A framework for optimizing OpenVX applications performance on embedded manycore accelerators

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

Germain Haugou
ETH Zurich
haugoug@iis.ee.ethz.ch

Andrea Marongiu, Luca Benini
University of Bologna
ETH Zurich
{a.marongiu,l.benini}@unibo.it

ABSTRACT
Nowadays Computer Vision application are ubiquitous, and their presence on embedded devices is more and more widespread. Heterogeneous embedded systems featuring a clustered manycore accelerator are a very promising target to execute embedded vision algorithms, but the code optimization for these platforms is a challenging task. Moreover, designers really need support tools that are both fast and accurate. In this work we introduce ADRENALINE, an environment for development and optimization of OpenVX applications targeting manycore accelerators. ADRENALINE consists of a custom OpenVX run-time and a virtual platform, and overall it is intended to provide support to enhance performance of embedded vision applications.

Categories and Subject Descriptors
B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; H.4 [**Image Processing and Computer Vision**]: Miscellaneous; I.6 [**Simulation and Modeling**]: Miscellaneous

General Terms
Design, Performance

Keywords
embedded vision, OpenVX, virtual platform, manycore

1. INTRODUCTION
Embedded Computer Vision (CV) technologies are gaining momentum, guided by killer applications such as gesture tracking, smart video surveillance, advanced driver assistance systems and augmented reality. CV is computation-demanding, but embedded systems have tight energy budget. Architectural heterogeneity is an effective design paradigm to build energy-efficient embedded vision systems. A common platform relies on system-on-chip (SoC) integration of a host processor and one or more programmable manycore accelerators (MCA) [2] [8] [4] [3]. MCAs provide tens to hundreds of small processing units, connected to a shared on-chip memory via a low-latency, high-throughput interconnection. For improved energy efficiency the memory hierarchy relies on explicitly managed scratchpads rather than coherent caches. MCA-based SoCs have the potential to deliver the peak performance/Watt required by CV applications, but computational power comes at the price of increased programming complexity. The role of programming models aimed at simplifying this task becomes thus paramount.

Representative examples of programming paradigms in this context are OpenCV [7] and OpenCL [5]. OpenCV is the de-facto standard for CV application development in general-purpose computers, but its mainstream version is not suitable for embedded MCAs, due to complex software dependencies and large code and data footprint. OpenCL is the de-facto standard for accelerator programming, but it exposes a very low-level API which requires skilled programmers and in-depth knowledge of the target hardware. To increase the productivity of application designers, it is important that the programming model exposes high-level constructs for the exploitation of parallel and heterogeneous resources. This is particularly true in the CV domain, where the expertise of application designers is typically on the algorithms.

OpenVX [6] is a cross-platform standard for imaging and vision, aimed at raising significantly the level of abstraction at which CV applications should be coded. Based on a C API, it is easy to use and transparent to specific architectural details. This approach enables the portability of vision applications across different heterogeneous platforms, delegating the performance tuning to hardware vendors, who provide an efficient run-time environment (RTE) with architecture-specific optimizations.

In this paper we describe our preliminary work with **ADRENALINE**[1], a framework for fast prototyping and optimization of OpenVX applications for MCA-based heterogeneous SoCs. ADRENALINE consists of an optimized OpenVX run-time for MCAs and a highly configurable virtual platform. The run-time system includes several optimizations for efficient memory hierarchy exploitation, but it can be easily extended to consider other optimization opportunities. In-line with state-of-the-art simulation techniques [9], the virtual platform offers fast simulation speed by delivering instruction-level accuracy for critical hardware components only (cores, memories, interconnect), while less relevant blocks are modeled at a higher level of abstraction. The virtual platform can be extended to model additional architectural blocks in a simple manner.

