter, N. Chandramoorthy, M. Debole, and V. Narayanan, "System-on-chip for biologically inspired vision applications," *Information Processing Society of Japan: Transactions on System LSI Design Methodology*, 2012.
[3] S. Greengard, "Computational photography comes into focus," *Commun. ACM*, 2014.
[4] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2012.
[5] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *Proceedings of the 2008 Symposium on Application Specific Processors*. IEEE, 2008.
[6] Qualcomm, "Computer Vision (FastCV)," https://developer.qualcomm.com/computer-vision-fastcv.
[7] Movidius, Ldt., "Myriad 1 Mobile Vision Processor," http://www.movidius.com/technology/myriad1/.
[8] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*. IEEE, 2012.
[9] Plurality Ltd, "The HyperCore Processor," http://www.plurality.com/hypercore.html.
[10] KALRAY Corporation, http://www.kalray.eu/.
[11] Adapteva, Inc., "Epiphany-IV 64-core 28nm Microprocessor," http://www.adapteva.com/products/silicon-devices/e64g401/.
[12] A. Vajda, *Programming many-core chips*. Springer, 2011.
[13] A. Heinecke, M. Klemm, and H. Bungartz, "From GPGPU to Many-Core: Nvidia Fermi and Intel Many Integrated Core Architecture," *Computing in Science & Engineering*, 2012.
[14] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Compiler Construction*. Springer, 2002.
[15] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 2011.
[16] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2013.
[17] Kronos Group, "The OpenVX API for hardware acceleration," http://www.khronos.org/openvx, 2013.
[18] OpenCV Library Homepage, http://www.opencv.com/.
[19] J. Coombs and R. Prabhu, "OpenCV on TIs DSP+ ARM® platforms: Mitigating the challenges of porting OpenCV to embedded platforms," *Texas Instruments*, 2011.
[20] Tegra Android Development Documentation Website, http://docs.nvidia.com/tegra/index.html.
[21] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, 2010.
[22] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012.
[23] P. Boudier and G. Sellers, "Memory System on Fusion APUs," *AMD fusion developer summit*, 2011.
[24] E. Ayguadé, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. Gonzàlez, F. Igual, D. Jiménez-González, J. Labarta *et al.*, "Extending openmp to survive the heterogeneous multi-core era," *International Journal of Parallel Programming*, 2010.
[25] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC First Experiences with Real-World Applications," in *Euro-Par 2012 Parallel Processing*. Springer, 2012.
[26] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally *et al.*, "Sequoia: programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006.
[27] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun, "Implementing domain-specific languages for heterogeneous parallel computing," *Ieee Micro*, 2011.
[28] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM review*, 2009.
[29] T. Lepley, P. Paulin, and E. Flamand, "A novel compilation approach for image processing graphs on a many-core platform with explicitly managed memory," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. IEEE, 2013.
[30] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han, "Comic: a coherent shared memory interface for cell be," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008.
[31] M. Gonzàlez, N. Vujic, X. Martorell, E. Ayguadé, A. E. Eichenberger, T. Chen, Z. Sura, T. Zhang, K. O'Brien, and K. O'Brien, "Hybrid access-specific software cache techniques for the cell be architecture," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008.
[32] A. Franceschelli, P. Burgio, G. Tagliavini, A. Marongiu, M. Ruggiero, M. Lombardi, A. Bonfietti, M. Milano, and L. Benini, "Mpopt-cell: A high-performance data-flow programming environment for the cell be processor," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*. ACM, 2011.
[33] Kronos Group, "The OpenCL 1.1 Specifications," http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, 2010.
[34] M. Sonka, V. Hlavac, R. Boyle *et al.*, *Image processing, analysis, and machine vision*. Thomson Toronto, 2008.
[35] M. Geilen, T. Basten, and S. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *Proceedings of the 42nd annual Design Automation Conference*. ACM, 2005.
[36] I. Pitas, *Digital image processing algorithms and applications*. Wiley, 2000.
[37] M. Magno, F. Tombari, D. Brunelli, L. Di Stefano, and L. Benini, "Multimodal abandoned/removed object detection for low power video surveillance systems," in *Advanced Video and Signal Based Surveillance, 2009. AVSS'09. Sixth IEEE International Conference on*. IEEE, 2009.
[38] F. Schubert, K. Schertler, and K. Mikolajczyk, "A hands-on approach to high-dynamic-range and superresolution fusion," in *Applications of Computer Vision (WACV), 2009 Workshop on*. IEEE, 2009.