ADRENALINE can be useful to several end users: (i) *researchers and/or SDK vendors* can explore various platform-specific optimizations, scheduling policies and algorithms for the implementation of the OpenVX support layer; (ii) *application developers* can explore different partitioning solutions

[1]Website: http://www-micrel.deis.unibo.it/adrenaline/

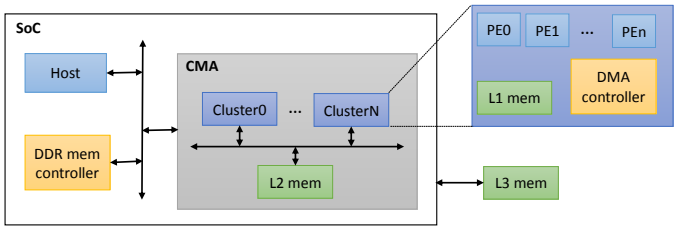

Figure 1: Reference architectural template.

(host vs accelerator, parallelization) for different applications; (iii) *platform engineers* can quickly evaluate different architectural configurations for a target CV application.

2. OPENVX PROGRAMMING MODEL

OpenVX [6] is a cross-platform C-based Application Programming Interface (API) ratified by the Khronos Group as an open standard. OpenVX provides a graph-oriented execution model, where nodes are instances of vision algorithms and arcs are implicitly defined by data dependencies.

An OpenVX program typically starts with the declaration of *data objects*, that are required to specify node parameters. Data objects may be declared as *virtual*, and in this case they are not guaranteed to reside in main memory on a permanent basis Virtual data have no specific dimension or format but just define dependencies between adjacent kernel nodes. Moreover, virtual data are not associated with any memory area accessible for read/write operations using the API functions.

After declaring data objects, a graph can be instantiated by calling node creation functions specific of the algorithms (*vision kernels*). Data objects are used as parameters, and nodes are linked together via data dependencies. The OpenVX standard defines a library of *predefined vision kernels*, but also supports the notion of *user defined kernels* to provide new functionalities. The standard defines 41 predefined kernels, which are fully supported by ADRENALINE.

As a main feature, OpenVX graphs must pass through a *verification* stage before their execution, with the aim to guarantee some mandatory properties:

- Parameter types must be consistent.
- No cycles are allowed in the graph.
- Only a single writer node per data object is allowed.
- Writes have higher priorities than reads.

Graphs can be executed as many times as needed after verification, even if changes require a new verification.

The OpenVX standard has been designed to support efficient execution on a wide range of architectures, maintaining a consistent vision acceleration API for application portability. In this scenario, vendors should provide implementations for different hardware architectures, such as CPUs, GPUs, DSPs, FPGAs, and dedicated ASICs. The high-level specification of OpenVX model hides all hardware details, but a vendor could implement a wide range of platform-specific optimization techniques.

3. ARCHITECTURAL TEMPLATE

The architectural template targeted by ADRENALINE (Figure 1) consists of a general-purpose host processor coupled with a clustered manycore accelerator (CMA), both integrated in an embedded system on chip (SoC) platform. The multi-cluster design is a common solution applied to overcome scalability limitations in modern manycore accelerators, such as STM STHORM [2], Plurality HAL [8], KALRAY MMPA [4] and PULP [3].

The processing elements (PEs) inside a cluster are fully independent RISC cores, supporting both SIMD and MIMD parallelism. Each PE is equipped with a private instruction cache, while a private data cache is not present to avoid coherency overhead and increase energy efficiency. All PEs share a L1 multi-banked tightly coupled data memory (TCDM) acting as a scratchpad. The TCDM has a number of ports equal to the number of memory banks to provide concurrent access to different memory locations. Communication between the cores and the TCDM is based on a single-cycle logarithmic interconnect, implementing a word-level interleaving scheme to reduce the access contention to TCDM banks. The architectural template also includes a L2 scratchpad memory at SoC level and an external L3 DDR accessible by means of a memory controller. Both host cores and PEs can access the whole memory space, that is modeled as a partitioned global address space (PGAS). A DMA engine enables communication with other clusters, the L2 memory and external peripherals.

The current version of ADRENALINE models a SoC with a single accelerator cluster, while future releases will also provide a multi-cluster environment.

4. ADRENALINE

ADRENALINE comes with a *virtual platform*, which models the target architecture template, and an *OpenVX run-time*, that is a framework optimized for manycore accelerators.

4.1 Virtual platform

The virtual platform is written in Python and C++. Python is used for the architecture configuration and high-level execution management, while C++ is used to provide an efficient implementation of internal models. A library of basic components is available, but custom blocks can also be implemented and assembled. Some additional details about the key pre-defined components follow:

- **OpenRISC core**. An Instruction Set Simulator (ISS) for the OpenRISC ISA [1] has been extended with timing models to emulate pipeline stalls. Different ISS can be plugged to the same interface;
- **Memories**. The memory blocks use a simple, constant-latency timing model;
- **L1 interconnect**. One transaction per memory bank is serviced at each cycle;
- **DMA**. The DMA sends a single synchronous request to the interconnect for each line to be transferred. Multiple lines can be buffered;
- **Shared instruction cache**. Composed of an dedicated interconnect and memory banks.

The current release models a single-core host (also based on the OpenRISC ISS) plus a single-cluster accelerator. ADRENALINE provides many tunable parameters, mainly: number of cores, HW FPU enabled/disabled, available memory (L1, L2 and L3 levels), DMA bandwidth/latency. ADRENALINE can be extended by defining new modules adhering to the described templates.

4.2 OpenVX run-time

Our OpenVX run-time for CMA architectures is based on a RTE layer that provides (i) explicit memory management and (ii) support for execution of graph-based data structures. As a main goal, The run-time enforces *localized execution*, which implies that when a kernel is executed on a cluster, read/write operations are always performed on local buffers in the L1 scratchpad memory. The major advantage of this approach is that processor stalls due to reads/writes on L1 are very limited, as the access latency is low. In real platforms the L1 is typically too small to contain a full image, moreover multiple kernels requires more L1 buffers to store intermediate results. As a solution to this problem, images are partitioned into smaller blocks, called *tiles*. Multiple tiles can use different L1 buffers, and the use of *double buffering* enables to overlap between input/output data transfers and computation.

OpenVX standard does not specify which patterns should be supported and how they drive the run-time optimizations. Based on literature [10], we consider five remarkable classes of vision operators:

1. *Point operators* (e.g. color conversion, threshold) compute the value of each output point from the corresponding input point.
2. *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.
3. *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.
4. *Global operators* (e.g. DFT) compute the value of a point in the output image using the whole input image.
5. *Geometric operators* (e.g. affine transforms) compute the value of a point in the output image using a non-rectangular input area.
6. *Statistical operators* (e.g. mean, histogram) compute statistical functions of image points.

To describe tiling patterns inside the run-time, we associate a *tiling descriptor* to each node parameter. Each descriptor defines a *computing area* and a *neighboring area*. The computing area 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. The neighboring area 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. Our framework supports tiling on both input and output images for point and local neighbor operators. In case of statistical operators, Tiling can be activated just on input images, and a persistent buffer is used to support reduction patterns. Global operators, and also most geometrical operators, do not support input tiling due to the irregular shape of the neighboring area. The managing of tiling in recursive neighbor operators is equivalent to local neighbor operators, but we also need to save state data between tiles. For each kernel we provide a *state buffer*, which save a set of data proportional to a tile width and height.

To support our run-time optimizations, additional steps are required after the basic graph verification:

- *Node scheduling* – a schedule is determined through a breadth-first visit, starting from the kernels connected to graph input data (head nodes). At each step a single kernel is selected for execution. For kernels which are executed on the accelerator, the code parallelism is exploited at node level.
- *Buffer allocation* – The initial number of L1 buffers is equals to the number of input images to the graph. For each kernel in the schedule list, the corresponding output buffers are allocated. A reference counting mechanisms is enabled, so a buffer can be reused if there is no further references to it.
- *Tile size propagation* – Overlap between adjacent tiles enforces data locality on buffers at the cost of transferring and/or computing the tile borders more than once. The final tile size is computed into two passes: a (i) forward pass, simulating a graph execution and simultaneously collecting the tiling constraints for each kernel; a (ii) backward pass, starting from the last simulated node and enlarging the tile size in accordance with computing and neighboring areas of the involved kernels.
- *Buffer sizing* – The current implementation uses an heuristic approach, setting the size of each buffer equal to its upper bound (full image size) and alternatively halving width and height for all buffers until the total memory footprint (i) fits the L1 available memory and (ii) is greater than the related lower bound (maximum tile size, including neighborhood).
- *Graph partitioning* – When the graph cannot be executed allocating all the intermediate tiles in L1 buffer, the graph is automatically partitioned into multiple sub-graphs.

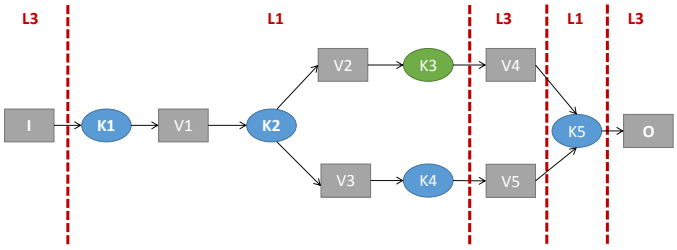

Figure 2: Application graph partitioning

At the end of graph verification stage, the framework derives an ordered partition set of the original OpenVX graph, each element consisting of a single host kernel or an OpenCL graph. At the execution stage single kernels are executed on the host, while OpenCL graphs are executed on the accelerator.

On the host side, we provide a set of kernels implemented on C using the OpenVX API to access data objects that has been introduced in Section 2, based on the reference implementation provided by Khronos. On the accelerator side, we provide a set of OpenCL kernels which access input/output parameters directly addressing the global memory space, without explicit use of DMA primitives and intermediate local buffers. All the boilerplate code related to tiling and localized execution is managed by the run-time on the basis of kernel descriptors and graph verification steps.

Using our OpenVX run-time, the orchestration of multiple kernel nodes and accelerator sub-graphs is totally transparent to the programmer. For instance, in the application graph depicted in Figure 2 we suppose that K3 is a statistical kernel (e.g., a histogram). Tiling cannot be used on

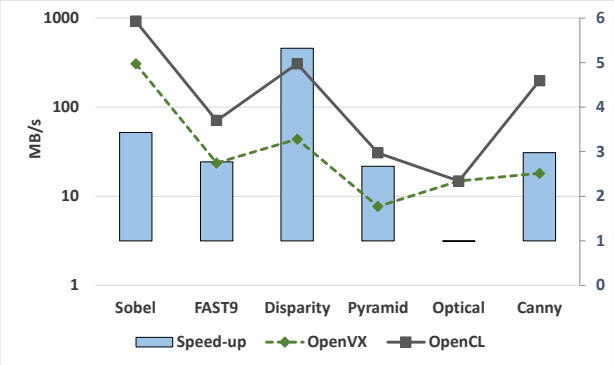

Figure 3: Speed-up of OpenVX compared to OpenCL on ADRENALINE

its output image V4, because each input tile contributes to sparse data in the result set (the histogram bins). Consequently, a memory boundary is added to its output, and this includes all the images read by K5. In this example K5 is a kernel of classes 1, 2 or 3, so a memory boundary is inserted to switch back its input images to L1 domain. To guarantee consistency in the host environment, I and O must reside on L3 domain. The same result cannot be obtained by fusing kernels and optimally tiling the generated loop, as tiling requirements of different kernels are not homogeneous. Overall, the use of OpenVX graphs enables a global level of optimization which is not possible under a single-function paradigm

5. EXPERIMENTAL RESULTS

We provide early evaluation using a set of vision benchmarks:

- *Sobel* is a an edge detector based on Sobel filter;
- *FAST9* implements the FAST9 corner detection algorithm;
- *Disparity* computes the stereo-matching disparity between two images;
- *Pyramid* creates a set of scaled and blurred images (a pyramid);
- *Optical* implements the Lucas-Kanade algorithm to measure optical flow field;
- *Canny* implements the Canny edge detector algorithm.

We configure the virtual platform to model a real embedded MCA (STMicroelectronics STHORM [2]): 16 cores with hardware floating point support, 256KB L1 memory, 2MB L2 memory, 500MB L3 memory, 250MB/s for DMA bandwidth, 450 cycles for DMA latency.

Figure 3 shows the speed-up of OpenVX compared to an implementation of the same benchmarks with OpenCL 1.1. Each OpenCL application is manually optimized to support asynchronous data transfers and double buffering. Tests are performed with an input image size of 640×480 pixels.

Figure 3 also reports the bandwidth requirements of the benchmarks for both OpenVX and OpenCL. These values are computed as the ratio between the total transferred bytes and the required computation time. The OpenVX version requires a lower bandwidth, with the exception of *Optical* that is implemented using a single kernel invoked multiple times. The bandwidth required by an the OpenCL application exceeds the available one (represented by the dashed line) in *Sobel* and *Disparity*, further limiting the speed-up of the OpenCL solution. Since each OpenCL kernel copies its outputs to L3 memory to pass data to the next one, the speed-up is closely related to the L3 bandwidth reduction.

Canny / FAST9	Time (Mcycles)
HOST / HOST	385
ACC / ACC	233
ACC / HOST	76
ACC / ACC	912

Table 1: Mapping of Canny and FAST9

As an example of application partitioning exploration, we have considered a single application graph containing both a Canny edge detector and a FAST9 corner detector[2]. Table 1 shows the timings resulting from the different mappings of FAST and Canny. For the considered implementations, the best solution is the one executing Canny on the accelerator and FAST9 on the host side.

6. CONCLUSIONS

We present ADRENALINE, a framework for fast prototyping and optimization of OpenVX applications, including an OpenVX run-time and a virtual platform. Current and future work is focused on i) supporting multi-cluster platform and run-time system; ii) FPGA-based platform models.

7. ACKNOWLEDGEMENTS

This work has been supported by the EU-funded research projects PHIDIAS (g.a. 318013) and P-SOCRATES (g.a. 611016), and by Eureka EuroStars project VAMPA (E! 7678).

8. REFERENCES

[1] OPENCORES Project: OpenRISC 1000 processor. `http://www.opencores.org/projects/or1k/`.
[2] 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.
[3] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini. Energy-efficient vision on the PULP platform for ultra-low power parallel computing. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, 2014.
[4] KALRAY Corporation. `http://www.kalray.eu/`.
[5] Kronos Group. The OpenCL 1.1 Specifications. `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`, 2010.
[6] Kronos Group. The OpenVX API for hardware acceleration. `http://www.khronos.org/openvx`, 2014.
[7] OpenCV Library Homepage. `http://www.opencv.com/`.
[8] Plurality Ltd. The HyperCore Processor. `http://www.plurality.com/hypercore.html`.
[9] S. Raghav, M. Ruggiero, A. Marongiu, C. Pinto, D. Atienza, and L. Benini. Gpu acceleration for simulating massively parallel many-core platforms. *IEEE Transactions on Parallel and Distributed Systems*, 2015.
[10] M. Sonka, V. Hlavac, R. Boyle, et al. *Image processing, analysis, and machine vision*. Thomson Toronto, 2008.

[2]Note that the OpenVX standard library provides its own FAST9 algorithm, which differs from the one used for the previous experiments